How to make your Dotfile management a painless affair

2018-06-05

by Mohammed Ajmal Siddiqui

In the first article, I introduced dotfiles. In this one, we'll cover their development and management.

> Note: This is the second article in the series and discusses more advanced aspects of dotfile management. To learn about what dotfiles are and the very fundamentals of dotfile management, read the first article.

In the last article, we added a few aliases and functions to the `.bash_profile` and the `.bashrc` file. We also learned that those aren't the only dotfiles available for us to customize.

This article focuses on making our approach to dotfile management more sophisticated and scalable. At this point, there is one important thing you need to bear in mind from this point onwards. Dotfiles are a matter of personal preference, and so is their management. You should manage them your way. This article only provides general guidelines and common ways to deal with recurring tasks in dotfile management.

Setting Up Your Environment

Start with creating a directory for your dotfiles and `cd` into it. I like to have mine in the `Projects` folder in my home directory, but this is up to you:

```
$ mkdir ~/Projects/dotfiles$ cd ~/Projects/dotfiles
```

This is where we'll have all our dotfiles. Let's start by making this a git repository.

```
$ git init
```

Let's start by moving the `.bash_profile` from the `HOME` directory to our new dotfiles directory.

```
$ mv ~/.bash_profile ~/Projects/dotfiles/.bash_profile
```

Let's commit this file.

```
$ git commit -am "Added .bash_profile"
```

And there you have it! This is exactly how you'd work on any other project, and that's exactly how you should manage your dotfiles.

Why version control, one might ask. People love committing their dotfiles to version control for a couple of reasons:

- Pushing the dotfiles to a remote repo allows people to share their dotfiles with others or access them remotely when they need them. This is also a secure way to back your dotfiles up.
- Version control allows you to see how your dotfiles evolve over time.

But if you start another terminal instance, you'll notice that your setup is broken! The terminal doesn't source your `.bash_profile` or `.bashrc` from a custom folder, as these files are expected to be found in the home directory.

So we need a way to keep our dotfiles in the `dotfiles` directory in sync with our home directory. You can do this in any way you want, even if it is as simple as copying all the files in your `dotfiles` directory to the home directory with a script. But there are more elegant approaches. Let's look at two of them.

The `rsync` Approach

One way to deal with the problem of having your dotfiles in a directory other than the home directory is to copy the files using a script. But there is a far better way to do this than using the `cp` command: the `rsync` command.

Rsync, which stands for "remote sync", is a remote and local file synchronization tool. It uses an algorithm that minimizes the amount of data copied by only moving the portions of files that have changed. Thus, this approach is both more efficient and more scalable when it comes to syncing dotfiles. This article covers rsync in more detail.

Run this command to sync your dotfiles directory with your home directory:

```
$ rsync . ~
```

The command works in exactly the same way as the `cp` command. It copies the contents of the source (the current directory: `.`) to the destination (the home directory: `~`).

However, you may have utility scripts in your dotfiles directory which you may not want to copy to the home directory. In this case, you can exclude these files using the `--exclude` flag. In fact, you would rather not sync the `.git` directory within your dotfiles folder with your home directory. So here's the updated command:

```
$ rsync --exclude ".git/" . ~
```

You can use the `--exclude` flag multiple times to exclude multiple files. This approach is used by mathiasbynens in his dotfiles.

Symlinking

Another approach to syncing dotfiles is creating symlinks from the dotfiles in your dotfiles directory to the home directory. If you don't know what symlinks are, I suggest you read about it here.

This is the approach I use in my dotfiles, and I do so because of one major advantage over file copying based approaches - autoupdating. The symlinks in the directory are essentially aliases to the original files in your dotfiles directory, so any changes you make are automatically reflected therein. Which means you don't have to run your sync command/script every time you make a change. This is super useful.

You can symlink your `.bash_profile` to the home directory using the `ln` command with the `-s` flag (and the `-v` flag to make it verbose):

```
$ ln -sv ~/Projects/dotfiles/.bash_profile ~
```

Now, whenever you save changes you make to your `.bash_profile`, they'll automatically be reflected in the home directory, and you can start a new terminal or `source .bash_profile` to see them in action.

Creating Utility Scripts for Syncing and Bootstrapping

At this point, you're executing terminal commands to sync your dotfiles with your home directory. This approach is almost impossible to scale the moment you have more than 2 - 3 files to deal with.

Hence, it is better to write a couple of utility scripts that will help keep your dotfiles in check. You should have at least one script in your dotfiles repo, the one you use for syncing. Your sync script should essentially use your syncing mechanism to sync your dotfiles with the home directory. It should also have a mechanism to exclude certain files from being synced. Files like the sync script, the bootstrap script, the `.git` directory, the README.md file, etc. should be excluded.

Currently, I have a `sync` function in my `bootstrap.exclude.sh` script that handles syncing and excludes any files that have a `.exclude` in the file name, in addition to the ones stated above. This is a fairly fail-safe mechanism. You can check it out here.

It is highly recommended that you have another script to bootstrap a new system with your dotfiles and setup your development environment.

One important thing you can use your bootstrap script for is installing packages and tools that you commonly use.

