ADMIN:DOCS

# NixOS: the Nix package manager guide

How Nix and NixOS work: derivations, the Nix store, declarative configuration management with flakes, plus rollbacks and isolation in practical sysadmin work.

**Category:** Linux beginners **Published:** 24.10.2024 **Last reviewed:** 08.09.2026

**Classic Linux distributions have followed a familiar pattern for decades:** A package manager unpacks files directly into shared system directories such as `/usr/bin`, `/usr/lib` or `/etc`. On systems with APT on Ubuntu, DNF on Fedora, Zypper on openSUSE or Pacman on Arch Linux that works reliably in daily use — but it hits principled limits as soon as contradictory requirements collide.

If an older administration script needs a particular version of an OpenSSL or Python library, while a newly installed web application strictly requires newer library versions, the global filesystem goes into conflict (*dependency hell*). When a package is updated, it overwrites shared files of other programs. If a system update fails or aborts mid-unpack, the system often remains in an inconsistent intermediate state that needs manual repair.

Nix chooses a radically different architecture model:

Instead of scattering software globally and with state into a mutable filesystem, Nix treats every software package as a purely functional, immutable unit. No package can overwrite another, and every program is linked in isolation to exactly the libraries it was built with.

> **Nix vs. NixOS: Nix** is the package-oriented toolbox and the associated functional configuration language. You can install Nix safely as a standalone package manager on existing distributions such as Debian, Ubuntu, Fedora, Arch Linux or macOS without endangering the base system. **NixOS**, by contrast, is a complete Linux distribution that

Source: https://admindocs.de/en/linux-beginners/nixos-nix-package-manager-guide

> sits on the Linux kernel and systemd, but manages its entire operating system declaratively through Nix.

## How Nix works and its core concepts

### The functional approach: immutability and derivations

In the mathematical sense a pure function always yields the same result for identical inputs, completely independent of external state or the time of day. Nix transfers exactly this principle to software packaging.

A software build in Nix is the result of a uniquely defined function:

- **Inputs:** *Source archives, build scripts, compilers, dependent libraries and build flags.*
- **Function (build process):** *An isolated execution environment without uncontrolled network access and without access to foreign host files.*
- **Output:** *An immutable directory in the central storage location, the Nix store.*

The formal build recipe for such a package is called a **derivation** in Nix (file extension `.drv`). A derivation describes declaratively:

- Which source archives must be downloaded (secured via cryptographic SHA-256 checksums).
- Which dependencies and compiler tools must be provided in which version.
- Which concrete phases (configuration, compilation, test runs, installation) are run.

Once the build is finished, the directory in the filesystem is write-protected (`read-only`). Even the root user does not subsequently modify installed binaries in the store.

### The build process in detail

Nix enforces reproducibility through strict encapsulation. During compilation, execution runs in a secured sandbox:




The 32-character hash in the directory name (e.g. `3b9a7f8e...`) is not a random product. It represents the cryptographic fingerprint of all inputs: source-code hash, compiler version, compiler flags set and the hashes of all included libraries.

If a single line in the source is changed or a dependent library is patched, the computed hash changes. Nix places the newly built package in a completely independent directory. Existing versions remain untouched and continue to work without interruption.

