The Ideal Documentation Suite for Software Developers

Potential deliverables for any developer-to-developer documentation suite

–Expert Support Staff

"What does Expert Support recommend for the contents of an ideal documentation suite for software targeted at software developers?"

Of course, the answer is complicated. Specific situations vary from company to company and for various software technologies.

This question led to several conversations with many of our senior writers. As you might expect, patterns and conventions began to emerge. The rest of this article summarizes the high-level consensus, and can serve as a checklist for any documentation suite aimed at a community of software developers.

As you'll see, this checklist boils down to a list of deliverable documents. Each one addresses a different set of assumptions about the reader, what they're trying to accomplish, and how they'll use each of these documents. The "reader experience" varies widely across these deliverables, but each one addresses a specific part of the developer experience.

This list of potential deliverables applies to any developer-to-developer documentation suite. This includes APIs, SDKs, platforms, frameworks, and other types of software development platforms. For simplicity, we'll refer to this collection of technologies as APIs, because they all have similar documentation needs.

Also, note that not every API will require everything on the list below. But every API team should consider everything on this list when determining their documentation priorities. Each deliverable addresses a different facet of the developer experience, and each one might be crucial to your API users' success.

The Ideal Documentation Suite for Software Developers

Technical Overview

Start with the **technical overview**. Explain what the technology is, what it does, who should use it, and most importantly, the unique benefits it can deliver. The benefits should be differentiating and compelling, as they're the only reasons why anyone should pay attention to the technology.

A good technical overview can provide a high-level description of how the underlying system works, how the software compares to alternatives, and why the unique benefits of this technology make it superior.

The technical overview should appeal to the developers you want to use your API (we'll call them the ***API users***), and can even discourage those who shouldn't be using it. No API can do everything, and being frank about the limitations of your technology is refreshing to API users, and sets expectations about what they can and cannot expect from your technology. Also, identifying the limits of your API will reduce the amount of time you'll waste supporting API users with unrealistic expectations.

Further, writing the technical overview improves your understanding of the big picture, creates context, and sets the stage for all that follows.

Reference

The **reference** is the key deliverable in the set. This document serves as the foundation for the rest of the suite, and may be the only material that more experienced API users ever consult.

Think of the reference as the online dictionary for the technology. Imagine that an API user is programming away, and can't quite remember how a particular function, class, method, or other element works. They'll want to look it up quickly, easily find the specifics they need, and get right back to work.

If you have an API, you need a reference. The effort you put into creating a reference that's complete, accurate, consistent, concise, and easy to use will pay dividends for as long as API users use your technology.

Because the reference is crucial to the success of the software, we're working on a white paper about how to create a good one. Stay tuned, as we'll have a lot more to share shortly.

This article is part of a series of articles on technical writing that Expert Support hosts on its website, expertsupport.com.
Expert Support is located in Los Altos, California and supplies contract technical writers to the computer and software industry.

Quick aside

Back in the 80's, we worked with Alan Fisher at Teknowledge, a high-flying AI start up of that era. Alan coined what I call "Fisher's Law of Software." Similar to Moore's Law, which suggests that the number of transistors in a dense integrated circuit doubles about every two years, Fisher's Law of Software says, "Every five years, there's a new name for the subroutine."

Installation/Setup/Configuration Guide

While most API provider teams know that they need a reference, this next deliverable is often overlooked. The **installation guide** tells a new API user how to set up their development workstation with the necessary software environment. Setup details can include how and where to get credentials for the VPN, access to wiki pages and bug tracking systems, GitHub repos, Slack channels, and information about other key developer tools.

This guide should include everything needed for setup and configuration, things that you think are obvious, but often are a complete mystery to newcomers. You may even have different versions of this, one for internal developers, one for external developers, one for partners, and so on.

We're consistently surprised at how often API providers don't have basic installation and setup documentation, even for their new employees. Instead, they rely on expensive hand holding by team members, or trial-and-error efforts by the new API user. Creating instructions is more efficient for everyone, and makes for a much better ramp-up experience for a new programmer joining the team.

