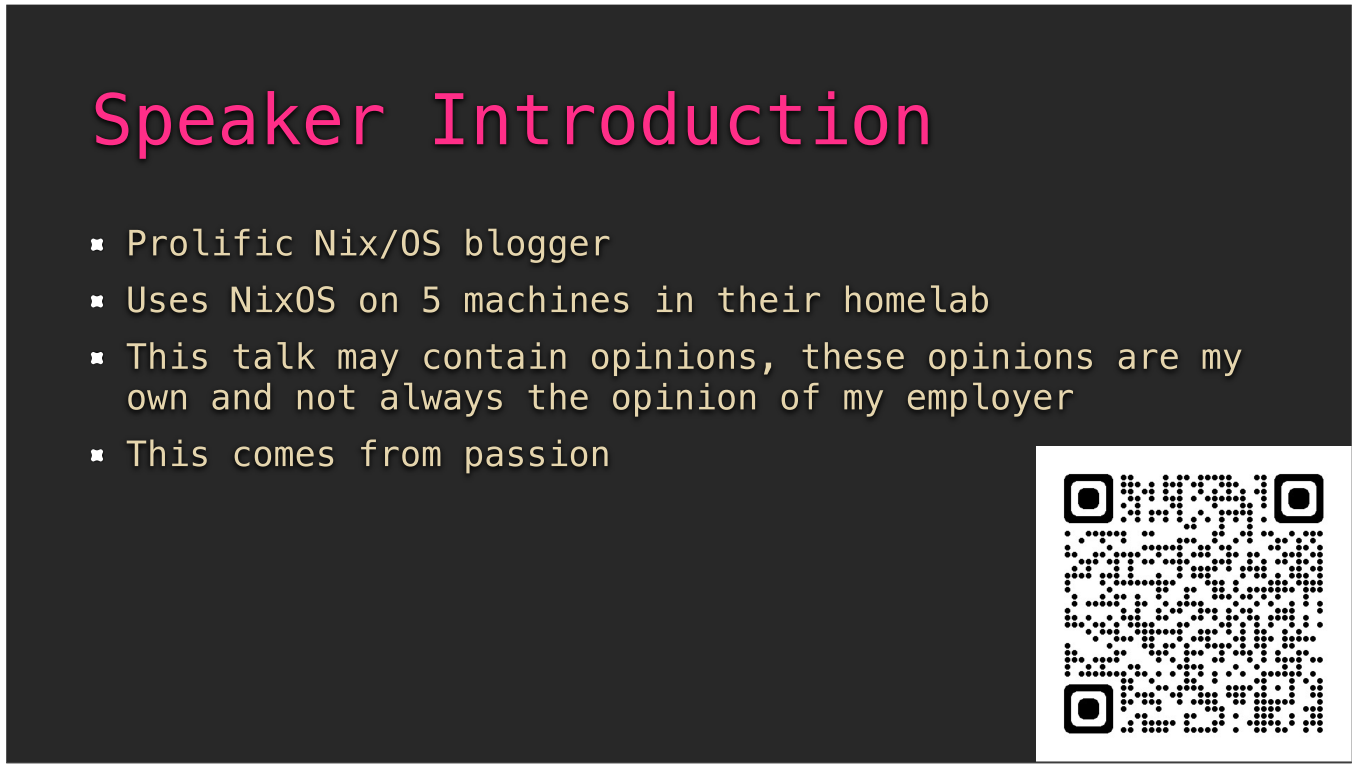
How Nix and NixOS Get So Close to Perfect

Xe
https://christine.website
PackagingCon 2021

Hello, my name is Xe and today we're gonna talk about NixOS. NixOS is one of my favorite tools for desktop and server linux; however it has a LOT of rough edges that detract from it being the perfect tool it could be. I'm gonna go over a lot of what makes it great, what detracts from that and what I'd love to see instead.

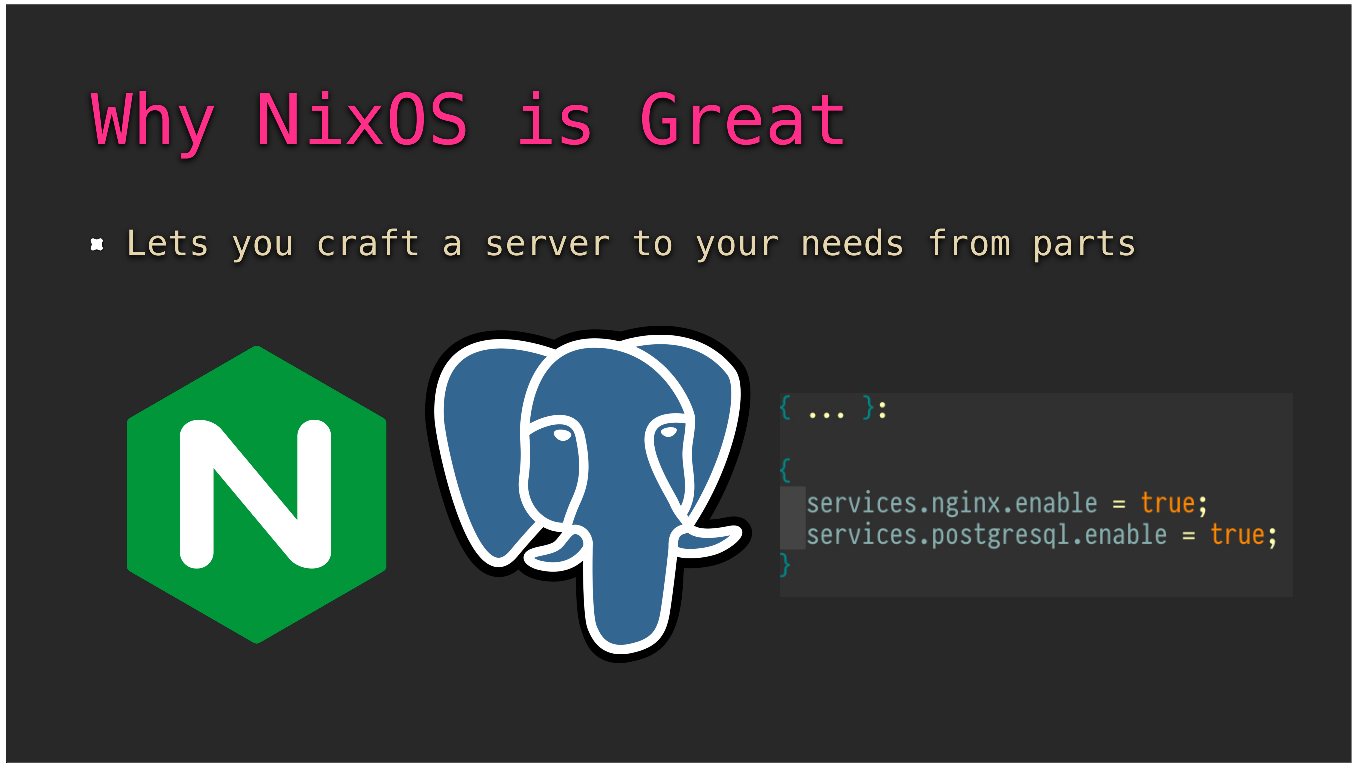


Speaker Introduction

- Prolific Nix/OS blogger
- Uses NixOS on 5 machines in their homelab
- This talk may contain opinions, these opinions are my own and not always the opinion of my employer
- This comes from passion

As I said, I'm Xe. I write a lot about Nix and NixOS and use it a lot personally. This presentation may contain opinions. These opinions are my own and not the opinion of my employer. I do not intend ill will to any people or their work in this presentation. I want this to be better because I am passionate about these tools and believe that it's the best way to do things.

The QR code on the slide links to my website christine.website.



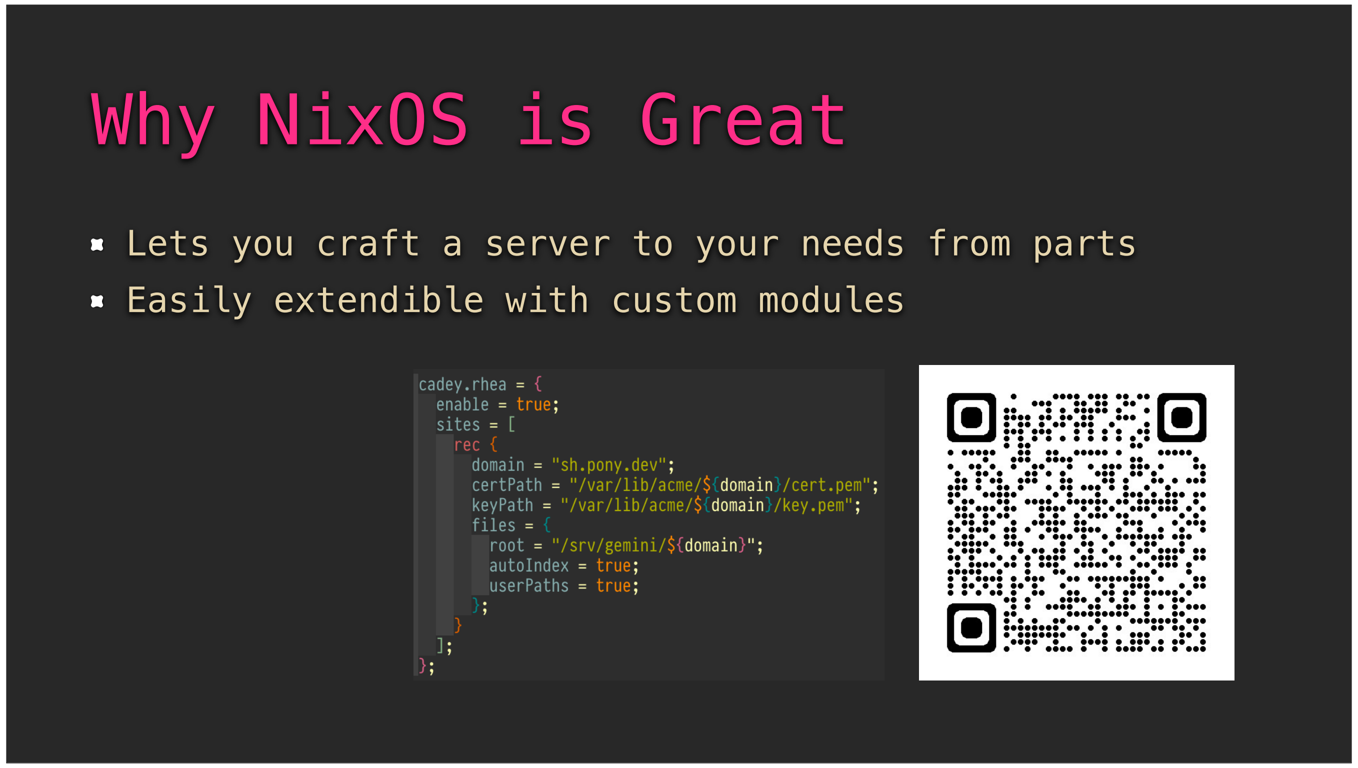
Why NixOS is Great

- Lets you craft a server to your needs from parts

```
{ ... }:

{
  services.nginx.enable = true;
  services.postgresql.enable = true;
}
```

NixOS is great because it lets you pick from cookie-cutter templates to make a server do _exactly_ what you want. It builds on the shoulders of giants to make it easy and effective to make your servers built to purpose.

