A fully reproducible Quarto book template

Powered by Github Actions and Nix

Bruno Rodrigues

19/10/2023

Table of contents

Welcome!

This book template gets built on Github Actions at each push

This Quarto book template gets automatically built on Github Actions each time you push changes. To ensure reproducibility, the Nix package manager gets used to install all the dependencies you need:

- An R version;
- A library of R packages;
- Quarto;
- TeXLive packages;
- any other system-level dependency that is required.

Because a specific `nixpkgs` revision gets used, *exactly* the same pieces of software get *always* installed. So you don't need to pin a specific version of R, nor use `{renv}`, nor make sure to use a fixed version of a runner (typically ubuntu-22.04) to ensure reproducibility of your book. The next chapter explain how Nix works in more detail and why it's enough to use it to ensure reproducibility.

Each time a commit gets pushed, a website gets built, an Epub for E-ink readers (such as the Kindle or the Kobo) and a PDF

get built. The PDF is also in the right format and ready for self-publishing on Amazon Kindle Direct Publishing.

1 Nix

This book template uses the Nix packager manager to handle the book's dependencies. This chapter introduces the Nix packager manager and `nixpkgs` quickly.

1.1 The Nix package manager

Nix is a package manager available for Linux, Windows (on WSL2) and macOS. Its mono-repository, `nixpkgs`, contains more than 80'000 packages, among them the entirety of the R packages released on CRAN and Bioconductor, as well as R itself (and RStudio, but only for Linux and Windows, as of writing). This means that it is possible to use the Nix package manager to install R, the R packages you require for your day-to-day work, and any other packages that you might need (for example, if you need Python and Python packages, you can install these as well).

The question that remains unanswered though, is why use the Nix package manager to install all this software instead of using the usual ways of first installing R, and then using `install.packages()` to install any required packages?

There are at least three reasons. The first is that it is possible to define so-called `default.nix` files that define an environment. This environment will contain all the packages that

you require, and will not interfere with any other packages installed on your system. This essentially means that you can have project-specific `default.nix` files, each specifying the requirements for specific projects. The second reason is that when installing a package that requires system-level dependencies, {rJava} for example, all the lower-level dependencies get automatically installed. Forget about reading error messages of `install.packages()` to find which system development library you need to install first. The third reason, is that you can pin a specific revision of `nixpkgs` to ensure reproducibility.

1.2 Ensuring reproducibility with Nix

The `nixpkgs` mono-repository is "just" a Github repsitory which you can find here: https://github.com/NixOS/nixpkgs. This repository contains Nix expressions to build and install more than 80'000 packages and you can search for installable Nix packages here.

Because `nixpkgs` is a Github repository, it is possible to use a specific commit hash to install the packages as they were at a specific point in time. For example, if you use this commit, 7c9cc5a6e, you'll get the very latest packages as of the 19th of October 2023, but if you used this one instead: 976fa3369, you'll get packages from the 19th of August 2023.

You can declare which revision of `nixpkgs` to use at the top of a `default.nix` file. Here is what such a file looks like:

```
let
 pkgs = import (fetchTarball
 "https://github.com/NixOS/nixpkgs/archive/976fa3369d722e76f37c
 {};
```

```
 rpkgs = builtins.attrValues {
  inherit (pkgs.rPackages) tidymodels vetiver
  targets xgboost;
};
 system_packages = builtins.attrValues {
  inherit (pkgs) R;
};
in
 pkgs.mkShell {
  buildInputs = [  rpkgs system_packages  ];
 }
```

As you can see, we import a specific revision of the `nixpkgs` Github repository to ensure that we always get the same packages in our environment.

If you're unfamiliar with Nix, this file can be quite scary. But don't worry, with my co-author Philipp Baumann we developed an R package called `{rix}` which generate this `default.nix` files for you.

1.3 The R {rix} package

`{rix}` is an R package that makes it very easy to generate very complex `default.nix` files. These files can in turn be used by the Nix package manager to build project-specific environments. The book's Github repository contains a file called `define_env.R` with the following content:

```r
library(rix)

rix(r_ver = "4.3.1",
```

```
    r_pkgs = c("quarto"),
    system_pkgs = "quarto",
    tex_pkgs = c(
      "amsmath",
      "fvextra",
      "environ",
      "fontawesome5",
      "orcidlink",
      "pdfcol",
      "tcolorbox",
      "tikzfill"
    ),
    ide = "other",
    shell_hook = "",
    project_path = ".",
    overwrite = TRUE,
    print = TRUE)
```

