OFICINA · PETECO / UTFPR

NixOS: Commit, Rebuild, Rollback

Um Linux declarativo, reproduzível e versionado com Git

Marco Antônio Kochem Vendramin
PET Engenharia de Computação (PETECO) · UTFPR Curitiba

Base: estudo da linguagem Nix e Dolstra (2006)

O nome da oficina é o nosso método

COMMIT Registrar a configuração declarativa no Git e publicá-la no GitHub.

REBUILD Reconstruir o NixOS e o ambiente de usuário a partir dos arquivos Nix.

ROLLBACK Retornar a gerações anteriores quando uma alteração causar problemas.

Não é só instalar Linux: é administrar seu estado de forma controlada.

A pergunta que motivou o Nix

Como instalar, atualizar e remover software sem quebrar o que já funciona?

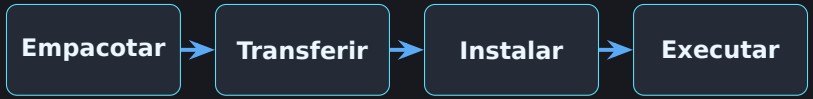
Dependências Muitas e nem sempre declaradas

Mudanças Versões alteram o ambiente

Reversão Voltar atrás precisa ser seguro

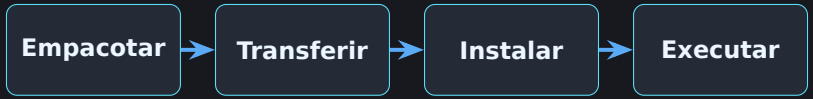
Fonte: Dolstra (2006, sec. 1.1, pp. 3–6).

O que significa implantar software?



- Descobrir e disponibilizar **todas as dependências**.

O que significa implantar software?



- Descobrir e disponibilizar **todas as dependências**.
- Gerenciar atualizações, remoções e consultas de componentes.
- Considerar limites externos: **hardware**, rede e serviços em execução.

Fonte: Dolstra (2006, sec. 1.1, pp. 3–6).

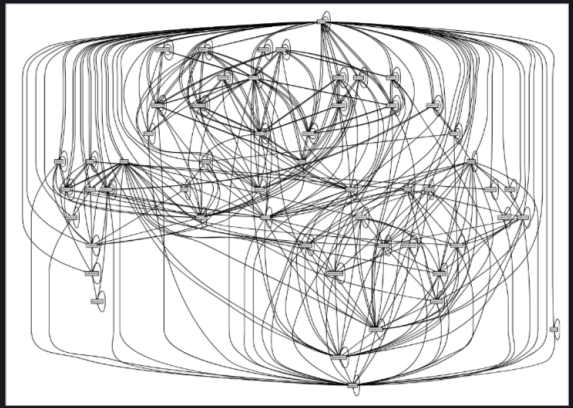
Dois problemas clássicos

DLL hell	Bit rot
Uma biblioteca compartilhada é **substituída ou removida**.	O código da aplicação **não mudou**.
Outros programas deixam de funcionar.	O ambiente evoluiu, e ela parou de funcionar.

Problema comum: o ambiente faz parte do software.

Fonte: Dolstra (2006, sec. 1.1, p. 5).

O inferno das dependências



Mozilla Firefox: grafo de dependências em tempo de execução.

The runtime dependency graph of Mozilla Firefox.

Fonte: Dolstra (2006, Figura 1.5, p. 10).

Gerenciamento de pacotes em Unix

- Ferramentas de pacotes sustentam **instalação, atualização e manutenção** de distribuições inteiras.
- Diretórios globais, como `/usr/bin`, centralizam nomes e componentes.
- O reaproveitamento de bibliotecas introduz **cadeias de dependências** cada vez mais complexas.

Tensão central Compartilhar componentes economiza recursos, mas exige controle rigoroso das versões.

Fonte: Dolstra (2006, sec. 1.2, pp. 6–11).

