Documentation system

Daniele Procida

Apr 30, 2021
## Contents

1 About the system
   1.1 Introduction ......................................................... 4
   1.2 Tutorials .......................................................... 5
   1.3 How-to guides ....................................................... 8
   1.4 Reference guides .................................................. 11
   1.5 Explanation ........................................................ 14
   1.6 About the structure ............................................... 16
   1.7 Who is using the system? .......................... 19

2 Video presentation ........................................ 21
The Grand Unified Theory of Documentation

- David Laing

There is a secret that needs to be understood in order to write good software documentation: there isn’t one thing called documentation, there are four.

They are: tutorials, how-to guides, technical reference and explanation. They represent four different purposes or functions, and require four different approaches to their creation. Understanding the implications of this will help improve most documentation - often immensely.
The documentation system outlined here is a simple, comprehensive and nearly universally-applicable scheme. It is proven in practice across a wide variety of fields and applications.

There are some very simple principles that govern documentation that are very rarely if ever spelled out. They seem to be a secret, though they shouldn’t be.

If you can put these principles into practice, it will make your documentation better and your project, product or team more successful - that’s a promise.

The system is widely adopted for large and small, open and proprietary documentation projects.
1.1 Introduction

1.1.1 The problem and the solution

The problem it solves

It doesn’t matter how good your product is, because if its documentation is not good enough, people will not use it. Even if they have to use it because they have no choice, without good documentation, they won’t use it effectively or the way you’d like them to.

Nearly everyone understands this. Nearly everyone knows that they need good documentation, and most people try to create good documentation. And most people fail.

Usually, it’s not because they don’t try hard enough. Usually, it’s because they are not doing it the right way.

This system is a way to make your documentation better, not by working harder at it, but by doing it the right way. The right way is the easier way - easier to write, and easier to maintain.

The ‘secret’

It’s not actually a secret and it certainly shouldn’t be: documentation needs to include and be structured around its four different functions: tutorials, how-to guides, technical reference and explanation. Each of them requires a distinct mode of writing. People working with software need these four different kinds of documentation at different times, in different circumstances - so software usually needs them all, and they should all be integrated into your documentation.

And documentation needs to be explicitly structured around them, and they all must be kept separate and distinct from each other.

<table>
<thead>
<tr>
<th></th>
<th>Tutorials</th>
<th>How-to guides</th>
<th>Reference</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>oriented to</td>
<td>learning</td>
<td>a goal</td>
<td>information</td>
<td>understanding</td>
</tr>
<tr>
<td>must</td>
<td>allow the newcomer</td>
<td>show how to solve a</td>
<td>describe the machinery</td>
<td>explain</td>
</tr>
<tr>
<td></td>
<td>to get started</td>
<td>specific problem</td>
<td></td>
<td></td>
</tr>
<tr>
<td>its form</td>
<td>a lesson</td>
<td>a series of steps</td>
<td>dry description</td>
<td>discursive explanation</td>
</tr>
<tr>
<td>analogy</td>
<td>teaching a small child</td>
<td>a recipe in a cookery book</td>
<td>a reference encyclopaedia article</td>
<td>an article on culinary social history</td>
</tr>
</tbody>
</table>

This division makes it obvious to both author and reader what material, and what kind of material, goes where. It tells the author how to write, and what to write, and where to write it. It saves the author from wasting a great deal of time trying to wrestle the information they want to impart into a shape that makes sense, because each of these kinds of documentation has only one job.

In fact, it’s extremely hard to maintain good documentation that doesn’t implicitly or explicitly recognise the quadrants of this scheme. The demands of each kind are different from those of the others, so any attempt at documentation that fails to maintain this structure suffers, as it’s pulled in different directions at once.

Once you understand the structure, it becomes a very useful tool for analysing existing documentation, and understanding what needs to be done to improve it.

In the following sections, each of these four parts is dealt with in detail.
1.1.2 Making documentation work

For authors

One of the biggest headaches that documentation maintainers have to deal with is not having a clear picture of what they should be doing. They write and rewrite, but find it hard to make it fit together in satisfactory ways.

