Moving to zsh

Apple has announced that in macOS 10.15 Catalina the default shell will be `zsh`.

In this series, I will document my experiences moving `bash` settings, configurations, and scripts over to `zsh`.

`zsh` (I believe it is pronounced *zee-shell*, though *zish* is fun to say) will succeed `bash` as the default shell. `bash` has been the default shell since Mac OS X 10.3 Panther.

> *This series has grown into a book: reworked and expanded with more detail and topics. Like my other books, I plan to update and add to it after release as well, keeping it relevant and useful. You can order it on the Apple Books Store now.*

Why?

The `bash` binary bundled with macOS has been stuck on version 3.2 for a long time now. `bash` v4 was released in 2009 and `bash` v5 in January 2019. The reason Apple has not switched to these newer versions is that they are licensed with GPL v3. `bash` v3 is still GPL v2.

`zsh`, on the other hand, has an 'MIT-like' license, which makes it much more palatable for Apple to include in the system by default. `zsh` has been available as on macOS for a long time. The `zsh` version on macOS 10.14 Mojave is fairly new (5.3). macOS 10.15 Catalina has the current `zsh` 5.7.1.

Is bash gone!?

No.

macOS Catalina still has the same `/bin/bash` (version 3.2.57) as Mojave and earlier macOS versions. This change is only for new accounts created on macOS Catalina. When you upgrade to Catalina, a user's default shell will remain what it was before.

Many scripts in macOS, management systems, and Apple and third party installers rely on `/bin/bash`. If Apple just yanked this binary in macOS 10.15 Catalina or even 10.16. Many installers and other solutions would break and simply cease to function.

Users that have `/bin/bash` as their default shell on Catalina will see a prompt at the start of each Terminal session stating that `zsh` is now the recommended default shell. If you want to continue using `/bin/bash`, you can supress this message by setting an environment variable in your `.bash_profile` or `.bashrc`.

```
export BASH_SILENCE_DEPRECATION_WARNING=1
```

You can also download and install a newer version of bash yourself. Keep in mind that custom bash installations reside in a different directory, usually `/usr/local/bin/bash`.

Will bash remain indefinitely?

Apple is strongly messaging that you should switch shells. This is different from the last switch in Mac OS X 10.3 Panther, when Apple switched the default to `bash`, but didn't really care if you remained on `tcsh`. In fact, `tcsh` is still present on macOS.

Apple's messaging should tell us, that the days of `/bin/bash` *are* numbered. Probably not *very* soon, but eventually keeping a more than ten year old version of `bash` on the system will turn into a liability. The built-in bash had to be patched in 2014 to mitigate the 'Shellshock' vulnerability. At some point Apple will consider the cost of continued maintenance too high.

Another clue is that a new shell appeared on macOS Catalina (and is mentioned in the support article). The 'Debian Almquist Shell' `dash` has been added to the lineup of shells. `dash` is designed to be a minimal implementation of the Posix standard shell `sh`. So far, in macOS (including Catalina),`sh` invokes `bash` in `sh`-compatibility mode.

As Apple's support article mentions, Catalina also adds a new mechanism for users and admins to change which shell handles `sh` invocations. MacAdmins or users can change the symbolic link stored in `/var/select/sh` to point to a shell other than `/bin/bash`. This changes which shell interprets scripts the `#!/bin/sh` shebang or scripts invoked with `sh -c`. Changing the interpreter for `sh` should not, but may change the behavior of several crucial scripts in the system, management tools, and in installers, but may be very useful for testing purposes.

All of these changes are indicators that Apple is preparing to remove `/bin/bash` at some, yet indeterminate, time in the future.

Do I need to wait for Catalina to switch to zsh?

No, `zsh` is available Mojave and on older macOS versions. You can start testing `zsh` or even switch your default shell already.

If you want to just see how `zsh` works, you can just open Terminal and type `zsh`:

```
$ zsh
MacBook%
```

The main change you will see is that the prompt looks different. `zsh` uses the `%` character as the default prompt. (You can change that, of course.) Most navigation keystrokes and other behaviors will remain the same as in `bash`.

If you want to already switch your default shell to `zsh` you can use the `chsh` command:

```
$ chsh -s /bin/zsh
```

This will prompt for your password. This command will not change the current shell, but all new ones, so close the current Terminal windows and tabs and open a new one.

How is zsh different?

Like `bash` ('Bourne again shell'), `zsh`derives from the 'Bourne' family of shells. Because of this common ancestry, it behaves very similar in day-to-day use. The most obvious change will be the different prompt.

The main difference between `bash` and `zsh` is configuration. Since `zsh` ignores the `bash` configuration files (`.bash_profile` or `.bashrc`) you cannot simply copy customized bash settings over to `zsh`. `zsh` has much more options and points to change `zsh` configuration and behavior. There is an entire eco-system of configuration tools and themes called `oh-my-zsh` which is very popular.

`zsh` also offers better configuration for auto-completion which is far easier than in `bash`.

I am planning a separate post, describing how to transfer (and translate) your configurations from `bash` to `zsh`.

What about scripting?

Since `zsh` has been present on macOS for a long time, you could start moving your scripts from `bash` to `zsh` right away and not lose backwards compatibility. Just remember to set the shebang in your scripts to `#!/bin/zsh`.

You will gain some features where `zsh` is superior to `bash` v3, such as arrays and associative arrays (dictionaries).

There is one exception where I would now recommend to use `/bin/sh` for your scripts: the Recovery system does *not* contain the `/bin/zsh` shell, even on the Catalina beta. This could still change during the beta phase, or even later, but then you still have to consider *older* macOS installations where `zsh` is definitely not present in Recovery.

When you plan to use your scripts or pkgs with installation scripts in a Recovery (or NetInstall, or bootable USB drive) context, such as Twocanoes MDS, installr or bootstrappr, then you *cannot* rely on `/bin/zsh`.

Since we now know that `bash` is eventually going away, the only common choice left is `/bin/sh`.

When you build an installer package, it can be difficult to anticipate all the contexts in which it might be deployed. So, for installation pre- and postinstall scripts, I would recommend using `/bin/sh` as the shebang from now on.

I used to recommend using `/bin/bash` for everything MacAdmin related. `/bin/sh` is definitely a step down in functionality, but it seems like the safest choice for continued support.

Summary

Overall, while the messaging from Apple is very interesting, the change itself is less dramatic than the headlines. Apple is not 'replacing' `bash` with `zsh`, at least not yet. Overall, we will have to re-think and re-learn a few things, but there is also much to be gained by finally switching from a ten-year-old shell to a new modern one!

This git repo has been shared by many on MacAdmins Slack: rothgar/mastering-zsh, I will certainly dive into that and share about my experiences here!

Moving to zsh, part 2: Configuration Files

Apple has announced that in macOS 10.15 Catalina the default shell will be `zsh`.

In this series, I will document my experiences moving `bash` settings, configurations, and scripts over to `zsh`.

> *This series has grown into a book: reworked and expanded with more detail and topics. Like my other books, I plan to update and add to it after release as well, keeping it relevant and useful. You can order it on the Apple Books Store now.*

In part one I talked about Apple's motivation to switch the default shell and urge existing users to change to `zsh`.

Since I am new to `zsh` as well, I am planning to document my process of transferring my personal `bash` setup and learning the odds and ends of `zsh`.

Many websites and tutorials leap straight to projects like oh-my-zsh or prezto where you can choose from hundreds of pre-customized and pre-configured themes.

While these projects are very impressive and certainly show off the flexibility and power of `zsh` customization, I feel this will actually prevent an understanding of how `zsh` works and how it differs from `bash`. So, I am planning to build my own configuration 'by hand' first.

At first, I actually took a look at my current `bash_profile` and cleaned it up. There were many aliases and functions which I do not use or broke in some macOS update. I the end, this is what I want to re-create in `zsh`:

- aliases
  - mostly shortcuts to `open` files with a specific application
- functions
  - show man pages in a dedicated Terminal window
  - some more simple functions
  - get the frontmost Finder window path
- shell settings
  - case-insensitive globbing
  - case-insensitive path-completion (for `bash` this is set in `.inputrc`)
  - command history, shared across windows and sessions
  - use BBEdit as the editor
- prompt:
  - show current working dir
  - show a colored symbol showing the last command's exit code
  - update the Terminal window title bar to show the cwd

Most of these should be fairly easy to transfer. Some might be… interesting.

But first, where do we put our custom `zsh` configuration?

zsh Configuration Files

`bash` has a list of possible files that it tries in predefined order. I have the description in my post on the `bash_profile`.

`zsh` also has a list of files it will execute at shell startup. The list of possible files is even longer, but somewhat more ordered.

all users	user	login shell	interactive shell	scripts	Terminal.app
`/etc/zshenv`	`.zshenv`	√	√	√	√
`/etc/zprofile`	`.zprofile`	√	x	x	√
`/etc/zshrc`	`.zshrc`	√	√	x	√

all users	user	login shell	interactive shell	scripts	Terminal.app
`/etc/zlogin`	`.zlogin`	√	x	x	√
`/etc/zlogout`	`.zlogout`	√	x	x	√

The files in `/etc/` will be launched (when present) for all users. The `.z*` files only for the individual user.

By default, `zsh` will look in the root of the home directory for the user `.z*` files, but this behavior can be changed by setting the `ZDOTDIR` environment variable to another directory (e.g. `~/.zsh/`) where you can then group all user `zsh` configuration in one place.

On macOS you could set the `ZDOTDIR` to `~/Documents/zsh/` and then use iCloud syncing (or a different file sync service) to have the same files on all your Macs. (I prefer to use `git`.)

`bash` will either use `.bash_profile` for login shells, or `.bashrc` for interactive shells. That means, when you want to centralize configuration for all use cases, you need to `source` your `.bashrc` from `.bash_profile` or vice versa.

`zsh` behaves differently. `zsh` will run *all* of these files in the appropriate context (login shell, interactive shell) when they exist.

`zsh` will start with `/etc/zshenv`, then the user's `.zshenv`. The `zshenv` files are *always* used when they exist, *even for scripts* with the `#!/bin/zsh` shebang. Since changes applied in the `zshenv` will affect `zsh` behavior in *all* contexts, you should you should be very cautious about changes applied here.

Next, when the shell is a login shell, `zsh` will run `/etc/zprofile` and `.zprofile`. Then for interactive shells (and login shells) `/etc/zshrc` and `.zshrc`. Then, again, for login shells `/etc/zlogin` and `.zlogin`. Why are there two files for login shells? The `zprofile` exists as an analog for `bash`'s and `sh`'s profile files, and `zlogin` as an analog for `ksh` login files.

Finally, there are `zlogout` files that can be used for cleanup, when a login shell exits. In this case, the user level `.zlogout` is read first, then the central `/etc/zlogout`. If the shell is terminated by an external process, these files might not be run.

Apple Provided Configuration Files

macOS Mojave (and earlier versions) includes `/etc/zprofile` and `/etc/zshrc` files. Both are very basic.

`/etc/zprofile` uses `/usr/libexec/path_helper` to set the default `PATH`. Then `/etc/zshrc` enables UTF–8 with `setopt combiningchars`.

Like `/etc/bashrc` there is a line in `/etc/zshrc` that would load `/etc/zshrc_Apple_Terminal` if it existed. This is interesting as `/etc/bashrc_Apple_Terminal` contains quite a lot of code to help `bash` to communicate with the Terminal application. In particular `bash` will send a signal to the Terminal on every new prompt to update the path and icon displayed in the Terminal window title bar, and provides other code relevant for saving and restoring Terminal sessions between application restarts.

However, there is no `/etc/zshrc_Apple_Terminal` and we will have to provide some of this functionality ourselves.

> *Note: As of this writing, `/etc/zshrc` in the macOS Catalina beta is different from the Mojave `/etc/zshrc` and provides more configuration. However, since Catalina is still beta, I will focus these articles on Mojave and earlier. Once Catalina is released, I may update these articles or write a new one for Catalina, if necessary.*

Which File to use?

When you want to use the `ZDOTDIR` variable to change the location of the other `zsh` configuration files, setting that variable in `~/.zshenv` seems like a good choice. Other than that, you probably want to *avoid* using the `zshenv` files, since it will change settings for *all* invocations of `zsh`, including scripts.

macOS Terminal considers every new shell to be a login shell *and* an interactive shell. So, in Terminal a new `zsh` will potentially run *all* configuration files.

For simplicity's sake, you should use just one file. The common choice is `.zshrc`.

Most tools you can download to configure `zsh`, such as 'prezto' or 'oh-my-zsh', will override or re-configure your `.zshrc`. You could consider moving your code to `.zlogin` instead. Since `.zlogin` is sourced *after* `.zshrc` it can override settings from `.zshrc`. However, `.zlogin` is *only* called for login shells.

The most common situation where you do *not* get a login shell with macOS Terminal, is when you switch to `zsh` from another shell by typing the `zsh` command.

I would recommend to put your configuration in *your* `.zshrc` file and if you want to use any of the theme projects, read and follow their instructions closely as to how you can preserve your configurations together with theirs.

Managing the shell for Administrators

MacAdmins may have the need to manage certain shell settings for their users, usually environment variables to configure certain command line tool's behaviors.

The most common need is to expand the `PATH` environment variable for third party tools. Often the third party tools in question will have elaborate postinstall scripts that attempt to modify the current user's `.bash_profile` or `.bashrc`. Sometimes, these tools even consider that a user might have changed the default shell to something other than `bash`.

On macOS, system wide changes to the `PATH` should be done by adding files to `/etc/paths.d`.

As an administrator you should be on the lookout for scripts and installers that attempt to modify configuration files on the user level, disable the scripts during deployment, and manage the required changes centrally. This will allow you to keep control of the settings even as tools change, are added or removed from the system, while preserving the user's custom configurations.

To manage environment variables other than `PATH` centrally, administrators should consider `/etc/zshenv` or adding to the existing `/etc/zshrc`. In these cases you should always monitor whether updates to macOS overwrite or change these files with new, modified files of their own.

Summary

There are many possible files where the `zsh` can load user configuration. You should use `~/.zshrc` for your personal configurations.

There are many tools and projects out there that will configure `zsh` for you. This is fine, but might keep you from really understanding how things work.

MacAdmins who need to manage these settings centrally, should use `/etc/paths.d` and similar technologies or consider `/etc/zshenv` or `/etc/zshrc`.

Apple's built-in support for `zsh` in Terminal is not as detailed as it is for `bash`.

Next: Part 3 – Shell Options

Moving to zsh, part 3: Shell Options

Apple has announced that in macOS 10.15 Catalina the default shell will be `zsh`.

In this series, I will document my experiences moving `bash` settings, configurations, and scripts over to `zsh`.

> *This series has grown into a book: reworked and expanded with more detail and topics. Like my other books, I plan to update and add to it after release as well, keeping it relevant and useful. You can order it on the Apple Books Store now.*

Now that we have chosen a file to configure our `zsh`, we need to decide on ‘what’ to configure and ‘how.’ In this post, I want to talk about `zsh`’s shell options.

As I have mentioned in the earlier posts, I am aware that there are many solutions out there that give you a pre-configured ‘shortcut’ into lots of `zsh` goodness. But I am interested in learning this the ‘hard way’ without shortcuts. Call me old-fashioned. (“Uphill! In the snow! Both ways!”)

In the previous post, I listed some features that I would like to transfer from my `bash` configuration. While researching how to implement these options in `zsh`, I found a few, new and interesting options in `zsh`.

