A Tutorial Introduction to Nix

Rohit Goswami · 2020-08-18 · tools · nix · workflow · python

> *Brief introduction to a nix based project workflow.*

Background

For CarpentryCon@Home 2020, along with Amrita Goswami, I am to prepare and deliver a workshop on "Reproducible Environments with the Nix Packaging System". In particular, as a community of practice lesson, the focus is not on packaging (as is typical of most Nix tutorials) nor on the Nix expression language itself, but instead on the use of Nix as a replacement for virtual environments using `mkShell`.

Materials

This is a Carpentries style single page lesson on setting up and working with Nix for reproducible environments. It was conceived to be a complimentary resource to the content of this repository, namely:

- Slides on Python packages with Nix
- Nix with R and devtools
- Statistical Rethinking and Nix
- An Etherpad
- Session recording

Ten seconds into Nix

A few words to keep in mind, in no particular order.

- Nix is based of good academic principles by Dolstra, de Jonge, and Visser (2004) and Dolstra, Löh, and Pierron (2010)
  - It has been used in large scientific projects for reproducibility (e.g. d-SEAMS of Goswami, Goswami, and Singh (2020))
- The Nix expression language is a **domain specific language**
  - Turing completeness is not a goal or a requirement
- Can leverage binary caches
  - Not always true, only when installed in `/nix`

Setup

For this particular tutorial, we will assume the standard Nix installation proceedure, that is, one where the installer has root access to create the initial `/nix` directory and set up the build users[^fn:1]. This follows directly from the Nix Manual:

```
# You need root permissions for this!!!
sh <(curl -L https://nixos.org/nix/install) --daemon
```

At this point we will also install the canonnical first package, the `hello` package, which simply outputs a friendly greeting.

```
nix-env -i hello
```

Note that the basic package search operation is `nix search` and it gives outputs which look like:

```
* nixpkgs.auctex (auctex-12.2)
  Extensible package for writing and formatting TeX files in GNU Emacs and XEma
cs

* nixpkgs.cask (cask-0.8.4)
  Project management for Emacs

* nixpkgs.emacs (emacs)
  The extensible, customizable GNU text editor

* nixpkgs.emacs-all-the-icons-fonts (emacs-all-the-icons-fonts)
  Icon fonts for emacs all-the-icons

* nixpkgs.emacs-nox (emacs-nox)
  The extensible, customizable GNU text editor

* nixpkgs.emacs25 (emacs)
  The extensible, customizable GNU text editor

* nixpkgs.emacs25-nox (emacs-nox)
  The extensible, customizable GNU text editor

* nixpkgs.emacs26 (emacs)
(stdin) 22/72 [29%]
 mS  1 zsh 2 make  3  nix                                  14:57  Tue 18 Aug
```

Figure 1: The `nix search emacs` output

Though this is not bad by any standard, we will try to get a more interactive management tool.

Basic Helpers

The first few things to obtain are:

nox

This is a better package management helper

niv

For pinning dependencies as discussed later

lorri

For working smoothly with project environments

Exercise 1

> *Try installing these and use `nox emacs` to test the output*

```
35 pydb-1.26 (unstable.pydb)
    Python debugger with GDB-like commands and Emacs bindings
36 python2.7-epc-0.0.5 (unstable.python27Packages.epc)
    EPC (RPC stack for Emacs Lisp) implementation in Python
37 python3.7-epc-0.0.5 (unstable.python37Packages.epc)
    EPC (RPC stack for Emacs Lisp) implementation in Python
38 python3.8-epc-0.0.5 (unstable.python38Packages.epc)
    EPC (RPC stack for Emacs Lisp) implementation in Python
39 tydra-1.0.2 (unstable.tydra)
    Shortcut menu-based task runner, inspired by Emacs Hydra
40 uemacs-2014-12-08 (unstable.uemacs)
    Torvalds Micro-emacs fork
41 vimplugin-vimacs-2016-03-24 (unstable.vimPlugins.vimacs)
    Vim-Improved eMACS: Emacs emulation plugin for Vim
42 vimacs-8.2.0701 (unstable.vimacs)
    Vim-Improved eMACS: Emacs emulation for Vim
43 webmacs-0.8 (unstable.webmacs)
    Keyboard-based web browser with Emacs/conkeror heritage
44 zile-2.4.14 (unstable.zile)
    Lightweight Emacs clone
Packages to install: ^CAborted!
```

