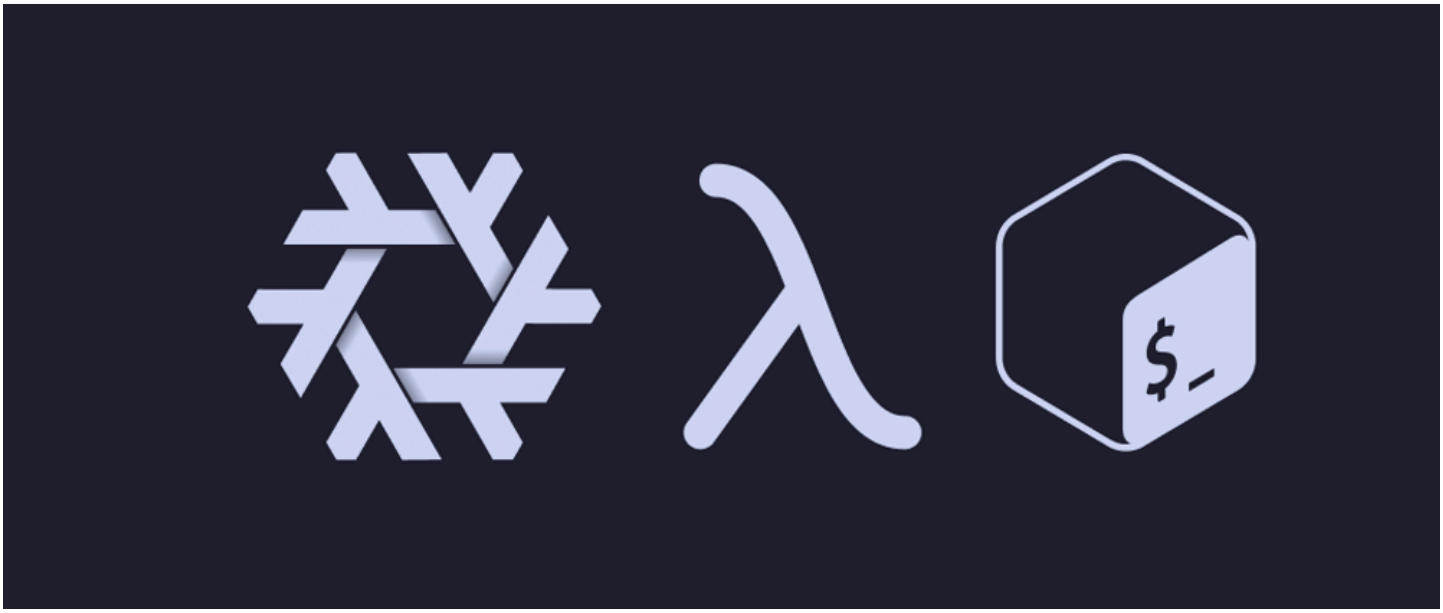


arnu515
Posted on 15 січ.

Easy development environments with Nix and Nix flakes!

#nix #tutorial #linux #webdev

Nix (3 Part Series)

1 My new Nix series!

2 Getting started with Nix and Nix Flakes

3 **Easy development environments with Nix and Nix flakes!**

In this article, we shall cover declarative development shells with Nix flakes! If you're new to Nix, I recommend checking out the previous two articles in this series to get a better understanding, since this article assumes that you've read the previous two already.

Let's revise: Creating a Shell

The previous article introduced you to the `nix shell` command, which downloads/builds packages and puts you in a shell environment with them in the `$PATH`.

> You need not run the examples given below, they're just provided for illustration purposes.

By default, `nix shell` drops you into a shell (which is your login shell) with the packages in `$PATH`. You can also specify an arbitrary command to run, for example:

```
# Postgres client
nix shell nixpkgs#postgres_17 --command psql
```

Similarly, you can also use another shell (that should exist in `$PATH`)

```
nix shell nixpkgs#nushell --command nushell  # https://www.nushell.sh/
```

You can also run multiple commands like so:

```
nix shell nixpkgs#gnumake --command sh -c "make && make install"
```

If you want to run the package itself, you need not use `nix shell`, since the `nix run` command also exists.

```
nix run nixpkgs#vim -- --log some.log
```

There are some more things that `nix shell` can do, which I will not be covering here, but they are covered in the reference manual.

