cat /var/log/life

[+]

# Managing Your Dotfiles

03 Aug 2014 · 10 min read — shared on Reddit

Unix command-line programs are usually configured using plain-text hidden files[1], commonly referred to as dotfiles, that are stored in the user's home directory. For developers who spend any amount of time in the terminal, tweaking and optimizing settings for command-line programs can be an extremely worthwhile investment. Something as easy as light shell customization can result in a significant increase in productivity.

Looking around the Internet, many individuals have open sourced their dotfiles for others to see. A quick GitHub search returns tens of thousands of results. Does this mean that you should run a quick `git clone` and be off and running someone's awesome dotfiles? **No.**

## Dotfiles Are Not Meant to Be Forked

Some developers believe that "dotfiles are meant to be forked". I disagree.

Dotfiles are supposed to contain *your personal settings* — what works for someone else isn't necessarily optimal for *you*. If certain configurations worked for everybody, those settings would have been built into programs as defaults. Blindly cloning someone else's dotfiles, especially without having an understanding of how everything works, is **not** the optimal approach.

That being said, I am a big fan of sharing dotfiles and taking inspiration from others' configurations. My own dotfiles have been open sourced, and parts of my configuration are inspired by other people's dotfiles. However, my dotfiles are my own personal settings, and I understand every bit of code and configuration in there. That is important. There are some people who have forked my dotfiles, but I do not recommend that approach. Copying parts of my configuration, however, is encouraged! It's best when you understand what you are using — that's why I've tried to keep my dotfiles organized and well-commented.

## Organizational Approaches

Most likely, as a developer, you will keep using and improving your dotfiles for your entire career. Your dotfiles will most likely be the *longest project you ever work on*. For this reason, it is worthwhile to organize your dotfiles project in a disciplined manner for maintainability and extensibility.

Getting started can feel like a daunting task, partly due to the numerous organizational approaches and choices to make. In this post, I will discuss the tradeoffs between some of the different approaches of structuring your dotfiles, and I'll explain the choices I ended up making in the process of setting up my own dotfiles.

## Version Control

It is worth keeping your configuration under version control for several reasons. Keeping a history of your changes allows you to safely experiment with new configurations, revert bad changes, and review the details of past changes. When working on multiple systems, choosing to use a version control system is a no-brainer — version control helps you keep your dotfiles in sync while properly dealing with changes being made on different systems.

It is possible to use dedicated tools such as rcm or homesick for this purpose. These tools usually handle both versioning and installation of dotfiles, which is accomplished by using a dedicated version control system behind the scenes and providing a front end designed specifically for managing dotfiles. These tools are additional dependencies that need to be installed prior to setting up your dotfiles. They are fairly heavyweight, so I prefer to avoid external dependencies in favor of a simpler, self-contained setup. As a bonus, there is one less thing that needs to be done when setting up new systems.

I prefer to use Git as my version control system (VCS). Other software such as Mercurial or Subversion would be equally well suited to the task. It is best to choose a VCS that you are comfortable using[2].

There are two main approaches for using a VCS to keep track of configuration files: you can have your entire home directory under version control (and ignore non-configuration files), or you can keep your configuration files in a separate directory

and copy or link them into place. The latter approach is vastly superior for several reasons:

- There is no possibility of accidentally deleting your files. With your entire home directory under version control, running something like a `git clean` could wipe out everything in your home directory that is not tracked by your VCS.
- It is possible to track configuration files that belong somewhere other than `$HOME`.
- Installing dotfiles on new machines is easier. All that is required is cloning your dotfiles followed by copying or linking files. Keeping the entire home directory under version complicates installation.

## Initialization

If you have already done some dotfiles customization, it's pretty easy to start tracking those configuration files in a VCS. Create a new folder to store your dotfiles, move everything over, perhaps running a quick `ls -A` in your home directory to make sure you haven't missed anything, and run a `git init` or the equivalent for your VCS.

If you haven't done any dotfiles customization, this is a great time to get started! Create a new repository for your dotfiles and start with something simple like shell prompt customization.

## Installation

With your dotfiles in their own repository, there are two possible ways to install dotfiles on systems: copying or symbolically linking files. Symbolic links are better — using symlinks, there is no need to manage discrepancies between copies. Changes to configuration files are changes to the working copy in the repository.

The best approach is to automate the installation process in a "one click" idempotent install script, a program that be run multiple times without having any effect beyond the initial application. That way, when setting up a new machine or syncing changes between machines, all it takes is the execution of a single program to complete the installation process.