The settings from `bash` which I *want* in `zsh` were:

- case-insensitive globbing
- command history, shared across windows and sessions

> *Note: `bash` in this series of posts specifically refers to the version of `bash` that comes with macOS as `/bin/bash` (v3.2.57).*
>
> *Note 2: Mono-typed lines starting with a `%` show commands and results from `zsh`. Mono-typed lines starting with `$` show commands and results in `bash`*

What are Shell Options?

Shell options are preferences for the shell's behavior. You are using shell options in `bash`, when you enable 'trace mode' for scripts with the `set -x` command or the `bash -x` option. (Note: this also works with `zsh` scripts.)

`zsh` has *a lot* of shell options. Many of these options serve the purpose of enabling (or disabling) compatibility with other shells. There are also many options which are specific to `zsh`.

You can set an option with the `setopt` command. For compatibility with other shells the `setopt` command and `set -o` have the same effect (set an option by name). The following commands set the same option:

```
set -o AUTO_CD
setopt AUTO_CD
```

The names or labels of the options are commonly written in all capitals in the documentation but in lowercase when listed with the `setopt` tool. The labels of the options are case insensitive and any underscores in the label are ignored. So, these commands set the same option:

```
setopt AUTO_CD
setopt autocd
setopt auto_cd
setopt autoCD
```

There are quite a few ways to negate or unset an option. First you can use `unsetopt` or `set +o`. Alternatively, you can prefix with `NO` or `no` to negate an option. The following commands all have the same effect of turning *off* the previously set option `AUTO_CD`

```
unsetopt AUTO_CD
set +o AUTO_CD
unsetopt autocd
setopt NO_AUTO_CD
setopt noautocd
```

Any options you change will only take effect in the current instance of `zsh`. When you want to change the settings for all new shells, you have to put the commands in one of the configuration files (usually `.zshrc`).

Showing the current Options

You can list the existing shell options with the `setopt` command:

```
% setopt
combiningchars
interactive
login
monitor
shinstdin
zle
```

This list only shows options are changed from the default set of options for `zsh`. These options are marked with `<D>` (default for all shell emulations) or `<Z>` (default for `zsh`) in the documentation or the `zshoptions` man page.

You can also get a list of all default `zsh` options with the command:

```
% emulate -lLR zsh
```

Some zsh Options I use

As I have mentioned before in my posts on `bash` configuration, I prefer minimal configuration changes, so I do not feel all awkward and lost when I have to work on an 'un-configured' Mac.

These configurations are a personal choice and you should pick and choose your own. You can find a full list of `zsh` options in the `zsh` Manual or with `man zshoptions`.

On the other hand, exploring the options allows us to explore a few useful `zsh` features.

Case Insensitive Globbing

Note: 'Globbing' is a unix/shell term that refers to the expansion of wildcard characters, such as `*` and `?` into full file paths and names. I.e. `~/D*` is expanded into `/Users/armin/Desktop /Users/armin/Documents /Users/armin/Downloads`

Since the file system on macOS is (usually) case-insensitive, I prefer globbing and tab-completion to be case-insensitive as well.

The `zsh` option which controls this is `CASE_GLOB`. Since we want globbing to be *case-insensitive*, we want to turn the option off, so:

```
setopt NO_CASE_GLOB
```

You can test this in the shell:

```
% ls ~/d*<tab>
```

In `zsh` tab completion will replace the wildcard with the actual result. So after the tab you will see:

```
% ls /Users/armin/Desktop /Users/armin/Documents
/Users/armin/Downloads
```

Using tab completion this way to see and possibly edit the actual replacement for wildcards is a useful safety net.

In `bash` hit the tab key will list possible completions, but not substitute them in the command prompt.

If you *do not* like this behavior in `zsh` then you can change to behavior similar to `bash` with:

```
setopt GLOB_COMPLETE
```

Automatic CD

Sometimes you enter the path to a directory, but forget the leading `cd`:

```
$ Library/Preferences/
bash: Library/Preferences/: is a directory

% Library/Preferences
zsh: permission denied: Library/Preferences
```

With `AUTO_CD` enabled in `zsh`, the shell will automatically change directory:

```
% Library/Preferences
% pwd
/Users/armin/Library/Preferences
```

This works with relative *and* absolute paths, including the `..`:

```
% ..
% pwd
/Users/armin/Library
% ../Desktop
% pwd
/Users/armin/Desktop
```

I have an `alias` in my `.bash_profile` that sets the `..` command to `cd ..`. Auto CD replaces that functionality and more.

Enable Auto CD with:

```
setopt AUTO_CD
```

Shell History

Shells commonly remember previously executed commands and allows you to recall them with the up and down arrow keys, search or special history commands.

Most of those keys work the same in zsh. However, there are a few things you need to configure for zsh history to work as you are used to with bash on macOS.

By default, zsh does *not* save its history when the shell exits. The history is 'forgotten' when you close a Terminal window or tab. To make zsh save its history to a file when it exits, you need to set a variable in the shell:

```
HISTFILE=${ZDOTDIR:-$HOME}/.zsh_history
```

Note: this is not a shell option but shell variable or parameter. I will cover some more of those later, You can find a list of variables used by zsh in the documentation.

The HISTFILE variable tells zsh where to store the history data. The syntax ${ZDOTDIR:-$HOME} means it will use the value of ZDOTDIR when it is set or default to the value of HOME otherwise. When a user has set the ZDOTDIR variable to group their configurations files in a specific directory, the history will be stored there as well.

By default zsh simply writes each command in its own line in the history file. You can view the file's contents with any text editor or list the last few commands:

```
% tail -n 10 ~/.zsh_history
```

You can make zsh add a bit more data (timestamp in unix epoch time and elapsed time of the command) by setting the EXTENDED_HISTORY shell option.

```
setopt EXTENDED_HISTORY
```

You can set limits on how many commands the shell should remember in the session and in the history file with the HISTSIZE and SAVEHIST variables:

```
SAVEHIST=5000
HISTSIZE=2000
```

When the shell reaches this limit the oldest commands will be removed from memory or the history file.

By default, when you exit `zsh` (for example, by closing the window or tab) this particular instance of `zsh` will *overwrite* an existing history file with its history. So when you have multiple Terminal windows or tabs open, they will all overwrite each others' histories eventually.

You can tell `zsh` to use a single, shared history file across the sessions and append to it rather than overwrite:

```
# share history across multiple zsh sessions
setopt SHARE_HISTORY
# append to history
setopt APPEND_HISTORY
```

Furthermore, you can tell `zsh` to update the history file after every command, rather than waiting for the shell to exit:

```
# adds commands as they are typed, not at shell exit
setopt INC_APPEND_HISTORY
```

When you use a shared history file, it will grow very quickly, and you may want to use some options to clean out duplicates and blanks:

```
# expire duplicates first
setopt HIST_EXPIRE_DUPS_FIRST
# do not store duplications
setopt HIST_IGNORE_DUPS
#ignore duplicates when searching
setopt HIST_FIND_NO_DUPS
# removes blank lines from history
setopt HIST_REDUCE_BLANKS
```

(some of these are redundant)

Most of the time you will access the history with the up arrow key to recall the last command, or maybe a few more steps. You can search through the history with `ctrl-R`

In `zsh`, you can also use the `!!` history substitution, which will be replaced with the entire last command. This is most commonly used in combination with `sudo`:

```
% systemsetup -getRemoteLogin
You need administrator access to run this tool... exiting!
% sudo !!
sudo systemsetup -getRemoteLogin
Password:
Remote Login: On
```

By default, the shell will show the command it is substituting before it is run. But at that point, it is too late to make any changes. When you set the `HIST_VERIFY` option, `zsh` will show the substituted command in the prompt instead, giving you a chance to edit or cancel it, or just confirm it.

```
% systemsetup -getRemoteLogin
You need administrator access to run this tool... exiting!
% sudo !!
% sudo systemsetup -getRemoteLogin
Password:
Remote Login: On
```

This works for other history substitutions such as `!$` or `!*`, as well. You can find all of `zsh`'s history expansions in the documentation.

Correction

When you mistype a command or path, the shell is usually unforgiving. In `zsh` you can enable correction. Then, the shell will make a guess of what you meant to type and ask whether you want do that instead:

```
% systemprofiler
zsh: correct 'systemprofiler' to 'system_profiler' [nyae]?
```

Your options are to

- `n`: execute as typed
- `y`: accept and execute the suggested correction
- `a`: abort and do nothing
- `e`: return to the prompt to continue editing

I have found this far less annoying and far more useful than I expected. Especially, since it works together with `AUTO_CD`:

```
% Dekstop
zsh: correct 'Dekstop' to 'Desktop' [nyae]?
```

You enable `zsh` correction with these options:

```
setopt CORRECT
setopt CORRECT_ALL
```

Reverting to defaults

Most of the changes mentioned here affect the interactive shell and will have little impact on `zsh` scripts. However, there are some options that do affect the behavior of things like variable substitutions which will affect scripts.

You can revert the options for the current shell to the default settings with the following command:

```
emulate -LR zsh
```

We encountered this command earlier when we listed the default settings. The `-l` option will list the settings rather than apply them.

If in doubt, it may be useful to add this at the beginning of your `zsh` scripts.

Next

In the next part we will take a look at aliases and functions.

Moving to zsh, part 4: Aliases and Functions

Apple has announced that in macOS 10.15 Catalina the default shell will be `zsh`.

In this series, I will document my experiences moving `bash` settings, configurations, and scripts over to `zsh`.

- Part 1: Moving to zsh
- Part 2: Configuration Files
- Part 3: Shell Options
- Part 4: Aliases and Functions *(this article)*
- Part 5: Completions
- Part 6: Customizing the `zsh` Prompt
- Part 7: Miscellanea
- Part 8: Scripting `zsh`

> *This series has grown into a book: reworked and expanded with more detail and topics. Like my other books, I plan to update and add to it after release as well, keeping it relevant and useful. You can order it on the Apple Books Store now.*

As I have mentioned in the earlier posts, I am aware that there are many solutions out there that give you a pre-configured 'shortcut' into lots of `zsh` goodness. But I am interested in learning this the 'hard way' without shortcuts. Call me old-fashioned. ("Uphill! In the snow! Both ways!")

Aliases

Aliases in `zsh` work just like aliases in `bash`. You declare an alias with the `alias` (built-in) command and it will work as a text replacement at the beginning of the command prompt:

```
alias ll='ls -al'
```

You can just copy your alias declarations from your `.bash_profile` or `.bashrc` to your `.zshrc`. I had aliases for `..` and `cd..` which are now handled by Auto CD and shell correction respectively, so I didn't bother to move those. (part 3: 'Shell Options')

After the alias is declared, you can use it *at the beginning of a command*. When you try to use the alias anywhere else in the command, the alias will not work:

```
% sudo ll
sudo: ll: command not found
```

Global Aliases

This is where `zsh` has an advantage. You can declare an alias as a 'global' alias, and then will be replaced *anywhere* in the command line:

```
% alias -g badge='tput bel'
% sudo badge        #<beeps> with privilege
```

Identifying Aliases

There is one more feature of `zsh` that is useful with aliases. The `which` command will show if a command stems from an alias substitution:

```
% which ll
ll: aliased to ls -l
```

However, when you try this with global aliases, the substitution occurs *before* the `which` command can evaluate the alias, which leads to an unexpected result:

```
% which badge
/usr/bin/tput
bel not found
```

You can suppress the alias substitution by escaping the first character or by quoting the entire alias name:

```
% which \badge
badge: globally aliased to tput bel
% which 'badge'
badge: globally aliased to tput bel
```

Functions

As with aliases, functions in your `zsh` configuration will work just as they did in `bash`.

```
function vnc() {
    open vnc://"$USER"@"$1"
}
```

This code in your `zsh` configuration file will define the `vnc` function and make it available in the shell.

Autoload Functions

However, `zsh` has some features which make using functions more flexible. There is (once again) a bit of configuration required to get this working.

Instead of declaring the function directly the configuration file, you can put the function in a separate file. `zsh` has a built-in variable called `fpath` which is an array of paths where `zsh` will look for files defining a function. You can add your own directory to this search path:

```
fpath+=~/Projects/dotfiles/zshfunctions
```

Just having a file in the directory is not enough. You still have to tell `zsh` that you want to use this particular function:

```
autoload vnc
```

This command tells `zsh`: "'Declare a function named `vnc`. To execute it, load a file named `vnc`, it is somewhere in the `fpath`."

Note: you often see the `-U` or `-Uz` option added to the `autoload` command. These options help avoid conflicts with your personal settings. They suppress alias substitution and `ksh`-style loading of functions, respectively.

The `vnc` file in my `zshfunctions` directory can look like this:

```
# uses the arguments as hostnames for `open vnc://` (Screen Sharing)
# uses the $USER username as default account name

for x in $@; do
    open vnc://"$USER"@"$x"
done
```

The `vnc` function will open a Screen Sharing session with the current user name pre-filled in.

Initializing Autoload Functions

You could also put the code in the function file into a `function` block:

```
function vnc() {
    for x in $@; do
        open vnc://"$vnc_user"@"$x"
    done
}

# initialization code
vnc_user="remote_admin"
alias screen_sharing='vnc'
```

The function name should match the function name declared with `autoload`.

When you have additional code *outside* the function, the `autoload` behavior changes. When the function is called for the first time, the function will be defined and the code outside the function will be run. The function itself will *not* be executed on the first run. On subsequent calls, the function will be executed and the code outside the function is ignored.

You can use this to provide setup and initialization code for the function. You can even have more functions defined in the function file. The above example declares and sets a variable to use for account name and an alias for the `vnc` command.

Since you have to run the function once for the initialization, you often see this syntax in the `zsh` configuration file:

```
autoload vnc && vnc
```

Which means ‘declare the function and if that succeeds run it.’

In some functions, the initialization code will already launch the function itself:

```
function vnc() {
    ...
}
```

```
# initialization
vnc_user="remote_admin"
vnc()
```

Since the behavior will vary from each `autoload`ed function to the next, be sure to study any documentation or the function's code.

Identifying Functions

Finally, the `which` command will show the function code:

```
 % which vnc
vnc () {
    for x in $@
    do
        open vnc://"$USER"@"$x"
    done
}
```

The `functions` command without any parameters, will print *all* functions (there will be a lot of them). Use `functions +` to just list the function names.

Debugging Functions

When you are working on complex autoloaded functions, you will at some point have to do some debugging. You can enable tracing for functions with

```
% functions -t vnc
% vnc Client.local
+vnc:1> x=Client.local
+vnc:2> open vnc://armin@Client.local
```

You can disable tracing for this function with `functions +t vnc`.

Next

In the next part we will enable, use and configure tab completions.

Moving to zsh, part 5: Completions

Apple has announced that in macOS 10.15 Catalina the default shell will be `zsh`.

In this series, I will document my experiences moving `bash` settings, configurations, and scripts over to `zsh`.

- Part 1: Moving to zsh
- Part 2: Configuration Files
- Part 3: Shell Options
- Part 4: Aliases and Functions
- Part 5: Completions *(this article)*
- Part 6: Customizing the `zsh` Prompt
- Part 7: Miscellanea
- Part 8: Scripting `zsh`

> *I am preparing a book on this topic, reworked and expanded with more detail and topics. Like my other books, I plan to update and add to it after release as well, keeping it relevant and useful. You can pre-order it on the Apple Books Store now.*

As I have mentioned in the earlier posts, I am aware that there are many solutions out there that give you a pre-configured 'shortcut' into lots of `zsh` goodness. But I am interested in learning this the 'hard way' without shortcuts. Call me old-fashioned. ("Uphill! In the snow! Both ways!")

