


# An introduction to Dotfiles: how to take control of your development environment

 [freecodecamp.org/news/dive-into-dotfiles-part-1-e4eb1003cff6](https://www.freecodecamp.org/news/dive-into-dotfiles-part-1-e4eb1003cff6)

May 27,  
2018



by Mohammed Ajmal Siddiqui

Note: This is a very basic, introductory article. If you already know the fundamentals of dotfile management, I'd recommend you read my [second article](#).

As developers, we strive to minimize the time we spend on redundant things, like setting up our environment, writing boilerplate code, and basically not doing anything that does not concern the fun part of coding - building new stuff.

In this context, imagine a perfect world where tiny commands carry out incredibly complex tasks tailored to your needs, where you could buy a new laptop today and install all the tools and packages you need and setup your development environment with nothing but a couple of terminal commands, and where everything is magic.

This digital fairyland can be made, and with ease. And there is a name for this magic: dotfiles.

Without further ado, let's unravel the secrets of the dotfiles!

## Introduction

---

**Note:** This article assumes that you're working with a Unix-like operating system and it relies heavily on Unix terminal commands and shell scripting. If you're not familiar with these, I recommend learning the basics and coming back here. [Here's](#) a primer to shell scripting.

In UNIX-like systems, a lot of configuration files and the like are preceded with a dot(.). These files are hidden by the OS by default, and even the `ls` command doesn't reveal their presence (we'll get to how to find these files in a bit). Since these files are preceded by a dot, they're called dotfiles. Duh.

So how do we find these legendary files if they're hidden by default? Pop open a terminal and do this:

Note: The "\$" sign is not meant to be typed in the terminal. It represents the fact that the text after it is supposed to be typed in a terminal prompt.

```
$ cd ~$ ls -a
```

So what does this do?

The first command ( `cd ~` ) moves into the home directory (the "~" symbol represents the home directory). The home directory is where most of your config files are found. So we move there first.

The second command lists the files and folders in the current directory. But there's some magic here. The `-a` flag instructs the command to include hidden files in the list.

Bingo! We can now see the dotfiles!

## Modifying the `.bash_profile`

Usually, the first file that most people modify when they enter the world of dotfiles is the `.bash_profile` or the `.bashrc`. And for good reason. This file is loaded when you start your terminal, and its commands are executed at terminal startup.

One reason why you might want to modify your `.bash_profile` is to customize the look of your terminal (to be specific, your terminal prompt). This is an art and a science in itself and probably should have an entire book dedicated to it, so I won't cover this topic much in this article. You can get started with customizing your prompt with [this](#) article.

Instead, let's look at two common shell constructs that are perhaps among the most important and useful parts of dotfiles: aliases and functions.

## Aliases

Aliases are simply short names/acronyms that you can assign to a longer sequence of commands in order to reduce how long you take to type it and thus increase your speed.

For example, almost every developer uses git. Anyone who uses the git CLI (and let's face it - you should be using the git CLI), probably has used long commands like these:

```
// commit changes$ git commit -m "some changes"

// push changes to the master branch of origin$ git push origin master
```

These commands are quite a bit to type. If you think they're not, you will change your mind after you start using aliases.

Type the following in your shell prompt:

```
$ alias gpom='git push origin master'
```

Now when you type `gpom`, `git push origin master` is executed! You've gone from 4 **words** to 4 **letters**! ?

But there's a problem. Close your terminal, restart it, and try `gpom` again. Your alias is gone! This is because the alias is defined for the current terminal session.

So how do we get around this and make our aliases stick?

Remember we talked about a file whose commands are executed when a terminal is started? Bingo!

Add the following line to your `.bash_profile` or `.bashrc` and save it:

```
alias gpom='git push origin master'
```

Now, whenever you start a bash terminal, the above alias is created. Life is already starting to get awesome!

**Note:** You can use the `nano` text editor to edit your text files. When in the home directory, type `nano .bash_profile` to open the file using nano, make your changes, and save the file by hitting `Ctrl+X` and then `y` when prompted. `Vim` is another text editor you can use.