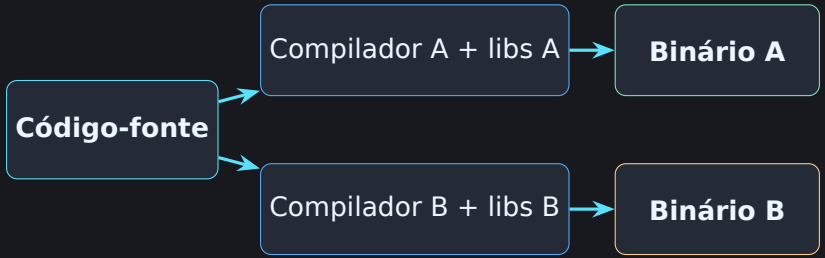
Dois caminhos para distribuir software

Pacotes binários	Compilação do código-fonte
Vantagens Instalação rápida; atualizações automatizáveis.	**Vantagens** Personalização, correções locais e controle de opções.
Limite Dependem das opções e versões disponíveis.	**Limite** Mais esforço, tempo e variação no ambiente de build.

Exemplos históricos de abordagem source-based: Gentoo e Ports do FreeBSD.

Fonte: Dolstra (2006, sec. 1.2, pp. 11–12).

Mesmo código-fonte ≠ mesmo binário



Ferramentas, bibliotecas e parâmetros do build também são entradas.

Implantação monolítica: a ideia

- Uma aplicação é distribuída com várias dependências próprias.
- Benefício: **menos conflitos** com programas vizinhos.
- Custo: **duplicação** de bibliotecas, transferência e espaço.

Aplicação A + biblioteca X

Aplicação B + biblioteca X

Fonte: Dolstra (2006, sec. 1.2, p. 12).

O custo do compartilhamento vs. duplicação

Dependências duplicadas	Dependências compartilhadas
$\Theta(NM)$	$\Theta(N+M)$
Cada uma das N aplicações guarda M dependências comuns.	Aplicações e dependências são armazenadas uma vez.

Modelo ilustrativo; assume componentes de custo comparável.

Fonte: Dolstra (2006, sec. 1.2, p. 12); modelo ilustrativo.

Bundles do macOS: um meio-termo

- Um **application bundle** agrupa os recursos necessários à aplicação.
- Em muitos casos, pode ser **copiado ou movido** sem instalação complexa.
- Funciona bem para aplicações finais, mas menos para dependências **finas e reutilizáveis**.

Limite Autossuficiência não resolve todo o gerenciamento de componentes compartilháveis.

Fonte: Dolstra (2006, sec. 1.2, p. 12).

Duas respostas históricas a conflitos

.NET / Global Assembly Cache

- Assemblies com identidade e versão.
- Versões coexistem no GAC.
- Solução específica do ecossistema .NET.

Zero Install

- Componentes obtidos conforme a demanda.
- Cache local e isolamento entre usuários.
- Implementação histórica tinha limitações de portabilidade.

Soluções parciais para uma questão mais geral: **gerenciar o ambiente completo.**

Fonte: Dolstra (2006, sec. 1.2, pp. 12–13).

Implantação puramente funcional

- Componentes definidos pelas **entradas declaradas** do build.

Implantação puramente funcional

- Componentes definidos pelas **entradas declaradas** do build.
- Cada variante é mantida em um **caminho próprio** na Nix Store.
- Diferentes versões podem **coexistir** sem sobrescrever outras.

Fonte: Dolstra (2006, secs. 1.4 e 2.1, pp. 14 e 19–24).

A Nix Store

/nix/store/

`hash-hello-2.1.1`



Os nomes reais usam uma representação hash no formato adotado pelo Nix; o exemplo acima foi abreviado.

O que entra na identidade de uma construção?

- Código-fonte
- Script de construção
- Argumentos e variáveis declaradas
- Compilador e linker
- Bibliotecas de build
- Ferramentas de construção

Consequência Mudar entradas relevantes pode produzir outra identidade e outro caminho na store.

Os ganhos do modelo funcional

- **Coexistência** de versões
- **Isolamento** entre componentes
- **Compartilhamento** quando idênticos
- Atualizações **atômicas**
- **Rollback** para gerações anteriores
- Coleta de lixo de itens não utilizados

Não é apenas instalar pacotes: é controlar a evolução do ambiente.

Fonte: Dolstra (2006, secs. 1.4 e 2.1, pp. 14 e 19–24).