What are Completions?

Man shells use the tab key (⇥) for completion. When you press that key, the shell tries to guess what you are typing and will complete it, or if the beginning of what you typed is ambiguous, suggest from a list of possible completions.

For example when you want to `cd` to your `Documents` folder, you can save typing:

```
% cd ~/Doc⇥
% cd ~/Documents/
```

When you hit the tab key, the system will complete the path to the `Documents` folder.

When the completion is ambiguous, the shell will list possible completions:

```
% cd ~/D⇥
Desktop/    Documents/  Downloads/
```

At this point, you can add a character or two to get to a unique completion, and hit the tab key again. In `zsh` you can also hit the tab key repeatedly to cycle through the suggested completions. In this example, the first tab keystroke will show the list, the second will complete `~/Desktop/`, the third completes `~/Documents`, and so on.

You can use tab completion commands as well:

```
% system⇥
system_profiler     systemkeychain      systemsetup
systemsoundserverd  systemstats
% system_⇥
% system_profiler
```

Not having to type path and file names saves time and *avoids errors*, especially with complex paths with spaces and other special characters:

```
% cd ~/Li⇥
% cd ~/Library/Appl⇥
% cd ~/Library/Application S⇥
Application Scripts/  Application Support/
% cd ~/Library/Application Su⇥
% cd ~/Library/Application Support/
```

Using tab completion is a huge productivity boost when using a shell.

Turning It On

In the default configuration, tab completion in `zsh` is very basic. It will complete commands and paths, but not much else. But you can enable a very powerful, and useful completion system.

`zsh` comes with a tool you can use to setup this completion system. When you run the `compinstall` command it will lead you through a complex and hard to understand list of menus which explains the options and will generate the code necessary to set this configuration up and add it to your `.zshrc` file or another configuration file of your choice.

Since the commands to configure the completion are quite arcane and hard to understand, this is a good way to get something to start out with. I will explain some of these options and commands in detail.

Whether you use `compinstall or not, to turn on the more powerful completion system, you need to add at least this command to your`zsh` configuration file:

```
autoload -Uz compinit && compinit
```

This will initialize the `zsh` completion system. The details of this system are documented here.

If you want to configure the system, the configuration commands (usually `zstyle` commands) should be added to the `zsh` configuration file *before* you enable the system. (This only matters for a few configurations, but as a general rule it is safer.)

All of these completion rules need to be loaded and prepared. `zsh`'s completion system creates a cache in the file `~/.zcompdump`. The first time you run `compinit` it might take a noticeable time, but subsequent runs should use this cache and be much faster.

Sometimes, especially when building and debugging your own completion files, you may need to delete this file to force a rebuild:

```
% rm -f ~/.zcompdump
% compinit
```

Case Insensitive Completion

Since the macOS file systems are usually case-insensitive, I prefer my tab-completion to be case-insensitive as well. For `bash` you configure that in the `~/.inputrc`. In `zsh` you modify the completion systems behavior with this (monstrous) command:

```
# case insensitive path-completion
zstyle ':completion:*' matcher-list 'm:{[:lower:][:upper:]}={[:upper:]
[:lower:]}' 'm:{[:lower:][:upper:]}={[:upper:][:lower:]} l:|=* r:|=*' 'm:
{[:lower:][:upper:]}={[:upper:][:lower:]} l:|=* r:|=*' 'm:{[:lower:]
[:upper:]}={[:upper:][:lower:]} l:|=* r:|=*'
```

I have seen many varieties for this configuration in different websites, but this is what `compinstall` adds when I select case-insensitive completion, so I am going with that.

Partial Completion

This is a particularly nice feature. You can type fragments of each path segment and the completion will try to complete them all at once:

```
% cd /u/lo/b⇥
% cd /usr/local/bin
```

```
% cd ~/L/P/B⇥
% ~/Library/Preferences/ByHost/
```

If the fragments are ambiguous, there are different strategies to what the completion system suggests. I have configured these like this:

```
# partial completion suggestions
zstyle ':completion:*' list-suffixes
zstyle ':completion:*' expand prefix suffix
```

Commands with built-in completion

`zsh` comes with several completion definitions for many commands. For example, when you type `cp` and then hit tab, the system will correctly assume you want to complete a file path and show the suggestions from the current working directory.

However, when you type `cp -⇥` the completion can tell from the `-` that you want to add an option to the command and suggest a list of options for `cp`, with short descriptions.

```
% cp -⇥
-H  -- follow symlinks on the command line in recursive mode
-L  -- follow all symlinks in recursive mode
-P  -- do not follow symlinks in recursive mode (default)
-R  -- copy directories recursively
-X  -- don't copy extended attributes or resource forks
-a  -- archive mode, same as -RpP
-f  -- force overwriting existing file
-i  -- confirm before overwriting existing file
```

```
-n  -- don't overwrite existing file
-p  -- preserve timestamps, mode, owner, flags, ACLs, and extended attributes
-v  -- show file names as they are copied
```

As the context of command prompt you are assembling changes, you may get different completion suggestions. For example, the completion for `ssh` will suggest host names:

```
% ssh armin@⇥
```

`zsh` comes with completion definitions for many common commands. Nevertheless, it can be helpful to just hit tab, especially when wondering about options.

On macOS completions are stored in `/usr/share/zsh/5.3/functions` (replace the`5.3` with `5.7.1` in Catalina). This directory stores many functions used with `zsh` and is in the default `fpath`. All the files in that directory that start with an underscore _ contain the completion definitions command. So, the file `_cp` contains the definition for the `cp` command. (Some of the definition files contain the definitions for multiple commands.)

Completions for macOS Commands

There are even a few macOS specific command that come with the default `zsh` installation.

```
% system_profiler ⇥⇥
```

macOS High Sierra and macOS Mojave come with `zsh` 5.3, which is now nearly two years old. `zsh` 5.3 contains less macOS specific completion definitions than the current `zsh` 5.7.1 which will is the pre-installed `zsh` in macOS Catalina. Some of the completions in 5.3 have also been updated in 5.7.1.

Tool	zsh 5.3	zsh 5.7.1
`caffeinate`		√
`defaults`	√	√
`fink`	√	√
`fs_usage`		√

Tool	zsh 5.3	zsh 5.7.1
`hdiutil`	√	√
`mdfind`		√
`mdls`		√
`mdutil`		√
`networksetup`		√
`nvram`		√
`open`	√	√
`osascript`		√
`otool`		√
`pbcopy/pbpaste`		√
`plutil`		√
`say`		√
`sc_usage`		√
`scselect`		√
`scutil`		√
`softwareupdate`	√	√
`sw_vers`		√
`swift`		√
`system_profiler`	√	√
`xcode_select`		√

Load bash completions

Since the default shell on macOS has been `bash` for so long, there are quite a few `bash` completion definitions for macOS commands and third party tools available. For example Tony Williams' `bash` completion for `autopkg` (post, Github).

You do *not* have to rewrite these completions, since the `zsh` completion system can use `bash` completion scripts as well: (add this to your `zsh` configuration file)

```
# load bashcompinit for some old bash completions
autoload bashcompinit && bashcompinit

[[ -r ~/Projects/autopkg_complete/autopkg ]] && source
~/Projects/autopkg_complete/autopkg
```

When you have multiple `bash` completion scripts you want to load, you only need to load `bashcompinit` once.

Build your own completions

Once you start using completions, you will want to have them *everywhere*. While many built-in completions exists, there are still many commands that lack a good definition.

Some commands, like the `swift` command line tool, have a built-in option to generate the completion syntax. You can then store that in a file and put it in your `fpath`:

```
% swift package completion-tool generate-zsh-script >_swift
```

> *Note: in the case of `swift`, its definition will conflict with the `_openstack` definition in `zsh` 5.3. You can fix this with the command `compdef _swift swift` after loading the completion system.*

Some commands provide a list of options and arguments with the `-h`/`--help` option. If this list follows a certain syntax, you can get a decent completion working with

```
% compdef _gnu_generic <command>
```

One example on macOS, where this has decent results is the `xed` command which opens a file or folder in Xcode.

But for best results, you will often have to build the description yourself. Unfortunately this is not a simple task. The syntax is meticulously, but also quite abstractly documented in the `zsh` documentation for the Completion System. I also found the ‘howto’ documentation in the `zsh-completions` repository very useful, as well as the ‘`zsh` Completion Style Guide.’

To avoid everyone re-inventing the wheel, I have started a repository on Github for macOS specific completion files. The page has the instructions on how to install them and I will welcome pull requests with contributions. Since I am just starting to learn this as well, I am sure there are improvements that can be made on the completions I have built so far and there are several commands where you can test your skills and build a new one.

I suggest the `#zsh` channel on the MacAdmins Slack for discussion.

Next

In the next post in this series, we will discuss how to configure `zsh`’s command line prompt.

Moving to zsh, part 6 – Customizing the zsh Prompt

Apple has announced that in macOS 10.15 Catalina the default shell will be `zsh`.

In this series, I will document my experiences moving `bash` settings, configurations, and scripts over to `zsh`.

> *This series has grown into a book: reworked and expanded with more detail and topics. Like my other books, I plan to update and add to it after release as well, keeping it relevant and useful. You can order it on the Apple Books Store now.*

As I have mentioned in the earlier posts, I am aware that there are many solutions out there that give you a pre-configured 'shortcut' into lots of `zsh` goodness. But I am interested in learning this the 'hard way' without shortcuts. Call me old-fashioned. ("Uphill! In the snow! Both ways!")

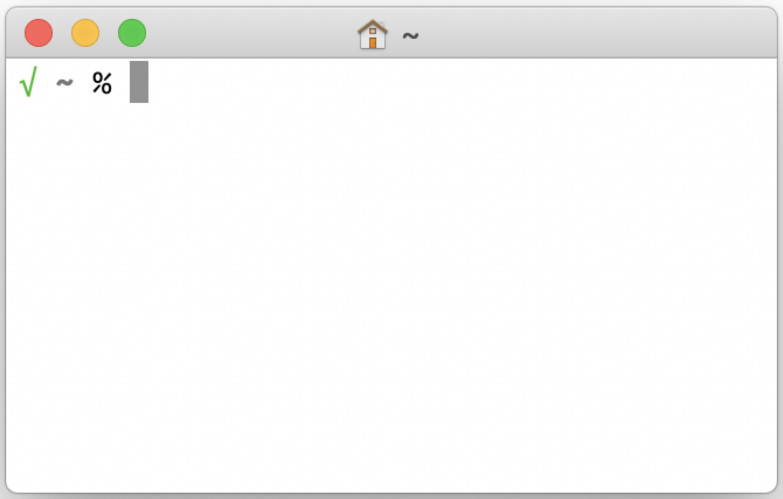
The default `bash` prompt on macOS is quite elaborate. It shows the username, the hostname, *and* the current directory.

```
Calypso:~ armin$
```

On the other hand, the default bash prompt doesn't show the previous command's exit code, a piece of information I find very useful. I have written before how I re-configured my `bash` prompt to have the information I want:

- Minimal Terminal Prompt
- Show Exit Code in your bash Prompt

Of course, I wanted to recreate the same experience in `zsh`.



```
√ ~ %
```

The only (visual) difference to my `bash` prompt is the `%` instead of the `$`.

> *Note: creating a file `~/.hushlogin` will suppress the status message at the start of each Terminal session in `zsh` as well as in `bash` (or any other shell).*

Basic Prompt Configuration

The basic `zsh` prompt configuration works similar to `bash`, even though it uses a different syntax. The different placeholders are described in detail in the `zsh` manual.

`zsh` uses the same shell variable `PS1` to store the default prompt. However, the variable names `PROMPT` and `prompt` are synonyms for `PS1` and you will see either of those three being used in various examples. I am going to use `PROMPT`.

The default prompt in `zsh` is `%m%#`. The `%m` shows the first element of the hostname, the `%#` shows a `#` when the current prompt has super-user privileges (e.g. after a `sudo -s`) and otherwise the `%` symbol (the default `zsh` prompt symbol).

The `zsh` default prompt is far shorter than the `bash` default, but even less useful. Since I work on the local system most of the time, the hostname bears no useful information, and repeating it every line is superfluous.

> *Note: you can argue that the hostname in the prompt is useful when you frequently have multiple terminal windows open to different hosts. This is true, but then the prompt is defined by the* remote *shell and its configuration files on the* remote *host. In your configuration file, you can test if the `SSH_CLIENT` variable is set and show a different prompt for remote sessions. There are more ways of showing the host in remote shell sessions, for example in the Terminal window title bar or with different window background colors.*

In our first iteration, I want to show the current working directory instead of the hostname. When you look through the list of prompt placeholders in the zsh documentation, you find `%d`, `%/`, and `%~`. The first two do exactly the same. The last substitution will display a path that starts with the user's home directory with the `~`, so it will shorten `/Users/armin/Projects/` to `~/Projects`.

> *Note: in the end you want to set your `PROMPT` variable in the `.zshrc` file, so it will take effect in all your `zsh` sessions. For testing, however, you can just change the `PROMPT` variable in the interactive shell. This will give you immediate feedback, how your current setup works.*

```
% PROMPT='%/ %# '
/Users/armin/Projects/dotfiles/zshfunctions %

% PROMPT='%~ %# '
~/Projects/dotfiles/zshfunctions %
```

Note the trailing space in the prompt string, to separate the final `%` or `#` from the command entry.

I prefer the shorter output of the `%~` option, but it can still be quite long, depending on your working directory. `zsh` has a trick for this: when you insert a number `n` between the `%` and the `~`, then only the last `n` elements of the path will be shown:

```
% PROMPT='%2~ %# '
dotfiles/zshfunctions %
```

When you do `%1~` it will show only the name of the working directory or `~` if it is the home directory. (This also works with `%/`, e.g. `%2/`.)

Adding Color

Adding a bit of color or shades of gray to the prompt can make it more readable. In `bash` you need cryptic escape codes to switch the colors. `zsh` provides an easier way. To turn the directory in the path blue, you can use:

```
PROMPT='%F{blue}%1~%f %# '
```

The `F` stands for 'Foreground color.' `zsh` understands the colors `black`, `red`, `green`, `yellow`, `blue`, `magenta`, `cyan` and `white`. `%F` or `%f` resets to the default text color. Furthermore, Terminal.app represents itself as a 256-color terminal to the shell. You can verify this with

```
% echo $TERM
xterm-256color
```

You can access the 256 color pallet with `%F{0}` through `%F{255}`. There are tables showing which number maps to which color:

- 256 Colors – Cheat Sheet – Xterm, HEX, RGB, HSL
- 256 Terminal colors and their 24bit equivalent (or similar)

So, since I want a dark gray for my current working dir in my prompt, I chose `240`, I also set it to bold with the `%B` code:

```
PROMPT='%B%F{240}%1~%f%b %# '
```

You can find a detailed list of the codes for visual effects in the documentation.

Dynamic Prompt

I wrote an entire post on how to get `bash` to show the color-coded exit code of the last command. As it turns out, this is *much* easier in `zsh`.

One of the prompt codes provides a 'ternary conditional,' which means it will show one of two expressions, depending on a condition. There are several conditions you can use. Once again the details can be found in the documentation.

There is one condition for the previous commands exit code:

```
%(?.<success expression>.<failure expression>)
```

This expression will use the `<success expression>` when the previous command exited successfully (exit code zero) and `<failure expression>` when the previous command failed (non-zero exit code). So it is quite easy to build an conditional prompt:

```
% PROMPT='%(?.√.?%?) %1~ %# '
√ ~ % false
?1 ~ %
```

You can get the √ character with option-V on the US or international macOS keyboard layout. The last part of the ternary `?%?` looks confusing. The first `?` will print a literal question mark, and the second part `%?` will be replaced with previous command's exit code.

