What nobody tells you about documentation

Daniele Procida May 19, 2017

21-26 minutes

However hard you work on documentation, it won't work for your software - unless you do it the right way.

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*, *explanation* and *technical reference.* They represent four different purposes or functions, and require four different approaches to their creation. Understanding the implications of this will help improve most software documentation - often immensely.

If you'd prefer to watch a video covering this topic, here is it (courtesy of PyCon Australia 2017).

Introduction

It doesn't matter how good your software is, because **if the documentation is not good enough, people will not use it**.

Even if for some reason 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 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.

In this article I will explain how you can 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.

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 secret

Documentation needs to include and be structured around its **four different functions:** *tutorials*, *how-to guides*, *explanation* and *technical reference.* 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 documentation needs to be explicitly structured around them, and they all must be kept separate and distinct from each other.

Tutorials

A tutorial:

- is **learning-oriented**
- allows the newcomer to get started
- is a lesson

Analogy: teaching a small child how to cook

How-to guides

A how-to guide:

- is **goal-oriented**
- shows how to solve a specific problem
- is a series of steps

Analogy: a recipe in a cookery book

Explanation

An explanation:

- is **understanding-oriented**
- explains
- provides background and context

Analogy: an article on culinary social history

Reference

A reference guide:

- is **information-oriented**
- describes the machinery
- is accurate and complete

Analogy: a reference encyclopaedia article

This division makes it obvious to both author and reader what information 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.

Project documentation

You may well ask: where do things like changelogs, contribution policies, and other information about the project fit into this scheme? The answer is that they do not - because they are, strictly speaking, *project documentation* rather than *documentation of the software itself*,

They can simply be kept in appropriately-named sections alongside the other material - as long as they are not mixed up *in* it.

With that in mind, let's explore each of the four key functions.

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.

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.

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.**

Good tutorials are very difficult to write. They need to be useful for the beginner, easy to follow, meaningful and extremely robust.

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.

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**.

If you'd like an analogy, 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.

How-to guides are quite distinct from tutorials. A how-to guide is an answer to a question that a true beginner might not even be able to 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.

How-to guides in software documentation tend to be done fairly well. They're also fun and easy to write.

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 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 THEM 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.

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 achieve common tasks.

Reference material should be **austere and to the point**.

The culinary analogy might be an ecyclopaedia article about an ingredient, describing its provenance, its behaviour, its chemical constituents, how it can be cooked.

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.

How to write good reference documentation

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-todate. Any discrepancy between the machinery and your description of it will inevitably lead a user astray.

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 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* and doesn't really do justice to the function.

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 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.

How to write good explanation

PROVIDE CONTEXT

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

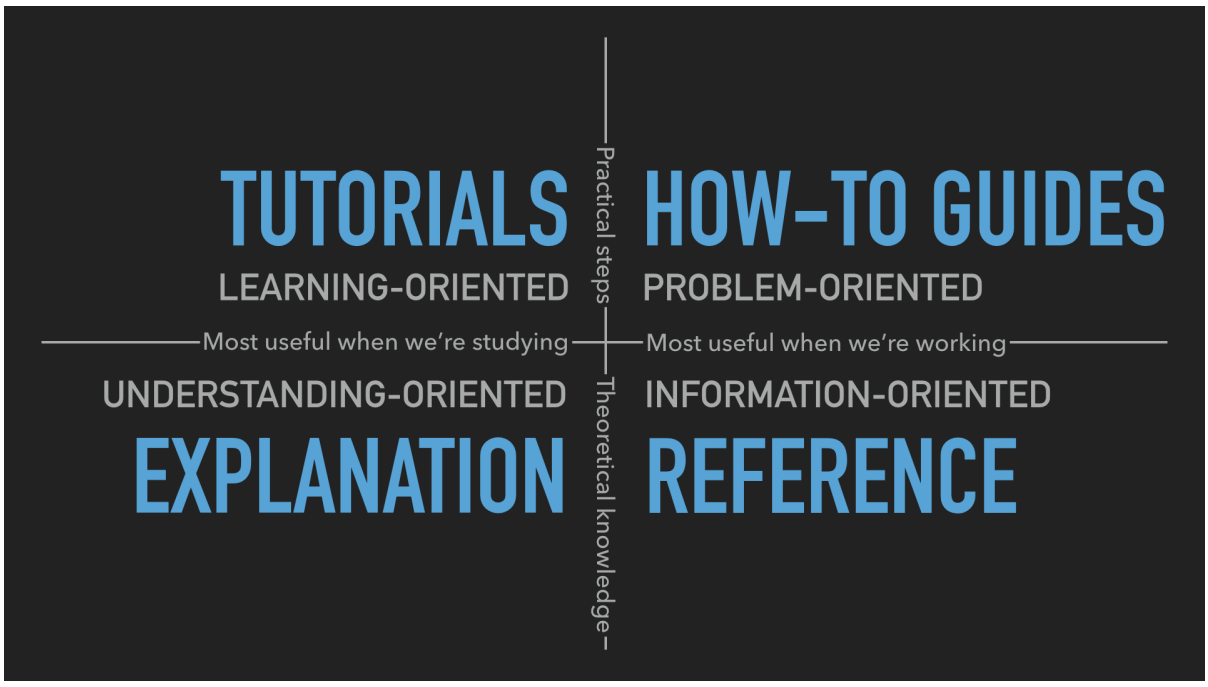
DON'T INSTRUCT, OR PROVIDE TECHNICAL REFERENCE

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 describe. These functions of documentation are already taken care of in other sections.

About the structure

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.



Tutorials and how-to guides are similar because they are both concerned with **describing practical steps**, while what *how-to guides* share with *technical reference*is that they're **what we need when we are actually at work, coding**. *Reference guides* and *explanation* are similar because they're concerned with **theoretical knowledge**, and finally, what *tutorials* have in common with *explanation* is that they **are most useful when we are studying**, rather than actually working:

	Most useful when we're studying	**Most useful when we're working**
Practical steps	Tutorials	How-to guides
Theoretical knowledge	Explanation	Reference

Given these overlaps, it's not surprising that the different kinds of documentation become confused and mixed in with each other.

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.

Some projects do adopt it fully, including Django (though this wasn't made explicit in earlier versions), and django CMS. It has proved its worth in both projects.

About the analysis

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.

Making documentation work

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.

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.

About the Divio Expert series

Our experts really do deserve the name. Between them they have decades of Django development experience; the depth and breadth of the expertise we are able able to offer is world-class. They include core developers of Django, django CMS and other notable projects, and the architects of very large systems such as the Divio Cloud infrastructure.

The new Divio Expert weblog series aims to share some of this expertise. For updates on new articles, follow our Twitter feed, or subscribe to our (very low volume) newsletter.