Quando uma dependência não foi declarada

Exemplo de vinculação

```
gcc -o program main.c ui.c -lssl
```

- Um **linker** convencional pode localizar bibliotecas em diretórios globais.
- A construção passa a depender de uma **entrada não declarada**.
- Com caminhos isolados, a dependência precisa estar acessível no ambiente de construção.

O isolamento reduz entradas implícitas; não significa pureza absoluta em todos os cenários.

Dependências retidas em executáveis

```
00000000  7f 45 4c 46 01 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  02 00 03 00 01 00 00 00  40 bb 04 08 34 00 00 00  |........@...4...|
...
00000130  04 00 00 00 2f 6e 69 78  2f 73 74 6f 72 65 2f 37  |..../nix/store/7|
00000140  32 62 79 32 69 77 35 77  64 38 69 68 72 35 79 37  |2by2iw5wd8ihr5y7|
00000150  6e 31 31 31 6a 66 77 6c  37 33 71 32 68 36 37 2d  |n111jfwl73q2h67-|
00000160  67 6c 69 62 63 2d 32 2e  33 2e 35 2f 6c 69 62 2f  |glibc-2.3.5/lib/|
00000170  6c 64 2d 6c 69 6e 75 78  2e 73 6f 2e 32 00 00 00  |ld-linux.so.2...|
...
00002670  73 74 6f 72 65 2f 35 6a  71 36 6a 67 6b 61 6d 78  |store/5jq6jgkamx|
00002680  6a 6a 64 64 6c 61 76 67  76 63 39 6e 76 30 6c 72  |jjddlavgvc9nv0lr|
00002690  6d 38 66 79 73 37 2d 6f  70 65 6e 73 73 6c 2d 30  |m8fys7-openssl-0|
000026a0  2e 39 2e 37 64 2f 6c 69  62 3a 2f 6e 69 78 2f 73  |.9.7d/lib:/nix/s|
...
```

Referências a componentes da **Nix Store** permanecem incorporadas ao executável.

`glibc` `openssl` `svn`

Fonte: Dolstra (2006, Figura 2.4, p. 23).

RPATH e o fecho de dependências

RPATH em um executável ELF

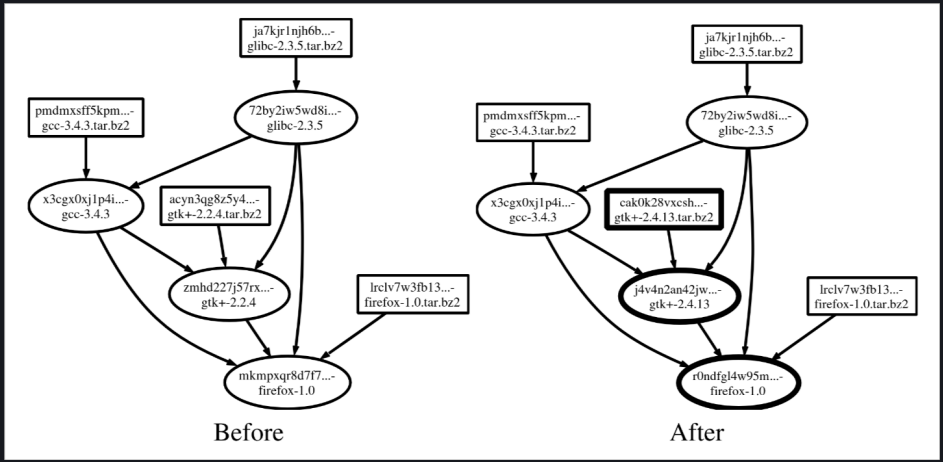
Pode registrar diretórios de bibliotecas dinâmicas, inclusive `/nix/store/.../lib`.

Closure (fecho)

Ao transferir um componente, é necessário incluir as dependências alcançáveis pelas referências relevantes.

Ideia principal O Nix identifica referências a caminhos da store para manter as dependências necessárias.

Propagação de mudanças na Nix Store



Antes e depois: uma mudança nas dependências de construção pode alterar os caminhos dos componentes dependentes.

Fonte: Dolstra (2006, Figura 2.3, p. 22).

Por que a linguagem Nix importa?

