Automagically assimilating NixOS machines into your Tailnet with Terraform

Read time in minutes: 33

Image generated by Eimis Anime Diffusion v1.0 -- a girl, Phoenix girl, fluffy hair, pixie cut, red hair, red eyes, chuunibyou, war, a hell on earth, Beautiful and detailed explosion, Cold machine, Fire in eyes, burning, Metal texture, Exquisite cloth, Metal carving, volume, best quality, Metal details, Metal scratch, Metal defects, masterpiece, best quality, best quality, illustration, highres, masterpiece, contour deepening, illustration, (beautiful detailed girl), beautiful detailed glow, green necklace, green earrings, kimono, fan, grin

For the sake of argument, let's say that you want to create all of your cloud infrastructure using Terraform, but you also want to use NixOS and Nix flakes. One of the main problems you will run into is the fact that Nix flakes and Terraform are both declarative and there's no easy way to shim Terraform states and Nix flake attributes. I think I've found a way to do this and today you're going to learn how to glue these two otherwise conflicting worlds together.

Requirements

In order to proceed with this tutorial as written, you will need to have the following things already set up:

- A Tailscale account.
- A Scaleway account.
- An amd64 Linux machine with Nix installed or an aarch64 Linux VM with either Apple Rosetta set up or `qemu-user` configured to let you run amd64 builds on an aarch64 host.
- An AWS Route 53 domain set up.
- A GitHub account.

<Mara> Pedantically, Scaleway can be replaced with any other server host. You can also remove all of the Tailscale-specific configuration. You can also use a different DNS provider. You may want to check the Terraform registry for your provider of choice. Most common and uncommon clouds *should* have a Terraform provider, but facts and circumstances may vary. GitHub can be replaced with any other git host.

I am also making the following assumptions when writing this tutorial:

- You have a Tailscale ACL tag named `tag:prod` that you can use Tailscale SSH to access.
- You have Nix flakes enabled.
- The device running all this is on your tailnet.

Making a new GitHub repo

One of the first things you will need to do is create a new GitHub repository. You can give it any name you like, but I named mine automagic-terraform-nixos.

Once you have created your repo, clone it locally:

```
git clone git@github.com:Xe/automagic-terraform-nixos.git
```

Create a `.gitignore` file with the following entries in it:

```
result
.direnv
.env
.terraform
```

Fetch credentials

Now that you have a new GitHub repository to store files in, you need to collect the various credentials that Terraform will use to control your infrastructure providers. For ease of use you will store them in a file called `.env` and use a shell command to load those values into your shell.

Variable	How to get it
`TAILSCALE_TAILNET`	Copy organization name from the admin panel.
`TAILSCALE_API_KEY`	Create an API key in the admin panel.
`SCW_ACCESS_KEY`	Create credentials in the console and copy the access key.
`SCW_SECRET_KEY`	Create credentials in the console and copy the secret key.

Next you will need to configure the AWS CLI, and by extension the default AWS API client. AWS has an excellent guide on doing this that I will not repeat here.

<Mara> If you don't have the AWS CLI installed, use `nix run nixpkgs#awscli2` in place of the `aws` command in that documentation.

Finally, set all of those variables into your environment with this command:

```
export $(cat .env |xargs -L 1)
```

<Mara> If you do this often, you may want to alias this command to `loaddotenv` in your shell profile.

Configuring Terraform

In your git repo, create a new file called `main.tf`. This is where your Terraform configuration is going to live. You can use any name you like, but the convention is to use `main.tf` for the "main" resources and any supplemental resources can live in their own files.

One of the best practices with Terraform is to store its view of the world in a non-local store such as Amazon S3. My state bucket is named `within-tf-state`, but your state bucket name will differ. Please see the upstream Terraform documentation for more information on how to establish such a state bucket.