Since aliases essentially replace the full command, you can aliases as part of a common multi command CLI tool like git to make all its commands easier. Just add this to your `.bash_profile` :

```
alias g='git'
```

And you can type "g" instead of "git" wherever you want to use "git". Sweet!

Here are a few common aliases you might wanna use:

```
alias home='cd ~'alias ..='cd ..'alias '?=man'# Git CLI aliasesalias g='git'alias gi='git init'alias gra='git
remote add'alias gs='git status'...# Aliases for NPMalias nr='npm run'alias ni='npm install'alias
nid='npm install -D'...
```

## Functions

---

Aliases can go a long way in improving our workflow, but there's one thing they can't do: work with arguments.

Let's say you were tired of executing two commands to make a new directory and `cd` into it:

```
$ mkdir new_folder$ cd new_folder
```

And you wanted to make an alias for this. But you can't, since both `mkdir` and `cd` take arguments, and you can't pass arguments to aliases.

So what now? Remember, there's a super common programming construct that takes arguments? Yup, functions! Shell scripts can have functions that can take arguments. Awesome! If you're a little rusty with functions in shell scripts, [here's](#) a little reminder.

You can turn the above sequence into a shell function like this (this example was taken from the [dotfiles of mathiasbynens](#), who has some of the most popular dotfiles around. Other people with excellent dotfiles to refer to are listed and linked to at the end of the article):

```
# Create a new directory and enter itfunction mkd() {  mkdir -p "$@" && cd "$_";}
```

Again, you can put this in your `.bash_profile` and the function will be accessible during any terminal session.

**Note:** You'll have to restart your terminal for any changes to your `.bash_profile` to take effect. If this is a chore, run `source .bash_profile` to add your changes to the current terminal session. Even better, in the spirit of dotfiles, make an alias like `alias reload='source .bash_profile' !`

## Dotfiles and Sharing

---

Why do people commit their development environments — their dotfiles — to version control? Why do they put it up on GitHub for everyone to see? Same reason as always: to track how your dotfiles evolve over time and, most importantly, to **share your dotfiles and inspire other people**.

If you look at any mature dotfiles repo, you'll realize that there are always snippets taken from other dotfiles repos and the like. They may even have multiple contributors and maintainers. We share dotfiles to collectively help each other build better environments and workflows.

This also allows people to use features of version control to make each others' dotfiles better. One example of this is using the GitHub Issue Tracker to discuss issues and improvements.

I was inspired to work on my dotfiles from the dotfiles of [pradyunsg](#), who has an impressive dotfiles repo of his own.

My own dotfiles are fairly basic and very immature right now, and they'll get better over time. But this also means that beginners in the world of dotfiles will be less intimidated when they check the repo out.

Like many other people, I've added some support for customization of my dotfiles, so it might be a good idea for people who are new to the idea of dotfiles to fork the repo and try making it their own. More details in the repository. Do check them out and give me feedback!

Here's a list of a people whose dotfiles are much more expansive and might inspire you:

## Conclusion

---

These are the very fundamentals of creating your development environment using dotfiles. However, there is a lot more to this, which we'll continue looking at in the next article in this series.

Some of the topics we'll look at in the next article in the series are:

- Creating an environment to organize, track and painlessly work with dotfiles
- Splitting our dotfiles to make managing them easier and more scalable (notice it is *dotfiles*, not *dotfile*)
- Writing scripts to setup (bootstrap) a new system with our dotfiles
- Making our dotfiles easy to share by adding support for customization

That concludes the first part of the series on dotfiles! Here's a link to the next one.

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](https://github.com/ajmalsiddiqui/autodot)

Also, connect with me on [GitHub](#) and [LinkedIn](#).

*Good luck and Happy Coding! :)*

Learn to code for free. freeCodeCamp's open source curriculum has helped more than 40,000 people get jobs as developers. Get started

