# Master Hugo Modules: Managing Themes As Modules

NICK

Apr 19, 2020

⏱ 8 minute read

**Master Hugo Modules**

**Modules** **Themes** **Hugo**

When I first encountered Hugo modules, I didn't really understand their value. As I worked on more and more Hugo projects, I started to feel the pain of typical git submodules. I started to read, and re-read documentation both at Hugo and Go, trying to wrap my mind around modules. I also read lots of questions, problems, and approaches in the Hugo Discourse space. Thanks to a lot of experimentation and tests, and time, they became clear to me—and very, very elegant. I decided to write out what I've learned in a series of articles, in part to thank the Hugo community, and in part to clarify my thinking.

To start, let's take a look at themes as modules, and why the typical method of handling a theme as a submodule isn't ideal.

## Working with Hugo themes as git submodules is a pain

Hugo's quick start guide demonstrates adding the excellent Ananke theme as a git submodule:

```
git submodule add https://github.com/budparr/gohugo-theme-ananke.git
themes/ananke
```

Submodules are an adequate solution for theme dependencies. Forestry demonstrates how you might update the theme to pick up bug fixes and new features:

```
git submodule update --remote
```

So what’s not to love about git submodules for dependency management?

## Git submodules: the curse of recursing

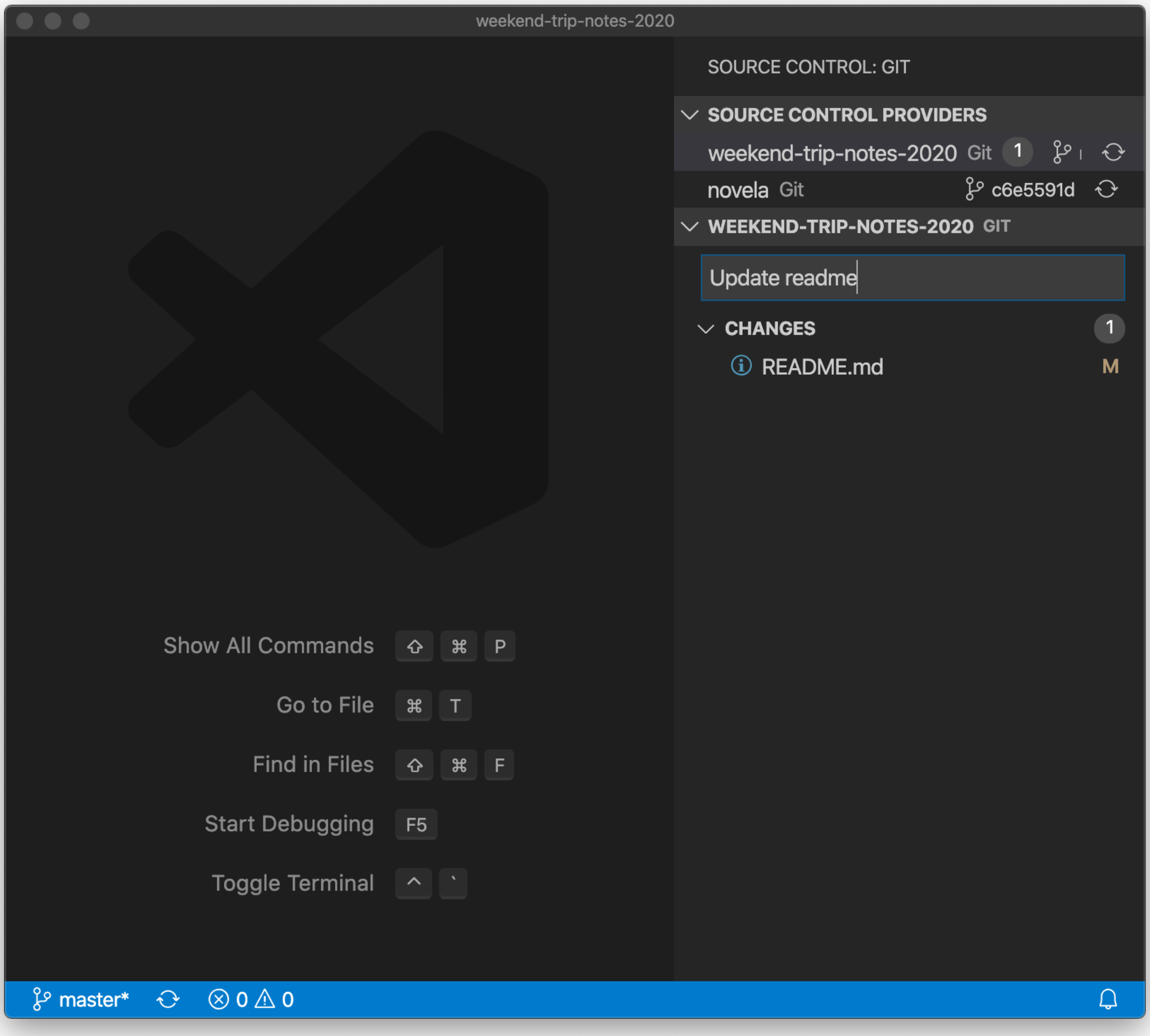
When you’re working with a team, anyone who wants to grab your project and contribute does so via the standard `git clone` — but with submodules, everyone must remember to initialize and update each submodule when cloning:

```
git clone --recurse-submodules https://gitlab.com/your-project-repo-
here
```