Descrição declarativa

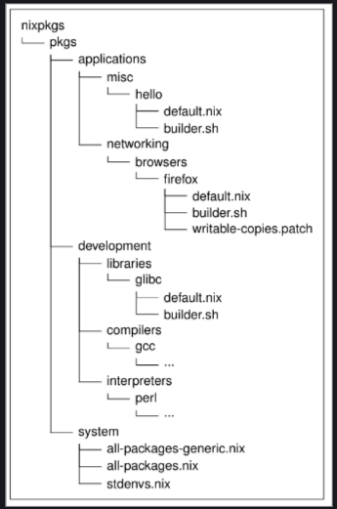
Declarar **o que compõe** um pacote ou sistema, em vez de depender apenas de alterações manuais acumuladas.

Avaliação funcional e preguiçosa

Expressões descrevem valores e dependências; a avaliação ocorre conforme a necessidade.

Configuração versionável + dependências explícitas

Nixpkgs: organização das expressões



- `applications`, `development` e `system` organizam as expressões.
- `default.nix`: descrição de um componente.
- `builder.sh`: etapas de construção do exemplo.
- **Repositório de definições** não é a mesma coisa que a **Nix Store**.

Fonte: Dolstra (2006, Figura 2.5, p. 26).

GNU Hello: a expressão `default.nix`

```
{stdenv, fetchurl, perl}: 2

stdenv.mkDerivation { 3
  name = "hello-2.1.1"; 4
  builder = ./builder.sh; 5
  src = fetchurl { 6
    url = http://ftp.gnu.org/pub/gnu/hello/hello-2.1.1.tar.gz;
    md5 = "70c9ccf9fac07f762c24f2df2290784d";
  };
  inherit perl; 7
}
```

- `{stdenv, fetchurl, perl}`: recebe argumentos.

Fonte: Dolstra (2006, Figura 2.6, p. 27).

GNU Hello: a expressão `default.nix`

```
{stdenv, fetchurl, perl}: 2

stdenv.mkDerivation { 3
  name = "hello-2.1.1"; 4
  builder = ./builder.sh; 5
  src = fetchurl { 6
    url = http://ftp.gnu.org/pub/gnu/hello/hello-2.1.1.tar.gz;
    md5 = "70c9ccf9fac07f762c24f2df2290784d";
  };
  inherit perl; 7
}
```

- `{stdenv, fetchurl, perl}`: recebe argumentos.
- `mkDerivation` descreve a construção.

Fonte: Dolstra (2006, Figura 2.6, p. 27).

GNU Hello: a expressão `default.nix`

```
{stdenv, fetchurl, perl}: 2

stdenv.mkDerivation { 3
  name = "hello-2.1.1"; 4
  builder = ./builder.sh; 5
  src = fetchurl { 6
    url = http://ftp.gnu.org/pub/gnu/hello/hello-2.1.1.tar.gz;
    md5 = "70c9ccf9fac07f762c24f2df2290784d";
  };
  inherit perl; 7
}
```

- `{stdenv, fetchurl, perl}`: recebe argumentos.
- `mkDerivation` descreve a construção.
- `src` obtém o código-fonte.

Fonte: Dolstra (2006, Figura 2.6, p. 27).

GNU Hello: a expressão `default.nix`

```
{stdenv, fetchurl, perl}: 2

stdenv.mkDerivation { 3
  name = "hello-2.1.1"; 4
  builder = ./builder.sh; 5
  src = fetchurl { 6
    url = http://ftp.gnu.org/pub/gnu/hello/hello-2.1.1.tar.gz;
    md5 = "70c9ccf9fac07f762c24f2df2290784d";
  };
  inherit perl; 7
}
```

- `{stdenv, fetchurl, perl}`: recebe argumentos.
- `mkDerivation` descreve a construção.
- `src` obtém o código-fonte.
- `inherit perl` fornece a dependência.

Fonte: Dolstra (2006, Figura 2.6, p. 27).

GNU Hello: o `builder.sh`

```
source $stdenv/setup [8]

PATH=$perl/bin:$PATH [9]

tar xvfz $src [10]
cd hello-*
./configure --prefix=$out [11]
make [12]
make install
```

- `source`: prepara `stdenv`.

Fonte: Dolstra (2006, Figura 2.7, p. 28).

GNU Hello: o `builder.sh`

```
source $stdenv/setup [8]

PATH=$perl/bin:$PATH [9]

tar xvfz $src [10]
cd hello-*
./configure --prefix=$out [11]
make [12]
make install
```

- `source`: prepara `stdenv`.
- `PATH`: inclui ferramentas declaradas.