Figure 2: The `nox emacs` output

More Dependable Dependencies

Standard Channels

Nix works by searching a repository (local or online) of package derivations. Indeed, we can pass `nix-env` any local fork of the main nixpkgs repo as well.

```
# don't run this, it is a large repo
git clone https://github.com/NixOS/nixpkgs.git $mynixdir
# make changes..
$EDITOR $nixpkgs/pkgs/applications/editors/emacs/default.nix
nix-env -i emacs -f $nixpkgs
```

- This might serve as a way to mass modify the `nixpkgs` in a pinch
  - However, we will almost **never** use this in practice
  - The overlay approach is much better
- It is also useful if we need to build local derivations

Pinning Dependencies

Compared to globally tracking the branches of `nixpkgs` or even local changes and forks, for project oriented workflows it is better to use `niv` which we obtained previously. In a nutshell, `niv` will generate a `json` file to keep track of dependencies and wraps it in a `nix` file we can subsequently import and use.

Project Setup

We are now in a position to start working with a project oriented workflow.

```
# Make directories
mkdir myFirstNix
cd myFirstNix
# Setup
niv init
niv update nixpkgs -b nixpkgs-unstable
```

At this stage your project should have the following structure:

```
tree myFirstNix
```

myFirstNix
└── nix
├── sources.json
└── sources.nix

1 directory, 2 files

We can now move on to the heart of this tutorial, the `nix-shell`. In a nutshell, running `nix-shell` when there is a defined `shell.nix` will spawn a virtual environment with the nix packages requested.

lorri and direnv

Though we haven't as yet generated a `shell.nix` we should point out that writing one by hand will mean that we need to rebuild the enviroment when we make changes using `nix-shell` every time. A more elegant approach is to offload the rebuilding of the environment to `lorri` which also has a neat `direnv` integrration. Let's try that out.

```
cd myFirstNix
lorri init
```

Aug 18 15:24:06.524 INFO wrote file, path: ./shell.nix
Aug 18 15:24:06.524 INFO wrote file, path: ./.envrc
Aug 18 15:24:06.524 INFO done

At this point we should now have:

```
tree -a myFirstNix
```

myFirstNix
├── .envrc
├── nix
│ ├── sources.json
│ └── sources.nix
└── shell.nix

1 directory, 4 files

We might want to take a quick look at what is being loaded into the environment and the `shell.nix` at this point.

```
let
  pkgs = import <nixpkgs> {};
in
pkgs.mkShell {
  buildInputs = [
    pkgs.hello
  ];
}
```