That’s not hard to do, but it’s **one extra step** and probably unusual in your routine `git clone` practice. I can’t count the number of times I’ve had a non-working Hugo project, only because I failed to clone correctly. You can always add this as part of the installation instructions to your repo’s readme.md, which is, of course, up to date and always referenced 🙄 , but again, it’s an extra step.

## Git submodules clutter code editors

The second problem with git submodules is that they are, correctly and by design, an entire additional git repository within your project. But when using code editors with integrated source control like Atom or Visual Studio Code, on every push, you’ll be confronted with choosing between the repositories for your commit message. This is not horrible, but again—it’s **one extra step**.


weekend-trip-notes-2020

SOURCE CONTROL: GIT

SOURCE CONTROL PROVIDERS

weekend-trip-notes-2020 Git 1

novela Git c6e5591d

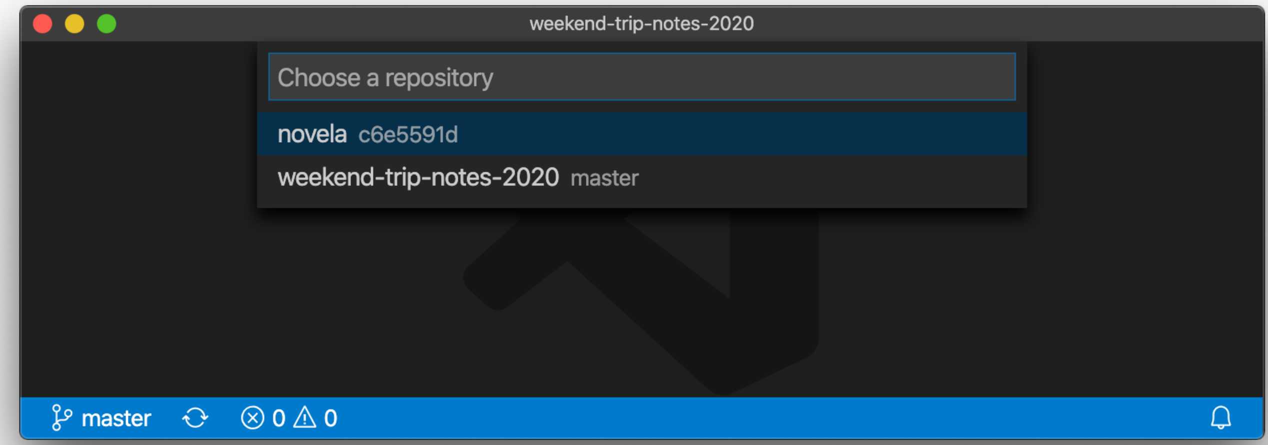
WEEKEND-TRIP-NOTES-2020 GIT

Update readme

CHANGES 1

README.md M

As I write this in Spring 2020, amid the COVID-19 crisis, more teams are working remotely than before. At Neoteric Design, we've adopted feature branch workflows to help manage development, communication, and QA — it works great. It does require agility in moving from branch to branch. Here, again, having multiple repositories means each branch checkout requires choosing between the project and a git submodule. Not horrible, but **one extra step**.


weekend-trip-notes-2020

Choose a repository

novela c6e5591d

weekend-trip-notes-2020 master

## Git submodules are hard to remove

Imagine you're early in a project, and reviewing a variety of themes for a documentation project. It's quite easy to add them as git submodules. But how do you remove submodules you no longer need? Git submodules aren't a tidy affair, in the sense that they leave traces in many areas of the parent repository. Consider this still ongoing discussion of removing submodules, started 10 years ago, on StackOverflow. Even the best solutions require a three to four line removal process.