Fonte: Dolstra (2006, Figura 2.7, p. 28).

GNU Hello: o `builder.sh`

```
source $stdenv/setup [8]

PATH=$perl/bin:$PATH [9]

tar xvfz $src [10]
cd hello-*
./configure --prefix=$out [11]
make [12]
make install
```

- `source`: prepara `stdenv`.
- `PATH`: inclui ferramentas declaradas.
- `tar`: extrai a fonte.

Fonte: Dolstra (2006, Figura 2.7, p. 28).

GNU Hello: o `builder.sh`

```
source $stdenv/setup [8]

PATH=$perl/bin:$PATH [9]

tar xvfz $src [10]
cd hello-*
./configure --prefix=$out [11]
make [12]
make install
```

- `source`: prepara `stdenv`.
- `PATH`: inclui ferramentas declaradas.
- `tar`: extrai a fonte.
- `configure --prefix=$out`: define destino.

Fonte: Dolstra (2006, Figura 2.7, p. 28).

GNU Hello: o `builder.sh`

```
source $stdenv/setup 8

PATH=$perl/bin:$PATH 9

tar xvfz $src 10
cd hello-*
./configure --prefix=$out 11
make 12
make install
```

- `source`: prepara `stdenv`.
- `PATH`: inclui ferramentas declaradas.
- `tar`: extrai a fonte.
- `configure --prefix=$out`: define destino.
- `make`: compila.

Fonte: Dolstra (2006, Figura 2.7, p. 28).

GNU Hello: o `builder.sh`

```
source $stdenv/setup [8]

PATH=$perl/bin:$PATH [9]

tar xvfz $src [10]
cd hello-*
./configure --prefix=$out [11]
make [12]
make install
```

- `source`: prepara `stdenv`.
- `PATH`: inclui ferramentas declaradas.
- `tar`: extrai a fonte.
- `configure --prefix=$out`: define destino.
- `make`: compila.
- `make install`: instala na saída.

Fonte: Dolstra (2006, Figura 2.7, p. 28).

GNU Hello: composição de dependências

```
...

rec { 13

  hello = (import ../applications/misc/hello 14 { 15
    inherit fetchurl stdenv perl;
  };

  perl = (import ../development/interpreters/perl) { 16
    inherit fetchurl stdenv;
  };

  fetchurl = (import ../build-support/fetchurl) {
    inherit stdenv; ...
  };

  stdenv = ...;
}
```

- `rec`: atributos podem referir-se entre si.

Fonte: Dolstra (2006, Figura 2.8, p. 30).

GNU Hello: composição de dependências

```
...

rec { 13

  hello = (import ../applications/misc/hello 14 { 15
    inherit fetchurl stdenv perl;
  };

  perl = (import ../development/interpreters/perl) { 16
    inherit fetchurl stdenv;
  };

  fetchurl = (import ../build-support/fetchurl) {
    inherit stdenv; ...
  };

  stdenv = ...;
}
```

- `rec`: atributos podem referir-se entre si.
- `import`: carrega e avalia a expressão.

Fonte: Dolstra (2006, Figura 2.8, p. 30).

GNU Hello: composição de dependências

```
...

rec { 13

  hello = (import ../applications/misc/hello 14 { 15
    inherit fetchurl stdenv perl;
  };

  perl = (import ../development/interpreters/perl) { 16
    inherit fetchurl stdenv;
  };

  fetchurl = (import ../build-support/fetchurl) {
    inherit stdenv; ...
  };

  stdenv = ...;
}
```

- `rec`: atributos podem referir-se entre si.
- `import`: carrega e avalia a expressão.
- `inherit`: fornece argumentos por nome.

Fonte: Dolstra (2006, Figura 2.8, p. 30).

GNU Hello: composição de dependências

```
...

rec { 13

  hello = (import ../applications/misc/hello 14 { 15
    inherit fetchurl stdenv perl;
  };

  perl = (import ../development/interpreters/perl) { 16
    inherit fetchurl stdenv;
  };

  fetchurl = (import ../build-support/fetchurl) {
    inherit stdenv; ...
  };

  stdenv = ...;
}
```

- `rec`: atributos podem referir-se entre si.
- `import`: carrega e avalia a expressão.
- `inherit`: fornece argumentos por nome.
- **Compor** é chamar a função com as dependências.