Code Snippet 1: `shell.nix`

```
eval "$(lorri direnv)"
```

Code Snippet 2: `.envrc`

The `.envrc` output is not very useful at a glance, however when we cd into the directory it is very verbose and explicit about what is being set up.

```
> /home/haozeke/Git/Github/NixOS/quip-nix/
direnv: loading ~/Git/Github/NixOS/quip-nix/.envrc
direnv: export +AR +AR_FOR_TARGET +AS +AS_FOR_TARGET +CC +CC_FOR_TA
RGET +CONFIG_SHELL +CXX +CXX_FOR_TARGET +HOST_PATH +IN_NIX_SHELL +L
D_FOR_TARGET +NIX_BINTOOLS +NIX_BINTOOLS_FOR_TARGET +NIX_BINTOOLS_W
RAPPER_TARGET_HOST_x86_64_unknown_linux_gnu +NIX_BINTOOLS_WRAPPER_T
ARGET_TARGET_x86_64_unknown_linux_gnu +NIX_BUILD_CORES +NIX_BUILD_T
OP +NIX_CC +NIX_CC_FOR_TARGET +NIX_CC_WRAPPER_TARGET_HOST_x86_64_un
known_linux_gnu +NIX_CC_WRAPPER_TARGET_TARGET_x86_64_unknown_linux_
gnu +NIX_CFLAGS_COMPILE +NIX_CFLAGS_COMPILE_FOR_TARGET +NIX_ENFORCE
_NO_NATIVE +NIX_HARDENING_ENABLE +NIX_INDENT_MAKE +NIX_LDFLAGS +NIX
_LDFLAGS_FOR_TARGET +NIX_LOG_FD +NIX_STORE +NM +NM_FOR_TARGET +OBJC
OPY +OBJCOPY_FOR_TARGET +OBJDUMP +OBJDUMP_FOR_TARGET +PERL5LIB +PIP
_PREFIX +PYTHONPATH +QUIP_ARCH +QUIP_INSTALLDIR +QUIP_STRUCTS_DIR +
RANLIB +RANLIB_FOR_TARGET +READELF +READELF_FOR_TARGET +SIZE +SIZE_
FOR_TARGET +STRINGS +STRINGS_FOR_TARGET +STRIP +STRIP_FOR_TARGET +a
llowSubstitutes +buildInputs +builder +configureFlags +depsBuildBui
ld +depsBuildBuildPropagated +depsBuildTarget +depsBuildTargetPropa
gated +depsHostHost +depsHostHostPropagated +depsTargetTarget +deps
TargetTargetPropagated +doCheck +doInstallCheck +extraClosure +name
 +nativeBuildInputs +nobuildPhase +origArgs +origBuilder +origExtra
Closure +origOutputs +origPATH +origSystem +out +outputs +patches +
phases +preHook +preferLocalBuild +propagatedBuildInputs +propagate
dNativeBuildInputs +shell +shellHook +stdenv +strictDeps +system ~L
D ~PATH
```

Figure 3: Sample output of the evaluation

Note that in order to set `lorri` up, we will need to set up a daemon.

```
systemctl --user start lorri
direnv allow
```

Pinning with niv

Note that in-spite of having set up `niv`, we have not yet used the sources defined therein. We will now fix this, by modifying `shell.nix`.

```
let
  sources = import ./nix/sources.nix;
  pkgs = import sources.nixpkgs { };
  inherit (pkgs.lib) optional optionals;
in
pkgs.mkShell {
  buildInputs = [
    pkgs.hello
  ];
}
```

Code Snippet 3: `shell.nix` with `niv`

Purity and Environments

There are a couple of things to note about this setup.

- The default shell is `bash`
- On occasion, depending on your Dotfiles you might have paths overriden in an annoying way

One workaround is to use nix shell with an argument:

```
nix-shell --run "bash"
```

- We can also pass `--pure` to the function, but at the cost of having to define many more dependencies for our shell

mkShell

The `mkShell` function is the focus of our tutorial, and we will mostly work around passing in different environments and hooks. Let us start by defining a hook.

Shell Hooks

Often, we will want to set an environment variable in our shell in advance. We should not use `direnv` for this, and instead we will focus on the `shellHook` option. Syntactically, this is of the form:

```
let
  hook = ''
  export myvar="Test"
  ''
in pkgs.mkShell {
  shellHook = hook;
}
```

Often we will describe variables in the let section in favor of cluttering the actual function call itself.

Overriding Global Packages

For overriding global packages, it is best to leverage the `config.nix` (which is commonly in `$HOME/.config/nixpkgs/config.nix`) file instead of the current environment, though it could be managed in a per-project setup as well. Consider the case where we need to disable tests for a particular packages, say `libuv`.

```
{
  packageOverrides = pkgs:
    with pkgs; {
      libuv = libuv.overrideAttrs (oldAttrs: {
        doCheck = false;
        doInstallCheck = false;
      });
    };
}
```

Code Snippet 4: A sample `config.nix` file

Python Dependencies

As R dependency management has been covered in an earlier post, we will focus on the management of `python` environments.

