

Minimal safe Bash script template

Published by **Maciej Radzikowski** on December 14, 2020

Contents

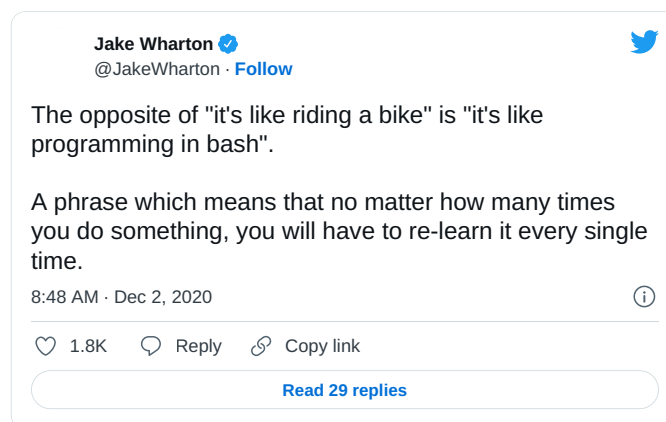
- [Why scripting in Bash?](#)
- [Bash script template](#)
 - [Choose Bash](#)
 - [Fail fast](#)
 - [Get the location](#)
 - [Try to clean up](#)
 - [Display helpful help](#)
 - [Print nice messages](#)
 - [Parse any parameters](#)
- [Using the template](#)
- [Portability](#)
- [Further reading](#)
- [Closing notes](#)

Bash scripts. Almost anyone needs to write one sooner or later. Almost no one says “yeah, I love writing them”. And that’s why almost everyone is putting low attention while writing them.

I won’t try to make you a Bash expert (since I’m not a one either), but I will show you a minimal template that will make your scripts safer. You don’t need to thank me, your future self will thank you.

Why scripting in Bash?

The best summary of Bash scripting appeared recently on my Twitter feed:



But Bash has something in common with another widely beloved language. Just like JavaScript, it won’t go away easily. While we can hope that Bash won’t become the main language for literally everything, it’s always somewhere near.

Bash **inherited the shell throne** and can be found on almost every Linux, including Docker images. And this is the environment in which most of the backend runs. So if you need to script the server application startup, a CI/CD step, or integration test run, Bash is there for you.

To glue few commands together, pass output from one to another, and just start some executable, Bash is the easiest and most native solution. While it makes perfect sense to write bigger, more complicated scripts in other languages, you can’t expect to have Python, Ruby, fish, or whatever another interpreter you believe

We use cookies to ensure that we give you the best experience on our website. If you continue to use this site I will assume that you are happy with it.

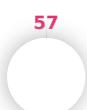
OK

without further ado, here it is.

```

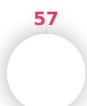
1  #!/usr/bin/env bash
2
3  set -Eeuo pipefail
4  trap cleanup SIGINT SIGTERM ERR EXIT
5
6  script_dir=$(cd "$(dirname "${BASH_SOURCE[0]}")" && /dev/null && pwd -P)
7
8  usage() {
9      cat << EOF # remove the space between << and EOF, this is due to web plugin issue
10     Usage: $(basename "${BASH_SOURCE[0]}") [-h] [-v] [-f] -p param_value arg1 [arg2...]
11
12     Script description here.
13
14     Available options:
15
16     -h, --help      Print this help and exit
17     -v, --verbose   Print script debug info
18     -f, --flag      Some flag description
19     -p, --param     Some param description
20     EOF
21     exit
22 }
23
24 cleanup() {
25     trap - SIGINT SIGTERM ERR EXIT
26     # script cleanup here
27 }
28
29 setup_colors() {
30     if [[ -t 2 ]] && [[ -z "${NO_COLOR-}" ]] && [[ "${TERM-}" != "dumb" ]]; then
31         NOFORMAT='\033[0m' RED='\033[0;31m' GREEN='\033[0;32m' ORANGE='\033[0;33m' BLUE='\033[0;34m' PURPLE='\033[0;35m' CYAN='\033[0;36m' YELLOW='\033[1;33m'
32     else
33         NOFORMAT='' RED='' GREEN='' ORANGE='' BLUE='' PURPLE='' CYAN='' YELLOW=''
34     fi
35 }
36
37 msg() {
38     echo >&2 -e "${1-}"
39 }
40
41 die() {
42     local msg=$1
43     local code=${2-1} # default exit status 1
44     msg "$msg"
45     exit "$code"
46 }
47
48 parse_params() {
49     # default values of variables set from params
50     flag=0
51     param=''
52
53     while :; do
54         case "${1-}" in
55             -h | --help) usage ;;
56             -v | --verbose) set -x ;;
57             --no-color) NO_COLOR=1 ;;
58             -f | --flag) flag=1 ;; # example flag
59             -p | --param) # example named parameter
60                 param="${2-}"
61                 shift
62                 ;;
63             -?*) die "Unknown option: $1" ;;
64             *) break ;;
65             esac
66         shift
67     done
68
69     We use cookies to ensure that we give you the best experience on our website. If you continue to use this site I will assume that you are happy with it.
70
71     OK

```



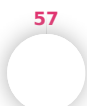
We use cookies to ensure that we give you the best experience on our website. If you continue to use this site I will assume that you are happy with it.