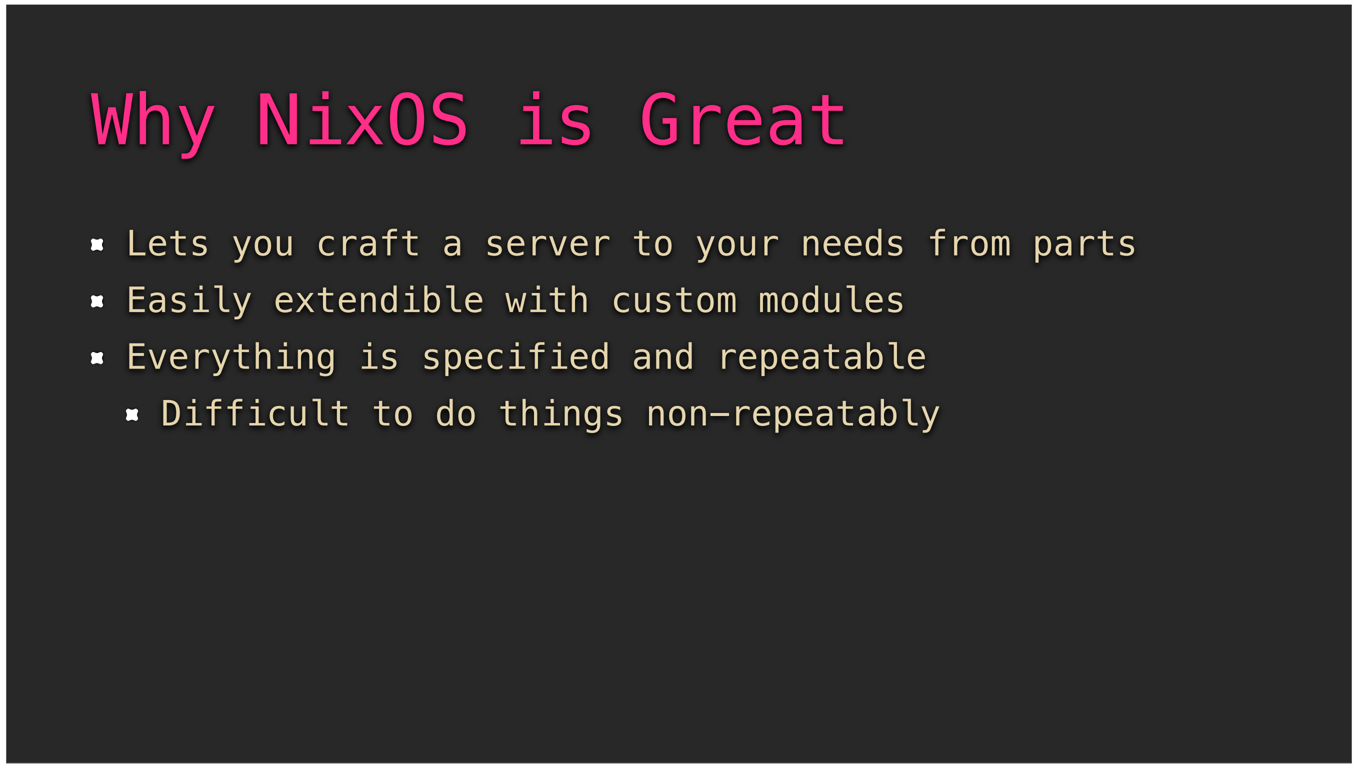


Why NixOS is Great

- Lets you craft a server to your needs from parts
- Easily extendible with custom modules

```
cadey.rhea = {
  enable = true;
  sites = [
    rec {
      domain = "sh.pony.dev";
      certPath = "/var/lib/acme/${domain}/cert.pem";
      keyPath = "/var/lib/acme/${domain}/key.pem";
      files = {
        root = "/srv/gemini/${domain}";
        autoIndex = true;
        userPaths = true;
      };
    }
  ];
};
```

This modularity also extends to your own custom modules. The QR code explains how to write your own modules for your own services.

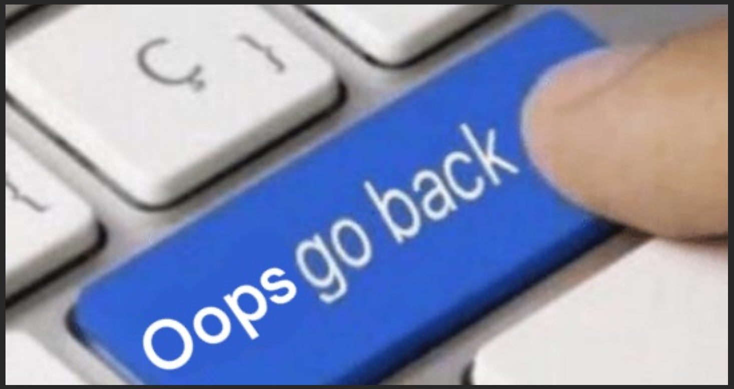

Why NixOS is Great

- Lets you craft a server to your needs from parts
- Easily extendible with custom modules
- Everything is specified and repeatable
  - Difficult to do things non-repeatably

Finally, one of the best parts of it is that NixOS makes it *hard* to do something the wrong way. You can't just hack up a systemd unit on the fly. You need to do it in the configuration management the right way. This makes it easier for you to ensure servers aren't being tampered with without going through a review process.

Why NixOS is Great

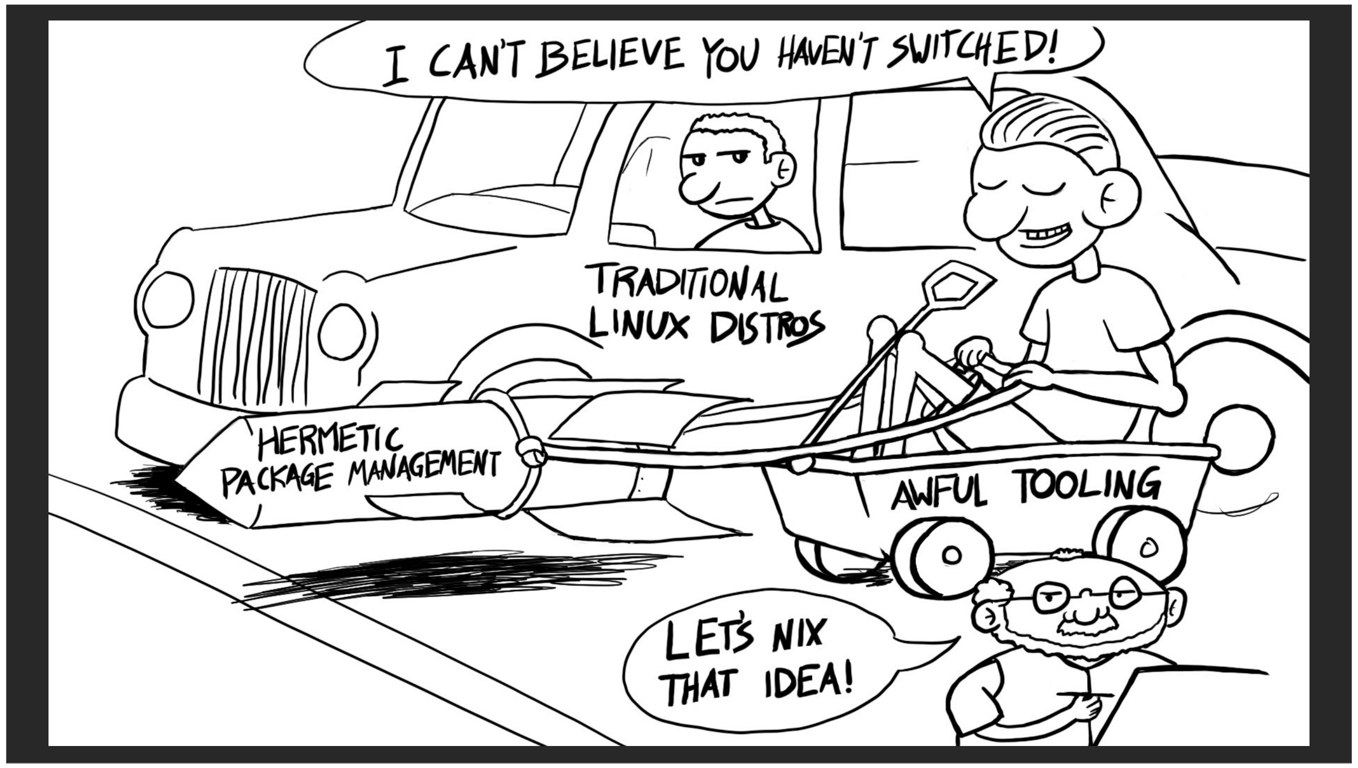
- NixOS lets you undo config changes



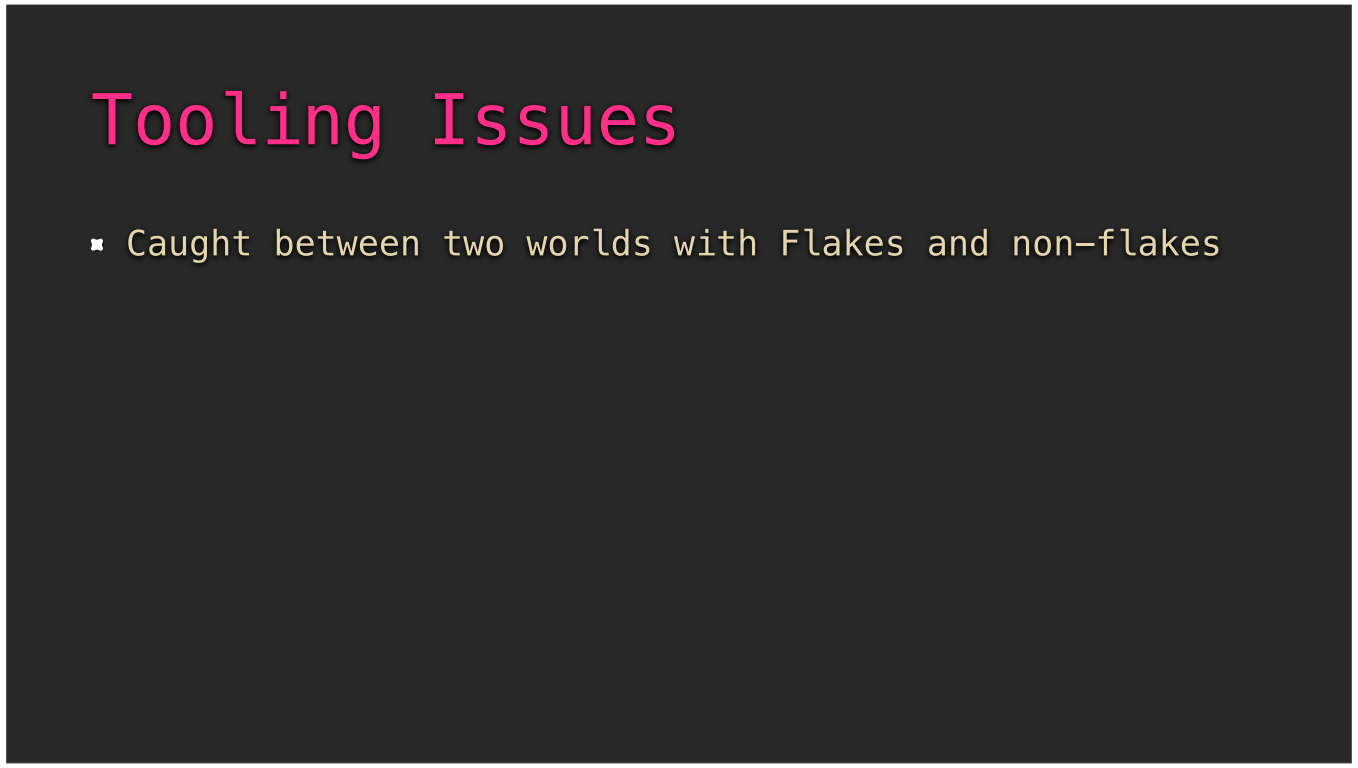
The biggest thing in my book is that NixOS lets you *undo* configuration changes. At worst you'll need to reboot into an older generation; but in general if you mess something up, you can go back. This is a life-saver.

<Mara> What's the catch?

You may be thinking "so, what's the catch?"


I CAN'T BELIEVE YOU HAVEN'T SWITCHED!
TRADITIONAL LINUX DISTROS
HERMETIC PACKAGE MANAGEMENT
AWFUL TOOLING
LET'S NIX THAT IDEA!

