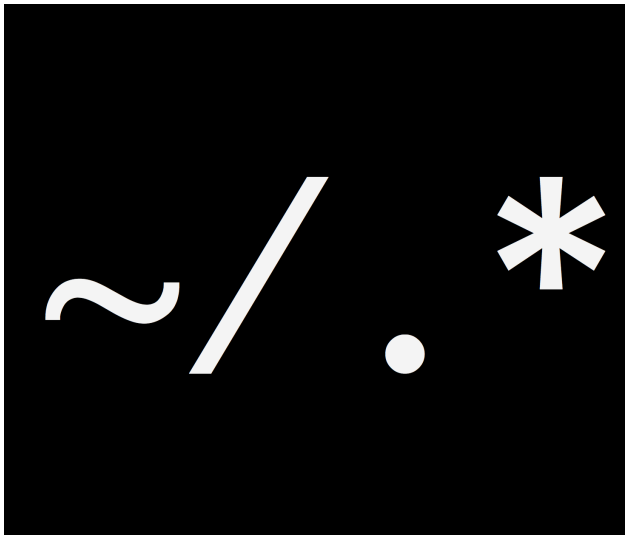

Getting Started With Dotfiles

Lars Kappert Follow
Jul 16, 2014 · 8 min read

You're The King Of Your Castle!

tl/dr; You can set up a new system using dotfiles and an installation script in minutes. It's not hard to create your own repository, and you'll learn a ton along the road. This is truly more about the journey than the destination!

Introduction

Dotfiles are used to customize your system. The "dotfiles" name is derived from the configuration files in Unix-like systems that start with a dot (e.g. *.bash_profile* and *.gitconfig*). For normal users, this indicates these are not regular documents, and by default are hidden in directory listings. For power users, however, they are a core tool belt.

There is a large dotfiles community. And with it comes a large number of repositories and registries containing many organized dotfiles, advanced installation scripts, dotfile managers, and mashups of things people collect in their own repositories.

This article will try to give an introduction to dotfiles in general, by means of creating a basic dotfiles repository with an installation script. It is only meant to provide some inspiration, some pointers to what is possible and where to look for when creating your own.

Note that this writeup has a focus on Linux and macOS based systems.

Automate All The Things!

Ideally, you store your personal files not on your machine only. If you have your files on either local drives (e.g. USB drive, NAS) or in the cloud (Dropbox, Google Docs, iCloud, etc., etc.), you save yourself from the risks of machine theft, damage, or hardware failure.

Now your documents, photos, etc. are kind of safe. Still, if you ever have to setup a system, you need to install every single application again. I can't count the times I needed to find the application's download page, download, install. Next. Next. Again. You forgot one. One more. And I did not even mention the plethora of system preferences and other configurations, which I usually can't remember when I need them. Again, I need to search.

So, how awesome is it that we can automate all this? You may not realize it, but most system tools, applications and settings can be installed in an automated fashion. I don't know about you, but this is like music to my ears!

Today, I could literally throw my laptop out of the window, buy a new one, and be up and running in a matter of minutes (not hours!). Without breaking a sweat (apart from the $$$).

Getting Started

It's pretty simple to get started. You need to organize your dotfiles in some directory. You could do this practically anywhere, like a USB drive or something. Since version control is great, a hosted git repository like GitHub is a great option to store your dotfiles.

An Example Dotfiles Repository

For this example, I'm just going to use a subset of my own dotfiles repo.

Structure

Below is the structure of my dotfiles repo. It's also what we'll use in our walk-through below.

```
.
├── git
│   ├── .gitconfig
│   └── .gitignore_global
```

```
├── install.sh
├── osxdefaults.sh
├── runcom
│   ├── .bash_profile
│   └── .inputrc
└── system
    ├── .alias
    ├── .env
    ├── .function
    ├── .path
    └── .prompt
```

The Dotfiles

We'll be taking a look at the following example dotfiles:

- .bash_profile
- .inputrc
- .alias
- .functions
- .env
- .prompt

.bash_profile

In a Bash shell, this file (or *.profile)* in your home directory is loaded first. What to put in the *.bash_profile* and other dotfiles is truly worth a book alone, but we're going to give it a quick shot here anyway. I like to use a small *.bash_profile* that links to several others that have a dedicated purpose, i.e. one file for the aliases, one for the functions, etc. Here's an example of how you can include (or actually "source" or execute) all files in a folder:

```
for DOTFILE in `find /Users/lars/Projects/.dotfiles`
do
  [ -f "$DOTFILE" ] && source "$DOTFILE"
done
```

Full examples include my own *.bash_profile,* Mathias's *.bash_profile.* Some people like to put most of their startup configuration in one file. This is perfectly fine, as long as you keep it sane and dense.

If you want to dive into startup scripts a bit more, Peter Ward explains about Shell startup scripts, and here's another about startup script loading order.

.inputrc

The behavior of line input editing and keybindings is stored in a *.inputrc* file. Here's an excerpt of my own:

```
# Make Tab autocomplete regardless of filename case
set completion-ignore-case on

# List all matches in case multiple possible completions are possible
set show-all-if-ambiguous on

# Flip through autocompletion matches with Shift-Tab.
"\e[Z": menu-complete

# Filtered history search
"\e[A": history-search-backward
"\e[B": history-search-forward
```

Full example: my *.inputrc.*

.alias

Aliases allow you to define shortcuts for commands, to add default arguments, and/or to abbreviate longer one-liners. Here are some examples:

```
alias l="ls -la"       # List in long format, include dotfiles
alias ld="ls -ld */"   # List in long format, only directories
alias ..="cd .."
alias ...="cd ../.."
alias ....="cd ../../.."

# Recursively remove .DS_Store files
alias cleanupds="find . -type f -name '*.DS_Store' -ls -delete"
```

Full examples: my *.alias*, Mathias's *.aliases*

.functions

Commands that are too complex for an alias (and perhaps too small for a stand-alone script) can be defined in a function. Functions can take arguments, making them more powerful.

```
# Create a new directory and enter it
function mk() {
  mkdir -p "$@" && cd "$@"
}

# Open man page as PDF
function manpdf() {
 man -t "${1}" | open -f -a /Applications/Preview.app/
}
```

Full example: Mathias's *.functions*.

