How we replaced Vagrant with devenv

What came before?

A few years ago we started using Vagrant with an Ubuntu virtual machine (VM) as the basis of our development environment. It served us well at the time, providing a good way to have the same environment as we had on our servers and a reasonably reproducible environment for all developers. It also integrated well with Ansible, which we adopted at the same time, and started the era of “GitOps” in my workplace.

Why switch?

As well as Vagrant has worked for us over the years, it has recently started to feel old and slow. In particular, the shared folder was becoming a problem as we started to work more with tools like Composer and NPM, as it adds a lot of latency. We worked around this by moving more of the code into the VM, but this caused other problems. The shared folder wasn’t case-sensitive, which occasionally caused problems when the code was deployed to production. It was also very slow to re-create or simply re-provision a VM, which caused friction during development. So we started looking for alternatives.

Why not containers?

Of course, containers came to mind very quickly as they are now the de facto standard in the industry. I think containers can be very useful for easily deploying and distributing applications. However, for development I think they are not the right tool, they are just not nice to use interactively. There are projects like distrobox that help with that, but containers themselves aren’t really designed for that. Also, the syntax of a container file is a pain in the ass.

Our application is in many places what you would call legacy, so we are not quite ready for containers anyway. As it turned out, this would also be a problem with devenv, at least partially.

The good thing about Nix/devenv is that it doesn’t prevent us from using containers in the future. In fact, it is supported to build containers directly from it.

What about WSL?

For a few years now, Windows has supported something called the Windows Subsystem for Linux (WSL). One idea was to use plain WSL to run the services we needed for our projects. After a bit of research and testing, it didn’t really seem practical to install the services directly into WSL, because the WSL environment is not that easy to manage. Unlike Vagrant, where you can easily spin up multiple VMs, with WSL you have to import and export them. However, during our testing we found that WSL is a really nice base for other tools. We use Windows as our operating system but have been

What else then?

I've been using NixOS on all my personal systems for just over three years now and it's served me very well so far. The community is now really starting to take off and there is a lot of development going on. For personal projects I played around with Nix Flakes, normal Nix Shells and one day I discovered the devenv.sh project. All tools to help you develop and/or package an application. Devenv looked very intriguing because it looked similar to the module syntax I was already used to from NixOS. This meant that you could, for example, enable a MySQL service by simply adding this code:

```
services.mysql.enable = true;
```

From my personal tests, devenv looked like a promising alternative to our Vagrant setup. I introduced it to my workplace and was allowed to build a proof of concept (PoC) to see if we hit any obvious roadblocks that could kill the project.

Building the proof of concept

To build the proof of concept and eventually our final environment, I used an Ubuntu 22.04 VM inside WSL. I also installed Nix with flakes enabled and direnv. Direnv is a simple but powerful tool that automatically loads an environment when you enter a directory. With it, we don't need to run any commands to activate the environment, it just happens automagically when you switch to the directory.

Devenv has its own files which you would normally use to configure the environment. But normal flakes are also supported, and I chose to use them because I thought it would be better to stay generic rather than learn an additional format. Even if it was similar to a normal nix file. Also, I have a vision to create additional packages from the flake, like documentation, minified CSS, etc, and this might be easier with plain flakes, but I honestly don't know for sure.

I can't share specific code, but in general our product is a web application that uses

- MariaDB for database
- PHP-FPM for executing code
- Nginx as the web server
- Plus some additional packages for document generation and developer tools

The bootstrapping of the application is still done with Ansible because the code already exists and we use it for our servers anyway. So it felt wasteful to throw it away for something else, and in general it worked quite well for what we were using it for.

Problems during the proof of concept

Getting the basics up and running was fairly easy and quick. It required some research, but overall it went well and even helped to deepen my understanding of the tools we were already using.

The most difficult part was getting PHPStorm, the IDE our developers use, to recognise the services within the environment. With editors like Emacs or VSCode this is no problem at all, they both have extensions to work with direnv and they work fine. The direnv extensions for PHPStorm are not very good, but this seems to be a problem with the way PHPStorm works, not the extensions themselves. We tried running PHPStorm directly inside WSL and as part of devenv.

localhost to 127.0.0.1 for some reason.

