Developer Guide

Table of contents

OneRoof Development Environment

Most `oneroof` development happens in the `dev` branch of this repo. To get started with doing so yourself, clone and switch to the `dev` branch like so:

```
git clone https://github.com/nrminor/oneroof.git && \
cd oneroof && \
git checkout dev
```

From there, you'll have a few tools to consider installing if you haven't already:

1. The Pixi package manager, available here: https://pixi.sh/latest/
2. The command runner Just, available here: https://just.systems/man/en/chapter_4.html
3. Pre-commit, available here: https://pre-commit.com/
4. Docker to use the repo Dockerfile as a dev container, available here: https://docs.docker.com/engine/install/
5. Quarto to modify documentation, available here: https://quarto.org/docs/get-started/
6. An editor with language support for Nextflow (or Groovy as a fallback), Python, and Quarto (or Markdown as a fallback).

Most of this project was written in VSCode, as it's currently the only editor with plugins for Nextflow, Quarto, and Python linters.

Pixi

Of these tools, Pixi is the only one that is essential for most use cases, as it handles installing the pipeline's dependencies from various `conda` registries as well as the Python Package Index. That said, you could even get away with not using `pixi` by manually installing the packages in `pyproject.toml` in a `conda` environment, using pip when needed. While this solution is inelegant compared to having a unified package manager like Pixi, it may be more familiar to some users.

If you are sticking with Pixi, run `pixi shell --frozen` in the project root before you get started. This will create a terminal environment with everything you need to work on and run `oneroof` (or rather: everything except Dorado. More on that below). The `--frozen` flag is critical; it tells Pixi to install the exact dependency versions recorded in `pixi.lock` as opposed to searching for newer versions where possible. This ensures reproducibility, as updates to the dependencies in `pixi.lock` will only be pushed when they have been tested on Linux and MacOS systems by the core maintainer team.

Just

While Just isn't as important as Pixi, I would still recommend installing it because of the conveniences it offers. With Just, the repo `justfile` provides a switchboard of command shorthands, including:

- `just docs`, which runs a series of Quarto commands to render and bundle the repo docs (including this file) as well as construct an updated readme
- `just py`, which lints and formats all the repo's Python files.
- `just docker`, which builds and pushes a new version of the repo Docker Image.
- `just env`, which instantiates the Pixi environment.
- `just all`, which does everything (`just doit` will do the same thing).

Run `just` in the same directory as the repo `justfile` to list all available recipes, and check out the Just Programmer's Manual for more about Just.

Pre-commit

If you run `pre-commit install` in the repo root directory, it will install the pre-commit hooks in our `.pre-commit-config.yaml`. These hooks make sure that formatting throughout the repo files are consistent before they can be committed. Again, not essential, but nice!

Docker

If you don't want to futz with all of the above or with setting up Dorado locally, you can also use the Docker Hub image `nrminor/dorado-and-friends:v0.2.2` as a dev container. It should every everything the pipeline and its dev environment needs, though its use as a dev container has not yet been tested.

Quarto

Quarto can be thought of as a renderer or compiler for documents written in supercharged Markdown. It can run code blocks, render to HTML, PDF, `reveal.js` presentations, websites, books, and dozens of other formats. As such, I use Quarto documents as the sort of "ur-format," and render it out to other formats as desired. To render the project readme or other documents, you will need Quarto installed. The Quarto config file, `_quarto.yml`, controls project level settings, including a post-render section telling it to regenerate the Github-markdown-formatted readme from `assets/index.qmd` (via the project's `just readme` recipe).