.env

Environment variables can go in another dotfile:

```
export
PATH="/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:$DOTFILES_DIR/bin"
export EDITOR="subl -w"
export CLICOLOR=1
export LSCOLORS=gxfxcxdxbxegedabagacad

# Tell grep to highlight matches
export GREP_OPTIONS='–color=auto'

# Case-insensitive globbing (used in pathname expansion)
shopt -s nocaseglob

# Autocorrect typos in path names when using `cd`
shopt -s cdspell
```

Full example: Mathias's *.exports*.

.prompt

A custom prompt can be convenient. You could, for example, show where you are in the directory tree, and/or which git branch you're currently working with. There's plenty of options here, but personally I'd like to keep this a bit easy on the eyes. Here's my prompt (without colors):

```
[lars ~/Projects/dotfiles master] $
```

Examples: my *.prompt*, Color Bash Prompt, Sexy Bash Prompt, How to customize…

Other Dotfiles

Many packages store their settings in a dotfile, e.g.:

- *.gitconfig* for Git
- *.vimrc* for Vim

Because these are basically simple text files, they are perfect to store in your dotfiles repo!

Installing the Dotfiles

To "activate" the dotfiles, you can either copy or symlink them from the home directory. Otherwise they're just sitting there being useless.

Beware you probably already have a *.bash_profile* and *.gitconfig* in the user folder. So please be careful here. With great power comes great responsibility. Probably it's best to backup important files before you're moving them around.

Let's assume you have the relevant dotfiles together in *~/.dotfiles*. You can create a symlink from here to the directory where they are expected (usually your home directory, ~):

```
ln -sv "~/.dotfiles/runcom/.bash_profile" ~
ln -sv "~/.dotfiles/runcom/.inputrc" ~
ln -sv "~/.dotfiles/git/.gitconfig" ~
```

We already have the core of a dotfiles setup.

Installation Script

You may want to have an installation script to automate symlinking the dotfiles in the repo to your home directory. But there's more we can put in a script that we run once to install a new system. See this *install.sh* for an example. Also make sure to check out other people's scripts for more ideas and inspiration.

To install the dotfiles on a new system, we can do so easily by cloning the repo:

```
git clone https://github.com/webpro/dotfiles.git
cd dotfiles
```

Then do the symlinking (either manually or with a script), et voilà! Now I would like to show you some more neat things you can do in your dotfiles.

Homebrew and Homebrew Cask

Let's install my favourite combo for package management in macOS, Homebrew and Homebrew-Cask:

```
ruby -e "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/master/install)"
brew tap caskroom/cask
brew tap caskroom/versions
```

This opens up a giant repository of system tools you can install from the command line. Here's a short sample:

```
brew install node
brew install git
brew install wget
```

And how about macOS applications? Thanks to Homebrew-Cask we have the power to install GUI applications in macOS from the command line:

```
brew cask install atom
brew cask install dropbox
brew cask install firefox
brew cask install google-chrome
brew cask install spotify
brew cask install sublime-text3
brew cask install virtualbox
brew cask install vlc
```

Please note that you can install multiple applications with a single command:

```
brew cask install alfred dash flux mou
```

macOS defaults

Many, many macOS settings can be set from the command line. Here's just a small sample to get an idea:

```
# Finder: show hidden files by default
defaults write com.apple.finder AppleShowAllFiles -bool true

# Automatically hide and show the Dock
defaults write com.apple.dock autohide -bool true

# Save screenshots to the desktop
defaults write com.apple.screencapture location -string
“$HOME/Desktop”

# Save screenshots in PNG format (other options: BMP, GIF, JPG, PDF,
TIFF)
defaults write com.apple.screencapture type -string “png”

# Display full POSIX path as Finder window title
defaults write com.apple.finder _FXShowPosixPathInTitle -bool true

# Disable the sound effects on boot
sudo nvram SystemAudioVolume=” “
```

We must credit Mathias Bynens here for creating and maintaining an awesome collection of macOS defaults in his dotfiles.

To apply the macOS defaults you've stored in e.g. *macosdefaults.sh*:

```
source macosdefaults.sh
```

This line is a perfect candidate to include in your installation script.

Updating Your System

It's fine to run the installer script again, e.g. to fix some symlinks or update packages (it should be idempotent). But it's better and faster to run a couple of update commands separately. Here are some example commands to put in an alias or function to update macOS, Homebrew, npm, and Ruby packages:

```
# Update App Store apps
sudo softwareupdate -i -a

# Update Homebrew (Cask) & packages
brew update
brew upgrade

# Update npm & packages
npm install npm -g
npm update -g

# Update Ruby & gems
sudo gem update –system
sudo gem update
```

Next steps

I have always enjoyed to wade through the various existing dotfiles repos and find real gems out there. Sometimes it takes real effort to make something work the way you want it to, but eventually it makes your dotfiles really *yours*.

You might have missed tools like Zsh, Vim, and many more. Well, my apologies for that, but you would never reach the end of this article.

In any case, there are plenty of great resources and dotfiles covering these as well. My curated awesome-dotfiles list might be a good start.

If you have nice ideas to share or want to collaborate, feel free to send me a tweet or open a PR!

Dotfiles OSX Shell Linux Macos