

Dotfiles não bastam

Seu ambiente não é uma pasta de configs

Gabs Fontes

Antes: o que são dotfiles?

Os arquivos que deixam seus programas do seu jeito.

- `~/.bashrc`: aliases e comportamento do shell.
- `~/.gitconfig`: preferências do Git.
- `~/.config/alacritty/alacritty.toml`: aparência do terminal.

Uma prática comum: guardar tudo num Git e criar **links simbólicos** nos lugares onde os programas procuram essas configs.

Clonei os dotfiles. E agora?

- Quais programas preciso instalar? Em quais versões?
- Se eu mudar uma cor, quantos arquivos preciso editar?
- E se eu quiser só parte desse setup em outra máquina?

Ainda tem coisa pra instalar e configurar.

O que eu quero resolver aqui?

1. **Compleitude:** trazer ferramentas e versões junto com as configs.
2. **Transformação:** mudar uma preferência sem editar tudo à mão.
3. **Composição:** escolher quais partes do setup vão em cada máquina.

Vamos fazer isso com **Nix + Home Manager**.

1. Tá, mas quem instala o programa?

| “Quero formatar JSON no editor.”

Pra isso, preciso de **Helix + configuração + jq**.

Hoje, numa máquina nova:

- Instalo o Helix.
- Instalo o jq.
- Linko o `config.toml`.

```
[[language]]
name = "json"
[language.formatter]
command = "jq"
args = ["."]
```

O `config.toml` não sabe de onde vem o `jq`.

Primeiro, o Helix em Nix

```
programs.helix = {  
  enable = true;  
  settings.editor = {  
    line-number = "relative";  
    soft-wrap.enable = true;  
  };  
};
```

O Helix continua lendo TOML:

```
[editor]  
line-number = "relative"  
[editor.soft-wrap]  
enable = true
```

Até aqui, só trocamos a sintaxe.

Agora o `jq` entra na config

```
programs.helix.languages.language = [{  
  name = "json";  
  formatter.command = "${pkgs.jq}/bin/jq";  
  formatter.args = ["."];  
}];
```

No `languages.toml` gerado:

```
[[language]]  
name = "json"  
[language.formatter]  
command = "/nix/store/...-jq-1.7.1/bin/jq"  
args = ["."]
```

Esse caminho traz o pacote junto. Não precisa procurar `jq` no `PATH`.

E não é só no editor

```
programs.git = {  
  enable = true;  
  settings.core.pager = "${pkgs.delta}/bin/delta";  
};
```

Na configuração do Git:

```
[core]  
  pager = "/nix/store/...-delta-0.18.2/bin/delta"
```

Versionar `.gitconfig` **não instala** `delta`.

2. Mudei a cor. Preciso editar tudo?

| “Quero mudar a paleta do ambiente.”

Hoje, o mesmo `#1e293b` está em:

- `alacritty.toml`, como fundo do terminal.
- `mako/config`, como fundo das notificações.
- `gtk.css`, como fundo das janelas.

A mesma cor, copiada em cada arquivo.

Isso já tem solução sem Nix

- **pywal:** gera paletas a partir de imagens e integra cores aos aplicativos.
- **flavours:** aplica esquemas Base16 com templates e hooks.

Por exemplo, no `config.toml` do flavours:

```
[[items]]
file = "~/.config/sway/config"
template = "sway"
subtemplate = "colors"
hook = "swaymsg reload"
```

```
[colors.primary]
background = '#{base00-hex}'
foreground = '#{base05-hex}'
```

Pra trocar cores e dar reload, não precisa de Nix.

Em Nix, a paleta é só dados

```
# palette.nix
{
  base00 = "1e293b";
  base05 = "e5e7eb";
  base0D = "a5b4fc";
}
```

```
# alacritty.nix e mako.nix importam a paleta
programs.alacritty.settings.colors.primary.background =
  "${palette.base00}";
services.mako.settings.global.background-color =
  "${palette.base00}dd";
```

Mudo `base00` uma vez. Alacritty e Mako mudam juntos.

Então Nix é só mais um template?

