Runnable Flakes (Nix From First Principles: Flake Edition #9)

Nov 10
2022

#Nix

This is part 9 of the Nix from First Principles: Flake Edition series.

In the last two parts, I covered using flakes for two different types of output: packages and developer environments. This time I'll cover a third type of output - runnable commands. These are what power the `nix run` command that you've seen a few times throughout the series, and now you will find out how to get this functionality for your own flakes.

Automatic `nix run` from packages

The first way that you can get a run configuration from your flake actually may not require you to do anything at all. If you don't add a custom run configuration to your flake, then it'll first build the package for your selected flake output (or `default` if you don't choose one). Then it'll look in that package's binary folder for a binary which has a name matching the package name.

To see this in action, let's focus in on the flake from part 7. For a reminder, here is the flake code, with two extra comments to highlight the relevant parts:

`flake.nix`

```
{
  inputs.nixpkgs.url = "github:NixOS/nixpkgs/nixos-22.05";

  outputs = {
    self,
    nixpkgs,
  }: let
    system = "x86_64-linux";
    pkgs = import nixpkgs {inherit system;};
  in {
    packages.${system}.default =
      pkgs.stdenv.mkDerivation
      {
        src = ./rust-hello;
        # The package name without the version is "rust-hello"
        name = "rust-hello-1.0";
        inherit system;
        nativeBuildInputs = [pkgs.cargo];
        buildPhase = ''
          cargo build --release
```

```
          '';
          # The binary output is at $out/bin/rust-hello
          installPhase = ''
            mkdir -p $out/bin
            cp target/release/rust-hello $out/bin/rust-hello
            chmod +x $out
          '';
        };
    };
}
```

Because building this package produces a binary called `rust-hello` in `$out/bin`, and this matches the package name of `rust-hello`, if you `nix run` this flake, it will already work. You can try it with the following command:

```
> nix run "gitlab:tonyfinn/nix-guide?dir=7-flakes/simple"
Hello, world!
```

This is also how running cargo from nixpkgs worked in part 7 when that `cargo init` command was used.

Customizing the binary to run

However, if you try the same with the debug package:

```
> nix run "gitlab:tonyfinn/nix-guide?dir=7-flakes/multiple-outputs#
error: unable to execute '/nix/store/12jm5znakhc7qccfhld4ak5l71airc
```

Because the package name of this variant is `rust-hello-debug`, it looks for a binary with that name, but the actual binary produced is called `rust-hello`. Since Nix's guessing fails here, you can tell it what the "main" binary is by setting `meta.mainProgram` on the derivation.

```
# Rest of flake omitted
{
    debug = pkgs.stdenv.mkDerivation {
        src = ./rust-hello;
        name = "rust-hello-debug-1.0";
        inherit system;
        nativeBuildInputs = [pkgs.cargo];
        meta.mainProgram = "rust-hello";
        buildPhase = ''
            cargo build
        '';
```

```
            installPhase = ''
                mkdir -p $out/bin
                cp target/debug/rust-hello $out/bin/rust-hello
                chmod +x $out
            '';
        };
    }
```

Full flake

The `meta` attribute provides information about a derivation that is not used by the derivation itself, but by other programs or people looking for information on it. One of these programs is `nix run`.

Now if you try running this updated version, you should see the following:

```
# Or just `nix run` to run your local copy
> nix run "gitlab:tonyfinn/nix-guide?dir=9-runnable-flakes/runnable
Hello, world!
```

Completely custom run configurations

`meta.mainProgram` is a great option when you have a package to run which is named differently to its binary, but in some situations you may want even more control over what happens. For example, you might want to add a run configuration to run a static web server for dev purposes.

To add a run configuration not tied to a built package, you can use the `apps` key of a flake. This takes two keys, `type`, and `program`. `type` must be set to `app` - at the moment this is the only supported type of run configuration. `program` is set to the path to a binary to execute.

This latter argument poses a complication - the binary is invoked with only the user specified arguments, but for this kind of usage you may want to prefill some or all arguments. One solution to this is to use `pkgs.createShellApplication` to make your command into a shell script.

For example:

```
{
    apps.${system}.default = let
      serv = pkgs.writeShellApplication {
        # Our shell script name is serve
        # so it is available at $out/bin/serve
        name = "serve";
        # Caddy is a web server with a convenient CLI interface
```

```
          runtimeInputs = [pkgs.caddy];
          text = ''
            # Serve the current directory on port 8090
            caddy file-server --listen :8090 --root .
          '';
        };
      in {
        type = "app";
        # Using a derivation in here gets replaced
        # with the path to the built output
        program = "${serv}/bin/serve";
      };
  }
```

Full flake

From here you can run `nix run` and visit http://localhost:8090/flake.nix, where you should see the contents of your flake.

That's it for this section, next time I'll cover some smaller extra features of flakes, including standard places to put tests and ways to make flakes that run on multiple systems more easily.