<Mara> If you don't set up a state bucket, Terraform will default to storing is state in the current working directory. This state file will include generated secrets such as a Tailscale authkey. It is best to store this in S3 to avoid leaking secrets in your GitHub repository on accident.

```
# main.tf
terraform {
  backend "s3" {
    bucket = "within-tf-state"
```

```
    key    = "prod"
    region = "us-east-1"
  }
}
```

Now that you have the state backend set up, you need to declare the providers that this Terraform configuration will use. This will help ensure that Terraform is fetching the right providers from the right owners. Add this block of Terraform configuration right below the `backend "s3"` block you just declared:

```
# main.tf
terraform {
  # below the backend "s3" config

  required_providers {
    aws = {
      source = "hashicorp/aws"
    }

    cloudinit = {
      source = "hashicorp/cloudinit"
    }

    tailscale = {
      source = "tailscale/tailscale"
    }

    scaleway = {
      source = "scaleway/scaleway"
    }
  }
}
```

This configuration needs a few variables for things that are managed in the outside world. Scaleway requires that every resource is part of a "project", and you will need to put that project ID into your configuration. The Scaleway provider also allows us to have a default project ID, so you're going to put your project ID in a variable.

The Route 53 (AWS DNS) zone will also be put in its own variable.

```
# main.tf
variable "project_id" {
  type        = string
  description = "Your Scaleway project ID."
}

variable "route53_zone" {
  type        = string
  description = "DNS name of your route53 zone."
}
```

You can load your defaults into `terraform.tfvars`

```
# terraform.tfvars
project_id = "2ce6d960-f3ad-44bf-a761-28725662068a"
route53_zone = "xeserv.us"
```

Change your project ID and Route 53 zone name accordingly.

Once that is done, you can configure the Scaleway provider. If you want to have all resources default to being provisioned in Scaleway's Paris datacentre, you could use a configuration that looks like this:

```
# main.tf
provider "scaleway" {
  zone       = "fr-par-1"
  region     = "fr-par"
  project_id = var.project_id
```

```
}
```

Now that you have all of the boilerplate declared, you can get Terraform ready with the command `terraform init`. This will automatically download all the needed Terraform providers and set up the state file in S3.

```
terraform init
```

<Mara> If you don't already have terraform installed, you can run it without installing it by replacing `terraform` with `nix run nixpkgs#terraform` in any of these commands

Now that Terraform is initialized, you can import your Route 53 zone into your configuration by creating a `data` resource pointing to it:

```
# main.tf
data "aws_route53_zone" "dns" {
  name = var.route53_zone
}
```

To confirm that everything is working correctly, run `terraform plan` and see if it reports that it needs to create 0 resources:

```
terraform plan
```

If it reports that your DNS zone does not exist, please verify the configuration in `terraform.tfvars` and try again.

Create the Tailscale authkey for your new NixOS server using the `tailscale_tailnet_key` resource:

```
# main.tf
resource "tailscale_tailnet_key" "prod" {
  reusable      = true
  ephemeral     = false
  preauthorized = true
  tags          = ["tag:prod"]
}
```

Next you will need to create the cloud-init configuration for this virtual machine. Cloud-init is not exactly the best tool out there to manage this kind of assimilation, but it is widely adopted enough because it does the job well enough that you can rely on it.

There's many ways to create a cloud-init configuration in Terraform, but I feel that it's best to use the cloudinit provider for this. It will let you assemble a cloud-init configuration from multiple "parts", but this example will only use one part.

```
data "cloudinit_config" "prod" {
  gzip          = false
  base64_encode = false

  part {
    content_type = "text/cloud-config"
    filename     = "nixos-infect.yaml"
    content = sensitive(<<-EOT
#cloud-config
write_files:
- path: /etc/NIXOS_LUSTRATE
  permissions: '0600'
  content: |
    etc/tailscale/authkey
- path: /etc/tailscale/authkey
  permissions: '0600'
  content: "${tailscale_tailnet_key.prod.key}"
- path: /etc/nixos/tailscale.nix
```

```
  permissions: '0644'
  content: |
    { pkgs, ... }:
    {
      services.tailscale.enable = true;

      systemd.services.tailscale-autoconnect = {
        description = "Automatic connection to Tailscale";
        after = [ "network-pre.target" "tailscale.service" ];
        wants = [ "network-pre.target" "tailscale.service" ];
        wantedBy = [ "multi-user.target" ];
        serviceConfig.Type = "oneshot";
        path = with pkgs; [ jq tailscale ];
        script = ''
          sleep 2
          status="$(tailscale status -json | jq -r .BackendState)"
          if [ $status = "Running" ]; then # if so, then do nothing
            exit 0
          fi
          tailscale up --authkey $(cat /etc/tailscale/authkey) --ssh
        '';
      };
    }
runcmd:
  - sed -i 's:#.*$::g' /root/.ssh/authorized_keys
  - curl https://raw.githubusercontent.com/elitak/nixos-infect/master/nixos-infect | NIXOS_IMPORT=./tailscale.ni
EOT
    )
  }
}
```

