

fleet

NixOS cluster manager

01 Getting Started 2

02 Features 6

03 Reference 20

04 Examples 22

01

Getting Started

Fleet 3

Configuring Hosts 5

Fleet

An NixOS cluster deployment tool.

Advantages over existing configuration systems (NixOps/Morph)

- Modules can configure multiple hosts at once (i.e. for wireguard/kubernetes installation)
- Secrets can be securely stored in Git (no one except target hosts can decrypt them), automatically regenerated, reencrypted, etc.
- Automatic rollback on deployment failure, which will work as long as system is passing initrd stage

Flake example

```
{
  description = "My cluster configuration";
  inputs = {
    nixpkgs.url = "github:nixos/nixpkgs";
    fleet = {
      url = "github:CertainLach/fleet";
      inputs.nixpkgs.follows = "nixpkgs";
    };
    flake-parts.url = "github:hercules-ci/flake-parts";
  };
  outputs = inputs:
    inputs.flake-parts.lib.mkFlake {inherit inputs;} {
      imports = [inputs.fleet.flakeModules.default];

      fleetConfigurations.default = {
        nixos = {
          # Shared NixOS configuration for all hosts
        };

        imports = [
          ./wireguard
        ];

        hosts.controlplane-1 = {
          system = "x86_64-linux";
          nixos = {
            imports = [
              ./controlplane-1/hardware-configuration.nix
              ./controlplane-1/configuration.nix
            ];
          };
        };
      };
    };
}
```

Secret generator example

```
{config, ...}: {
  secrets = {
    gitlab-initial-root = {
      generator = {mkPassword}: mkPassword {};
      owner = "gitlab";
```

```
      group = "gitlab";
    };
    gitlab-secret = {
      generator = {mkPassword}: mkPassword {};
      owner = "gitlab";
      group = "gitlab";
    };
  };
  services.gitlab = {
    enable = true;
    initialRootPasswordFile = config.secrets.gitlab-initial-root.secret.path;
    secrets.secretFile = config.secrets.gitlab-secret.secret.path;
  };
}
```

Configuring Hosts

Each host in a fleet configuration represents a machine in your cluster.

Defining Hosts

```
fleetConfigurations.default = {
  hosts.web-1 = {
    system = "x86_64-linux";
    nixos = {
      imports = [
        ./web-1/hardware-configuration.nix
        ./web-1/configuration.nix
      ];
    };
  };

  hosts.web-2 = {
    system = "x86_64-linux";
    nixos = {
      imports = [
        ./web-2/hardware-configuration.nix
        ./web-2/configuration.nix
      ];
    };
  };
};
```

Host Systems

Every host must specify a `system` attribute. Supported systems:

- `x86_64-linux`
- `aarch64-linux`
- `armv7l-linux`
- `armv6l-linux`

Inline NixOS Configuration

NixOS configuration can be specified inline:

```
hosts.monitoring = {
  system = "x86_64-linux";
  nixos = {
    imports = [ ./monitoring/hardware-configuration.nix ];
    services.prometheus.enable = true;
    services.grafana.enable = true;
    networking.firewall.allowedTCPPorts = [ 3000 9090 ];
  };
};
```

02

Features

Multi-host Modules 7

Fleet Secrets Management System 8

Automatic Rollback 14

Fleet-usbd 15

Multi-host Modules

Fleet modules configure multiple hosts simultaneously. Unlike standard NixOS modules which configure a single machine, fleet modules have access to all hosts in the cluster.

Basic Module

```
# wireguard/default.nix
{ config, ... }: {
  hosts = builtins.mapAttrs (name: host: {
    nixos.networking.wireguard.interfaces.wg0 = {
      ips = [ host.config.wireguardIp ];
      peers = builtins.filter (p: p.name != name)
        (builtins.attrValues config.hosts);
    };
  }) config.hosts;
}
```

Using Modules

Modules are imported in the `fleetConfigurations` block:

```
fleetConfigurations.default = {
  imports = [
    ./wireguard
    ./monitoring
  ];

  hosts.node-1 = { ... };
  hosts.node-2 = { ... };
};
```

Multi-instance Modules

Modules can be parameterized and imported multiple times:

```
fleetConfigurations.default = {
  imports = [
    (import ./kubernetes { hosts = ["cp-1" "cp-2"]; })
    (import ./kubernetes { hosts = ["cp-3" "cp-4"]; })
  ];
};
```

Fleet Secrets Management System

Overview

Secret management system is a built-in way to deploy secrets to remote hosts, similar to agenix and other systems.