Using `nix shell` to create a development environment

This can be done, but it is tedious. Imagine having to type dozens of dependencies manually by hand every time you want to enter your development environment, or even if you put it in a script, or use the `--stdin` argument, you'll still have the risk of installing a different version of the package than required. You may also like to customize the shell environment a bit more, like populating environment variables, running commands before the shell starts, etc.

Instead, there's a better way..

Creating a Development Shell with Flakes

Development shells (or shell environments as they're called in non-flake land) are a feature of Nix that allow you to all the things mentioned above — install specific versions of packages, set environment variables, run commands before the shell starts, etc.

Let's start with the same `flake.nix` as last time. You could copy it from here, but I'd like to digress a bit by introducing Nix templates!

Templates are a way for you to create, well template folders which are declared by a flake. Nix can then copy those files over from a flake into your current directory with the `nix flake init` command. Nix makes templates declarative and versioned too!

> Contrary to the command's reference manual, there's no need for the template itself to be a flake!

A template is pretty easy to create, you just need to add an attribute set called `templates` in your flake's `outputs`. The keys of the attrset are the template names, and the values are an attribute set with three fields: `description`, a description of the template; `path`, the path to the template folder; and `welcomeText`, some text that is output when someone uses the template.

I've made the simple flake from last article into a template hosted at this article series' GitLab page, it looks like this:

```
// https://gitlab.com/arnu515-tutorials/nix/-/blob/master/flake.nix?ref_type=

{
  outputs = {...}: {
    // ...
    templates.simple = {
      description = "A simple flake";
      path = ./a-simple-flake;
      welcomeText = ''
        Welcome to Nix!

        This flake exports a single package, `hello`, which can be
        executed by running:

        $ nix run #.hello

        Check out my article series on Nix for more information!
        https://dev.to/arnu515/my-new-nix-series-2cc3
      '';
    };
```

```
    };
}
```

You can use a template by using the `nix flake init` command. Beware that `nix flake init` will copy the template's files to the current directory, but it will not overwrite existing files:

```
nix flake init --template gitlab:arnu515-tutorials/nix#simple
```

> Note that Nix checks the fragment `simple` (called a flake output name) starting from the `templates` attrset, then the rest of the flake, unlike in `nix shell/run/build`, which started from the `packages` attrset, then moved to the `legacyPackages` attrset, then the rest of the flake.

Creating a development shell

Now let's create the development shell. This will be created in the `devShells` attrset, which looks just like the `packages` attrset, i.e. it has another attrset inside it for each system (like `x86_64-linux`), and those attrsets declare development shells inside them.

Just like `packages`, a development shell named `default` is the default development shell.

Here's a simple shell that has the `hello` package:

```
{
  description = "Nix devshells!";

  inputs = {
    nixpkgs.url = "github:nixos/nixpkgs/nixpkgs-24.11";
  };

  outputs = {nixpkgs, ...}: let
    system = "x86_64-linux";
    pkgs = import nixpkgs { inherit system; };
  in {
    devShells.${system}.simple = pkgs.mkShell {
      packages = [ pkgs.hello ];
      shellHook = ''
        ${pkgs.hello}/bin/hello
      '';
    };
```

```
    };
  }
```

There's quite a few changes from earlier! First, the `hello` package was removed, since we don't need it for this demonstration. Then, two variables called `system` and `pkgs` were created using the `let ... in ...` construct. Finally, we create a shell using the `pkgs.mkShell` function, specifying `packages` to be put in path, and a `shellHook` to run before the shell starts. Let's dissect this one-by-one.

The `${...}` syntax does string interpolation, replacing itself with the evaluated string inside the braces. `pkgs.hello` evaluates to the nix store path of the `hello` package. Thus, we need to append `/bin/hello` to actually point to the `hello` executable. Note that just `hello` could also have been specified, since it will be available in `$PATH`, but it is better to specify an absolute path like this so you know where your dependencies are coming from.

