Declarative NixOS containers

2020-12-28

NixOS' containers allow you to run separate lightweight NixOS instances on the same machine. This can be interesting if you want to deploy multiple services on the same host that each need a custom OS configuration. NixOS' containers do *not* provide full security out of the box (just like docker). They do give you a separate chroot, but a privileged user (root) in a container can escape the container and become root on the host system. With that disclaimer out of the way (we have some solutions at the bottom of this post), let's look at an example.

Suppose we wanted to make a container called `wasabi` that hosts a simple HTTPD server. The configuration would look something like this:

```
  containers.wasabi = {
    ephemeral = true;
    autoStart = true;
    config = { config, pkgs, ... }: {
        services.httpd.enable = true;
        services.httpd.adminAddr = "foo@example.org";
        networking.firewall.allowedTCPPorts = [ 80 ];
    };
  };
```

After a `nixos-rebuild switch`, we will see that a new service is started `container@wasabi`. If we `curl localhost` then we will see that it works:

```
$ curl 'http://localhost'
<html><body><h1>It works!</h1></body></html>

$ systemctl status container@wasabi
● container@wasabi.service - Container 'wasabi'
     Loaded: loaded (/nix/store/...-unit-container-wasabi.service/container@wasabi.servic
     Active: active (running) since Thu 2020-12-24 14:22:49 UTC; 1h 15min ago
```

The container punched a hole through the firewall of the host and allowed us to access the hosted content, even from other computers than our own. But how can we see the status of the HTTPD daemon? Running `systemctl status httpd` on our server will show us nothing.

```
$ systemctl status httpd
Unit httpd.service could not be found.
```

Logging in to the container

To see the HTTPD service, we need to log into the container with:

```
sudo nixos-container root-login wasabi
```

Once in there, we see that the HTTPD service is indeed running:

```
[root@wasabi:~]# systemctl status httpd
● httpd.service - Apache HTTPD
     Loaded: loaded (/nix/store/...-unit-httpd.service/httpd.service; enabled; vendor pre
     Active: active (running) since Thu 2020-12-24 14:22:49 UTC; 12min ago
     ...
```

We can also find the server logs in `/var/logs/httpd` directory in the container.

To preserve state or not to preserve state

By default, nix-containers are stateful, files you modify while logged in to your container will persist over restarts and updates of the container. Just like your document folder that remains untouched by `nixos-rebuid`. Files managed by nix cannot be modified as they are symlinked from the read only `/nix/store` shared by host and container. So don't store secrets in the store if you don't trust the container fully.

We can also ensure that a container starts "fresh" every time it is updated or reloaded. To do this we set `containers.wasabi.ephemeral = true`. My general recommendation for configuration management is that you want as less state in your containers as possible. This ensures that you can `nix-rebuild` on another host and still have everything working.

Mounts

But sometimes there is important state you want to keep: uploaded files, database contents and so on. How can we manage those? You can preserve data in a mount. For this example, let's imagine that we want to preserve our HTTPD logs. To do this, we use the `containers.<name>.bindMounts` option:

```
    containers.wasabi.bindMounts = {
      "/var/log/httpd" = {
        hostPath = "/mnt/wasabiData/";
        isReadOnly = false;
      };
    };
```

The configuration above specifies that `/var/log/httpd` in the container should be linked to `/mnt/wasabiData` on the host (machine running the containers). For this to work the folders should exist and for HTTPD to have write privileges on the folder in the container we should declare the folders as follows, with `systemd.tmpfiles` (it's a bad name, I know). In the config of the container (the body of the function in

`containers.wasabi.config`), we must ensure that the `/var/log/httpd` directory is a directory (`d`) and that it is owned by user `wwwrun` (first one) and the group `wwwrun` (second one). The user and group must be set differently depending on the needs of your system, if you don't set the user and group properly, HTTPD will not be allowed to write to the folder. The easiest way to find out what user and group you need is to log into the container before you set up the mount and find out the permissions with `ls -l` in the parent directory.

```
containers.wasabi = {
    ...
    config = { config, pkgs, ... }: {
       systemd.tmpfiles.rules = [
        "d /var/log/httpd 700 wwwrun wwwrun -"
       ];
       ...
    };
}
```

We must also ensure that `/mnt/wasabiData/` exists on the host. Do *not* use `tempfiles` to achieve this, as this can cause confusing problems when you redeploy the system without modification to the container. In that case, the tempfiles on the host get executed, changing the permissions of the mounted directory in a way that may conflict with the configuration inside your container (your container will suddenly lose access to the data).

Data stored in mounted folders will be preserved even if the container is set to be ephemeral.

Networking and port forwarding

By default, declarative nix containers can use the network of the host. They can initiate connections to anywhere and listen on any port.

If you want to do any kind of port forwarding or reverse proxies you must set *all* of the following properties on your container

```
  containers.wasabi = {
    privateNetwork = true;
    hostAddress = "192.168.100.2"
    localAddress = "192.168.100.11";
  }
```

You may adjust the IP addresses to your liking. By setting `privateNetwork` to true, the containers network is decoupled from the hosts network. It gets its own virtual interface `ve-wasabi`. The container can not directly listen on ports on the host, and it cannot initiate connections to the outside world. The only connections it can have is to the host.

