

An incomplete rant about the state of the documentation for NixOS

This article was originally published at

<https://gist.github.com/joepie91/5232c8f1e75a8f54367e5dfcfd573726>.

Historical note: I wrote this rant in 2017, originally intended to be posted on the NixOS forums. This never ended up happening, as discussing the (then private) draft already started driving changes to the documentation approach. The documentation has improved since this was written, however some issues remain to this day at the time of writing this remark, in 2024. The rant ends abruptly, because I never ended up finishing it - but it still contains a lot of useful points regarding documentation quality, and so I am preserving it here.

I've now been using NixOS on my main system for a few months, and while I appreciate the technical benefits a lot, I'm constantly running into walls concerning documentation and general problem-solving. After discussing this briefly on IRC in the past, I've decided to post a rant / essay / whatever-you-want-to-call-it here.

An upfront note

My frustration about these issues has built up considerably over the past few months, more so because I know that *from a technical perspective* it all makes a lot of sense, and there's a lot of potential behind NixOS. However, I've found it pretty much impenetrable on a getting-stuff-done level, because the documentation on many things is either poor or non-existent.

While my goal here is to *get things fixed* rather than just complaining about them, that frustration might occasionally shine through, and so I might come across as a bit harsh. This is not my intention, and there's no ill will towards any of the maintainers or users. I just want to address the issues head-on, and get them fixed effectively.

To address any "*just send in a PR*" comments ahead of time: while I do know how to write good documentation (and I do so [on a regular basis](#)), I still don't understand much of how NixOS and nixpkgs are structured, exactly *because* the documentation is so poorly accessible. I couldn't fix the documentation myself if I wanted to, simply because I don't have the understanding required to do so, and I'm finding it very hard to obtain that understanding.

One last remark: throughout the rant, I'll be posing a number of questions. These are not necessarily all questions that I *still have*, as I've found the answer to several of them after hours of research - they just serve to illustrate the interpretation of the documentation from the point of view of a beginner, so there's no need to try and answer them in this thread. These are just the type of questions that should be anticipated and answered in the documentation.

Types of documentation

Roughly speaking, there are three types of documentation for anything programming-related:

1. Reference documentation
2. Conceptual documentation
3. Tutorials

In the sections below, "tooling" will refer to any kind of to-be-documented thing - a function, an API call, a command-line tool, and so on.

Reference documentation

Reference documentation is intended for readers who are *already familiar* with the tooling that is being documented. It typically follows a rigorous format, and defines things such as function names, arguments, return values, error conditions, and so on. Reference documentation is generally considered the "single source of truth" - whatever behaviour is specified there, is what the tooling should actually do.

Some examples of reference documentation:

- <https://nodejs.org/api/querystring.html>
- <https://doc.rust-lang.org/std/>

Reference documentation generally assumes all of the following:

- The reader understands the purpose of the tooling
- The reader understands the concepts that the tooling uses or implements
- The reader understands the relation of the tooling to other tooling

Conceptual documentation

Conceptual documentation is intended for readers who do not yet understand the tooling, but are already familiar with the environment (language, shell, etc.) in which it's used.

Some examples of conceptual documentation:

- <http://crypto.net/~joepie91/blog/2016/05/11/what-is-promise-try-and-why-does-it-matter/>
- [https://hughfdjackson.com/javascript/prototypes-the-short\(est-possible\)-story/](https://hughfdjackson.com/javascript/prototypes-the-short(est-possible)-story/)
- <https://doc.rust-lang.org/stable/book/the-stack-and-the-heap.html>

Good conceptual documentation doesn't make any assumptions about the background of the reader or what other tooling they might already know about, and explicitly indicates any prior knowledge that's required to understand the documentation - preferably including a link to documentation about those "dependency topics".

Tutorials

Tutorials can be intended for two different groups of readers:

1. Readers who don't yet understand the environment (eg. "Introduction to Bash syntax")
2. Readers who don't *want* to understand the environment (eg. "How to build a full-stack web application")

While I would consider tutorials pandering to the second category actively harmful, they're a thing that exists nevertheless.

Some examples of tutorials:

- <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide>
- <https://zellwk.com/blog/crud-express-mongodb/>
- <http://www.freewebmasterhelp.com/tutorials/phpmysql>

Tutorials don't make *any* assumptions about the background of the reader... but they have to be read from start to end. Starting in the middle of a tutorial is not likely to be useful, as tutorials are more designed to "hand-hold" the reader through the process (without necessarily understanding *why* things work how they work).

The current state of the Nix(OS) documentation

Unfortunately, the NixOS documentation is currently lacking in all three areas.

The official Nix, NixOS and nixpkgs manuals attempt to be all three types of documentation - tutorials (like [this one](#)), conceptual documentation (like [this](#)), and reference documentation (like [this](#)). The wiki *sort of* tries to be conceptual documentation (like [here](#)), and does so a little better than the manual, but... the wiki is being shut down, and it's still far from complete.

The most lacking aspect of the NixOS documentation is currently the conceptual documentation. What is a "derivation"? Why does it exist? How does it relate to what I, as a user, want to do? How is the Nix store structured, and what guarantees does this give me? What is the difference between `/etc/nixos/configuration.nix` and `~/.nixpkgs/config.nix`, and can they be used interchangeably? Is `nixpkgs` just a set of packages, or does it also include tooling? Which tooling is provided by Nix the package manager, which is provided by NixOS, and which is provided by nixpkgs? Is this different on non-NixOS, and why?