Getting Started/Hello World

This next deliverable can jumpstart the real work. The **getting started guide** provides an introduction to using the API with some simple practical examples. The getting started section is often overlooked in an API documentation suite.

A getting started guide usually includes a *Hello World* recipe (more on recipes below). If you aren't familiar with *Hello World*, it's the shortest functional program that you can write using an API. The program is a complete coding example that a developer can compile, execute, run, and see simple results. Originally, these programs ran just enough code to produce the message "Hello World" on a screen or printer.

Completing this how-to recipe delivers a quick emotional win to the new API user, and serves as a simple test to validate that the environment is set up properly. If constructed using good coding style, the *Hello World* recipe can be used by API users as a foundation for their first programs. Often, these first programs are written while completing tutorials, which is the next deliverable on our list.

Tutorial (Training Materials)

After the API user has set up their work environment, a **tutorial** can help them understand the basics. Tutorials are particularly helpful if your technology is complex.

Tutorials typically define key terms, describe how to use the technology, explain how to organize the code, present programming conventions, and so on. Effective tutorials can efficiently give new API users a solid, basic understanding of your entire system. Lessons in a self-paced tutorial should be clearly labeled with the skills they impart, provide sign posts for how long each lesson takes, and explain how each lesson fits into the overall instruction set.

Depending on the breadth and depth of the technology, tutorials serve as the starting point for training materials. As adoption grows and the technology matures, market demand for quicker and easier onramps can lead to a variety of training tools, classes, and programs. Tutorials help jumpstart that process.

Recipes and Cookbooks

If your API is straightforward, you might be able to skip the tutorial and dive right into recipes. **Recipes** are simple, ordered, step-by-step instructions for specific tasks with the software technology.

Ask yourself the following question: What are the top five tasks every API user should do to quickly get the highest value impact from our technology? Write those down. Now, explain how to accomplish those tasks with a recipe for each.

Quick wins that deliver real value early in the game are hugely important. They provide proof that deciding to use your technology was a good decision. Next, they give the new API user confidence. By realizing a quick win, and being able to be recognized for producing that quick win, API users begin to establish positive momentum, and look forward to using more of your system.

A collection of such recipes forms a **cookbook**. The more recipes you have, the better; however, the most valuable or popular ones should always be listed first.

With food, recipes give cooks the measurements and techniques to prepare meals like chefs. The same is true for APIs and related software development platforms. Simple, step-by-step instructions are powerful in that they provide the information needed for API users to become adept with your technology. This can accelerate customer success, deliver real value sooner, and build momentum for your technology within the organization adopting it.

Software Design Guide

Of course, a couple of cookbooks full of great recipes won't transform a cook into a great chef. Similarly with software, some developers stick to the basics (cooks), while others aspire to master the platform and even invent new ways to realize the value it provides (chefs).

While a **design guide** isn't needed in every instance, if your software technology is complex and has a broad set of potential applications, it's a must. Enabling developers to achieve true mastery of the API requires another level of information. In the design guide, the system architects can teach API users about the elegant features of their design, and how to apply that elegance in the context of software development. This document identifies, explains, and justifies recommendations for best practices.

Instead of step-by-step instructions for accomplishing specific tasks (recipes), a design guide explains the big conceptual ideas and how they relate to one another. It also explains the design tradeoffs that were made when constructing the system architecture, the reasoning behind those design choices, and why the technology was created, organized, and constructed the way it was.

These higher-level explanations help API users get inside the heads of the platform architects and think about solving problems in the same way as those architects. This helps API users master the design paradigm that the architects envisioned, and use that understanding to realize even more value from that paradigm, becoming chefs of your API.

Think of it this way:

> Mastering a design paradigm is to cut-and-paste coding
>
> as
>
> Being a master chef is to cooking with recipes

While you can appreciate and value the work of both cooks and chefs, the designers of a great API hope to inspire users to make the most of their work and use their technology for innovation. Cultivating such an ecosystem expands the value that a platform creates, and can help API providers realize the full potential of their technology.

