Nix party tricks

Building EC2s, images, and lambda packages with NixOS

Alexander Flurie

<2022-07-12 Tue>

Pittsburgh AWS User Group

Outline

Preamble

Demo Overview

Extremely abbreviated intro to nix

First party trick: nix for managing development environments

Second party trick: nix for managing ec2s

Third party trick: nix for managing lambda runtimes

Preamble

Nix is magic

Surprise, this is a magic show!

Nix is a special kind of magic for specifying pretty much any output you could want.

Nothing up my sleeve. . .

Behold, a fresh VM.

I am clicking through these things because I am very serious.

. . . but cards on the table

There's always something to install when you start.

For those playing along at home

This text is expository, written for flavor and background

This text highlights an important definition

Satoshi Nakamoto: last known alias of Charles Ponzi

This text is instructional

Green means go! Do as I say.

Demo Overview

Purpose

- survey of a bunch of common problems and demonstrate solutions with nix
- whirlwind tour of some great nix ecosystem tooling
- code is public: `https://github.com/flurie/nix-party-tricks`

Let's install nix!

Installing nix

- Go to `https://nixos.org`
- select Download
- Follow multi-user installation instructions (unless you're on something weird like WSL)

And let’s install some things to make our lives easier (and this demo shorter)

Rosetta 2

```
softwareupdate --install-rosetta
```

git

```
nix-env -iA nixpkgs.git
```

cachix

```
nix-env -iA nixpkgs.cachix
```

add some stuff to `/etc/nix/nix.conf`

```
experimental-features = nix-command flakes
trusted-users = root $(whoami)
```

explicitly add cachix cache

```
cachix use flurie && sudo pkill nix-daemon
```

Extremely abbreviated intro to nix

Nix the First: Language

Main features:

- functional
- dynamic
- lazy
- base language is tiny
- Haskell influence (though much divergence since)

Nix the First: Language (con't)

Quirky type system:

- strings have native multiline support
- URIs
- paths (relative and absolute)
- no advanced objects, everything is a set (map)
- first-class functions

Nix the Second: Package Manager

nixpkgs

- Fundamental unit: the derivation
- Built with and extends Nix language
- Largest, most active package repository of its kind
- Many smaller ecosystems, especially by language (2nix)

Nix the Second: Package Manager (con't)

Figure 1: The Dirty Secret

Nix the Third: Linux Distribution

NixOS

- Built on top of nixpkgs and systemd
- Familiar to users of gentoo and arch
- Adds in modules for system-level configurability

Nix the Fourth: *misc* tooling

Tools worth knowing

- home-manager: nix for `$HOME`
- nix-darwin: nix for macOS
- cachix: arbitrary caching for nix derivations
- Hercules CI: CI/CD for nix derivations

First party trick: nix for managing development environments

Misc tools for environment management

Tools we will use in this section

- direnv: automate environment switching in shell
- devshell: manage all your development tools per-project with a simple configuration file

Let's install direnv!

Installing direnv

- Go to `https://direnv.net/#basic-installation`
- Follow the NixOS instructions (because I'm not installing Homebrew, boo!) for non-NixOS systems
- Hook direnv into shell

Let’s grab the code. . .

Clone me on GitHub

```
git clone
https://github.com/flurie/nix-party-tricks.git
```

. . . and then let the magic take hold

direnv holds a lot of power, so be careful with what you allow.

Using nix with direnv provides an additional level of security.

Time to take the ride.

```
direnv allow
```

Tour our new powers

Figure 2: I'm in devshell! I'm in normal shell!

A note about creds

Be safe

- Never store credentials in a long-lived plaintext config file!
- use `credential_process` to grab creds safely

```
# ~/.aws/credentials

[default]
credential_process = access_keys_from_csv
```

Enter AWS with train

Set the stage for more magic

```
cp -r "$PRJ_ROOT"/support/.aws ~/.aws
```

You can try this at home, but don't leave the files sitting around.

```
Log in to AWS console

Create new programmatic IAM credentials

Download the csv to our devshell root
```

Time to test the thing out

Putting it all together

```
aws sts get-caller-identity
```

Second party trick: nix for managing ec2s

Preamble

terraform to stand up the host

```
cd $PRJ_ROOT/terraform/ec2
terraform init
terraform apply
```

Misc tools for deployment management

Tools we will use in this section

- cachix: arbitrary caching for nix derivations
- deploy-rs: deploy NixOS to anywhere from anywhere
- nixos-generators: generate NixOS machine images of any kind

NixOS on AWS three ways

#1: ec2 user data

NixOS on AWS three ways - #1

```
# main.tf
resource "aws_instance" "nixos" {

  # ...some parts omitted

  root_block_device {
    # need this to be big enough to build things
    volume_size = 20
  }

  user_data = <<END
### https://nixos.org/channels/nixos-22.05 nixos

{ config, pkgs, modulesPath, ... }:
{
  # nix uses same string interpolation as terraform, so we must escape it here
  imports = [ "$${modulesPath}/virtualisation/amazon-image.nix" ];
  ec2.hvm = true;
  system.stateVersion = "22.05";
  environment.systemPackages = with pkgs; [ nix-direnv direnv git ];
  networking.hostName = "nixos-aws";
}
END
}
```

NixOS on AWS three ways - #1

We can now enter the machine.

terraform output into ssh config file + hosts file line?