Secrets are encrypted using system's host ssh key (/etc/ssh/ssh_host_ed25519_key), which is not required to build the remote system/add secret to fleet configuration, fleet users are encrypting secrets using received public key instead, they don't need the root access to receive the public encryption key.

Per-host and shared secrets

Secrets come in two flavours; both are declared with the same `secrets.<name>` syntax, but in different scopes:

Per-host

declared inside a host's `nixos` block. Generated and encrypted only for that one host. Use for things like a machine-specific tokens, or per-peer WireGuard keys.

Shared

declared at the top-level fleet config with `expectedOwners = [...]`. The same value is encrypted independently for every owner. Each owning host must claim it with `secrets.<name>.generator = "shared"` in its `nixos` block (the assertion at modules/nixos/secrets.nix enforces this).

```
{ config, fleetLib, ... }:
{
  # Shared - top-level, with expectedOwners
  secrets."wg-psk" = {
    expectedOwners = [ "alpha" "beta" ];
    generator = fleetLib.mkBase64Bytes { count = 32; };
    # Optional: rotate when ownership changes
    regenerateOnOwnerAdded = true;
    regenerateOnOwnerRemoved = true;
  };

  hosts.alpha.nixos = { ... }: {
    # Per-host - lives only on alpha
    secrets."alpha-api-token".generator = fleetLib.mkPassword { };

    # Claim of the shared secret declared above
    secrets."wg-psk".generator = "shared";
  };

  hosts.beta.nixos = { ... }: {
    secrets."wg-psk".generator = "shared";
  };
}
```

See the WireGuard example for a fuller example combining both forms.

Built-in secret generators

Fleet ships a set of generators in [fleetLib] covering the common cases. Pass them directly as `generator` (shown here as shared secrets; the same calls work in a per-host `nixos.secrets.<name>.generator` block - just drop `expectedOwners`):

```
{ config, lib, fleetLib, ... }:
{
  secrets = {
    # Random 32-char ASCII password (single encrypted `secret` part)
    db-password = {
      expectedOwners = [ "db" ];
      generator = fleetLib.mkPassword { };
    };

    # 64-char password
    long-password = {
      expectedOwners = [ "db" ];
      generator = fleetLib.mkPassword { size = 64; };
    };

    # 32 random bytes, raw/hex/base64 encoded
    api-token = {
      expectedOwners = [ "api" ];
      generator = fleetLib.mkBytes { count = 32; encoding = "base64"; };
    };
    session-key = {
      expectedOwners = [ "api" ];
      generator = fleetLib.mkHexBytes { count = 32; };
    };
    wg-psk = {
      expectedOwners = [ "alpha" "beta" ];
      generator = fleetLib.mkBase64Bytes { count = 32; };
    };

    # Ed25519 keypair, two parts: plaintext `public`, encrypted `secret`
    # `noEmbedPublic` strips the embedded public half, leaving only the 32-byte private scalar
    garage-rpc = {
      expectedOwners = [ "storage" ];
      generator = fleetLib.mkEd25519 { encoding = "base64"; };
    };

    # X25519 keypair (e.g. WireGuard peer keys) - usually per-host, shown shared here for brevity
    wg-key = {
      expectedOwners = [ "alpha" ];
      generator = fleetLib.mkX25519 { encoding = "base64"; };
    };

    # RSA keypair, PEM-encoded; default size is 4096
    tls-key = {
      expectedOwners = [ "edge" ];
      generator = fleetLib.mkRsa { size = 2048; };
    };

    # OpenSSH keypair (ssh-keygen format): encrypted `secret`, plaintext `public`; default type is ed25519
    git-deploy-key = {
      expectedOwners = [ "ci" ];
      generator = fleetLib.mkOpenSshKey { };
    };

    # Tor onion service key, found with mkp224o on the machine running fleet: encrypted `secret`
    # (tor's hs_ed25519_secret_key file), plaintext `hostname` (`<address>.onion`).
    # `prefix` is a base32 string or a list of them; each character multiplies the search time by 32
    site-onion = {
      expectedOwners = [ "edge" ];
```

```
      generator = fleetLib.mkOnion { prefix = "site"; };
    };
  };
}
```

The onion address is known during evaluation, e.g. for links to it: `config.secrets.site-onion.hostname.data`. The key file is accepted by `services.tor.relay.onionServices.<name>.secretKey`, or by obice's `onion.key`: `services.obice.virtualHosts."example.org".onion.key = config.secrets.site-onion.secret.path;`.

Interactive generators

These prompt the operator on the host running `fleet` and are useful for secrets that can't be generated algorithmically (vendor API keys, manually-issued certificates, etc.). They are impure - the value you type is what gets encrypted.