OK



We use cookies to ensure that we give you the best experience on our website. If you continue to use this site I will assume that you are happy with it.

OK



We use cookies to ensure that we give you the best experience on our website. If you continue to use this site I will assume that you are happy with it.

OK



We use cookies to ensure that we give you the best experience on our website. If you continue to use this site I will assume that you are happy with it.

OK

```
# check required params and arguments
[[ -z "${param-}" ]] && die "Missing required parameter: param"
[[ ${#args[@]} -eq 0 ]] && die "Missing script arguments"

return 0
}

parse_params "$@"
setup_colors

# script logic here

msg "${RED}Read parameters:${NOFORMAT}"
msg "- flag: ${flag}"
msg "- param: ${param}"
msg "- arguments: ${args[*]}"
```

The idea was to not make it too long. I don't want to scroll 500 lines to the script logic. At the same time, I want some strong foundations for any script. But Bash is not making this easy, lacking any form of dependencies management.

One solution would be to have a separate script with all the boilerplate and utility functions and execute it at the beginning. The downside would be to have to always attach this second file everywhere, losing the "simple Bash script" idea along the way. So I decided to put in the template only what I consider to be a minimum to keep it possible short.

Now let's look at it in more detail.

Choose Bash

```
1 #!/usr/bin/env bash
```

Script traditionally starts with a shebang. For the **best compatibility**, it references `/usr/bin/env`, not the `/bin/bash` directly. Although, if you read comments in the linked StackOverflow question, even this can fail sometimes.

Fail fast

```
1 set -Eeuo pipefail
```

57

We use cookies to ensure that we give you the best experience on our website. If you continue to use this site I will assume that you are happy with it.

[OK](#)

What will happen, if the `backups` directory does not exist? Exactly, you will get an error message in the console, but before you will be able to react, the file will be already removed by the second command.

For details on what options exactly `set -Euo pipefail` changes and how they will protect you, I refer you to the [article I have in my bookmarks for a few years now](#).

Although you should know that there are some [arguments against setting those options](#).

Get the location

```
1 script_dir=$(cd "$(dirname "${BASH_SOURCE[0]}")" &&/dev/null && pwd -P)
```

This line **does its best** to define the script's location directory, ~~and then we cd to it. Why?~~

Often our scripts are operating on paths relative to the script location, copying files and executing commands, assuming the script directory is also a working directory. And it is, as long as we execute the script from its directory.

But if, let's say, our CI config executes script like this:

```
1 /opt/ci/project/script.sh
```

then our script is operating not in project dir, but some completely different workdir of our CI tool. We can fix it, by going to the directory before executing the script:

```
1 cd /opt/ci/project && ./script.sh
```

But it's much nicer to solve this on the script side. So, if the script reads some file or executes another program from the same directory, call it like this:

```
1 cat "$script_dir/my_file"
```

At the same time, the script does not change the workdir location. If the script is executed from some other directory and the user provides a relative path to some file, we will still be able to read it.

Try to clean up

```
1 trap cleanup SIGINT SIGTERM ERR EXIT
2
3 cleanup() {
4     trap - SIGINT SIGTERM ERR EXIT
5     # script cleanup here
6 }
```

Think about the `trap` like of a `finally` block for the script. At the end of the script – normal, caused by an error or an external signal – the `cleanup()` function will be executed. This is a place where you can, for example, try to remove all temporary files created by the script.

Just remember that the `cleanup()` can be called not only at the end but as well having the script done any part of the work. Not necessarily all the resources you try to cleanup will exist.

Display helpful help

57

```
1 usage() {
2     cat << EOF # remove the space between << and EOF, this is due to web plugin issue
3
4     We use cookies to ensure that we give you the best experience on our website. If you continue to use this site I will assume that you are happy with it.
5
6     OK
7 }
```



```
exit
}
```

Having the `usage()` relatively close to the top of the script, it will act in two ways:

- to **display help** for someone who does not know all the options and does not want to go over the whole script to discover them,
- as a **minimal documentation** when someone modifies the script (for example you, 2 weeks later, not even remembering writing it in the first place).

I don't argue to document every function here. But a short, nice script usage message is a required minimum.