## Nix compared with classic tools

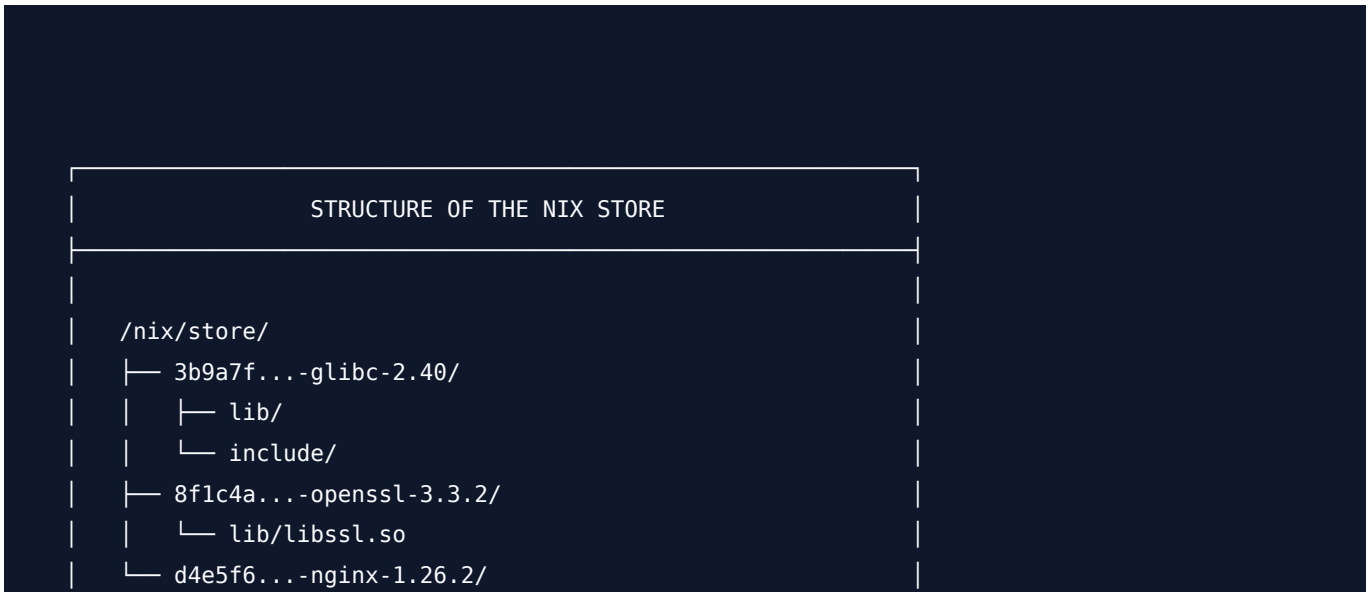
| Feature | Classic package managers (APT, DNF, Pacman) | Container runtimes (Docker, Podman) | Nix / NixOS |
|---|---|---|---|

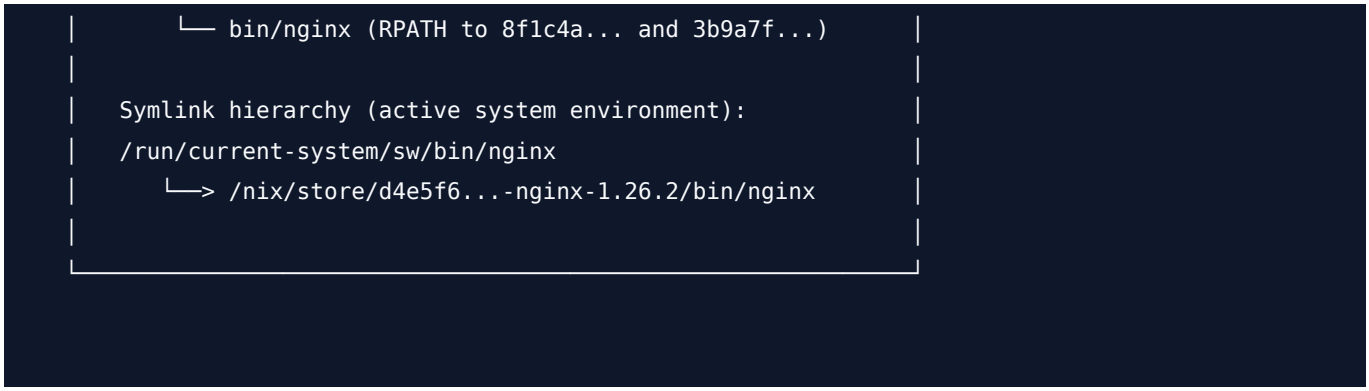
| | | | |
|---|---|---|---|
| **Paradigm** | Imperative (filesystem is changed step by step) | Image-based (encapsulated filesystem layers) | Functional and declarative |
| **Filesystem structure** | Shared FHS ( `/usr/bin` , `/usr/lib` ) | Own namespaces per container | Path isolation in `/nix/store` |
| **Parallel libraries** | Conflict-prone with the same name | Fine across separate containers | Native via unique store hashes |
| **Rollback behavior** | Manual, incomplete or filesystem-dependent | Image swap with restart | Atomic switch via symlinks |
| **Reproducibility** | Time-dependent on the repository mirror | Depends on base images and build steps | Fully deterministic |
| **Overhead** | Minimal (direct host binaries) | Memory and I/O overhead from layers | Minimal (native binaries, deduplication) |

