Nixifying a Haskell project using nixpkgs

Welcome to the Nixify Haskell projects series, where we start our journey by integrating a Haskell application, particularly one using a PostgreSQL database, into a single-command deployable package. By the end of this article, you'll have a flake.nix file that's set to build the project, establish the development environment, and execute the Haskell application along with all its dependent services like PostgreSQL and PostgREST. We'll be using todo-app as a running case study throughout the series, demonstrating the process of building a Haskell project and effectively managing runtime dependencies, such as databases and other services, thereby illustrating the streamlined and powerful capabilities Nix introduces to Haskell development.

Pre-requisites

- A basic understanding of the Nix and Flakes is assumed. See Rapid Introduction to Nix
- To appreciate why Nix is a great choice for Haskell development, see Why Choose Nix for develoment?

Nixify Haskell package

Let's build a simple flake for our Haskell project, `todo-app`. Start by cloning the todo-app repository and checking out the specified commit.

```
git clone https://github.com/juspay/todo-app.git
cd todo-app
git checkout 076185e34f70e903b992b597232bc622eadfcd51
```

Here's a brief look at the `flake.nix` for this purpose:

```
{
  inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs/nixpkgs-unstable";
  };
  outputs = { self, nixpkgs }:
    let
```

```
      system = "aarch64-darwin";
      pkgs = nixpkgs.legacyPackages.${system};
      overlay = final: prev: {
        todo-app = final.callCabal2nix "todo-app" ./. { };
      };
      myHaskellPackages = pkgs.haskellPackages.extend overlay;
    in
    {
      packages.${system}.default = myHaskellPackages.todo-app;
      apps.${system}.default = {
        type = "app";
        program = "${self.packages.${system}.default}/bin/todo-app";
      };
    };
}
```

Now, let's dissect it.

haskellPackages

The official manual explains the Haskell's infrastructure in nixpkgs detail. For our purposes, the main things to understand are:

- `pkgs.haskellPackages` is an attribute set containing all Haskell packages within `nixpkgs`.
- We can "extend" this package set to add our own Haskell packages. This is what we do when creating `myHaskellPackages`.
- We add the `todo-app` package to `myHaskellPackages` (a package set derived from `pkgs.haskellPackages`), and then use that when defining the flake package, `packages.${system}.default`, below.

Exploring `pkgs.haskellPackages`

You can use nix repl to explore any flake's output. In the repl session below, we locate and build the `aeson` package:

```
nix repl github:nixos/nixpkgs/nixpkgs-unstable
nix-repl> pkgs = legacyPackages.${builtins.currentSystem}

nix-repl> pkgs.haskellPackages.aeson
«derivation /nix/store/sjaqjjnizd7ybirh94ixs51x4n17m97h-aeson-2.0.3.0.drv

nix-repl> :b pkgs.haskellPackages.aeson

This derivation produced the following outputs:
```

```
    doc -> /nix/store/xjvm45wxqasnd5p2kk9ngcc0jbjhx1pf-aeson-2.0.3.0-doc
    out -> /nix/store/1dc6b11k93a6j9im50m7qj5aaa5p01wh-aeson-2.0.3.0
```

callCabal2nix

We used `callCabal2nix` function from nixpkgs to build the `todo-app` package above. This functio generates a Haskell package Derivation from its source, utilizing the "cabal2nix" program to convert a cabal file into a Nix derivation.

Overlay

Info

- NixOS Wiki on Overlays
- Overlay implementation in fixed-points.nix>

To *extend* the `pkgs.haskellPackages` package set above, we had to pass what is known as an "overlay". This allows us to either override an existing package or add a new one.

In the repl session below, we extend the default Haskell package set to override the `shower` package to be built from the Git repo instead:

```
nix-repl> :b pkgs.haskellPackages.shower

This derivation produced the following outputs:
  doc -> /nix/store/crzcx007h9j0p7qj35kym2rarkrjp9j1-shower-0.2.0.3-doc
  out -> /nix/store/zga3nhqcifrvd58yx1l9aj4raxhcj2mr-shower-0.2.0.3

nix-repl> myHaskellPackages = pkgs.haskellPackages.extend
    (self: super: {
       shower = self.callCabal2nix "shower"
         (pkgs.fetchgit {
            url = "https://github.com/monadfix/shower.git";
            rev = "2d71ea1";
            sha256 = "sha256-vEck97PptccrMX47uFGjoBVSe4sQqNEsclZOYfEMTns=";
         }) {};
    })

nix-repl> :b myHaskellPackages.shower

This derivation produced the following outputs:
```

```
doc -> /nix/store/vkpfbnnzyywcpfj83pxnj3n8dfz4j4iy-shower-0.2.0.3-doc
out -> /nix/store/55cgwfmayn84ynknhg74bj424q8fz5rl-shower-0.2.0.3
```

Notice how we used `callCabal2nix` to build a new Haskell package from the source (located in the specified Git repository).