This structure resolves those questions by making clear distinctions and separations. They make documentation that is easier to write and maintain, that’s easier to use and to find one’s way around in.

The documentation doesn’t write itself - but it’s now possible to write it without also having to wrestle with poor fit, or unclear scope or doubt about what should be included or what style to adopt. It becomes much clearer what to write, how to write it, and where to put it.

For readers

It serves users better, because for all the different phases in the cycle of their interaction with the software they will find the right kind of documentation, that serves the needs of that moment.

Writing documentation that explicitly and distinctly addresses each of the four quadrants helps the software attract and keep more users, who will use it more effectively - and that is one of the things the creators of software want most of all.

1.2 Tutorials

Tutorials are lessons that take the reader by the hand through a series of steps to complete a project of some kind. They are what your project needs in order to show a beginner that they can achieve something with it.

They are wholly learning-oriented, and specifically, they are oriented towards learning how rather than learning that. You are the teacher, and you are responsible for what the student will do. Under your instruction, the student will execute a series of actions to achieve some end.

The end and the actions are up to you, but deciding what they should be can be hard work. The end has to be meaningful, but also achievable for a complete beginner.

The important thing is that having done the tutorial, the learner is in a position to make sense of the rest of the documentation, and the software itself.

Most software projects have really bad - or non-existent - tutorials. Tutorials are what will turn your learners into users. A bad or missing tutorial will prevent your project from acquiring new users.

Of the sections describing the four kinds of documentation, this is by far the longest - that’s because tutorials are the most misunderstood and most difficult to do well. The best way of teaching is to have a teacher present, interacting with the student. That’s rarely possible, and our written tutorials will be at best a far-from-perfect substitute. That’s all the more reason to pay special attention to them.

Tutorials need to be useful for the beginner, easy to follow, meaningful and extremely robust, and kept up-to-date. You might well find that writing and maintaining your tutorials can occupy as much time and energy as the the other three parts put together.
1.2.1 Analogy from cooking

Consider an analogy of teaching a child to cook.

*What* you teach the child to cook isn’t really important. What’s important is that the child finds it enjoyable, and gains confidence, and wants to do it again.

*Through* the things the child does, it will learn important things about cooking. It will learn what it is like to be in the kitchen, to use the utensils, to handle the food.

This is because using software, like cooking, is a matter of craft. It’s knowledge - but it is *practical* knowledge, not *theoretical* knowledge.

When we learn a new craft or skill, we always begin learning it by doing.

1.2.2 How to write good tutorials

**Allow the user to learn by doing**

**In the beginning, we only learn anything by doing** - it’s how we learn to talk, or walk.

In your software tutorial, your learner needs to *do* things. The different things that they do while following your tutorial need to cover a wide range of tools and operations, building up from the simplest ones at the start to more complex ones.
Get the user started

It’s perfectly acceptable if your beginner’s first steps are hand-held baby steps. It’s also perfectly acceptable if what you get the beginner to do is not the way an experienced person would, or even if it’s not the ‘correct’ way - a tutorial for beginners is not the same thing as a manual for best practice.

The point of a tutorial is to get your learner **started on their journey**, not to get them to a final destination.

Make sure that your tutorial works

One of your jobs as a tutor is to inspire the beginner’s confidence: in the software, in the tutorial, in the tutor and, of course, in their own ability to achieve what’s being asked of them.

There are many things that contribute to this. A friendly tone helps, as does consistent use of language, and a logical progression through the material. But the single most important thing is that **what you ask the beginner to do must work**. The learner needs to see that the actions you ask them to take have the effect you say they will have.

If the learner’s actions produce an error or unexpected results, your tutorial has failed - even if it’s not your fault. When your students are there with you, you can rescue them; if they’re reading your documentation on their own you can’t - so you have to prevent that from happening in advance. This is without doubt easier said than done.

Ensure the user sees results immediately

**Everything the learner does should accomplish something comprehensible, however small.** If your student has to do strange and incomprehensible things for two pages before they even see a result, that’s much too long. The effect of every action should be visible and evident as soon as possible, and the connection to the action should be clear.