Note that they also require GUI environment on the machine where the systems are built first time. This is intentional, because fleet console optput is quite cluttered during the deployment, and it is hard to properly show it. If this constraint bothers you - feel free to implement those however you like, all of those generators are not handled in a special way on fleet level: lib/default.nix.

```
{
  secrets = {
    # Single-line input via kdialog
    stripe-api-key = {
      expectedOwners = [ "billing" ];
      generator = fleetLib.mkAskPass {
        prompt = "Stripe live secret key";
      };
    };

    # Multi-line input opened in Kate; useful for pasted certificates/blobs
    upstream-ca = {
      expectedOwners = [ "edge" ];
      generator = fleetLib.mkAskFile {
        header = "Paste upstream CA bundle (PEM)";
        part = "cert";
      };
    };

    # Convenience wrapper that pre-populates an env-file template
    service-env = {
      expectedOwners = [ "app" ];
      generator = fleetLib.mkAskEnv {
        header = "Fill in upstream credentials";
        variables = [ "DATABASE_URL" "API_KEY" "SECRET_TOKEN" ];
      };
    };
  };
}
```

Operators and out-of-band delivery

Regular secrets are encrypted to the host key and decrypted by the host during activation. That doesn't work for files needed before activation (e.g. initrd secrets, which are read by the bootloader installation), or before the host has a key at all (`nixos-install`, the host key itself).

Such secrets can be owned by **operators** instead. Operator keys are age plugin recipients, e.g. age-plugin-gpg keys backed by gpg-agent or a smartcard:

```
{
  fleet.operatorKeys.lach = "age1gpg1a952kqarfahjjxuqp3sjru6sln7w3hjv8s02lm";
}
```

Operators are referenced in secret owners as `@name`: in `expectedOwners` of shared secrets, or in the `expectedOwners` of a host secret (the host itself is implied there). A secret owned by operators is encrypted **only** to operator keys. Its encrypted parts can't be decrypted by hosts, so they have to be delivered by fleet out-of-band, with `oobInstall` mapping part names to absolute paths:

```
{ config, fleetLib, ... }:
{
  secrets.initrd-ssh-host-ed25519 = {
    generator = fleetLib.mkOpenSshKey { };
    expectedOwners = [ "@lach" ];
    mode = "0600";
    oobInstall.secret = "/var/lib/initrd-ssh/ssh_host_ed25519_key";
  };
  boot.initrd.network.ssh.hostKeys = [ config.secrets.initrd-ssh-host-ed25519.secret.path ];
}
```

For out-of-band parts, `path` and `stablePath` point to the `oobInstall` path, and the parts are not installed into `/run/secrets`. During `fleet deploy` (`test`, `boot` and `switch`), before anything is activated, fleet checks every out-of-band part on the host, decrypts the outdated ones with the operator key on the machine fleet runs on, and writes them as root (owner and group of such secrets must be `root`). With `deploy_kind=nixos-install` they are written under `/mnt` before `nixos-install` runs.

Fleet keeps a root-only marker for each delivered part in `/var/lib/fleet/oob`, containing the hash of the encrypted value and of the written file. While both match, delivery is skipped, so operator keys are only needed when the secret or the file on the host changes. Anyone else with access to the host can keep deploying unrelated changes.

`fleet secret oob-install` delivers parts without deploying. With `--local --root /mnt` it writes to a disk of the host mounted on the machine fleet runs on, which is useful for installations done by hand.

Shared operator-owned secrets don't need reencryption when hosts are added to `expectedOwners`. When the operator set changes, fleet reencrypts the secret locally, which requires the key of one of the current operators.

Notes:

- The age plugin (`age-plugin-gpg`) has to be in `PATH` of fleet, and operator secrets are always generated on the machine fleet runs on (generators can't use `impureOn`), as encryption to gpg keys needs the local gpg-agent.
- Operator-owned secrets must deliver all of their encrypted parts out-of-band; plaintext parts can still be used normally.
- `fleet usbd` deployments can't deliver secrets, they rely on parts that were already delivered.

Host key

The host primary ssh key can be managed the same way, which allows to install hosts with all of their secrets already in place:

```
{ fleetLib, ... }:
{
  fleet.hostKeySecret = "ssh-host-ed25519";
  secrets.ssh-host-ed25519 = {
```

```
    generator = fleetLib.mkOpenSshKey { };
    expectedOwners = [ "@lach" ];
    mode = "0600";
    oobInstall = {
      secret = "/etc/ssh/ssh_host_ed25519_key";
      public = "/etc/ssh/ssh_host_ed25519_key.pub";
    };
  };
}
```

The `public` part of `fleet.hostKeySecret` becomes the host encryption key (`hosts.<name>.encryptionKey` in `fleet.nix`), and is evaluated before other secrets of the host. If the host already has a different encryption key, evaluation fails: either import the existing key in the generator, or remove the `encryptionKey` and regenerate secrets of the host.

