# Better BASHing Through Technology

28 Aug 2020

I write a lot of bash scripts for both my day job and my personal projects, and while they are functional, bash scripts always seem to lack that structure that I want, especially when compared to writing something in Go or C#. The main problem I have with bash scripts is that when I use functions, I lose the ability to log things.

For example the `get_config_path` function will print the path to the configuration file, which will get consumed by the `do_work` function:

```
get_config_path() {
  echo "Finding Machine Configurtaion"

  if [ -n "$CONFIG_PATH" ] && [ -e "$CONFIG_PATH" ]; then
    # echo "Using Configuration from environment"
    echo "$CONFIG_PATH"
    return
  fi

  if [ -e "~/.config/demo/config.toml" ]; then
    # echo "Using Configuration directory file"
    echo "~/.config/demo/config.toml"
    return
  fi

  # echo "Unable to find configuration"
  exit 1
}

do_work() {
  local -r config=$(get_config_path)

  # actual work...
}
```

The problem is, if I include the `echo` statements which are log lines, the `config` variable in the `do_work` function will contain them too.

In the past, this has caused me to use functions in bash very sparingly; usually with things that are short that I don't need to add much logging to. However, I was recently building some AMIs, and happened to be browsing the Consul and Vault terraform module source, which uses several bash scripts which are written in a different style, which makes them vastly more maintainable.

So let's have a look at the new structure I use, which is heavily based off these scripts.

## Better Bash Scripts

Before we get to the implementation, make sure you are using ShellCheck it provides static analysis of your scripts, pointing out many things like missing quotes, or incorrectly handling arrays. It has plugins for all your favourite editors too.

## 0. General Styles and Constructs

All variables should be declared `readonly`, and `local` if possible, to help prevent surprise values from being present if other functions forget to do the same.

```
#top level
readonly SCRIPT_NAME="$(basename "$0")"

some_method() {
```

```
some_method() {
  # within a method
  local -r config_path="$1"
}
```

Functions should assign their passed arguments to named variables as the first thing they do, preferably matching the variable name they are passed, which helps later when you are searching through a script for usages of “config_file” and not having to find other names/aliases for the same value.

```
read_config() {
  local -r config_file="$1"
  local -r skip_validation="$2"

  # ...
}

invoke() {
  # ...

  read_config "$config_file" "$skip_validation"
}
```

# 1. Error Handling

It should go without saying, but you really need to start your scripts with the following:

```
#!/bin/bash

set -euo pipefail;
```

There are many better articles on what these specifically do, but suffice to say:

- `e` causes the script to stop on errors
- `u` causes it to error on undefined variables being used
- `o pipefail` causes a non-zero exit code from any command in a pipeline to fail the script too (rather than just the last command.)

# 2. Logging

The real advantage of this structure is we get to have log statements! This is achieved by doing all logging to `stderr` instead of `stdout`. We use a standardised `log` function across all the scripts, which also includes the script’s name so when calling other scripts you can see which one wrote the log line:

```
readonly SCRIPT_NAME="$(basename "$0")"

log() {
  local -r level="$1"
  local -r message="$2"
  local -r timestamp=$(date +"%Y-%m-%d %H:%M:%S")

  >&2 echo -e "${timestamp} [${level}] [$SCRIPT_NAME] ${message}"
}
```

Invoking the function is `log "INFO" "Some status"` or `log "WARN" "Something concerning"` etc.

# 3. Error Checking

We have some standard assertion functions which are used by the script when starting up to validate arguments:

```
assert_not_empty() {
  local -r arg_name="$1"
  local -r arg_value="$2"

  if [[ -z "$arg_value" ]]; then
```

```
    log "ERROR" "The value for '$arg_name' cannot be empty"
    exit 1
  fi
}

assert_is_installed() {
  local -r name="$1"

  if [[ ! $(command -v "$name") ]]; then
    log "ERROR" "The binary '$name' is required by this script but is not installed or in the system's
PATH."
    exit 1
  fi
}
```

## 4. Argument parsing

When scripts need to take parameters in, I prefer to use long-flag style, as they are little more readable for people checking invocations again in the future. This function is usually always called `run`, and is the last function defined, and is invoked immediately after definition, passing in all script arguments (`run "$@"`):

```
run() {
  local namespace=""
  local suffix=""
  local dry_run="false"

  while [[ $# -gt 0 ]]; do
    local key="$1"
```

```
    local key="$1"

    case "$key" in
      --namespace)
        namespace="$2"
        shift
        ;;
      --suffix)
        assert_not_empty "$key" "$2"
        suffix="$2"
        shift
        ;;
      --dry-run)
        dry_run="true"
        ;;
      --help)
        print_usage
        exit
        ;;
      *)
        log "ERROR" "Unrecognized argument: $key"
        print_usage
        exit 1
        ;;
    esac

    shift
  done

  # mandatory flag validation
  assert_not_empty "--namespace" "$namespace"

  # make sure tools are installed
  assert_is_installed "vault"
  assert_is_installed "openssl"
  assert_is_installed "jq"

  # do the work!
  local -r cert=$(generate_cert "$suffix")

  store_cert "$namespace" "$cert" "$dry_run"

}

run "$@"
```

The validation uses the `assert_not_empty` function defined above, which is used in two ways: after the `while` loop to check mandatory values have been filled in, and within the `case` statement for optional flags values.

We also use `assert_is_installed` to validate that utilities we need are installed, such as `vault`, `openssl` and `jq`

The `print_usage` function is just a set of `echo` statements giving all the flags, and an example of invokation:

```
print_usage() {
  echo
  echo "Usage: $SCRIPT_NAME [OPTIONS]"
  echo
  echo "This script creates a new certiticate, and it installs it into the right namespace"
  echo
  echo "Options:"
```

```
  echo "Options:"
  echo
  echo -e "  --namespace\tThe namespace to install the certificate in"
  echo -e "  --suffix\tAn optional suffix for the hostname"
  echo -e "  --dry-run\tDon't install the certificate"
  echo
  echo "Example:"
  echo
  echo "  $SCRIPT_NAME --namespace test --dry-run"
}
```

# Usage

I keep a single template file which has all of this written into it, and new scripts start off with a copy-paste of the template. Could it be DRYer? Sure, but then I have to deal with dependency management, and it's just not worth the hassle and overhead.

bash

« Sharing Docker Layers Between Build Agents Isolated Docker Multistage Images »