Meir Gabay
Posted on Oct 24, 2021 • Updated on May 25 • Originally published at meirg.co.il

Writing Bash Scripts Like A Pro - Part 1 - Styling Guide

#bash #programming #tutorials #devops

Writing Bash scripts can be challenging if you don't know the quirks and perks. In my mother tongue, we use the Yiddish word for quirks and perks; it's called "Shtickim" (plural of "Shtick"). Are you ready to learn more about Bash's "Shtickim"?

This blog post is part of a series that I'm working on to preserve the knowledge for future me that forgets stuff, to assist new colleagues, and indulge programmers like you who wish to love Bash as I do. So let's begin, shall we?

It's A Scripting Language

It's important to remember that Bash is a scripting language, which means it doesn't offer the standard functionalities that a programming language has to offer, such as:

- Object-Oriented Programming is not supported natively
- There are no external libraries like Python's requests or Node's axios, though it is possible to use external applications such as curl

- Variables Typing is not supported, and all values are evaluated as *strings*. However, it is possible to use numbers by using specific commands, such as test equality with -eq and increment a variable with ((VAR_NAME+1)). Nevertheless, there's a "weak" way of declaring variables type with the declare command.
- Bash's Associative array like Python's dict or JavaScript's Object is supported from version Bash v4.4, and it's important to remember that macOS is shipped with Bash v3.2 (we'll get to that in future blog posts of this series)
- There is no "source of truth" for naming convention. For example, how would you name a global variable? `Pascal_Case`? `snake_case`? `SCREAMING_SNAKE_CASE`?

As you already guessed, "Bash programmers" (if there is such a thing) face many challenges. The above list is merely the tip of the iceberg.

Here are great blog posts that share the same feelings as I do:

- Foolproof Your Bash Script
- Best Practices for Writing Bash Scripts
- Bash best practices

Now that we've covered the fact that I'm in love with Bash, I want to share that feeling with you; here goes.

Variables Naming Convention

Here's how I name variables in my Bash scripts

Type	Scope	Convention
Environment	Global	MY_VARIABLE
Global	Global	_MY_VARIABLE
Local	Function	my_variable

In my older Bash scripts, the names of the variables were hard to interpret. Changing to this naming convention helped me a lot to understand the scope of variables and their purpose.

And of course, we gotta' see some practical example, so here's how I implement the above naming convention in my `good_vibes.sh` application.

good_vibes.sh

```
#!/usr/bin/env bash
# ^ This is called a Shebang
# I'll cover it in future blog posts


# Global variables are initialized by Env Vars.
# I'm setting a default value with "${VAR_NAME:-"DEFAULT_VALUE"}"
_USER_NAME="${USER_NAME:-"$USER"}"
_USER_AGE="${USER_AGE:-""}"


complement_name(){
  local name="$1"
  echo "Wow, ${name}, you have a beautiful name!"
}


complement_age(){
  local name="$1"
  local age="$2"
  if [[ "$age" -gt "30" ]]; then
    echo "Seriously ${name}? I thought you were $((age-7))"
  else
    echo "Such a weird age, are you sure it's a number?"
  fi
}


main(){
  # The only function that is not "pure"
  # This function is tightly coupled to the script
  complement_name "$_USER_NAME"
  complement_age "$_USER_NAME" "$_USER_AGE"
}


# Invokes the main function
main
```

```
main
```

good_vibes.sh - Execution and output

```
export USER_NAME="Julia" USER_AGE="36" && \
bash good_vibes.sh

# Output
Wow, Julia, you have a beautiful name!
Seriously Julia? I thought you were 29
```

Let's break down the `good_vibes.sh` application to a "set of rules" that can be implemented in your scripts.

Code block spacing

Two (2) blank rows between each block of code make the script more readable.

Indentation

I'm using two (2) spaces, though it's totally fine to use four (4) spaces for indentation. Just make sure you're not mixing between the two.

Curly braces

If it's a `${VARIABLE} concatenated with string`, use curly braces as it makes it easier to read.

In case it's a `"$LONELY_VARIABLE"` there's no need for that, as it will help you realize faster if it's "lonely" or not.

The primary purpose for curly braces is for performing a Shell Parameter Expansion, as demonstrated in the Global variables initialization part.

Squared brackets

Using **double** `[[ ]]` squared brackets makes it easier to read conditional code blocks. However, do note that using double squared brackets is not supported in Shell sh; instead, you should use single brackets `[ ]`.