You can add colors in the ternary expression as well:

```
PROMPT='%(?.%F{green}√.%F{red}?%?)%f %B%F{240}%1~%f%b %# '
```

Another interesting conditional code is `!` which returns whether the shell is privileged (i.e. running as root) or not. This allows us to change the default prompt symbol from `%` to something else, while maintaining the warning functionality when running as root:

```
% PROMPT='%1~ %(!.#.>) '
~ > sudo -s
~ # exit
~ >
```

Complete Prompt

Here is the complete prompt we assembled, with all the parts explained:

```
PROMPT='%(?.%F{green}√.%F{red}?%?)%f %B%F{240}%1~%f%b %# '
```

`%(?.√.?%?)`	if return code `?` is 0, show √, else show `?%?`
`%?`	exit code of previous command
`%1~`	current working dir, shortening home to `~`, show only last `1` element
`%#`	`#` with root privileges, `%` otherwise
`%B %b`	start/stop bold
`%F{...}`	text (foreground) color, see table
`%f`	reset to default textcolor

Right Sided Prompt

`zsh` also offers a right sided prompt. It uses the same placeholders as the 'normal' prompt. Use the `RPROMPT` variable to set the right side prompt:

```
% RPROMPT='%*'
√ zshfunctions %                    11:02:55
```

`zsh` will automatically hide the right prompt when the cursor reaches it when typing a long command. You can use all the other substitutions from the left side prompt, including colors and other visual markers in the right side prompt.

Git Integration

zsh includes some basic integration for version control systems. Once again there is a voluminous, but hard to understand description of it in the documentation.

I found a better, more specific example in the 'Pro git' documentation. This example will show the current branch on the right side prompt.

I have changed the example to include the repo name and the branch, and to change the color.

```
autoload -Uz vcs_info
precmd_vcs_info() { vcs_info }
precmd_functions+=( precmd_vcs_info )
setopt prompt_subst
RPROMPT=\$vcs_info_msg_0_
zstyle ':vcs_info:git:*' formats '%F{240}(%b)%r%f'
zstyle ':vcs_info:*' enable git
```

In this case `%b` and `%r` are placeholders for the VCS (version control system) system for the branch and the repository name.

There are `git` prompt solutions other than the built-in module, which deliver more information. There is a script in the `git` repository, and many of the larger `zsh` theme projects, such as 'oh-my-zsh' and 'prezto' have all kinds of git status widgets or modules or themes or what ever they call them.

Summary

You can spend (or waste) a lot of time on fine-tuning your prompt. Whether these modifications really improve your productivity is a matter of opinion.

In the next post, we will cover some miscellaneous odds and ends that haven't yet really fit into any of preceding posts.

Moving to zsh – part 7: Miscellanea

Apple has announced that in macOS 10.15 Catalina the default shell will be `zsh`.

In this series, I will document my experiences moving `bash` settings, configurations, and scripts over to `zsh`.

- Part 1: Moving to zsh
- Part 2: Configuration Files
- Part 3: Shell Options
- Part 4: Aliases and Functions
- Part 5: Completions
- Part 6: Customizing the `zsh` Prompt
- Part 7: Miscellanea (*this article*)
- Part 8: Scripting `zsh`

> *This series has grown into a book: reworked and expanded with more detail and topics. Like my other books, I plan to update and add to it after release as well, keeping it relevant and useful. You can order it on the Apple Books Store now.*

As I have mentioned in the earlier posts, I am aware that there are many solutions out there that give you a pre-configured 'shortcut' into lots of `zsh` goodness. But I am interested in learning this the 'hard way' without shortcuts. Call me old-fashioned. ("Uphill! In the snow! Both ways!")

We have covered the general aspects of configuring your `zsh` environment and enabling some of its features to make your work more productive. However, there are `zsh` features that didn't quite fit in earlier posts, but also don't warrant a post of their own. So I am gathering them here.

multiIO

Terminal commands can take input from a file or a previous command (`stdin`) and have two different outputs: `stdout` and `stderr`. In `bash` you can redirect each of these to a single other destination.

For example, you can redirect the output of a command to a file:

```
% system_profiler SPHardwareDataType >hardwareinfo.txt
```

In `zsh` you can redirect to (and from) multiple sources. In the simplest form you can write the output to two files:

```
% system_profiler SPHardwareDataType >hardwareinfo.txt
>computerinfo.txt
```

This is of course not a very realistic case. Since the pipe `|` is a form of re-direction, you can combine output to a file with a pipe:

```
% system_profiler SPHardwareDataType >hardwareprofile.txt | cat
```

Instead of piping to cat, you can also redirect to `stdout` (or `&1`) *as well as* to a file:

```
% system_profiler SPHardwareDataType >&1 >hardwareprofile.txt
```

Note that the order of doing this is important. The construct `>file.txt >&1` would redirect the output to `file.txt` and then redirect the output *again* to where `stdout` or `1` is going, so it would be redundant.

When combined with pipes and other commands multiIO can become very useful:

```
% system_profiler SPHardwareDataType >hardwareprofile.txt | awk
'/Serial Number/ { print $4 }' >&1 >serialnumber.txt
```

You can use multiIO for input as well:

```
% sort </usr/share/calendar/calendar.freebsd
</usr/share/calendar/calendar.computer
```

And while this not directly related, but somewhat close, in `zsh`, this

```
% <hardwareinfo.txt
```

is equivalent to `more hardwareinfo.txt`.

Recursive Globbing with **

You can use the `**` to denote an arbitrary string that can span multiple directories in a path.

For example:

```
% echo Library/Preferences/**/com.apple.screensaver.*plist
Library/Preferences/ByHost/com.apple.screensaver.BBCCDDEE-AABB-
CCDD-ABCD-00AABBCCDDEE.plist
Library/Preferences/com.apple.screensaver.plist
```

In this case the `**` matches nothing as well as `/ByHost/`.

Note: when used on large folder structures this glob can take a while. So use with care.

Connected array Variables

We already encountered the `fpath` variable in earlier posts. You can see its contents with the echo command:

```
% echo $fpath
/Users/armin/Projects/mac-zsh-completions/completions/
/Users/armin/Projects/dotfiles/zshfunctions
/usr/local/share/zsh/site-functions /usr/share/zsh/site-functions
/usr/share/zsh/5.3/functions
```

Interestingly enough, `zsh` also has an `FPATH` variable, which is a colon-separated list of directories:

```
% echo $FPATH
/Users/armin/Projects/mac-zsh-
completions/completions/:/Users/armin/Projects/dotfiles/zshfuncti
```

```
ons:/usr/local/share/zsh/site-functions:/usr/share/zsh/site-
functions:/usr/share/zsh/5.3/functions
```

Since the `fpath` variable is an array, I only changed the `fpath` variable in my `zshrc`.I never set or changed the `FPATH`, yet it reflects the changes made to the `fpath` variable.

When you see the type of both variables, you get an idea that something is going on:

```
% echo ${(t)fpath}
array-special
% echo ${(t)FPATH}
scalar-special
```

The `fpath` and `FPATH` are connected in `zsh`. Changes to one affect the other. This allows use of more flexible and powerful array operations through the `fpath` 'aspect' of the value, but also provides compatibility to tools that expect the traditional colon-separated format in `FPATH`.

You will not be surprised to hear that `zsh` uses the same 'magic' with the `PATH` variable and its array counterpart `path`.

This means that you can continue to use `path_helper` to get your `PATH` from the files in `/etc/paths` and `/etc/paths.d`. (Well, you don't have to, because on macOS this is done for all users in `/etc/zprofile`.) But then you can manipulate the `path` variable with array functions, like:

```
path+=~/bin
```

You get the useful aspects of both syntaxes.

Suffix Aliases

I learnt this one *after* writing the aliases part.

Suffix aliases take effect on the last part of a path, so usually the file extension. A suffix alias will assign a command to use when you just type a file path in the command line.

For example, you can a suffix alias for the `txt` file extension:

```
alias -s txt="open -t"
```

When you then type a path ending with `.txt` and no command, `zsh` will execute `open -t /path/to/file.txt`.

The `open -t` command opens a file in the default application set for the `txt` file extension in Finder. You probably want to set the suffix alias to `bbedit` or `atom` or something like that rather than `open -t`.

You can use other command line tools for the suffix alias:

```
alias -s log="tail -f"
```

Then, typing `/var/log/install.log` will show the last lines of that file and update the output when the file changes. If you prefer the graphical user interface, you can use the `open -a` command to assign suffix aliases to applications:

```
alias -s log="open -a Console"
```

You can even create a suffix alias using a different alias:

```
alias pacifist="open -a Pacifist"
alias -s pkg=pacifist
```

Together with the AutoCD option, this can improve your application-shell interactions a lot.

Bindkey for History Search

Most of the keyboard shortcuts in `zsh` work the same way as they do in `bash`. I have found one change that has proven quite useful:

```
^[[A' up-line-or-search # up arrow bindkey

^[[B' down-line-or-search # down arrow
```

These two commands will change the behavior of the up and down arrow keys from just switching to the previous command, to `searching`. This means that when you start typing a command and then hit the up key, rather than just replacing what you already typed with the previous command, the shell will instead *search* for the latest command in the history starting with what you already typed.

There are *many* commands or 'widgets' you can assign to keystrokes with the `bindkey` command. You can find a list of default 'widgets' in the documentation.

Conclusion

This concludes the part of the series about configuring `zsh`. When I set out I wanted to recreate the environment I had built in `bash`. Along the way I found a few features in `zsh` that seemed worth adding to my toolkit.

After nearly two months of working in `zsh`, there are already some features I would miss terribly when switching back to `bash` or a plain, unconfigured `zsh`. Most important is the powerful tab-completion. But features like AutoCD, MultiIO, and flexible aliases, are useful tools as well.

The dynamic loading of functions from files in the `fpath` was initially confusing, but it allows configurations and functions to be split out into their own, which simplifies "modularizing" and sharing.

In the next (and last) post, I will cover the changes when scripting with `zsh` vs `bash`.

Moving to zsh, part 8 – Scripting zsh

Apple has announced that in macOS 10.15 Catalina the default shell will be `zsh`.

In this series, I will document my experiences moving `bash` settings, configurations, and scripts over to `zsh`.

- Part 1: Moving to `zsh`
- Part 2: Configuration Files
- Part 3: Shell Options
- Part 4: Aliases and Functions
- Part 5: Completions
- Part 6: Customizing the `zsh` Prompt
- Part 7: Miscellanea
- Part 8: Scripting `zsh` (*this article*)

> *This series has grown into a book: reworked and expanded with more detail and topics. Like my other books, I plan to update and add to it after release as well, keeping it relevant and useful. You can order it on the Apple Books Store now.*

This is the final article in this series. (If I ever announce an eight part series again, please somebody intervene!) However, I am quite sure it will not be the last post on `zsh`

All the previous posts described how `zsh` works *as an interactive* shell. The interactive shell is of course the most direct way we use a shell and configuring the shell to your taste can bring a huge boost in usefulness and productivity.

The other, equally, important aspect of a shell is running script files. In the simplest perspective, script files are just series of interactive commands, but of course they will get complex very quickly.

sh, bash, or zsh?

Should you even script in `zsh`? The argument for `bash` has been that it has been pre-installed on every Mac OS X since 10.2. The same is true for `zsh`, with one exception: the `zsh` binary is **not** present on the Recovery system. It is also not present on a NetInstall or External Installation System, but these are less relevant in a modern deployment workflow, which has to work with Secure Boot Macs.

If you plan to run a script from Recovery, such as an `installr` or `bootstrappr` script or as part of an MDS workflow, your only choices are `/bin/sh` and `/bin/bash`. The current `/bin/bash` binary is 12 years old, and Apple is messaging its demise. I would not consider that a future proof choice. So, if your script may run in a Recovery context, i would recommend `/bin/sh` over either `/bin/bash` or `/bin/zsh`

Since installation packages can be run from the Recovery context as well, and you cannot really always predict in which context your package will be used, I would extend the recommendation to use `/bin/sh` for *all installation scripts* as well.

While `sh` is surely ubiquitous, it is also a 'lowest common denominator', so it is not a very comfortable scripting language to work in. I recommend using `shellcheck` to verify all your `sh` scripts for `bash`isms that might have crept in out of habit.

When you can ensure your script will only run on a full macOS installation, `zsh` is good choice over `sh`. It is pre-installed on macOS, and it offers better and safer language options than `sh` and some advantages over `bash`, too. Deployment scripts, scripts pushed from management systems, launch daemons and launch agents, and script you write to automate your admin workflows (such as building packages) would fall in this category.

You can also choose to stick with `bash`, but then you should start installing and using your own `bash` 5 binary instead of the built in `/bin/bash`. This will give you newer security updates and features and good feeling that when Apple does eventually yank the `/bin/bash` binary, your scripts will keep working.

Admins who want to keep using Python for their scripts are facing a similar problem. Once you choose to use a non-system version of `bash` (or `python`), it is *your* responsibility to install and update it on all your clients. But that is what system management tools are for. We will have to get used to managing our own tools as well as the users' tools, instead of relying on Apple.

Shebang

To switch your script from using `bash` to `zsh`, you have to change the shebang in the first line from `#!/bin/bash` to `#!/bin/zsh`.

If you want to distinguish your `zsh` script files, the you can also change the script's file extension from `.sh` to `.zsh`. This will be especially helpful while you transfer scripts from `bash` to `zsh` (or `sh`). The file extension will have no effect on which interpreter will be used to run the script. That is determined by the shebang, but the extension provides a visible clue in the Finder and Terminal.

zsh vs bash

Since `zsh` derives from the same Bourne shell family as `bash` does, most commands, syntax, and control structures will work just the same. `zsh` provides alternative syntax for some of the structures.

`zsh` has several options to control compatibility, not only for `bash`, but for other shells as well. We have already seen that options can be used to enable features specific for `zsh`. These options can significantly change how `zsh` interprets your scripts.

Because you can never quite anticipate in which environment your particular `zsh` will be launched in, it is good practice to reset the options at the beginning of your script with the `emulate` command:

```
emulate -LR zsh
```

After the `emulate` command, you can explicitly set the shell options your script requires.

The `emulate` command also provides a `bash` emulation:

```
emulate -LR bash
```

This will change the `zsh` options to closely emulate `bash` behavior. Rather than relying on this emulation mode, I would recommend actually using `bash`, even if you have to install and manage a newer version yourself.

Word Splitting in Variable Substitutions

Nearly all syntax from `bash` scripts will 'just work' in `zsh` as well. There are just a few important differences you have to be aware of.

The most significant difference, which will affect most scripts is how `zsh` treats word splitting in variable substitutions.

Recap: `bash` behavior

In `bash` substituted variables are split on whitespace when the substitution is not quoted. To demonstrate, we will use a function that counts the number of arguments passed into it. This way we can see whether a variable was split or not:

```
#!/bin/bash
export PATH=/usr/bin:/bin:/usr/sbin:/sbin

function countArguments() {
    echo "${#@}"
}

wordlist="one two three four five"

echo "normal substitution, no quotes:"
countArguments $wordlist
# -> 5

echo "substitution with quotes"
countArguments "$wordlist"
# -> 1
```

In `bash` and `sh` the contents of the variable split into separate arguments when substituted without the quotes. Usually you do *not* want the splitting to occur. Hence the rule: "always quote variable substitutions!"