Next, let's talk about what `pkgs = import nixpkgs { inherit system; };` does. The `import` function takes in a path, and evaluates it if it is a path to a Nix file, or evaluates the `default.nix` file within the directory if it is a path to a directory (which is the case when a flake reference, like `nixpkgs` is passed to `import`), and returns the evaluation. Since `nixpkgs` points to the nixpkgs flake, which is a directory, it evaluates the `default.nix` file present within the flake, which in the case of nixpkgs returns a function accepting, among others, an argument for the current `system`, which is what we have passed to it. It then returns the set of packages for that system, which is what is bound to `pkgs`.

The `pkgs.mkShell` function creates a development shell / shell environment for us. A shell environment is actually just another derivation (that's why it is possible to run `nix develop` on packages itself). In fact, `pkgs.mkShell` is just a wrapper around `stdenv.mkDerivation`, with some conveniences like specifying `packages` instead of `nativeBuildInputs`, and concatenating `shellHook`s from all inputs, as visible in its source code. There also exists a `pkgs.mkShellNoCC`, which does the same as `pkgs.mkShell`, but does not introduce a C compiler in the shell. This is useful if you're developing projects that do not need a C compiler installed.

Let's enter this shell by running:

```
$ nix develop .#simple
Hello, world!
(nix:nix-shell-env) bash-5.2$ which cc  # a C compiler was added to thes shel
```

```
/nix/store/888bkaqdpfpx72dd8bdc69qsqlgbhcvf-gcc-wrapper-13.3.0/bin/cc
(nix:nix-shell-env) bash-5.2$ exit

$ which cc  # different C compiler!
/usr/bin/cc
```

> With `nix develop`, the flake output name is searched starting from `devShells`, then `packages`, then `legacyPackages`, and finally the whole flake.

An actual development environment

The earlier examples were really simple! Let's create an actual development environment with a *better* shell, a NodeJS install, pnpm, and even a redis database!

```
{
  description = "Nix devshells!";

  inputs = {
    nixpkgs.url = "github:nixos/nixpkgs/nixos-24.11";
  };

  outputs = {nixpkgs, ...}: let
    system = "x86_64-linux";
    pkgs = import nixpkgs { inherit system; };

    # A custom config for valkey
    # Declative configs!
    valkeyConfStr = ''
requirepass super-strong-password
port 12345
    '';

    # This special package writes the valkey configuration to
    # a text file. For more such packages, see:
    # https://nixos.org/manual/nixpkgs/stable/#trivial-builder-writeTextFile
    #
    # The package outputs the path to the file
    valkeyConf = pkgs.writeTextFile {
      name = "valkey.conf";
      text = valkeyConfStr;
    };
  in {
    devShells.${system} = {
      simple = pkgs.mkShell {
        packages = [ pkgs.hello ];
        shellHook = ''
```

```
            ${pkgs.hello}/bin/hello
          '';
        };

        default = pkgs.mkShell {
          packages = [
            # newer versions of redis are not packaged in
            # nixpkgs, so we're using valkey, an open-source
            # fork of redis maintained by the Linux Foundation
            pkgs.valkey

            pkgs.nodejs_22
            pkgs.nodePackages.pnpm

            # the friendly interactive shell!
            pkgs.fish
          ];

          shellHook = ''
            ${pkgs.valkey}/bin/valkey-server ${valkeyConf} &
            exec fish
          '';
        };
      };
    };
}
```

The `default` shell in the above flake adds Valkey, NodeJS `22`, the PNPM package manager, and the `fish` shell to the environment. It also starts Valkey in the background through a shell hook, passing it a custom config (declared via Nix!) and runs `fish` so we're dropped in the `fish` shell instead of our login shell.

> The `exec` command replaces the currently running process with the command specified. If `fish` was run without `exec`, it would drop you into your login shell with the shell environment (or run any additional commands in the `shellHook` if there were any) after you exit fish instead of exiting the devshell as expected. With `exec`, when you exit `fish`, it'll exit the environment too.