The conclusion of each section of a tutorial, or the tutorial as a whole, must be a meaningful accomplishment.

Make your tutorial repeatable

**Your tutorial must be reliably repeatable.** This not easy to achieve: people will be coming to it with different operating systems, levels of experience and tools. What’s more, any software or resources they use are quite likely themselves to change in the meantime.

The tutorial has to work for all of them, every time.

Tutorials unfortunately need regular and detailed testing to make sure that they still work.

Focus on concrete steps, not abstract concepts

**Tutorials need to be concrete**, built around specific, particular actions and outcomes.

The temptation to introduce abstraction is huge; it is after all how most computing derives its power. But all learning proceeds from the particular and concrete to the general and abstract, and asking the learner to appreciate levels of abstraction before they have even had a chance to grasp the concrete is poor teaching.
Provide the minimum necessary explanation

Don’t explain anything the learner doesn’t need to know in order to complete the tutorial. Extended discussion is important - just not in a tutorial. In a tutorial, it is an obstruction and a distraction. Only the bare minimum is appropriate. Instead, link to explanations elsewhere in the documentation.

Focus only on the steps the user needs to take

Your tutorial needs to be focused on the task in hand. Maybe the command you’re introducing has many other options, or maybe there are different ways to access a certain API. It doesn’t matter: right now, your learner does not need to know about those in order to make progress.

1.2.3 Example from Divio’s documentation

Have a look at our tutorials.

Django

This tutorial is recommended as an introduction for all Python/Django developers.

- 1. Create a new Django project
- 2. Set up your project locally
- 3. Deploy your application to the cloud
- 4. Add and configure a Postgres database
- 5. Configure static file serving
- 6. Add and configure media storage
- 7. Apply some refinements

In particular, see the tutorial for Django. The promise that the tutorial makes is: if you have the basic knowledge required to follow this tutorial, and you follow its directions, you will end up with with a working Django web application, complete with Postgres database, S3 media storage, and so on. In order to work as a tutorial, it has to fulfil that promise.

Note that it doesn’t tell you what you will learn, just what you will do. The learning comes out of that doing. The tutorial takes full responsibility for what you will do and the order in which you will do it.

1.3 How-to guides

How-to guides take the reader through the steps required to solve a real-world problem.

They are recipes, directions to achieve a specific end - for example: how to create a web form; how to plot a three-dimensional data-set; how to enable LDAP authentication.

They are wholly goal-oriented.

How-to guides are wholly distinct from tutorials and must not be confused with them:

- A tutorial is what you decide a beginner needs to know.
- A how-to guide is an answer to a question that only a user with some experience could even formulate.
In a how-to guide, you can assume some knowledge and understanding. You can assume that the user already knows how to do basic things and use basic tools.

Unlike tutorials, how-to guides in software documentation tend to be done fairly well. They’re also fun and easy to write.

### 1.3.1 Analogy from cooking

Think about a recipe, for preparing something to eat.

A recipe has a clear, defined end. It addresses a specific question. It shows someone - who can be assumed to have some basic knowledge already - how to achieve something.

Someone who has never cooked before can’t be expected to follow a recipe with success, so a recipe is not a substitute for a cooking lesson. At the same time, someone who reads a recipe would be irritated to find that it tries to teach basics that they know already, or contains irrelevant discussion of the ingredients.

### 1.3.2 How to write good how-to guides

**Provide a series of steps**

**How-to guides must contain a list of steps, that need to be followed in order** (just like tutorials to). You don’t have to start at the very beginning, just at a reasonable starting point. How-to guides should be reliable, but they don’t need to have the cast-iron repeatability of a tutorial.
Focus on results

How-to guides must focus on achieving a practical goal. Anything else is a distraction. As in tutorials, detailed explanations are out of place here.

Solve a particular problem

A how-to guide must address a specific question or problem: How do I . . . ?

This is one way in which how-to guides are distinct from tutorials: when it comes to a how-to guide, the reader can be assumed to know what they should achieve, but don’t yet know how - whereas in the tutorial, you are responsible for deciding what things the reader needs to know about.