`zsh` behavior: no splitting

`zsh` will *not* split a variable when substituted. With `zsh` the contents of a variable will be kept in one piece:

```
#!/bin/zsh
emulate -LR zsh # reset zsh options
export PATH=/usr/bin:/bin:/usr/sbin:/sbin

function countArguments() {
    echo "${#@}"
}

wordlist="one two three four five"

echo "normal substitution, no quotes:"
countArguments $wordlist
# -> 1

echo "substitution with quotes"
countArguments "$wordlist"
# -> 1
```

The positive effect of this is that you do not have to worry about quoting variables all the time, making `zsh` less error prone, and much more like other scripting and programming languages.

Splitting Arrays

The `wordlist` variable in our example above is a *string*. Because of this it returns a count of `1`, since there is only one element, the string itself.

If you want to loop through multiple elements of a list

In `bash` this happens, whether you want to or not, unless you explicitly tell `bash` not to split by quoting the variable.

In `zsh`, you have to explicitly tell the shell to split a string into its components. If you do this naïvely, by wrapping the string variable in the parenthesis to declare and array, it will not work:

```
wordlist="one two three"
wordarray=( $wordlist )

for word in $wordarray; do
    echo "->$word<-"
done
```

```
#output
->one two three<-
```

> *Note: the `for` loop echoes every item in the array. I have added the `->` characters to make the individual items more visible. In the subsequent examples, I will not repeat the `for` loop, but only show its output. So the above example will be shortened to:*

```
wordarray=( $wordlist )
->one two three<-
```

There are few options to do this right.

Revert to `sh` behavior

First, you can tell `zsh` to revert to the `bash` or `sh` behavior and split on any whitespace. You can do this by pre-fixing the variable substitution with an `=`:

```
wordarray=( ${=wordlist} )
->one<-
->two<-
->three<-
```

Note: if you find yourself using the `=` frequently, you can also re-enable `sh` style word splitting with the `shwordsplit` option. This will of course affect *all* substitutions in the script until you disable the option again.

```
setopt shwordsplit
wordarray=( $wordlist )
->one<-
->two<-
->three<-
```

This option can be very useful when you quickly need to convert a `bash` script to a `zsh` script. But you will also re-enable all the problems you had with unintentional word splitting.

Splitting Lines

If you want to be more specific and split on particular characters, `zsh` has a special substitution syntax for that:

```
macOSversion=$(sw_vers -productBuild) # 10.14.6
versionParts=${(s/./)macOSVersion}
->10<-
->14<-
->6<-
```

If you want to split on a newline character `\n` the syntax is slightly different:

```
citytext="New York
Rio
Tokyo"

cityarray=( ${(ps/\n/)citytext} )
->New York<-
->Rio<-
->Tokyo<-
```

Since newline is a common character to split text on, there is a short cut:

```
cityarray=( ${(f)citytext} )
```

Since the newline character *is* a legal character in file names, you should use zero-terminated strings where possible:

```
foundDirs=$(find /Library -type d -maxdepth 1 -print0)
dirlist=${(ps/\0/)foundDirs}
```

Again, there is a shortcut for this:

```
dirlist=${(0)foundDirs}
```

Array index starts at 1

Once you have split text into an array, remember, that in zsh array indices start at 1:

```
% versionList=( ${(s/./)$(sw_vers -productVersion)} )
% echo ${versionList[1]}
10
% echo ${versionList[2]}
14
% echo ${versionList[3]}
6
```

If you think this is wrong and absolutely require a zero-based index, you can set the KSH_ARRAYS shell option:

```
% setopt KSH_ARRAYS
% echo ${versionList[0]}
10
% echo ${versionList[1]}
14
% echo ${versionList[2]}
6
% echo ${versionList[3]}

```

Conclusion

Switching your scripts from bash to zsh requires a bit more work than merely switching out the shebang. However, since /bin/bash will still be present in Catalina, you do not have to move all scripts immediately.

Moving to sh instead of zsh can be safer choice, especially for package installation scripts.

In zsh, there always seems to be some option to disable or enable a particular behavior.

This concludes my series on switching to zsh on macOS. I hope you found it helpful.

After having worked with zsh for a few weeks, I already find some of its features indispensable. I am looking forward to discovering and using more features over time. When I do, I will certainly share them here.

Moving to zsh

Apple has announced that in macOS 10.15 Catalina the default shell will be `zsh`.

In this series, I will document my experiences moving `bash` settings, configurations, and scripts over to `zsh`.

- Part 1: Moving to zsh *(this article)*
- Part 2: Configuration Files
- Part 3: Shell Options
- Part 4: Aliases and Functions
- Part 5: Completions
- Part 6: Customizing the `zsh` Prompt
- Part 7: Miscellanea
- Part 8: Scripting `zsh`

`zsh` (I believe it is pronounced *zee-shell*, though *zish* is fun to say) will succeed `bash` as the default shell. `bash` has been the default shell since Mac OS X 10.3 Panther.

> *This series has grown into a book: reworked and expanded with more detail and topics. Like my other books, I plan to update and add to it after release as well, keeping it relevant and useful. You can order it on the Apple Books Store now.*

Why?

The `bash` binary bundled with macOS has been stuck on version 3.2 for a long time now. `bash` v4 was released in 2009 and `bash` v5 in January 2019. The reason Apple has not switched to these newer versions is that they are licensed with GPL v3. `bash` v3 is still GPL v2.

`zsh`, on the other hand, has an 'MIT-like' license, which makes it much more palatable for Apple to include in the system by default. `zsh` has been available as on macOS for a long time. The `zsh` version on macOS 10.14 Mojave is fairly new (5.3). macOS 10.15 Catalina has the current `zsh` 5.7.1.

Is bash gone!?

No.

macOS Catalina still has the same `/bin/bash` (version 3.2.57) as Mojave and earlier macOS versions. This change is only for new accounts created on macOS Catalina. When you upgrade to Catalina, a user's default shell will remain what it was before.

Many scripts in macOS, management systems, and Apple and third party installers rely on `/bin/bash`. If Apple just yanked this binary in macOS 10.15 Catalina or even 10.16. Many installers and other solutions would break and simply cease to function.

Users that have `/bin/bash` as their default shell on Catalina will see a prompt at the start of each Terminal session stating that `zsh` is now the recommended default shell. If you want to continue using `/bin/bash`, you can supress this message by setting an environment variable in your `.bash_profile` or `.bashrc`.

```
export BASH_SILENCE_DEPRECATION_WARNING=1
```

You can also download and install a newer version of bash yourself. Keep in mind that custom bash installations reside in a different directory, usually `/usr/local/bin/bash`.

Will bash remain indefinitely?

Apple is strongly messaging that you should switch shells. This is different from the last switch in Mac OS X 10.3 Panther, when Apple switched the default to `bash`, but didn't really care if you remained on `tcsh`. In fact, `tcsh` is still present on macOS.

Apple's messaging should tell us, that the days of `/bin/bash` *are* numbered. Probably not *very* soon, but eventually keeping a more than ten year old version of `bash` on the system will turn into a liability. The built-in bash had to be patched in 2014 to mitigate the 'Shellshock' vulnerability. At some point Apple will consider the cost of continued maintenance too high.

Another clue is that a new shell appeared on macOS Catalina (and is mentioned in the support article). The 'Debian Almquist Shell' `dash` has been added to the lineup of shells. `dash` is designed to be a minimal implementation of the Posix standard shell `sh`. So far, in macOS (including Catalina),`sh` invokes `bash` in `sh`-compatibility mode.

As Apple's support article mentions, Catalina also adds a new mechanism for users and admins to change which shell handles `sh` invocations. MacAdmins or users can change the symbolic link stored in `/var/select/sh` to point to a shell other than `/bin/bash`. This changes which shell interprets scripts the `#!/bin/sh` shebang or scripts invoked with `sh -c`. Changing the interpreter for `sh` should not, but may change the behavior of several crucial scripts in the system, management tools, and in installers, but may be very useful for testing purposes.

All of these changes are indicators that Apple is preparing to remove `/bin/bash` at some, yet indeterminate, time in the future.

Do I need to wait for Catalina to switch to zsh?

No, `zsh` is available Mojave and on older macOS versions. You can start testing `zsh` or even switch your default shell already.

If you want to just see how `zsh` works, you can just open Terminal and type `zsh`:

```
$ zsh
MacBook%
```

The main change you will see is that the prompt looks different. `zsh` uses the `%` character as the default prompt. (You can change that, of course.) Most navigation keystrokes and other behaviors will remain the same as in `bash`.

If you want to already switch your default shell to `zsh` you can use the `chsh` command:

```
$ chsh -s /bin/zsh
```

This will prompt for your password. This command will not change the current shell, but all new ones, so close the current Terminal windows and tabs and open a new one.

How is zsh different?

Like `bash` ('Bourne again shell'), `zsh`derives from the 'Bourne' family of shells. Because of this common ancestry, it behaves very similar in day-to-day use. The most obvious change will be the different prompt.

The main difference between `bash` and `zsh` is configuration. Since `zsh` ignores the `bash` configuration files (`.bash_profile` or `.bashrc`) you cannot simply copy customized bash settings over to `zsh`. `zsh` has much more options and points to change `zsh` configuration and behavior. There is an entire eco-system of configuration tools and themes called `oh-my-zsh` which is very popular.

`zsh` also offers better configuration for auto-completion which is far easier than in `bash`.

I am planning a separate post, describing how to transfer (and translate) your configurations from `bash` to `zsh`.

What about scripting?

Since `zsh` has been present on macOS for a long time, you could start moving your scripts from `bash` to `zsh` right away and not lose backwards compatibility. Just remember to set the shebang in your scripts to `#!/bin/zsh`.

You will gain some features where `zsh` is superior to `bash` v3, such as arrays and associative arrays (dictionaries).

There is one exception where I would now recommend to use `/bin/sh` for your scripts: the Recovery system does *not* contain the `/bin/zsh` shell, even on the Catalina beta. This could still change during the beta phase, or even later, but then you still have to consider *older* macOS installations where `zsh` is definitely not present in Recovery.

When you plan to use your scripts or pkgs with installation scripts in a Recovery (or NetInstall, or bootable USB drive) context, such as Twocanoes MDS, installr or bootstrappr, then you *cannot* rely on `/bin/zsh`.

Since we now know that `bash` is eventually going away, the only common choice left is `/bin/sh`.

When you build an installer package, it can be difficult to anticipate all the contexts in which it might be deployed. So, for installation pre- and postinstall scripts, I would recommend using `/bin/sh` as the shebang from now on.

I used to recommend using `/bin/bash` for everything MacAdmin related. `/bin/sh` is definitely a step down in functionality, but it seems like the safest choice for continued support.

Summary

Overall, while the messaging from Apple is very interesting, the change itself is less dramatic than the headlines. Apple is not 'replacing' `bash` with `zsh`, at least not yet. Overall, we will have to re-think and re-learn a few things, but there is also much to be gained by finally switching from a ten-year-old shell to a new modern one!

This git repo has been shared by many on MacAdmins Slack: rothgar/mastering-zsh, I will certainly dive into that and share about my experiences here!

Moving to zsh, part 2: Configuration Files

Apple has announced that in macOS 10.15 Catalina the default shell will be `zsh`.

In this series, I will document my experiences moving `bash` settings, configurations, and scripts over to `zsh`.

- Part 1: Moving to zsh
- Part 2: Configuration Files *(this article)*
- Part 3: Shell Options
- Part 4: Aliases and Functions
- Part 5: Completions
- Part 6: Customizing the `zsh` Prompt
- Part 7: Miscellanea
- Part 8: Scripting `zsh`

> *This series has grown into a book: reworked and expanded with more detail and topics. Like my other books, I plan to update and add to it after release as well, keeping it relevant and useful. You can order it on the Apple Books Store now.*

In part one I talked about Apple's motivation to switch the default shell and urge existing users to change to `zsh`.

Since I am new to `zsh` as well, I am planning to document my process of transferring my personal `bash` setup and learning the odds and ends of `zsh`.

Many websites and tutorials leap straight to projects like oh-my-zsh or prezto where you can choose from hundreds of pre-customized and pre-configured themes.

While these projects are very impressive and certainly show off the flexibility and power of `zsh` customization, I feel this will actually prevent an understanding of how `zsh` works and how it differs from `bash`. So, I am planning to build my own configuration 'by hand' first.

At first, I actually took a look at my current `bash_profile` and cleaned it up. There were many aliases and functions which I do not use or broke in some macOS update. I the end, this is what I want to re-create in `zsh`:

- aliases
  - mostly shortcuts to `open` files with a specific application
- functions
  - show man pages in a dedicated Terminal window
  - some more simple functions
  - get the frontmost Finder window path
- shell settings
  - case-insensitive globbing
  - case-insensitive path-completion (for `bash` this is set in `.inputrc`)
  - command history, shared across windows and sessions
  - use BBEdit as the editor
- prompt:
  - show current working dir
  - show a colored symbol showing the last command's exit code
  - update the Terminal window title bar to show the cwd

Most of these should be fairly easy to transfer. Some might be… interesting.

But first, where do we put our custom `zsh` configuration?

zsh Configuration Files

`bash` has a list of possible files that it tries in predefined order. I have the description in my post on the `bash_profile`.

`zsh` also has a list of files it will execute at shell startup. The list of possible files is even longer, but somewhat more ordered.

all users	user	login shell	interactive shell	scripts	Terminal.app
`/etc/zshenv`	`.zshenv`	√	√	√	√
`/etc/zprofile`	`.zprofile`	√	x	x	√
`/etc/zshrc`	`.zshrc`	√	√	x	√

all users	user	login shell	interactive shell	scripts	Terminal.app
`/etc/zlogin`	`.zlogin`	√	x	x	√
`/etc/zlogout`	`.zlogout`	√	x	x	√

The files in `/etc/` will be launched (when present) for all users. The `.z*` files only for the individual user.

By default, `zsh` will look in the root of the home directory for the user `.z*` files, but this behavior can be changed by setting the `ZDOTDIR` environment variable to another directory (e.g. `~/.zsh/`) where you can then group all user `zsh` configuration in one place.

On macOS you could set the `ZDOTDIR` to `~/Documents/zsh/` and then use iCloud syncing (or a different file sync service) to have the same files on all your Macs. (I prefer to use `git`.)

`bash` will either use `.bash_profile` for login shells, or `.bashrc` for interactive shells. That means, when you want to centralize configuration for all use cases, you need to `source` your `.bashrc` from `.bash_profile` or vice versa.

`zsh` behaves differently. `zsh` will run *all* of these files in the appropriate context (login shell, interactive shell) when they exist.

`zsh` will start with `/etc/zshenv`, then the user's `.zshenv`. The `zshenv` files are *always* used when they exist, *even for scripts* with the `#!/bin/zsh` shebang. Since changes applied in the `zshenv` will affect `zsh` behavior in *all* contexts, you should you should be very cautious about changes applied here.

Next, when the shell is a login shell, `zsh` will run `/etc/zprofile` and `.zprofile`. Then for interactive shells (and login shells) `/etc/zshrc` and `.zshrc`. Then, again, for login shells `/etc/zlogin` and `.zlogin`. Why are there two files for login shells? The `zprofile` exists as an analog for `bash`'s and `sh`'s profile files, and `zlogin` as an analog for `ksh` login files.

Finally, there are `zlogout` files that can be used for cleanup, when a login shell exits. In this case, the user level `.zlogout` is read first, then the central `/etc/zlogout`. If the shell is terminated by an external process, these files might not be run.