Custom secret generator example

```
{
  secrets = {
    "my-secret" = {
      # Shared secret will be encrypted for those two hosts
      expectedOwners = [ "host1" "host2" ];
      # We can force secret to be regenerated when the owner was added
      regenerateOnOwnerAdded = true;

      # generator is a function from pkgs extended with helpers, so if you need something special, you can
just get
      # it from arguments, fleet will do callPackage for you, and provide you with the packages for the machine
      # that will be generating this secret
      generator = {
        mkImpureSecretGenerator,
        openssl,
      }:
        mkImpureSecretGenerator {
          # Impure secret generator executes arbitrary code on arbitrary host over SSH
          impureOn = "host3";

          # Secret may produce multiple parts, for example, for RSA keypair it would be secret and public part,
          # where only secret part should be encrypted. Non-encrypted parts are available as nix store path
and as
          # an expression, encrypted parts will be only available on owning machines after the deployment.
          parts.secret.encrypted = true;

          # In impure secret generator, script is a code that thould prepare the target directory structure
          # Target directory should contain one file for every declared part (parts argument in this attrset),
          # and may contain expires_at/created_at files if you want to use it with PKI.
          #
          # Secret parts should be encoded/encrypted to be consumed by fleet, you can use `gh` alias for that,
          # it will be prepared implicitly, and for arguments - look at the `fleet-generator-helper` section
of the docs.
          script = ''
            mkdir $out
            ${getExe openssl} rand -base64 16 | gh private -o $out/secret
          '';
        };
    };
  }
}
```

fleet-generator-helper

```
Usage: fleet-generator-helper <COMMAND>

Commands:
  public    Encode public part from stdin
  private   Encrypt private part from stdin
  help      Print this message or the help of the given subcommand(s)

Options:
  -h, --help  Print help
```

```
Encode public part from stdin

Usage: fleet-generator-helper public [OPTIONS] --output <OUTPUT>

Options:
  -o, --output <OUTPUT>

  -e, --encoding <ENCODING>
          Possible values:
          - raw:    Do not encode data, store as is
          - base64: Encode as base64 (with padding)
          - hex:    Encode as hex (without leading 0x)

          [default: raw]

  -h, --help
          Print help (see a summary with '-h')
```

```
Encrypt private part from stdin

Usage: fleet-generator-helper private [OPTIONS] --output <OUTPUT>

Options:
  -o, --output <OUTPUT>

  -e, --encoding <ENCODING>
          Possible values:
          - raw:    Do not encode data, store as is
          - base64: Encode as base64 (with padding)
          - hex:    Encode as hex (without leading 0x)

          [default: raw]

  -h, --help
          Print help (see a summary with '-h')
```

Automatic Rollback

Fleet automatically rolls back deployments that fail, preventing bricked machines.

How It Works

When deploying a new system configuration:

1. Fleet activates the new configuration
2. A health check verifies the system booted successfully
3. If the check fails, the previous configuration is restored

Rollback works as long as the system passes the initrd stage. Be careful with changes to root filesystem mounts — those are harder to recover from.

Deployment Process

```
# Deploy to all hosts
fleet deploy

# Deploy to specific host
fleet deploy web-1

# Deploy without rollback (not recommended)
fleet deploy --no-rollback
```

Fleet-usbd

Fleet-usbd

For airgapped deployments/system updates for low-tech users with limited internet access, it is better to have something more manual than pusher.

fleet-usbd is a simple update daemon, which watches for a USB stick with the system update to be inserted into the machine, finds its manifest on it, copies the closure from the USB stick, switches the current system generation, renames the manifest to *.done and reboots.

The stick is discovered by volume label; the label is only a discovery hint, not a trust anchor. Detection is a systemd device unit on /dev/disk/by-label/<label> with the daemon service bound to it (WantedBy=), no udisks dependency.

Update flow

1. Stick inserted, device unit starts the service, stick is mounted.
2. Daemon scans usbd-update-* files and tries to decrypt each; the one encrypted to this host's key is its manifest (the daemon has no naming secret, so it cannot derive its manifest name directly). The system name inside is checked as well.
3. If the manifest's system is already the current generation - noop. Guards against reinsertion when the .done rename previously failed.
4. Closure is copied from the stick. Hash verification is part of the copy; copying already-present paths is a noop.
5. New generation is set as the boot default (boot, not switch), so the existing on-boot health-check rollback applies.
6. Manifest is renamed to *.done.
7. Reboot.