Putting It All Together

```
Missing: COMMAND

Usage: todo-app [--version] COMMAND

  optparse subcommands example

todo-app (076185e) [!+] via λ
❯ nix run .#default -- --help
warning: Git tree '/Users/shivaraj/demo/todo-app' is dirty
optparse-sub-example - a small example program for optparse-applicative with
```

Nixifying Development Shells

Our existing flake lets us *build* `todo-app`. But what if we want to *develop* it? Typically, Haskell development involves tools like cabal and ghcid. These tools require a GHC environment with the packages specified in the `build-depends` of our cabal file. This is where `devShell` comes in, providing an isolated environment with all packages required by the project.

Here's the `flake.nix` for setting up a development shell:

```
{
  inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs/nixpkgs-unstable";
  };
  outputs = { self, nixpkgs }:
    let
      system = "aarch64-darwin";
      pkgs = nixpkgs.legacyPackages.${system};
      overlay = final: prev: {
        todo-app = final.callCabal2nix "todo-app" ./. { };
      };
      myHaskellPackages = pkgs.haskellPackages.extend overlay;
    in
    {
      devShells.${system}.default = myHaskellPackages.shellFor {
        packages = p : [
          p.todo-app
        ];
        nativeBuildInputs = with myHaskellPackages; [
          ghcid
```

```
            cabal-install
          ];
        };
      };
  }
```

shellFor

A Haskell devShell can be provided in one of the two ways. The default way is to use the (language-independent) `mkShell` function (Generic shell). However to get full IDE support, it is best to use the (haskell-specific) `shellFor` function, which is an abstraction over `mkShell` geared specifically for Haskell development shells

- Every Haskell package set (such as `pkgs.haskellPackages`), exposes `shellFor` function, which returns a devShell with GHC package set configured with the Haskell packages in that package set.
- As arguments to `shellFor` - generally, we only need to define two keys `packages` and `nativeBuildInputs`.
  - `packages` refers to *local* Haskell packages (that will be compiled by cabal rather than Nix).
  - `nativeBuildInputs` refers to programs to make available in the `PATH` of the devShell.

Let's run!

```
    6  import qualified TodoApp.Request as TR
    5
    4  newtype Opts = Opts {optCommand :: Command}
    3
-   2  data Command
-   1    = Add String
-  18    | Delete Int
-   1    | Done Int
-   2    | View
-   3    | ViewAll
```

```
❯ nix develop
warning: Git tree '/User

todo-app (076185e) [!+]
0-env)
❯ which cabal
/nix/store/2lb3hv03annkc

todo-app (076185e) [!+]
0-env)
```

Nixifying External Dependencies

We looked at how to package a Haskell package, and thereon how to setup a development shell. Now we come to the final part of this tutorial, where we will see how to package external dependencies (like Postgres). We will demonstrate how to initiate a Postgres server using Nix without altering the global system state.

Here's the `flake.nix` for making `nix run .#postgres` launch a Postgres server:

```
{
  inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs/nixpkgs-unstable";
  };
  outputs = { self, nixpkgs }:
  let
    system = "aarch64-darwin";
    pkgs = nixpkgs.legacyPackages.${system};
  in
  {
    apps.${system}.postgres = {
      type = "app";
      program =
        let
          script = pkgs.writeShellApplication {
            name = "pg_start";
            runtimeInputs = [ pkgs.postgresql ];
            text =
            ''
              # Initialize a database with data stored in current project dir
              [ ! -d "./data/db" ] && initdb --no-locale -D ./data/db

              postgres -D ./data/db -k "$PWD"/data
            '';
          };
        in "${script}/bin/pg_start";
    };
  };
}
```

This flake defines a flake app that can be run using `nix run`. This app is simply a shell script that starts a Postgres server. nixpkgs provides the convenient `writeShellApplication` function to generate such a script. Note that `"${script}"` provides the path in the `nix/store` where the application is located.