Apple Provided Configuration Files

macOS Mojave (and earlier versions) includes `/etc/zprofile` and `/etc/zshrc` files. Both are very basic.

`/etc/zprofile` uses `/usr/libexec/path_helper` to set the default `PATH`. Then `/etc/zshrc` enables UTF–8 with `setopt combiningchars`.

Like `/etc/bashrc` there is a line in `/etc/zshrc` that would load `/etc/zshrc_Apple_Terminal` if it existed. This is interesting as `/etc/bashrc_Apple_Terminal` contains quite a lot of code to help `bash` to communicate with the Terminal application. In particular `bash` will send a signal to the Terminal on every new prompt to update the path and icon displayed in the Terminal window title bar, and provides other code relevant for saving and restoring Terminal sessions between application restarts.

However, there is no `/etc/zshrc_Apple_Terminal` and we will have to provide some of this functionality ourselves.

> *Note: As of this writing, `/etc/zshrc` in the macOS Catalina beta is different from the Mojave `/etc/zshrc` and provides more configuration. However, since Catalina is still beta, I will focus these articles on Mojave and earlier. Once Catalina is released, I may update these articles or write a new one for Catalina, if necessary.*

Which File to use?

When you want to use the `ZDOTDIR` variable to change the location of the other `zsh` configuration files, setting that variable in `~/.zshenv` seems like a good choice. Other than that, you probably want to *avoid* using the `zshenv` files, since it will change settings for *all* invocations of `zsh`, including scripts.

macOS Terminal considers every new shell to be a login shell *and* an interactive shell. So, in Terminal a new `zsh` will potentially run *all* configuration files.

For simplicity's sake, you should use just one file. The common choice is `.zshrc`.

Most tools you can download to configure `zsh`, such as 'prezto' or 'oh-my-zsh', will override or re-configure your `.zshrc`. You could consider moving your code to `.zlogin` instead. Since `.zlogin` is sourced *after* `.zshrc` it can override settings from `.zshrc`. However, `.zlogin` is *only* called for login shells.

The most common situation where you do *not* get a login shell with macOS Terminal, is when you switch to `zsh` from another shell by typing the `zsh` command.

I would recommend to put your configuration in *your* `.zshrc` file and if you want to use any of the theme projects, read and follow their instructions closely as to how you can preserve your configurations together with theirs.

Managing the shell for Administrators

MacAdmins may have the need to manage certain shell settings for their users, usually environment variables to configure certain command line tool's behaviors.

The most common need is to expand the `PATH` environment variable for third party tools. Often the third party tools in question will have elaborate postinstall scripts that attempt to modify the current user's `.bash_profile` or `.bashrc`. Sometimes, these tools even consider that a user might have changed the default shell to something other than `bash`.

On macOS, system wide changes to the `PATH` should be done by adding files to `/etc/paths.d`.

As an administrator you should be on the lookout for scripts and installers that attempt to modify configuration files on the user level, disable the scripts during deployment, and manage the required changes centrally. This will allow you to keep control of the settings even as tools change, are added or removed from the system, while preserving the user's custom configurations.

To manage environment variables other than `PATH` centrally, administrators should consider `/etc/zshenv` or adding to the existing `/etc/zshrc`. In these cases you should always monitor whether updates to macOS overwrite or change these files with new, modified files of their own.

Summary

There are many possible files where the `zsh` can load user configuration. You should use `~/.zshrc` for your personal configurations.

There are many tools and projects out there that will configure `zsh` for you. This is fine, but might keep you from really understanding how things work.

MacAdmins who need to manage these settings centrally, should use `/etc/paths.d` and similar technologies or consider `/etc/zshenv` or `/etc/zshrc`.

Apple's built-in support for `zsh` in Terminal is not as detailed as it is for `bash`.

Next: Part 3 – Shell Options

Moving to zsh, part 3: Shell Options

Apple has announced that in macOS 10.15 Catalina the default shell will be `zsh`.

In this series, I will document my experiences moving `bash` settings, configurations, and scripts over to `zsh`.

- Part 1: Moving to zsh
- Part 2: Configuration Files
- Part 3: Shell Options *(this article)*
- Part 4: Aliases and Functions
- Part 5: Completions
- Part 6: Customizing the `zsh` Prompt
- Part 7: Miscellanea
- Part 8: Scripting `zsh`

> *This series has grown into a book: reworked and expanded with more detail and topics. Like my other books, I plan to update and add to it after release as well, keeping it relevant and useful. You can order it on the Apple Books Store now.*

Now that we have chosen a file to configure our `zsh`, we need to decide on 'what' to configure and 'how.' In this post, I want to talk about `zsh`'s shell options.

As I have mentioned in the earlier posts, I am aware that there are many solutions out there that give you a pre-configured 'shortcut' into lots of `zsh` goodness. But I am interested in learning this the 'hard way' without shortcuts. Call me old-fashioned. ("Uphill! In the snow! Both ways!")

In the previous post, I listed some features that I would like to transfer from my `bash` configuration. While researching how to implement these options in `zsh`, I found a few, new and interesting options in `zsh`.

The settings from `bash` which I *want* in `zsh` were:

- case-insensitive globbing
- command history, shared across windows and sessions

> *Note: `bash` in this series of posts specifically refers to the version of `bash` that comes with macOS as `/bin/bash` (v3.2.57).*
>
> *Note 2: Mono-typed lines starting with a `%` show commands and results from `zsh`. Mono-typed lines starting with `$` show commands and results in `bash`*

What are Shell Options?

Shell options are preferences for the shell's behavior. You are using shell options in `bash`, when you enable 'trace mode' for scripts with the `set -x` command or the `bash -x` option. (Note: this also works with `zsh` scripts.)

`zsh` has *a lot* of shell options. Many of these options serve the purpose of enabling (or disabling) compatibility with other shells. There are also many options which are specific to `zsh`.

You can set an option with the `setopt` command. For compatibility with other shells the `setopt` command and `set -o` have the same effect (set an option by name). The following commands set the same option:

```
set -o AUTO_CD
setopt AUTO_CD
```

The names or labels of the options are commonly written in all capitals in the documentation but in lowercase when listed with the `setopt` tool. The labels of the options are case insensitive and any underscores in the label are ignored. So, these commands set the same option:

```
setopt AUTO_CD
setopt autocd
setopt auto_cd
setopt autoCD
```

There are quite a few ways to negate or unset an option. First you can use `unsetopt` or `set +o`. Alternatively, you can prefix with `NO` or `no` to negate an option. The following commands all have the same effect of turning *off* the previously set option `AUTO_CD`

```
unsetopt AUTO_CD
set +o AUTO_CD
unsetopt autocd
setopt NO_AUTO_CD
setopt noautocd
```

Any options you change will only take effect in the current instance of `zsh`. When you want to change the settings for all new shells, you have to put the commands in one of the configuration files (usually `.zshrc`).

Showing the current Options

You can list the existing shell options with the `setopt` command:

```
% setopt
combiningchars
interactive
login
monitor
shinstdin
zle
```

This list only shows options are changed from the default set of options for `zsh`. These options are marked with `<D>` (default for all shell emulations) or `<Z>` (default for `zsh`) in the documentation or the `zshoptions` man page.

You can also get a list of all default `zsh` options with the command:

```
% emulate -lLR zsh
```

Some zsh Options I use

As I have mentioned before in my posts on `bash` configuration, I prefer minimal configuration changes, so I do not feel all awkward and lost when I have to work on an 'un-configured' Mac.

These configurations are a personal choice and you should pick and choose your own. You can find a full list of `zsh` options in the `zsh` Manual or with `man zshoptions`.

On the other hand, exploring the options allows us to explore a few useful `zsh` features.

Case Insensitive Globbing

Note: 'Globbing' is a unix/shell term that refers to the expansion of wildcard characters, such as `*` and `?` into full file paths and names. I.e. `~/D*` is expanded into `/Users/armin/Desktop /Users/armin/Documents /Users/armin/Downloads`

Since the file system on macOS is (usually) case-insensitive, I prefer globbing and tab-completion to be case-insensitive as well.

The `zsh` option which controls this is `CASE_GLOB`. Since we want globbing to be *case-insensitive*, we want to turn the option off, so:

```
setopt NO_CASE_GLOB
```

You can test this in the shell:

```
% ls ~/d*<tab>
```

In `zsh` tab completion will replace the wildcard with the actual result. So after the tab you will see:

```
% ls /Users/armin/Desktop /Users/armin/Documents
/Users/armin/Downloads
```

Using tab completion this way to see and possibly edit the actual replacement for wildcards is a useful safety net.

In `bash` hit the tab key will list possible completions, but not substitute them in the command prompt.

If you *do not* like this behavior in `zsh` then you can change to behavior similar to `bash` with:

```
setopt GLOB_COMPLETE
```

Automatic CD

Sometimes you enter the path to a directory, but forget the leading `cd`:

```
$ Library/Preferences/
bash: Library/Preferences/: is a directory

% Library/Preferences
zsh: permission denied: Library/Preferences
```

With `AUTO_CD` enabled in `zsh`, the shell will automatically change directory:

```
% Library/Preferences
% pwd
/Users/armin/Library/Preferences
```

This works with relative *and* absolute paths, including the `..`:

```
% ..
% pwd
/Users/armin/Library
% ../Desktop
% pwd
/Users/armin/Desktop
```

I have an `alias` in my `.bash_profile` that sets the `..` command to `cd ..`. Auto CD replaces that functionality and more.

Enable Auto CD with:

```
setopt AUTO_CD
```

Shell History

Shells commonly remember previously executed commands and allows you to recall them with the up and down arrow keys, search or special history commands.

Most of those keys work the same in `zsh`. However, there are a few things you need to configure for `zsh` history to work as you are used to with `bash` on macOS.

By default, `zsh` does *not* save its history when the shell exits. The history is 'forgotten' when you close a Terminal window or tab. To make `zsh` save its history to a file when it exits, you need to set a variable in the shell:

```
HISTFILE=${ZDOTDIR:-$HOME}/.zsh_history
```

Note: this is not a shell option but shell variable or parameter. I will cover some more of those later, You can find a list of variables used by `zsh` in the documentation.

The `HISTFILE` variable tells `zsh` where to store the history data. The syntax `${ZDOTDIR:-$HOME}` means it will use the value of `ZDOTDIR` when it is set or default to the value of `HOME` otherwise. When a user has set the `ZDOTDIR` variable to group their configurations files in a specific directory, the history will be stored there as well.

By default `zsh` simply writes each command in its own line in the history file. You can view the file's contents with any text editor or list the last few commands:

```
% tail -n 10 ~/.zsh_history
```

You can make `zsh` add a bit more data (timestamp in unix epoch time and elapsed time of the command) by setting the `EXTENDED_HISTORY` shell option.

```
setopt EXTENDED_HISTORY
```

You can set limits on how many commands the shell should remember in the session and in the history file with the `HISTSIZE` and `SAVEHIST` variables:

```
SAVEHIST=5000
HISTSIZE=2000
```

When the shell reaches this limit the oldest commands will be removed from memory or the history file.

By default, when you exit `zsh` (for example, by closing the window or tab) this particular instance of `zsh` will *overwrite* an existing history file with its history. So when you have multiple Terminal windows or tabs open, they will all overwrite each others' histories eventually.

You can tell `zsh` to use a single, shared history file across the sessions and append to it rather than overwrite:

```
# share history across multiple zsh sessions
setopt SHARE_HISTORY
# append to history
setopt APPEND_HISTORY
```

Furthermore, you can tell `zsh` to update the history file after every command, rather than waiting for the shell to exit:

```
# adds commands as they are typed, not at shell exit
setopt INC_APPEND_HISTORY
```

When you use a shared history file, it will grow very quickly, and you may want to use some options to clean out duplicates and blanks:

```
# expire duplicates first
setopt HIST_EXPIRE_DUPS_FIRST
# do not store duplications
setopt HIST_IGNORE_DUPS
#ignore duplicates when searching
setopt HIST_FIND_NO_DUPS
# removes blank lines from history
setopt HIST_REDUCE_BLANKS
```

(some of these are redundant)

Most of the time you will access the history with the up arrow key to recall the last command, or maybe a few more steps. You can search through the history with `ctrl-R`

In `zsh`, you can also use the `!!` history substitution, which will be replaced with the entire last command. This is most commonly used in combination with `sudo`:

```
% systemsetup -getRemoteLogin
You need administrator access to run this tool... exiting!
% sudo !!
sudo systemsetup -getRemoteLogin
Password:
Remote Login: On
```

By default, the shell will show the command it is substituting before it is run. But at that point, it is too late to make any changes. When you set the `HIST_VERIFY` option, `zsh` will show the substituted command in the prompt instead, giving you a chance to edit or cancel it, or just confirm it.

```
% systemsetup -getRemoteLogin
You need administrator access to run this tool... exiting!
% sudo !!
% sudo systemsetup -getRemoteLogin
Password:
Remote Login: On
```

This works for other history substitutions such as `!$` or `!*`, as well. You can find all of `zsh`'s history expansions in the documentation.

Correction

When you mistype a command or path, the shell is usually unforgiving. In `zsh` you can enable correction. Then, the shell will make a guess of what you meant to type and ask whether you want do that instead:

```
% systemprofiler
zsh: correct 'systemprofiler' to 'system_profiler' [nyae]?
```

Your options are to

- `n`: execute as typed
- `y`: accept and execute the suggested correction
- `a`: abort and do nothing
- `e`: return to the prompt to continue editing

I have found this far less annoying and far more useful than I expected. Especially, since it works together with `AUTO_CD`:

```
% Dekstop
zsh: correct 'Dekstop' to 'Desktop' [nyae]?
```

You enable `zsh` correction with these options:

```
setopt CORRECT
setopt CORRECT_ALL
```

Reverting to defaults

Most of the changes mentioned here affect the interactive shell and will have little impact on `zsh` scripts. However, there are some options that do affect the behavior of things like variable substitutions which will affect scripts.

You can revert the options for the current shell to the default settings with the following command:

```
emulate -LR zsh
```

We encountered this command earlier when we listed the default settings. The `-l` option will list the settings rather than apply them.

If in doubt, it may be useful to add this at the beginning of your `zsh` scripts.

Next

In the next part we will take a look at aliases and functions.

Moving to zsh, part 4: Aliases and Functions

Apple has announced that in macOS 10.15 Catalina the default shell will be `zsh`.

In this series, I will document my experiences moving `bash` settings, configurations, and scripts over to `zsh`.

- Part 1: Moving to zsh
- Part 2: Configuration Files
- Part 3: Shell Options
- Part 4: Aliases and Functions *(this article)*
- Part 5: Completions
- Part 6: Customizing the `zsh` Prompt
- Part 7: Miscellanea
- Part 8: Scripting `zsh`

> *This series has grown into a book: reworked and expanded with more detail and topics. Like my other books, I plan to update and add to it after release as well, keeping it relevant and useful. You can order it on the Apple Books Store now.*

As I have mentioned in the earlier posts, I am aware that there are many solutions out there that give you a pre-configured 'shortcut' into lots of `zsh` goodness. But I am interested in learning this the 'hard way' without shortcuts. Call me old-fashioned. ("Uphill! In the snow! Both ways!")