Print nice messages

```
1 setup_colors() {
2   if [[ -t 2 ]] && [[ -z "${NO_COLOR-}" ]] && [[ "${TERM-}" != "dumb" ]]; then
3     NOFORMAT='\033[0m' RED='\033[0;31m' GREEN='\033[0;32m' ORANGE='\033[0;33m' BLUE='\033[0;34m' PURPLE='\033[0;35m' CYAN='\033[0;36m' YELLOW='\033[1;33m'
4   else
5     NOFORMAT='' RED='' GREEN='' ORANGE='' BLUE='' PURPLE='' CYAN='' YELLOW=''
6   fi
7 }
8
9 msg() {
10  echo >&2 -e "${1-}"
11 }
```

Firstly, remove the `setup_colors()` function if you don't want to use colors in text anyway. I keep it because I know I would use colors more often if I wouldn't have to google codes for them every time.

Secondly, those **colors are meant to be used with the `msg()` function only**, not with the `echo` command.

The **`msg()` function is meant to be used to print everything that is not a script output**. This includes all logs and messages, not only the errors. Citing the great [12 Factor CLI Apps](#) article:

In short: stdout is for output, stderr is for messaging.

Jeff Dickey, who [knows a little about building CLI apps](#)

That's why in most cases you shouldn't use colors for `stdout` anyway.

Messages printed with `msg()` are sent to `stderr` stream and support special sequences, like colors. And colors are disabled anyway if the `stderr` output is not an interactive terminal or **one of the standard parameters** is passed.

Usage:

```
1 msg "This is a ${RED}very important${NOFORMAT} message, but not a script output value!"
```

To check how it behaves when the `stderr` is not an interactive terminal, add a line like above to the script. Then execute it redirecting `stderr` to `stdout` and piping it to `cat`. Pipe operation makes the output no longer being sent directly to the terminal, but to the next command, so the colors should be disabled now.

```
$ ./test.sh 2>&1 | cat
This is a very important message, but not a script output value!
```

Parse any parameters 57

```
1 parse_params() {
2   # default values of variables not from env
3
4   We use cookies to ensure that we give you the best experience on our website. If you continue to use this site I will assume that you are happy with it.
5   OK
```

```

-v | --verbose) set -x ;;
--no-color) NO_COLOR=1 ;;
-f | --flag) flag=1 ;; # example flag
-p | --param) # example named parameter
    param="${2-}"
    shift
    ;;
-?*) die "Unknown option: $1" ;;
*) break ;;
esac
shift
done

args=("$@")

# check required params and arguments
[[ -z "${param-}" ]] && die "Missing required parameter: param"
[[ $#args[@] -eq 0 ]] && die "Missing script arguments"

return 0
}

```

If there is anything that makes sense to be parametrized in the script, I usually do that. Even if the script is used only in a single place. It makes it easier to copy and reuse it, which often happens sooner than later. Also, even if something needs to be hardcoded, usually there is a better place for that on a higher level than the Bash script.

There are **three main types of CLI parameters** – flags, named parameters, and positional arguments. The `parse_params()` function supports them all.

The only one of the common parameter patterns, that is not handled here, is **concatenated multiple single-letter flags**. To be able to pass two flags as `-ab`, instead of `-a -b`, some additional code would be needed.

The `while` loop is a manual way of parsing parameters. In every other language you should use one of the **built-in parsers** or **available libraries**, but, well, this is Bash.

An example flag (`-f`) and named parameter (`-p`) are in the template. Just change or copy them to add other params. And do not forget to update the `usage()` afterward.

The important thing here, usually missing when you just take the first google result for Bash arguments parsing, is **throwing an error on an unknown option**. The fact the script received an unknown option means the user wanted it to do something that the script is unable to fulfill. So user expectations and script behavior may be quite different. It's better to prevent execution altogether before something bad will happen.

There are two alternatives for parsing parameters in Bash. It's `getopt` and `getopts`. There are **arguments both for and against** using them. I found those tools not best, since by default `getopt` on macOS is **behaving completely differently**, and `getopts` does not support long parameters (like `--help`).

Using the template

Just copy-paste it, like most of the code you find on the internet.

Well, actually, it was quite honest advice. With Bash, there is no universal `npm install` equivalent.

After you copy it, you only need to change 4 things:

- `usage()` text with script description
- `cleanup()` content
- parameters in `parse_params()` – leave the `--help` and `--no-color`, but replace the example ones: `-f` and `-p`
- actual script logic

Portability

57

I tested the template on MacOS (with default, archaic Bash 3.2) and several Docker images: Debian, Ubuntu, CentOS, Amazon Linux, Fedora. It works.

We use cookies to ensure that we give you the best experience on our website. If you continue to use this site I will assume that you are happy with it.

