# Updating NixOS local VMs

Posted on 2019-12-01 by eon@patapon.info, Tags: #nix #nixos

So you have deployed a NixOS server and want to test a configuration change. How cool would it be to test it locally before sending it over the wire and hope for the best? Well NixOS provides an easy way to start a local VM using your server configuration...

## Intro

This post will demonstrate how to build and run a VM from a NixOS configuration and then update the configuration of the running VM on the fly.

This can be useful for different use-cases:

- testing the migration to a new config
- testing the upgrade to a newer nixpkgs version
- iterate faster when developping a NixOS module

The work machine doesn't need to be NixOS, it can be any system with nix installed.

## Building a local VM

Let's say you have your server `configuration.nix` file at hand and you want to use that to build a local vm.

Let's take a simple example with this file:

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

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

  boot.loader.grub.enable = true;
  boot.loader.grub.version = 2;
  boot.loader.grub.device = "/dev/vda";

  networking.hostName = "fripon";
  networking.domain = "patapon.info";

  i18n.defaultLocale = "en_US.UTF-8";
  time.timeZone = "Europe/Paris";

  services.openssh.enable = true;
  services.openssh.permitRootLogin = "yes";
  services.timesyncd.enable = true;

  environment.systemPackages = with pkgs; [
    vim htop dnsutils inetutils
  ];
```

```
  users.mutableUsers = false;
  users.users.root.hashedPassword =
    "$6$IJFASoJI$7x650JQGObqKBxPhxXhmegiWED.XUmolNfwHQW1jf.2NGvWQ7uF6yh2A5Sq67Lkj.9twhoCSZkoMFc

  # This value determines the NixOS release with which your system is to be
  # compatible, in order to avoid breaking some software such as database
  # servers. You should change this only after NixOS release notes say you
  # should.
  system.stateVersion = "18.09"; # Did you read the comment?
```

Nothing fancy, openssh, a root password set, locales, hostname etc...

We also need the `./hardware-configuration.nix` file from the server.

Next we can try to build a VM using this configuration using `nixos-rebuild` (if you don't have `nixos-rebuild` command read the next section):

```
$ NIXOS_CONFIG=$(pwd)/configuration.nix nixos-rebuild build-vm
building Nix...
building the system configuration...
error: The option `services.timesyncd.enable' has conflicting definitions, in `/nix/var/nix/prc
(use '--show-trace' to show detailed location information)
```

Ok that didn't go well.

So we used the `nixos-rebuild build-vm` command and set the `NIXOS_CONFIG` env variable to feed our configuration to `nixos-rebuild`. By default `nixos-rebuild` will use `/etc/nixos/configuration.nix`.

Error message says that there is a conflicting option between our configuration and `/nix/var/nix/profiles/per-user/root/channels/nixpkgs/nixos/modules/virtualisation/qemu-vm.nix`. But wait, where does that come from?

The thing is `nixos-rebuild build-vm` does a magic trick which is to automatically include this `qemu-vm.nix` module in our configuration to build the local vm. If we look at `nixpkgs/nixos/default.nix` we see this:

```
# This is for `nixos-rebuild build-vm'.
vmConfig = (import ./lib/eval-config.nix {
  inherit system;
  modules = [ configuration ./modules/virtualisation/qemu-vm.nix ];
}).config;
```

Ok so looking at `qemu-vm.nix` we can quickly find that this module wants to disable `timesyncd`:

```
# Don't run ntpd in the guest.  It should get the correct time from KVM.
services.timesyncd.enable = false;
```

So a quick and easy fix in our configuration would be to define:

```
services.timesyncd.enable = lib.mkDefault true;
```

With this the `qemu-vm.nix` module will be able to disable `timesyncd` but when we will actually deploy this configuration to our server `timesyncd` will be enabled.

The `qemu-vm.nix` module is interesting because it overrides things for us in order to run our configuration in a VM. For example the disk layout, bootloader...

Let's try again!

```
$ NIXOS_CONFIG=$(pwd)/configuration.nix nixos-rebuild build-vm
building Nix...
building the system configuration...
querying info about '/nix/store/rqdzyjiq5xi4sz0c5pm9882g3bk4hg9s-nixos-vm' on 'https://cache.ni
downloading 'https://cache.nixos.org/rqdzyjiq5xi4sz0c5pm9882g3bk4hg9s.narinfo'...
querying info about '/nix/store/rc0hnpn8vhl494xvrdniph2r0225qfcf-closure-info' on 'https://cach
[...]
building '/nix/store/rad4rzgrrmifprjw2mhmifvkk9gwiyyr-nixos-system-fripon-19.09pre-git.drv'...
building '/nix/store/c6c8yhblcrsp36585im8aczaq65zb4ra-closure-info.drv'...
building '/nix/store/mllcw7pviw6a29n18vmrgq3i1piax2c9-run-nixos-vm.drv'...
building '/nix/store/ngmyx87fl7sqwlnmr9cwp2njjk3iwnji-nixos-vm.drv'...