# Architecture: Nix store, profiles and generations

## The Nix store ( `/nix/store` ): isolation through hashes

The foundation of every Nix system is the directory `/nix/store` . All binaries, dynamic libraries, configuration snippets and manual pages live here side by side:






Classic Linux programs look up dynamic libraries at runtime through the dynamic linker (`ld.so`) in global paths such as `/lib64` or `/usr/lib64`. If a file changes there, that affects all programs on the system.

Nix breaks this coupling: at compile time Nix writes the exact, immutable path of the required libraries directly into the binary header (via the ELF attribute `RPATH` or `RUNPATH`). A program under `/nix/store/...-nginx` loads its dependencies exclusively from the cryptographically linked store directories. A system-wide upgrade of a library does not touch running or older programs in any way.

> **Inspecting store dependencies:** With `nix-store -q --references /path/to/binary` you can list exactly, for every binary, which other paths in the store are included as a dependency. Conversely `nix-store -q --referrers /path/to/library` shows which installed programs actively reference a specific library.

## Profiles and generations: atomic updates and rollbacks

Because dozens of versions of the same program can coexist peacefully in `/nix/store`, it must be defined which programs are visible to a user in the search path (`$PATH`). That job is done by **profiles**.

A profile is a directory tree of symbolic links (symlinks) that points at the desired programs in the Nix store. Every modification of a profile creates a new, immutable **generation**:

Source: https://admindocs.de/en/linux-beginners/nixos-nix-package-manager-guide

```
# Install a package in the current user profile
nix profile install nixpkgs#htop

# Show all packages installed in the active profile
nix profile list

# Inspect the history of previous profile generations
nix profile history
```

When a package is installed, updated or removed, Nix does not copy files around. Instead a new symlink tree is created in the background. Only when that tree is fully ready does Nix swing the profile pointer to the new generation.

If unexpected behavior appears after an update, the profile can be reset to the previous state in fractions of a second:

```
# Immediate rollback to the immediately previous generation
nix profile rollback

# Switch specifically to a given generation number
nix profile rollback --to 3
```

Because the switch uses the atomic POSIX operation `rename()` on symlinks, there are never half or incomplete installation states.

## Temporary environments with `nix shell`

An outstanding advantage in sysadmin daily work is the ability to run tools ad hoc without installing them permanently on the host or dragging dependencies into the base system.

Previously you used `nix-shell -p` for this. In the modern Nix standard the flake-capable command `nix shell` takes over:

```
# Open a temporary interactive shell with Python 3.12 and the PostgreSQL client
nix shell nixpkgs#python312 nixpkgs#postgresql_16
```

Inside this subshell `python3` and `psql` are available immediately. As soon as you leave the shell with `exit`, your regular environment is unchanged: no binary remains in your global `$PATH`, no configuration was modified.

**□□ Practical example:**

You must analyze a complex JSON logfile on a server where neither `jq` nor `ripgrep` is installed. Instead of changing the base system through administrative package managers, you start an isolated session:

```
# Start a temporary environment with the required analysis tools
nix shell nixpkgs#jq nixpkgs#ripgrep

# Run the analysis directly
rg "ERROR" /var/log/app/service.log | jq '.payload.error_code'

# Leave the environment
exit
```

After leaving the subshell, a call to `which jq` shows that the system stayed clean. The Nix store still holds the data in cache, but the system directory remains untouched.