Don’t explain concepts

A how-to guide should not explain things. It’s not the place for discussions of that kind; they will simply get in the way of the action. If explanations are important, link to them.

Allow for some flexibility

A how-to guide should allow for slightly different ways of doing the same thing. It needs just enough flexibility in it that the user can see how it will apply to slightly different examples from the one you describe, or understand how to adapt it to a slightly different system or configuration from the one you’re assuming. Don’t be so specific that the guide is useless for anything except the exact purpose you have in mind.

Leave things out

Practical usability is more valuable than completeness. Tutorials need to be complete, end-to-end guides; how-to guides do not. They can start and end where it seems appropriate to you. They don’t need to mention everything that there is to mention either, just because it is related to the topic. A bloated how-to guide doesn’t help the user get speedily to their solution.

Name guides well

The title of a how-to document should tell the user exactly what it does. How to create a class-based view is a good title. Creating a class-based view or worse, Class-based views, are not.

1.3.3 Example from Divio’s documentation

Have a look at our how-to guides.
How-to guides

Working in the local development environment

- Get started with the Divio CLI
- Run the local server in Live configuration

Building your Docker application

- Manage a project’s base image
- Install system packages

Adding new functionality to a project

- Set up Sass CSS compilation
- Configure Application Performance Monitoring
- Configure media serving on a custom domain

The development pipeline

- Configure external Git hosting
- Use Git to manage your project

Each one of these is an answer to a question, or problem: how do I...? Each title can clearly be preceded by the words “How to”. Each one is a recipe, that takes you through the steps required to complete a specific task.

Although both the tutorials and the how-to guides serve the needs of the user, the tutorials are led by the author who knows what the user needs to know, while the how-to guides are led by the user who asks the questions.

1.4 Reference guides

Reference guides are technical descriptions of the machinery and how to operate it.

Reference guides have one job only: to describe. They are code-determined, because ultimately that’s what they describe: key classes, functions, APIs, and so they should list things like functions, fields, attributes and methods, and set out how to use them.

Reference material is information-oriented.

By all means technical reference can contain examples to illustrate usage, but it should not attempt to explain basic concepts, or how to achieve common tasks.

Reference material should be austere and to the point.

Note that description does include basic description of how to use the machinery - how to instantiate a particular class, or invoke a certain method, for example, or precautions that must be taken when passing something to a function. However this is simply part of its function as technical reference, and emphatically not to be confused with a how-to guide - describing correct usage of software (technical reference) is not the same as showing how to use it to achieve a certain end (how-to documentation).

For some developers, reference guides are the only kind of documentation they can imagine. They already understand their software, they know how to use it. All they can imagine that other people might need is technical information about it.

Reference material tends to be written well. It can even - to some extent - be generated automatically, but this is never sufficient on its own.
1.4.1 Analogy from cooking

Consider an encyclopaedia article about an ingredient, say ginger.

When you look up *ginger* in a reference work, what you want is *information* about the ingredient - information describing its provenance, its behaviour, its chemical constituents, how it can be cooked.

You expect that whatever ingredient you look up, the information will be presented in a similar way. And you expect to be informed of basic facts, such as *ginger is a member of the family that includes turmeric and cardamom*.

This is also where you’d expect to be alerted about potential problems, such as: *ginger is known to provoke heartburn in some individuals* or: *ginger may interfere with the effects of anticoagulants, such as warfarin or aspirin*.

1.4.2 How to write good reference guides

Structure the documentation around the code

*Give reference documentation the same structure as the codebase*, so that the user can navigate both the code and the documentation for it at the same time. This will also help the maintainers see where reference documentation is missing or needs to be updated.
Be consistent

In reference guides, structure, tone, format must all be consistent - as consistent as those of an encyclopaedia or dictionary.

Do nothing but describe

The only job of technical reference is to describe, as clearly and completely as possible. Anything else (explanation, discussion, instruction, speculation, opinion) is not only a distraction, but will make it harder to use and maintain. Provide examples to illustrate the description when appropriate.

Avoid the temptation to use reference material to instruct in how to achieve things, beyond the basic scope of using the software, and don’t allow explanations of concepts or discussions of topics to develop. Instead, link to how-to guides, explanation and introductory tutorials as appropriate.