For example, I am a Node.js developer and I use a Mac, so I can use the homebrew package manager to install tools and utilities that I usually use. I can include something like this in my bootstrap script:

```
# Make sure we're using the latest Homebrew
```

```
brew update
```

```
# Upgrade any already-installed formulae
```

```
brew upgrade
```

```
# NodeJS
```

```
brew install node
```

```
# Heroku
```

```
brew install heroku
```

```
# Yarn - an alternative to npm
```

```
brew install yarn
```

```
# Docker for containerization
```

```
brew install docker
```

This script installs Node, Heroku, yarn and Docker on my Mac. Say I end up formatting my Mac or buying a new one. I don't have to install all the things I use manually. Instead, I can clone my dotfiles from my remote repository and run the bootstrap script, which sets everything up for me. Since you may have a lot of things you use, it is best to separate this out into its own file. Check my `brew.exclude.sh` file out for an example.

Your bootstrap script should handle these things:

1. Making any relevant directories that you use (for example the `~/Projects` directory).
2. Call your sync script to sync your dotfiles with the home directory.
3. Install all the tools, utilities, languages, etc. that you commonly use.

My bootstrap script is pretty minimal and handles all these things, so that might be a good place to start.

Splitting Up Your .bash_profile

As we add a lot of aliases and functions, we start to realize that the `.bash_profile` becomes rather big and cumbersome to manage. Let's fix this problem.

The `source` command can be used within a script to execute the commands in the file given as an argument to the command. So we can create additional files to hold our functions and aliases and source them into our `.bash_profile`. It is a common convention to call these files `.functions` and `.aliases` respectively.

Create `.functions` and `.aliases` by using this command (while in your dotfiles directory):

```
$ touch .functions .aliases
```

Now cut and paste all the functions from your `.bash_profile` into `.functions` and all the aliases into `.aliases`. Finally, add the following lines to your `.bash_profile`:

```
source .functionssource .aliases
```

This is what's happening when you open a new terminal window:

1. The `.bash_profile` is evaluated.
2. The `source .functions` command is executed and thus you can now use your functions.
3. The `source .aliases` command is executed and thus you can now use your aliases.

You can split the contents of your `.bash_profile` in any way you please and just `source` the relevant files. Note that you'll need the `.functions` and `.aliases` files in your home directory, so make sure you sync your dotfiles folder with the home directory for your changes to take effect.

Now that we have our dotfile management workflow in place with a dotfiles repository, a mechanism to sync the contents of the repo with the home directory, and the ability to split our code into manageable files, we can happily hack away at our development environment.

While this is sufficient to play with your dotfiles, there is one more important aspect of dotfiles that should be addressed: sharing.

Adding Support For Customization

Hosting your dotfiles in a public repository is seldom enough for them to be shareable. If you would like others to experiment with your dotfiles, there are a few things that you should do:

1. Make sure there is some documentation on how to install and use your dotfiles. This usually goes in the README.md of your project. The mechanism you use to exclude files from being synced with the home directory should also exclude this file. You should also take a look at the README files of popular dotfile repos to get a sense of what people put in there. Here's a link to mine.
2. If possible, support a mechanism for people to easily customize your dotfiles without having to delve deep into them. This mechanism is completely your choice, though I'll share mine below.

> Note: The rest of this section discusses my approach to allowing painless customization of my dotfiles. I personally love my approach (and that's why I use it, duh) but every person has their own way of doing things. I suggest you check out other dotfile repos for inspiration. I'd be happy to hear about your approaches in the comments section.

In order to support customization, all of my main dotfiles end with something like this (you can find an example of my `.functions` file here).

```
# This should be the last line of the file# For local changes# Don't make edits below this[ -f ".functions.local" ] && source ".functions.local"
```

Basically, each file named `.filename` (for example) ends with something like:

```
[ -f ".filename.local" ] && source ".filename.local"
```

This command checks to see whether a file with the same filename but an extension of `.local` exists, and if it does, sources it.

Since this is the last line of the file, the `.filename.local` file is the last to be sourced and so all the settings and configurations defined in it persist and can override the ones put in other files.

This allows people experimenting with my dotfiles to customize them without having to modify my code at all! Neat, huh?

Also, all `.local` files are ignored in my `.gitignore`.

Next Steps

At this point, you know almost all you need to know about dotfile management, but there are a few things that this and the previous article

miss out on:

- Customizing your prompt (this is truly an art and you should invest time in this)
- Using a dotfile manager such as dotty (or autodot, which is something I came up with)

I'd recommend looking into these things when you can.

Conclusion

That's all for this series on dotfile management. I'd love to hear your opinions about this article, and more than that, to see how creative you guys get with your dotfiles. Reach out to me in the comments to tell me how you liked this article and share any other dotfile management tricks you know of. Also, I'd be very happy if you take a minute to gimme feedback on my dotfiles, or suggest improvements using GitHub issues.

I loved the idea of dotfiles so much that it inspired me to create a basic dotfile management framework - **autodot**. The framework is in its infancy, so I'm looking for enthusiastic people who can give me feedback for the framework, contribute to it by telling me about bugs and making feature requests, and contribute to the code and documentation. Do take some time out for this! :)

ajmalsiddiqui/autodot
*autodot — A dotfile management system that makes sharing your dotfiles easy while keeping you in the loop.*github.com

Also, connect with me on GitHub and LinkedIn.

Good luck and Happy Coding! :)