This stands in the way of quick iterations, experimentation, and rapid testing of multiple options. In the past, I adopted a "try then fry" approach, where I'd add the submodule, test things out, then trash the entire project and re-clone. But come on: we're JAMstack developers working in Hugo. Surely there's a more elegant solution!

Hugo modules to the rescue.

## Hugo modules make dependency management easy

> Hugo Modules are the core building blocks in Hugo. A module can be your main project or a smaller module providing one or more of the 7 component types defined in Hugo: static, content, layouts, data, assets, i18n, and archetypes. (Hugo documentation)

Hugo Modules were introduced in Hugo 0.56.0. Think of them as components, external to your project, that can be made available to your main Hugo project in any of the 7 standard directories. When declaring modules, you specify where they come from and in what directory you'd like them mounted. So, for example, a map dataset might be mounted to /data/maps/ – and, once installed, used as if it were local.

Hugo modules have some cool features:

- Hugo downloads modules automatically; they can be "hot replaced" while your server is running, and explicitly vendored to your project directory for easy portability and version locking
- Hugo modules are built on Go modules, which have a sophisticated version resolution algorithm — semver versioning
- During development, you can easily make and test changes to a module locally

### What can you do with Hugo modules?

Hugo modules encapsulate content, features, and functionality in a reliable and easily shared format. So yes, you can mount themes to /themes/. But consider what else you might do, when working on multiple projects, to keep your work modular and DRY:

- You can integrate a package of open source icons
- You could manage a network of related sites that share standard legally required content
- You could create a master library of Hugo shortcodes used by your clients
- You could create a package to handle structured data and microformats
- You could integrate a design system of UI components, or a branding system

- You can abstract business logic into function-like Hugo partials

In short, anything that can be in the 7 primary directories of a Hugo project can be made into a module and shared across multiple projects. Inspired? Let's start by using Hugo themes as modules.

## How to use Hugo themes as modules

### Install requirements

1. Install Go. As developers, we'll need Go to run the commands for Hugo modules. End users won't, though, once modules are vendored, so don't be concerned about this requirement and portability.
2. Update to the latest version of Hugo. I often find myself running a few versions behind, and some commands came in later than 0.56.0.

Make sure `hugo mod help` is working before continuing — it has all the commands we'll need.

### Initialize your Hugo project

Note well, from the documentation: "A module can be your main project or a smaller module…" Here, we're going to initialize the main project as a module, then include the theme as a module, too.

```
hugo mod init your-project-repo
```

this is going to create a go.mod file in the root of your project. In the excellent Discourse discussion "Hugo modules for 'dummies'," there's a bit of ambiguity on what to pass to `hugo mod init`. If we take the Hugo documentation website as a guide, it's a good practice to use your repository name. It makes for clear and easy reading, too: `module github.com/gohugoio/hugoDocs` is more obvious to me than the name of a theme or other label.

### Add a theme as a Hugo module

Now for the fun part: let's pull in a theme. Update the: [module]

```
[[module.imports]]
    path = "path-to-theme"
    # project theme
```

Note well that this path does not want a protocol — `path = "gitlab.com/neotericdesign-tools/cosette"` but no `https://`. As a standard practice, I like to note in a comment that this is the theme. Hugo is smart and mounts this as if it were at `/themes/your-theme/`, but it's nice to be explicit.

