NixOS

- Setting up Bookstack
- A *complete* listing of operators in Nix, and their predence.
- Setting up Hydra
- Fixing root filesystem errors with fsck on NixOS
- Stepping through builder steps in your custom packages
- Using dependencies in your build phases
- Source roots that need to be renamed before they can be used
- Error: `error: cannot coerce a function to a string`
- `buildInputs` vs. `nativeBuildInputs`?
- QMake ignores my `PREFIX`/`INSTALL_PREFIX`/etc. variables!
- Useful tools for working with NixOS
- Proprietary AMD drivers (fglrx) causing fatal error in i387.h
- Installing a few packages from `master`
- GRUB2 on UEFI
- Unblock ports in the firewall on NixOS
- Guake doesn't start because of a GConf issue
- FFMpeg support in youtube-dl
- An incomplete rant about the state of the documentation for NixOS
- gparted doesn't let me create a btrfs partition!
- Dealing with Bookstack breakage in a NixOS 25.11 upgrade

Setting up Bookstack

Turned out to be pretty simple.

```
deployment.secrets.bookstack-app-key = {
  source = "../private/bookstack/app-key";
  destination = "/var/lib/bookstack/app-key";
  owner = { user = "bookstack"; group = "bookstack"; };
  permissions = "0700";
};

services.bookstack = {
  enable = true;
  hostname = "wiki.slightly.tech";
  maxUploadSize = "10G";
  appKeyFile = "/var/lib/bookstack/app-key";
  nginx = { enableACME = true; forceSSL = true; };
  database = { createLocally = true; };
};
```

Server was running an old version of NixOS, 23.05, where MySQL doesn't work in a VPS (anymore). Upgraded the whole thing to 24.11 and then it Just Worked.

Afterwards, run:

```
bookstack bookstack:create-admin
```

... in a terminal on the server to set up the primary administrator account. Done.

A *complete* listing of operators in Nix, and their predence.

> This article was originally published at https://gist.github.com/joepie91/c3c047f3406aea9ec65eebce2ffd449d.

> The information in this article has since been absorbed into the official Nix manual. It is kept here for posterity. It may be outdated by the time you read this.

Lower precedence means a stronger binding; ie. this list is sorted from strongest to weakest binding, and in the case of equal precedence between two operators, the associativity decides the binding.

Prec	Abbreviation	Example	Assoc	Description
1	SELECT	`e . attrpath [or def]`	none	Select attribute denoted by the attribute path `attrpath` from set `e`. (An attribute path is a dot-separated list of attribute names.) If the attribute doesn't exist, return `default` if provided, otherwise abort evaluation.
2	APP	`e1 e2`	left	Call function `e1` with argument `e2`.
3	NEG	`-e`	none	Numeric negation.
4	HAS_ATTR	`e ? attrpath`	none	Test whether set `e` contains the attribute denoted by `attrpath`; return true or false.
5	CONCAT	`e1 ++ e2`	right	List concatenation.
6	MUL	`e1 * e2`	left	Numeric multiplication.
6	DIV	`e1 / e2`	left	Numeric division.
7	ADD	`e1 + e2`	left	Numeric addition, or string concatenation.
7	SUB	`e1 - e2`	left	Numeric subtraction.
8	NOT	`!e`	left	Boolean negation.

Prec	Abbreviation	Example	Assoc	Description
9	UPDATE	`e1 // e2`	right	Return a set consisting of the attributes in `e1` and `e2` (with the latter taking precedence over the former in case of equally named attributes).
10	LT	`e1 < e2`	left	Less than.
10	LTE	`e1 <= e2`	left	Less than or equal.
10	GT	`e1 > e2`	left	Greater than.
10	GTE	`e1 >= e2`	left	Greater than or equal.
11	EQ	`e1 == e2`	none	Equality.
11	NEQ	`e1 != e2`	none	Inequality.
12	AND	`e1 && e2`	left	Logical AND.
13	OR	`e1 || e2`	left	Logical OR.
14	IMPL	`e1 -> e2`	none	Logical implication (equivalent to `!e1 || e2`).

Setting up Hydra

> This article was originally published at https://gist.github.com/joepie91/c26f01a787af87a96f967219234a8723 in 2017. The NixOS ecosystem constantly changes, and it may not be relevant anymore by the time you read this article.

Just some notes from my attempt at setting up Hydra.

Setting up on NixOS

No need for manual database creation and all that; just ensure that your PostgreSQL service is running (`services.postgresql.enable = true;`), and then enable the Hydra service (`services.hydra.enable`). The Hydra service will need a few more options to be set up, below is my configuration for it:

```
    services.hydra = {
        enable = true;
        port = 3333;
        hydraURL = "http://localhost:3333/";
        notificationSender = "hydra@cryto.net";
        useSubstitutes = true;
        minimumDiskFree = 20;
        minimumDiskFreeEvaluator = 20;
    };
```

Database and user creation and all that will happen automatically. You'll only need to run `hydra-init` and then `hydra-create-user` to create the first user. Note that you *may* need to run these scripts as root if you get permission or filesystem errors.