OK

Further reading

When creating a CLI script, in Bash or any better other language, there are some universal rules. Those resources will guide you on how to make your small scripts and big CLI apps reliable:

- [Command Line Interface Guidelines](#)
- [12 Factor CLI Apps](#)
- [Command line arguments anatomy explained with examples](#)

Closing notes

I'm not the first and not the last to create a Bash script template. One good alternative is [this project](#), although a little bit too big for my everyday needs. After all, I try to keep Bash scripts as small (and rare) as possible.

When writing Bash scripts, use the IDE that supports [ShellCheck](#) linter, like JetBrains IDEs. It will prevent you from doing [a bunch of things](#) that can backfire on you.

My Bash script template is also available as GitHub Gist (under [MIT license](#)):

[🔗 script-template.sh](#)

If you found any problems with the template, or you think something important is missing – let me know in the comments.

Update 2020-12-15

After a lot of comments here, on [Reddit](#), and [HackerNews](#), I did some improvements to the template. See revisions history in [gist](#).

Update 2020-12-16

Link to this post reached the [front page of Hacker News \(#7\)](#). This was completely unexpected.

Did you like this article? Join the newsletter for updates about new content from me.

 First name

 Email address

SUBSCRIBE

You can also [follow me on](#) [🐦 Twitter](#) or [subscribe via](#) [📡 RSS](#)

Categories: **PROGRAMMING**

Tags: [bash](#) [cli](#) [level:beginner](#) [shell](#)

✉ Subscribe ▼



57

Join the discussion

B *I* U       

We use cookies to ensure that we give you the best experience on our website. If you continue to use this site I will assume that you are happy with it.

OK

Nice template!

One thing though: `cd`-ing to the script dir is going to break for any script which accept relative paths as arguments. This can be avoided by wrapping the `cd` call with a `pushd $PWD > /dev/null` and `popd > /dev/null`.

👍 6 🗨️ Reply



Maciej Radzikowski 1 year ago

🗨️ Reply to [Aurélien Gâteau](#)

Author

Thank you! That's a quite nice solution for relative paths, I didn't know it. Let me test it out and update the template 😊

👍 4 🗨️ Reply



Stephen Rees 1 year ago

🗨️ Reply to [Aurélien Gâteau](#)

In a similar vein, I was thinking it would be polite to return the user to the directory they were in prior to running the script, and the push and pop would be useful for that.

👍 1 🗨️ Reply



John Kugelman 1 year ago

🗨️ Reply to [Stephen Rees](#)

No worries there. Scripts can't change the working directory of the parent shell.

👍 2 🗨️ Reply



Roeniss Moon 1 year ago

🗨️ Reply to [John Kugelman](#)

It's not always. It depends on how you execute the script.

👍 0 🗨️ Reply



Arne 1 year ago

🗨️ Reply to [Aurélien Gâteau](#)

You can also avoid `cd`-ing altogether using `realpath`.

```
script_dir=$(realpath "$(dirname "${BASH_SOURCE[0]}")")
```

👍 3 🗨️ Reply



John Kugelman 1 year ago

You can avoid the “scroll all the way to the bottom” problem by adding a `main()` function. This lets you put the main script logic near the top and keeps the logical flow top-to-bottom.

```
main() {
  # script logic
}
```

...

```
# bottom of file
main "$@"
```

I do this in all of my scripts.

👍 8 🗨️ Reply

57



Peter Forret 1 year ago

🗨️ Reply to [John Kugelman](#)

I had the same remark when I read this!



Reply to [John Kugelmann](#)

Author

Fair point. But to have nice top-to-bottom overview of what's happening in the script, I would insist on having functions (from top):

- usage
- parse_params
- main

This allows to read script and get to know what it does without jumping around. First you read docs, then params, then main. So only thing to put lower would be cleanup and messaging functions, which are not so long.

But I agree that it is an improvement that can be applied here.

👍 1 🗨️ Reply



Peter Forret 1 year ago

Reply to [Maciej Radzikowski](#)

I use a kind of CSV format for flags/options/parameters, and usage/option initialisation/option parsing is automatically derived from that.

Cf: <https://github.com/pforret/bashew/blob/master/template/normal.sh>

👍 0 🗨️ Reply



Paul 1 year ago

Thanks for sharing this, and for trying to raise the bar on Bash script quality. We've all had "throwaway" scripts that didn't get thrown away.

Since you were so conscientious about testing across OSs, allow me to suggest you add FreeBSD to the list!