So we now run PHPStorm on Windows, connected to WSL, with all the tools running inside WSL.

The other big problem was WSL itself. I installed it on about 5 machines and each time it was a bit different but we got it to work on all of them. This surprised me as it is a tool that comes directly from Microsoft, I expected it to “just work”. We had our problems and bugs with Virtualbox and Vagrant, but installation was always a breeze.

For me, the more interesting issues were with our application itself. Because we weren’t using a standard Ubuntu-based environment, but one based on Nix, we discovered, and are still discovering, a lot of hidden assumptions baked into our application. The thing about Nix is that it gives an application exactly the dependencies defined for it, but only those. The application is isolated from the other packages. The advantage of this is that you know exactly what you need to run an application, and you don’t have collisions between dependencies if another application needs a different version. Because of these limitations, we discovered

- Missing configurations that are available by default on an Ubuntu system.
- Missing packages that we didn’t specifically install on Ubuntu, but that were present and used by our application.
- Hard-coded paths to binaries in `/usr/bin/`.

We also discovered many hard-coded paths where our application expected to find itself. These were discovered because we were now running our application in any directory we wanted. With Vagrant, we always mapped the code to the same specific location. So even if the repository was in a different place, to the application it always looked like it was in the same place.

These application-related problems were a good thing, because solving them made our application more robust and portable. We would have discovered many of them if we had moved to containers. But with this approach, we discovered even more because an Ubuntu-based container would probably have provided the same or similar defaults as a normal VM.

I had to package a few applications that weren’t available in nixpkgs, but they were quite easy to package and I will try to upstream them. However, one of them is a proprietary third party application, so I may not be able to.

Going forward

After a few weeks of testing the PoC, we had a meeting to discuss the testers’ overall impressions and how we wanted to proceed. We didn’t find any critical problems, and the testers thought it worked so much better that we decided to go ahead. In particular, the improved speed was a big factor.

Unit tests weren’t really faster, but browser response times were improved by a factor of ten in some places. Our devenv setup now takes about 3 minutes to rebuild the application from scratch, and less than 5 seconds to restart it and display the website after it was stopped.

For our final setup, we wanted a way to provide global tools and configurations within the WSL so that every developer had the same baseline to work with and all the tools available from the start. We thought about using Ansible to bootstrap and standardise the WSL, but we had a bit of a chicken and egg problem because we had to somehow install Ansible in order to use it.

From my personal systems I knew about home-manager. A tool for installing user applications and configurations, I decided to use it to set up the WSL. Since we already had Nix installed, it was very easy to install home-manager. In addition, we don’t have to write cleanup tasks when we remove or move files as with Ansible, and we don’t have to

added some escape hatches for our developers in case they have some personal configurations they want to apply. However, the main idea is to find as much common configuration as possible and make it available through Home Manager.

Once you have the Windows side of WSL ready to go, it is very easy to set up a new development environment. I'm not a big fan of the cloud, but the idea of Chromebooks, that you can throw them out the window, get a new one and be up and running in a couple of minutes, always looked very cool. Our setup now allows for that, at least partially. Hopefully you pushed your code to the server first ;).

Final thoughts

Personally, I'm very happy with this development because I see a lot of potential in the Nix ecosystem. From my point of view as a DevOps/systems person, it just makes a lot of sense. I'm excited to see what we can do with it in the future, maybe one day we'll have fully declarative NixOS systems instead of Ubuntu.

On the other hand, it is a bit scary to implement all this. Even though Nix itself is quite old at this point[1], it feels very young and new. It is a bit of a niche application at the moment, although it is gaining a lot of traction. I sometimes worry that I'm getting too excited about this technology and wanting to use it everywhere I can. However, I hope that my colleagues would hold me back if that were the case :).

Another concern is that we're moving away from our servers, as the development environment and the servers are no longer the same, and this could lead to instability. This is still a concern, but the long term plan is to create containers from our nix-based setup and move the servers closer to our development environment.

Footnotes

1. Nix started more than 20 years ago; Docker, by comparison, is only ten years old. ↩