### Get the project's module dependencies

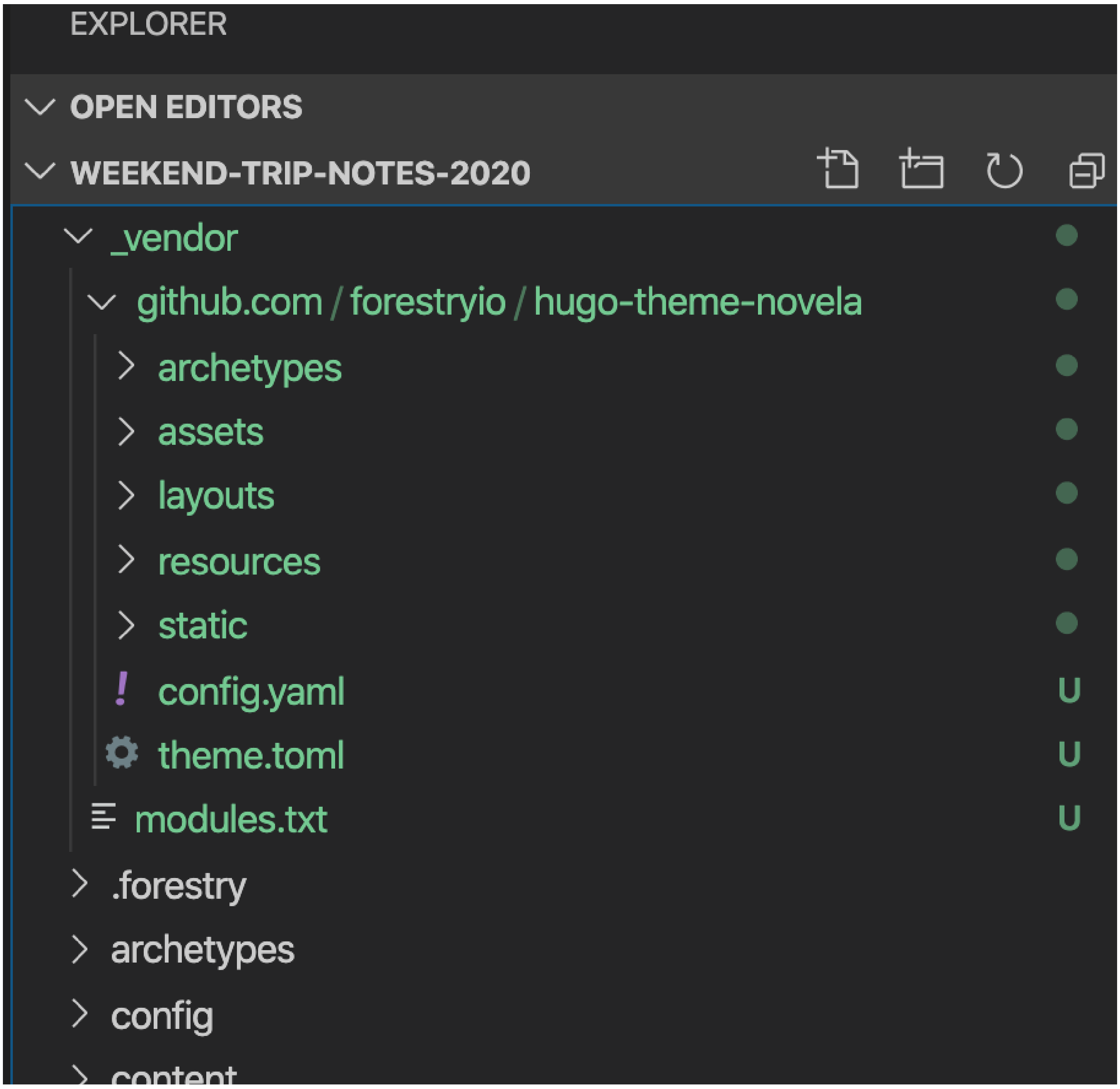
While you can just `hugo server` and watch the magic happen, it's instructive to be explicit, too. Running `hugo mod get` walks through your config and resolve external dependencies. At this point, with no theme in your /themes/ folder and no explicit theme declared in your config.toml — no need to declare `theme = "my-theme-name"` once you're on modules! — you can run `hugo server` and see the magic.

