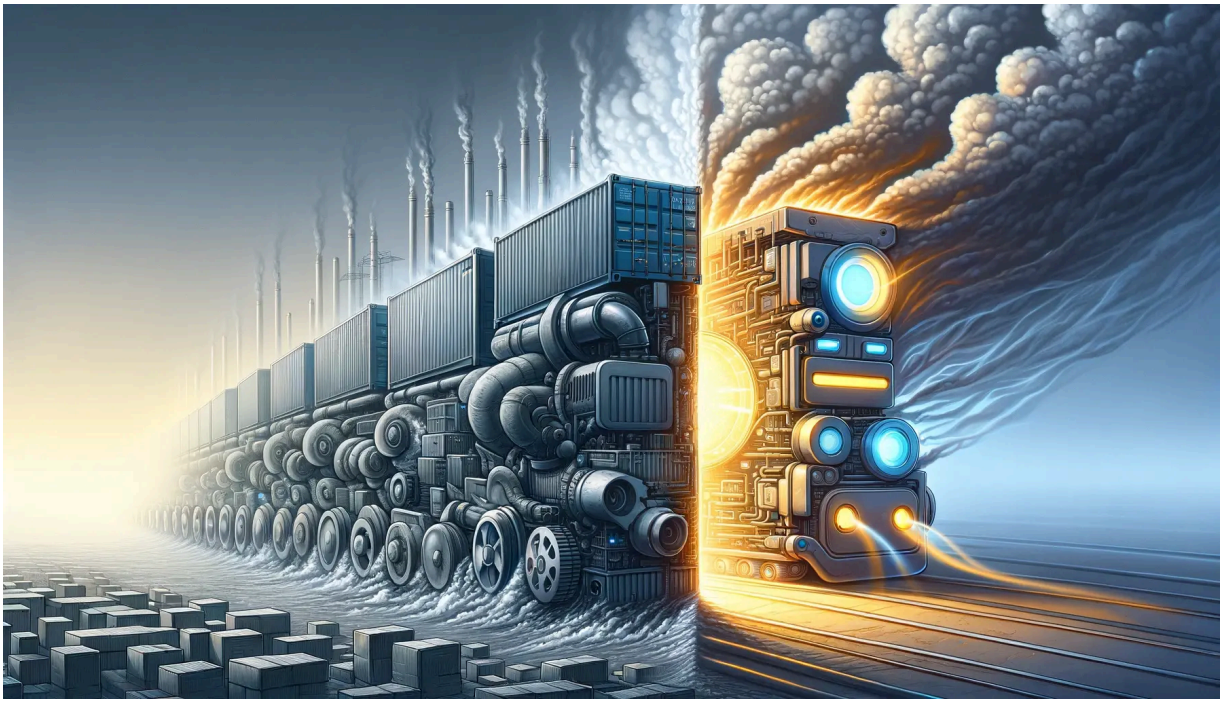
Replacing docker-compose with Nix for development

Ever since I first started using Nix for development, I have enjoyed the simplicity of setup: `nix develop`, make the code change and see it work. That's all well and good, but when your project keeps growing, you need to depend on external services like databases, message brokers, etc. And then, a quick search will tell you that docker is the way to go. You include it, add one more step in the setup guide, increasing the barrier to entry for new contributors. Not to mention, eating up all the system resources [1] on my not so powerful, company-provided MacBook.

This, along with the fact that we can provide one command to do more than just running external services (more about this at the end of the post), made us want to replace docker-compose with Nix in Nammayatri (Form now on, I'll use 'NY' as the reference for it).

Nammayatri

NY is an open-source auto rickshaw booking platform, based in India.



What does it take?

Turns out, there is not a lot of things that we need to do: we need to be able to run services natively, across platforms (so that my MacBook doesn't drain its battery running a database), and integrate with the existing flake.nix (to avoid an extra step in the setup guide).

If you've ever used NixOS before, you might be familiar with the way services are managed. Let's take a quick look at an example to understand if that will help us arrive at a solution for our problem.

NixOS services

Running services in NixOS is a breeze. For example, running a PostgreSQL Database is as simple as adding one line to your configuration:

```
{
  services.postgresql.enable = true;
}
```

This starts the database natively, with a global data directory, without the need for a container. That's great. What we need, however, is the same simplicity but with a project-specific data directory, applicable to macOS and other Linux distributions.

Cross-platform NixOS-like services

In the last section, we saw how easy it is to run services in NixOS. We are looking for something similar for our development environment that runs across platforms. Additionally, the solution should:

- Allow for running multiple instances of the same service (NY uses multiple instances of PostgreSQL and Redis).
- Ensure that services and their data are project-specific.

These were the exact problems services-flake was designed to solve. Along with running services natively, it also integrates with your project's `flake.nix` .

services-flake

How does services-flake solve them?