`{rix}` ships the `rix()` function which takes several arguments. These arguments allow you to specify an R version, a list of R packages, a list of system packages, TeXLive packages and other options that allow you to specify your requirements. Running this code generates this `default.nix` file:

```
# This file was generated by the {rix} R package
v0.4.1 on 2023-10-19
# with following call:
# >rix(r_ver =
"976fa3369d722e76f37c77493d99829540d43845",
#  > r_pkgs = c("quarto"),
#  > system_pkgs = "quarto",
#  > tex_pkgs = c("amsmath",
#  > "fvextra",
```

```
#  > "environ",
#  > "fontawesome5",
#  > "orcidlink",
#  > "pdfcol",
#  > "tcolorbox",
#  > "tikzfill"),
#  > ide = "other",
#  > project_path = ".",
#  > overwrite = TRUE,
#  > print = TRUE,
#  > shell_hook = "")
# It uses nixpkgs' revision
976fa3369d722e76f37c77493d99829540d43845 for
reproducibility purposes
# which will install R version 4.3.1
# Report any issues to
https://github.com/b-rodrigues/rix
let
 pkgs = import (fetchTarball
 "https://github.com/NixOS/nixpkgs/archive/976fa3369d722e76f3
 {};
 rpkgs = builtins.attrValues {
  inherit (pkgs.rPackages) quarto;
};
  tex = (pkgs.texlive.combine {
  inherit (pkgs.texlive) scheme-small amsmath
  fvextra environ fontawesome5 orcidlink pdfcol
  tcolorbox tikzfill;
});
 system_packages = builtins.attrValues {
  inherit (pkgs) R glibcLocalesUtf8 quarto;
};
  in
```

```
pkgs.mkShell {
  LOCALE_ARCHIVE = if pkgs.system ==
  "x86_64-linux" then
  "${pkgs.glibcLocalesUtf8}/lib/locale/locale-archive"
  else "";
  LANG = "en_US.UTF-8";
  LC_ALL = "en_US.UTF-8";
  LC_TIME = "en_US.UTF-8";
  LC_MONETARY = "en_US.UTF-8";
  LC_PAPER = "en_US.UTF-8";
  LC_MEASUREMENT = "en_US.UTF-8";

  buildInputs = [  rpkgs tex system_packages  ];

}
```

You can now use this file to work on your book locally by first building the environment and then use it. To know more about using `default.nix` files on a day-to-day basis, read this vignette.

In the next chapter, I'm going to explain how this book gets built on Github Actions.

2 Building on Github Actions with Nix

2.1 Setup

Just like when building using the usual approches, you first need to build the book locally, on your computer, once. For this, after having generated the `default.nix` file, you can build the environment using `nix-build`, and then drop in a shell with `nix-shell` (if this previous sentence is confusing, make sure you read the vignette linked at the end of the previous chapter).

Once in that shell, run `quarto publish gh-pages`. This will render the book, and make sure that everything gets setup properly. If the book does not render, this could mean that you're missing some dependency. Make sure to specify all the requirements in the `define_env.R` script and that you re-generated the `default.nix` file. If the `quarto publish gh-pages` command succeeds, you're all set. Editing the book and pushing will build the book on Github Actions.

2.2 The Github Actions workflow file

Here is what the workflow file looks like:

```
name: Build book using Nix

on:
  push:
    branches:
      - main
      - master

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
    - name: Checkout Code
      uses: actions/checkout@v3

    - name: Install Nix
      uses:
      DeterminateSystems/nix-installer-action@main
      with:
        logger: pretty
        log-directives: nix_installer=trace
        backtrace: full

    - name: Nix cache
      uses:
      DeterminateSystems/magic-nix-cache-action@main

    - name: Build development environment
      run: |
        nix-build

    - name: Publish to GitHub Pages (and render)
```

```
uses:
b-rodrigues/quarto-nix-actions/publish@main
env:
  GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
```

The first step *Checkout code* makes the code availalbe to the rest of the steps. I then install Nix on this runner using the Determinate Systems `nix-installer-action` then I use another action from Determinate Systems, the `magic-nix-cache-action`. This action caches all the packages so that they don't need to get re-downloaded each time a change gets pushed, speeding up the process by a lot. The development environment gets then built using `nix-build`.

Finally, an action I defined runs, `quarto-nix-actions/publish`. This is a fork of the `quarto-actions/publish` action which you can find here. My fork simply makes sure that the `quarto render` and `quarto publish` commands run in the Nix environment defined for the project.

3 Conclusion

In conclusion, Nix rocks.