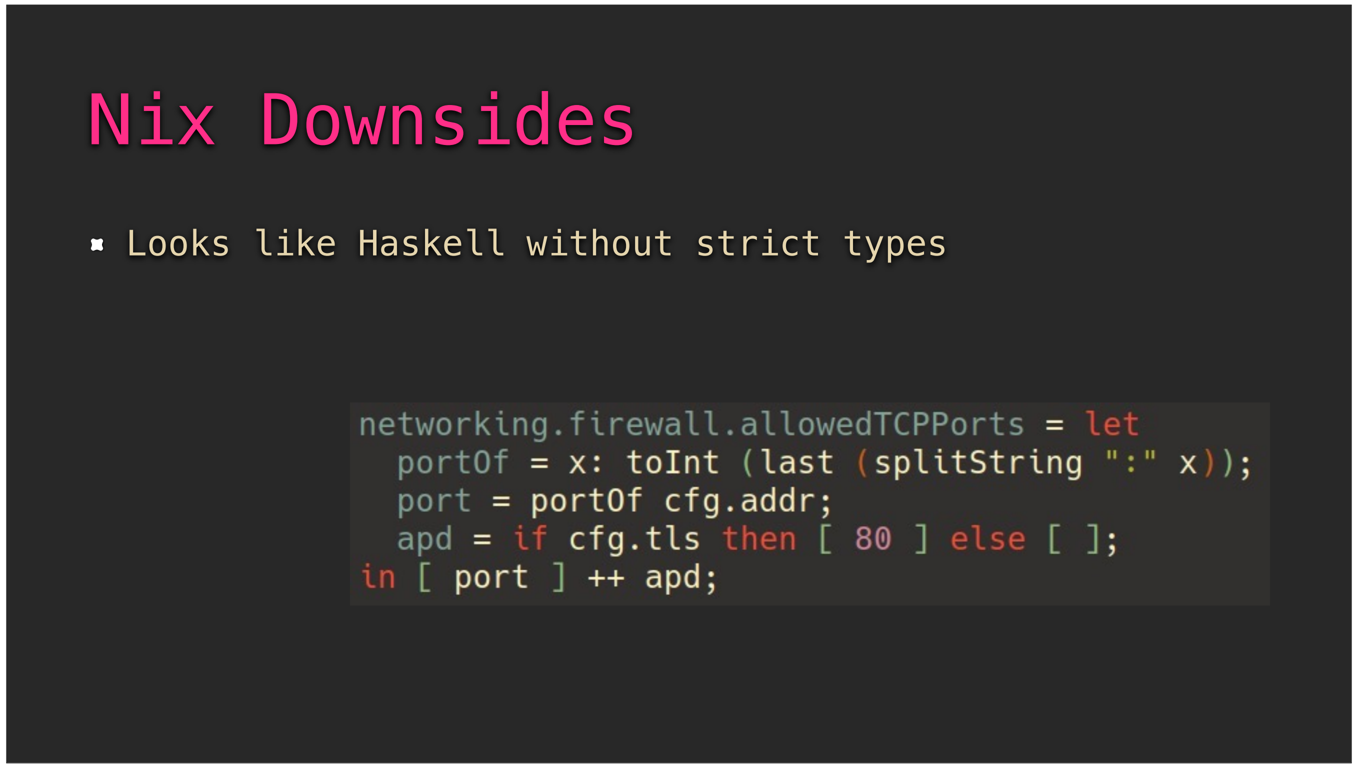
Unfortunately the biggest catches are the tooling and the documentation, two of the most important parts of this stack.



Tooling Issues

- Caught between two worlds with Flakes and non-flakes

Right now the Nix universe is in the middle of switching to a new hermetic view of the world they are calling Flakes. I haven't been able to understand much of the jargon involved, but it's soft-splitting the community between people that use flakes and people that don't use flakes.

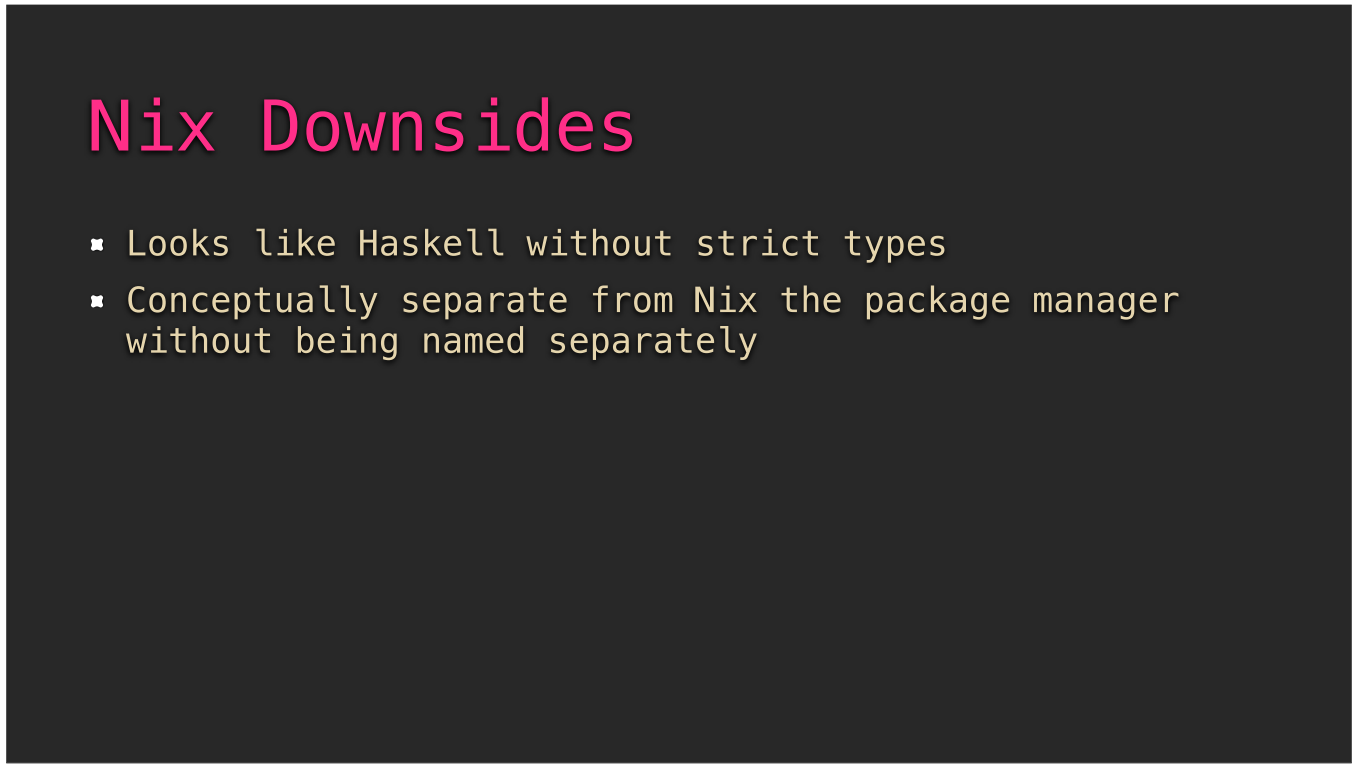


Nix Downsides

- Looks like Haskell without strict types

```
networking.firewall.allowedTCPPorts = let
  portOf = x: toInt (last (splitString ":" x));
  port = portOf cfg.addr;
  apd = if cfg.tls then [ 80 ] else [ ];
in [ port ] ++ apd;
```

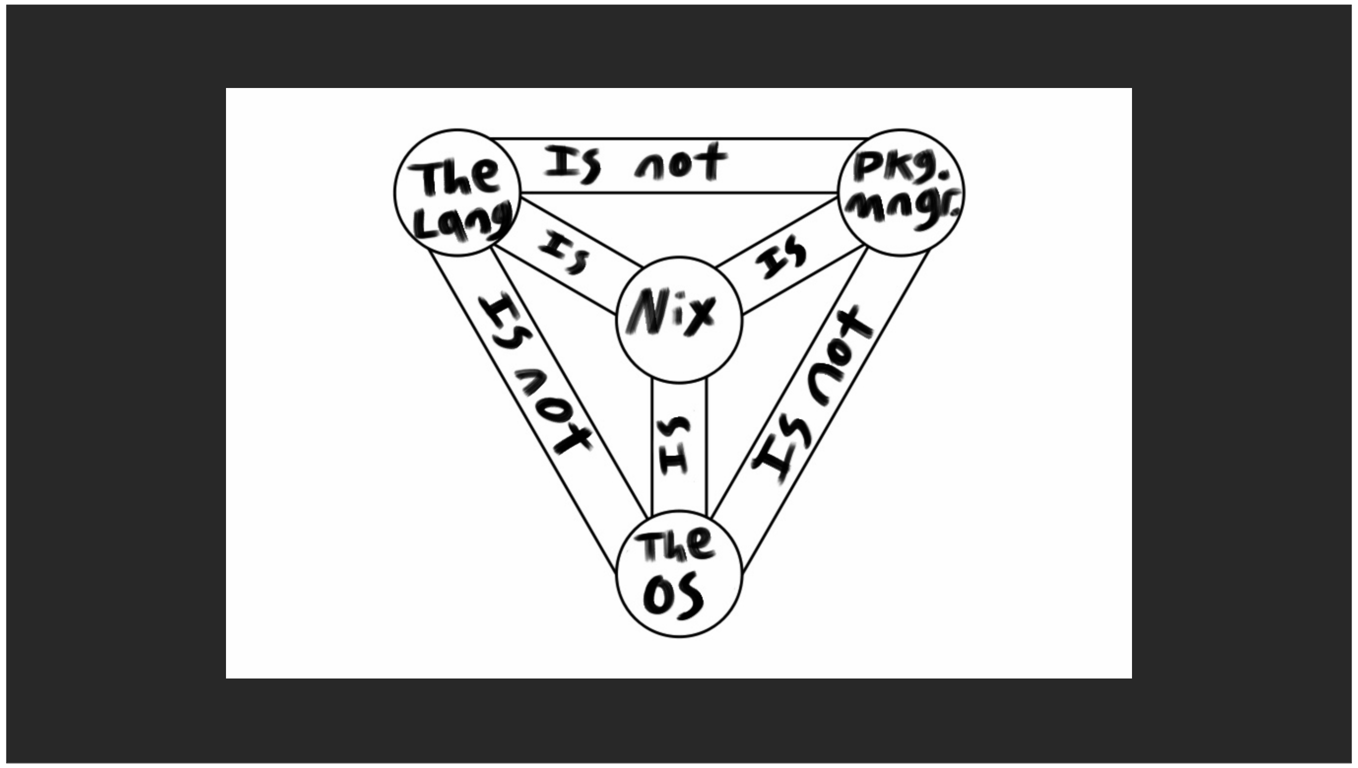
This is a little snippet of Nix code that derives a firewall port from a host:port address that some programs use. This bit of code has a large comment on it in the corresponding file explaining what it does, but if you aren't familiar with Haskell or other functional languages it may be hard to understand.



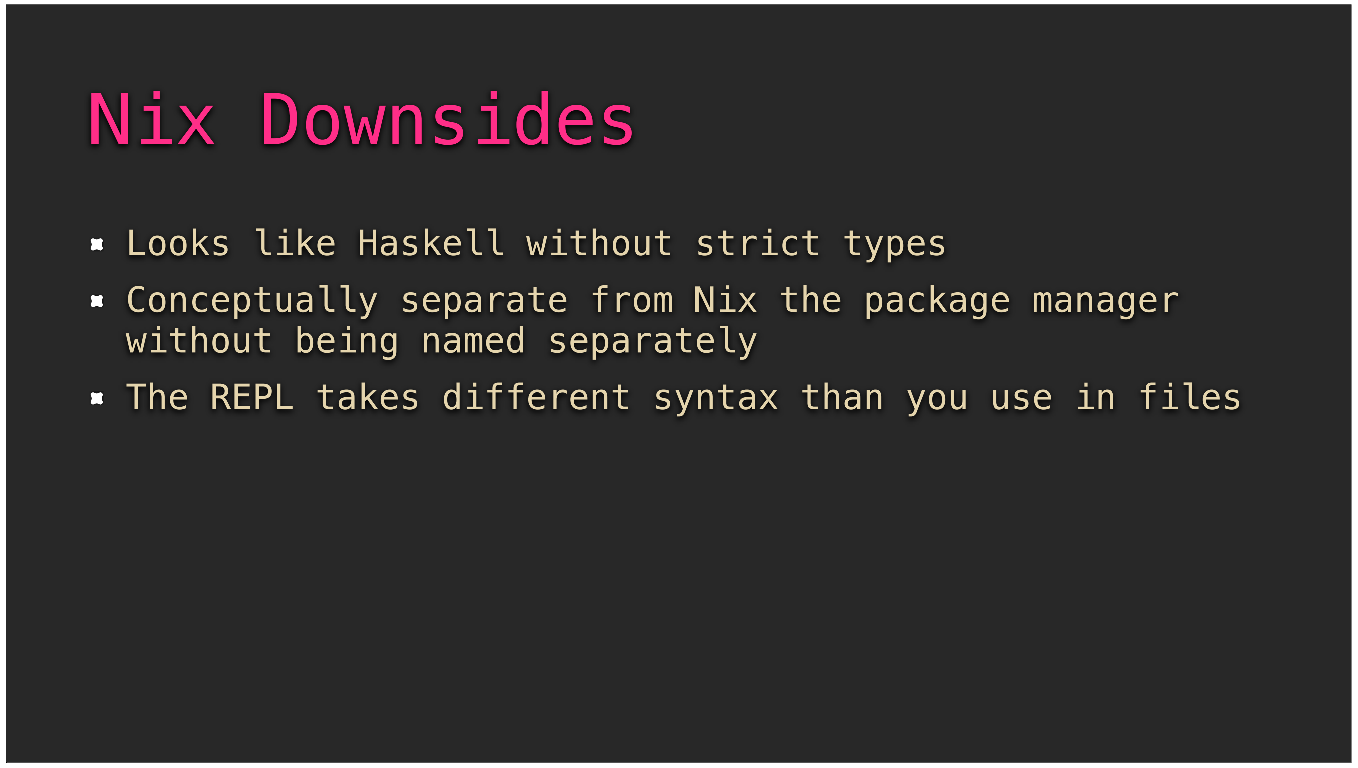
Nix Downsides

- Looks like Haskell without strict types
- Conceptually separate from Nix the package manager without being named separately