Aliases

Aliases in `zsh` work just like aliases in `bash`. You declare an alias with the `alias` (built-in) command and it will work as a text replacement at the beginning of the command prompt:

```
alias ll='ls -al'
```

You can just copy your alias declarations from your `.bash_profile` or `.bashrc` to your `.zshrc`. I had aliases for `..` and `cd..` which are now handled by Auto CD and shell correction respectively, so I didn't bother to move those. (part 3: 'Shell Options')

After the alias is declared, you can use it *at the beginning of a command*. When you try to use the alias anywhere else in the command, the alias will not work:

```
% sudo ll
sudo: ll: command not found
```

Global Aliases

This is where `zsh` has an advantage. You can declare an alias as a 'global' alias, and then will be replaced *anywhere* in the command line:

```
% alias -g badge='tput bel'
% sudo badge        #<beeps> with privilege
```

Identifying Aliases

There is one more feature of `zsh` that is useful with aliases. The `which` command will show if a command stems from an alias substitution:

```
% which ll
ll: aliased to ls -l
```

However, when you try this with global aliases, the substitution occurs *before* the `which` command can evaluate the alias, which leads to an unexpected result:

```
% which badge
/usr/bin/tput
bel not found
```

You can suppress the alias substitution by escaping the first character or by quoting the entire alias name:

```
% which \badge
badge: globally aliased to tput bel
% which 'badge'
badge: globally aliased to tput bel
```

Functions

As with aliases, functions in your zsh configuration will work just as they did in bash.

```
function vnc() {
    open vnc://"$USER"@"$1"
}
```

This code in your zsh configuration file will define the vnc function and make it available in the shell.

Autoload Functions

However, zsh has some features which make using functions more flexible. There is (once again) a bit of configuration required to get this working.

Instead of declaring the function directly the configuration file, you can put the function in a separate file. zsh has a built-in variable called fpath which is an array of paths where zsh will look for files defining a function. You can add your own directory to this search path:

```
fpath+=~/Projects/dotfiles/zshfunctions
```

Just having a file in the directory is not enough. You still have to tell zsh that you want to use this particular function:

```
autoload vnc
```

This command tells zsh: "'Declare a function named vnc. To execute it, load a file named vnc, it is somewhere in the fpath."

Note: you often see the -U or -Uz option added to the autoload command. These options help avoid conflicts with your personal settings. They suppress alias substitution and ksh-style loading of functions, respectively.

The vnc file in my zshfunctions directory can look like this:

```
# uses the arguments as hostnames for `open vnc://` (Screen Sharing)
# uses the $USER username as default account name

for x in $@; do
    open vnc://"$USER"@"$x"
done
```

The `vnc` function will open a Screen Sharing session with the current user name pre-filled in.

Initializing Autoload Functions

You could also put the code in the function file into a `function` block:

```
function vnc() {
    for x in $@; do
        open vnc://"$vnc_user"@"$x"
    done
}

# initialization code
vnc_user="remote_admin"
alias screen_sharing='vnc'
```

The function name should match the function name declared with `autoload`.

When you have additional code *outside* the function, the `autoload` behavior changes. When the function is called for the first time, the function will be defined and the code outside the function will be run. The function itself will *not* be executed on the first run. On subsequent calls, the function will be executed and the code outside the function is ignored.

You can use this to provide setup and initialization code for the function. You can even have more functions defined in the function file. The above example declares and sets a variable to use for account name and an alias for the `vnc` command.

Since you have to run the function once for the initialization, you often see this syntax in the `zsh` configuration file:

```
autoload vnc && vnc
```

Which means 'declare the function and if that succeeds run it.'

In some functions, the initialization code will already launch the function itself:

```
function vnc() {
    ...
}
```

```
# initialization
vnc_user="remote_admin"
vnc()
```

Since the behavior will vary from each `autoload`ed function to the next, be sure to study any documentation or the function's code.

Identifying Functions

Finally, the `which` command will show the function code:

```
 % which vnc
vnc () {
    for x in $@
    do
        open vnc://"$USER"@"$x"
    done
}
```

The `functions` command without any parameters, will print *all* functions (there will be a lot of them). Use `functions +` to just list the function names.

Debugging Functions

When you are working on complex autoloaded functions, you will at some point have to do some debugging. You can enable tracing for functions with

```
% functions -t vnc
% vnc Client.local
+vnc:1> x=Client.local
+vnc:2> open vnc://armin@Client.local
```

You can disable tracing for this function with `functions +t vnc`.

Next

In the next part we will enable, use and configure tab completions.

Moving to zsh, part 5: Completions

Apple has announced that in macOS 10.15 Catalina the default shell will be `zsh`.

In this series, I will document my experiences moving `bash` settings, configurations, and scripts over to `zsh`.

- Part 1: Moving to zsh
- Part 2: Configuration Files
- Part 3: Shell Options
- Part 4: Aliases and Functions
- Part 5: Completions *(this article)*
- Part 6: Customizing the `zsh` Prompt
- Part 7: Miscellanea
- Part 8: Scripting `zsh`

> *I am preparing a book on this topic, reworked and expanded with more detail and topics. Like my other books, I plan to update and add to it after release as well, keeping it relevant and useful. You can pre-order it on the Apple Books Store now.*

As I have mentioned in the earlier posts, I am aware that there are many solutions out there that give you a pre-configured 'shortcut' into lots of `zsh` goodness. But I am interested in learning this the 'hard way' without shortcuts. Call me old-fashioned. ("Uphill! In the snow! Both ways!")

What are Completions?

Man shells use the tab key (⇥) for completion. When you press that key, the shell tries to guess what you are typing and will complete it, or if the beginning of what you typed is ambiguous, suggest from a list of possible completions.

For example when you want to `cd` to your `Documents` folder, you can save typing:

```
% cd ~/Doc⇥
% cd ~/Documents/
```

When you hit the tab key, the system will complete the path to the `Documents` folder.

When the completion is ambiguous, the shell will list possible completions:

```
% cd ~/D⇥
Desktop/    Documents/  Downloads/
```

At this point, you can add a character or two to get to a unique completion, and hit the tab key again. In `zsh` you can also hit the tab key repeatedly to cycle through the suggested completions. In this example, the first tab keystroke will show the list, the second will complete `~/Desktop/`, the third completes `~/Documents`, and so on.

You can use tab completion commands as well:

```
% system⇥
system_profiler     systemkeychain      systemsetup
systemsoundserverd  systemstats
% system_⇥
% system_profiler
```

Not having to type path and file names saves time and *avoids errors*, especially with complex paths with spaces and other special characters:

```
% cd ~/Li⇥
% cd ~/Library/Appl⇥
% cd ~/Library/Application S⇥
Application Scripts/  Application Support/
% cd ~/Library/Application Su⇥
% cd ~/Library/Application Support/
```

Using tab completion is a huge productivity boost when using a shell.

Turning It On

In the default configuration, tab completion in `zsh` is very basic. It will complete commands and paths, but not much else. But you can enable a very powerful, and useful completion system.

`zsh` comes with a tool you can use to setup this completion system. When you run the `compinstall` command it will lead you through a complex and hard to understand list of menus which explains the options and will generate the code necessary to set this configuration up and add it to your `.zshrc` file or another configuration file of your choice.

Since the commands to configure the completion are quite arcane and hard to understand, this is a good way to get something to start out with. I will explain some of these options and commands in detail.

Whether you use `compinstall or not, to turn on the more powerful completion system, you need to add at least this command to your`zsh` configuration file:

```
autoload -Uz compinit && compinit
```

This will initialize the `zsh` completion system. The details of this system are documented here.

If you want to configure the system, the configuration commands (usually `zstyle` commands) should be added to the `zsh` configuration file *before* you enable the system. (This only matters for a few configurations, but as a general rule it is safer.)

All of these completion rules need to be loaded and prepared. `zsh`'s completion system creates a cache in the file `~/.zcompdump`. The first time you run `compinit` it might take a noticeable time, but subsequent runs should use this cache and be much faster.

Sometimes, especially when building and debugging your own completion files, you may need to delete this file to force a rebuild:

```
% rm -f ~/.zcompdump
% compinit
```

Case Insensitive Completion

Since the macOS file systems are usually case-insensitive, I prefer my tab-completion to be case-insensitive as well. For `bash` you configure that in the `~/.inputrc`. In `zsh` you modify the completion systems behavior with this (monstrous) command:

```
# case insensitive path-completion
zstyle ':completion:*' matcher-list 'm:{[:lower:][:upper:]}={[:upper:]
[:lower:]}' 'm:{[:lower:][:upper:]}={[:upper:][:lower:]} l:|=* r:|=*' 'm:
{[:lower:][:upper:]}={[:upper:][:lower:]} l:|=* r:|=*' 'm:{[:lower:]
[:upper:]}={[:upper:][:lower:]} l:|=* r:|=*'
```

I have seen many varieties for this configuration in different websites, but this is what `compinstall` adds when I select case-insensitive completion, so I am going with that.

Partial Completion

This is a particularly nice feature. You can type fragments of each path segment and the completion will try to complete them all at once:

```
% cd /u/lo/b⇥
% cd /usr/local/bin
```

```
% cd ~/L/P/B⇥
% ~/Library/Preferences/ByHost/
```

If the fragments are ambiguous, there are different strategies to what the completion system suggests. I have configured these like this:

```
# partial completion suggestions
zstyle ':completion:*' list-suffixes
zstyle ':completion:*' expand prefix suffix
```

Commands with built-in completion

`zsh` comes with several completion definitions for many commands. For example, when you type `cp` and then hit tab, the system will correctly assume you want to complete a file path and show the suggestions from the current working directory.

However, when you type `cp -⇥` the completion can tell from the `-` that you want to add an option to the command and suggest a list of options for `cp`, with short descriptions.

```
% cp -⇥
-H  -- follow symlinks on the command line in recursive mode
-L  -- follow all symlinks in recursive mode
-P  -- do not follow symlinks in recursive mode (default)
-R  -- copy directories recursively
-X  -- don't copy extended attributes or resource forks
-a  -- archive mode, same as -RpP
-f  -- force overwriting existing file
-i  -- confirm before overwriting existing file
```

```
-n  -- don't overwrite existing file
-p  -- preserve timestamps, mode, owner, flags, ACLs, and extended attributes
-v  -- show file names as they are copied
```

As the context of command prompt you are assembling changes, you may get different completion suggestions. For example, the completion for `ssh` will suggest host names:

```
% ssh armin@⇥
```

`zsh` comes with completion definitions for many common commands. Nevertheless, it can be helpful to just hit tab, especially when wondering about options.

On macOS completions are stored in `/usr/share/zsh/5.3/functions` (replace the`5.3` with `5.7.1` in Catalina). This directory stores many functions used with `zsh` and is in the default `fpath`. All the files in that directory that start with an underscore _ contain the completion definitions command. So, the file `_cp` contains the definition for the `cp` command. (Some of the definition files contain the definitions for multiple commands.)

Completions for macOS Commands

There are even a few macOS specific command that come with the default `zsh` installation.

```
% system_profiler ⇥⇥
```

macOS High Sierra and macOS Mojave come with `zsh` 5.3, which is now nearly two years old. `zsh` 5.3 contains less macOS specific completion definitions than the current `zsh` 5.7.1 which will is the pre-installed `zsh` in macOS Catalina. Some of the completions in 5.3 have also been updated in 5.7.1.

Tool	zsh 5.3	zsh 5.7.1
`caffeinate`		√
`defaults`	√	√
`fink`	√	√
`fs_usage`		√

Tool	zsh 5.3	zsh 5.7.1
`hdiutil`	√	√
`mdfind`		√
`mdls`		√
`mdutil`		√
`networksetup`		√
`nvram`		√
`open`	√	√
`osascript`		√
`otool`		√
`pbcopy/pbpaste`		√
`plutil`		√
`say`		√
`sc_usage`		√
`scselect`		√
`scutil`		√
`softwareupdate`	√	√
`sw_vers`		√
`swift`		√
`system_profiler`	√	√
`xcode_select`		√

Load bash completions

Since the default shell on macOS has been `bash` for so long, there are quite a few `bash` completion definitions for macOS commands and third party tools available. For example Tony Williams' `bash` completion for `autopkg` (post, Github).

You do *not* have to rewrite these completions, since the `zsh` completion system can use `bash` completion scripts as well: (add this to your `zsh` configuration file)

```
# load bashcompinit for some old bash completions
autoload bashcompinit && bashcompinit

[[ -r ~/Projects/autopkg_complete/autopkg ]] && source
~/Projects/autopkg_complete/autopkg
```

When you have multiple `bash` completion scripts you want to load, you only need to load `bashcompinit` once.

Build your own completions

Once you start using completions, you will want to have them *everywhere*. While many built-in completions exists, there are still many commands that lack a good definition.

Some commands, like the `swift` command line tool, have a built-in option to generate the completion syntax. You can then store that in a file and put it in your `fpath`:

```
% swift package completion-tool generate-zsh-script >_swift
```

> *Note: in the case of `swift`, its definition will conflict with the `_openstack` definition in `zsh` 5.3. You can fix this with the command `compdef _swift swift` after loading the completion system.*

Some commands provide a list of options and arguments with the `-h`/`--help` option. If this list follows a certain syntax, you can get a decent completion working with

```
% compdef _gnu_generic <command>
```

One example on macOS, where this has decent results is the `xed` command which opens a file or folder in Xcode.

But for best results, you will often have to build the description yourself. Unfortunately this is not a simple task. The syntax is meticulously, but also quite abstractly documented in the `zsh` documentation for the Completion System. I also found the 'howto' documentation in the `zsh-completions` repository very useful, as well as the '`zsh` Completion Style Guide.'

To avoid everyone re-inventing the wheel, I have started a repository on Github for macOS specific completion files. The page has the instructions on how to install them and I will welcome pull requests with contributions. Since I am just starting to learn this as well, I am sure there are improvements that can be made on the completions I have built so far and there are several commands where you can test your skills and build a new one.

I suggest the `#zsh` channel on the MacAdmins Slack for discussion.

Next

In the next post in this series, we will discuss how to configure `zsh`'s command line prompt.

Moving to zsh, part 6 – Customizing the zsh Prompt

Apple has announced that in macOS 10.15 Catalina the default shell will be `zsh`.

In this series, I will document my experiences moving `bash` settings, configurations, and scripts over to `zsh`.

- Part 1: Moving to zsh
- Part 2: Configuration Files
- Part 3: Shell Options
- Part 4: Aliases and Functions
- Part 5: Completions
- Part 6: Customizing the `zsh` Prompt *(this article)*
- Part 7: Miscellanea
- Part 8: Scripting `zsh`

> *This series has grown into a book: reworked and expanded with more detail and topics. Like my other books, I plan to update and add to it after release as well, keeping it relevant and useful. You can order it on the Apple Books Store now.*

As I have mentioned in the earlier posts, I am aware that there are many solutions out there that give you a pre-configured 'shortcut' into lots of `zsh` goodness. But I am interested in learning this the 'hard way' without shortcuts. Call me old-fashioned. ("Uphill! In the snow! Both ways!")

