Quarto project scripts are awesomeness

Chris von Csefalvay

Blog post published October 22, 2023 in *Chris von Csefalvay*
https://doi.org/10.59350/cyv5c-8nd34

Abstract

Tip If you're here for the DOI matching script, the link is here. Quarto is a great tool for reproducible research. It is also a great tool for building websites. After about a decade of Wordpress-based websites, I've moved to Quarto primarily for two reasons. First, most of my content is largely static.

Keywords

Epidemiology (Subfield), Quarto, Python

Recommended citation

von Csefalvay, C. (2023, October 22). Quarto project scripts are awesomeness. *Chris Von Csefalvay*. https://doi.org/10.59350/cyv5c-8nd34

Copyright

Copyright © Chris von Csefalvay 2023. Distributed under the terms of the Creative Commons Attribution 4.0 International License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

Tip

If you're here for the DOI matching script, the link is here.

Quarto is a great tool for reproducible research. It is also a great tool for building websites. After about a decade of Wordpress-based websites, I've moved to Quarto primarily for two reasons. First, most of my content is largely static. I don't quite need the full power of a CMS, and the performance cost of dynamic generation I didn't need was quite steep (the economic cost was nothing special, but I'm a cheapskate). Second, I wanted to be able to write in Markdown and integrate better with notebooks. Quarto lets me do both of these.

What Quarto also does, however, is something super awesome: project scripts!

Project whats?

A project script in Quarto is, basically, a piece of Python, Lua, R or Typescript code that is executed at some point during rendering. This is super useful, because it allows you to do things like, say, automatically generate a list of publications from a BibTeX file – which is among others what I'm using it for. Essentially any programmatic function can be implemented in a project script.

```
flowchart TD
    A[Rendering pipeline start] --> B(Pre-render scripts)
    B --> C(Page rendering)
    C --> D(Post-render scripts)
    D --> E[Rendering pipeline end]
```

Figure 1: Rendering pipeline in Quarto with pre- and post-render scripts.

Quarto distinguishes between pre-render and post-render scripts. No prizes for guessing the difference, but here it goes: pre-render scripts are executed before the page is rendered, and post-render scripts are executed after the page is rendered. Pre-render scripts are particularly useful as they can modify the source `.qmd` (Quarto markdown) files before rendering them to HTML or whatever their ultimate evolutionary destiny is.

To make project scripts work, you need to declare them in your `_quarto.yml` file:

```yaml
project:
  type: website
  pre-render:
    - scripts/pre_create_papers_file.py
    - scripts/pre_check_skierg_records.py
```

This declares the scripts `scripts/pre_create_papers_file.py` and `scripts/pre_check_skierg_records.py` as pre-render scripts. You can also declare post-render scripts, with the `post-render` key.

A (poorly) worked example: generating a publication list

On my website, I use Quarto project scripts to maintain a list of my papers. This list is generated from a BibTeX file, which makes it a little more convenient to maintain my publications, and somewhat future-proofed in case I want a different format.[1]

[1] I know there's a canonically better solution that uses pandoc and Lua. I don't care. This works for me, and I don't have to learn Lua.

You can see the full project script here, but here's the gist of it:

```
flowchart TD
    A[BibTeX file] --> a("load_bibtex_file()")
    a --> B[BibTeX database object]
    B --> b("render_as_nlm()")
    b --> C{book or article?}
    C --> D[article]
    C --> E[book]
    D --> d("render_article_as_nlm()")
    d --> F[article entry]
    E --> e("render_book_as_nlm()")
    e --> G[book entry]
    F --> H[all entries]
    G --> H
    H --> h("generate_list_by_year()")
    h --> J[final set of entries]
    J --> j("write_into_file()")
    j --> K["papers/index.qmd"]
```

Figure 2: My super clumsy pre-render script to build my bibliography.