Most of the official documentation - including the wiki - is structured more like a very extensive tutorial. You're told, step by step, *what* to do... but not why any of it matters, what it's for, or how to use these techniques in different situations. [This wiki section](#) is a good example. What does `overrideDerivation` actually *do*? What's the difference with `override`? What's the difference between 'attributes' and 'arguments'? Why is there a random link about the Oracle JDK there? Is the `src` *completely* overridden, or just the attributes that are specified there? What if I want to reevaluate all the other attributes based on the changes that I've made - for example, regenerating the `name` attribute based on a changed `version` attribute? Are any of these tools useful in *other* scenarios that aren't directly addressed here?

The "[Nix pills](#)" sort of try to address this lack of conceptual information, and are quite informational, but they have their problems too. They are not clearly structured (where's the index of all the articles?), the text formatting can be hard to read, and it is still half of a tutorial - it can be hard to understand later pills without having read earlier ones, because they're not fully self-contained. On top of that, they're third-party documentation and not part of the official documentation.

The official manuals have a number of formatting/structural issues as well. The single-page format is frankly *horrible* for navigating through - finding anything on the page is difficult, and following links to other things gets messy fast. Because it's all a single page, every tab has the exact same title, it's easy to scroll past the section you were reading, and so on. Half the point of the web is to have *hyperlinked content across multiple documents*, but the manuals completely forgo that and create a really poor user experience. It's awful for search engines too, because no matter what you search for, you always end up *on the exact same page*.

Another problem is the fact that I have to say "manuals" - there are *multiple manuals*, and the distinction between them is not at all clear. Because it's unclear what functionality is provided by what part of the stack, it usually becomes a hunt of going through all three manuals ctrl+F'ing for some keywords, and hoping that you will run into the thing you're looking for. Then once you (hopefully) do, you have to be careful not to accidentally scroll away from it and lose your reference. There's really no good reason for this separation; it just makes it harder to cross-reference between different parts of the stack, and most users will be using all of them anyway.

The manual, as it is, is not a viable format. While I understand that the wiki had issues with outdated information, it's still a far better *structure* than a set of single-page manuals. I'll go into more detail at the end of this rant, but my proposed solution here would be to follow a wiki-like format for the official documentation.

Missing documentation

Aside from the issues with the documentation *format*, there are also plenty of issues with its *content*. Many things are fully undocumented, especially where `nixpkgs` is concerned. For example, nothing says that I should be using `callPackage_i686` to package something with 32-bits dependencies. Or how to package something that requires the user to manually add a source file from their filesystem using `nix-prefetch-url`, or using `nix-store --add-fixed`. And what's the difference between those two anyway? And why is there a separate `qt5.callPackage`, and when do I need it?

There are a *ton* of situations where you need oddball solutions to get something packaged. In fact, I would argue that this is the *majority* of cases - most of the easy pickings have been packaged by now, and the tricky ones are left. But as a new user that just wants to get an application working, I end up spending *several hours* on each of the above questions, and I'm still not convinced that I have the right answer. Had somebody taken 10 minutes to document this, even if just as a rough note, it would have saved me *hours* of work.

No clear path to solutions

When faced with a given packaging problem, it's not at all obvious how to get to the solution. There's no obvious process for fixing or debugging issues, and error messages are often cryptic or poorly formatted. What does "cannot coerce a set to a string" mean, and why is it happening? How can I duct-tape-debug something by adding a `print` statement of some variety? Is there an interactive debugger of some sort?

It's very difficult to learn enough about NixOS internals to figure out what the *right* way is to package any given thing, and because there's no good feedback on what's *wrong* either, it's too hard to get anything packaged that isn't a standard autotools build. There's no "Frequently Asked Questions" or "Common Packaging Problems" section, nor have I found any useful tooling for analyzing packaging problems in more detail. I've had to write some of this tooling myself!

The documentation should anticipate the common problems that new users run into, and give them some hints on where to start looking. It currently completely fails to do so, and assumes that the users will figure out the relation between things themselves.

Reading code

Because of the above issues, often the only solution is to read the code of existing packages, and try to infer from their expressions how to approach certain problems - but that comes with its own set of problems. There does not appear to be a consistent way of solving packaging problems in NixOS, and almost every package seems to have invented its own way of solving the same problems that other packages have already solved. After several hours of research, it often turns out that half the solutions are either outdated or just wrong. And then I *still* have no idea what the optimal solution is, out of the remaining options.

This is made worse by the serious lack of comments in `nixpkgs`. Barely any packages have comments *at all*, and frequently there are complex multi-level abstractions in place to solve certain problems, but with absolutely no information to explain *why* those abstractions exist. They're not exactly self-evident either. Then there are the packages that *do* have comments, but they're aimed at the *user* rather than the *packager* - one such example is the [Guake package](#). Essentially, it seems the repository is absolutely full of hacks with no standardized way of solving problems, no doubt helped by the fact that existing solutions simply *aren't documented*.

This is a tremendous waste of time for everybody involved, and makes it very hard to package anything unusual, often to the point of just giving up and hacking around the issue in an impure way. Right now we have what seems like a significant amount of people doing the same work over and over and over again, resulting in different implementations every time. If people took the time to *document* their solutions, this problem would pretty much instantly go away. From a technical point of view, there's absolutely no reason for packaging to be this hard to do.

Tooling

On top of all this, the tooling seems to change *constantly* - abstractions get deprecated, added, renamed, moved, and so on. Many of the `stdenv` abstractions aren't documented, or their documentation is incomplete. There's no clear way to determine which tooling is still in use, and which tooling has been deprecated.

The tooling that *is* in use - in particular the command-line tooling - is often poorly designed from a usability perspective. Different tools using different flags for the same purpose, behaving differently in different scenarios for no obvious reason. There's a [UX proposal](#) that seems to fix many of these problems, but it seems to be more or less dead, and its existence is not widely known.

Revision #1

Created 11 December 2024 16:10:28 by joepie91

Updated 11 December 2024 16:14:45 by joepie91