The relationship between each of the main parts of the Nix stack is confusing at first because of the number of conflicting names for different things. It kinda looks like this:



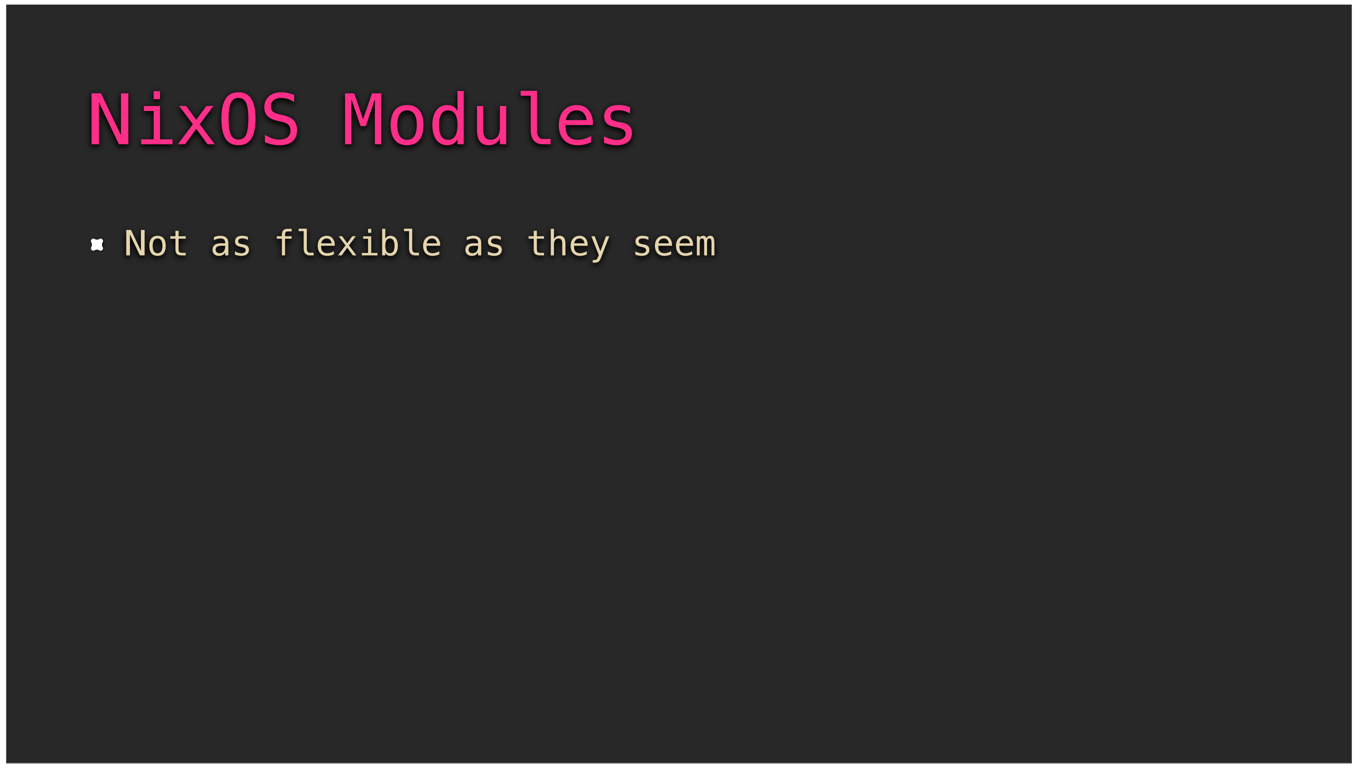
The relationship between Nix the language, Nix the package manager and NixOS is similar to the holy trinity or javascript equality rules.



Nix Downsides

- Looks like Haskell without strict types
- Conceptually separate from Nix the package manager without being named separately
- The REPL takes different syntax than you use in files

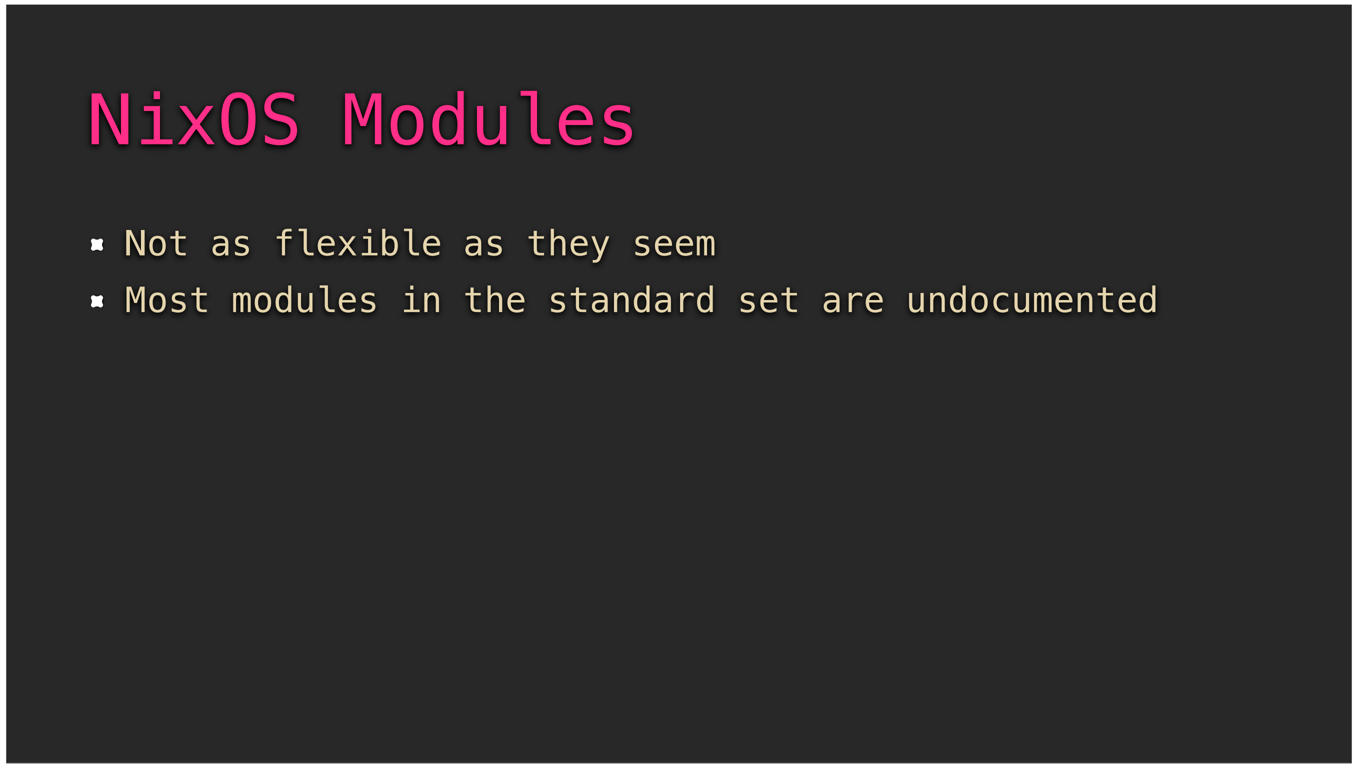
The REPL can take different syntax than is used in Nix files. This was probably done to allow for quicker hacking, but can be confusing since the expressions you hacked up in the REPL are not the same things that are allowed in Nix files.



NixOS Modules

- Not as flexible as they seem

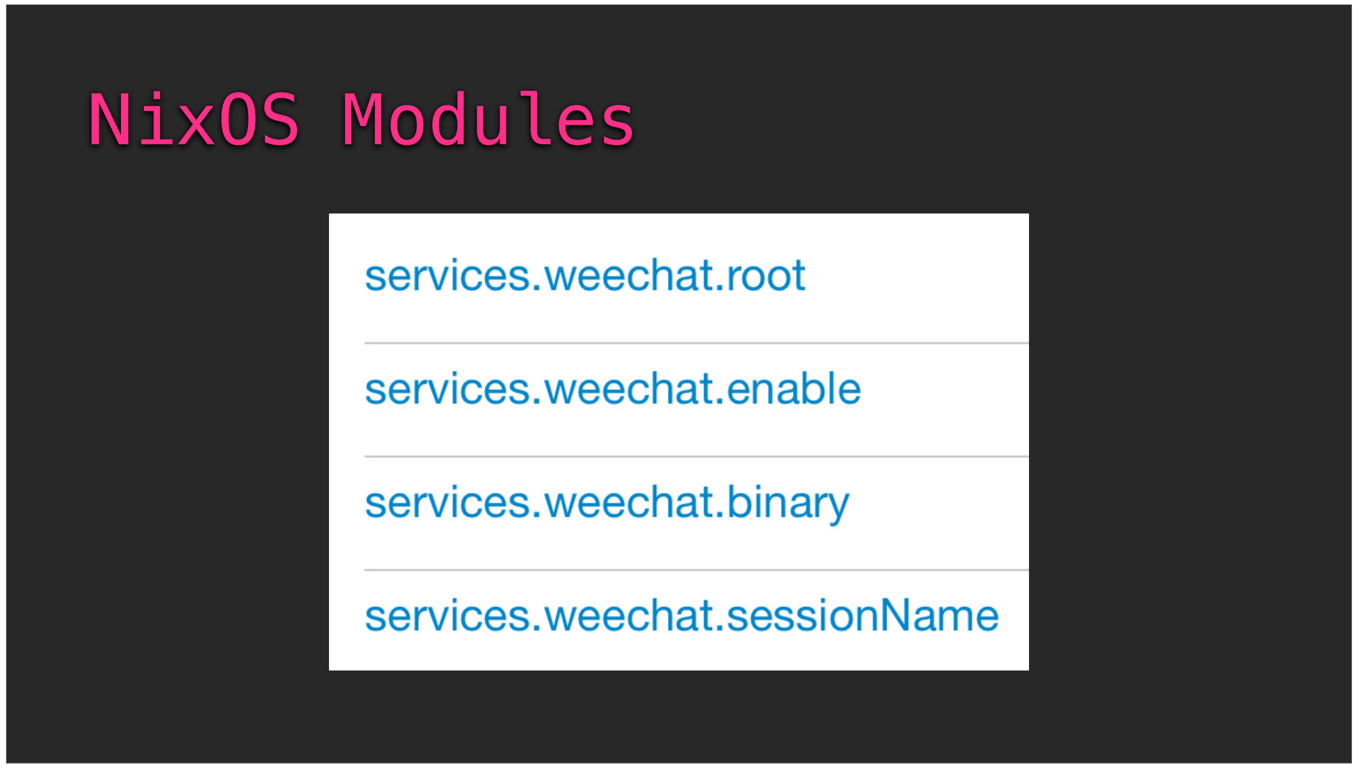
NixOS modules are a fantastic abstraction that let you piece together complicated software like it was nothing. NixOS modules are effectively templates for reaching a desired system state. This makes them a really good starting place, but they don't go all the way that I'd like to see them go.



NixOS Modules

- Not as flexible as they seem
- Most modules in the standard set are undocumented

Most of the modules in the standard set are not documented in the NixOS manual. This includes nginx. There is a search site that lets you query the list of options in the standard set, however this makes it hard to get started with something unfamiliar.