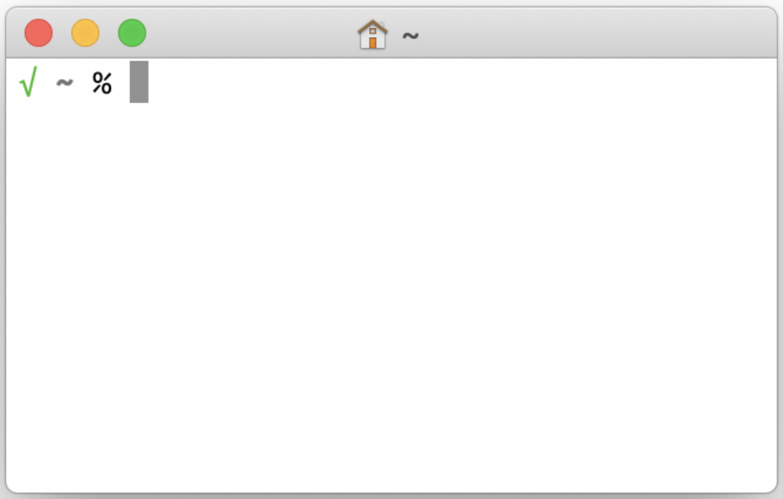
The default `bash` prompt on macOS is quite elaborate. It shows the username, the hostname, *and* the current directory.

```
Calypso:~ armin$
```

On the other hand, the default bash prompt doesn't show the previous command's exit code, a piece of information I find very useful. I have written before how I re-configured my `bash` prompt to have the information I want:

- Minimal Terminal Prompt
- Show Exit Code in your bash Prompt

Of course, I wanted to recreate the same experience in `zsh`.



```
√ ~ %
```

The only (visual) difference to my `bash` prompt is the `%` instead of the `$`.

> *Note: creating a file `~/.hushlogin` will suppress the status message at the start of each Terminal session in `zsh` as well as in `bash` (or any other shell).*

Basic Prompt Configuration

The basic `zsh` prompt configuration works similar to `bash`, even though it uses a different syntax. The different placeholders are described in detail in the `zsh` manual.

`zsh` uses the same shell variable `PS1` to store the default prompt. However, the variable names `PROMPT` and `prompt` are synonyms for `PS1` and you will see either of those three being used in various examples. I am going to use `PROMPT`.

The default prompt in `zsh` is `%m%#`. The `%m` shows the first element of the hostname, the `%#` shows a `#` when the current prompt has super-user privileges (e.g. after a `sudo -s`) and otherwise the `%` symbol (the default `zsh` prompt symbol).

The `zsh` default prompt is far shorter than the `bash` default, but even less useful. Since I work on the local system most of the time, the hostname bears no useful information, and repeating it every line is superfluous.

> *Note: you can argue that the hostname in the prompt is useful when you frequently have multiple terminal windows open to different hosts. This is true, but then the prompt is defined by the* remote *shell and its configuration files on the* remote *host. In your configuration file, you can test if the `SSH_CLIENT` variable is set and show a different prompt for remote sessions. There are more ways of showing the host in remote shell sessions, for example in the Terminal window title bar or with different window background colors.*

In our first iteration, I want to show the current working directory instead of the hostname. When you look through the list of prompt placeholders in the zsh documentation, you find `%d`, `%/`, and `%~`. The first two do exactly the same. The last substitution will display a path that starts with the user's home directory with the `~`, so it will shorten `/Users/armin/Projects/` to `~/Projects`.

> *Note: in the end you want to set your `PROMPT` variable in the `.zshrc` file, so it will take effect in all your `zsh` sessions. For testing, however, you can just change the `PROMPT` variable in the interactive shell. This will give you immediate feedback, how your current setup works.*

```
% PROMPT='%/ %# '
/Users/armin/Projects/dotfiles/zshfunctions %

% PROMPT='%~ %# '
~/Projects/dotfiles/zshfunctions %
```

Note the trailing space in the prompt string, to separate the final `%` or `#` from the command entry.

I prefer the shorter output of the `%~` option, but it can still be quite long, depending on your working directory. `zsh` has a trick for this: when you insert a number `n` between the `%` and the `~`, then only the last `n` elements of the path will be shown:

```
% PROMPT='%2~ %# '
dotfiles/zshfunctions %
```

When you do `%1~` it will show only the name of the working directory or `~` if it is the home directory. (This also works with `%/`, e.g. `%2/`.)

Adding Color

Adding a bit of color or shades of gray to the prompt can make it more readable. In `bash` you need cryptic escape codes to switch the colors. `zsh` provides an easier way. To turn the directory in the path blue, you can use:

```
PROMPT='%F{blue}%1~%f %# '
```

The `F` stands for 'Foreground color.' `zsh` understands the colors `black`, `red`, `green`, `yellow`, `blue`, `magenta`, `cyan` and `white`. `%F` or `%f` resets to the default text color. Furthermore, Terminal.app represents itself as a 256-color terminal to the shell. You can verify this with

```
% echo $TERM
xterm-256color
```

You can access the 256 color pallet with `%F{0}` through `%F{255}`. There are tables showing which number maps to which color:

- 256 Colors – Cheat Sheet – Xterm, HEX, RGB, HSL
- 256 Terminal colors and their 24bit equivalent (or similar)

So, since I want a dark gray for my current working dir in my prompt, I chose `240`, I also set it to bold with the `%B` code:

```
PROMPT='%B%F{240}%1~%f%b %# '
```

You can find a detailed list of the codes for visual effects in the documentation.

Dynamic Prompt

I wrote an entire post on how to get `bash` to show the color-coded exit code of the last command. As it turns out, this is *much* easier in `zsh`.

One of the prompt codes provides a 'ternary conditional,' which means it will show one of two expressions, depending on a condition. There are several conditions you can use. Once again the details can be found in the documentation.

There is one condition for the previous commands exit code:

```
%(?.<success expression>.<failure expression>)
```

This expression will use the `<success expression>` when the previous command exited successfully (exit code zero) and `<failure expression>` when the previous command failed (non-zero exit code). So it is quite easy to build an conditional prompt:

```
% PROMPT='%(?.√.?%?) %1~ %# '
√ ~ % false
?1 ~ %
```

You can get the √ character with option-V on the US or international macOS keyboard layout. The last part of the ternary `?%?` looks confusing. The first `?` will print a literal question mark, and the second part `%?` will be replaced with previous command's exit code.

You can add colors in the ternary expression as well:

```
PROMPT='%(?.%F{green}√.%F{red}?%?)%f %B%F{240}%1~%f%b %# '
```

Another interesting conditional code is `!` which returns whether the shell is privileged (i.e. running as root) or not. This allows us to change the default prompt symbol from `%` to something else, while maintaining the warning functionality when running as root:

```
% PROMPT='%1~ %(!.#.>) '
~ > sudo -s
~ # exit
~ >
```

Complete Prompt

Here is the complete prompt we assembled, with all the parts explained:

```
PROMPT='%(?.%F{green}√.%F{red}?%?)%f %B%F{240}%1~%f%b %# '
```

`%(?.√.?%?)`	if return code `?` is 0, show √, else show `?%?`
`%?`	exit code of previous command
`%1~`	current working dir, shortening home to `~`, show only last `1` element
`%#`	`#` with root privileges, `%` otherwise
`%B` `%b`	start/stop bold
`%F{...}`	text (foreground) color, see table
`%f`	reset to default textcolor

Right Sided Prompt

`zsh` also offers a right sided prompt. It uses the same placeholders as the 'normal' prompt. Use the `RPROMPT` variable to set the right side prompt:

```
% RPROMPT='%*'
√ zshfunctions %                    11:02:55
```

`zsh` will automatically hide the right prompt when the cursor reaches it when typing a long command. You can use all the other substitutions from the left side prompt, including colors and other visual markers in the right side prompt.

Git Integration

`zsh` includes some basic integration for version control systems. Once again there is a voluminous, but hard to understand description of it in the documentation.

I found a better, more specific example in the 'Pro git' documentation. This example will show the current branch on the right side prompt.

I have changed the example to include the repo name and the branch, and to change the color.

```
autoload -Uz vcs_info
precmd_vcs_info() { vcs_info }
precmd_functions+=( precmd_vcs_info )
setopt prompt_subst
RPROMPT=\$vcs_info_msg_0_
zstyle ':vcs_info:git:*' formats '%F{240}(%b)%r%f'
zstyle ':vcs_info:*' enable git
```

In this case `%b` and `%r` are placeholders for the VCS (version control system) system for the branch and the repository name.

There are `git` prompt solutions other than the built-in module, which deliver more information. There is a script in the `git` repository, and many of the larger `zsh` theme projects, such as 'oh-my-zsh' and 'prezto' have all kinds of git status widgets or modules or themes or what ever they call them.

Summary

You can spend (or waste) a lot of time on fine-tuning your prompt. Whether these modifications really improve your productivity is a matter of opinion.

In the next post, we will cover some miscellaneous odds and ends that haven't yet really fit into any of preceding posts.

Moving to zsh – part 7: Miscellanea

Apple has announced that in macOS 10.15 Catalina the default shell will be `zsh`.

In this series, I will document my experiences moving `bash` settings, configurations, and scripts over to `zsh`.

- Part 1: Moving to zsh
- Part 2: Configuration Files
- Part 3: Shell Options
- Part 4: Aliases and Functions
- Part 5: Completions
- Part 6: Customizing the `zsh` Prompt
- Part 7: Miscellanea (*this article*)
- Part 8: Scripting `zsh`

> *This series has grown into a book: reworked and expanded with more detail and topics. Like my other books, I plan to update and add to it after release as well, keeping it relevant and useful. You can order it on the Apple Books Store now.*

As I have mentioned in the earlier posts, I am aware that there are many solutions out there that give you a pre-configured 'shortcut' into lots of `zsh` goodness. But I am interested in learning this the 'hard way' without shortcuts. Call me old-fashioned. ("Uphill! In the snow! Both ways!")

We have covered the general aspects of configuring your `zsh` environment and enabling some of its features to make your work more productive. However, there are `zsh` features that didn't quite fit in earlier posts, but also don't warrant a post of their own. So I am gathering them here.

multiIO

Terminal commands can take input from a file or a previous command (`stdin`) and have two different outputs: `stdout` and `stderr`. In `bash` you can redirect each of these to a single other destination.

For example, you can redirect the output of a command to a file:

```
% system_profiler SPHardwareDataType >hardwareinfo.txt
```

In `zsh` you can redirect to (and from) multiple sources. In the simplest form you can write the output to two files:

```
% system_profiler SPHardwareDataType >hardwareinfo.txt
>computerinfo.txt
```

This is of course not a very realistic case. Since the pipe `|` is a form of re-direction, you can combine output to a file with a pipe:

```
% system_profiler SPHardwareDataType >hardwareprofile.txt | cat
```

Instead of piping to cat, you can also redirect to `stdout` (or `&1`) *as well as* to a file:

```
% system_profiler SPHardwareDataType >&1 >hardwareprofile.txt
```

Note that the order of doing this is important. The construct `>file.txt >&1` would redirect the output to `file.txt` and then redirect the output *again* to where `stdout` or `1` is going, so it would be redundant.

When combined with pipes and other commands multiIO can become very useful:

```
% system_profiler SPHardwareDataType >hardwareprofile.txt | awk
'/Serial Number/ { print $4 }' >&1 >serialnumber.txt
```

You can use multiIO for input as well:

```
% sort </usr/share/calendar/calendar.freebsd
</usr/share/calendar/calendar.computer
```

And while this not directly related, but somewhat close, in `zsh`, this

```
% <hardwareinfo.txt
```

is equivalent to `more hardwareinfo.txt`.

Recursive Globbing with **

You can use the `**` to denote an arbitrary string that can span multiple directories in a path.

For example:

```
% echo Library/Preferences/**/com.apple.screensaver.*plist
Library/Preferences/ByHost/com.apple.screensaver.BBCCDDEE-AABB-
CCDD-ABCD-00AABBCCDDEE.plist
Library/Preferences/com.apple.screensaver.plist
```

In this case the `**` matches nothing as well as `/ByHost/`.

Note: when used on large folder structures this glob can take a while. So use with care.

Connected array Variables

We already encountered the `fpath` variable in earlier posts. You can see its contents with the echo command:

```
% echo $fpath
/Users/armin/Projects/mac-zsh-completions/completions/
/Users/armin/Projects/dotfiles/zshfunctions
/usr/local/share/zsh/site-functions /usr/share/zsh/site-functions
/usr/share/zsh/5.3/functions
```

Interestingly enough, `zsh` also has an `FPATH` variable, which is a colon-separated list of directories:

```
% echo $FPATH
/Users/armin/Projects/mac-zsh-
completions/completions/:/Users/armin/Projects/dotfiles/zshfuncti
```

```
ons:/usr/local/share/zsh/site-functions:/usr/share/zsh/site-
functions:/usr/share/zsh/5.3/functions
```

Since the `fpath` variable is an array, I only changed the `fpath` variable in my `zshrc`.I never set or changed the `FPATH`, yet it reflects the changes made to the `fpath` variable.

When you see the type of both variables, you get an idea that something is going on:

```
% echo ${(t)fpath}
array-special
% echo ${(t)FPATH}
scalar-special
```

The `fpath` and `FPATH` are connected in `zsh`. Changes to one affect the other. This allows use of more flexible and powerful array operations through the `fpath` 'aspect' of the value, but also provides compatibility to tools that expect the traditional colon-separated format in `FPATH`.

You will not be surprised to hear that `zsh` uses the same 'magic' with the `PATH` variable and its array counterpart `path`.

This means that you can continue to use `path_helper` to get your `PATH` from the files in `/etc/paths` and `/etc/paths.d`. (Well, you don't have to, because on macOS this is done for all users in `/etc/zprofile`.) But then you can manipulate the `path` variable with array functions, like:

```
path+=~/bin
```

You get the useful aspects of both syntaxes.

Suffix Aliases

I learnt this one *after* writing the aliases part.

Suffix aliases take effect on the last part of a path, so usually the file extension. A suffix alias will assign a command to use when you just type a file path in the command line.

For example, you can a suffix alias for the `txt` file extension:

```
alias -s txt="open -t"
```

When you then type a path ending with `.txt` and no command, `zsh` will execute `open -t /path/to/file.txt`.

The `open -t` command opens a file in the default application set for the `txt` file extension in Finder. You probably want to set the suffix alias to `bbedit` or `atom` or something like that rather than `open -t`.

You can use other command line tools for the suffix alias:

```
alias -s log="tail -f"
```

Then, typing `/var/log/install.log` will show the last lines of that file and update the output when the file changes. If you prefer the graphical user interface, you can use the `open -a` command to assign suffix aliases to applications:

```
alias -s log="open -a Console"
```

You can even create a suffix alias using a different alias:

```
alias pacifist="open -a Pacifist"
alias -s pkg=pacifist
```

Together with the AutoCD option, this can improve your application-shell interactions a lot.

Bindkey for History Search

Most of the keyboard shortcuts in `zsh` work the same way as they do in `bash`. I have found one change that has proven quite useful:

```
^[[A' up-line-or-search # up arrow bindkey

^[[B' down-line-or-search # down arrow
```

These two commands will change the behavior of the up and down arrow keys from just switching to the previous command, to `searching`. This means that when you start typing a command and then hit the up key, rather than just replacing what you already typed with the previous command, the shell will instead *search* for the latest command in the history starting with what you already typed.

There are *many* commands or 'widgets' you can assign to keystrokes with the `bindkey` command. You can find a list of default 'widgets' in the documentation.

Conclusion

This concludes the part of the series about configuring `zsh`. When I set out I wanted to recreate the environment I had built in `bash`. Along the way I found a few features in `zsh` that seemed worth adding to my toolkit.

After nearly two months of working in `zsh`, there are already some features I would miss terribly when switching back to `bash` or a plain, unconfigured `zsh`. Most important is the powerful tab-completion. But features like AutoCD, MultiIO, and flexible aliases, are useful tools as well.

The dynamic loading of functions from files in the `fpath` was initially confusing, but it allows configurations and functions to be split out into their own, which simplifies "modularizing" and sharing.

In the next (and last) post, I will cover the changes when scripting with `zsh` vs `bash`.