Writing Good Bash Scripts

An ancient craft for modern DevOps times

Posted on March 25, 2020

Bash Scripts

In recent years, with the rise of DevOps practice, shell scripting is more and more important. As most popular shell is Bash, when we talk about scripting we are thinking of Bash scripts. For sure it lacks higher-level constructs like data structures or JSON support but it makes up with simplicity for common things. Bash scripting is very efficient in cases when it is enough to combine a few Linux commands to get the work done.

How to Not Shoot Yourself in the Foot?

Shell scripts tend to morph into ***Big ball of mud*** **(https://en.wikipedia.org/wiki/Big_ball_of_mud)**. Scripting can be very forgiving and therefore sloppiness can creep in very easily. To counter this, it is very beneficial to introduce rigorous code style checks. Peer reviews are also helpful.

Code Style

Not every code style will appeal to everybody. Because it is a somewhat subjective subject (pardon the pun), one should give to the will of the group. As a good place to start, **Google Shell Style Guide (http://google.github.io/styleguide/shellguide.html)** is a comprehensive guide on how to write clean shell scripts.

To catch nasty code issues there is **ShellCheck (https://github.com/koalaman/shellcheck)** - A *shell script static analysis tool*. It can be used as a command in the terminal but there are editor plugins as well. Look it in action below. Shellcheck complains because there is no shebang and variable isn't quoted.

```
$ echo "echo \$1" | shellcheck -

In - line 1:
echo $1
^-- SC2148: Tips depend on target shell and yours is unknown. Add a
shebang.
     ^-- SC2086: Double quote to prevent globbing and word splittin
g.
```

Unofficial Bash Strict Mode

How to make a script more robust and reliable? Actually, there is a simple trick. Add at the beginning of the script line with `set -euo pipefail`. Why? Because:

- `-e` will make script fail if any command fails
- `-u` will error script when uninitialized variable is accessed
- `-o pipefail` will stop on the first error when piping. By default, errors will be masked by piping.

For more comprehensive description of what is happening visit **this blog post (http://redsymbol.net/articles/unofficial-bash-strict-mode/)**. **This blog post (https://sipb.mit.edu/doc/safe-shell/)** is also a good article on the same subject.

When I see the usage of these options I immediately have more confidence using that script. It is that powerful.

Defensive Bash Scripting

I'm a believer that using functions and local variables makes a script better. Why? Well, Bash offers local variables that belong to the function. This fact by itself is enough to recommend usage of functions. When variable can't be made `local` try the best to make it `readonly`. We know that mutability is a source of a lot of bugs so taking mutability from the equation makes a room for scripts with less bugs. I recommend reading **this blog post (http://kfirlavi.herokuapp.com/blog/2012/11/14/defensive-bash-programming/)** from 2012. Nothing major has changed since.

Simple Bash Script

So how your most simple Bash script should look like? I recommend you start with something like below.

```
#!/usr/bin/env bash
set -eu -o pipefail

function main() {
  echo 'Implement me!'
}

main "$@"
```

Let's go through the script line by line.

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

Why not go with `#!/bin/bash`? There is a good discussion about the topic on **StackOverflow (https://stackoverflow.com/questions/21612980/why-is-usr-bin-env-bash-superior-to-bin-bash/21613044)**. Using `/usr/bin/env` will search path variable. This offers a possibility not to use default bash console. So it is more flexible.

```
set -eu -o pipefail
```

Setting options has been discussed above in **strict mode**.

```
function main() {
```

Using functions comes from recommendation from **defensive** programming. Script logic will be implemented in this `main` function. Function allows the usage of local variables.

```
main "$@"
```

This is an entry into the script. It calls `main` function and forwards all of the arguments to it.

You can download above template from **this link (../code/bash-min.sh)**.

Bash Script as Template for Command Line Utility

What if you are writing some command-line utility using Bash? Some say it is wiser not to. But still, in some cases, it might be the best tool for the job. And this is all it matters. If this is the case, then you are better off with a script with more functionality already built-in. Functionality that you will probably need are:

- Program arguments parsing
- Argument validation
- Script usage printout
- Ability to debug the script
- All other sanity defaults set like **Unofficial Bash Strict Mode** and **Defensive Bash Scripting**

Your starting point might look like one below. I will make this script available at **this link (https://mresetar.github.io/code/bash.sh)**.

```
 #!/usr/bin/env bash
set -eu -o pipefail

readonly E_USAGE=99
readonly SCRIPTNAME="${0##*/}"

if [[ "${_DEBUG:-}" == "true"  ]]; then
  set -x
fi

function usage() {
  cat >&2 <<EOF
Usage:
  ./${SCRIPTNAME} -p param1

Example:
  ./${SCRIPTNAME} -p param1

EOF

  exit "${E_USAGE}"
}

function main() {
  declare param1

  while getopts ":p:" optchar; do
    case "${optchar}" in
        p)
          param1="${OPTARG}"
          ;;
        *)
          echo "ERROR: Unknown flag '${OPTARG}'" >&2
          usage
          ;;
      esac
  done

  if [[ -z "${param1+x}" ]]; then
```

```
    usage
  fi

  echo "Script arguments:"
  for i in "$@"; do
    echo "${i}"
  done
}

main "$@"
```

Creating New Script Using Alias

To make life easier for yourself you can create an alias (or better a function) for this action. I've created 2 functions for the creation of a minimal script and one for CLI tool. Functions are available **here (https://mresetar.github.io/code/bash-func.sh)**. To add the functions to your `~/.bashrc` you can use this line:

```
curl -s https://mresetar.github.io/code/bash-func.sh | tee -a ~/.ba
shrc
```

After launching a new Bash terminal you could use them with `bash-min` and `bash-cli` commands. e.g.

```
$ bash-min min.sh
$ ./min.sh
Implement me!

$ bash-cli cli.sh
$ ./cli.sh
Usage:
  ./cli.sh -p param1

Example:
  ./cli.sh -p param1
```

Summary

- Link for simple Bash script starter: **download minimal script (https://mresetar.github.io/code/bash-min.sh)**
- Link for more complex CLI Bash script starter: **download script (https://mresetar.github.io/code/bash.sh)**
- Add both scripts to the `~/.bashrc` to use them as `bash-min <new_file>` and `bash-cli <new_file>`:

```
curl -s https://mresetar.github.io/code/bash-func.sh | tee -a ~/.bashrc
```

I wish you safe and pleasant shell scripting :)

Tags: **bash (/tags#bash)** **cli (/tags#cli)** **devops (/tags#devops)**


← **PREVIOUS POST (/2020-03-20-SQUASHING-DOCKER-IMAGES-FOR-SMALLER-SIZE/)**

What do you think?

5 Responses

Upvote Funny Love Surprised Angry Sad

0 Comments **mresetar** **Disqus' Privacy Policy** **Login**

Favorite 1 **Tweet** **Share** **Sort by Best**

Start the discussion…

LOG IN WITH **OR SIGN UP WITH DISQUS**

Name

Be the first to comment.

Subscribe **Add Disqus to your siteAdd DisqusAdd** **Do Not Sell My Data**

(/feed.xml) **(mailto:${github-username}@gmail.com)**

(https://github.com/mresetar) **(https://twitter.com/MiroslavResetar)**

(https://linkedin.com/in/mresetar) **(https://www.flickr.com/photos/161431263@N06/)**

Theme by **beautiful-jekyll (https://deanattali.com/beautiful-jekyll/)**