Run it!

```
todo-app (076185e) [!+] via λ
❯ psql
zsh: command not found: psql

todo-app (076185e) [!+] via λ
❯ # We need a shell with psql

todo-app (076185e) [!+] via λ
❯ nix run nixpkgs#
```

```
selecting default max_co
selecting default shared
selecting default time z
creating configuration f
running bootstrap script
performing post-bootstra
syncing data to disk ...

initdb: warning: enablin
You can change this by e
```

Combining All Elements

Now it's time to consolidate all the previously discussed sections into a single `flake.nix`. Additionally, we should incorporate the necessary apps for `postgrest` and `createdb`. `postgrest` app will start the service and `createdb` will handle tasks such as loading the database dump, creating a database user, and configuring the database for postgREST.

```
{
  inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs/nixpkgs-unstable";
  };
  outputs = { self, nixpkgs }:
    let
      system = "aarch64-darwin";
      pkgs = nixpkgs.legacyPackages.${system};
      overlay = final: prev: {
        todo-app = final.callCabal2nix "todo-app" ./. { };
      };
      myHaskellPackages = pkgs.haskellPackages.extend overlay;
    in
    {
      packages.${system}.default = myHaskellPackages.todo-app;

      devShells.${system}.default = myHaskellPackages.shellFor {
        packages = p: [
          p.todo-app
        ];
        buildInputs = with myHaskellPackages; [
          ghcid
          cabal-install
          haskell-language-server
        ];
      };

      apps.${system} = {
        default = {
          type = "app";
          program = "${self.packages.${system}.default}/bin/todo-app";
        };
        postgres = {
```

```
          type = "app";
          program =
            let
              script = pkgs.writeShellApplication {
                name = "pg_start";
                runtimeInputs = [ pkgs.postgresql ];
                text =
                  ''
                    # Initialize a database with data stored in current projec
                    [ ! -d "./data/db" ] && initdb --no-locale -D ./data/db

                    postgres -D ./data/db -k "$PWD"/data
                  '';
              };
            in
            "${script}/bin/pg_start";
        };
        createdb = {
          type = "app";
          program =
            let
              script = pkgs.writeShellApplication {
                name = "createDB";
                runtimeInputs = [ pkgs.postgresql ];
                text =
                  ''
                    # Create a database of your current user
                    if ! psql -h "$PWD"/data -lqt | cut -d \| -f 1 | grep -qw
                      createdb -h "$PWD"/data "$(whoami)"
                    fi

                    # Load DB dump
                    psql -h "$PWD"/data < db.sql

                    # Create configuration file for postgrest
                    echo "db-uri = \"postgres://authenticator:mysecretpassworc
                    db-schemas = \"api\"
                    db-anon-role = \"todo_user\"" > data/db.conf
                  '';
              };
            in
            "${script}/bin/createDB";
        };
        postgrest = {
          type = "app";
          program =
            let
              script = pkgs.writeShellApplication {
                name = "pgREST";
                runtimeInputs = [ myHaskellPackages.postgrest ];
                text =
                  ''
```

```
                    postgrest ./data/db.conf
                  '';
                };
              in
              "${script}/bin/pgREST";
            };
          };
        };
      }
```

For the complete souce code, visit here.

> forAllSystems
>
> The source code uses `forAllSystems`, which was not included in the tutorial above to maintain simplicity. Later, we will obviate `forAllSystems` and simplify the flake further using flake-parts.

Video Walkthrough

```
❯ # For vscode you will have to enter devShell and open `code` from the shell and you will have HLS

todo-app (076185e) [+] via λ
❯ # Assuming you have installed Haskell extensions

todo-app (076185e) [+] via λ
❯ hx flake.nix

todo-app (076185e) [!+] via λ took 7s
❯ # First let's see HLS
```

Conclusion

This tutorial pratically demonstrated why Nix is a great choice for Haskell development:

- **Instantaneous Onboarding**: There is no confusion about how to setup the development environment. It is `nix run .#postgres` to start the postgres server, `nix run .#createdb` to setup the database and `nix run .#postgrest` to start the Postgrest web server. This happens in a reproducible way, ensuring every developer gets the same environment.
- **Boosted Productivity**: The commands mentioned in the previous points in conjunction with `nix develop` is all that is needed to make a quick change and see it in effect.
- **Multi-Platform Support**: All the commands mentioned in the previous points will work in the same way across platforms.

In the next tutorial part, we will modularize this `flake.nix` using flake-parts.

Links to this page

direnv: manage dev environments

> Since both nixpkgs and haskell-flake use Nix expressions that read the `.cabal` file to get dependency information, you will want the devshell be recreated every time a `.cabal` file changes. This can be achieved using the `watch_file` function. Modify your `.envrc` to contain:

Nixify Haskell projects

> ☑ Nixifying a Haskell project using nixpkgs