The installer can be something as simple as a shell script like this:

```
#!/bin/bash

BASEDIR="$(cd "$(dirname "${BASH_SOURCE[0]}")" && pwd)"

# vim
ln -s ${BASEDIR}/vimrc ~/.vimrc
ln -s ${BASEDIR}/vim/ ~/.vim

# zsh
ln -s ${BASEDIR}/zshrc ~/.zshrc

# git
ln -s ${BASEDIR}/gitconfig ~/.gitconfig
```

This works, and a lot of people use scripts just like it, but it's not very robust. Among other things, existing files could be overwritten as a result of the installation process. It's possible to write a more robust shell script that accomplishes the same thing, and many people have done so. Looking through people's collections of dotfiles on GitHub, there are hundreds of ad-hoc install scripts (with varying degrees of robustness) that have been written for this particular purpose (and I had written one too). On the other hand, there are more heavyweight tools like vcsh that have more functionality (but most tools are external dependencies that have to be installed before your dotfiles).

I prefer sticking to approaches that do not require external dependencies that are not tracked by my dotfiles repository. In the past, I used a custom install script to install my dotfiles. In my opinion, that's a fine approach.

Currently, I use Dotbot to install my dotfiles (full disclosure: I am the author of Dotbot). It's actually tracked directly in my dotfiles repository as a git submodule, and it has no external dependencies, so there is no extra work required when setting up new systems. All I have to do is run `./install`, and everything else is automated. For me, this is a perfect balance — it is lightweight, clean, and robust, and it is a dependency that is managed directly in my dotfiles repository.

Dotbot can provide "one click" idempotent installations that are specified using JSON files.

**Update (12/8/2014):** Dotbot now supports a much cleaner YAML-based (and fully backwards-compatible!) syntax for configurations.

Here is the configuration that does the equivalent of the script above:

```
[
    {
        "link": {
            "~/.vimrc": "vimrc",
            "~/.vim": "vim/",
            "~/.zshrc": "zshrc",
            "~/.gitconfig": "gitconfig"
        }
    }
]
```

Dotbot can do more than just link files (as you can see in a more sophisticated configuration). For more details, see the project page. To see Dotbot in action, you can take a look at public dotfiles repositories that are using Dotbot. Here are a few examples:

- anishathalye/dotfiles
- crzysdrs/dotfiles

Whether you choose to use a custom install script or a specialized tool, symbolically linking files is superior to copying, and having an idempotent install script greatly simplifies the process. Tools like Dotbot or alternatives such as rcm, homesick, vcsh, dots, deedot, GNU Stow, homedir, dotfiles, sync-dotfiles, dotfile-manager, or dot-files can make your life much easier.

## Dependencies and Plugins

There are many handy scripts and plugins available on the Internet that can make your life a lot easier, and it can be convenient to include these as part of your dotfiles. One approach to managing these is to copy files directly into your dotfiles repository and treat them like any other configuration files. This method might be slightly easier than alternatives if you plan on modifying these plugins directly, but it isn't the best approach for most cases. Most plugins will provide simple ways to customize settings, and most likely, you will want to be able to update your plugins easily, and this is painful when plugin files have been copied into place.

Submodules are the best way of managing dependencies[3] that are under version control. Dependency versions are tracked properly, and it is easy to check out the proper versions of dependencies and update dependencies. Many VCSs have first-class support for this feature — Git has submodules and Mercurial has subrepositories. My

description will be given in terms of Git submodules. The commands for other VCSs are analogous.

Adding new submodules to your repository is easy. For example, if you wanted to add Dotbot as a submodule in the `dotbot` directory, you could run the following command:

```
git submodule add https://github.com/anishathalye/dotbot dotbot
```

Initializing all your submodules and checking out the specified versions is also easy. This operation is idempotent, and it can be run after cloning or updating your dotfiles:

```
git submodule update --init --recursive
```

Upgrading submodules to the latest published version can be done by running this command:

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

After upgrading your dotfiles, the new versions of plugins can be checked into your git repository.

All kinds of interesting plugins are available on the Internet. You can get syntax highlighting for your shell. You can manage your vim plugins with ease using Pathogen. Your vim plugins (along with pathogen itself) can all be installed as submodules. For the most part, anything that can be installed by running a `git clone` can be installed as a submodule by running `git submodule add` instead.

When using submodules in your dotfiles, it is handy to have submodule initialization as part of your installation script. Adding `git submodule update --init --recursive` to your installation process will take care of that. This can be done using Dotbot by adding the following shell command:

```
[
    {
        "shell": [
            ["git submodule update --init --recursive",
                "Installing submodules"]
        ]
    }
]
```

# Local Customization

When managing dotfiles on multiple machines, the majority of your configuration will be the same between machines, and there will be some minor differences between installations. One way to handle this is to use branches in your main dotfiles repository, but that can quickly get out of hand. The cleanest way is to have a main dotfiles repository and have a separate repository for local customizations that override defaults. To see how I did this, you can take a look at how I organized my dotfiles-local repository. This has the additional advantage that your main dotfiles can be open sourced, and your local customizations can be kept private (because they may contain things such as SSH aliases, etc).