At the time of writing, Scaleway doesn't have a prebaked NixOS image for creating new servers. One route you could take would be to make your own prebaked image and then customize it as you want, but I think it's more exciting to use nixos-infect to convert an Ubuntu install into a NixOS install. The `runcmd` block at the end of the cloud-config file tells cloud-init to run nixos-infect to rebuild the VPS into NixOS unstable, but you can change this to any other version of NixOS.

<Cadey> I personally use NixOS unstable on my servers because I value things being up to date and rolling release.

This sounds a bit arcane (and at some level it is), but at a high level it relies on the `/etc/NIXOS_LUSTRATE` file as described in the NixOS manual section on installing NixOS from another Linux distribution. You will use cloud-init in the Ubuntu side to plop down the tailscale authkey into `/etc/tailscale/authkey` on the target machine and then making sure it gets copied to the NixOS install by putting the path `etc/tailscale/authkey` into the `NIXOS_LUSTRATE` file.

One of the other things you *could* do here is install Tailscale and authenticate to its control plane in the Ubuntu side and then add `var/lib/tailscale` to the `NIXOS_LUSTRATE` file, but I feel that could take a bit longer than it already takes to infect the cloud instance with NixOS.

One of the features that nixos-infect has is the ability to customize the target NixOS install with arbitrary Nix expressions. This configuration puts a NixOS module into `/etc/nixos/tailscale.nix` that does the following:

- Enables Tailscale's node agent tailscaled
- Creates a systemd oneshot job (something that runs as a one-time script rather than a persistent service) that will authenticate the machine to Tailscale and set up Tailscale SSH

The oneshot will read the relevant authkey from `/etc/tailscale/authkey`, which is why it is moved over from Ubuntu.

<Cadey> Strictly speaking, you don't *have to* create a floating IP address to attach to the server, but it is the best practice to do this. If you replace your production host in the future it may be a good idea to have its IPv4 address remain the same. DNS propagation takes *forever*.

```
# main.tf
resource "scaleway_instance_ip" "prod" {}

resource "scaleway_instance_server" "prod" {
  type        = "DEV1-S"
  image       = "ubuntu_jammy"
```

```
    ip_id       = scaleway_instance_ip.prod.id
    enable_ipv6 = true
    cloud_init  = data.cloudinit_config.prod.rendered
    tags        = ["nixos", "http", "https"]
  }
```

Finally you can create `prod.your.domain` DNS entries with this configuration:

```
  resource "aws_route53_record" "prod_A" {
    zone_id = data.aws_route53_zone.dns.zone_id
    name    = "prod"
    type    = "A"
    records = [scaleway_instance_ip.prod.address]
    ttl     = 300
  }

  resource "aws_route53_record" "prod_AAAA" {
    zone_id = data.aws_route53_zone.dns.zone_id
    name    = "prod"
    type    = "AAAA"
    records = [scaleway_instance_server.prod.ipv6_address]
    ttl     = 300
  }
```

<Mara> The reason behind creating two separate DNS entries is an exercise for the reader.

```
{
  inputs = {
    nixpkgs.url = "nixpkgs/nixos-unstable";
    flake-utils.url = "github:numtide/flake-utils";
  };

  outputs = { self, nixpkgs, flake-utils }:
    let
      mkSystem = extraModules:
        nixpkgs.lib.nixosSystem rec {
          system = "x86_64-linux";
          modules = [
            # bake the git revision of the repo into the system
            ({ ... }: { system.configurationRevision = self.sourceInfo.rev; })
          ] ++ extraModules;
        };
    in flake-utils.lib.eachSystem [ "x86_64-linux" "aarch64-linux" ] (system:
      let pkgs = import nixpkgs { inherit system; };
      in rec {
        devShells.default =
          pkgs.mkShell { buildInputs = with pkgs; [ terraform awscli2 ]; };
      }) // {
        # TODO: put nixosConfigurations here later
      };
}
```