To demonstrate the readability, here's a "complex" conditional code block:

```
if [[ "$USER_NAME" = "Julia" || "$USER_NAME" = "Willy" ]] \
if [[ "$USER_NAME" = "Julia" || "$USER_NAME" = "Willy" ]] \
   && [[ "$USER_AGE" -gt "30" ]]; then
  echo "Easy to read right?"
fi

# Mind that `||` is replaced with `-o`, see https://acloudguru.com/blog/engin
# Thank you William Pursell
if [ "$USER_NAME" = "Julia" -o "$USER_NAME" = "Willy" ] \
   && [ "$USER_AGE" -gt "30" ]; then
  echo "No idea why but I feel lost with single brackets."
fi
```

In case you didn't notice, you've just learned that `||` stands for `OR` and `&&` stands for `AND`. And the short -gt expression means `greater than` when using numbers. Finally, the `\` character allows breaking rows in favor of making the code more readable.

> **Shtick**: Using `\` with an extra space `\ <- extra space` can lead to weird errors. Make sure there are no trailing spaces after `\`.

I assume that using `[[ ]]` feels more intuitive since most conditional commands are doubled `&& ||`.

Variable initialization

Global variables are initialized with Environment Variables and are set with default values in case of empty Environment variables.

As mentioned in the `good_vibes.sh` comments, I'm setting a default value with

```
"${VAR_NAME:-"DEFAULT_VALUE"}"
```

In the above snippet, the text `DEFAULT_VALUE` is hardcoded, and it's possible to replace it with a variable. For example

```
_USER_NAME="${USER_NAME:-"$USER"}"
```

Functions and local function variables

Functions names and `local` function variables names are `snake_cased`. You might want to change functions names to `lowerCamelCase`, and of course, it's your call.

Coupling a function to the script is a common mistake, though I do sin from time to time, and you'll see Global/Environment variables in my functions, but that happens when I know that "this piece of code won't change a lot".

Oh, and make sure you don't use `$1` or any other argument directly; always use `local var_name="$1"`.

```
_USER_NAME="${USER_NAME:-"$USER"}"

# Bad - coupled
coupled_username(){
  echo "_USER_NAME = ${_USER_NAME}"
}

# Good - decoupled
decoupled_username(){
  local name="$1"
  echo "name = ${name}"
}

# Usage
coupled_username
decoupled_username "$_USER_NAME"
```

Functional Programming

This topic relates to **Functions and local function variables**, where functions are as "pure" as possible. As you can see in `good_vibes.sh`, almost everything is wrapped in a function, except for **Initializing Global variables**.

I don't see the point of writing the `init_vars` function, whose purpose is to deal with Global variables. However, I do find myself adding a `validate_vars` function from time to time, which goes over the Global variables and validates their values. I'm sure there's room for debate here, so feel free to comment with your thoughts.

Final Words

The "Good Vibes Application" mostly covered how to write a readable Bash script following the Functional Programming paradigm.

If you feel that there's a need to change how you name variables and functions, go for it! As long as it's easy to understand your code, you're on the right track.

The next blog posts in this series will cover the following topics:

- Error handling
- Retrieving JSON data from an HTTP endpoint
- Background jobs and watching file for changes with fswatch
- Git Repository structure - adding Bash scripts to existing repositories or creating a new repository with a Bash CLI application
- Publishing a Bash CLI as a Docker image

And more, and more ... I'm just going to spit it all out to blog posts. Feel free to comment with questions or suggestions for my next blog posts.

Discussion (29)

William Pursell • Oct 25 '21

Unless newer versions of bash are playing fast and loose with the language, ' if ["$USER_NAME" = "Julia" || ..' is an error. You could write if test "$USER_NAME" = Julia || ..., but the [command requires that its final argument be]. The || is not an argument.

Meir Gabay • Oct 25 '21 • Edited on Oct 25

Superb comment, I never use single `[ ]` brackets, so I'm less familiar with how it all works. I'll fix my answer to fit the proper syntax, thanks!

Darius Juodokas • Oct 27 '21

About the ${} and lonely variables... I disagree with this exception.

Three reasons.

1. CONSISTENCY!!! To write clean and easy to read scripts one must write consistent code. Consistent style, consistent patterns, consistent tooling.