Requirements

One file per store path

: it should be possible to rewrite the system update on USB stick in incremental fashion: unchanged store paths keep their exact file name and bytes, only added/removed paths change on the stick.

Deterministic, non-leaking names and ciphertext

: file names and file contents must be byte-stable across rewrites (otherwise incremental sync degrades to full rewrite), but must not leak store path names/contents. Mix a secret into the derivation: per-file key and file name are derived from (naming secret, narHash); the naming secret is project-wide, writer-side state in the file referenced by the `usbd.namingSecretFile` option (relative to the project directory, generated on first write). Hosts never need it - each manifest carries the keys for its paths - and because it is shared across hosts, a store path used by two hosts maps to the same file name and key, so shared sticks dedup naturally. Derivation must use narHash, not the store path: input-addressed paths can be rebuilt nondeterministically, yielding the same store path with different contents, and a same-name-different-bytes file breaks both immutability and size-only synchronization. Note age cannot be used for the path files - its encryption is randomized, so ciphertext would churn on every rewrite.

Manifest encrypted to the host ssh key

the manifest is encrypted the same way fleet already performs secret encryption (age, ssh-ed25519 host key recipient). It carries the system name and the per-path metadata listed in the copying section.

Manifest should include system name

update daemon should check if the system update is suitable for it.

User feedback via callbacks

how the user learns "update running / done / safe to remove" differs per client (screen, beep, LED). The daemon only exposes callback hooks for overriding; no built-in feedback.

Backwards-compatible updates

an old usbd must be able to apply updates produced by a newer fleet (forward compatibility from the daemon's point of view). Consequences: the on-stick format core (discovery label, manifest name and encryption, name/key derivation, file cipher, narinfo fields) is frozen from v1; the manifest is self-describing and old daemons ignore unknown fields, so evolution is additive-only. A truly breaking format change is still possible in two hops: every update ships the whole system including usbd itself, so the writer can emit the old format once to upgrade the daemon, then switch.

Copying store paths

The daemon exposes the stick as a standard nix binary cache on localhost and lets nix substitution perform the copy. This reuses nix machinery for hash verification, signature checking and skipping already-present paths - no diffing logic in the daemon.

1. Manifest carries, per store path: store path, narHash, narSize, references, build-time signature, on-stick file name and per-file key - everything needed to synthesize a narinfo.
2. Daemon binds an ephemeral HTTP listener on 127.0.0.1 serving the binary cache protocol: /nix-cache-info, synthesized <hash>.narinfo (Sig: from build time, URL: nar/<derived-name>, Compression: zstd) and /nar/<derived-name>, which streaming-decrypts the stick file (chunked AEAD) into the .nar.zst; nix does the decompression and NAR hash check itself.
3. The copy goes through the nix-eval FFI (Store::open on the localhost cache, copy_to into the local store), not a subprocess. A random token path prefix in the cache URL keeps other local processes from pulling decrypted NARs off the listener.
4. Signatures are verified by nix against trusted-public-keys; the deployer signing key is pinned there by the fleet module.
5. The daemon holds a temporary GC root on every update path for the duration of the copy (temp roots protect not-yet-valid paths individually, so intermediate paths are covered before the toplevel becomes reachable), so a concurrent GC cannot race it. The listener is shut down after the copy, then the normal flow continues: set boot generation, rename manifest, reboot.

Stick writing is the mirror image and also goes through the FFI: query path infos of the system closure, sign, dump each path to NAR, zstd-compress, encrypt with the derived per-file key, write under the derived name, write the age-encrypted manifest last.

Stick format

FAT32 (mkfs.vfat): universally supported and more robust against unclean ejects than exFAT. The 4 GiB file

size limit is handled by splitting large encrypted files into fixed-size chunks; chunk names are derived from (deployment secret, narHash, chunk index) and the manifest lists the chunk sequence per path. This is plain size-splitting, unrelated to dedup chunking. Filesystem is case-insensitive; derived file names should be lowercase.

Layout:

/usbd-update-<derive(hostname)>
: age-encrypted manifest for one host, renamed to *.done after a successful update. One manifest per host, so several hosts can share a stick.

/update-data/<xx>/<name>
: encrypted path chunks under derived names, sharded into 256 subdirectories by the first two name characters.