NixOS Modules

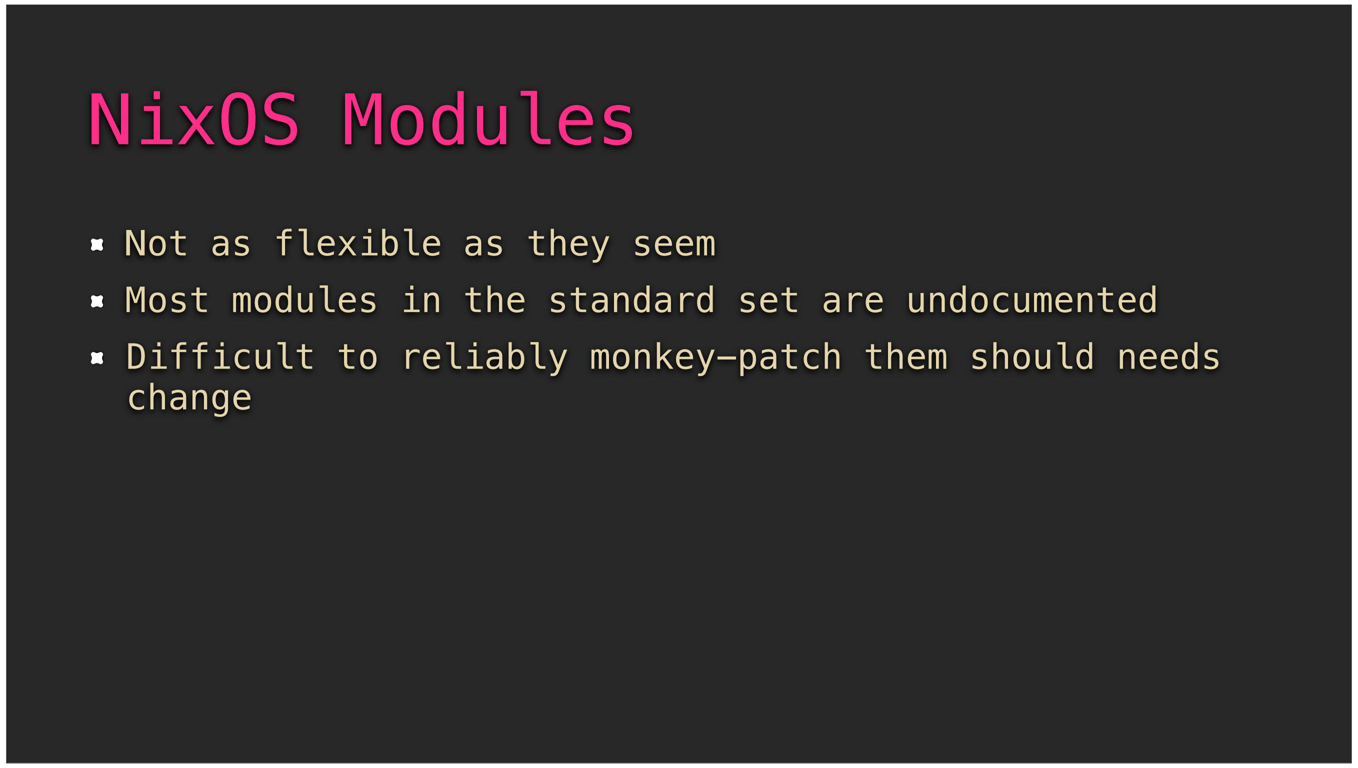
services.weechat.root

services.weechat.enable

services.weechat.binary

services.weechat.sessionName

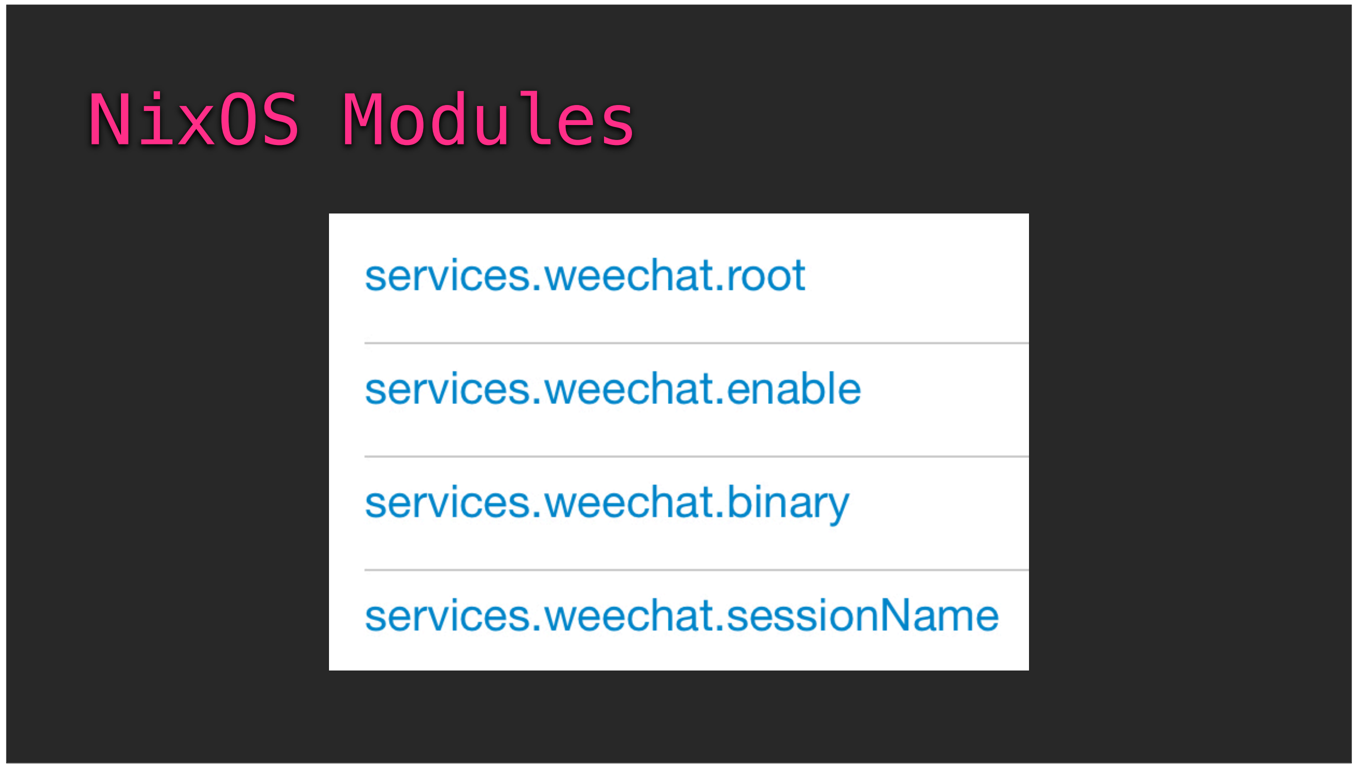
As an example of an obscure module, let's look at the WeeChat module. WeeChat is the main IRC client that I use. NixOS offers a module to have NixOS manage an installation of WeeChat for you so you can attach to it when it's relevant to you.



NixOS Modules

- Not as flexible as they seem
- Most modules in the standard set are undocumented
- Difficult to reliably monkey-patch them should needs change

NixOS modules go get you most of the way there, they will set up Postgres, nginx or other things; but they will only go that far. If you need to do fancy monkey-patching you will need to rely on the side effects of the module rather than changing the module itself. Most modules in the standard set are also undocumented, which can make it hard to figure out what to do when you are growing beyond them.



NixOS Modules

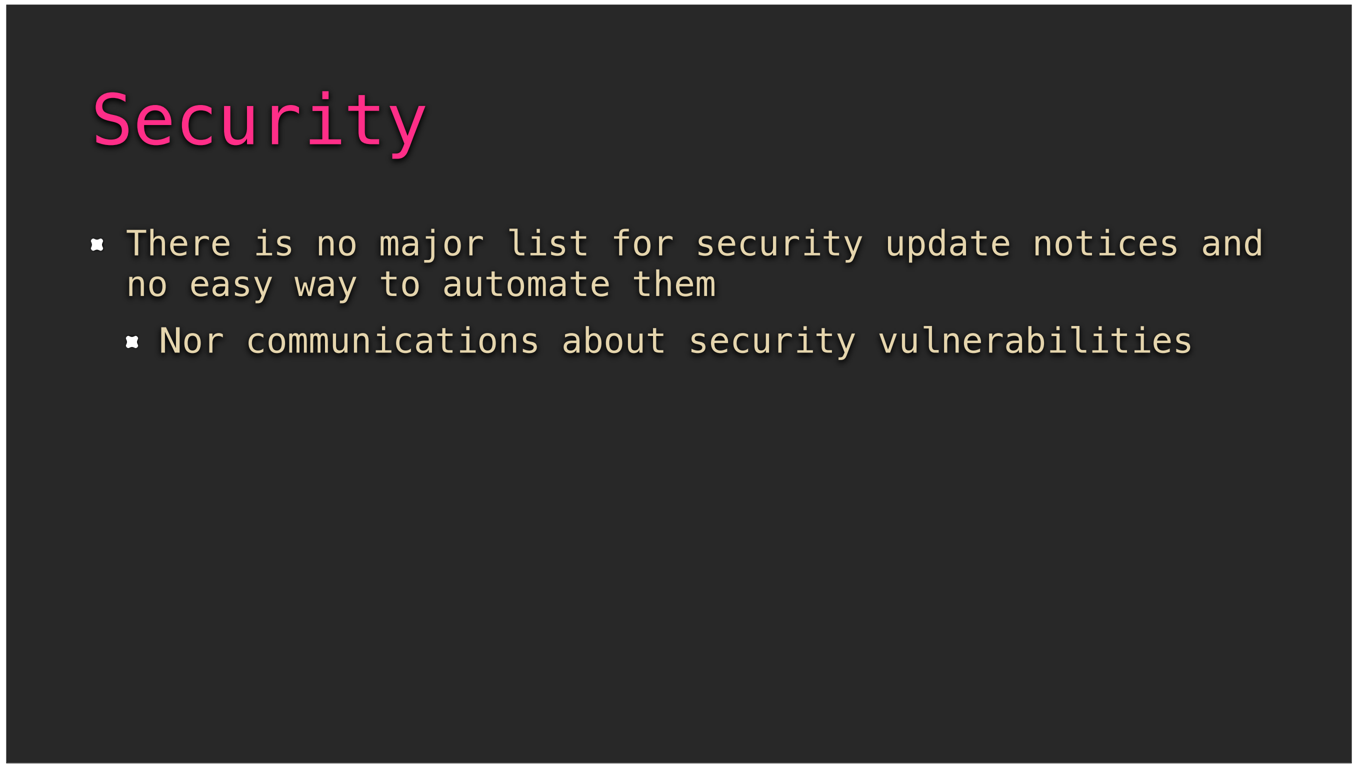
services.weechat.root

services.weechat.enable

services.weechat.binary

services.weechat.sessionName

Let's go back to this WeeChat module for a moment. It's great for being able to run _one_ session of WeeChat at once, but you'll have problems if you want to have multiple managed instances of WeeChat for something like a shared shellbox. You'd have to either write your own NixOS module to allow this or use NixOS containers (and at that point why not use Docker?).



Security

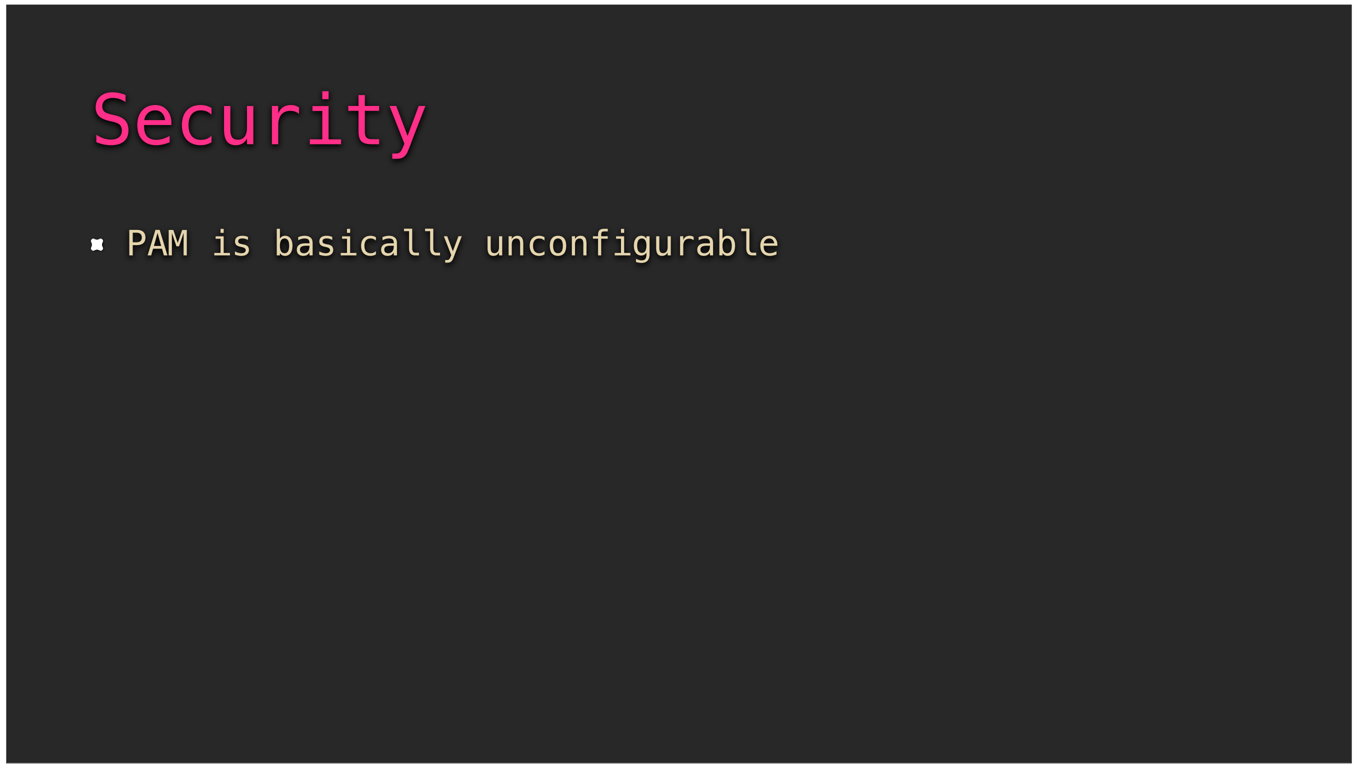
- There is no major list for security update notices and no easy way to automate them
  - Nor communications about security vulnerabilities

Something important for production workloads is the ability to answer the question "what do I do when a security update happens for something in my system?" NixOS does not make this easy. There is no formal security communications list (even though the package manager would make this a lot easier due to store paths being globally unique) and there are not regular communications about security vulnerabilities. This makes it hard to get NixOS systems validated for SOC2 or other certifications.