Generic Environments

We can define existing packages as follows (and can check for existence with `nox`) using the `let..in` syntax.

```
let
  # Niv
  sources = import ./nix/sources.nix;
  pkgs = import sources.nixpkgs { };
  inherit (pkgs.lib) optional optionals;
  # Python
  pythonEnv = pkgs.python38.withPackages (ps: with ps;[
    numpy
    toolz
  ]);
in pkgs.mkShell {
  buildInputs = with pkgs; [
    pythonEnv

    black
    mypy

    libffi
    openssl
  ];
}
```

Code Snippet 5: Shell with basic `python` environment

Project Local Pip

We can leverage a trick from here to set a local directory for `pip` installations, which boils down to some path hacking.

```
let
  # Niv
  sources = import ./nix/sources.nix;
  pkgs = import sources.nixpkgs { };
  inherit (pkgs.lib) optional optionals;
  # Python
  pythonEnv = pkgs.python38.withPackages (ps: with ps;[
    numpy
    toolz
  ]);
  hook = ''
     export PIP_PREFIX="$(pwd)/_build/pip_packages"
     export PYTHONPATH="$(pwd)/_build/pip_packages/lib/python3.8/site-packages:$PYTHONPATH"
     export PATH="$PIP_PREFIX/bin:$PATH"
     unset SOURCE_DATE_EPOCH
  '';
in pkgs.mkShell {
  buildInputs = with pkgs; [
    pythonEnv

    black
    mypy

    libffi
    openssl
  ];
  shellHook = hook;
}
```

Note that this is discouraged as we will lose the caching capabilities of nix.

Non-Standard Python

For more control over the environment, we can define it in more detail with some overlays.

```
let
  python = pkgs.python38.override {
    packageOverrides = self: super: {
      pytest = super.pytest.overridePythonAttrs (old: rec {
        doCheck = false;
        doInstallCheck = false;
      });
    };
  };
  myPy = python.withPackages
    (p: with p; [ numpy pip pytest ]);
in pkgs.mkShell {
  buildInputs = with pkgs; [
    myPy
  ];
}
```

We have used both overriden packages and standard packages in the above formulation.

Building Packages

For cases where we are certain that no existing package is present (use `nox`) we can also build them. Take `f90wrap` as an example, and we will use the Github version, rather than the PyPi version (the difference is in the source fetch function).

```
f90wrap = self.buildPythonPackage rec {
  pname = "f90wrap";
  version = "0.2.3";
  src = pkgs.fetchFromGitHub {
    owner = "jameskermode";
    repo = "f90wrap";
    rev = "master";
    sha256 = "0d06nal4xzg8vv6sjdbmg2n88a8h8df5ajam72445mhzk08yin23";
  };
  buildInputs = with pkgs; [ gfortran stdenv ];
  propagatedBuildInputs = with self; [
    setuptools
    setuptools-git
    wheel
    numpy
  ];
  preConfigure = ''
    export F90=${pkgs.gfortran}/bin/gfortran
  '';
  doCheck = false;
  doIstallCheck = false;
};
```

This is quite involved, discuss.