/gc/<derive(hostname)>
: plaintext list of chunk names referenced by that host's update (bare names, the shard prefix is implied). The writer refreshes its own list, then deletes only data files referenced by no list, so several hosts' updates can share one stick without the writers deleting each other's files.

/logs/<timestamp>-<random>.export.gpg
: journal dump in `journalctl -o export` format, written by the daemon after every run when `logRecipientFile` is set. Encrypted with OpenPGP to the operator key, so it can be decrypted with a smartcard. The random suffix avoids collisions between hosts sharing a stick (the daemon cannot derive its host id, and a plaintext hostname would leak).

FAT32 allows at most 65536 directory entries per directory; a 32-character name costs 4 entries (3 LFN + 1 short), so a flat directory would cap out at ~16000 files. The 256-way shard gives each subdirectory its own budget (~4M files total), and as a side effect keeps the volume root nearly empty, so even a FAT16-formatted stick with its fixed 512-entry root directory works.

Synchronization

Stick content should be synchronizable from http with a known tool that works on Windows. Flat content-addressed layout plus one manifest file satisfies this: rclone handles it (single portable exe, can be pre-installed on the stick itself with a .bat wrapper running `rclone sync --progress`; rclone has no native GUI and its experimental web GUI downloads assets from github on first run, so it is unusable here). Files are immutable under their derived names, so size-only comparison suffices. Sync should use --delete-before: rclone does not order transfers and by default deletes extraneous files last, which would require the stick to hold the old and new closure simultaneously. A partially synced stick is safe: the copy is hash-verified and the closure must be complete before the switch happens; the daemon should stat all manifest-referenced files upfront to report a partially synced stick cleanly.

Usage

On the host, enable the daemon and trust the update signing key:

```
{
  services.fleet-usbd = {
    enable = true;
    signPublicKeys = [ "client-1:..." ];
```

```
    # Optional user feedback script: receives copying/switching/done/noop/failed
    # hook = ./usbd-hook.sh;
    # Optional builtin feedback: blink connected Nitrokey 3 devices while working.
    # Blinking stopped and the machine rebooted = success; stopped without
    # a reboot = failed, send the stick back (it carries the encrypted logs).
    # The blink is the standard CTAPHID wink: white, firmware-fixed duration,
    # no colors - the nk3 admin app exposes nothing else.
    # nk3Wink = true;
    # Optional journal back-channel, encrypted to this OpenPGP key
    # logRecipientFile = ./operator.asc;
  };
}
```

The daemon is bound to the volume label (FLEETUSBD by default) via a systemd device unit; inserting the stick runs fsck and applies the update.

Writing the stick (expects an inserted stick with the matching label, mounted via udisks):

```
mkfs.vfat -n FLEETUSBD /dev/sdX1
fleet usbd write HOSTNAME
```

The signing keypair is generated once with `nix key generate-secret --key-name client-1`; the secret key file is referenced from the fleet configuration (or overridden with `--sign-key`):

```
{
  hosts.HOSTNAME.usbd.signingKeyFiles = [ "client-1.secret" ];
}
```

Paths are resolved relative to the fleet project directory and deliberately are not nix paths, so the secret key is never imported into the store.

Reading a journal dump from the stick:

```
gpg -d logs/2026-07-07-10-10-20-1a2b.export.gpg | systemd-journal-remote -o /tmp/metis.journal -
journalctl --file /tmp/metis.journal -b
```

Not a requirement

Rollback prevention

not a concern for any of the current clients.

Update authenticity in the daemon

handled by nix itself via store signatures during the copy; nothing to do on the usbd level.

Chunking of large paths

attic-style sub-path chunking would help large paths that change slightly, but conflicts with ease of synchronization; skipped.

Encrypted journal on the host

journald has no built-in journal encryption (FSS sealing is tamper-evidence only), which is why logs are exported and encrypted on the way to the stick instead.

GC of old generations

not a daemon problem.

Hiding cross-host linkage

the naming secret is shared project-wide for dedup, so an observer of a shared stick can tell which files two hosts have in common (not what they are). Per-host secrets would hide this at the cost of duplicating shared paths; not a concern for the current clients.

03

Reference

Migration Guide 21

Migration Guide

fleet.nix <unset> 0.1.0

Add version field

Set it to 0.1.0; This field specifies which version of fleet do you use for cluster management, breaking changes will also break this value to make sure you read this guide.

Move every secret part

Before it was only public and private, now it can be any number of parts.