Let's enter the shell:

```
$ nix develop
155483:C 15 Jan 2025 15:56:27.317 * oO0OoO0OoO0Oo Valkey is starting oO0OoO0O
155483:C 15 Jan 2025 15:56:27.317 * Valkey version=8.0.1, bits=64, commit=000
155483:C 15 Jan 2025 15:56:27.317 * Configuration loaded
155483:M 15 Jan 2025 15:56:27.318 * monotonic clock: POSIX clock_gettime
```

```
                .+^+.
            .+##########+.
        .+########+########+.           Valkey 8.0.1 (00000000/0) 64 bit
    .+#########+'     '+#########+.
  .########+'     .+.     '+########.    Running in standalone mode
  |####+'     .+#######+.     '+####|    Port: 12345
  |###|   .+#################+.   |###|    PID: 155483
  |###|   |#####*'' ''*#####|   |###|
  |###|   |####'  .-.  '####|   |###|
  |###|   |###(  (@@@)  )###|   |###|          https://valkey.io
  |###|   |####.  '-'  .####|   |###|
  |###|   |#####*.   .*#####|   |###|
  |###|   '+######|   |######+'   |###|
  |####+.     +##|   |#+'     .+####|
  '#######+   |##|        .+########'
     '+###|   |##|    .+#########+'
         '|   |####+#########+'
              +##########+'
                 '+v+'

155483:M 15 Jan 2025 15:56:27.320 * Server initialized
155483:M 15 Jan 2025 15:56:27.320 * Loading RDB produced by Valkey version 8.
155483:M 15 Jan 2025 15:56:27.320 * RDB age 2934 seconds
155483:M 15 Jan 2025 15:56:27.320 * RDB memory usage when created 0.91 Mb
155483:M 15 Jan 2025 15:56:27.320 * Done loading RDB, keys loaded: 0, keys ex
155483:M 15 Jan 2025 15:56:27.320 * DB loaded from disk: 0.001 seconds
155483:M 15 Jan 2025 15:56:27.320 * Ready to accept connections tcp

> ps
    PID TTY          TIME CMD
 153101 pts/3    00:00:00 fish  # this is login shell
 153205 pts/3    00:00:04 fish  # this is the fish process started by `nix de
 155483 pts/3    00:00:00 valkey-server  # valkey is running in the backgroun
 155654 pts/3    00:00:00 ps
```

And you see that valkey has started in the background! To exit valkey, you need to kill it yourself, like so:

```
> kill 155483
```

> If valkey isn't killed, it'll continue running in the background, even *after* you exit the devshell!

If you want to kill valkey when you exit the shell, you can change your `shellHook` to kill the valkey process on exit:

```
  shellHook = ''
    ${pkgs.valkey}/bin/valkey-server ${valkeyConf} &
    fish
    echo Killing valkey server
    # `ps` lists all processes, `grep` searches for a line with `valkey-server`
    # in it. `awk` grabs the first column (the PID) from the output, and
    # `xargs` sends stdin as arguments to `kill`.
    ps | grep valkey-server | awk "{printf \$1}" | xargs kill
    exit
  '';
```

> **Improvement needed:** If you have another way to achieve this, then please let me know in the comments!

We can't use `exec` anymore, since we need to execute commands after `fish` exits.

Now running `nix develop` starts valkey, drops you into a shell, and when you exit the shell, valkey will automatically be killed!

Pinning package versions:

This usually isn't required, since flakes are already pinned to exact commits via. the `flake.lock` file, which *should* be checked into version control. So even if you enter a development shell ten years from now, you'll have the same version of all packages as the `flake.lock` (provided that GitHub still exists :P).

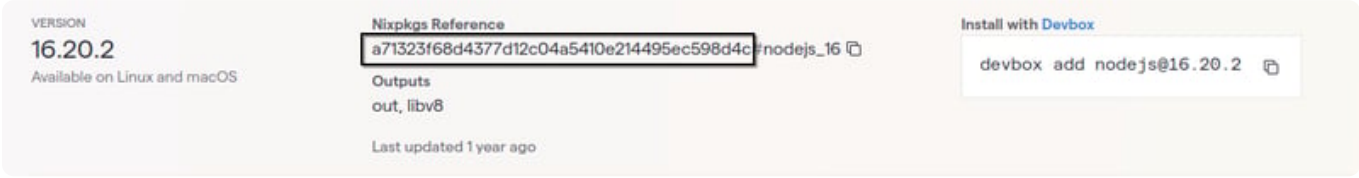
To update package versions, you can run `nix flake update`, which will fetch the latest commit of the branch of all your inputs.

But if you still want a specific version of a package, down to the patch version, you can use a `nixpkgs` input that has exactly that package. Do note that you *may have to build that package yourself* if the Nix build cache doesn't have it, as is common with quite old versions of packages.