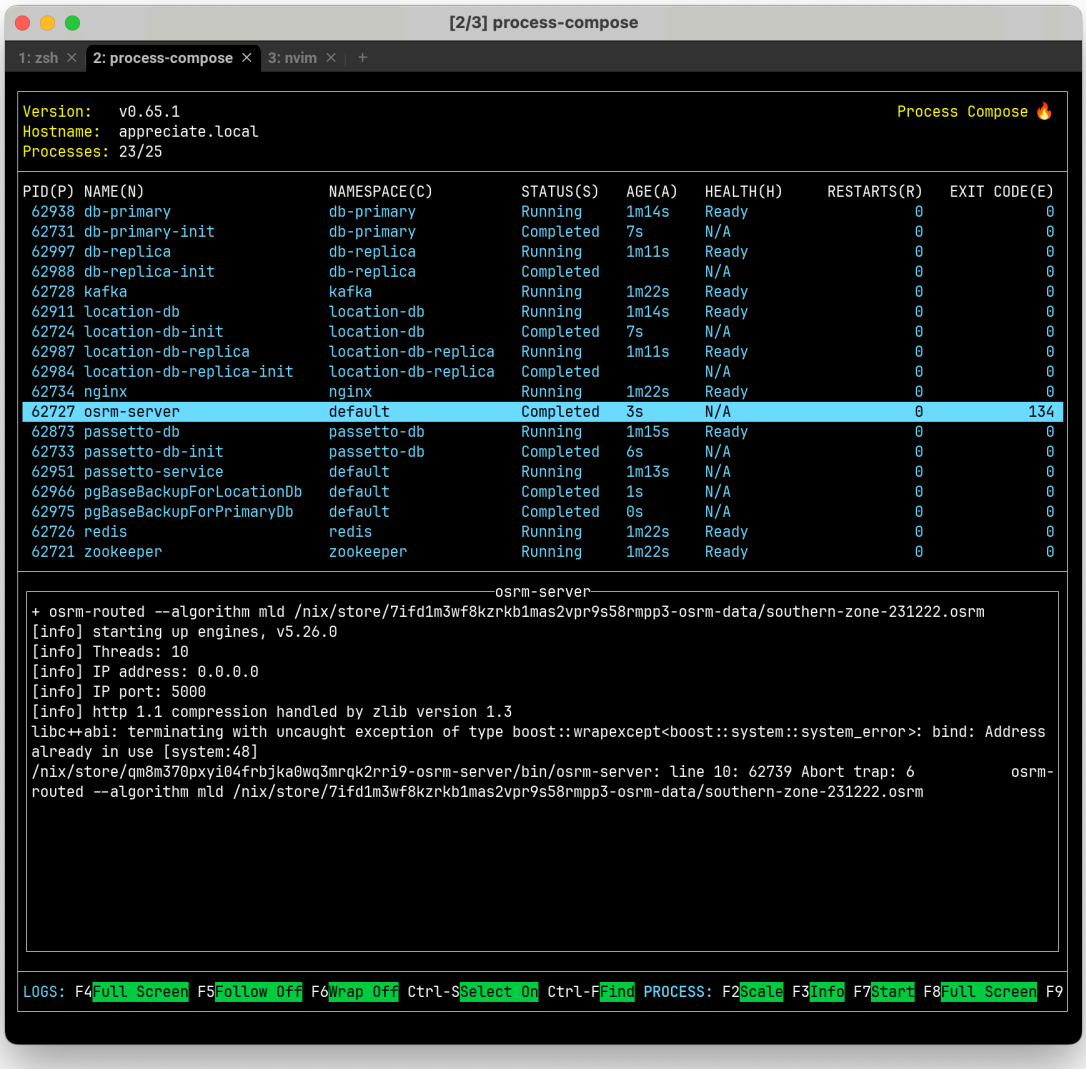
- It uses flake-parts for the module system (that's the simplicity aspect), and process-compose-flake for managing services, along with providing a

TUI app to monitor them.

- To address the need for running multiple instances, services-flake exports a `multiService` library function.
- By default, the data of each service is stored under `./data/<service-name>`, where `./` refers to the path where the process-compose app, exported by the project flake is run (usually in the project root).

Let's get started

Now that we have all the answers, it's time to replace docker-compose in NY with services-flake. We will only focus on a few services to keep it simple; for more details, refer to the PR.



PostgreSQL

NY uses about 3 instances of PostgreSQL databases.

One of them is exported by passetto (passetto is a Haskell application that encrypts data before storing it in postgres), and using it looks like:

```
{
  services.passetto.enable = true;
}
```

By leveraging the module system, we can hide the implementation details and only expose the `passetto` service to the user, enabling its use as shown above.

The other two instances are configured by the postgres-with-replica module. This module starts two services (`primary` and `replica` databases) and a pg-basebackup process (to synchronize `replica` with `primary` during initialization). For the user, it appears as follows:

```
{
  services.postgres-with-replica.enable = true;
}
```

Redis

NY uses Redis as a cache and clustered version of it as a key-value database. Redis service comprises a single node, while the clustered version has 6 nodes (3 masters and 3 replicas). Adding them to the project is as simple as:

```
{
  services.redis.enable = true;
  services.redis-cluster.enable = true;
}
```

Cool things

By no longer depending on Docker, we can now run the entire NY backend with one command, and it's all defined in a single place.

That's not all; we can also share the NY backend module to do much more, such as defining load-test configurations and running them in CI/local environments, again, with just one command. In this case, we take the module to run the entire NY stack and then extend it to add a bunch of load-test processes before bringing the whole thing to an end (as the load-test ends).

This is what running them looks like:

```
# Run load-test
nix run github:nammayatri/nammayatri#load-test-dev

# Run the entire backend
nix run github:nammayatri/nammayatri#run-mobility-stack-nix
```

Up next

Sharing services-flake modules deserves a separate post, so we will delve into this topic more in the next post.

Footnotes

1. The high resource consumption is due to docker running the containers on a VM, there is an initiative to run containers natively on macOS, but it is still in alpha and requires a lot of additional steps to setup. One such step is disabling SIP, which a lot of company monitored devices might not be allowed to do.