Give internet access

To allow our container to initiate connections to the public internet we need to set up network address translation (NAT). This will allow our containers to open non-privileged ports (> 1024) on the host to send and receive packets

to the outside world. To do this, add the following to the *host* config. (with `eth0` the name of your real network interface)

```
networking.nat.enable = true;
networking.nat.internalInterfaces = [ "ve-wasabi" ];
networking.nat.externalInterface = "eth0";
```

You can add the names of all containers with `privateNetwork` set to true that need internet access. To allow access to the internet to all your containers with a private network you can set `networking.nat.internalInterfaces = [ "ve-*" ];`

Note: This only if the container needs to connect remote servers (like databases), it is not needed to reply to incoming traffic coming form, for example a reverse proxy service on the host.

Reverse proxies

Before you start remapping ports, it might be interesting to realize that this is not necessary for all applications. Consider that our host is a service that host various HTTP based services in the containers `wasabi`, `sambal` and `tabasco`. With IP addresses `192.168.100.11`, `192.168.100.22` and `192.168.100.33`. We can use a nginx instance with Let's Encrypt certificates that allows us to dispatch incoming requests to the right container.

```
security.acme.acceptTerms = true;
security.acme.email = "letsencrypt@example.com";
services.nginx = {
  enable = false;
  recommendedProxySettings = true;
  recommendedTlsSettings = true;
  virtualHosts = {
    "wasabi.example.com" = {
      enableACME = true;
      forceSSL = true;
      locations."/".proxyPass = "http://192.168.100.11:80";
    };
    "samabal.example.com" = {
      enableACME = true;
      forceSSL = true;
      locations."/".proxyPass = "http://192.168.100.22:80";
    };
    "tabasco.example.com" = {
      enableACME = true;
      forceSSL = true;
      locations."/".proxyPass = "http://192.168.100.33:80";
    };
  };
};
```

Tip: you might want to put the IP addresses in variables.

Real Port Forwarding

If you only have one HTTP host or if the solution above does not work for you, you can use real port forwarding. The example below forwards port 22 on the container to port 2222 on the host, and forwards port 80 on the container to 8080 on the host. The ports should be opened by both the container's firewall and the hosts' firewall.

```
containers.wasabi.forwardPorts = [
  {
    containerPort = 22;
    hostPort = 2222;
    protocol = "tcp";
  }
  {
    containerPort = 80;
    hostPort = 8080;
    protocol = "tcp";
  }
];
```

Notes:

- Unfortunately, IPv6 forwarding is not supported (issue) yet.
- The loopback interface is explicitly excluded when forwarding ports. This means that we cannot `curl localhost:8080` on the host but other devices on the network can `curl myIP:8080`.

Underpinnings

NixOS containers are based on **systemd-nspawn**, a fancy chroot in the systemd-container program.

If you run into trouble, it might be interesting to check out the man pages of the project systemd-nspawn (1) and systemd-nspawn (5) and ofcourse the systemd-nspawn page on ArchWiki.

Security

A quick online search for "systemd-nspawn security" will tell you that it is "not secure". By default, NixOS containers are "privileged containers", these are containers where the user id zero inside the container has the same meaning outside the container. With some tricks, the root user inside the container can escape the container. (This issue also affects docker and the likes).

There are two ways around this: 1) don't run vulnerable programs in your container as root, 2) make the container unprivileged. Option 1) will probably work for you, but if not, I'll briefly show you what option 2) entails.

Unprivileged containers

Luckily there is a dim ray of hope: We can drop the privileges of a container to a non-privileged user with nspawn's `-U` option (set `containers.wasabi.extraFlags = [ "-U" ];`). This option ensures that the root user inside the container does not have UID `0` outside the container but rather something like `1815543862`. This works, but there are a lot of downsides to this regarding communication with the host and the outside world:

- You cannot listen on ports below 1024 in the container, not even as root (but we can easily tell httpd to listen on 8080)
- `bindMounts` break because there is no way to change the permissions of the mount to the right thing, because root in the container is not allowed to alter the permissions.
- `nixos-container root-login` is not compatible with these kinds of permissions

But if you are OK with that, you should be fine.

Note: if you find a nice way to fix some of these problems, let me know, or even better open a PR adding a `privileged` option to the containers in nixpkgs.

Stripping capabilities

Another way to reduce the capabilities of a container is by using `containers.wasabi.dropCapabilities` to remove some capabilities assigned to the container by default. A list of capabilities can be found in the capabilities (7) manpage, the capabilities assigned by default can be found in the "security options" section of the systemd-nspawn (1) . This section also holds more tricks to be added with `containers.wasabi.extraFlags`.

Sources

- Runtimes And the Curse of the Privileged Container by Christian Brauner, June 18, 2019: An excellent read on container security
- The NixOS containers module implementaion
- systemd-nspawn on ArchWiki
- A lot of manpages

Beard Hat Code
blog@beardhatcode.be
beardhatcode
robbertgs
A buffer overflow of all the things that go on under my hat while I'm coding...