Can't run `hydra-*` utility scripts / access the web interface due to database errors

If you already have a `services.postgresql.authentication` configuration line from elsewhere (either another service, or your own `configuration.nix`), it may be conflicting with the one specified in the

Hydra service. There's an open issue about it here.

Can't login

After running `hydra-create-user` in your shell, you may be running into the following error in the web interface: "Bad username or password."

When this occurs, it's likely because the `hydra-*` utility scripts stored your data in a local SQLite database, rather than the PostgreSQL database you configured. As far as I can tell, this happens because of some missing `HYDRA_*` environment variables that are set through `/etc/profile`, which is only applied on your next login. Simply opening a new shell is not enough.

As a workaround until your next login/boot, you can run the following to obtain the command you need to run to apply the new environment variables in your current shell:

```
cat /etc/profile | grep set-environment
```

... and then run the resulting command (including the dot at the start, if there is one!) in the shell you intend to run the `hydra-*` scripts in. If you intend to run them as root, make sure you run the `set-environment` script *in the root shell* - using `sudo` will make the environment variables get lost, so you'll be stuck with the same issue as before.

Fixing root filesystem errors with fsck on NixOS

If you run into an error like this:

```
An error occurred in stage 1 of the boot process, which must mount the root filesystem on
`/mnt-root` and then start stage 2. Press one of the following keys:

r) to reboot immediately
*) to ignore the error and continue
```

Then you can fix it like this:

1. Boot into a live CD/DVD for NixOS, or some other environment that has `fsck` installed, but *not* your installed copy of NixOS (as that will mount the root filesystem) (source)
2. Run `fsck -yf /dev/sda1` where you replace `/dev/sda1` with your root filesystem. (source)
   - If you're on a (KVM) VPS, it'll probably be `/dev/vda1`. If you're using LVM (even on a VPS), then you need to specify your logical volume instead (eg. `/dev/vg_main/lv_root`, but it depends on what you've named it).

> The above command will automatically agree to whatever suggestion fsck makes. **This can technically lead to data loss!**

Many distributions will give you an option to drop down into a shell from the error directly, but NixOS does not do that. In theory you could add the `boot.shell_on_fail` flag to the boot options for your existing installation, but for reasons that I didn't bother debugging any further, the installed `fsck` was unable to fix the issues.

Stepping through builder steps in your custom packages

> This article was originally published at https://gist.github.com/joepie91/b0041188c043259e6e6e1059d026eff301.

1. Create a temporary building folder in your repository (or elsewhere) and enter it: `mkdir test && cd test`
2. `nix-shell ../main.nix -A packagename` (assuming the entry point for your custom repository is `main.nix` in the parent directory)
3. Run the phases individually by entering their name (for a default phase) or doing something like `eval "$buildPhase"` (for an overridden phase) in the Nix shell - a summary of the common ones: `unpackPhase`, `patchPhase`, `configurePhase`, `buildPhase`, `checkPhase`, `installPhase`, `fixupPhase`, `distPhase`

More information about these phases can be found here. If you use a different builder, you may have a different set of phases.

Don't forget to clear out your `test` folder after every attempt!

Using dependencies in your build phases

> This article was originally published at https://gist.github.com/joepie91/b0041188c043259e6e1059d026eff301.

You can just use string interpolation to add a dependency path to your script. For example:

```
{
  # ...
  preBuildPhase = ''
    ${grunt-cli}/bin/grunt prepare
  '';
  # ...
}
```

Source roots that need to be renamed before they can be used

> This article was originally published at https://gist.github.com/joepie91/b0041188c043259e6e6e1059d026eff301.

Some applications (such as Brackets) are very picky about the directory name(s) of your unpacked source(s). In this case, you might need to rename one or more source roots *before* `cd`ing into them.

To accomplish this, do something like the following:

```
{
  # ...
  sourceRoot = ".";

  postUnpack = ''
    mv brackets-release-${version} brackets
    mv brackets-shell-${shellBranch} brackets-shell
    cd brackets-shell;
  '';
  # ...
}
```

This keeps Nix from trying to move into the source directories immediately, by explicitly pointing it at the current (ie. top-most) directory of the environment.

Error: `error: cannot coerce a function to a string`

> This article was originally published at https://gist.github.com/joepie91/b0041188c043259e6e1059d026eff301.