In your fleet.nix file, look at every record like this:

```
gitlab-initial-root = {
   createdAt = "2024-03-01T15:54:32.983358495Z";
   public = "example";
   secret = "vp%d6wO#0#D2.../dgCA+v4Gf:YG";
};
```

And modify it as following:

```
gitlab-initial-root = {
   createdAt = "2024-03-01T15:54:32.983358495Z";
   public.raw = "<PLAINTEXT>example";
   secret.raw = ''
      <ENCRYPTED><Z85-ENCODED>
      vp%d6wO#0#D2.../dgCA+v4Gf:YG
   '';
};
```

Default encoding was also changed from `Z85` to `base64`. This conversion will be done by fleet automatically.

Update references to secrets in fleet/nixos configurations

Instead of `config.secrets.secret-name.secretPath` use `config.secrets.secret-name.secret.path`, instead of `config.secrets.secret-name.stableSecretPath` use `config.secrets.secret-name.secret.stablePath`, instead of `config.secrets.secret-name.public` use `config.secrets.secret-name.public.data`.

04

Examples

WireGuard Mesh Network 23

WireGuard Mesh Network

Full-mesh WireGuard VPN with automatic key generation, cross-host peer wiring, and hash-derived IPv6 ULA addresses.

Directory Structure

```
.
├── flake.nix
└── wireguard/
    └── default.nix
```

flake.nix

```nix
{...}: {
  outputs =
    inputs:
    inputs.flake-parts.lib.mkFlake { inherit inputs; } {
      imports = [
        inputs.fleet.flakeModule
      ];
      fleetConfigurations.default = {
        imports = [
          (import ./wireguard { iface = "testwg"; })
        ];
        cluster.wireguard.testwg = {
          hosts = [
            "bernard"
            "edgeworth"
          ];
          port = 51825;
        };
        hosts.bernard = {
          system = "x86_64-linux";
          network.externalIps = [
            "78.11.11.11"
          ];
        };
        hosts.edgeworth = {
          system = "aarch64-linux";
        };
      };
    };
}
```

Module is imported with a parameterized interface name (`testwg`), then configured declaratively via `cluster.wireguard.testwg`. Hosts without `network.externalIps` (like edgeworth) are treated as NAT/roaming — no endpoint is set for them, peers connect outbound only.

wireguard/default.nix

```nix
{ iface }:
{ config, lib, fleetLib, ... }:
let
  inherit (lib.attrsets) genAttrs optionalAttrs;
  inherit (lib.strings) substring;
  inherit (lib.lists)
```

```
    filter
    sort
    unique
    length
    elemAt
    ;
  inherit (lib.options) mkOption;
  inherit (lib.types) listOf str int;
  inherit (lib) hashString lessThan;

  cfg = config.cluster.wireguard.${iface};
  hostNames = sort lessThan cfg.hosts;
  hostHash = name: hashString "sha256" (name + iface);
  hostSubnet =
    name:
    let
      h = hostHash name;
    in
    "fd00:${substring 0 4 h}:${substring 4 4 h}:${substring 8 4 h}";
  hostIPv6 = name: "${hostSubnet name}::1";
in
{
  options.cluster.wireguard.${iface} = {
    hosts = mkOption {
      description = "Hostnames participating in this wireguard mesh";
      type = listOf str;
    };
    port = mkOption {
      description = "WireGuard listen port";
      type = int;
      default = 51824;
    };
  };

  config = {
    assertions = [
      {
        assertion = length (unique (map hostSubnet hostNames)) == length hostNames;
        message = "${iface}: hostname-derived IPv6 subnet collision detected";
      }
    ];
    secrets."${iface}-psk" = {
      expectedOwners = hostNames;
      generator = fleetLib.mkBase64Bytes { count = 32; };
    };

    hosts = genAttrs hostNames (name: {
      nixos =
        {
          config,
          nixosHosts,
          hosts,
          ...
        }:
        let
          peerNames = filter (n: n != name) hostNames;
        in
        {
          secrets."${iface}-key" = {
            generator = fleetLib.mkX25519 { encoding = "base64"; };
          };
          secrets."${iface}-psk" = {
            generator = "shared";
```

```
        };

        networking.wireguard.interfaces.${iface} = {
          ips = [ "${hostIPv6 name}/64" ];
          listenPort = cfg.port;
          privateKeyFile = config.secrets."${iface}-key".secret.path;

          peers = map (
            peerName:
            {
              publicKey = nixosHosts.${peerName}.secrets."${iface}-key".public.data;
              presharedKeyFile = config.secrets."${iface}-psk".secret.path;
              allowedIPs = [ "${hostSubnet peerName}::/64" ];
            }
            // optionalAttrs (hosts.${peerName}.network.externalIps or [ ] != [ ]) {
              endpoint = "${elemAt hosts.${peerName}.network.externalIps 0}:${toString cfg.port}";
            }
          ) peerNames;
        };

        networking.firewall.allowedUDPPorts = [ cfg.port ];
      };
    });
  };
  _file = ./default.nix;
}
```

