PDF

Package parameters and overrides with callPackage

Contents

Nix ships with a special-purpose programming language for creating packages and configurations: the Nix language. It is used to build the Nix package collection, known as Nixpkgs.

Being purely functional, the Nix language allows declaring custom functions to abstract over common patterns. One of the most prominent patterns in Nixpkgs is parametrisation of package recipes.

Overview

Nixpkgs is a sizeable software project on its own, with coding conventions and idioms that have emerged over the years. It has established a convention of composing parameterised packages with automatic settings through a function named `callPackage`. This tutorial shows how to use it and why it's beneficial.

What will you learn?

- Using `callPackage` to invoke package recipes that follow Nixpkgs conventions
- Overriding package parameters
- Creating interdependent package sets

What do you need?

Skip to main content

- First experience with packaging existing software

How long does it take?

- 45 minutes

Automatic function calls

Create a new file `hello.nix`, which could be a typical package recipe as found in Nixpkgs: A function that takes an attribute set, with attributes corresponding to derivations in the top-level package set, and returns a derivation.

hello.nix

```
{ writeShellScriptBin }:
writeShellScriptBin "hello" ''
  echo "Hello, world!"
''
```

Detailed explanation

Now create a file `default.nix` with the following contents:

default.nix

```
let
  pkgs = import <nixpkgs> { };
in
pkgs.callPackage ./hello.nix { }
```

Realise the derivation in `default.nix` and run the executable that is produced:

```
$ nix-build
$ ./result/bin/hello
Hello, world!
```

The argument `writeShellScriptBin` gets filled in automatically when the function in `hello.nix` is evaluated. For every attribute in the function's argument, `callPackage` passes an attribute from the `pkgs` attribute set if it exists.

Skip to main content

It may appear cumbersome to create the extra file `hello.nix` for the package in such a simple setup. We have done so because this is exactly how Nixpkgs is organised: Every package recipe is a file that declares a function. This function takes as arguments the package's dependencies.

Parameterised builds

Change the `default.nix` to produce an attribute set of derivations, with the attribute `hello` containing the original derivation:

default.nix

```
let
  pkgs = import <nixpkgs> { };
in
{
  hello = pkgs.callPackage ./hello.nix { };
}
```

When building the attribute `hello`, by accessing it with the `-A` / `--attr` option, the result will be the same as before:

```
$ nix-build -A hello
$ ./result/bin/hello
Hello, world!
```

Also change `hello.nix` to add an additional parameter `audience` with default value `"world"`:

hello.nix

```
{
  writeShellScriptBin,
  audience ? "world",
}:
writeShellScriptBin "hello" ''
  echo "Hello, ${audience}!"
''
```

This also does not change the result.

Things get more interesting when changing `default.nix` to make use of this new

Skip to main content

default.nix

```
 let
   pkgs = import <nixpkgs> { };
 in
 {
-  hello = pkgs.callPackage ./hello.nix { };
+  hello = pkgs.callPackage ./hello.nix { audience = "people"; };
 }
```

This attribute is passed on to the argument of the function defined in `hello.nix`: The same syntax can also be used to explicitly set the automatically discovered arguments, such as `writeShellScriptBin`, but that doesn't make sense here.

Try it out:

```
$ nix-build -A hello
$ ./result/bin/hello
Hello, people!
```

This pattern is used widely in Nixpkgs: For example, functions which represent Go programs often have a parameter `buildGoModule`. It is common to find expressions like `callPackage ./go-program.nix { buildGoModule = buildGo116Module; }` to change the default Go compiler version. Nixpkgs is therefore not simply a huge library of pre-configured packages, but a collection of functions – package *recipes* – for customising packages and even entire ecosystems (for example "All Python packages using my custom interpreter") on the fly without duplicating code.
`callPackage` adds more convenience by allowing parameters to be customised *after the fact* using the returned derivation's `override` function.