Probably caused by a syntax ambiguity when invoking functions within a list. For example, the following will throw this error:

```
{
  # ...
  srcs = [
    fetchurl {
      url = "https://github.com/adobe/brackets-shell/archive/${shellBranch}.tar.gz";
      sha256 = shellHash;
    }
    fetchurl {
      url = "https://github.com/adobe/brackets/archive/release-${version}.tar.gz";
      sha256 = "00yc81p30yamr86pliwd465ag1lnbx8j01h7a0a63i7hsq4vvvvg";
    }
  ];
  # ...
}
```

This can be solved by adding parentheses around the invocations:

```
{
  # ...
  srcs = [
    (fetchurl {
      url = "https://github.com/adobe/brackets-shell/archive/${shellBranch}.tar.gz";
      sha256 = shellHash;
    })
    (fetchurl {
      url = "https://github.com/adobe/brackets/archive/release-${version}.tar.gz";
      sha256 = "00yc81p30yamr86pliwd465ag1lnbx8j01h7a0a63i7hsq4vvvvg";
    })
  ];
  # ...
}
```

`buildInputs` vs. `nativeBuildInputs`?

> This article was originally published at https://gist.github.com/joepie91/b0041188c043259e6e1059d026eff301.

More can be found here.

- **buildInputs:** Dependencies for the (target) system that your built package will eventually run on.
- **nativeBuildInputs:** Dependencies for the system where the build is being created.

The difference only really matters when cross-building - when building for your own system, *both* sets of dependencies will be exposed as `nativeBuildInputs`.

QMake ignores my `PREFIX`/`INSTALL_PREFIX`/etc. variables!

> This article was originally published at https://gist.github.com/joepie91/b0041188c043259e6e1059d026eff301.

QMake does not have a standardized configuration variable for installation prefixes - `PREFIX` and `INSTALL_PREFIX` only work if the project files for the software you're building specify it explicitly.

If the project files have a hardcoded path, there's still a workaround to install it in `$out` anyway, *without* source code or project file patches:

```
{
  # ...
  preInstall = "export INSTALL_ROOT=$out";
  # ...
}
```

This `INSTALL_ROOT` environment variable will be picked up and used by `make install`, *regardless* of the paths specified by QMake.

Useful tools for working with NixOS

> This article was originally published at https://gist.github.com/joepie91/67316a114a860d4ac6a9480a6e1d9c5c. Some links have been removed, as they no longer exist, or are no longer updated.

Online things

- Package search
- Options search
- A list of channels, and when they were last updated

Development tooling

- A `.drv` file parser in JS
- rnix, a Nix (language) parser in Rust

(Reference) documentation

- Nix manual
- A *complete* list of Nix operators (the list in the official manual is incomplete)
- nixpkgs manual
- NixOS manual
- Official NixOS wiki

Tutorials and examples

- Step-by-step walkthrough of the Nix language

- A shorter primer of the Nix language (probably a better option if you already know another programming language)
- Nix pills (a series of articles about different aspects of Nix; ongoing work on a compact edition can be found here)
- Example configurations
- Hardware configurations (includes configurations for dealing with quirks on specific hardware and models)

Community and support

- The Nix forum
- The Matrix space

Miscellaneous notes and troubleshooting

- My Nix and NixOS notes: see the rest of the articles in this chapter!
- My Hydra setup notes

Proprietary AMD drivers (fglrx) causing fatal error in i387.h

> This article was originally published at https://gist.github.com/joepie91/ce9267788fdcb37f5941be5a04fcdd0f. It should no longer be applicable, but is preserved here in case a similar issue reoccurs in the future.

If you get this error:

```
/tmp/nix-build-ati-drivers-15.7-4.4.18.drv-0/common/lib/modules/fglrx/build_mod/2.6.x/firegl_public.c:194:22: fatal error: asm/i387.h: No such file or directory
```

... it's because the drivers are not compatible with your current kernel version. I've worked around it by adding this to my `configuration.nix`, to switch to a 4.1 kernel:

```
{
  # ...
  boot.kernelPackages = plgs.linuxPackages_4_1;
  # ...
}
```

Installing a few packages from `master`

> This article was originally published at https://gist.github.com/joepie91/ce9267788fdcb37f5941be5a04fcdd0f.

> You probably want to install from `unstable` instead of `master`, and you probably want to do it differently than described here (eg. importing from URL or specifying it as a Flake). This documentation is kept here for posterity, as it is still helpful to understand how to import a *local* copy of a nixpkgs into your configuration.

1. `git clone https://github.com/NixOS/nixpkgs.git /etc/nixos/nixpkgs-master`
2. Edit your `/etc/nixos/configuration.nix` like this:

```
{ config, pkgs, ... }:

let
  nixpkgsMaster = import ./nixpkgs-master {};

  stablePackages = with pkgs; [
    # This is where your packages from stable nixpkgs go
  ];

  masterPackages = with nixpkgsMaster; [
    # This is where your packages from `master` go
    nodejs-6_x
  ];
in {
  # This is where your normal config goes, we've just added a `let` block

  environment = {
    # ...

    systemPackages = stablePackages ++ masterPackages;
  };

  # ...
}
```

GRUB2 on UEFI

> This article was originally published at https://gist.github.com/joepie91/ce9267788fdcb37f5941be5a04fcdd0f.