Security

- There are no hard written rules for backports

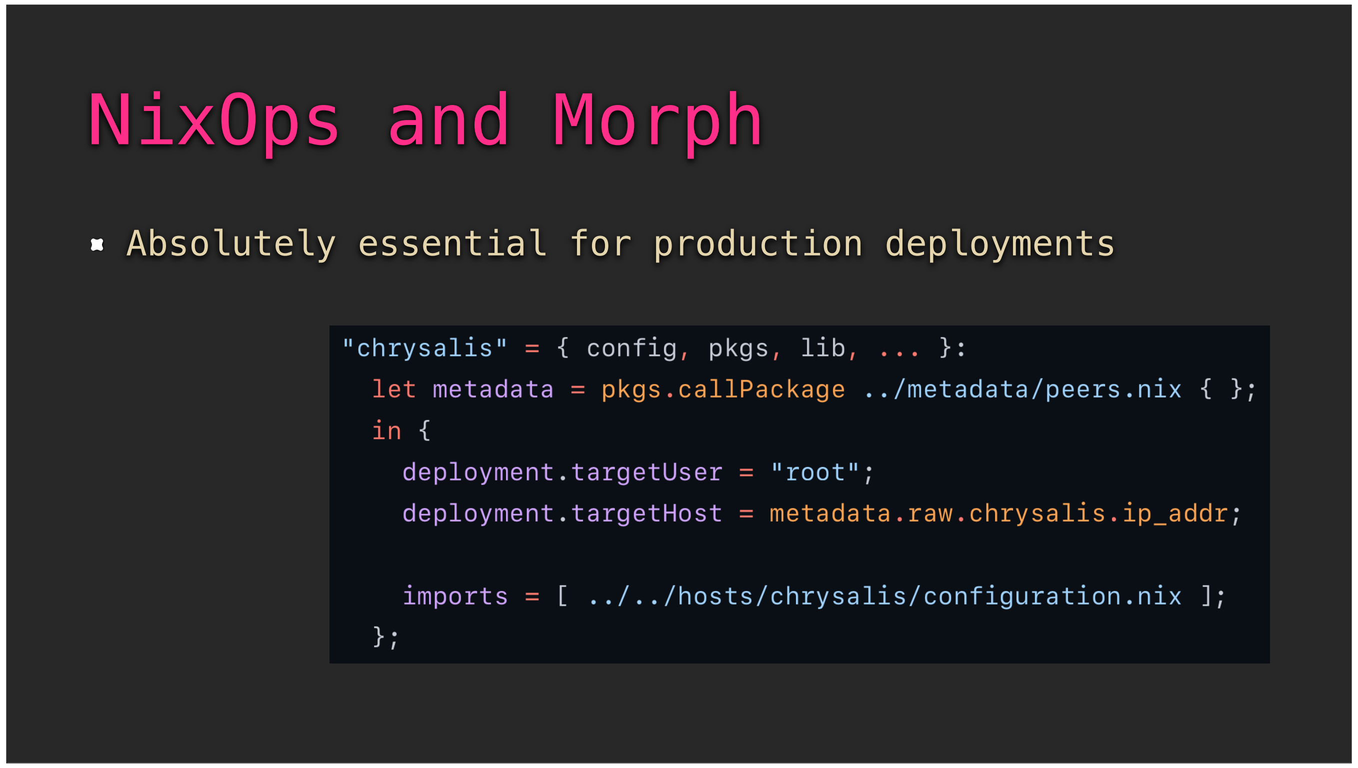
There are no hard written rules for when to back port fixes. While the system *is* stable and everything is up to date at release, it will bitrot as time advances to infinity. Most of the time people run NixOS unstable to work around this, but that may be less desirable for production workloads.



Security

- PAM is basically unconfigurable

If you want to configure PAM to do something special for your needs and it's not already in nixpkgs, you are doomed. Also see my earlier comments about NixOS modules not being flexible enough.



NixOps and Morph

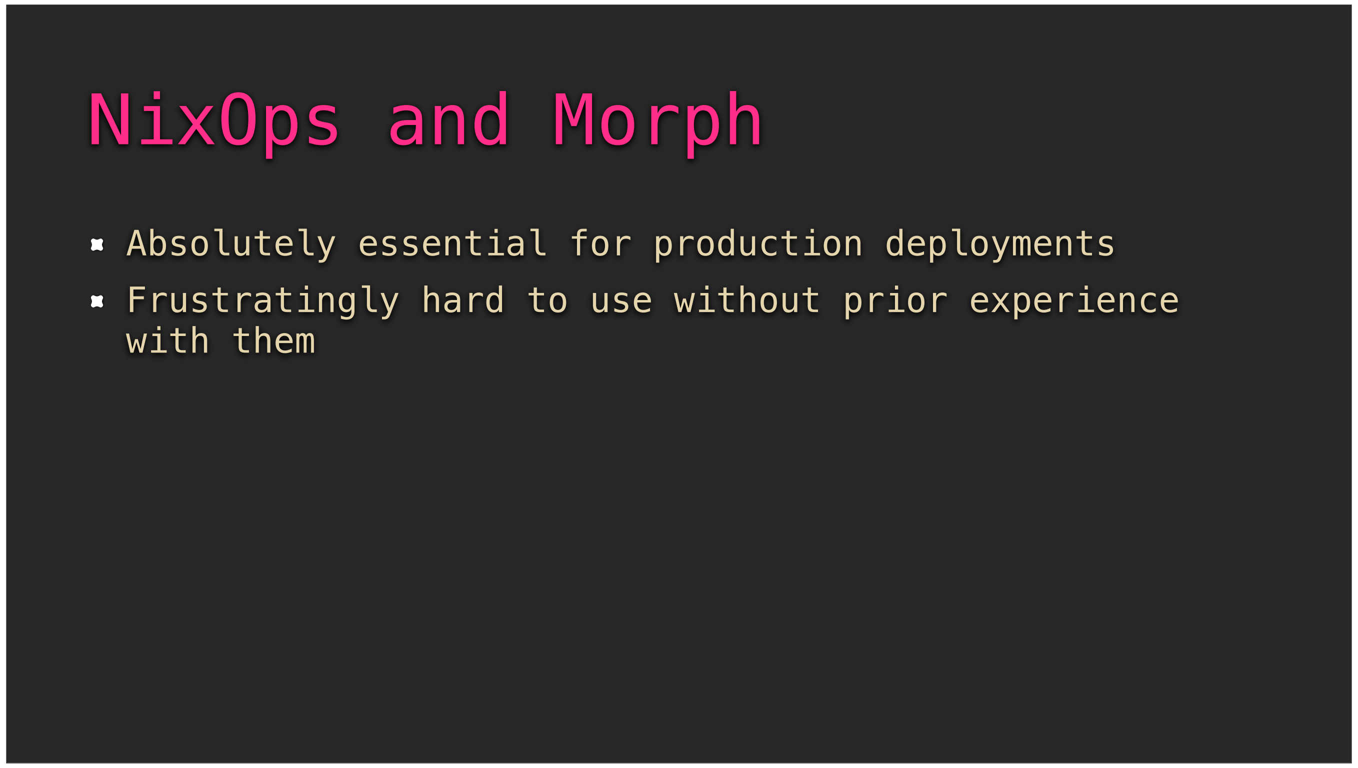
- Absolutely essential for production deployments

```
"chrysalis" = { config, pkgs, lib, ... }:
  let metadata = pkgs.callPackage ../metadata/peers.nix { };
  in {
    deployment.targetUser = "root";
    deployment.targetHost = metadata.raw.chrysalis.ip_addr;

    imports = [ ../../hosts/chrysalis/configuration.nix ];
  };
```

Tools like NixOps and Morph are very important for production rollouts. They allow you to make sure that you can describe the state of *your entire fleet* with the same NixOS module syntax that you use for a local machine. Unfortunately they are bit by the same poor documentation issues that plague other parts of the Nix stack. Here I have a little screenshot of part of my home network Morph configuration, this points Morph to the Mac Pro under my desk.

There are other options than using NixOS and morph, but these two are the target of my ire at the moment.

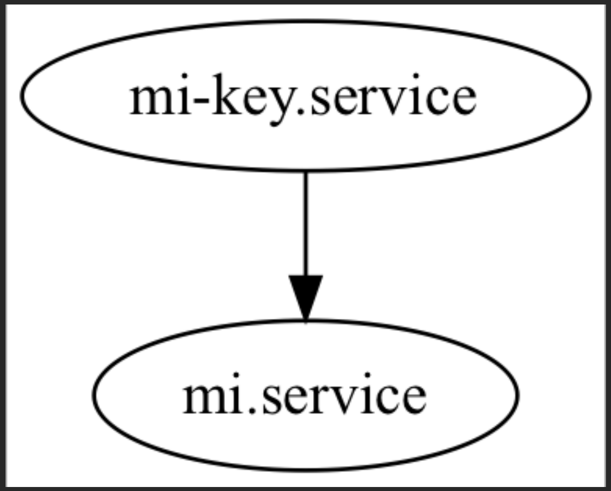


NixOps and Morph

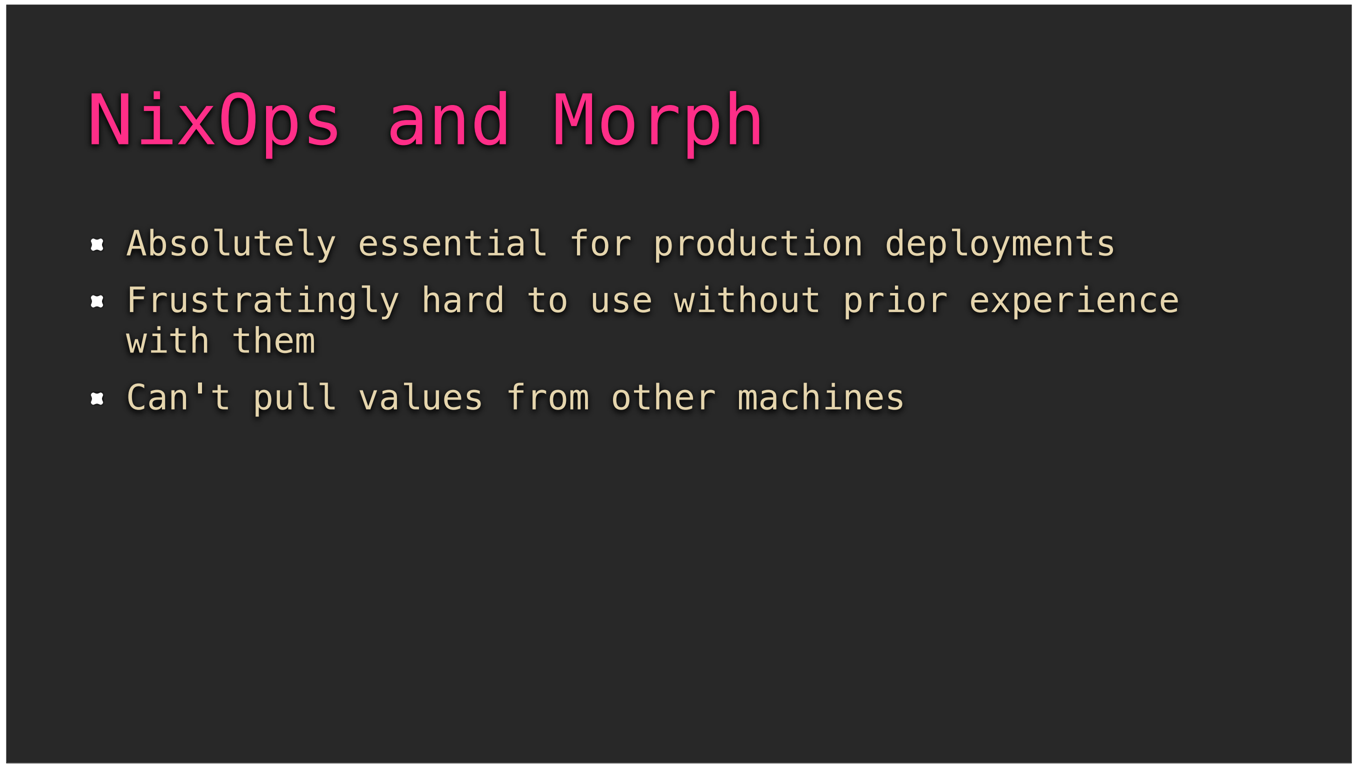
- Absolutely essential for production deployments
- Frustratingly hard to use without prior experience with them

Part of the reason I make all my NixOS configs public is to allow people to learn off of what I've figured out so that people have *something* to go off when figuring out how to do complicated things. Without them, I fear that people would have NO IDEA what to do.