Done.  The virtual machine can be started by running /nix/store/rqdzyjiq5xi4sz0c5pm9882g3bk4hg
```

Nice the VM configuration was built and the build output is a script that runs the VM. We can run it easily with:

```
$ ./result/bin/run-fripon-vm
```

This will start the VM using `qemu`.

The first build might take some time because your `/nix/store` needs to be populated with all the dependencies required by the configuration. The step that actually create the disk image and run the VM is extremelly fast because the `/nix/store` is shared between the host and the VM.

# Accessing the VM with ssh

We can pass `qemu` network options through `QEMU_NET_OPTS` env variable:

```
$ QEMU_NET_OPTS=hostfwd=tcp::2221-:22 ./result/bin/run-fripon-vm
$ ssh root@localhost -p 2221
```

This makes `qemu` listen on port `2221` and forward all connections to the port `22` of the VM.

# Building without `nixos-rebuild`

Actually `nixos-rebuild build-vm` doesn't do anything special. It's just this same as building the `vm` attribute of `nixpkgs/nixos/default.nix`:

```
$ NIXOS_CONFIG=$(pwd)/configuration.nix nix-build '<nixpkgs/nixos>' -A vm
```

Or:

```
$ nix-build '<nixpkgs/nixos>' -A vm --arg configuration ./configuration.nix
```

# Using a specific version of nixpkgs

So we are able to build a VM from a NixOs configuration but it's using `nixpkgs` from our local host. Our server might be on a different release or commit of `nixpkgs`.

There is multiple ways to pin `nixpkgs` to a specific version. For now a simple way would be to use a `nixpkgs` local git clone at a certain commit and use that for our VM build by overriding `NIX_PATH`:

```
$ NIX_PATH=nixpkgs=${HOME}/vcs/nixpkgs nix-build '<nixpkgs/nixos>' -A vm --arg configuration ./
```

# Testing only the build of the config

Since now we built a script that allows us to run our configuration in a VM. The build of the VM configuration is a dependency of that script. The build of a configuration results in a directory layout containing all system files every time.

This is usually done using `nixos-rebuild build` which in fact is the same as building the `system` attribute of `nixpkgs/nixos/default.nix`:

```
$ NIX_PATH=nixpkgs=${HOME}/vcs/nixpkgs nix-build '<nixpkgs/nixos>' -A system --arg configuratio
```

But in this case the config does not use the `qemu-vm.nix` module. What we want is to build the configuration with the `qemu-vm.nix` module so that we can deploy it on a running local VM.

# Building the config for VM use

To build the configuration for the VM we need to write a bit of nix in a `default.nix` file for example:

```
{ configuration
, system ? builtins.currentSystem
}:

let

  eval = modules: import <nixpkgs/nixos/lib/eval-config.nix> {
    inherit system modules;
  };

in {

  vmSystem =
    (eval [ configuration <nixpkgs/nixos/modules/virtualisation/qemu-vm.nix> ]).config.system.b

}
```

First we define an `eval` function which takes a list of NixOS modules and calls `nixpkgs/nixos/lib/eval-config.nix` to evaluate them.

Next, we expose a `vmSystem` attribute that uses `eval` with our system configuration and the `qemu-vm.nix` module. We return the `config.system.build.toplevel` attribute of the resulting evaluation.

This attribute give us the root of the system layout:

```
$ NIX_PATH=nixpkgs=${HOME}/vcs/nixpkgs nix-build -A vmSystem --arg configuration ./configuratic
[...]
/nix/store/kr13nc1725pmm6biwbjr9yr7rjy7hahm-nixos-system-fripon-19.09.git.3ad23e3
$ ls -l /nix/store/kr13nc1725pmm6biwbjr9yr7rjy7hahm-nixos-system-fripon-19.09.git.3ad23e3
.r-xr-xr-x  13k root  1 Jan  1970 activate
lrwxrwxrwx   91 root  1 Jan  1970 append-initrd-secrets -> /nix/store/2dggzxanjra05bljab2wgl1wg
dr-xr-xr-x    - root  1 Jan  1970 bin
.r--r--r--    0 root  1 Jan  1970 configuration-name
lrwxrwxrwx   51 root  1 Jan  1970 etc -> /nix/store/mldhlrvyldvdj7lg66yzsbzim3y59anq-etc/etc
.r--r--r--    0 root  1 Jan  1970 extra-dependencies
dr-xr-xr-x    - root  1 Jan  1970 fine-tune
lrwxrwxrwx   65 root  1 Jan  1970 firmware -> /nix/store/1lq1wqmzr5ap66yhnl3lv67py1sm88d3-firmw
.r-xr-xr-x 5.4k root  1 Jan  1970 init
.r--r--r--    9 root  1 Jan  1970 init-interface-version
lrwxrwxrwx   71 root  1 Jan  1970 initrd -> /nix/store/mrx22ach3pgn0ic7wnnk33836smdy9ld-initrd-
lrwxrwxrwx   65 root  1 Jan  1970 kernel -> /nix/store/gvg17g25nr4jbq2pc26107yiqvk3m0d8-linux-4
lrwxrwxrwx   58 root  1 Jan  1970 kernel-modules -> /nix/store/0bryzvzfy0s4yh53zkzl4rrn30c5a8nj
.r--r--r--   24 root  1 Jan  1970 kernel-params
.r--r--r--   17 root  1 Jan  1970 nixos-version
lrwxrwxrwx   55 root  1 Jan  1970 sw -> /nix/store/xvk85q1ymllz43k657c2a7248qbvx4wd-system-path
.r--r--r--   12 root  1 Jan  1970 system
lrwxrwxrwx   55 root  1 Jan  1970 systemd -> /nix/store/7yypimpzkxjh5dm2aajdx4051l1xlw72-system
```

At this point it's possible to check the build of a particular configuration file without running the VM.

Also we can imagine injecting our own modules specific to the VM configuration to the `eval` function if necessary.

# Updating a running local VM with a new config

So we know how to build and run a local VM. We have also a way to build the configuration with the `qemu-vm.nix` module. Now it would be cool to be able to build a new configuration locally and if that works activate it in the running VM.

Usually to update a running NixOS system the `nixos-rebuild switch` is used. It looks by default for a configuration at `/etc/nixos/configuration.nix`, build it and activate it on the host. Behind the scenes it mainly runs 3 commands:

1. `system=$(nix-build '<nixpkgs/nixos>' -A system)`
2. `nix-env -p /nix/var/nix/profiles/system --set $system`
3. `$system/bin/switch-to-configuration switch`

The activation of the new system is done by the `switch-to-configuration` script. This script takes care of migrating the state of the current system according to the new configuration. For example: start new services, unmount filesystems, updating the bootloader...

We need another piece to copy the new system to the running VM: `nix-copy-closure`:

```
NAME
       nix-copy-closure - copy a closure to or from a remote machine via SSH

SYNOPSIS
       nix-copy-closure [--to | --from] [--gzip] [--include-outputs] [--use-substitutes | -s]
                        paths
```

`nix-copy-closure` is used to copy nix stores paths between nix enabled machines. It is quite efficient because if some path is already present on the target machine it won't be uploaded. It can also use substitutes meaning that if some path yout want to upload is present in a binary cache it will be pulled from it. This is very useful when your bandwidth is limited.

So given a local VM running and accessible via ssh on port `2221` we can deploy a new configuration with a little script. Let's name it `deploy.sh`:

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

set -e

NIX_PATH=nixpkgs=${HOME}/vcs/nixpkgs
PROFILE=/nix/var/nix/profiles/system

# Build the VM system
outPath=$(nix-build -A vmSystem --arg configuration ./configuration.nix)
# Upload to the VM
NIX_SSHOPTS="-p 2221" nix-copy-closure --to "root@localhost" --gzip $outPath
# Activate the new system
ssh -p 2221 root@localhost nix-env --profile "$PROFILE" --set "$outPath"
ssh -p 2221 root@localhost $outPath/bin/switch-to-configuration test
```

Our script uses `switch-to-configuration test` instead of `switch`. `switch` activates the configuration and updates the bootloader with a new entry for this configuration. But in our case the VM doesn't have any bootloader so we use `test` which just activate the new configuration.

So in order we can:

1. Build the VM run script:

```
NIX_PATH=nixpkgs=${HOME}/vcs/nixpkgs nix-build '<nixpkgs/nixos>' -A vm --arg configuratio
```

2. Run the VM with some qemu options to have SSH access:

```
QEMU_NET_OPTS=hostfwd=tcp::2221-:22 ./result/bin/run-fripon-vm
```

3. Make some changes in the configuration

4. Deploy the new configuration with our deploy script

```
./deploy.sh
these derivations will be built:
  /nix/store/3r07vbxl1irc03xnca8xkcklx8329i2f-system-path.drv
  ...
building '/nix/store/4zvybkbjdxxazsy2fmc2jzi3cyps63ms-nixos-system-fripon-19.09.git.3ad23
Password:
copying 10 paths...
copying path '/nix/store/ivlni1xcihrnf1hprfci3x88lwjyzvsh-system-path' to 'ssh://root@loc
...
Password:
Password:
activating the configuration...
setting up /etc...
reloading user units for root...
setting up tmpfiles
reloading the following units: dbus.service
```

Great! The new configuration was successfully built, uploaded to the VM and then activated. Though we had to input the SSH password multiple times because no SSH keys are setup.

## Conclusion

All the bits and pieces described here give us a really powerful way to iterate on a NixOS configuration especially if you don't want to mess with a production server.

Obviously all of this needs some lifting to make it more easy to use but hopefully with this you can build something that works for you.

Running Sway and friends with home-manager systemd user services ›