> These instructions are most likely outdated. They are kept here for posterity.

This works fine. You need your `boot` section configured like this:

```
{
  # ...
  boot = {
    loader = {
      gummiboot.enable = false;

      efi = {
        canTouchEfiVariables = true;
      };

      grub = {
        enable = true;
        device = "nodev";
        version = 2;
        efiSupport = true;
      };
    };
  };
  # ...
}
```

Unblock ports in the firewall on NixOS

> This article was originally published at https://gist.github.com/joepie91/ce9267788fdcb37f5941be5a04fcdd0f.

The firewall is enabled by default. This is how you open a port:

```
{
  # ...
  networking = {
    # ...

    firewall = {
      allowedTCPPorts = [ 24800 ];
    };
  };
  # ...
}
```

Guake doesn't start because of a GConf issue

> This article was originally published at https://gist.github.com/joepie91/ce9267788fdcb37f5941be5a04fcdd0f. It may or may not still be relevant.

> *From nixpkgs:* GNOME's GConf implements a system-wide registry (like on Windows) that applications can use to store and retrieve internal configuration data. That concept is inherently impure, and it's very hard to support on NixOS.

1. Follow the instructions here.
2. Run the following to set up the GConf schema for Guake: `gconftool-2 --install-schema-file $(readlink $(which guake) | grep -Eo '\/nix\/store\/[^\/]+\/')"share/gconf/schemas/guake.schemas"`. This will not work if you have changed your Nix store path - in that case, modify the command accordingly.

You may need to re-login to make the changes apply.

FFMpeg support in youtube-dl

> This article was originally published at https://gist.github.com/joepie91/ce9267788fdcb37f5941be5a04fcdd0f. It may no longer be necessary.

Based on this post:

```
{
  # ...
  stablePackages = with pkgs; [
    # ...
    (python35Packages.youtube-dl.override {
      ffmpeg = ffmpeg-full;
    })
    # ...
  ];
  # ...
}
```

(To understand what `stablePackages` is here, see this entry.)

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, moreso 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://cryto.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://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 tp 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.

gparted doesn't let me create a btrfs partition!

If the `btrfs` option in GParted's partition type list is grayed out, add `btrfs-progs` to your `system.environmentPackages` and rebuild your system. After restarting GParted, the option should be available.

Dealing with Bookstack breakage in a NixOS 25.11 upgrade

In 25.11, the NixOS service module for Bookstack unfortunately saw a number of breaking changes that were never added to the release notes.

Here's what you need to fix to get it to work again:

- You need to add a `base64:` prefix to the key in your app key file. The old key generation stored it without this prefix, and then inserted it at runtime. This runtime fix has been removed in a recent refactor of the service module, and so old key files no longer work in 25.11. To fix it, prepend the content in your `APP_KEY_FILE` with `base64:` yourself. Don't forget to deploy your secrets again if you're using a deployment tool, and restart `phpfpm-bookstack` to have the change take effect. See also this thread.
- You need to configure MySQL (or more likely, MariaDB) yourself, the module no longer does this for you. This may not be necessary if you're upgrading on the same machine, but it'll probably bite you if you ever move to a different machine, so I'd recommend just adding it to your configuration anyway. See the example below.
- You need to specify most of the application settings yourself, as most of the configuration defaults have disappeared. See the example below.

```
□□services.bookstack = {
□□□# ...
□□□# database = { createLocally = true; }; # No longer works in NixOS 25.11
□□□settings = {
□□□□APP_KEY_FILE = "/var/lib/bookstack/app-key";
□□□□DB_HOST = "localhost";
□□□□DB_PORT = 3306;
□□□□DB_DATABASE = "bookstack";
□□□□DB_USERNAME = "bookstack";
□□□□SESSION_SECURE_COOKIE = true; # Only set to true when you're serving over TLS (which I hope
you are!)
□□□};
            # ...
```

```
  };

  # For Bookstack
  services.mysql = let
    bsConfig = config.services.bookstack.settings;
  in {
    enable = true;
    package = pkgs.mariadb;
    ensureDatabases = [ bsConfig.DB_DATABASE ];
    ensureUsers = [{
      name = bsConfig.DB_USERNAME;
      ensurePermissions = {
        "${bsConfig.DB_DATABASE}.*" = "ALL PRIVILEGES";
      };
    }];
  };
```

The above example **does not include** the mail settings, which have *also* been removed from the defaults. Have a look at the last pre-breakage version of the module if you need to replicate these defaults.