The outputs function may look a bit weird here, but we're doing two things with it:

- Creating a development environment (devShell) with terraform and the AWS cli installed for both amd64 and aarch64 linux systems
- Setting up for defining `nixosConfigurations` dynamically

It's also worth noting that the `mkSystem` function defined at the top of the outputs function will bake in the git commit of the custom configuration into the resulting NixOS configuration. This will make it impossible to deploy changes that are not committed to git.

Gluing the two worlds together

Now you can do the exciting bit: glue the two worlds of Nix flakes and Terraform together using the `local-exec` provisioner and a shell script like this:

```
#!/usr/bin/env bash

set -e
[ ! -z "$DEBUG" ] && set -x

USAGE(){
    echo "Usage: `basename $0` <server_name>"
    exit 2
}

if [ -z "$1" ]; then
    USAGE
fi

server_name="$1"
public_ip="$2"

ssh_ignore(){
    ssh -o UserKnownHostsFile=/dev/null -o StrictHostKeyChecking=no $*
}

ssh_victim(){
    ssh_ignore root@"${public_ip}" $*
}

mkdir -p "./hosts/${server_name}"
echo "${public_ip}" >> ./hosts/"${server_name}"/public-ip

until ssh_ignore "root@${server_name}" uname -av
do
    sleep 30
done

scp -o UserKnownHostsFile=/dev/null -o StrictHostKeyChecking=no "root@${server_name}:/etc/nixos/hardware-configu

rm -f ./hosts/"${server_name}"/default.nix
cat <<-EOC >> ./hosts/"${server_name}"/default.nix
{ ... }: {
  imports = [ ./hardware-configuration.nix ];

  boot.cleanTmpDir = true;
  zramSwap.enable = true;
  networking.hostName = "${server_name}";
  services.openssh.enable = true;
  services.tailscale.enable = true;
  networking.firewall.checkReversePath = "loose";
  users.users.root.openssh.authorizedKeys.keys = [
    "ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIM6NPbPIcCTzeEsjyx0goWyj6fr2qzcfKCCdOUqg0N/v" # alrest
  ];
  system.stateVersion = "23.05";
}
EOC

git add .
git commit -sm "add machine ${server_name}: ${public_ip}"
nix build .#nixosConfigurations."${server_name}".config.system.build.toplevel

export NIX_SSHOPTS='-o UserKnownHostsFile=/dev/null -o StrictHostKeyChecking=no'
nix-copy-closure -s root@"${public_ip}" $(readlink ./result)
ssh_victim nix-env --profile /nix/var/nix/profiles/system --set $(readlink ./result)
ssh_victim $(readlink ./result)/bin/switch-to-configuration switch

git push
```

Add the provisioner script to your `scaleway_instance_server` by adding this block of configuration right at the end of its definition:

```
# main.tf
resource "scaleway_instance_server" "prod" {
```

```
    # ...

    provisioner "local-exec" {
      command = "${path.module}/assimilate.sh ${self.name} ${self.public_ip}"
    }

    provisioner "local-exec" {
      when    = destroy
      command = "rm -rf ${path.module}/hosts/${self.name}"
    }
  }
```

This will trigger the `assimilate.sh` script to run every time a new instance is created and delete host-specific configuration when an instance is destroyed.