👍 1 🗨️ Reply



Kirk Bater 1 year ago

I have a similar scaffold set up. I like your colors part, I'm probably going to steal that 😊

With that said, one small usability improvement you could add is instead of `break` ing on the arg parsing loop, push those into another var. By having the `break` there instead of pushing to another var, you don't allow your scripts users to add flags after the positional args, which is annoying when trying to debug why the script isn't working by just adding `-v` to the end of the command, you have to then put the flag towards the front of the command. Example: in `myscript.sh -f --flag2 myarg1 myarg2 -v` the `-v` would be put into the args variable instead of being parsed as a flag.

So it would be something like this:

```
args=()
while :; do
  case "${1-}" in
    -h | --help)
      usage
      ;;
    *)
      args+=($1)
      ;;
  esac
  shift
done
```

👍 0 🗨️ Reply



Petter 1 year ago

The template contains both `NO_COLOR` and `NOCOLOR`. They do different things. This can be a bit confusing.

👍 0 🗨️ Reply



Maciej Radzikowski 1 year ago

We use cookies to ensure that we give you the best experience on our website. If you continue to use this site I will assume that you are happy with it.

OK

Savvas • 1 year ago

The pipefail is good, but has a bad side effect. If the originating command does not produce any output, the pipefail will consider it as a failed one. Case in point, when checking for files, according to same pattern and want to count them or even the output of the pipe would be used elsewhere. If none, the pipefail will be considered it failed.

I did also some kind of templating, and the approach I used is to have all these code in one file, and included in the other scripts. In fact, in order to make it easier, instead of loading again and again the same code, I use a code like this.

I have a function in my template code, name `exit_code`. If that does not exist, then try to find that library (named `load.functions` in my case) in the path and home directory (you may add whatever you like) and then include it with the source command (or mostly known as ,)

```
if [[ type -t exit_code"" != 'function' ]]; then
    DIR="$HOME $(echo $PATH | tr ':' ' ')"
    mylib=$(find "$DIR" -type f -name load.functions -printf '%T@ %p\n' 2>/dev/null | sort -n | tail -1 | cut -f2- --delimiter=' ')
    [[ -z ${mylib} || ! -f ${mylib} ]] && { echo "cannot load functions. abort"; exit 1; }
    source "${mylib}"
else
    echo "cannot load functions. abort"
    exit 1
fi
```

👍 0 🗨️ 0 ➡️ Reply



Author

Maciej Radzikowski • 1 year ago

🗨️ Reply to [Savvas](#)

Hey, this approach with separate files and even auto-loading is nice. The drawback it's only for own workstation. My goal here is to have something relatively small that will work in all kind of environments.

👍 1 🗨️ 0 ➡️ Reply

**Savvas** • 1 year ago

🗨️ Reply to [Maciej Radzikowski](#)

except if you distribute the code, you distribute both files. But this makes it easier to have real good and possibly several pages long, library code, elsewhere, and not pollute the other scripts.

👍 0 🗨️ 0 ➡️ Reply

**Brian K. White** • 1 year ago

🗨️ Reply to [Savvas](#)

He has an explicit design goal not to have multiple parts. It is a legit goal for the stated reasons. Yes it's a compromise. So is using Bash at all in the first place.

Improvements that violate the design goal are not improvements, they are some totally other answer to some totally other question.

👍 1 🗨️ 0 ➡️ Reply

**Benjamin HENRION** • 1 year ago

We should have a built-in bash library that simplifies a lot of functionality, like:

1. check that a file exist, like `check_file(filename)`
2. check that a directory exist, like `check_dir(dirname)`

👍 -1 🗨️ 0 ➡️ Reply

57

**Savvas** • 1 year ago

🗨️ Reply to [Benjamin HENRION](#)

```
[[ -f {filename} ]] && { code if true } || { code if false, file does not exist }
the same for directories instead of { filename }
```

That was a bad example, because the solution was a onliner.
Is there a good bash library with lots of useful functions?

👍 0 🗨️ Reply



Peter Forret 1 year ago

🗨️ Reply to Benjamin HENRION

Check out <https://github.com/pforret/bashew>

It's got option parsing, text output functions, hash, slugify, lower/uppercase, folder cleanup, ...

👍 0 🗨️ Reply



David 1 year ago

🗨️ Reply to Benjamin HENRION

Here's one that I wrote myself: <https://gitlab.com/methuselah-0/bash-coding-utils.sh>

👍 0 🗨️ Reply



Richard 1 year ago

Nice minimal template, thanks for posting it!

I had the same thoughts expressed in several other comments: NO_COLOR vs. NOCOLOR confusion, main() function wrapping, and quickly appending args on subsequent runs (i.e., add trailing -v) all seem like good improvements.

I would use --debug (and no -d) for set -x , which I think of as debugging by the script **author** (set -x is rather low-level output to an average user). I would use a --verbose

-v options for printing more task specific details to the script **user**. I wouldn't expose debugging with -d because such a common letter is likely to be desired for a task specific option.