NixOps and Morph



Let's talk about "keys", they are what NixOps and Morph call secret values. Normally you want to sequence a service start so that its secrets are ready before the service starts. You can do this by making sure that the -key.service correlating to the key name in your NixOps config is explicitly set before that, but the documentation doesn't make this very clear. I had to figure this out by searching GitHub for NixOS code.



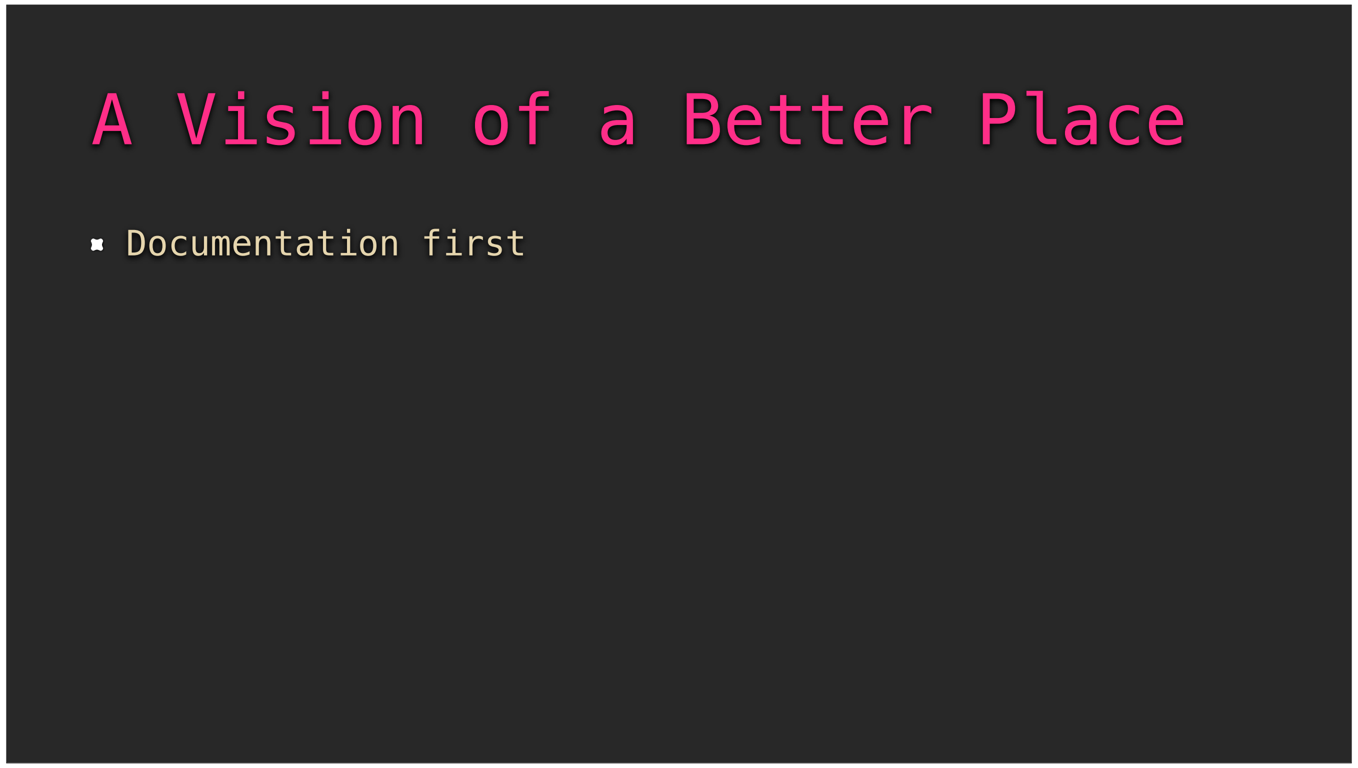
NixOps and Morph

- Absolutely essential for production deployments
- Frustratingly hard to use without prior experience with them
- Can't pull values from other machines

This may not come up very often, but sometimes it's very important to be able to make a cohesive configuration using data from other machines in your setup, such as dynamically obtained IP addresses from things like Tailscale or host SSH keys. It's nontrivial to pull this information from other machines like you can with Ansible or Chef.

A Vision of a Better Place

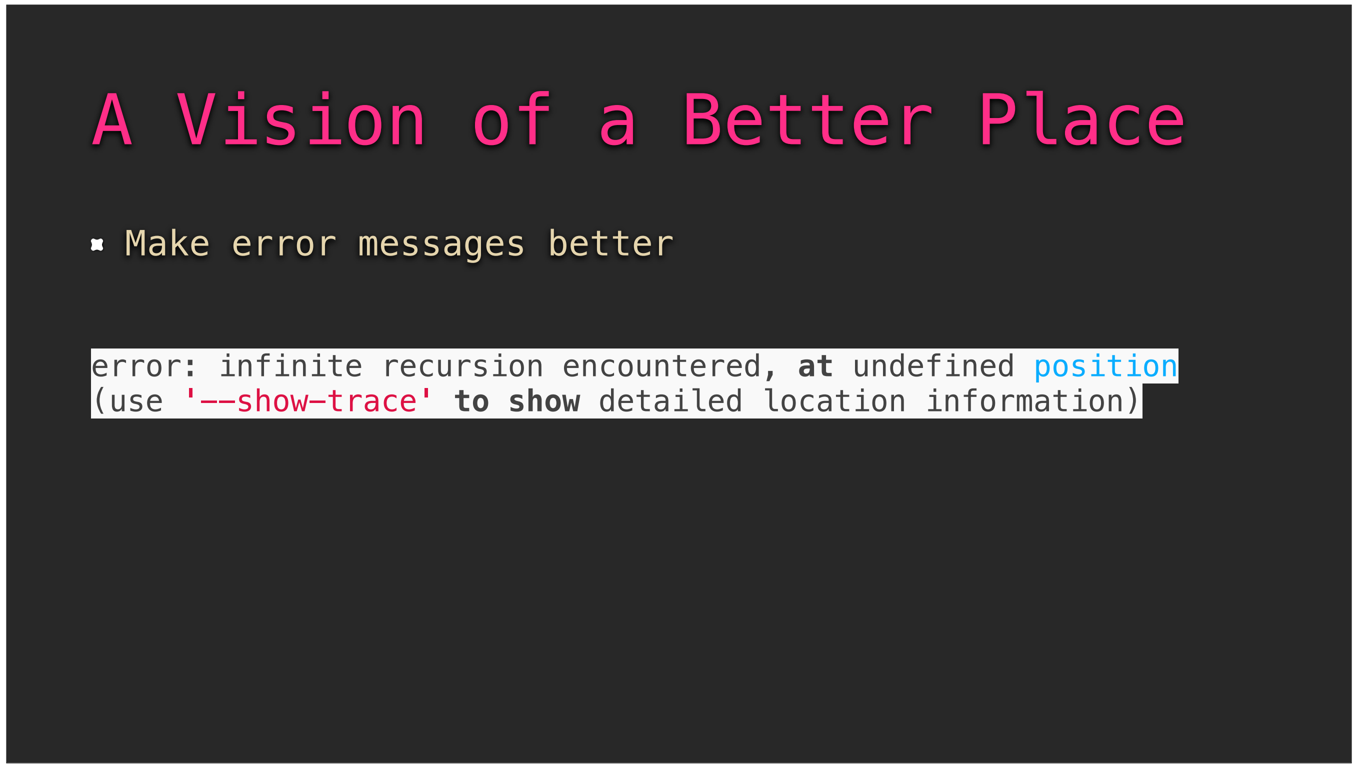
Finally, let's talk about a better world. These are my visions for what could be better, but I realize that some of them are difficult or impossible.



A Vision of a Better Place

- Documentation first

One of the most important things to work on when introducing new things like the Nix ecosystem is documentation. Good documentation is the difference between people getting absolutely lost and having a good enough experience to go on to become a full-time user. There should be nothing in the standard library without a section in the documentation on how and when to use it.

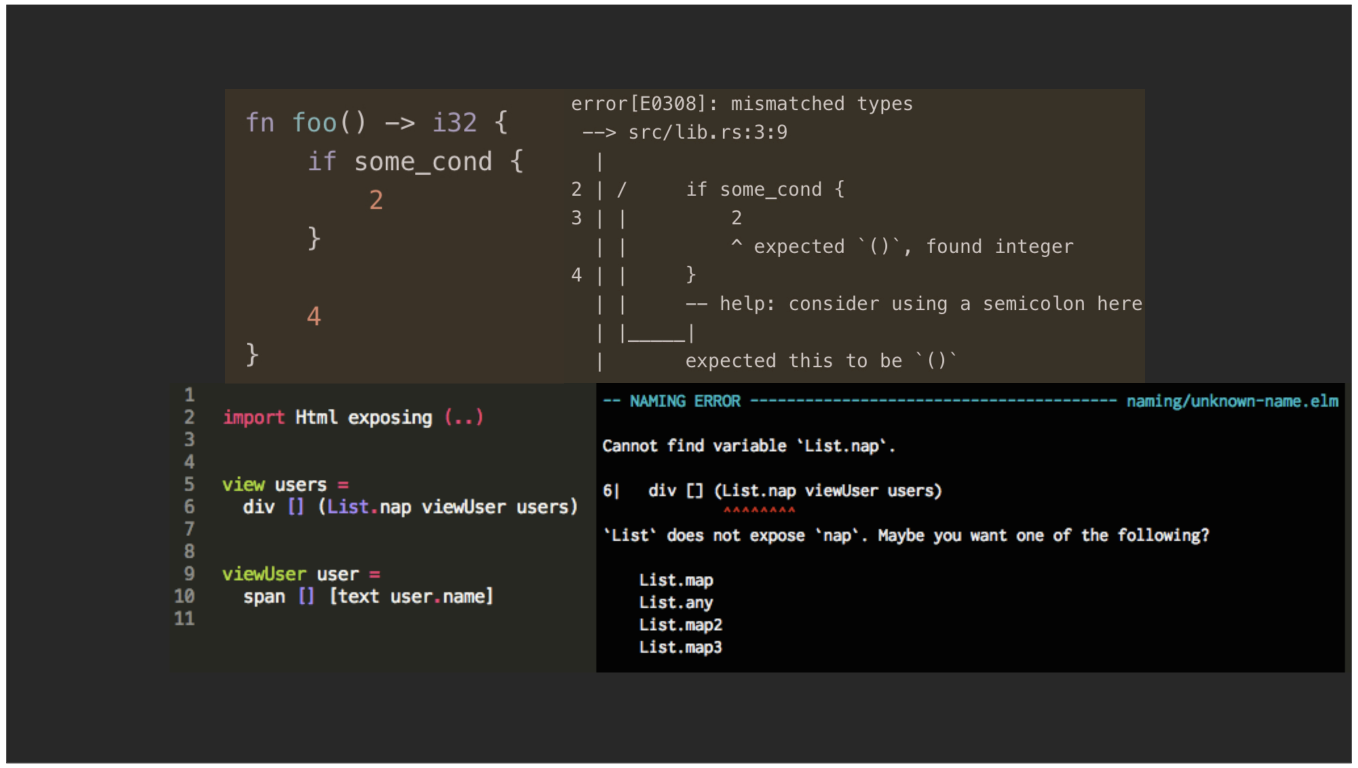


A Vision of a Better Place

- Make error messages better

```
error: infinite recursion encountered, at undefined position
(use '--show-trace' to show detailed location information)
```