Then you can hook up the `nixosConfigurations` output to the folder structure that script creates by adding the following configuration to your `flake.nix` file:

```
  }) // {
    nixosConfigurations = let hosts = builtins.readDir ./hosts;
    in builtins.mapAttrs (name: _: mkSystem [ ./hosts/${name} ]) hosts;
  };
```

This works because I am making hard assumptions about the directory structure of the `hosts` folder in your git repository. When I wrote this configuration, I assumed that the `hosts` folder would look something like this:

```
  hosts
  └── tf-srv-naughty-perlman
      ├── default.nix
      ├── hardware-configuration.nix
      └── public-ip
```

Each host will have its own folder named after itself with configuration in `default.nix` and that will point to any other relevant configuration (such as `hardware-configuration.nix`). Because this directory hierarchy is predictable, you can get a listing of all the folders in the `hosts` directory using the `builtins.readDir` function:

```
  nix-repl> builtins.readDir ./hosts
  { tf-srv-naughty-perlman = "directory"; }
```

Then you can use `builtins.mapAttrs` to loop over every key->value pair in the attribute set that `builtins.readDir` returns and convert the hostnames into NixOS system definitions:

```
  nix-repl> hosts = builtins.readDir ./hosts
  nix-repl> builtins.mapAttrs (name: _: ./hosts/${name}) hosts
  { tf-srv-naughty-perlman = /home/cadey/code/Xe/automagic-terraform-nixos/hosts/tf-srv-naughty-perlman; }
```

<Mara> The rest of this is an exercise for the reader.

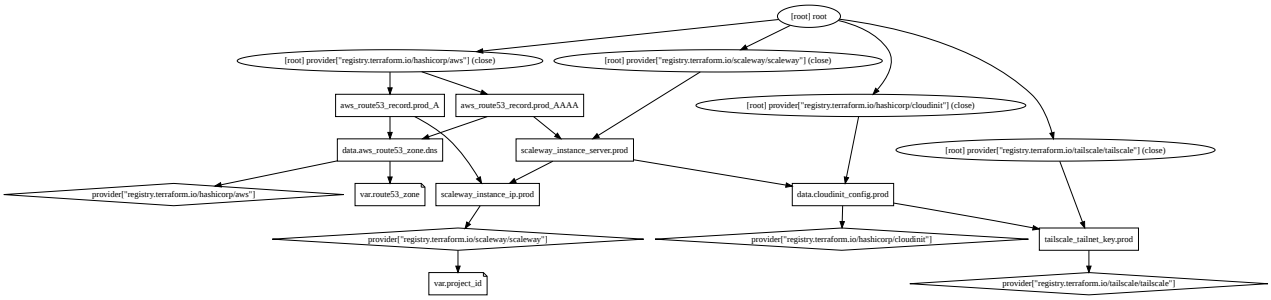
Creating your server

Finally, now that everything is put into place you can create your server using `terraform apply`:

```
  terraform apply
```

Terraform will print off a list of things that it thinks it needs to do. Please read this over and be sure that it's proposing a plan that makes sense to you. When you are satisfied that Terraform is going to do the correct thing, follow the instructions it gives you. If you are not satisfied it's going to do the correct thing, press control-c.

Let it run and it will automatically create all of the infrastructure you declared in `main.tf`. The entire graph of infrastructure should look something like this:



<Mara> If that is too small for you, click here. There is a lot going on in the graph because Terraform lists everything and its ultimate dependents.

You can SSH into the server using this command:

```
ssh root@generated-server-name
```

Manually pushing configuration changes

There are many NixOS tools that you can use to push configuration changes like deploy-rs, but you can also manually push configuration changes by following these three steps:

- Build the new system configuration for the target machine
- Copy the system configuration to the target machine
- Activate that new configuration

You can automate these steps using a script like the following:

```
#!/usr/bin/env bash
# pushify.sh

set -e
[ ! -z "$DEBUG" ] && set -x

# validate arguments
USAGE(){
    echo "Usage: `basename $0` <server_name>"
    exit 2
}

if [ -z "$1" ]; then
    USAGE
fi

server_name="$1"
public_ip=$(cat ./hosts/${server_name}/public-ip)

ssh_ignore(){
    ssh -o UserKnownHostsFile=/dev/null -o StrictHostKeyChecking=no $*
}

ssh_victim(){
    ssh_ignore root@"${public_ip}" $*
}

# build the system configuration
nix build .#nixosConfigurations."${server_name}".config.system.build.toplevel

# copy the configuration to the target machine
export NIX_SSHOPTS='-o UserKnownHostsFile=/dev/null -o StrictHostKeyChecking=no'
nix-copy-closure -s root@"${public_ip}" $(readlink ./result)

# register it to the system profile
ssh_victim nix-env --profile /nix/var/nix/profiles/system --set $(readlink ./result)
```

```
  # activate the new configuration
  ssh_victim $(readlink ./result)/bin/switch-to-configuration switch
```

You can use it like this:

```
  ./pushify.sh generated-server-name
```

Rollbacks

To roll back a configuration, SSH into the server and run `nixos-rebuild --rollback switch`.

