Graham Christensen

Erase your darlings

immutable infrastructure for mutable systems

posted on April 13 2020

I erase my systems at every boot.

Over time, a system collects state on its root partition. This state lives in assorted directories like `/etc` and `/var`, and represents every under-documented or out-of-order step in bringing up the services.

> “Right, run `myapp-init`.”

These small, inconsequential “oh, oops” steps are the pieces that get lost and don’t appear in your runbooks.

> “Just download ca-certificates to … to fix …”

Each of these quick fixes leaves you doomed to repeat history in three years when you’re finally doing that dreaded RHEL 7 to RHEL 8 upgrade.

> “Oh, `touch /etc/ipsec.secrets` or the l2tp tunnel won’t work.”

Immutable infrastructure gets us *so* close

Immutable infrastructure is a wonderfully effective method of eliminating so many of these forgotten steps. Leaning in to the pain by deleting and replacing your servers on a weekly or monthly basis means you are constantly testing and exercising your automation and runbooks.

The nugget here is the regular and indiscriminate removal of system state. Destroying the whole server doesn’t leave you much room to forget the little tweaks you made along the way.

These techniques work great when you meet two requirements:

- you can provision and destroy servers with an API call
- the servers aren’t inherently stateful

Long running servers

There are lots of cases in which immutable infrastructure *doesn’t* work, and the dirty secret is **those servers need good tools the most.**

Long-running servers cause long outages. Their runbooks are outdated and incomplete. They accrete tweaks and turn in to an ossified, brittle snowflake — except its arms are load-bearing.

Let’s bring the ideas of immutable infrastructure to these systems too. Whether this system is embedded in a stadium’s jumbotron, in a datacenter, or under your desk, we *can* keep the state under control.

FHS isn’t enough

The hard part about applying immutable techniques to long running servers is knowing exactly where your application state ends and the operating system, software, and configuration begin.

This is hard because legacy operating systems and the Filesystem Hierarchy Standard poorly separate these areas of concern. For example, `/var/lib` is for state information, but how much of this do you actually care about tracking? What did you configure in `/etc` on purpose?

The answer is probably not a lot.

You may not care, but all of this accumulation of junk is a tarpit. Everything becomes harder: replicating production, testing changes, undoing mistakes.

New computer smell

Getting a new computer is this moment of cleanliness. The keycaps don’t have oils on them, the screen is perfect, and the hard drive is fresh and unspoiled — for about an hour or so.

Let’s get back to that.

How is this possible?

NixOS can boot with only two directories: `/boot`, and `/nix`.

`/nix` contains read-only system configurations, which are specified by your `configuration.nix` and are built and tracked as system generations. These never change. Once the files are created in `/nix`, the only way to change the config's contents is to build a new system configuration with the contents you want.

Any configuration or files created on the drive outside of `/nix` is state and cruft. We can lose everything outside of `/nix` and `/boot` and have a healthy system. My technique is to explicitly opt in and *choose* which state is important, and only keep that.

How this is possible comes down to the boot sequence.

For NixOS, the bootloader follows the same basic steps as a standard Linux distribution: the kernel starts with an initial ramdisk, and the initial ramdisk mounts the system disks.

And here is where the similarities end.

NixOS's early startup

NixOS configures the bootloader to pass some extra information: a specific system configuration. This is the secret to NixOS's bootloader rollbacks, and also the key to erasing our disk on each boot. The parameter is named `systemConfig`.

On every startup the very early boot stage knows what the system's configuration should be: the entire system configuration is stored in the read-only `/nix/store`, and the directory passed through `systemConfig` has a reference to the config. Early boot then manipulates `/etc` and `/run` to match the chosen setup. Usually this involves swapping out a few symlinks.

If `/etc` simply doesn't exist, however, early boot *creates* `/etc` and moves on like it were any other boot. It also *creates* `/var`, `/dev`, `/home`, and any other core directories that must be present.

Simply speaking, an empty `/` is *not surprising* to NixOS. In fact, the NixOS netboot, EC2, and installation media all start out this way.

Opting out

Before we can opt in to saving data, we must opt out of saving data *by default*. I do this by setting up my filesystem in a way that lets me easily and safely erase the unwanted data, while preserving the data I do want to keep.

My preferred method for this is using a ZFS dataset and rolling it back to a blank snapshot before it is mounted. A partition of any other filesystem would work just as well too, running `mkfs` at boot, or something similar. If you have a lot of RAM, you could skip the erase step and make `/` a tmpfs.

Opting out with ZFS

When installing NixOS, I partition my disk with two partitions, one for the boot partition, and another for a ZFS pool. Then I create and mount a few datasets.

My root dataset:

```
# zfs create -p -o mountpoint=legacy rpool/local/root
```

Before I even mount it, I **create a snapshot while it is totally blank**:

```
# zfs snapshot rpool/local/root@blank
```

And then mount it:

```
# mount -t zfs rpool/local/root /mnt
```

Then I mount the partition I created for the `/boot`:

```
# mkdir /mnt/boot
# mount /dev/the-boot-partition /mnt/boot
```

Create and mount a dataset for `/nix`:

```
# zfs create -p -o mountpoint=legacy rpool/local/nix
# mkdir /mnt/nix
# mount -t zfs rpool/local/nix /mnt/nix
```

And a dataset for `/home`:

```
# zfs create -p -o mountpoint=legacy rpool/safe/home
# mkdir /mnt/home
# mount -t zfs rpool/safe/home /mnt/home
```

And finally, a dataset explicitly for state I want to persist between boots:

```
# zfs create -p -o mountpoint=legacy rpool/safe/persist
# mkdir /mnt/persist
# mount -t zfs rpool/safe/persist /mnt/persist
```

> *Note:* in my systems, datasets under `rpool/local` are never backed up, and datasets under `rpool/safe` are.