This way a maintainer/reader won't have to switch to "oh, that's a different variable expression. wait, is that dollar sign a part of the thing, or is it to be expanded..? oh, it's expanded. Good. So it's a variable" mode.

2. READABILITY. Tightly coupled to #1. Moreover, it's A LOT easier to read script and know which parts are dynamic and which aren't when the dynamic ones have clearer visual contrast. The ${} expression introduces 3 characters that make a word look like a variable. $ only has 1 char. It's easier to spot that large clunky ${myvar} than $myvar in the code. Regardless whether the value is lonely, at the EOL/SOL, on the left or spelled in reverse (consistency again!).

3. SAFETY. Writing `local age=$1` or even `local age=${1}` is not the best idea. What if the function was called w/o the parameter? The fn logic will most likely break. I've seen oh too many mistakes like that ending up in production environments going down or even worse -- getting completely destroyed and requiring a full rebuild (means days-weeks of work). Don't do that! Whenever you're accepting and reassigning function parameters, verify whether you've been passed any. One way to do that is `[ -n "${1}" ] || return 1`, but that's just tedious and inelegant to write the same block of code in every function. Bash expansion provides a very handy safety switch: `local age=${1:?Age not provided}`. Or a fallback to default value: `local age=${1:-30}`. This way you can easily control what your variables are. If the value wasn't provided, in first case the function will return 1 and print the error message to stderr "Age not provided". In the second case the fn will continue execution after auto-assigning 30 to the variable age (if ${1} is empty).

How does that relate to safety? By writing ${myVar} you will get the habbit of always thinking "do I need to use bash's variable expansion here? Perhaps default values? Or failfast safety switches? Substitution perhaps..?". Back in the days I was using $ I used to forget to check what's inside the $1 and write failfast mechanisms. By using ${1} now my hand automatically adds the :? or :- and makes me think twice. Even if IDC if the value is empty - I always leave it as `local age=${1:-}` just to make it absolutely clear that this function can tollerate missing parameter(s).

I think I should write my own post series about shell/bash scripting.....

Meir Gabay • Oct 27 '21 • Edited on Oct 27

I would reply to this comment properly, if you hadn't added the last part

> I think I should write my own post series about shell/bash scripting

Go for it.

Darius Juodokas • Oct 27 '21

If you have a different opinion, please, do share :) I'd like to know your opinion and motivation behind "lonely variables"

baggiponte • Oct 31 '21

So which naming convention shall I use inside a loop? `for _VAR in`? Thank you for the guide!

Meir Gabay • Oct 31 '21

The way I name it

```
declare -a _ITEMS=("first" "second" "third")
for item in "${_ITEMS[@]}"; do
    echo "Item name is ${item}"
done
```

Output

```
Item name is first
Item name is second
Item name is third
```

I usually name my iterator as `item`, even if it's a `LIST_OF_ROWS` or

ARRAY_OF_THINGS I usually pick item as my iterator (in any lang).

So, as you can see, there's no clear preference, just go with what suits you, and if you find `item` to be good for you, go for it

baggiponte • Nov 7 '21

Thank you for the reply! Guess a guide on arrays would also be super useful - the example you made right here was more clarifying than many web articles I have read...

Meir Gabay • Nov 7 '21

I'll definitely cover arrays in my next blog post, thank you for the feedback, much appreciated!

baggiponte • Nov 7 '21

Also, off the top of my mind: getopt for flags and parallel to avoid writing loops in scripts (I'm just being creative)

Meir Gabay • Nov 8 '21 • Edited on Nov 8

Would you believe me if I told you that I've never, ever, used `getopts` or `paralllel` ? 🙈 I Never had the need to ... I found better alternatives that were easier to "memorize".

getopts - See bargs; A framework for creating a Bash CLI. You would expect me to use `getops` for the "Usage" menu ... But I haven't, I used other tricks to make it work

parallel - I've never processed big files in Bash, if I need to go down this road, I'll probably use Go, where it's more inclined towards parallelism and threading. I do use `background jobs`, but mainly for short processes which don't require "blocks of data". Instead, I use `background jobs`, which is simply adding `&` to a command and `wait` in the bottom of the file to wait for it to finish. For example "downloading 5 files in parallel" or "encrypting 10 files in parallel".

```
curl -o file1 https://example.com/file1 &
curl -o file2 https://example.com/file2 &
curl -o file3 https://example.com/file3 &
curl -o file4 https://example.com/file4 &
curl -o file5 https://example.com/file5 &

wait # for all jobs to finish ...
```

It's also a matter of time/knowledge; if you know `getopts` and `parallel` then use them. If you don't, feel free to pass it, as I haven't found it "mandatory", but that's me.

baggiponte • Nov 8 '21

I didn’t know about bargs! That’s precious advice, thank you!