Good error messages are the difference between enlightenment and confusion. This error is an infamous one within Nix (no, adding the --show-trace flag does not help). Consider projects like Elm and Rust and how they handle errors:



```rust
fn foo() -> i32 {
    if some_cond {
        2
    }

    4
}
```

```
error[E0308]: mismatched types
 --> src/lib.rs:3:9
  |
2 | /     if some_cond {
3 | |         2
  | |         ^ expected `()`, found integer
4 | |     }
  | |     -- help: consider using a semicolon here
  | |_____|
  |       expected this to be `()`
```

```elm
1
2  import Html exposing (..)
3
4
5  view users =
6    div [] (List.nap viewUser users)
7
8
9  viewUser user =
10   span [] [text user.name]
11
```

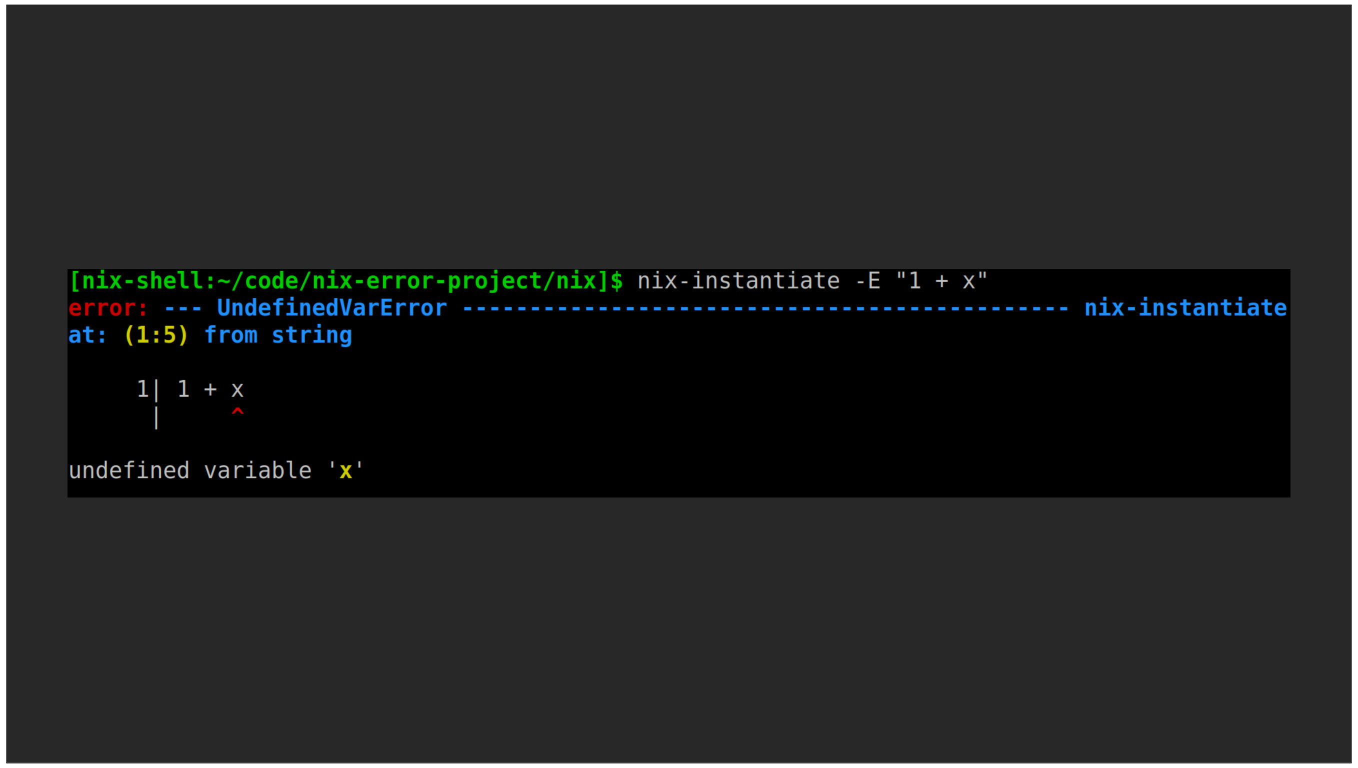
```
-- NAMING ERROR ------------------------------------- naming/unknown-name.elm

Cannot find variable `List.nap`.

6|   div [] (List.nap viewUser users)
             ^^^^^^^^
`List` does not expose `nap`. Maybe you want one of the following?

    List.map
    List.any
    List.map2
    List.map3
```

Here's an example of some error messages from Rust and Elm that more precisely explain what is going on and how to fix the problem. Rust even has ID numbers for the error messages that you can google to get more information about what's going on, which really helps to track down issues. There is work in progress to improve them, and it looks like this:

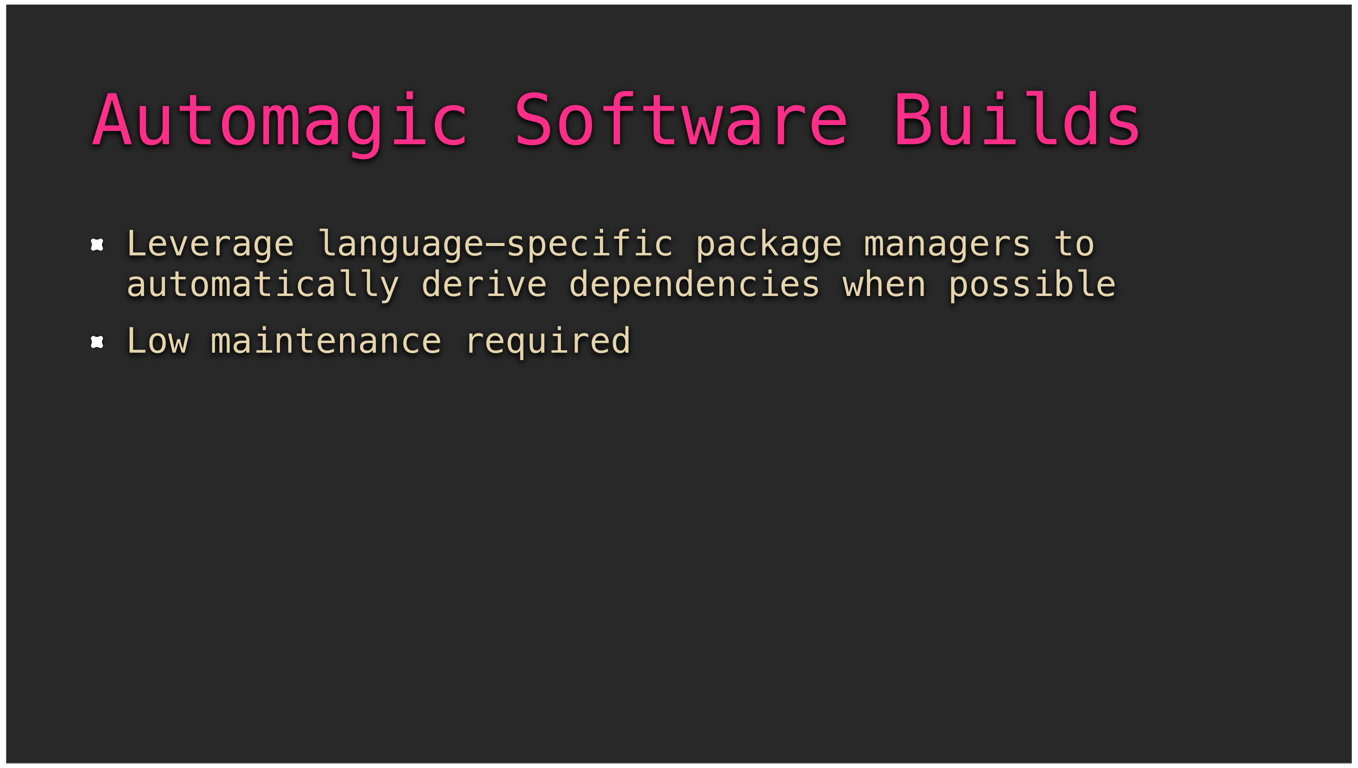


```
[nix-shell:~/code/nix-error-project/nix]$ nix-instantiate -E "1 + x"
error: --- UndefinedVarError --------------------------------------------- nix-instantiate
at: (1:5) from string

     1| 1 + x
      |     ^

undefined variable 'x'
```

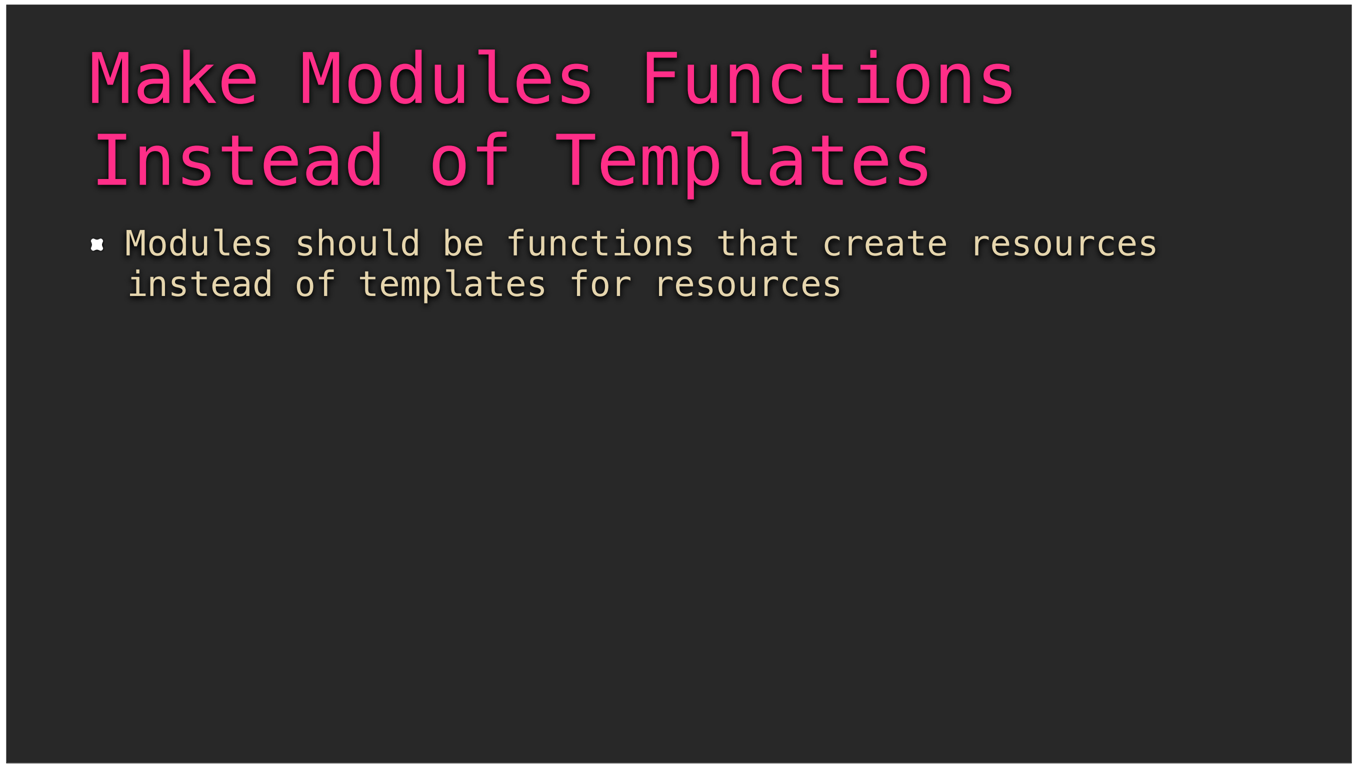
Much better!



Automagic Software Builds

- Leverage language-specific package managers to automatically derive dependencies when possible
- Low maintenance required

Another way that things could get a lot better is by taking advantage of language-specific package managers to automatically derive dependencies from their lockfiles. There are tools to do this, but ideally I'd like to see them be automatically generated, a-la import from derivation. This existing state of the world makes packaging things like Go programs annoyingly iterative and nontrivial.



Make Modules Functions Instead of Templates

- Modules should be functions that create resources instead of templates for resources

Maybe modules should be more like functions than template expansions. I am not sure how this would work, but it would be nice to make the outputs more fungible than with templates.

Questions? Ping me on Twitter

@theprincessxena

christine.website

And that about wraps it up! NixOS is pretty great until it isn't, and that latter part is frustrating because I wish this could be the obvious choice. It just isn't.

If you have any questions, please feel free to ping me on Twitter, or on the contact page on my website. I enjoy answering questions and openly welcome them. I'll stick around in the chat for a bit and answer questions if you want to ask them there.