> Before doing that, do search nixpkgs for alternate versions of packages that may exist, for example, nixpkgs has node 18, 20, and 22 in it. Many popular packages are split into different nixpkgs for major versions.

With the disclaimers out of the way, let's learn how you can pin a specific version of a package. For this example, we'll add Node 16 (an end-of-life version that you totally **shouldn't** be using) to the environment.

First, we need to find a commit of the `nixpkgs` repo that has Node 16. The latest commit should be enough. There are tools like NixHub that help with exactly that. If you check the page for `nodejs` on NixHub, you can see all the versions of node that were ever published on `nixpkgs`. Copy the `nixpkgs` commit for `nodejs 16.20.2`, i.e. the part in "Nixpkgs Reference", except the flake output name, but do keep it in mind (`#nodejs_16`), which happens to be `a71323f68d4377d12c04a5410e214495ec598d4c`.



Now, add that to our environment like so:

```
{
  description = "Nix devshells!";

  inputs = {
    nixpkgs.url = "github:nixos/nixpkgs/nixos-24.11";

    # NEW
    nodejs_16_nixpkgs.url = "github:nixos/nixpkgs/a71323f68d4377d12c04a5410e2
  };

  outputs = {nixpkgs, nodejs_16_nixpkgs, ...}: let
    system = "x86_64-linux";
    pkgs = import nixpkgs { inherit system; };

    # NEW
    nodejs_16 = (import nodejs_16_nixpkgs { inherit system; }).nodejs_16;

    # ...
  in {
    devShells.${system} = {
      # ...
      default = pkgs.mkShell {
        packages = [
          pkgs.valkey

          # NEW: replaced node 22 and pnpm with node 16
          nodejs_16

          pkgs.fish
        ];

        # ...
```

```
        };
      };
    };
  }
```

Running `nix develop` will actually give us an error claiming that this version of Node is end-of-life, which it is.

```
$ nix develop
error:
       … while calling the 'derivationStrict' builtin
         at <nix/derivation-internal.nix>:34:12:
           33|
           34|   strict = derivationStrict drvAttrs;
             |            ^
           35|

       … while evaluating derivation 'nix-shell'
         whose name attribute is located at /nix/store/2csx2kkb2hxyxhhmg2xs9j

       … while evaluating attribute 'nativeBuildInputs' of derivation 'nix-sh
         at /nix/store/2csx2kkb2hxyxhhmg2xs9jfyypikwwk6-source/pkgs/stdenv/ge
          379|       depsBuildBuild              = elemAt (elemAt dependencie
          380|       nativeBuildInputs           = elemAt (elemAt dependencie
             |       ^
          381|       depsBuildTarget             = elemAt (elemAt dependencie

       (stack trace truncated; use '--show-trace' to show the full, detailed

       error: Package ‘nodejs-16.20.2’ in /nix/store/bxygxxbgcc7s82wn8a8wdp4g


       Known issues:
        - This NodeJS release has reached its end of life. See https://nodejs

       You can install it anyway by allowing this package, using the
       following methods:

       a) To temporarily allow all insecure packages, you can use an environm
          variable for a single invocation of the nix tools:

            $ export NIXPKGS_ALLOW_INSECURE=1

          Note: When using `nix shell`, `nix build`, `nix develop`, etc with
                then pass `--impure` in order to allow use of environment var
```

```
    b) for `nixos-rebuild` you can add ‘nodejs-16.20.2’ to
       `nixpkgs.config.permittedInsecurePackages` in the configuration.nix
       like so:

         {
           nixpkgs.config.permittedInsecurePackages = [
             "nodejs-16.20.2"
           ];
         }

    c) For `nix-env`, `nix-build`, `nix-shell` or any other Nix command yo
       ‘nodejs-16.20.2’ to `permittedInsecurePackages` in
       ~/.config/nixpkgs/config.nix, like so:

         {
           permittedInsecurePackages = [
             "nodejs-16.20.2"
           ];
         }
```

Fortunately, the error also provides an easy fix:

```
$ NIXPKGS_ALLOW_INSECURE=1 nix develop --impure
# valkey output truncated ...
# you can append > /dev/null to the valkey command in shellHook
# to supress this output

$ node -v
v16.20.2
```

Success! You're now using an old, unsupported version of node!