Ting Smith • Jun 2

this blog is impressive, makes me learn a lot

Shrihan • Oct 26 '21

```
sudo comment "Great stuff! Will wait for the next parts!"
```

Piotr Szeptynski • Oct 27 '21

You don't need sudo for this. 😉

Meir Gabay • Oct 26 '21

Thanks! Much appericiated!

MorphZG • Oct 25 '21

Thanks for sharing. Will follow and wait for more.

Meir Gabay • Oct 25 '21

Kudos, will post the next part soon :)

Ioan Indreias • Oct 26 '21

I'm usually set environment variables for the Bash scripts (or any other executable) like:
$ USER_NAME="Julia" USER_AGE="36" good_vibes.sh

Meir Gabay • Oct 27 '21

Depends on the context ...
For local usage, sure, that works great.

For CI/CD processes, usually, values/secrets are injected with env vars ...

My mindset - fetch default values from env vars. A matter of approach/opinion I guess

Piotr Szeptynski • Oct 27 '21

Me too. That's better than exporting them to the env.

Bence Szabo • Oct 26 '21

Nice! I'm happy to see that somebody agrees that bash is still relevant and is worth to learn.
Will you cover the readonly "qualifier"?

Meir Gabay • Oct 26 '21

Meir Gabay • Oct 26 '21

I'm using Bash across most of my projects, so for me it's definitely here to stay :)

Truth be told, I haven't used [readonly] in any script, ever. I guess it's best practice to create immutable variables with:

```
declare -r VAR_NAME="CONSTANT_VALUE"
VAR_NAME="trying-to-change-value"
# Output
# bash: VAR_NAME: readonly variable
```

Looks cool and easy to implement, I might adopt it in my future scripts.

zoulja • Oct 26 '21

Really nice article, but some explanation is needed.

> Oh, and make sure you don't use $1 or any other argument directly; always use local var_name="$1".

Why?

Meir Gabay • Oct 26 '21

@zoulja Good point!

I prefer using named variables instead of positional arguments. So the logic of my code is based on names instead of indexes, such as `$1`, `$2` and so on. This way, even if the order of given arguments is changed, I still maintain my function's logic.

And of course, let's learn by example; assuming we create the `greet()` function

```
greet(){
  local name="$1"
  local age="$2"
  echo "Hello ${name}, you're ${age} years old."
}

# Usage
greet "Willy" "33"
```

Now, I want to change the positional arguments, so the function consumes `age` and then `name`

```
greet(){
  # I switched between $1 and $2
  # Also changed the order of variables so it makes sense
  local age="$1"
  local name="$2"

  # Didn't touch the logic
  echo "Hello ${name}, you're ${age} years old."
}

# Usage
greet "33" "Willy"
```

I hope that explains it

Tatiana • Oct 26 '21

Great timing! I was just looking into creating a couple scripts today.

Meir Gabay • Oct 26 '21

Glad I could help!

King o' Hell • Oct 27 '21

Can I have your editor's color profile?

Meir Gabay • Oct 27 '21 • Edited on Oct 27

Well .. As you can those are code blocks, not screenshots, so the coloring comes from dev.to Markdown CSS.

To make code blocks beautiful in Markdown, simply add the relevant lang at the beginning of the code block, like:

```
echo "awesome"
```

I added `bash` right after the first 3 backticks `

Code of Conduct • Report abuse

Meir Gabay

I'm passionate about studying and teaching. DevOps Engineer @ Binah.ai

LOCATION
Israel

EDUCATION
Industrial Engineer, B.Sc @ Shenkar ; Exact Science Education and Technology, M.A @ Tel-Aviv Univ.

WORK
DevOps Engineer @ Binah.ai

JOINED
Sep 3, 2019

More from Meir Gabay

How To Recover Secrets From GitHub Actions
#devops #cicd #githubactions #secrets

How To Test A GitHub Action
#actionshackathon21 #devops #cicd #github

How To Develop A Progressive Web Application On An Android Device

How To Develop A Progressive Web Application On An Android Device

#javascript #typescript #vue #programming