Dev Environments

On NixOS, we have a variety of methods to set up development environments, with the most ideal approach being a complete definition of each project's development environment through its own `flake.nix` . However, this can be somewhat cumbersome in practice, as it requires crafting a `flake.nix` and then running `nix develop` for each instance. For temporary projects or when one simply wants to glance at the code, this approach is somewhat overkill.

A compromise is to divide the development environment into three tiers:

1. **Global Environment**: This typically refers to the user environment managed by home-manager.
   - Universal development tools: `git` , `vim` , `emacs` , `tmux` , and the like.
   - Common language SDKs and package managers: `rust` , `openjdk` , `python` , `go` , among others.
2. **IDE Environment**:
   - Taking neovim as an example, home-manager creates a wrapper for neovim that encapsulates its dependencies within its own environment, preventing contamination of the global environment.
   - Dependencies for neovim plugins can be added to the neovim environment via the `programs.neovim.extraPackages` parameter, ensuring the IDE operates smoothly.
   - However, if you use multiple IDEs (such as emacs and neovim), they often rely on many of the same programs (like lsp, tree-sitter, debugger, formatter, etc.). For ease of management, these shared dependencies can be placed in the global environment. Be cautious of potential dependency conflicts with other programs in the global environment, particularly with python packages, which are prone to conflicts.
3. **Project Environment**: Each project can define its own development environment (`devShells`) via `flake.nix` .
   - To simplify, you can create generic `flake.nix` templates for commonly used languages in advance, which can be copied and modified as needed.
   - The project environment takes the highest precedence (added to the front of the PATH), and its dependencies will override those with the same name in the global environment. Thus, you can control the version of project dependencies via the project's `flake.nix` , unaffected by the global environment.

Templates for Development Environments

We have learned how to build development environments, but it's a bit tedious to write `flake.nix` for each project.

Luckily, some people in the community have done this for us. The following repository contains development environment templates for most programming languages. Just copy and paste them:

- MordragT/nix-templates
- the-nix-way/dev-templates

If you think the structure of `flake.nix` is still too complicated and want a simpler way, you can consider using the following project, which encapsulates Nix more thoroughly and provides users with a simpler definition:

- cachix/devenv

If you don't want to write a single line of nix code and just want to get a reproducible development environment with minimal cost, here's a tool that might meet your needs:

- jetpack-io/devbox

Dev Environment for Python

The development environment for Python is much more cumbersome compared to languages like Java or Go because it defaults to installing software in the global environment. To install software for the current project, you must create a virtual environment first (unlike in languages such as JavaScript or Go, where virtual environments are not necessary). This behavior is very unfriendly for Nix.

By default, when using pip in Python, it installs software globally. On NixOS, running `pip install` directly will result in an error:

```bash
› pip install -r requirements.txt
error: externally-managed-environment

× This environment is externally managed

```

```
╰─> This command has been disabled as it tries to modify the immutable
    `/nix/store` filesystem.

    To use Python with Nix and nixpkgs, have a look at the online documentation:
    <https://nixos.org/manual/nixpkgs/stable/#python>.

note: If you believe this is a mistake, please contact your Python installation
hint: See PEP 668 for the detailed specification.
```

Based on the error message, `pip install` is directly disabled by NixOS. Even when attempting `pip install --user`, it is similarly disabled. To improve the reproducibility of the environment, Nix eliminates these commands altogether. Even if we create a new environment using methods like `mkShell`, these commands still result in errors (presumably because the pip command in Nixpkgs itself has been modified to prevent any modification instructions like `install` from running).

However, many project installation scripts are based on pip, which means these scripts cannot be used directly. Additionally, the content in nixpkgs is limited, and many packages from PyPI are missing. This requires users to package them themselves, adding a lot of complexity and mental burden.

One solution is to use the `venv` virtual environment. Within a virtual environment, you can use commands like pip normally:

```shell
python -m venv ./env
source ./env/bin/activate
```

Alternatively, you can use a third-party tool called `virtualenv`, but this requires additional installation.

For those who still lack confidence in the venv created directly with Python, they may prefer to include the virtual environment in `/nix/store` to make it immutable. This can be achieved by directly installing the dependencies from `requirements.txt` or `poetry.toml` using Nix. There are existing Nix packaging tools available to assist with this:

> Note that even in these environments, running commands like `pip install` directly will still fail. Python dependencies must be installed through `flake.nix` because the data

> is located in the `/nix/store` directory, and these modification commands can only be executed during the Nix build phase.

- python venv demo
- poetry2nix

The advantage of these tools is that they utilize the lock mechanism of Nix Flakes to improve reproducibility. However, the downside is that they add an extra layer of abstraction, making the underlying system more complex.

Finally, in some more complex projects, neither of the above solutions may be feasible. In such cases, the best solution is to use containers such as Docker or Podman. Containers have fewer restrictions compared to Nix and can provide the best compatibility.

Loading comments...