Dorado

Dorado is the one tricky dependency that isn't handled by Pixi, as it is not published outside of the compiled executables in Oxford Nanopore's GitHub Releases. To install it locally on your own system, you'll need to take additional care managing `$PATH` and downloading Dorado's models yourself, as the Dockerfile does. Here's how the Docker file handles the Linux version of Dorado it installs

```
wget --quiet https://cdn.oxfordnanoportal.com/software/analysis/dorado-0.7.1-
linux-x64.tar.gz && \
tar -xvf dorado-0.7.1-linux-x64.tar.gz && \
rm -rf dorado-0.7.1-linux-x64.tar.gz

export  PATH=$PATH:$HOME/dorado-0.7.1-linux-x64/bin:$HOME/dorado-0.7.1-linux-
x64/lib:$HOME/dorado-0.7.1-linux-x64

dorado download
```

The process will be similar for MacOS users; we recommend consulting the above Dorado releases for the executable you should download. Feel free to raise an issue or a PR for more specific instructions in this section!

Nextflow organization

This pipeline follows a slightly shaved-down project organization to the nf-core projects, which you can think of a standardized means of organizing Nextflow `.nf` scripts. Like most Nextflow pipelines, the pipeline entrypoint is `main.nf`. `main.nf` performs some logic to call one of the workflow declaration scripts in the `workflows/` directory. Workflow declaration scripts themselves call subworkflow declaration scripts in the `subworkflows/` directory. Subworkflow scripts then call modules in the `modules/` directory; these scripts actually do work, as they contain the individual processes that make up each workflow run.

If this seems like a lot, trust your instincts. Here's why it's worth it anyway:

1. It atomizes each building block for the pipeline, making it much easier to make the pipeline flexible to the inputs provided by the user. The pipeline doesn't just run in one way; it can run in many ways. It also makes it possible to reuse the same processes at multiple stages of the pipeline, like we do with our `reporting.nf` module.
2. It makes it exceptionally easy to add functionality by plugging in new module scripts.
3. It also makes it exceptionally easy to switch out or reorder modules or bring in new subworkflows, e.g., a phylogenetics subworkflow. In other words, this project structure makes it easier for the maintainers *to refactor*.
4. Having defined individual workflow and module files here makes it easier to move around files and reuse Nextflow code for other workflows in the future. Instead of needing to scroll down a very long monolithic Nextflow script, just take the one file you need and get to work.

Like most Nextflow pipelines, `oneroof` also has a few other important directories:

- `bin/`, which contains executable scripts that are called in various processes/modules.
- `lib/`, which contains Groovy utility functions that can be called natively within Nextflow scripts.
- `conf/`, which contains Nextflow configuration files with defaults, in our case, for the two workflow options.

Nextflow configuration

`oneroof` comes with a large suite of parameters users can tweak to customize each run to their needs, in addition to profiles for controlling the environment the pipeline runs in. For now, we recommend users review the table in the repo `README.md` for documentation on the most commonly used parameters. That said, we expect to have more configuration documentation here soon.

Python development

We're big fans of the Ruff linter and formatter, and we use it liberally in the writing of our Python scripts. To do so yourself in VSCode, we offer the following configuration for your User Settings JSON:

```
{
  "[python]": {
    "editor.defaultFormatter": "charliermarsh.ruff",
    "editor.codeActionsOnSave": {
      "source.fixAll.ruff": "explicit",
      "source.organizeImports.ruff": "explicit"
    }
  },
  "ruff.lineLength": 88,
  "ruff.lint.select": ["ALL"],
  "ruff.lint.ignore": ["D", "S101"],
  "ruff.nativeServer": "on",
  "notebook.defaultFormatter": "charliermarsh.ruff",
}
```

At first, the lints may seem daunting, but virtually all lints come with persuasive documentation in the Ruff rule docs. Ultimately, our strict compliance with Ruff lints is inspired by the similar level of robustness that strict lints afford in the Rust ecosystem. We want our software to be resilient, performant, formatted in a familiar style, and reproducible in the long-term. Complying with as many lints as possible will only help, not harm, that end, even if it's a bit annoying or overwhelming in the short-term. It's also possible that some of the lints—e.g., using Python's `pathlib` library for all path handling in places like `with open(...)` context managers, or never using `os.system()` calls—will also make you a better programmer.