Add a third attribute `hello-folks` to `default.nix` and set it to `hello.override` called with a new value for `audience`:

default.nix

```
 let
   pkgs = import <nixpkgs> { };
 in
-{
+rec {
   hello = pkgs.callPackage ./hello.nix { audience = "people"; };
+  hello-folks = hello.override { audience = "folks"; };
 }
```

Skip to main content

Note

The resulting attribute set is now recursive (by the keyword `rec`). That is, attribute values can refer to names from within the same attribute.

`override` passes `audience` to the original function in `hello.nix` - it *overrides* whatever arguments have been passed in the original `callPackage` that produced the derivation `hello`. All the other parameters will remain the same. This is especially useful and can be often found on packages that provide many options to customise a package.

Building `hello-folks` attribute and running the resulting executable will again produce a new version of the script:

```
$ nix-build -A hello-folks
$ ./result/bin/hello
Hello, folks!
```

A real-world example is the `neovim` package recipe, which has overridable arguments such as `extraLuaPackages`, `extraPythonPackages`, or `withRuby`. Currently these parameters are only discoverable by reading the source code, which can be found by following the link to 📦 Source on search.nixos.org/packages.

Interdependent package sets

You can actually create your own version of `callPackage`! This comes in handy for package sets where the recipes depend on each other.

Note

The following examples do not show the "called" files, as they are not necessary for understanding the principle.

Consider the following recursive attribute set of derivations:

default.nix

```
let
  pkgs = import <nixpkgs> { };
in
```

Skip to main content

```
  b = pkgs.callPackage ./b.nix { inherit a; };
  c = pkgs.callPackage ./c.nix { inherit b; };
  d = pkgs.callPackage ./d.nix { };
  e = pkgs.callPackage ./e.nix { inherit c d; };
}
```

Note

Here, `inherit a;` is equivalent to `a = a;`.

Previously declared derivations are passed as arguments to other derivations through `callPackage`.

In this case you have to remember to manually specify all arguments required by each package in the respective Nix file that are not in Nixpkgs. If `./b.nix` requires an argument `a` but there is no `pkgs.a`, the function call will produce an error. This can become quite tedious quickly, especially for larger package sets.

Use `lib.callPackageWith` to create your own `callPackage` based on an attribute set.

default.nix

```
let
  pkgs = import <nixpkgs> { };
  callPackage = pkgs.lib.callPackageWith (pkgs // packages);
  packages = {
    a = callPackage ./a.nix { };
    b = callPackage ./b.nix { };
    c = callPackage ./c.nix { };
    d = callPackage ./d.nix { };
    e = callPackage ./e.nix { };
  };
in
packages
```

This requires some explanation.

First of all note that instead of a recursive attribute set, the names we operate on are now assigned in a `let` binding. It has the same property as recursive sets: Names on the left can be used in expressions on the right of the equal sign (`=`). This is how we can refer to `packages` when we merge its contents with the pre-existing attribute set `pkgs` using the `//` operator.

Your custom `callPackages` now makes available all the attributes in `pkgs` *and* `packages` to

Skip to main content

`packages` is being built up recursively with each call.

The last bit may make your head spin. This construction is only possible because the Nix language is lazily evaluated. That is, values are only computed when they are actually needed. It allows passing `packages` around without having fully defined it.

Each package's dependencies are now implicit at this level (they are still explicit in each of the package files), and `callPackage` resolves them *automagically*. This relieves you from dealing with them manually, and precludes configuration errors that may only surface late into a lengthy build process.

Of course this small example is still manageable in the original form. And the implicitly recursive variant can obscure the structure for software developers not familiar with lazy evaluation, making it harder to read for them than it was before. But this benefit really pays off for large constructions, where it is the amount of code that would obscure the structure, and where manual modifications would become cumbersome and error-prone.

Summary

Using `callPackage` does not only follow Nixpkgs conventions, which makes your code easier to follow for experienced Nix users, but it also gives you some benefits for free:

1. Parametrized builds
2. Overrideable builds
3. Concise implementation of interdependent package sets

References

- Nixpkgs manual: `callPackageWith`

Next steps

- Working with local files - learn to package your own projects with Nix
- Module system deep dive - learn to wield the functional programming magic behind NixOS