# NixOS: the fully declarative operating system

While Nix as a package manager runs on arbitrary operating systems, **NixOS** takes this concept to the extreme: the entire operating system — from kernel parameters and filesystem mounts through network routing and systemd units to user accounts — is described as a declarative desired state in configuration files.

## Declarative system control with configuration.nix

The central control file of a standard NixOS system lives under `/etc/nixos/configuration.nix`. Instead of typing commands such as `systemctl enable`, `useradd` or `ufw allow` by hand, you define the machine as code:

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

{
  imports = [
    ./hardware-configuration.nix
  ];

  # Bootloader and EFI integration
  boot.loader.systemd-boot.enable = true;
  boot.loader.efi.canTouchEfiVariables = true;

  # Network identity and firewall
  networking.hostName = "production-srv01";
  networking.firewall.allowedTCPPorts = [ 22 80 443 ];

  # Harden the SSH service
  services.openssh = {
    enable = true;
    settings.PermitRootLogin = "no";
    settings.PasswordAuthentication = false;
  };

  # System-wide base packages
  environment.systemPackages = with pkgs; [
```

```
      vim
      git
      curl
      htop
      tmux
      ripgrep
      jq
    ];

    # Unprivileged administrator account
    users.users.adminuser = {
      isNormalUser = true;
      extraGroups = [ "wheel" "networkmanager" ];
      openssh.authorizedKeys.keys = [
        "ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIExampleKeyValidAdmin admin@workstation"
      ];
    };

    # State compatibility of the NixOS installation
    system.stateVersion = "26.05";
  }
```

The declarative definition of SSH keys and firewall rules replaces manual configuration files under `/etc/ssh/sshd_config`. How you establish modern protective measures such as hardware-based authentication and adaptive blocklists is covered in more depth in the guide to Linux server hardening with FIDO2 and CrowdSec.

> ⚠ **Common misconception about `system.stateVersion`:** The parameter `system.stateVersion = "26.05"` does **not** control the operating-system upgrade! It only tells NixOS with which default assumptions for stateful data (e.g. database data paths or systemd file formats) the system was originally set up. Changing this variable without a targeted data migration can damage services. Upgrades are instead controlled exclusively through the package sources (channels or flake inputs) and the subsequent build.

To activate the defined configuration, one call of the rebuild tool is enough:

```
# Evaluate the system, build a new generation and switch to it immediately
sudo nixos-rebuild switch
```

NixOS analyzes the difference between the current state and the new definition, generates the required systemd services, adjusts configuration files under `/etc` and restarts affected daemons atomically.

**□□ Practical example:**

You want to test a faulty system change without risk. You accidentally changed the SSH port in `/etc/nixos/configuration.nix` or configured a service incorrectly and applied it.

To return immediately to the proven state, you run:

```
# Immediate return to the previous error-free system generation
sudo nixos-rebuild switch --rollback
```

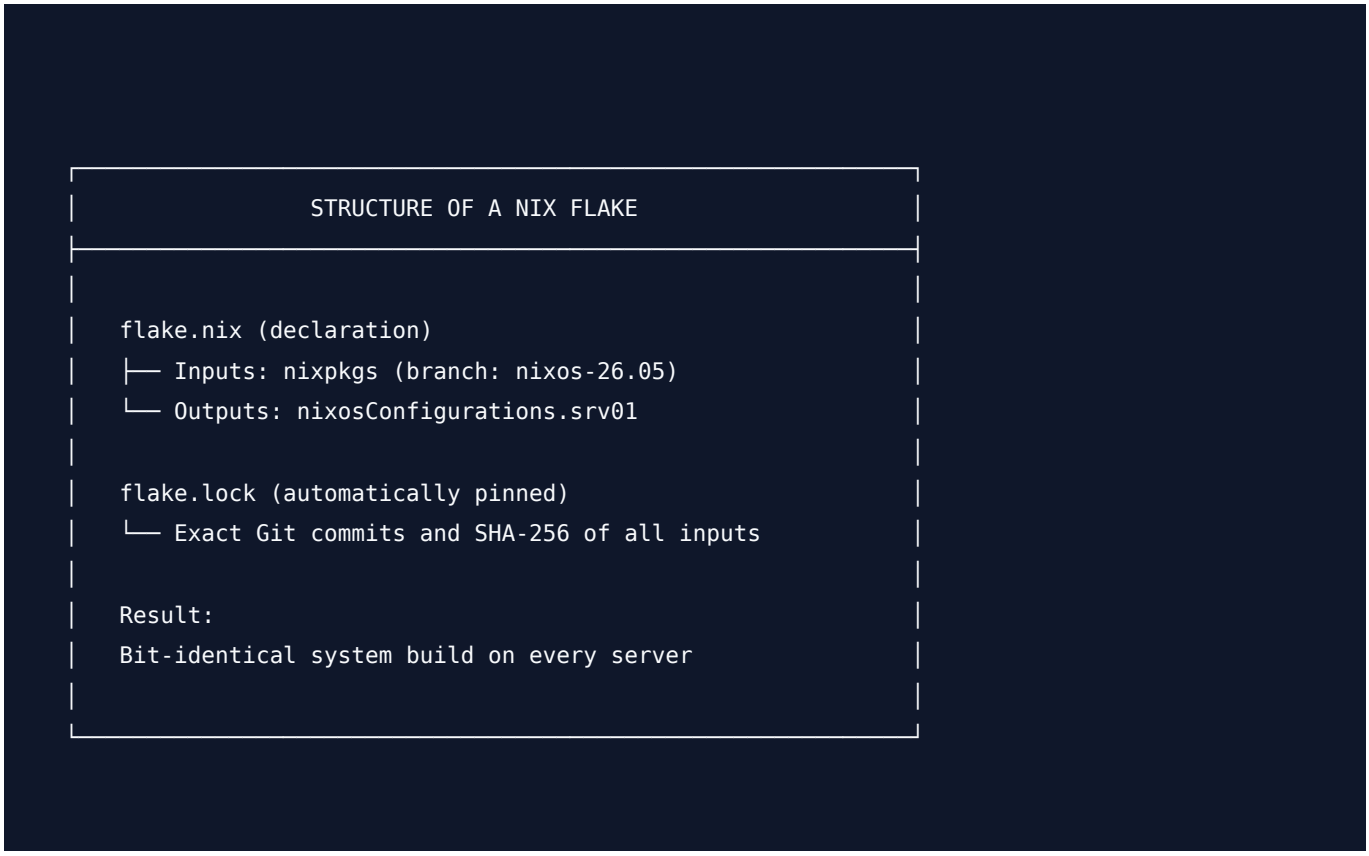
If a misconfiguration is so severe that the system no longer starts properly or there is no network access, you simply choose the previous menu entry at boot in the `systemd-boot` or GRUB menu. Every generation forms an independent, bootable system kernel.

## Modern configuration with flakes (as of 2026)

Earlier Nix versions relied on so-called *channels*. Channels had the drawback that the build state depended on the exact fetch time: if you ran `nix-channel --update` on two different servers on different days, the system could install diverging software states despite an identical configuration file.

**Flakes** solve this weakness completely. A flake is a self-contained project directory with two essential files:

1. `flake.nix`: Describes input sources (*inputs*) and provided artefacts (*outputs*).
2. `flake.lock`: An automatically generated lock file that pins every input to an immutable Git commit hash and SHA-256 value.




A practical `flake.nix` for a modern NixOS 26.05 system looks like this:

```
{
  description = "Production server flake according to the 2026 standard";

  inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";
  };

  outputs = { self, nixpkgs, ... }: {
    nixosConfigurations.production-srv01 = nixpkgs.lib.nixosSystem {
      system = "x86_64-linux";
      modules = [
```