Setting up automatic updates

One of the neat and chronically underdocumented features of NixOS is the system.autoUpgrade module. This allows a NixOS system to periodically poll for changes in its configuration or updates to NixOS itself and apply them automatically. It will even reboot if the kernel was upgraded.

In order to set it up, create a folder named `common` and put the following file in it:

```
  # common/default.nix
  { ... }: {
    system.autoUpgrade = {
      enable = true;
      # replace this with your GitHub repo
      flake = "github:Xe/automagic-terraform-nixos";
    };
  }
```

Then add `./common` to the list of modules in the `mkSystem` function like this:

```
  mkSystem = extraModules:
    nixpkgs.lib.nixosSystem rec {
      system = "x86_64-linux";
      modules = [
        ./common
        ({ ... }: { system.configurationRevision = self.sourceInfo.rev; })
      ] ++ extraModules;
    };
```

Commit these changes to git and deploy the configuration to your server:

```
  git add .
  git commit -sm "set up autoUpgrade"
  git push
  ./pushify.sh generated-server-name
```

Your NixOS machines will automatically pull changes to your GitHub repository once per day somewhere around `04:40` in the morning, local time. You can manually trigger this by running the following command:

```
  ssh root@generated-server-name
  systemctl start nixos-upgrade.service
  journalctl -fu nixos-upgrade.service
```

Exercises for the reader

This tutorial has told you everything you need to know about setting up new NixOS servers with Terraform. Here are some exercises that you can do to help you learn new and interesting things about configuring your new NixOS machines:

- Set up backups to borgbase.
- Set up encrypted secret management with agenix.
- Create an AWS IAM user for your machine and copy the secret files to it. How would you do that programmatically with a new machine? Hint: `NIXOS_LUSTRATE` can help! Use that for Let's Encrypt.
- Try some of the services listed in the NixOS manual. How would you expose one of them over Tailscale?

- How would you make an instance on Vultr using this Terraform manifest? How about Digital Ocean?
- How would you attach a VPC to your server and expose it to your other machines as a subnet router with Tailscale?
- Set up a Pleroma server. Be sure to use Let's Encrypt to get an HTTPS certificate!

I hope this was enlightening! Enjoy your new servers and have fun exploring things in NixOS!

▶ Share on Mastodon

This article was posted on M12 07 2022. Facts and circumstances may have changed since publication Please contact me before jumping to conclusions if something seems wrong or unclear.

Series: nix-flakes

Tags: `Terraform` `NixOS` `Scaleway`

- @cadey spicy lisp without the DSL composition though! But I will stop tooting this horn and finish reading your post :)
- @cadey I am more of a Guix girl but I will read it nonetheless because you wrote it!!
- @cwebber i'm sure there's things you can learn from it ^^
- @cadey
> One of the main problems you will run into is the fact that Nix flakes and Terraform are both declarative and there's no easy way to shim Terraform states and Nix flake attributes. I think I've found a way to do this and today you're going to learn how to glue these two otherwise conflicting worlds together. DSL composition? Sounds like a job for a lisp ;)
- @cwebber nix is basically Haskell and Haskell is just spicy lisp
- @cadey oh no
- @kouhai this has very cursed things explained very plainly too

The art for Mara was drawn by Selicre.

The art for Cadey was drawn by ArtZorea Studios.

Like what you see? Donate on Patreon like these awesome people!

Looking for someone for your team? Take a look here.

See my salary transparency data here.

Served by /nix/store/l26lms3paxdy3cm5bf93zlz298vas4s5-xesite-3.0.0/bin/xesite, see source code here.