Dotfiles: automating macOS system configuration

kalis.me/dotfiles-automating-macos-system-configuration

October 29, 2017

It was in October of 2015 when I was casually stalking my Lead Developer's GitHub profile. There, I happened to come across a repository named *dotfiles*. When I asked him about it he told me that it contained scripts to set up his computer from scratch, installing all of his packages and setting up his environment. This introduced me to the concept of dotfiles. Immediately excited, I forked his repo into my own and got started.

Creating my dotfiles

After initially forking the repo I started customising it to my likings. It began with listing my own packages and adding a few function aliases in my `.bashrc`, but soon I noticed that this was not nearly enough, so I started researching ways to improve my system configuration. Luckily, there are a lot of other people sharing their own dotfiles to learn from.

Some of the best resources I found along the way were:

- Zach Holman's dotfiles, which is definitely one of the most well-known dotfile repositories around the web. With good reason, as they contain all the features a dotfiles configuration really needs.
- Mathias Bynens' dotfiles, which helped a lot with configuring macOS settings. Many people include a version of this configuration file in their own macOS dotfiles.
- github.dotfiles.io, a great resource with links to many more exemplary dotfiles.
- awesome-dotfiles, another extensive list to start out and look for inspiration.

Why dotfiles are awesome

When I started, my system configuration was all over the place, with too many packages to handle, and settings I didn't even remember setting. By fully documenting everything that is necessary to make your computer *yours*, you get a better overview about what makes your workstation tick. **This is already a big improvement, as it allows you to think critically about your configuration.** Besides, it offers peace of mind to know how your computer is set up. With dotfiles, you can mess around with new packages and settings quite easily, without ever having to worry about *really* breaking something important. If you would break something, your dotfiles are always right there to back you up.

Photo by Simson Petrol / Unsplash

More importantly though, it allows you to stop worrying about your computer breaking down or other doomsday scenarios. I remember how I accidentally spilled a glass of water over my laptop a few years ago. Back then, I felt my heart drop into my stomach, because I knew the consequences would be grave if my computer broke down. Today, however, I really wouldn't break a sweat. With my documents in the cloud, and my dotfiles in place, I can have a new computer ready in hours, providing the exact same user experience as the first one.

Dotfiles, how do they work?

I hope you're as convinced as I am about setting up your dotfiles, but how do you get started?

I'll begin with saying that my current dotfiles look *nothing* like those I started out with. Since then, I switched to a new shell, and I changed my main text editor *twice*. **The most important thing is to get started**, and make changes later if necessary.

The components

To get started, let's see what the most important parts are to any sophisticated dotfiles.

1. Package and application installation

The packages and applications you run on your computer are one of the most important things to document, and one of the more tedious jobs to do if they need to be reinstalled manually. Which is why we're going to automate it. Luckily, Homebrew and Homebrew Cask on macOS can go a long way for us. With its Brewfile capabilities, we can run `brew bundle` to install all packages and applications listed in our Brewfile. With mas, we can even install Mac App Store-exclusive applications in the same way.

This Brewfile covers a lot of packages, but for the rest (e.g. npm packages) we'll have to write an installation script. If you're using Linux, you will have to use the same method for regular packages as well.

```
#! /usr/bin/env sh

DIR=$(dirname "$0")
cd "$DIR"

COMMENT=\#*

sudo -v


brew bundle

find * -name "*.list" | while read fn; do
    cmd="${fn%.*}"
    while read package; do
        if [[ $package == $COMMENT ]];
        then continue
        fi
        if [[ $cmd == code* ]]; then
            echo "$cmd $package"
            $cmd $package
        else
            echo "$cmd install $package"
            $cmd install $package
        fi
    done < "$fn"
done
```

This script looks for *.list files in the script folder, then installs every package listed in the files, using the corresponding installer (so npm.list will be installed using npm, and apt.list will be installed with apt). New lists can easily be added when necessary.

2. macOS system settings

Now that every required package has been restored to the computer, it is time to get the system settings going. Thankfully, we have the aforementioned dotfiles repository of Mathias Bynens. This includes a huge script containing system settings. All that's left to do is to go through the list, and change the settings to our own preferences.