```
          ./configuration.nix
        ];
      };
    };
  }
```

To enable flakes permanently on a system, the following directive is placed in `/etc/nixos/configuration.nix` (or in `~/.config/nix/nix.conf` for standalone Nix):

```
nix.settings.experimental-features = [ "nix-command" "flakes" ];
```

The system is then updated and activated directly from the flake:

```
# Rebuild and apply the system from the flake definition
sudo nixos-rebuild switch --flake .#production-srv01

# Raise all dependencies in flake.lock in a controlled way to the latest upstream
nix flake update
```

## Project-isolated development environments with `nix develop`

Flakes not only enable configuration of complete operating systems, they also transform development and administration workflows. Through the `devShells` attribute a complete working environment can be versioned in the source repository:

```
{
  description = "Development environment for infrastructure scripts";

  inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";
  };

  outputs = { self, nixpkgs }:
    let
      pkgs = nixpkgs.legacyPackages.x86_64-linux;
    in {
      devShells.x86_64-linux.default = pkgs.mkShell {
        buildInputs = with pkgs; [
          python312
          python312Packages.requests
          python312Packages.pyyaml
          ansible-lint
        ];

        shellHook = ''
          echo "Infrastructure DevShell loaded (Python 3.12 and Ansible tools active)"
        '';
      };
    };
}
```

Every team member who runs `nix develop` in this directory receives, to the second, the same toolchain with the same interpreter versions — regardless of whether Ubuntu, Fedora, NixOS or macOS is in use as the host. How you develop your own automation scripts for such environments in a structured way is covered in the practical Bash fundamentals course for Linux administrators.

## Installation: standalone Nix vs. full NixOS

Depending on the starting point and goal there are two established deployment models:

- **Standalone installation on existing Linux systems:**

On existing servers or workstations (Debian, Ubuntu, Arch Linux, macOS) the official multi-user daemon is set up. Unprivileged build users ( `nixbld1` through `nixbld32` ) are created who run compilation strictly without root rights:

```
# Run the official multi-user installation script
sh <(curl -L https://nixos.org/nix/install) --daemon
```

- **Full NixOS installation on bare metal or VMs:**

After starting the minimal NixOS installation image you partition the target drives (preferably with GPT, EFI system partition and ext4/Btrfs) and mount them under `/mnt` :

```
# Detect target-system hardware automatically and generate templates
nixos-generate-config --root /mnt

# Start installation according to the generated /mnt/etc/nixos/configuration.nix
nixos-install
```

## Advanced package management and DevOps workflows

### Overlays and overrides: your own patches and build flags

In daily operations the need occasionally arises to compile software with special compiler flags or to apply your own security patch before it is officially released in the upstream repository.

Instead of forking the entire package repository, Nix offers **overrides** (for individual packages) and **overlays** (for the global package ecosystem):




### Targeted override in `configuration.nix`:

```
environment.systemPackages = with pkgs; [
  # Build Vim with explicit Python3 support, but without graphical X11 libraries
  (vim.override {
    python3Support = true;
    guiSupport = false;
  })
];
```

### Global overlay for patches:

```
nixpkgs.overlays = [
```

```
    (final: prev: {
      nginx = prev.nginx.overrideAttrs (oldAttrs: {
        patches = (oldAttrs.patches or []) ++ [
          ./patches/custom-security-fix.patch
        ];
      });
    })
  ];
```

Nix applies the overlay deterministically: all system components that refer to `nginx` then automatically use the patched version in the store.

## Parallel software versions without name conflicts

Because Nix does not use global paths such as `/usr/lib`, different major and minor versions of the same programming language or database library can be installed and operated in parallel on the same system:

```
# Register Python 3.11 and Python 3.12 in the user profile at the same time
nix profile install nixpkgs#python311 nixpkgs#python312
```

Classic package managers fail on such requirements because header files and symlinks collide. In Nix each package gets its individual store path. Conflicts are architecturally excluded.

> ⚠ **Warning about the obsolete `nix-env`:** In older guides you often hit commands such as `nix-env -iA nixpkgs.htop`. In modern Nix, `nix-env` is an **anti-pattern** because it produces poorly pinned, hard-to-trace states. Use `nix shell` for ad-hoc tools, `nix profile` or Home Manager for user installations, and the declarative `configuration.nix` for systems.

Source: https://admindocs.de/en/linux-beginners/nixos-nix-package-manager-guide

## Binary caches and substituters: fast delivery

Despite the functional source-code approach, not every program has to be compiled locally on a Nix system.

When Nix is about to build a package, it first computes the resulting hash of the derivation. Then Nix checks whether this hash already exists on a precompiled binary cache (*substituter*):




By default Nix queries the official cache `cache.nixos.org`. If the finished archive is there, it is downloaded and unpacked in the local `/nix/store`. Only if user-defined flags or your own patches change the hash does Nix automatically switch to local compilation in the sandbox.

In enterprise networks you can add your own binary caches (e.g. via Attic, Harmonia or Cachix) so that self-built software packages are built once centrally in a CI/CD pipeline and delivered in seconds to hundreds of target systems.

## Building OCI containers with `pkgs.dockerTools`

A powerful use of Nix in DevOps is creating Docker and OCI containers. Instead of a classic `Dockerfile` with many uncontrolled `RUN apt-get update` steps, Nix produces container images purely declaratively and reproducibly:

Source: https://admindocs.de/en/linux-beginners/nixos-nix-package-manager-guide

```
{ pkgs ? import <nixpkgs> {} }:

pkgs.dockerTools.buildLayeredImage {
  name = "production-app";
  tag = "latest";
  contents = [
    pkgs.nodejs_22
    pkgs.curl
  ];
  config = {
    Cmd = [ "${pkgs.nodejs_22}/bin/node" "/app/server.js" ];
    WorkingDir = "/app";
    ExposedPorts = {
      "3000/tcp" = {};
    };
  };
}
```

```
# Build a container image deterministically without a running Docker daemon
nix-build docker-image.nix

# Load the produced tarball image directly into the local Docker engine
docker load < result
```

Because Nix knows exactly which files from the store are needed, the produced images contain neither unused package-manager leftovers nor temporary cache files. The result is extremely lean, minimally attackable container images.

Source: https://admindocs.de/en/linux-beginners/nixos-nix-package-manager-guide

# Maintenance, garbage collection and system diagnosis

## Garbage collection and store deduplication

Because Nix does not overwrite old files on updates and keeps previous system generations for rollbacks, disk use in `/nix/store` grows continuously.

Cleanup happens through the two-stage process of **garbage collection**:

1. **Discard old generations:** As long as a symlink (e.g. from an earlier system or profile generation) points at a store path, Nix considers that package active (*GC root*). Only when old generations are deleted is the path released for cleanup.
2. **Free space:** Store paths that are no longer referenced are deleted.

```
# 1. Delete only currently orphaned packages without a link from the store
nix-collect-garbage

# 2. Discard old profile and system generations and clean thoroughly
sudo nix-collect-garbage -d

# 3. Alternative via the modern flake CLI:
nix profile wipe-history
nix store gc
```

## Store optimization through hard links

If different packages contain identical files (e.g. identical license texts, help files or unchanged binaries), Nix can deduplicate them via hard links:

```
# Replace identical files across the entire Nix store with hard links
```

```
nix-store --optimise
```

This process often saves 20 to 40 percent of disk space without endangering isolation, because all files in the store are write-protected.

**Anchor automatic cleanup in `configuration.nix`:**

```
# Enable automatic deduplication on every build
nix.settings.auto-optimise-store = true;

# Weekly garbage collection for generations older than 14 days
nix.gc = {
  automatic = true;
  dates = "weekly";
  options = "--delete-older-than 14d";
};
```

## Debugging with `nix repl` and system diagnosis

When it is unclear why a particular package cannot be deleted or which configuration options exist, the built-in diagnostic commands help:

```
# Open the interactive evaluation environment of the Nix language
nix repl --expr 'import <nixpkgs> {}'

# Check why a package remains on the system (uncover the dependency chain)
nix why-depends /run/current-system nixpkgs#openssl

# Verify and repair consistency and SHA-256 signatures of the entire store
```

```
sudo nix-store --verify --check-contents --repair
```

## Command Reference (Cheatsheet)

| Command / invocation | Context | Purpose / function in sysadmin daily work |
|---|---|---|
| `nix shell nixpkgs#package` | Temporary | Starts a subshell with a provided tool without host installation |
| `nix run nixpkgs#package -- [args]` | Temporary | Runs a binary directly from the cache without holding it persistently |
| `nix develop` | Project | Loads the development and toolchain environment defined in `flake.nix` |
| `nix profile list` | User | Shows all software packages installed in the active user profile |
| `nix profile install nixpkgs#package` | User | Installs a package in isolation in the logged-in user's profile |
| `nix profile history` | User | Lists all previous profile generations chronologically |
| `nix profile rollback` | User | Resets the user profile atomically to the previous generation |
| `sudo nixos-rebuild switch` | NixOS | Evaluates `/etc/nixos/configuration.nix`, builds the system and activates it immediately |
| `sudo nixos-rebuild switch --rollback` | NixOS | Switches the entire operating system immediately back to the last generation |
| `sudo nixos-rebuild boot` | NixOS | Builds the system for the next reboot without disturbing running services |
| `nix flake update` | Flakes | Updates all dependencies in `flake.lock` to the latest upstream commit |

| Command / invocation | Context | Purpose / function in sysadmin daily work |
|---|---|---|
| `nix-collect-garbage -d` | Maintenance | Deletes old generations and removes unused store paths completely |
| `nix store optimise` | Maintenance | Deduplicates identical files in `/nix/store` via hard links |
| `nix why-depends /run/current-system <package>` | Diagnosis | Shows the exact dependency path why a package remains in the store |
| `sudo nix-store --verify --check-contents` | Diagnosis | Checks all store files for bit errors against their expected checksums |

## Further Resources

| Resource / documentation | Type | Description / purpose |
|---|---|---|
| Official NixOS documentation | Handbook | Complete reference manual for system administration and configuration |
| Nixpkgs manual | Developer docs | Detailed guidelines for packaging, overlays and build helpers |
| NixOS package and option search | Search portal | Central web search for packages and all available `configuration.nix` options |
| Nix Pills guide | Tutorial | In-depth course in the functional programming language and derivations |
| Nix flakes reference | Concept docs | Official documentation of the modern flake architecture and lock files |
| NixOS Community Discourse | Forum | Official discussion and support platform of the worldwide Nix community |

## Conclusion

Nix and NixOS require a fundamental shift of thinking from administrators: away from manual, stateful interventions on the server, toward a purely declarative, functional infrastructure

Source: https://admindocs.de/en/linux-beginners/nixos-nix-package-manager-guide

model.

Anyone who masters the initial learning curve is rewarded with a platform that completely eliminates a whole class of classic administration problems. Updates lose their risk through reliable, atomic rollbacks; development environments can be reproduced bit-exactly; and server configurations can be managed without gaps through version-control systems such as Git.

> **Practical tip for getting started:** Start with Nix as a standalone package manager on your familiar Linux distribution to test temporary environments with `nix shell` and first flakes. Once the concepts of derivations and store hashes are internalized, the step to a fully declarative NixOS server is seamless and safe.

If after declarative package management you want to take the next logical step to full infrastructure automation, the guide to Ansible: automation fundamentals for Linux administrators shows how you roll out configurations idempotently across heterogeneous server landscapes.

Source: https://admindocs.de/en/linux-beginners/nixos-nix-package-manager-guide