Be accurate

These descriptions must be accurate and kept up-to-date. Any discrepancy between the machinery and your description of it will inevitably lead a user astray.

1.4.3 Example from Divio’s documentation

Have a look at an example from our technical reference section.

This is a typical reference guide (in this case, for our Divio CLI).

Description is all this article does, setting out in a complete and accurate form the functions, commands and options of the tool.

It’s hardly a friendly or engaging read, but its purpose is to make looking up information about functionality as swift and distraction-free as possible.
1.5 Explanation

Explanation, or discussions, clarify and illuminate a particular topic. They broaden the documentation’s coverage of a topic. They are understanding-oriented.

Explanations can equally well be described as discussions; they are discursive in nature. They are a chance for the documentation to relax and step back from the software, taking a wider view, illuminating it from a higher level or even from different perspectives. You might imagine a discussion document being read at leisure, rather than over the code.

This section of documentation is rarely explicitly created, and instead, snippets of explanation are scattered amongst other sections. Sometimes, the section exists, but has a name such as Background or Other notes or Key topics - these names are not always useful.

Discussions are less easy to create than it might seem - things that are straightforward to explain when you have the starting-point of someone’s question are less easy when you have a blank page and have to write down something about it.

A topic isn’t defined by a specific task you want to achieve, like a how-to guide, or what you want the user to learn, like a tutorial. It’s not defined by a piece of the machinery, like reference material. It’s defined by what you think is a reasonable area to try to cover at one time, so the division of topics for discussion can sometimes be a little arbitrary.

1.5.1 Analogy from cooking

McGEE on FOOD & COOKING
AN ENCYCLOPEDIA OF KITCHEN SCIENCE, HISTORY AND CULTURE

“One of the greatest cookery books ever written” Heston Blumenthal
HAROLD Mcgee
Think about a work that discusses food and cooking in the context of history, science and technology. It’s *about* cooking and the kitchen.

It doesn’t teach, it’s not a collection of recipes, and it doesn’t just describe.

Instead, it analyses, considers things from multiple perspectives. It might explain why it is we now do things the way we do, or even describe bad ways of doing things, or obscure alternatives.

It deepens our knowledge and makes it richer, even if it isn’t knowledge we can actually apply in any practical sense - but it doesn’t need to be, in order to be valuable.

It’s something we might read at our leisure, away from the kitchen itself, when we want to think about cooking at a higher level, and to understand more about the subject.

### 1.5.2 How to write a good explanation

**Provide context**

Explanations are the place for **background and context** - for example, *Web forms and how they are handled in Django*, or *Search in django CMS*.

They can also explain why things are so - design decisions, historical reasons, technical constraints.

**Discuss alternatives and opinions**

Explanation can consider alternatives, or multiple different approaches to the same question. For example, in an article on Django deployment, it would be appropriate to consider and evaluate different web server options.

Discussions can even consider and weigh up contrary opinions - for example, whether test modules should be in a package directory, or not.

**Don’t instruct, or provide technical reference**

Explanation should do things that the other parts of the documentation do not. It’s not the place of an explanation to instruct the user in how to do something. Nor should it provide technical description. These functions of documentation are already taken care of in other sections.

### 1.5.3 Example from Divio’s documentation

Have a look at our explanation section (titled “Background” - the name is not important as long as the purpose is clear).
These articles don’t teach anything. They don’t tell the user what to do. They aren’t reference guides. They just discuss particular topics. The user doesn’t need to know about (for example) caching or CDN or how we manage environment variables in order to use the platform or achieve any particular task, but the time is likely to come when someone’s experience and use of the platform will be improved by having a clearer, better, deeper understanding of those things.

These articles provide the bigger picture, the context. Users are human beings; maybe they don’t strictly need to know why we do a certain thing a certain way, but knowing it might well provide them with a kind of satisfaction and comfort that makes them a happier user of the product.

