Automatically managing remote sources with npins

Contents

The Nix language can be used to describe dependencies between files managed by Nix. Nix expressions themselves can depend on remote sources, and there are multiple ways to specify their origin, as shown in Towards reproducibility: pinning Nixpkgs.

For more automation around handling remote sources, set up `npins` in your project:

```
$ nix-shell -p npins --run "npins init --bare; npins add github nixos nixpkgs --bra
```

This command will fetch the latest revision of the Nixpkgs 23.11 release branch. In the current directory it will generate `npins/sources.json`, which will contain a pinned reference to the obtained revision. It will also create `npins/default.nix`, which exposes those dependencies as an attribute set.

Import the generated `npins/default.nix` as the default value for the argument to the function in `default.nix` and use it to refer to the Nixpkgs source directory:

```
1 {
2   sources ? import ./npins,
3   system ? builtins.currentSystem,
4   pkgs ? import sources.nixpkgs { inherit system; config = {}; overlays = []; },
5 }:
6 {
7   package = pkgs.hello;
8 }
```

`nix-build` will call the top-level function with the empty attribute set `{}`, or with the

Skip to main content

programmatically.

Add `npins` to the development environment for your project to have it readily available:

```
 {
   sources ? import ./npins,
   system ? builtins.currentSystem,
   pkgs ? import sources.nixpkgs { inherit system; config = {}; overlays = []; },
 }:
-{
+rec {
   package = pkgs.hello;
+  shell = pkgs.mkShellNoCC {
+    inputsFrom = [ package ];
+    packages = with pkgs; [
+      npins
+    ];
+  };
 }
```

Also add a `shell.nix` to enter that environment more conveniently:

```
1 (import ./. {}).shell
```

See Dependencies in the development shell for details, and note that here you have to pass an empty attribute set to the imported expression, since `default.nix` now contains a function.

Overriding sources

As an example, we will use the previously created expression with an older version of Nixpkgs.

Enter the development environment, create a new directory, and set up npins with a different version of Nixpkgs:

```
$ nix-shell
[nix-shell]$ mkdir old
[nix-shell]$ cd old
[nix-shell]$ npins init --bare
[nix-shell]$ npins add github nixos nixpkgs --branch nixos-21.11
```

Create a file `default.nix` in the new directory, and import the original one with the

Skip to main content

```
1 import ../default.nix { sources = import ./npins; }
```

This will result in a different version being built:

```
$ nix-build -A build
$ ./result/bin/hello --version | head -1
hello (GNU Hello) 2.10
```

Sources can also be overridden on the command line:

```
nix-build .. -A build --arg sources 'import ./npins'
```

Migrating from `niv`

A previous version of this guide recommended using `niv`, a similar pin manager written in Haskell.

If you have a project using `niv`, you can import remote source definitions into `npins`:

```
npins import-niv
```

> **Warning**
>
> All the imported entries will be updated, so they won't necessarily point to the same commits as before!

Next steps

- Check the built-in help for more information:

  ```
  npins --help
  ```

- For more details and examples of the different ways to specify remote sources, see Towards reproducibility: pinning Nixpkgs.