We load a BibTeX file (`load_bibtex_file()`), parse it, use a function to determine whether we're rendering a book or an article entry (I don' have any other types of publications, but if I did, I'd add them here), and dispatch rendering to the appropriate function (`render_article_as_nlm()` for articles, `render_book_as_nlm()` for books). We sort the entries into a Markdown format with years separated by level 1 headings (`generate_list_by_year()`) and finally write it into the `papers/index.qmd` file (`write_into_file()`). There's some minor magic going on behind the scenes, such as capturing my name and setting it in bold (which is a bit of a convention in academic lists of publications), but that's the gist of it.

The result is a list of publications that is automatically generated from a BibTeX file, and is always up to date (as up to date as the BibTeX file, anyway). I don't have to manually update it... sort of.

Wait, automagically updated?

The second half of what makes project scripts awesome is that they integrate with the rendering process. If the rendering process in turn integrates with some CI that watches for changes, then you can have a website that is automatically updated whenever you push a change to the repository. This is what I do with my website: I have a GitHub action that watches for changes to the `main` branch, and if it detects a change, it runs the rendering process and pushes the result to the `gh-pages` branch. This means that if I were to change the BibTeX file, it would trigger a re-render, and the list of publications would be automatically updated.

There are some tricks to keep in mind here. Most importantly, because this requires some custom Python packages, I had to slightly amend my Github publishing workflow:

```
flowchart TD
    A["actions/checkout@v4"] --> B["quarto-dev/quarto-actions/setup@v2"]
    B --> C["actions/setup-python@v4"]
    C --> D("pip install -r requirements.txt")
    D --> E["quarto-dev/quarto-actions/publish@v2"]
    style D stroke-width:8px;
    style C stroke-width:8px;
```

Figure 3: Integrating the pre-render script into the Github Actions CI/CD framework. The parts that need to be integrated into the publishing workflow are highlighted with bold outlines.

In particular, I had to explicitly specify the Python version and install the requirements. This is because the `quarto-actions/setup` action does not install the requirements, and the `quarto-actions/publish` action does not install Python. This is admittedly a bit of a nuisance to work around, but really not all that big a deal in the grand scheme of things.

One thing to keep in mind when writing your script is the context from which it will be executed. In particular, if you're using a CI/CD pipeline, you will need to make sure that the script can find the files it needs. In my case, I had to explicitly specify the path to the BibTeX file, as well as the path to the output file. This is because the script is executed from the root of the repository, but the BibTeX file is in the `papers` directory, and the output file is in the `papers` directory, too. Testing locally for deployment in CI/CD is notoriously hellish, but with a little bit of elbow grease,[2] you can get it to work.

[2] And by 'a little bit', I mean absolute tons of it.

One caveat: rendering with pre- and post-render project scripts may be relatively expensive in terms of time. My silly SkiErg world record script adds a good 20-25 seconds of processing time to each rendering run – not a lot, but it does stack up eventually. The documentation helpfully discloses that Quarto provides an environment variable, `QUARTO_PROJECT_RENDER_ALL`, which is set to `1` if it's a full render. It may make sense to do a more manual caching here.

Things you can do with project scripts

Seriously, the possibilities of what you can do with project scripts are pretty much endless. I have a project script that checks the world records list on the official SkiErg website and makes sure my index page lists my accurate records. This is completely silly, but it's also a great example of how you can use project scripts to automate just about everything:

- A script I built pulls DOIs minted by Rogue Scholar and appends them to posts.
- A lot of projects use it for housekeeping: via makefiles
- The bambi project uses it to build a changelog, which is a great idea for applications that do need it
- This one cool master's thesis uses pre-render project scripts to load variables from a YAML file into envvars, and I've seen the same done with Rprofile files – obviously, these are much more pertinent to projects where the bulk of the project consists of notebooks/code execution rather than websites, but again, the idea is the same
- Obviously, I'm not the only one to use it for reference management
- You can use it to download static content into your project, too!

The sky's the limit! I'm currently working on rendering my CV from a JSON file via a project script, as well as a few other cool things. I keep my project scripts here – I hope they will inspire you to do something cool with your own project scripts!