Make sure to use the IP given by terraform.

```
ssh -i /tmp/nixos-ssh.pem root@{IP}
```

NixOS on AWS three ways - #1

Let’s pull down the party tricks repo here as well. . .

```
git clone
https://github.com/flurie/nix-party-tricks.git
```

NixOS on AWS three ways - #1

...and activate the devshell!

```
cd nix-party-tricks && direnv allow
```

First way done!

NixOS on AWS three ways

#2: deploy-rs

NixOS on AWS three ways - #2

```
deploy = {
  nodes = {
    "aws" = {
      sshUser = "root";
      sshOpts = [ "-i" "/tmp/nixos-ssh.pem" ];
      hostname = "nixos-aws";
      profiles.hello = {
        path = deploy-rs.lib.x86_64-linux.activate.custom
          nixpkgs.legacyPackages.x86_64-linux.hello "./bin/hello";
      };
      profiles.system = {
        path = deploy-rs.lib.x86_64-linux.activate.nixos
          self.nixosConfigurations.aws;
      };
    };
  };
};
```

NixOS on AWS three ways - #2

let's make sure it's in our /etc/hosts for later

```
sudo echo "{IP}  nixos-aws" >> /etc/hosts
```

NixOS on AWS three ways - #2

copy the key over so we can deploy from the machine, then shell in

```
scp -i /tmp/nixos-ssh.pem /tmp/nixos-ssh.pem root@nixos
ssh -i /tmp/nixos-ssh.pem root@nixos-aws
```

NixOS on AWS three ways - #2

First deploy: “hello world”

```
# the -s skips the checks, saving us some time
# don't do this at home
cd nix-party-tricks
deploy .#aws.hello -s
```

NixOS on AWS three ways - #2

Second deploy: NixOS system running nginx

```
{
  services.nginx = { enable = true; };
  networking.firewall.allowedTCPPorts = [ 80 ];
}
```

Let’s deploy!

```
deploy .#aws.system -s
```

NixOS on AWS three ways - #2

Now we should get the nginx splash page in a browser

```
visit http://nixos-aws in a browser
```

Second way done!

NixOS on AWS three ways

#3: nixos-generators

NixOS on AWS three ways #3

```
packages.x86_64-linux.awsImage = let system = "x86_64-linux";
      in nixos-generators.nixosGenerate {
        pkgs = nixpkgs.legacyPackages.${system};
        modules = [
          # new hostname for new machine
          networking.hostName = "nixos-aws-ami";
          # mostly stuff you've seen before...
            services.nginx = {
              enable = true;
              virtualHosts.${networking.hostName} = {
                # except now we're serving something special
                root = "${self.packages."${system}".default}/www";
              };
            };
        ];
        format = "amazon";
};
```

NixOS on AWS three ways - #3

Let's use our shiny new ec2 for this!

But before we do, let's make our user creds available for the sake of simplicity.

```
# from our local
scp -i /tmp/nixos-ssh.pem ./$(whoami)_accessKeys.csv \
    root@nixos-aws:~/nix-party-tricks/

ssh -i /tmp/nixos-ssh.pem root@nixos-aws
```

NixOS on AWS three ways - #3

Now let's build the image!

```
cd $PRJ_ROOT/terraform/ami
nix build .#awsImage
terraform init
terraform apply
```

If we're lucky, it will hit the cached version of my image and spare us.

If we're not, I made a trivial change at some point and never cached it, requiring a rebuild.

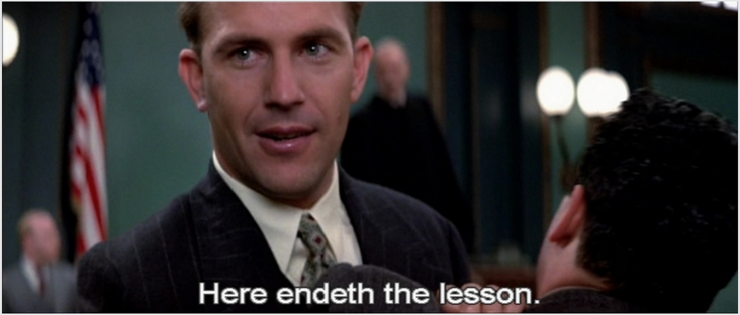
Declarative build systems are ruthless.

NixOS on AWS three ways - #3

Now we should get something special in a browser

```
visit http://nixos-aws-ami in a browser
```

Third way done!



Third party trick: nix for managing lambda runtimes

Preamble

We will have to manage some stuff by hand.

Terraform *really* doesn't want to manage container images. Providers that can make it happen expect to build with docker.

