jzbruno.github.io

Bash Style Guide

Written for Bash 4.4 or greater.

Setting standards around Bash script style makes scripts more reliable, easier to understand, easier to debug, and easier to update, not only for the creator but for other contributors as well.

This is worth the investment and will save you time in the long run.

This guide borrows heavily from

- Unofficial Bash Strict Mode
- Google Shell Style Guide
- BashFAQ

A useful tool to lint your Bash scripts is ShellCheck.

Consistency where reasonable

The following styles are largely a matter of personal taste. In each case choose one for the project and *stick with it*. Consistency will make the code base easier to read and update. If there is already mixed usage in the code base, try to choose the most common usage.

Indentation:

- 2 space
- **4 space preferred**
- tabs

Variable names:

- **myVariable preferred**
- my_variable

File names:

- my-script

- **my-script.sh preferred**
- my_script
- my_script.sh

Comments

Comments tend to become out of date and rot easily. Keep them minimal and only explain non-obvious decisions and why they were made.

For example, if a piece of code has to be written a certain way because of a bug in a binary that is being called, comment about that.

Use /usr/bin/env to find bash

Use of */bin/bash*, */usr/bin/bash*, */bin/sh* or similar references a specific location for Bash. This requires Bash to be installed in a specific location and isn't as portable, especially for running scripts in Docker containers.

Use */usr/bin/env* to search the user's path for the first occurrence of Bash.

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

> Be aware this can cause issues if the user has more than one version of Bash in their path and the version found is incompatible with the script.

Use strict mode

To enable strict mode start each script with the following

```
set -euo pipefail
```

- **-e:** Abort the script at the first error when a command exits with a non-zero status. This allows the script to fail fast and makes it easier to determine where a script failed.
- **-u:** Attempt to use undefined variable outputs error message, and forces an exit. This prevents un-expected inputs being passed around and makes it easier to find tricky bugs caused by unset variables.
- **-o pipefail:** Set the exit status of a pipeline to the rightmost command exit status with a non-zero status, or zero if all commands exit successfully. This prevents complex pipelines from

continuing even if the first command failed and makes it easier to debug where the error occurred.

Use debug mode

Bash provides two verbose modes that make debugging much easier than using echo or print statements.

- Using the -v option will print each command run by a script.
- Using the -x option will print both the command run and expands variables. ***Preferred***

Given the simple script *hello.sh*

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

set -euo pipefail

message="Hello $(whoami)!"
echo "${message}"
```

Running the script as normal produces

```
$ bash hello.sh
Hello p2723614!
```

Running the script with -x produces

```
$ bash -x hello.sh
+ set -euo pipefail
++ whoami
+ message='Hello p2723614!'
+ echo 'Hello p2723614!'
Hello p2723614!
```

Give $n variables names

Naming variables provides a form of documentation as code especially for Bash functions which don't define variables in the function signature.

This makes it easier to understand what the $n variable is suppose to contain.

```
sayHello() {
    words="${1}"
    echo "${words}"
}
```

Check for required binaries

Checking for required binaries before the script runs allows the script to fail fast if it is being run in an environment that is improperly configured. It provides a form of dependency documentation as well.

```
if ! type jq &>/dev/null; then
        err "Missing required command 'jq'."
        exit 1
fi
```

Use double square brackets in if statements *[[ ]]*

Double square brackets should be used over single square brackets due to fewer surprises in how they behave. Some of the features are:

- Support for regular expression matching.
- Operators are similar to other programming languages and more familiar.
- Support for the negation operator *!*.
Support for conditional evaluation *|	* and *&&*.

> Note that this is not portable but I generally only care about Bash and do not attempt to run my scripts with other shells.

Use parenthesis for command substitution *$( )*

The parenthesis *$( )* form of command substitution should be used over the backtick form `` ` ` ``. This is also due to fewer surprises in how it behaves. Some of the features are:

- Nested quoting is cleaner.
- Nested command substitution is cleaner.

Use *$var* variable references over *${var}*

This is just a personal preference but should be used consistently. Seems cleaner to reference variables without brackets unless necessary, and it isn't necessary often.

Use double quotes around all variable references ""

Any variable that can contain whitespace should be quoted to avoid unintended splitting of the value. It is usually safer to just quote all variables that shouldn't be split to avoid unintended behavior.

Send error messages to stderr

It is common to suppress stdout message from a command. If your script writes all messages to stdout errors will also be suppressed and it will hide the cause of a failure.

```
>&2 echo "Missing argument 'name'."
```

Putting the redirection in front makes it more obvious this output will be sent to stderr.

A simple *err* function can be used too

```
err() {
    >&2 echo "$@"
}

err "Missing argument 'name'."
exit 1
```

Template

style-guides/bash-template.sh