1.6 About the structure

1.6.1 Why isn’t this obvious?

This structure is clear, and it works, but there is a reason why it’s not so obvious, and that is the way the characteristics of each quadrant of the documentation overlap with those of its neighbours in the scheme.
Each of the quadrants is similar to its two neighbours:

- **tutorials and how-to guides** are both concerned with **describing practical steps**
- **how-to guides and technical reference** are both **what we need when we are at work, coding**
- **reference guides and explanation** are both concerned with **theoretical knowledge**
- **tutorials and explanation** are both **most useful when we are studying**, rather than actually working

**The tendency to collapse**

Given these overlaps, it’s not surprising that the different kinds of documentation become confused and mixed in with each other. In fact, there is a natural gravitational pull of these distinct types of documentation to each other, and it is hard to resist. Its effect is to collapse the structure, and that is why so much documentation looks like this:
1.6.2 Adoption of the system

Though it’s rare to find it clear examples of it used fully, a great deal of documentation recognises, in different ways, each of these four functions.

Good examples of the scheme in substantial projects include:

- the Divio Developer Handbook
- Django’s documentation
- django CMS’s documentation

It’s possible to use the system even in very minimal documentation, for example CoReport (an open-source COVID-19 reporting project). Here, applying the system creates a framework for future documentation, helping ensure that new material will conform.

Sometimes the documentation is so minimal that not all quadrants are ready to be represented, as in the case of Getting started with Java and Spring-boot, which includes only a tutorial, how-to and reference material.

But I never wanted to do DevOps! is the written material that accompanies a popular workshop. The documentation strictly separate the tutorial, the steps learners are to follow, from the explanation (Further reading). Both belong to the most useful when we are studying side of the system, so it’s natural to include them in a workshop.

In each case though, however minimal or even incomplete, the system is respected and the clear distinction between sections and their purposes will benefit the author and user right away, and help guide the expansion of the material as it develops in the future.
1.6.3 About the analysis and its application

The analysis of documentation in this article is based on several years of experience writing and maintaining documentation, and much time spent considering how to improve it.

It’s also based on sound principles that come from a variety of disciplines. For example, its conception of tutorials has a pedagogical basis; it posits a tutor and a learner, and considers using software to be a craft in which abstract understanding of general principles follows from concrete steps that deal with particulars.

The system is presented regularly at talks and interactive workshops. The analysis has been applied to numerous projects, including large internal documentation sets, and has repeatedly procured benefits of usability and maintainability, across a very wide range of technical subject matter.

1.7 Who is using the system?

Not listed here?

If you’re using the system, or are in the process of adopting it, please let us know if you’d like to be listed here.

This is an incomplete list of projects, products and organisations that have adopted the system in their own bodies of documentation. In some cases the adoption remains partial or is still a work in progress.

- BrachioGraph, the cheapest, simplest pen-plotter
- BeeWare, the write-once-deploy-anywhere project, for Toga, Briefcase, Rubicon and Rubicon Java.
- Bosch (internal)
- Ciw, the discrete event simulation library
- Cloudflare Workers docs (related article, New and improved Workers Docs)
- Divio
- Django
- django CMS
- edo, a library for Evolutionary Dataset Optimisation
- Ericsson (internal)
- Gensim, How to Author Gensim Documentation
- Snowpack, a frontend build tool, designed for the modern web
- StrongLoop/LoopBack by IBM
- Lisk
- Matching, a games theory resource allocation library
- NashPy, a Python mathematical library for computing Nash equilibria
- nbchkr, a system for assessing students’ assignments in Jupyter Notebooks
- NumPy, the scientific Python library (related article, Documentation as a way to build Community)
- PDFminer.six
- PostgREST
Documentation system

- PIconnect
- Tesla Motors (internal)
- WebAccess/DMP
- Wechaty
- Zalando (internal)

1.7.1 Other mentions and references of interest

- Django Axes proposal
- GitLab, GitLab’s Data Team documentation guide
- Julia language proposal
- Why You Should Document Your Work As a Data Scientist
- Koninglijke Bibliotheek (National Library of the Netherlands) research software lab
- Tutorials in Jenkins user documentation
- TYPO3
If you’d prefer to watch a video covering this topic, here is it (courtesy of PyCon Australia 2017).