This secondary repository should have branches for all your different machines (or groups of machines). You should have an install script for this repository as well. I use Dotbot for both my main dotfiles repository and my local customizations.

It is easy to structure your dotfiles so settings are easy to override. The basic pattern is to check for the existence of an overriding file and source the file if it exists.

## Shell

You can add the following to the end of your shell rc file to enable overriding:

```
if [ -f ~/.zshrc_local ]; then
    source ~/.zshrc_local
fi
```

If you're using a different shell, you probably want to name the file differently.

## Git

You can add the following to the end of your `.gitconfig` file to enable overriding:

```
[include]
        path = ~/.gitconfig_local
```

## Vim

You can add the following to the end of your `.vimrc` file to enable overriding:

```
let $LOCALFILE=expand("~/.vimrc_local")
if filereadable($LOCALFILE)
```

```
    source $LOCALFILE
endif
```

### tmux

You can add the following to the end of your `.tmux.conf` file to enable overriding[4]:

```
if-shell "[ -f ~/.tmux_local.conf ]" 'source ~/.tmux_local.conf'
```

## Miscellaneous Tips

### Path to Dotfiles

It is nice to be able to put your dotfiles anywhere on a system and have your files linked into place. Sometimes, it can be handy to have a persistent path to your dotfiles repository. This can be done by always keeping your dotfiles in a specific directory such as `~/dotfiles`, but there is a cleaner way to achieve the same result. You can create a symbolic link from `~/.dotfiles` to your dotfiles as a part of your installation process. This can be done with Dotbot by adding the following link:

```
[
    {
        "link": {
            "~/.dotfiles": ""
        }
    }
]
```

### Tracking Custom Scripts

You can keep track of your custom shell scripts in your dotfiles, too! The easiest way to do this is to create a `bin/` directory for your shell scripts, and then add the following to your shell rc file:

```
export PATH=~/.dotfiles/bin:${PATH}
```

For this to work properly, make sure you followed the tip above to create `~/.dotfiles` as a symbolic link to your dotfiles directory.

### Tracking Config Files in Other Directories

Sometimes, you may need to keep track of config files in other directories such as `~/.config`, and sometimes, there will be some files in there that you don't want to track in your dotfiles. You could handle this by keeping the entire directory under version control (and linked into place) and ignoring everything except for specific files in a `.gitignore` file. However, this complicates installation when the directory already exists, and it could be dangerous if you ever ran a command like `git clean`. There is a better way to solve this problem.

**Update (12/8/2014):** A new Dotbot feature makes this incredibly easy to do in a clean way. The old method works just fine too.

As a part of your installation process, you can create the directory structure if it doesn't already exist (using `mkdir -p`), and then you can link individual files or folders into place. This can be done with Dotbot too. For example, this is how I'm setting up my Terminator config that resides in the `config/terminator/` directory in my dotfiles repository:

```
[
    {
        "shell": [
            ["mkdir -p ~/.config", "Creating config folder"]
        ]
    },
    {
        "link": {
            "~/.config/terminator": "config/terminator/"
        }
    }
]
```

## What's Next?

Now that you've got your dotfiles set up, what's next? There are tons of great resources on how to customize individual tools, from your command shell to your text editor. You can tweak everything to the finest detail, and then you can keep updating your settings as your preferences change over time.

If you don't have any personal data in your repository, you can consider open sourcing your dotfiles so others can learn from your setup.

1. In Unix, files beginning with a `.` (a period) are hidden files. The reason for this is explained in this post. Oddly enough, the result of what was essentially a software bug that was introduced in the early 1970s is still present in modern Unix systems! ↩

2. If you are not yet comfortable using a version control system, it is worth learning how to use one! Today, Git is the program of choice for most software developers. For an excellent reference, take a look at this site. ↩

3. There is some controversy over the effectiveness of submodules. While submodules are not necessarily the right choice for every project, they work very well for this use case! Using submodules becomes tricky if you are changing code or making commits in submodule repositories, but for plugins in your dotfiles, this will not be the case. ↩

4. This does not work in older versions of tmux due to this bug. It works just fine in tmux 1.9a. ↩

## Recent Posts

**Experiments in Constraint-based Graphic Design** 12 Dec 2019

**Gemini: A Modern LaTeX Poster Theme** 19 Jul 2018

**Turning a MacBook into a Touchscreen with $1 of Hardware** 03 Apr 2018