👍 0 🗨️ Reply



Author

Maciej Radzikowski 1 year ago

🗨️ Reply to Richard

Thanks for the comment! I applied some changes and updated the template.

With verbose and debug – I see your point and agree in the case of bigger scripts and CLI apps.

Here I hope no one will build a big CLI app based on this template! I was thinking more about quite simple utility scripts that we all create to get things working.

👍 0 🗨️ Reply



Bruno Paulino 1 year ago

This is Gold! Thanks a lot for sharing this.

👍 0 🗨️ Reply



Brian K. White 1 year ago

script_dir=\${0%/*}

both basename and dirname are built-in in this way. This is just dirname.

For completeness, basename is \${0##*/}

👍 -1 🗨️ Reply



Jon Hell 1 year ago

getopts is also another way to handle the flags that are passed in a more structured way. You can specify what flags are possible and flag arguments can be separated by a space or directly after the flag. multiple flags can be applied with one single "-" to get compounding options (ex. rm -rf)

I think that is the standard way of handling flags in more complex scripts

👍 0 🗨️ Reply

57



Brian K. White 1 year ago


🗨️ Reply to Jon Hell


We use cookies to ensure that we give you the best experience on our website. If you continue to use this site I will assume that you are happy with it.

OK

the console, but before you will be able to react, the file will be already removed by the second command.

```
if [ -d ./backups ]; then mv important_file ./backups/; fi
```


 Last edited 1 year ago by Dan B

 0   Reply



Author

Maciej Radzikowski  1 year ago

 Reply to [Dan B](#)

Sure. It was just a simple example that could be fixed in several ways. The point is that every command can fail, and that can have more or less serious consequences on how the rest of the script will behave.

Handling possible failure of only the parts that can have an impact on the next commands is all good until you miss one spot. That's why I believe it's better to make Bash behave more like most other languages – something fails, the process is stopped.

 0   Reply



Dan B  1 year ago

 Reply to [Maciej Radzikowski](#)

>That's why I believe it's better to make Bash behave more like most other languages – something fails, the process is stopped.


I disagree because I take that to be re-inventing the wheel. In most of the shell scripts I've seen it's very common to see things like:

```
if [ -x /etc/rc.d/rc.httpd ]; then
    /etc/rc.d/rc.httpd start
fi
```

I guess it sounds dismissive or unsexy to say "this is the convention, just stick to this" but I mean it's just worked and continues to work without having to beat your shell script into submission to act like it's written in a more modern scripting language. That's why IMO, shell scripts are great for sys-admin type stuff like moving files around and setting up the environment but it's ok to reach for another scripting language when you want something more portable and does fancier things. Shell scripts can also run Ruby one-liners or other Python scripts too, you can use them as "wrappers" or "glue" for these other things

 0   Reply



Bob Smith  1 year ago


Am I right in thinking that msg goes to stderr, but die goes to stdout?
Seems like the wrong way round to me.

 0   Reply



Author

Maciej Radzikowski  1 year ago

 Reply to [Bob Smith](#)

No, why? die is using msg to print text, so it will go to stderr.

 0   Reply



Adrian  1 year ago

Hey, nice article, i really like it.

Only thing i don't agree with is the change directory part.

I would much rather instead of that have the script determine its parent directory, if at all needed.

57 This article comes to mind which solved this problem for me: <https://stackoverflow.com/questions/59895/how-to-get-the-source-directory-of-a-bash-script-from-within-the-script-itself>

 0   Reply

We use cookies to ensure that we give you the best experience on our website. If you continue to use this site I will assume that you are happy with it.

OK

jvh 1 year ago

The failure mode of `script_dir=...` deserves a little more explanation. My first reaction is always to add `|| exit` on anything that does a `cd` but that isn't needed here because you have `&& pwd -P`. Except that bash disables `-e` when evaluating a substitution inside a subshell (I have no idea why that seems like a good idea). So if the `cd` fails, it doesn't terminate the subshell but because you have `&& pwd -P` the exit status of the failing `cd` is the exit status of the subshell setting `script_dir` and so the entire assignment fails and the shell exits.

Unfortunately it exits silently because you're discarding `stderr` so when you're falling foul of the inevitable 😊 symlink race condition, the script just silently and mysteriously does nothing.

You're also in danger, I think, of Mysterious Bad Behaviour because you're trusting whatever `PATH` is set to. You're also trusting that any functions in the environment `export -f` don't do unexpected things. (That last one is quite difficult, if not impossible, to guard against.)

👍 0 🗨️ 1 ➡️ Reply

Maciej Radzikowski 1 year ago

🗨️ Reply to jvh

Author

Wow, that's a very in-deep analysis of that part. Thank you. You are right, having a script mysteriously exiting on one of the first lines like that would be amusing to debug.