Container image tools

Tools we will use in this section

- docker-tools: nixpkgs native OCI-compatible image builder
- colima: no-fuss container runtimes for macOS and Linux

Create ECR repo

```
aws ecr create-repository \
    --repository-name nix
```

Lambda One

The setup

```
let
  pythonEnv = pkgs.python39.withPackages (ps: with ps; [ awslambdaric ]);
  entrypoint = pkgs.writeScriptBin "entrypoint.sh" ''
    #!${pkgs.bash}/bin/bash
    if [ -z "$AWS_LAMBDA_RUNTIME_API" ]; then
      exec ${pkgs.aws-lambda-rie}/bin/aws-lambda-rie ${pythonEnv}/bin/python3 -m awslambdaric $@
    else
      exec ${pythonEnv}/bin/python3 -m awslambdaric $@
    fi
  '';
  app = pkgs.writeScriptBin "app.py" ''
    #!${pythonEnv}/bin/python3

    import sys

    def handler(event, context):
        return "Hello from AWS Lambda using Python" + sys.version + "!"
  '';
in
# ...
```

Lambda One (con't)

The image

```
pkgs.dockerTools.buildLayeredImageWithNixDb {
  name = "nix-lambda";
  tag = "latest";
  contents = [ pkgs.bash pkgs.coreutils pythonEnv app pkgs.aws-lambda-rie ];
  config = {
    Entrypoint = [ "${entrypoint}/bin/entrypoint.sh" ];
    Cmd = [ "app.handler" ];
    WorkingDir = "${app}/bin";
  };
}
```

Build and push

Build the image

```
# starting on the build machine
nix build .#lambdaSimple
# all nix builds get a symlink to ./result by default.
# since this is a raw archived OCI image, we can load the path directly.
docker load < result
```

Push the image

Docker login to ECR

```
aws ecr get-login-password --region us-east-2 | \
    docker login --username AWS --password-stdin \
    "$(aws sts get-caller-identity | jq -r '.Account')
```

now tag and push to ECR

```
scripts/tag_and_push_lambda.sh
```

```
#! /usr/bin/env nix-shell
#! nix-shell -i bash -p jq
docker tag "$(docker images nix-lambda --format '{{.ID}}')" \
  "$(aws sts get-caller-identity | jq -r '.Account').dkr.ecr.us-east-2.amazonaws.com/nix:latest"
docker push \
  "$(aws sts get-caller-identity | jq -r '.Account').dkr.ecr.us-east-2.amazonaws.com/nix:latest"
```

Now terraform the rest

More terraform

```
cd $PRJ_ROOT/terraform/lambda
terraform init
terraform apply
```

See the results

Calling our function

```
curl ${function_url}
```

Lambda Two

Let's add some real packages

The setup

```
mangum = with pkgs.python39.pkgs;
  buildPythonPackage rec {
    pname = "mangum";
    version = "0.15.0";

    src = fetchPypi {
      inherit pname version;
      sha256 = "sha256-EuhIBhmLI7vVpUubacGu88YhdzRyGbtXyOwRS4prhTc=";
    };

    buildInputs = [ typing-extensions ];

    pythonImportsCheck = [ "mangum" ];

    meta = with pkgs.lib; {
      description = "AWS Lambda support for ASGI applications";
      homepage = "https://github.com/jordaneremieff/mangum";
      license = licenses.mit;
      maintainers = with maintainers; [ ];
    };
  };
pythonEnv =
  pkgs.python39.withPackages (ps: with ps; [ awslambdaric mangum fastapi ]);
```

Lambda Two (con’t)

The app

```
app = pkgs.writeScriptBin "app.py" ''
  #!${pythonEnv}/bin/python3

  from fastapi import FastAPI
  from mangum import Mangum

  app = FastAPI()


  @app.get("/")
  def read_root():
      return {"Hello": "World"}


  @app.get("/items/{item_id}")
  def read_item(item_id: int, q: str = None):
      return {"item_id": item_id, "q": q}

  handler = Mangum(app, lifespan="off")
'';
```

Lambda Two - Up and running

```
nix build .#lambdaApi
docker load < result
tag_and_push_lambda
```

And now we cheat

For the sake of simplicity

Let's just refresh the image in the console.

Lambda Two - Testing

Let's try it in a browser

- / should get us a hello world
- `/docs` should get us the fastapi swagger
- `/items/foo` should get us some stuff back

Lambda Three

Let's do something with our packages. The setup is the same, but the app is different.

The setup

```
 app = pkgs.writeScriptBin "app.py" ''
   #!${pythonEnv}/bin/python3

   from fastapi import FastAPI
   from fastapi.staticfiles import StaticFiles
   from mangum import Mangum

   app = FastAPI()


   @app.get("/")
   def read_root():
       return {"Hello": "World"}

   app.mount("/nixdocs", StaticFiles(directory="${nixPartyTricksDocs}/www", html=True),
       name="nixdocs")

   handler = Mangum(app, lifespan="off")
 '';
# ...
```

Lambda Two - Up and running

```
nix build .#lambdaDocs
docker load < result
tag_and_push_lambda
```

I will repeat and cheat once again

Just do this

Refresh the image once more.

Lambda Three - Testing

Let's try it in a browser

- `/nixdocs/index.html` should have something very interesting for us. I wonder what it could be?

That's it. That's the talk.

Figure 3: Any questions?