> Be warned! If you're reading this article in the future, and decide to use this version of NodeJS, then don't be surprised if you have to build NodeJS from scratch! (It's not a fast build, by the way).

Automatic shell environments with `direnv`

A devshell discussion with Nix is not complete without introducing `direnv`. `direnv` is a tool that *very simply* runs commands based on the current directory. It is tedious to run `nix develop` to get into a development shell every time. It's also tedious to remember to exit the dev shell when you're done. `direnv` automatically does this for you, so it's a valuable addition! It also allows you to use these shells within non-terminal editors like VSCode and JetBrains.

Just a small fix:

There's a small thing we need to change in `flake.nix`. It's quite a bad idea to start Valkey every time the devshell is opened, since more than one instance of a dev shell can exist at the same time. Instead, wrap the valkey command

```
{
  # ...
  outputs = {nixpkgs, nodejs_16_nixpkgs, ...}: let
    # ...

    # NEW
    valkeyScript = pkgs.writeShellScriptBin
      "start-valkey"
      ''
        ${pkgs.valkey}/bin/valkey-server ${valkeyConf}
      '';
  in {
    devShells.${system} = {
      # ...
      default = pkgs.mkShell {
        packages = [
          pkgs.valkey

          nodejs_16

          pkgs.fish
        ];

        shellHook = ''
          # NEW: Replace valkey startup and shutdown code with this
          export PATH="${valkeyScript}/bin:$PATH"
          # NEW: Add exec back
          exec fish
        '';
      };
    };
  };
}
```

This change adds a shell script called `start-valkey` to the environment, which runs valkey with the specified configuration. Now, to start valkey, you just run `start-valkey`, and press `Ctrl-C` to stop it. This allows us to have multiple shells in that environment now.

Install `direnv`:

`direnv` is packaged in most linux distros, but since this is a Nix guide after all, let's install it via. nix by running this command:

```
nix profile install nixpkgs#direnv
```

You also need to hook `direnv` into your shell, since `direnv` needs to know when you've changed directories. Follow the instructions on this page for your shell. For `bash`, you need to add `eval "$(direnv hook bash)"` to the end of your `~/.bashrc`, and restart your terminal after that.

To tell `direnv` we want it to start the development shell when we `cd` into this folder, add an `.envrc` in the folder, with the following contents:

```
# Need to set `stty sane` for fish
# https://github.com/direnv/direnv/issues/967#issuecomment-1987134113
if [[ $SHELL =~ "fish" ]]; then
  stty sane
fi

use flake
```

This will make `direnv` automatically enter your development environment. Just run `direnv allow`, to allow executing this `.envrc`, and you're automatically dropped into the development environment, with glorious Node 16 available!

> If you're using `fish` as your login shell, remove `exec fish` from the `shellHook` in the flake, otherwise `direnv` will go on an infinite loop, since the `fish` spawned by the devshell spawns `direnv` which sees that it should enter the devshell, which spawns a `fish`, which spawns a `direnv`, ... and so on.

Other alternatives to development shells built with Nix

If writing a devshell on your own seems more complicated than necessary, you can use tools like Devenv or Devbox (by the same team that built NixHub), which are both built on Nix. Devenv provides nice wrappers to automatically add languages, services (like postgres or redis), etc. on top of your flake, without having to do the shenanigans we had to do with Valkey. Devbox on the other hand, lets you skip writing Nix entirely, since they have their own CLI and lock file that pull packages from nixpkgs.

Conclusion

You've now learned one of the most powerful features of nix! In the next few articles, we'll cover packaging for Nix!

If you really liked this article and would like to support me, here are some ways:

- GitHub Sponsors
- Use my DigitalOcean referral link
- Use my Prisma referral link

Thank you so much!

Nix (3 Part Series)

1 My new Nix series!

2 Getting started with Nix and Nix Flakes

3 **Easy development environments with Nix and Nix flakes!**

Top comments (0)

arnu515

JOINED

10 черв. 2020 р.

More from arnu515

Getting started with Nix and Nix Flakes

#nix #linux #raspberrypi #tutorial

My new Nix series!

#nix #linux #devops

ChatCrafters - Chat with AI powered personas

#cloudflarechallenge #devchallenge #ai #webdev