👍 0 🗨️ 1 ➡️ Reply

William Pursell 1 year ago

Providing a link to some of the arguments against `errexit` is useful, but I think you are overlooking those arguments too glibly. In particular, I'll quote from the article you link help ensure people don't miss it. You should never use `set -e`:

`set -e` has changed behavior between bash versions, triggering fatal errors in one version, ignoring the same non-zero return value in another. In addition, whether a command returning non-zero causes a fatal error or not depends on the context it is run in. So in practice, you have to go over every command and decide how to handle their return value anyway. It's the only way to get reliable error handling. `set -e` adds nothing useful to the table.

See <http://mywiki.woledge.org/BashFAQ/105> for some examples of the weird cases you have to deal with.

📝 Last edited 1 year ago by William Pursell

👍 0 🗨️ 1 ➡️ Reply

Maciej Radzikowski 1 year ago

🗨️ Reply to William Pursell

Author

I start to have a feeling that posting

use "`set -e`" in your scripts!

on unix forum is like posting "tabs vs spaces" on any programming channel 😊

📝 Last edited 1 year ago by Maciej Radzikowski

👍 0 🗨️ 1 ➡️ Reply

Eric Nemchik 1 year ago

Very good stuff here. Some things I'll apply to my various projects. Much of what you're doing is similar to (albeit smaller than) a project I consider to be my greatest bash project <https://github.com/GhostWriters/DockSTARTer/blob/master/main.sh> which I made with very similar principals in mind mostly derived from resources I documented here <https://github.com/GhostWriters/DockSTARTer/blob/master/.github/CONTRIBUTING.md#shell-scripts>

I will very likely take a shot at getting somewhere down the middle between what you have and what I have, and I'll reach out if I come up with anything worth discussing.

Very happy to see you've written this!

👍 0 🗨️ 1 ➡️ Reply

```
function do_pf() {  
  
    MSG="$1"  
  
    printf '%s\n' "$MSG"  
  
}  
  
function do_npf() {  
  
    MSG="$1"  
  
    printf '%s' "$MSG"  
  
}
```

and if you want something like printed borders:

```
BLCHAR=' - '  
  
BORDER='80'  
  
  
function do_bl() {  
  
    BCHAR="$1"  
  
  
    if [[ "$BCHAR" == '' ]];  
  
    then  
  
        BCHAR="$BLCHAR"  
  
    fi  
  
  
    printf -v borderline '%*s' "$BORDER"  
  
    echo ${borderline// /$BCHAR}  
  
}
```

And for logging:

```
function do_log() {  
  
    LOGMSG="$1"  
  
    RUNLOGFILE="$RUNLOG"  
  
    LOGDATE=date "+%m-%d-%Y.%H%M.%N"  
  
    printf '%s\n' "$LOGDATE|$LOGMSG" >> "$RUNLOGFILE"
```



We use cookies to ensure that we give you the best experience on our website. If you continue to use this site I will assume that you are happy with it.

OK

```

do_log "${FUNCNAME[0]}"

do_pf "hello"

}

```

✎ Last edited 1 year ago by digi

👍 0 🗨️ ➡️ Reply

Jie Zheng 1 year ago

What a nice template here. It's really a style guide for bash.
May i recommend another functionality to be include in the template, that is daemon.
A daemon functionality is also minimal. Start a process in background and get the pid, shutdown a process using a pid file. This feature is needed for daily usage.

👍 0 🗨️ ➡️ Reply

Gert van den Berg 1 year ago

tput should be used instead of hard-coded colors – escape codes differ for different terminal types...

👍 0 🗨️ ➡️ Reply

Roeniss Moon 1 year ago

Hi Maciej. May I translate this post to my blog with the link to this post?

👍 0 🗨️ ➡️ Reply

Maciej Radzikowski 1 year ago

🗨️ Reply to [Roeniss Moon](#)

Yes, go ahead! Please share a link when you publish it.

👍 1 🗨️ ➡️ Reply

Roeniss Moon 1 year ago

🗨️ Reply to [Maciej Radzikowski](#)

<https://velog.io/@roeniss/간결하고-안전한-Bash-스크립트-템플릿>

here it is. actually this is my first translation for blog post. it was really fun and helpful to my understanding about script-world. Thank you again for giving me such a rare chance.

quick question : where my thumbnail came from? I can't remember when I send this to your blog lol

👍 0 🗨️ ➡️ Reply

Maciej Radzikowski 1 year ago

🗨️ Reply to [Roeniss Moon](#)

Great!

And thumbnail is fetched from <https://gravatar.com/>

👍 0 🗨️ ➡️ Reply

xshoji 1 year ago


Nice template!
I had make similar concept tool before too.
See my tool If you like lol.