## Tips for working with Hugo modules

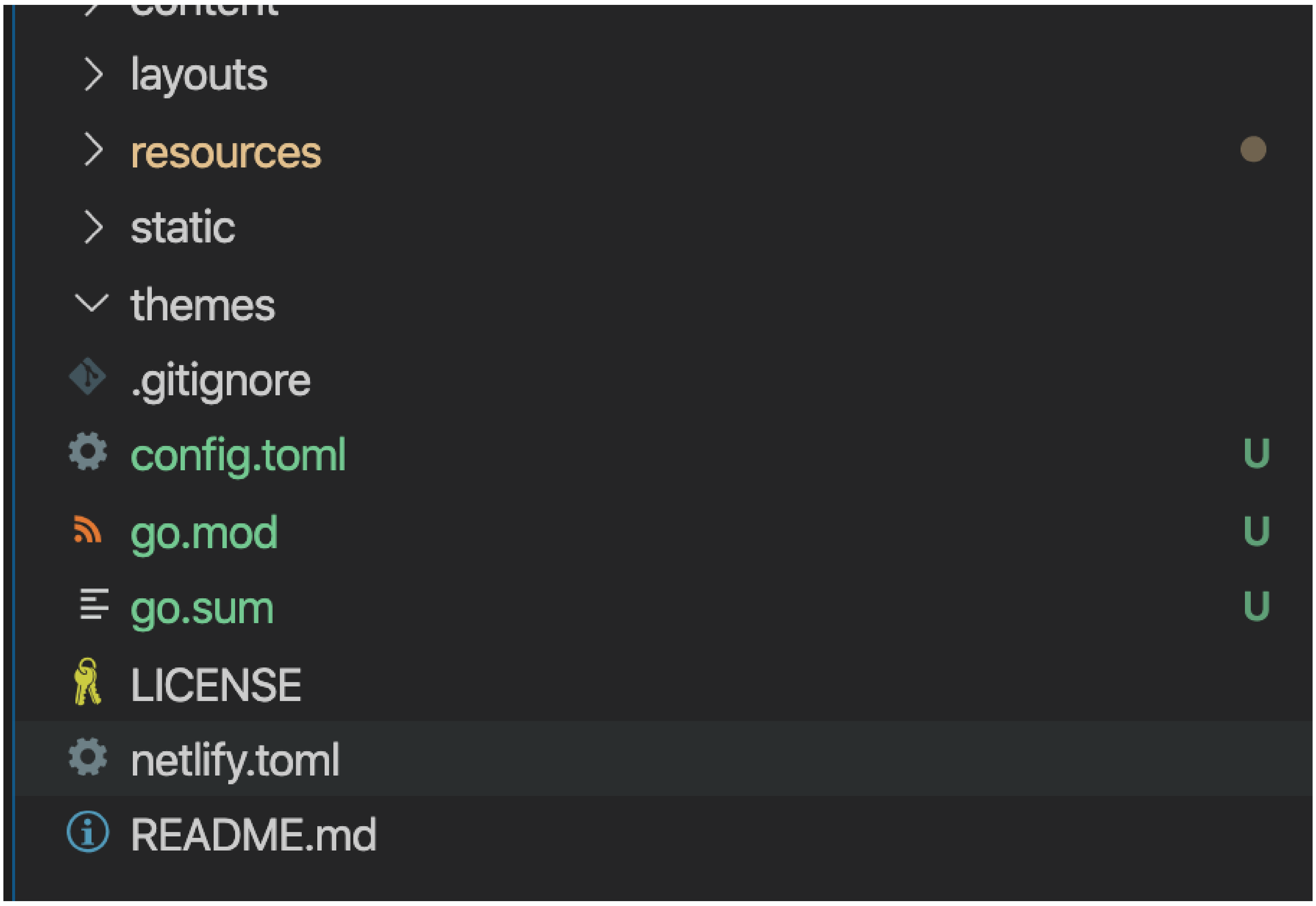
In the process of understanding how to work with Hugo modules, I came across several useful workflows.

### Confirm your expectations by vendoring your modules

Resolved modules are stored in a Hugo module cache directory. Use `hugo mod vendor` to test your expectations: it vendors all module dependencies to a `\_vendor` directory in your project root.



```
EXPLORER
OPEN EDITORS
WEEKEND-TRIP-NOTES-2020
  _vendor
    github.com / forestryio / hugo-theme-novela
      archetypes
      assets
      layouts
      resources
      static
      config.yaml                                   U
      theme.toml                                    U
    modules.txt                                     U
  .forestry
  archetypes
  config
  content
```



This is especially useful when you want to confirm where a module has been mounted, or in debugging older themes that aren't module ready. This happened to me where not every file was captured, for reasons that still aren't clear to me. Feel free to delete the `\_vendor` directory, adjust configurations, then re-vendor at any time.

## Working with go.mod and go.sum files

> A module declares its identity in its go.mod via the module directive, which provides the *module path*. [Modules · golang/go Wiki · GitHub]

Running `hugo mod init your-project-repo` generates the go.mod file, and thus initializes your project as a module. You can undo that, or redo that, at any time, by deleting the go.mod file and re-initializing the project. When you've resolved external dependencies, you'll see them automatically in the require statement in the go.mod file. The go.sum file contains [cryptographic checksums of the content of the module version]. It's not a lock file — so it retains checksums for modules even after they've been removed from your config.

A key takeaway: this is not like git submodules. You can blow go.mod and go.sum files, and you're free — and you can reinitialize as many times as you'd like during development.

## Key Hugo modules commands

- `hugo mod get` - resolves dependencies declared in your project config
- `hugo mod vendor` - put dependencies info `_vendor` and use those going forward

- `hugo mod graph` - prints a modular dependency graph
- `hugo mod tidy` - cleans up go.mod, based on dependencies declared in your project config

## Up next in the Mastering Hugo Modules series

Working with Hugo modules provides incredible benefits for us as developers. I'll continue to explore the gains in a series of articles, covering modules in private repos, theme components, speeding up module development time, modules as "functions," and more. Feel free to drop me a note on Twitter if you have ideas or questions.