Fonte: Dolstra (2006, Figura 2.8, p. 30).

Expressão, construção e composição



Distinção Avaliar a expressão não é o mesmo que executar a compilação.

Fonte: elaboração própria com base em Dolstra (2006, Figs. 2.6–2.8).

MD5 no exemplo de 2006; SHA-256 hoje

Exemplo histórico

```
md5 = "70c9...";
```

MD5: resumo de **128 bits**; colisões criptográficas conhecidas.

Prática recomendada

```
hash = "sha256-...";
```

SHA-256: resumo de **256 bits**, normalmente em formato SRI.

Não confundir O hash do arquivo baixado e o identificador do caminho na Nix Store cumprem papéis diferentes.

Instalar, atualizar e reverter o GNU Hello

Exemplo histórico com `nix-env`

```
nix-env -f .../all-packages.nix -i hello
nix-env -q
nix-env --rollback
```

- A atualização produz uma nova versão sem sobrescrever a antiga.
- As gerações permitem voltar a um estado anterior do ambiente.
- A interface `nix-env` é **histórica**; o slide seguinte mostra um fluxo moderno do NixOS.

Fonte: Dolstra (2006, sec. 2.3, gerenciamento de pacotes).

Da teoria à prática: o sistema como código

O sistema deixa de ser uma coleção de mudanças manuais.

Ele passa a ser uma configuração declarada e versionada.

NixOS Sistema, serviços e hardware

Home Manager
Ambiente e preferências pessoais

Flakes + Git
Dependências fixadas e histórico

Fontes: documentação NixOS, Home Manager, nix.dev e estudo autoral.

As quatro peças da oficina

NixOS

Declara **serviços, usuários, rede e pacotes** do sistema.

Flakes

Organiza **entradas e saídas**; fixa revisões no `flake.lock`.

Home Manager

Declara **dotfiles, programas e opções** do usuário.

Git / GitHub

Guarda **commits**, histórico e cópia remota da configuração.

Configuração + dependências + histórico = reconstrução controlada

Fontes: documentação NixOS; Home Manager; Nix Flakes; Git.

1. NixOS: o arquivo `configuration.nix`

```
{ pkgs, ... }:
{
  networking.hostName = "doki-notebook";
  networking.networkmanager.enable = true;

  services.openssh.enable = true;
  environment.systemPackages = with pkgs; [ git vim ];
}
```

Onde atua? No nível do **sistema operacional**: serviços e ferramentas comuns.

Trecho parcial: configurações de inicialização, partições e hardware permanecem em outros módulos.

Fonte: NixOS Manual; exemplo didático de autoria própria.

2. Por que o Home Manager?

Sem Home Manager	Com Home Manager
• Configurações dispersas em `~/.config`.	• **Programas e dotfiles declarados** em Nix.
• Edição manual ou scripts de cópia.	• Configurações reutilizáveis em outras máquinas.
• Mais etapas para repetir o ambiente.	• **Ativação conjunta** com NixOS, como módulo.

Ideia O NixOS gerencia o computador; o Home Manager gerencia o ambiente do usuário.

Fonte: Home Manager Manual; síntese de autoria própria.

Home Manager: declarando o ambiente pessoal

```
{ pkgs, ... }:
{
  home.username = "marco";
  home.homeDirectory = "/home/marco";
  home.stateVersion = "26.05";

  home.packages = with pkgs; [ htop wget ];
  programs.git = {
    enable = true;
    settings.init.defaultBranch = "main";
  };
}
```

`home.stateVersion` é um marcador de compatibilidade, não uma opção a atualizar a cada lançamento.

Fonte: Home Manager Manual; exemplo didático de autoria própria.

Integração: um único rebuild



Com Home Manager integrado, não é preciso usar `home-manager switch` separadamente.

Fontes: documentação de Home Manager como módulo NixOS.

3. Flakes: o problema das versões

A pergunta

Se duas máquinas têm o **mesmo** `configuration.nix`, por que podem obter **pacotes diferentes**?

`flake.nix`

Declara as fontes: `nixpkgs`, Home Manager e resultados como `nixosConfigurations`.