<https://github.com/xshoji/bash-script-starter>

👍 0 🗨️ ➡️ Reply

Üllar 1 year ago

Thanks for the wonderful template; I've pretty much reverted all my projects to this in one form or another! A question though: what is the significance of the hvnhen in for example `case "${1-}"`


 Reply to Üllar

I did some digging as I couldn't leave it alone, and from the [Shell Parameter Expansion documentation](#) it's clear that "Omitting the colon results in a test only for a parameter that is unset. Put another way, if the colon is included, the operator tests for both parameter's existence and that its value is not null; if the colon is omitted, the operator tests only for existence."

Thus, when using the so called strict mode in Bash scripts, *not* adding the hyphen as I did, will cause the scripts to fail because of an unbound variable when no arguments are given to the script, but the hyphen without the colon will test whether the parameter is unset and if it is then sets to an empty string.

Please correct me if I'm wrong.


I'm still a bit confused about why the hyphen is necessary in `-?*)` ...

 Last edited 1 year ago by Üllar

 0   Reply



Author


Maciej Radzikowski  1 year ago Reply to Üllar

Hey. Sorry for late reply 😊

The `-u` flag ("nounset") makes bash exit with error if variable is not defined. If you change it to `case "${1}"` then you will get "line 54: 1: unbound variable" error. This is because the while loop goes over and over until the variable in `${1}` is empty. So we must make sure the bash will not throw an error in this situation.

The hyphen in the `-?*)` is a different story. There we are doing pattern matching, and this checks if the text starts with at least 1 hyphen. If so, we see that it is a flag or parameter that we did not expect.

 1   Reply

**Songmu**  1 year ago



When we trap a signal, we cannot know the type of signal received in the cleanup function, but we can know the type of signal in the following way, just for reference.

```
#!/bin/bash
set -Eeuo pipefail

cleanup() {
  echo "Trapped signal: $1"
  # script cleanup here
}

trap_sig() {
  for sig ; do
    trap "cleanup $sig" "$sig"
  done
}

trap_sig INT TERM ERR EXIT PIPE
```

 2   Reply

**Roland**  1 year ago

When I was with a company, I started developing there a "shell framework", means a library written in Shell (not simply Bash). I was able to generalize many things and had "hooks" where scripts can add their special code to the script logic.

Sadly I never started again to make a FLOSS version.

 0   Reply

57

JSON that contains something like {"success": true/false}, and when that value is true, exits with exitcode 0, and when the value is false, it exits with exitcode 1. Clearly wrapping such a tool inside a script means abandoning the automatic capture of the errors and allowing the script to process the next line normally where you would check the value of the JSON path, in the OLD annoying way...

👍 0 🗨️ ➡️ Reply



yash • 6 months ago

knowledgeable content

👍 0 🗨️ ➡️ Reply



A M • 5 months ago

Amazing script template! Keep up the amazing work.

👍 0 🗨️ ➡️ Reply



Hi, I'm Maciej 🤖

I'm a Software Developer and Architect, member of the AWS Community Builders. I do serverless AWS, a bit of frontend, and really - whatever needs to be done.

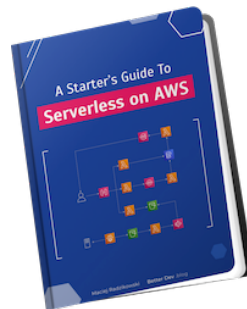
[Learn a little bit more...](#)



Get Free Ebook

Join **200+ subscribers** that receive my spam-free newsletter.

You will get **A Starter's Guide To Serverless on AWS** - my ebook about serverless best practices, Infrastructure as Code, AWS services, and architecture patterns.



I WANT FREE EBOOK

Follow me

57

Twitter



We use cookies to ensure that we give you the best experience on our website. If you continue to use this site I will assume that you are happy with it.

OK



Twitter Threads

Shorter bits of knowledge →

Recent Posts

- › [The AWS CDK, Or Why I Stopped Being a CDK Skeptic](#)
- › [Things I Found Interesting #5](#)
- › [Decision Tree: choose the right AWS messaging service](#)
- › [Personal backup to Amazon S3 – cheap and easy](#)
- › [6 Common Pitfalls of AWS Lambda with Kinesis Trigger](#)

Categories

- › [AWS \(13\)](#)
- › [Programming \(1\)](#)
- › [Things I Found Interesting \(5\)](#)
- › [Tools \(6\)](#)
- › [Uncategorized \(1\)](#)

[SERIES](#)

[ABOUT](#)

[CONTACT](#)

[PRIVACY POLICY](#)

Hestia | Developed by Themeisle

57

We use cookies to ensure that we give you the best experience on our website. If you continue to use this site I will assume that you are happy with it.

OK