And now safely erasing the root dataset on each boot is very easy: after devices are made available, roll back to the blank snapshot:

```
{
  boot.initrd.postDeviceCommands = lib.mkAfter ''
    zfs rollback -r rpool/local/root@blank
  '';
}
```

I then finish the installation as normal. If all goes well, your next boot will start with an empty root partition but otherwise be configured exactly as you specified.

Opting in

Now that I'm keeping no state, it is time to specify what I do want to keep. My choices here are different based on the role of the system: a laptop has different state than a server.

Here are some different pieces of state and how I preserve them. These examples largely use reconfiguration or symlinks, but using ZFS datasets and mount points would work too.

Wireguard private keys

Create a directory under `/persist` for the key:

```
# mkdir -p /persist/etc/wireguard/
```

And use Nix’s wireguard module to generate the key there:

```
{
  networking.wireguard.interfaces.wg0 = {
    generatePrivateKeyFile = true;
    privateKeyFile = "/persist/etc/wireguard/wg0";
  };
}
```

NetworkManager connections

Create a directory under `/persist`, mirroring the `/etc` structure:

```
# mkdir -p /persist/etc/NetworkManager/system-connections
```

And use Nix’s `etc` module to set up the symlink:

```
{
  etc."NetworkManager/system-connections" = {
    source = "/persist/etc/NetworkManager/system-connections/";
  };
}
```

Bluetooth devices

Create a directory under `/persist`, mirroring the `/var` structure:

```
# mkdir -p /persist/var/lib/bluetooth
```

And then use systemd’s tmpfiles.d rules to create a symlink from `/var/lib/bluetooth` to my persisted directory:

```
{
  systemd.tmpfiles.rules = [
    "L /var/lib/bluetooth - - - - /persist/var/lib/bluetooth"
  ];
}
```

SSH host keys

Create a directory under `/persist`, mirroring the `/etc` structure:

```
# mkdir -p /persist/etc/ssh
```

And use Nix's openssh module to create and use the keys in that directory:

```
{
  services.openssh = {
    enable = true;
    hostKeys = [
      {
        path = "/persist/ssh/ssh_host_ed25519_key";
        type = "ed25519";
      }
      {
        path = "/persist/ssh/ssh_host_rsa_key";
        type = "rsa";
        bits = 4096;
      }
    ];
  };
}
```

ACME certificates

Create a directory under `/persist`, mirroring the `/var` structure:

```
# mkdir -p /persist/var/lib/acme
```

And then use systemd's tmpfiles.d rules to create a symlink from `/var/lib/acme` to my persisted directory:

```
{
  systemd.tmpfiles.rules = [
    "L /var/lib/acme - - - - /persist/var/lib/acme"
  ];
}
```

Answering the question "what am I about to lose?"

I found this process a bit scary for the first few weeks: was I losing important data each reboot? No, I wasn't.

If you're worried and want to know what state you'll lose on the next boot, you can list the files on your root filesystem and see if you're missing something important:

```
# tree -x /
├── bin
│   └── sh -> /nix/store/97zzcs494vn5k2yw-dash-0.5.10.2/bin/dash
├── boot
├── dev
├── etc
│   ├── asound.conf -> /etc/static/asound.conf
... snip ...
```

ZFS can give you a similar answer:

```
# zfs diff rpool/local/root@blank
M       /
+       /nix
+       /etc
+       /root
+       /var/lib/is-nix-channel-up-to-date
+       /etc/pki/fwupd
+       /etc/pki/fwupd-metadata
... snip ...
```

Your stateless future

You may bump in to new state you meant to be preserving. When I'm adding new services, I think about the state it is writing and whether I care about it or not. If I care, I find a way to redirect its state to `/persist`.

Take care to reboot these machines on a somewhat regular basis. It will keep things agile, proving your system state is tracked correctly.

This technique has given me the "new computer smell" on every boot without the datacenter full of hardware, and even on systems that do carry important state. I have deployed this strategy to systems in the large and small: build farm servers, database servers, my NAS and home server, my raspberry pi garage door opener, and laptops.

NixOS enables powerful new deployment models in so many ways, allowing for systems of all shapes and sizes to be managed properly and consistently. I think this model of ephemeral roots is yet another example of this flexibility and power. I would like to see this partitioning scheme become a reference architecture and take us out of this eternal tarpit of legacy.

Posts

- NixOS on the Framework
- Flakes are such an obviously good thing
- Erase your darlings
- ZFS Datasets for NixOS
- Optimising Docker Layers for Better Caching with Nix
- an EPYC NixOS build farm
- cache.nixos.org, now more local!
- Prometheus and the NixOS System Version
- NixOS on a Dell 9560
- How to use a NixOS Linux Server for Time Machine Backups
- Pip Install with Docker and Fixing the ascii decode error
- Packer - Create AMI with EBS Volumes with VolumeType
- Enable certificate revocation in Chrome
- How to delete all (or most) jobs from a beanstalk tube from the shell
- Why a MySQL Slave Created from an LVM Snapshot Would Mark Tables Corrupt
- Listing Users with Database Access

About

Graham works on NixOS.

- **E-Mail:** graham@grahamc.com
- **Phone:** +1-407-670-9980
- **GitHub:** github.com/grahamc
- **Twitter:** @grhmc