3. Shell configuration

It doesn't matter whether you prefer Bash, Fish, or Zsh, the command line is arguably one of the most important tools for any power user, so configuring it in the right way is an equally important task. I personally use Fish, which I would whole-heartedly recommend for its simplicity and ease-of-use, but the process for configuring other shells is quite similar.

```
#! /usr/bin/env sh

DIR=$(dirname "$0")
cd "$DIR"

. ../scripts/symlink.sh

SOURCE="$(realpath .)"
DESTINATION="$(realpath ~/.config/fish)"

echo "Source path:\t\t $SOURCE"
echo "Destination path:\t $DESTINATION"

echo "Creating destination folders"
mkdir -vp "$DESTINATION/functions"
mkdir -vp "$DESTINATION/completions"

find * -name "*.fish" | while read fn; do
   symlink "$SOURCE/$fn" "$DESTINATION/$fn"
done

grep /usr/local/bin/fish /etc/shells &> /dev/null
if [ $? -ne 0 ]; then
   sudo bash -c "echo /usr/local/bin/fish >> /etc/shells"
   sudo chsh -s /usr/local/bin/fish
fi


fish -c "setup"
```

Here we see one of the returning patterns of symlinking the needed configuration files from the repository to their designated location on the file system. These configuration files include a quite small `config.fish`, the Fish version of `.bashrc`, and some custom Fish functions under the `functions/` directory, as well as custom shell-completions under the `completions/` directory.

```
function symlink() {
   OVERWRITTEN=""
   if [ -e "$2" ] || [ -h "$2" ]; then
      OVERWRITTEN="(Overwritten)"
      rm -r "$2"
   fi
   echo "$2 -> $1 $OVERWRITTEN"
   ln -s "$1" "$2"
}
```

You might have noticed the `symlink` function, which is imported from a file with the same name. This is a small wrapper around `ln -s`, with some more verbosity, since symlinking is such a big part of dotfiles.

4. Text Editor configuration

As you will notice, the process for this is quite similar to the one we saw for Shell configuration, and the other ones for any application-specific configurations. I decided to give text editor configuration it's own section, as it is something that many developers use on a day-to-day basis. Now, the configuration for a text editor is very dependent your editor of choice. Back when I used Sublime Text, I remember that it was specifically cumbersome to correctly add all plugins to my dotfiles, but for Atom and Visual Studio Code, this is a lot easier.

```
#! /usr/bin/env sh

DIR=$(dirname "$0")
cd "$DIR"

. ../scripts/symlink.sh

SOURCE="$(realpath .)"
DESTINATION="$(realpath ~/Library/Application\ Support/Code/User)"

echo "Source path:\t\t $SOURCE"
echo "Destination path:\t $DESTINATION"

echo "Creating destination folders"
mkdir -vp "$DESTINATION"

find * -not -name "setup.sh" -type f | while read fn; do
    symlink "$SOURCE/$fn" "$DESTINATION/$fn"
done
```

My configuration for VS Code consists of one `settings.json` file, combined with its own list of packages under the package installation. I also have a separate `.vimrc` from amix/vimrc, so that I still have some syntax highlighting when I have to do some in-terminal programming.

5. General application configuration

By now, all packages and applications are installed, and most of your environment has been configured. However, there are other applications for which the configurations need to be saved. This is the part where you can repeat the pattern from before, symlinking the configuration files to their designated location on the file system. I personally have such files for Git, Karabiner, and Hammerspoon.

These five points are the main building blocks to setting up dotfiles, but you can extend them with anything that you feel is missing from your own set up. One thing that has increased my productivity by quite a margin is repository management, which I have expanded on in a different article.

Tying the dotfiles together

Now that we've seen which different parts dotfiles should contain, the only thing left is tying it all together. To do this we will need to have a setup script calling all others.

```
#! /usr/bin/env sh


sudo -v


while true; do sudo -n true; sleep 60; kill -0 "$$" || exit; done 2>/dev/null &

DIR=$(dirname "$0")
cd "$DIR"

xcode-select --install


echo "./packages/setup.sh"
./packages/setup.sh

find * -name "setup.sh" -not -wholename "packages*" | while read setup; do
    echo "./$setup"
    ./$setup
done
```

In this script we first install the Xcode command line tools and then execute the package installation. This is necessary because some of the other setup scripts rely on having these applications installed. Fish can't be configured, for instance, without having Fish installed on the system.

With this bootstrap script set up, the dotfiles are complete, and they're ready to be cloned and run on a new computer. The only thing needed on a new computer is Git and Homebrew, after which you can install your dotfiles by running:

```
ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
brew install git
git clone git@github.com:rkalis/dotfiles.git
cd dotfiles
./bootstrap.sh
```

Hopefully, this article has helped with understanding how dotfiles work, and why you should set them up. If you enjoyed reading this, please consider sharing this on Facebook, Twitter or LinkedIn, or giving my dotfiles repository a ☆ on GitHub.

Let me know about your own dotfiles in the comments below!