Setting Versions

We can finally generalize our `shell.nix` to default to `python 3.8` but also take a command through `--argstr`:

```
nix-shell --argstr pythonVersion 36 --run "bash"
```

Where we need to simply define the option at the top of the file, with a default.

```
{ pythonVersion ? "38" }:
# Define
let
  sources = import ./nix/sources.nix;
  pkgs = import sources.nixpkgs { };
  inherit (pkgs.lib) optional optionals;
  hook = ''
    # Python Stuff
     export PIP_PREFIX="$(pwd)/_build/pip_packages"
     export PYTHONPATH="$(pwd)/_build/pip_packages/lib/python3.8/site-packages:$PYTHONPATH"
     export PATH="$PIP_PREFIX/bin:$PATH"
     unset SOURCE_DATE_EPOCH
  '';
  # Apparently pip needs 1980 or above
  # https://github.com/ento/elm-doc/blob/master/shell.nix
  python = pkgs."python${pythonVersion}".override {
    packageOverrides = self: super: {
      pytest = super.pytest.overridePythonAttrs (old: rec {
        doCheck = false;
        doInstallCheck = false;
      });
      ase = super.ase.overridePythonAttrs (old: rec {
        doCheck = false;
        doInstallCheck = false;
      });
      f90wrap = self.buildPythonPackage rec {
        pname = "f90wrap";
        version = "0.2.3";
        src = pkgs.fetchFromGitHub {
          owner = "jameskermode";
          repo = "f90wrap";
          rev = "master";
          sha256 = "0d06nal4xzg8vv6sjdbmg2n88a8h8df5ajam72445mhzk08yin23";
        };
        buildInputs = with pkgs; [ gfortran stdenv ];
        propagatedBuildInputs = with self; [
          setuptools
          setuptools-git
          wheel
          numpy
        ];
        preConfigure = ''
          export F90=${pkgs.gfortran}/bin/gfortran
        '';
        doCheck = false;
        doInstallCheck = false;
      };
    };
  };
  myPy = python.withPackages
    (p: with p; [ ase ipython ipykernel scipy numpy f90wrap pip ]);
in pkgs.mkShell {
  buildInputs = with pkgs; [
    # Required for the shell
    zsh
    perl
    git
    direnv
    fzf
    ag
    fd

    # Building thigns
    gcc9
    gfortran
```

```
    openblas

    myPy
    # https://github.com/sveitser/i-am-emotion/blob/294971493a8822940a153ba1bf211bad3ae396e6/gpt2/shell.nix
  ];
  shellHook = hook;
}
```

Code Snippet 6: Full `shell.nix`
This is enough to cover almost all use-cases for `python` environments.

Build Helpers

Note that we can speed up some aspects of fetch with the prefetch commands:

```
nix-prefetch-git $giturl
nix-prefetch-url $url
```

In practice, some trial and error is easier.

Supplementary Reading Material

Though these are in no means exhaustive, they may offer a slightly more advanced or different focus than the material covered here.

Core Content

- Manuals
  - Nix and Nixpkgs
- Nix Wiki
  - Nix Cheatsheet
- Language Sections

Learning Paths

- Nix pills
- Official tutorials
- Nix dev has some nice opinionated tips

Personal Correspondence

Tyson Whitehead from Compute Canada was kind enough to bring the folllowing additional training materials:

- A wiki pertaining to usage of Nix on in an HPC setting
- SWC style workshop materials from TECC 2018
- SHARCNET live presentation materials from 2018

Conclusions

The standard dive into Nix is based on building derivations and playing with language, which is in no means a bad one, just too long for the time allocated. The best way to get into Nix is to start using it for everything.

References

Dolstra, Eelco, Merijn de Jonge, and Eelco Visser. 2004. "Nix: A Safe and Policy-Free System for Software Deployment," 15.
Dolstra, Eelco, Andres Löh, and Nicolas Pierron. 2010. "NixOS: A Purely Functional Linux Distribution." *Journal of Functional Programming* 20 (5-6): 577--615. <https://doi.org/10/dfrgtj>.
Goswami, Rohit, Amrita Goswami, and Jayant K. Singh. 2020. "D-SEAMS: Deferred Structural Elucidation Analysis for Molecular Simulations." *Journal of Chemical Information and Modeling* 60 (4): 2169--77. <https://doi.org/10.1021/acs.jcim.0c00031>.

```
assume the multi-user installation
```