Deploying Containers on NixOS

Burak Kiran
yeah, it ain't no luck

Infrastructure Dec 03, 2024

Managing infrastructure on your own machine can be cumbersome and scary. Much of the rhetoric out there would have you believe that it’s not possible or very dangerous to run and manage your own server. There have been some great advances with tools like containers(Docker and Podman) and NixOS that makes this easier than ever.

Why Bother with NixOS?

Most of my background is in DevOps and infrastructure. I set out to find the easiest way to manage multiple websites and applications on my own servers. Installing and managing Kubernetes seemed like a nightmare, and deploying with Docker Compose felt neither elegant nor easy enough to justify using it.

Then I came across NixOS. I found a way to adapt it to meet my needs for managing different application containers on a single machine—or even across a fleet. NixOS is a Linux distribution that offers a unique approach: immutable and declarative builds. This allows you to define the state of a machine in a single configuration file, which, as an infrastructure professional, is something I really like.

This approach fits me because of how I think about software. I love being able to see everything running on a machine in one place, defined declaratively. It enables you to create a configuration for your machine and seamlessly apply it to the host.

Containers, Containers

I have considerable experience running containers in production environments, usually on Kubernetes. Becuase of this, I prefer to package my software as a container. Many of my workflows are built around the useage of containers.

For me, containers are simply a means to get things up and running in production quickly. While I can easily get a Docker image of my software packaged, how can I get it running on NixOS?

1. Setting up Virtualization and Podman

There is an option in NixOS called vitualization. This allows for all different types of virtualization but we want to virtualize at the OS level(how Docker and Podman works). So here we're going to pick one to use and and enable it. I'm using Podman but you can choose Docker if you like.

```
virtualisation = {
  podman = {
    enable = true;
  };
};
```

2. Adding Our Container

Okay so now that our virtualization is turned on, we want to start a container. We do that by defining a option in `virtualisation` called `oci-containers.containers`. For each container we want to run, we create and entry here. There are a few options that I always use:

- **image**: Defines the container image we want to run.
- **environment**: Define environment variables that you want to be exposed within the container.
- **entrypoint**: Define a command to run on container startup(if needed).

```
virtualisation = {
    podman = {
      enable = true;
    };
    oci-containers.containers = {
      my-appication = {
        image = "myregistry.com/myApplication:latest";
        entrypoint = "/root/main";
        environment = {
            DEV_MODE = "false";
        };
      };
```

```
    };
};
```

3. Private Registry, No Problem

Your container may not actually work at the previous step because it's behind a private registry. It's good practice to put your containers behind one and if you do, you need a way to authenticate. This is done using the `login` configuration.

```
virtualisation = {
    podman = {
      enable = true;
    };
    oci-containers.containers = {
      my-appication = {
        login = {
            registry = "https://myregistry.com";
            username = "myRegistryUsername";
            passwordFile = "/root/registry-password.txt";
        };
        image = "myregistry.com/myApplication:latest";
        entrypoint = "/root/main";
        environment = {
            DEV_MODE = "false";
        };
      };
    };
};
```

4. Opening it Up to the Outside

For things like web servers, we need a way to expose our container to the outside world. This is achieved using ports on both the host and the container. First, we add the `ports` configuration to our container setup. At this point, our container is accessible to the machine itself. To make it accessible from outside the machine, we specify the following configuration: `networking.firewall.interfaces.ens4.allowedTCPPorts`. The complete configuration will look like this:

```
networking.firewall.interfaces.ens4.allowedTCPPorts = [
  8090
];
```

```
  virtualisation = {
      podman = {
        enable = true;
      };
      oci-containers.containers = {
        my-appication = {
          login = {
              registry = "https://myregistry.com";
              username = "myRegistryUsername";
              passwordFile = "/root/registry-password.txt";
          };
          image = "myregistry.com/myApplication:latest";
          ports = ["8090:8000"];
          entrypoint = "/root/main";
          environment = {
              DEV_MODE = "false";
          };
        };
      };
  };
```

Interacting With Your Running Container

Containers are started as systemd processes. You can use systemd commands interact and debug each running container. Also since our virutaulization is enabled, we can use our Podman and Docker commands.

Here are my favorite tools for debugging my running services. Systemd processes are named `podmain-{container-name}.service` so in our example it would be `podman-my-appication.service`

Starting, Stoping and Statuses

To get the current status of the application and last lines of logs we want to use the `status` command. This is useful for doing a quick check on if anything failed or just getting the last lines of logs.

```
systemctl status {service}
```

Let's say you want to start and stop a process. We can use the respective `start` and `stop` commands.

```
systemctl start {service}
systemctl stop {service}
```

The Process Journal

`systemctl` is a useful command but it does not show us all the logs. To be able to view all the logs of a systemd serice, we need to use the `journalctl`.

```
journalctl -u {service} -b
```

Container Commands

To get a list of all the running podman services. Often times I use this command to get the container ID.

```
podman ps
```

We can get more granular and specify the service name to get the ID of the container

```
podman ps -aqf "name={service}"
```

Often times, I'll jump into the contain to run some one off command or dig through internals to find an issue that I could not find on the logs exposed to the systemd service. We'll do that with the `exec` command and use a shell that's avaliable in your container(in this example im using `sh`).

```
podman exec -it {container-id} /bin/sh
```

What are You Waiting For?

NixOS has some great attributes, especially if you're willing to invest the time to learn and understand its configuration language. It's the easiest Linux distribution I've used that allows me to go from configuration to working container orchestration seamlessly while not loosing sight that we're running on Linux infrastructure. This also makes continuous deployment much simpler—we'll explore that in future posts.

Have questions about what you read? Get in touch now

Contact Me

Learn More →