How It Works

Multi-Instance Module

The outer function `{ iface }:` makes the module parameterizable — import it multiple times with different interface names to create independent WireGuard meshes with separate keys and subnets:

```
imports = [
  (import ./wireguard { iface = "internal"; })
  (import ./wireguard { iface = "management"; })
];
```

IPv6 ULA Addressing

Each host gets a unique `fd00::/64` subnet derived from `sha256(hostname + iface)`. No manual IP assignment needed — deterministic, collision-checked via assertion.

For example, `bernard` on interface `testwg` gets `fd00:1c86:78d4:dcef::1/64`.

Secrets

`${iface}-key`

Per-host X25519 keypair via `fleetLib.mkX25519 { encoding = "base64"; }`. Private key encrypted in fleet state, public key plaintext — readable by other hosts at build time through `nixosHosts.${peerName}.secrets."${iface}-key".public.data`.

`${iface}-psk`

Shared preshared key via `fleetLib.mkBase64Bytes { count = 32; }`. Defined at fleet level with `expectedOwners`, each host claims with `generator = "shared"`.

NAT Handling

Endpoint is set only for peers that have `network.externalIps`:

```
// optionalAttrs (hosts.${peerName}.network.externalIps or [ ] != [ ]) {
  endpoint = "...";
}
```

Hosts behind NAT initiate connections outward. Add `persistentKeepalive = 25;` if needed to maintain NAT mappings.

Deployment

Single command generates all secrets and deploys:

```
fleet --only bernard --only edgeworth deploy switch
```

No separate key generation step — fleet generates and provisions secrets automatically.

Fleet State After Deploy

The fleet state file records encryption keys, per-host keypairs (separate distributions), and shared PSK:

```
{
  hosts = {
    bernard.encryptionKey = "ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIJ25i6...";
    edgeworth.encryptionKey = "ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIELc1m...";
  };
  secrets = {
    testwg-key = [
      {
        owners = [ "bernard" ];
        createdAt = "2026-04-23T16:39:29.532714437Z";
        public.raw = "<PLAINTEXT>T7F6A+6jU87kz4c8GftttyU+xB0KhHIaplDV65psb3M=";
        secret.raw = "<ENCRYPTED><BASE64-ENCODED>...";
      }
      {
        owners = [ "edgeworth" ];
        createdAt = "2026-04-23T16:39:43.337788671Z";
        public.raw = "<PLAINTEXT>QgERRQnE+blQaVNMNiQZ+uk+HEOFKVTmvg4O+MqQ0hA=";
        secret.raw = "<ENCRYPTED><BASE64-ENCODED>...";
      }
    ];
    testwg-psk = {
      owners = [ "bernard" "edgeworth" ];
      createdAt = "2026-04-23T16:39:30.126077659Z";
      secret.raw = "<ENCRYPTED><BASE64-ENCODED>...";
    };
  };
}
```

Note how `testwg-key` is a list — each host has its own distribution with a unique keypair. `testwg-psk` is a single distribution shared by both hosts.

Verification

`wg show` on edgeworth:

```
interface: testwg
  public key: QgERRQnE+blQaVNMNiQZ+uk+HEOFKVTmvg4O+MqQ0hA=
  private key: (hidden)
  listening port: 51825

peer: T7F6A+6jU87kz4c8GftttyU+xB0KhHIaplDV65psb3M=
  preshared key: (hidden)
  endpoint: 78.11.11.11:51825
  allowed ips: fd00:1c86:78d4:dcef::/64
  latest handshake: 1 second ago
  transfer: 616 B received, 760 B sent
```

`wg show` on bernard:

```
interface: testwg
  public key: T7F6A+6jU87kz4c8GftttyU+xB0KhHIaplDV65psb3M=
  private key: (hidden)
  listening port: 51825

peer: QgERRQnE+blQaVNMNiQZ+uk+HEOFKVTmvg4O+MqQ0hA=
  preshared key: (hidden)
  endpoint: 109.11.11.11:51825
  allowed ips: fd00:cac2:10cb:6ee9::/64
  latest handshake: 7 minutes, 18 seconds ago
  transfer: 468 B received, 380 B sent
```

Public keys match between state file and `wg show`. edgeworth has no external IP in fleet config, yet bernard sees its endpoint — edgeworth initiated the connection outward through NAT.