Template `.mustache`

```
[colors.primary]
background = '#{base00-hex}'
foreground = '#{base05-hex}'
```

Nix → Home Manager

```
colors.primary = {
  background = "${palette.base00}";
  foreground = "${palette.base05}";
};
```

Com cores equivalentes, ambos produzem:

```
[colors.primary]
background = "#1e293b"
foreground = "#e5e7eb"
```

Até aqui, empate.

Certo. Então o que o Nix acrescenta?

Uma paleta

- gera a configuração do terminal
- gera a configuração das notificações
- entra no build de um tema GTK

Podemos usar essa paleta como entrada do build de um pacote.

Nix cuida do build e das dependências. Home Manager configura os programas pra usar o pacote gerado.

Dá até pra rebuildar o tema GTK

Um exemplo real: **Materia**, com as cores do ambiente.

```
materiaTheme = name: colors:
  pkgs.materia-theme.overrideAttrs (old: {
    # ... dependências e demais cores omitidas
    postPatch = ''
      cat > /build/gtk-colors << EOF
        BG=${colors.base00}
        FG=${colors.base05}
        ACCENT_BG=${colors.base0D}
      EOF
      ./change_color.sh -o ${name} /build/gtk-colors \
        -i False -t "$out/share/themes"
    '';
  });
```

Isso gera um pacote do Materia com as nossas cores.

3. Quero uma funcionalidade, não um programa

Pra escolher partes por máquina, primeiro preciso definir as partes.

Em vez de começar por “**configurar Hyprland**”:
“**quero bloquear minha sessão**”.

```
bloquear-sessao.nix
├─ habilita e configura Hyprlock
├─ adiciona o atalho no Hyprland
└─ configura o bloqueio automático no Hypridle
```

Pra bloquear a sessão, configuramos esses três programas juntos.

Um programa, várias funcionalidades

```
bloquear-sessao.nix —┬─→ Hyprlock
                      │├─→ Hypridle
                      └─→ Hyprland: atalho para bloquear

capturar-tela.nix —┬─→ ferramenta de captura
                   └─→ Hyprland: atalho para capturar
```

Não precisa enfiar tudo num `hyprland.nix` gigante.

O módulo de bloqueio define um atalho. O de captura define outro.

Cada módulo coloca sua parte

```
# bloquear-sessao.nix
{ pkgs, ... }: {
  programs.hyprlock.enable = true;
  wayland.windowManager.hyprland.settings.bind = [
    "SUPER, L, exec, ${pkgs.hyprlock}/bin/hyprlock"
  ];
}
```

```
# capturar-tela.nix
{ pkgs, ... }: {
  wayland.windowManager.hyprland.settings.bind = [
    "SUPER, P, exec, ${pkgs.grim}/bin/grim"
  ];
}
```

Os dois atalhos entram na lista final.

Agora, o que eu quero em cada máquina?

```
# laptop.nix
imports = [
  ./editor.nix
  ./cli.nix
  ./desktop.nix
  ./bloquear-sessao.nix
  ./capturar-tela.nix
];
```

```
# headless.nix
imports = [ ./editor.nix ./cli.nix ];
```

Pra levar isso pra outra máquina, importamos os módulos.

Mas eu preciso trocar de distro?

Não! Dá pra usar Home Manager no Arch, Ubuntu, WSL...

- Comece com editor e ferramentas de terminal.
- Não precisa migrar tudo de uma vez.
- A parte gráfica e a integração com o sistema podem dar mais trabalho.

Home Manager **standalone** é independente da configuração do sistema.

Vale a pena pra todo mundo?

- Você precisa repetir o setup em outras máquinas.
- Troca cores, fontes ou outras preferências com frequência.
- Suas máquinas têm configs em comum, mas também têm diferenças.

Só tem dois arquivos estáticos e um computador? Talvez não precise disso.

Aprender Nix leva tempo, e algumas integrações dão trabalho.
Seus dados continuam precisando de **backup**.

A ideia é essa

Compleitude: configs, ferramentas e versões juntas.

Transformação: mudar a paleta num lugar só, inclusive pro build do tema.

Composição: escolher as funcionalidades de cada máquina pelos imports.

Perguntas?