`flake.lock`

Registra revisões concretas das dependências externas, inclusive seus identificadores.

Ponto central `git pull` recupera o lock; `nix flake update` modifica deliberadamente o lock.

Anatomia de um `flake.nix`

```
{
  description = "NixOS versionado";
  inputs.nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";

  outputs = { self, nixpkgs }: {
    nixosConfigurations.doki-notebook =
      nixpkgs.lib.nixosSystem {
        system = "x86_64-linux";
        modules = [ ./configuration.nix ];
      };
  };
}
```

`inputs`: de onde vêm as dependências. `outputs`: o que o projeto disponibiliza.

Fonte: nix.dev, Flakes; exemplo didático de autoria própria.

Home Manager como *input* de um Flake

```
inputs = {
  nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";
  home-manager = {
    url = "github:nix-community/home-manager/release-26.05";
    inputs.nixpkgs.follows = "nixpkgs";
  };
};

# Dentro de nixosSystem.modules:
modules = [
  ./configuration.nix
  home-manager.nixosModules.home-manager
  { home-manager.users.marco = import ./home.nix; }
];
```

`follows` faz o Home Manager utilizar a entrada `nixpkgs` do Flake principal. Trechos ilustrativos de uma configuração completa.

Fonte: Home Manager Manual, módulo NixOS; exemplo didático de autoria própria.

4. Organizando o repositório GitHub

```
nixos-dotfiles/
  flake.nix
  flake.lock
  hosts/
    doki-notebook/
      configuration.nix
      hardware-configuration.nix
  modules/common.nix
  home/marco.nix
  README.md
```

- **hosts/**: diferenças de hardware e máquina.
- **modules/**: opções comuns, reutilizáveis.
- **home/**: ambiente do usuário.
- **flake.lock**: revisões exatas de entradas.

Fonte: exemplo didático de autoria própria; documentação Git e Nix Flakes.

O GitHub armazena o quê?

Vai para o repositório

- `flake.nix` e `flake.lock`
- Módulos Nix e Home Manager
- Arquivos de configuração versionáveis
- Histórico de *commits*

Não é um backup completo

- Binários da *Nix Store*
- Documentos e dados pessoais
- Bancos de dados e estado de aplicativos
- Senhas e chaves privadas

Versionamos a descrição do sistema, não uma imagem do disco.

Commit: registrando uma mudança

```
cd ~/nixos-dotfiles

git status
git diff
git add flake.nix flake.lock hosts modules home
git commit -m "Configura NixOS e Home Manager"
git push
```

Commit Marca uma versão dos **arquivos declarativos** e publica seu histórico no remoto.

Confira os arquivos antes do `git add`; segredos em texto puro não devem ser commitados.

Fonte: documentação Git; exemplo didático de autoria própria.

Clone ou pull: trazendo a configuração

Primeira vez em uma instalação

```
git clone git@github.com:marcokvendramin/nixos-dotfiles.git
cd nixos-dotfiles
```

Próximas vezes

```
cd ~/nixos-dotfiles
git pull --ff-only
```

Atenção `git pull` **atualiza os arquivos**; não altera o NixOS em execução.

Fonte: documentação Git; exemplo didático de autoria própria.

Rebuild: construindo e ativando o sistema

Construir sem ativar

```
sudo nixos-rebuild build --flake .#doki-notebook
```

Construir e ativar

```
sudo nixos-rebuild switch --flake .#doki-notebook
```

`.`: Flake do diretório atual. `#doki-notebook`: saída nomeada em `nixosConfigurations`.

Uma reconstrução pode ativar NixOS e Home Manager juntos.

Fonte: NixOS Manual, nixos-rebuild; exemplo didático.

Git Pull ≠ Flake Update ≠ Rebuild

`git pull` Traz os arquivos versionados, incluindo `flake.lock`.

`nix flake update` Atualiza revisões das dependências no `flake.lock`.

`nixos-rebuild switch` Constrói e ativa a configuração declarada.

Três ações diferentes. Três momentos sob controle.