Build, Test, Integration, and Deployment Guide

Some API technologies require additional documentation that explains the build and test cycle, as well as particulars about integrating the software and deploying it on specific platforms.

A software development kit (SDK) often works on multiple platforms and integrates with other systems, including embedded systems. The tools in an SDK assist the programmer with tasks that programmers have to do that don't include coding. These tools can include compilers, debuggers, libraries, and test harnesses, all specific to a hardware platform and operating system combination.

The **build, test, integration, and deployment guide** explains how to use these tools for the platforms that the SDK was designed to support. It might have a section on troubleshooting the build process. It also might discuss specifics on how to package, distribute, and install the resulting software. Sometimes this document is split up into separate documents, one for each platform.

Release Notes

After the first release of any API, every additional update should include comprehensive release notes. When done well, **release notes** accelerate the adoption of the latest technology by the user community, reduce the time and effort required to adopt the upgrade, and eliminate confusion about changes in the latest version.

While good release notes are concise, they usually include a brief description of the goals of the release (adding features, fixing bugs, improving performance, whatever), what was changed in the API, step-by-step instructions for adopting the new version, and advice for avoiding known incompatibilities or other problems.

It's a good idea to circulate a draft of the upcoming release notes at the beginning of the development process instead of waiting until the end to prepare them. Sharing what you hope to say to users about your API update at the onset of the sprint helps align the efforts of the team, and often surfaces issues or misunderstandings early in the process when they're still manageable.

FAQs and Knowledge Bases

When an API is deployed and adopted, the user community will inevitably have questions about its use. When those questions are asked and answered, the API provider will want some way to capture and share that dialog with other users who might have the same questions.

The most popular questions and answers can be compiled into a **frequently asked questions (FAQ) document**. The FAQ might be a static document with your selection of the questions that your users might ask. Better yet, you might use **a knowledge base** app, similar to StackOverflow, to capture, curate, and sort the most common questions. These systems automatically move popular questions and answers to prominent positions so visitors can easily discover them.

Regardless of the tools used to construct it, an effective knowledge base for your API should cater to the informational needs of your audience. For example, the knowledge base should adopt the same organizational model as your documentation (again, to reduce cognitive load), link to the documentation as appropriate, and use the same terminology in the same way.

The Ideal Documentation Suite for Software Targeted at Software Developers

In closing, let's review the checklist:

- *Technical Overview*
- *Reference*
- *Installation/Setup/Configuration Guide*
- *Getting Started/Hello World*
- *Tutorial (Training Materials)*
- *Recipes and Cookbooks*
- *Software Design Guide*
- *Build, Test, Integration, and Deployment Guide*
- *Release notes*
- *FAQs and knowledge bases*

This list of deliverables covers all the bases, and pushes the API user along a learning path that leads to deeper understanding over time. We first wrote about our experience with this approach here. One of our writers pointed out this pragmatic checklist echoes ideas found in the DITA Concept, Task, Reference model, as well as Bloom's taxonomy of learning, and while our approach is drawn from experience writing about software, it's not surprising our techniques follow similar patterns found in other sources.

When produced well, experience suggests this documentation suite helps flatten the learning curve and helps developers achieve value faster. Of course, all of this information has to be produced and deployed via documentation technology in a way that makes it easy to produce and maintain, as well as easy for API users to find and use. You need to make dozens of tools choices to pull this off, but that's another article.

We're very interested to know what you think. Is anything missing from this list? Is something on this list that shouldn't be?

And of course, if you need help with this work, contact us here. If you are a person who loves this kind of work and can help us with these sorts of projects, please do let us know here.

As always, we look forward to hearing from you.

(WriteTheDocs Lightning Talk Slides here.)

More information can be found on our website:

https://expertsupport.com

This article is the result of a collaborative effort among Expert Support Staff, with significant contributions from these Senior Technical Writers: Judy Bogart, Denny Brown, Jan E. Clayton, Ellen Levy Finch, Paul Gustafson, and David Welsch.