Atualização consciente das dependências

```
cd ~/nixos-dotfiles

# Atualizar as revisoes dos inputs
nix flake update

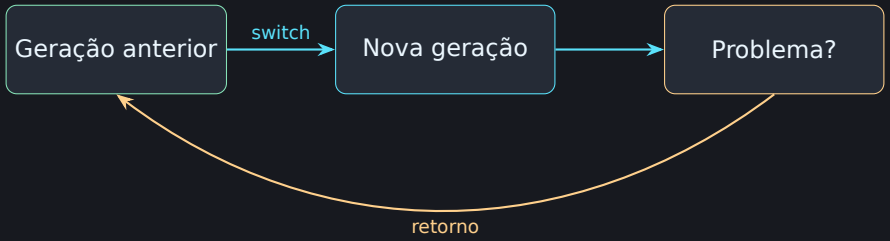
# Conferir antes de ativar
git diff flake.lock
sudo nixos-rebuild build --flake .#doki-notebook
sudo nixos-rebuild switch --flake .#doki-notebook

# Salvar a revisao testada
git add flake.lock
git commit -m "Atualiza dependencias"
git push
```

Sem `nix flake update`, a reconstrução utiliza as revisões presentes no `flake.lock`.

Fonte: nix.dev, Flakes; Git; NixOS.

Por que *Rollback* é parte do nome?



- As gerações anteriores podem permanecer disponíveis na store.
- A seleção de uma geração antiga **não exige sobrescrever** os pacotes novos.
- Git recupera o **código declarativo**; NixOS recupera **gerações do sistema**.

Fonte: Dolstra (2006), perfis e gerações; documentação NixOS.

Rollback: duas camadas de recuperação

Voltar à geração anterior do NixOS

```
sudo nixos-rebuild switch --rollback
```

Desfazer um commit e reconstruir

```
git log --oneline
git revert <hash-do-commit>
sudo nixos-rebuild switch --flake .#doki-notebook
```

Rollback não substitui backup de dados pessoais; gerações antigas também podem ser removidas pela coleta de lixo.

Fontes: NixOS, rollback; Git, revert.

Um detalhe que evita erros com Git + Flakes

Arquivo novo? Adicione ao índice do Git.

```
git add modules/bluetooth.nix
```

- Um arquivo novo ainda **não rastreado** pode ficar fora da fonte do Flake.
- É possível testar alterações locais antes de criar um commit.
- **Nunca** colocar segredos em texto puro no Git ou na Nix Store.

Fonte: documentação Nix Flakes, fontes do tipo Git.

O ciclo completo da oficina



Um sistema configurado como projeto: auditável, reutilizável e reversível.

O que conseguimos — e o que não prometemos

Reproduzir	Ainda requer cuidados
• Pacotes e serviços declarados.	• Partições e hardware diferentes.
• Configurações do Home Manager.	• Dados de aplicativos e documentos.
• Dependências fixadas no lock.	• Segredos, chaves e credenciais.
• Histórico e evolução no Git.	• Disponibilidade das fontes externas.

Síntese: o que muda com o Nix?

Sem integração declarativa

- Alterações dispersas em arquivos e comandos.
- Versões dependentes do momento de instalação.
- Recuperação exige repetir procedimentos.
- Reversões frequentemente trabalhosas.

Com NixOS + Flakes + Git

- Módulos e configurações versionados.
- Dependências externas fixadas no lock.
- Reconstrução guiada pela declaração.
- Gerações e histórico de commits.

Fonte: Dolstra (2006); documentação NixOS, Home Manager e Flakes.

Fontes para continuar aprendendo

- **Dolstra (2006):** *The Purely Functional Software Deployment Model*. Tese de doutorado, Universidade de Utrecht.
- **NixOS / Nix:** `https://nixos.org/learn/`
- **Linguagem e Flakes:** `https://nix.dev/`
- **Home Manager:** `https://wiki.nixos.org/wiki/Home_Manager`
- **Git no NixOS:** `https://wiki.nixos.org/wiki/Git`

Exemplos históricos de 2006 foram preservados e diferenciados dos comandos de Flakes e NixOS atuais.

Commit. Rebuild. Rollback.

Seu sistema pode ter código, histórico e gerações.

Marco Antônio Kochem Vendramin

PET Engenharia de Computação (PETECO) · UTFPR Curitiba

LinkedIn: linkedin.com/in/marcokvendramin

GitHub: github.com/marcokvendramin

Obrigado!