# Shell Style Guide

**of**

# All projects

**Version 0.1**

**No. SG-0001**

**King Mampreh OJSC**

**17/06/2020**

## Revision History

| Editor | Date | Overview | Version |
|---|---|---|---|
| simon.simonyan@gmail.com | 17/06/2020 | Initial Document | 0.1 |

# Table of Contents

# Chapter 1

# Comments

## 1.1 Shebang

### Rule 1.1

Do put **shebang** at the beginning of a script file.

Why? On UNIX-like systems, scripts should always start with a shebang line. The system call "execve" (that is responsible for starting programs) relies on an executable, having either an executable header or a shebang line.

Why? If the file has executable permissions, but no shebang line and does seem a text file, the behaviour depends on the shell that you're running in. Since there is no guarantee that the script was actually written for that shell, this can work or fail spectacularly.

```
#!/bin/bash
```

# Chapter 2

# Formatting

## 2.1 Indentation

### Rule 2.1

Do use 2 spaces for indentation.

Do not use tabs.

Why? A tab could be a different number of columns depending on environment, but a space is always one column.

```
find ${DOX_DIR} -type f -name 'Doxyfile' -print0 |  while IFS= read -r -d $'\0' DOXYFILE; do

  spec_dir="${DOXYFILE%/*}"
  spec_name="${SPEC_DIR##*/}"

  revision_history=RevisionHistory
  revision_history_md="${revision_history.md}"
  revision_history_tex="latex/${revision_history.tex}"

  # Generate latex for Revision History
  cd "${SPEC_DIR}"

  if [[ ! -d latex ]]; then
    mkdir latex
  fi

  # ...
done
```

## 2.2 User defined variable

### Rule 2.2

Do declare all global varables at the top of a script file.

Why? Shell script allows to use variables after declaration.

Why? Grouping of the variables at the top of the script is important for someone else to be informed which variables are used in the script and in order to change the value of any variable.

```
#!/bin/bash

PLANTUML_STYLE=$1
if [[ -z "${PLANTUML_STYLE}" ]]; then
  PLANTUML_STYLE=classic
fi

TARGET=$2
# Some checkings for TARGET

TARGET_ANDROID="android"
TARGET_IOS="ios"
TARGET_WINDOWS="windows"

if [[ "${TARGET}" == "${TARGET_ANDROID}" ]]; then
  CMAKE="$ANDROID_PATH/cmake/$CMAKE_VERSION/bin/cmake"
else
  CMAKE=cmake
fi
```

# Chapter 3

# Naming Conventions

## 3.1 User defined global variable

### Rule 3.1

Do use underscores with upper case for user defined global variable declaration.

Why? There are accepted coding styles for variable naming conventions. This style is chosen, because system variables also declares in the same way.

```
TARGET_ANDROID="android"
TARGET_IOS="ios"
TARGET_WINDOWS="windows"
```

## 3.2 Function name

### Rule 3.2

Do use camelCase style for function names.

Do put parentheses after the function name.

Why? This style is chosen within other accepted styles for function naming conventions to differentiate variable and function names.

Why? Parentheses after name indicates that it is a function.

```
function nvmInstallNode () {
  echo "=> Installing Node.js version ${NODE_VERSION}"
  nvm install "${NODE_VERSION}"

  CURRENT_NVM_NODE="$(nvm_version current)"
  if [[ "$(nvm_version "${NODE_VERSION}")" == "${CURRENT_NVM_NODE}" ]]; then
    echo "=> Node.js version $NODE_VERSION has been successfully installed"
  else
    echo >&2 "Failed to install Node.js ${NODE_VERSION}"
  fi
}
```

## 3.3 User defined local variable

### Rule 3.3

Do use underscores with lower case to declare local variable.

Why? This style is chosen to differentiate global and local variables.

```
function verify () {
  for item in $(ls *.$1) ; do
    name="${item%.*}"
    if [[ ! -f "${name.$2}" ]]; then
      echo "Error: $2 file for '$NAME' does not exist. Aborting."
      exit 1
    fi
  done
}
```

## 3.4 Define a path

### Rule 3.4

Do use suffix **_DIR** or **_PATH** for variable to define a path.

Do not use suffix **_DIR** when declared object is not a drectory.

Why? Adding suffixes, makes variable more visible and readable that it is describes path or directory.

```
LIBRARIES_PATH="${EXECUTABLE_DIR}/../../libraries"
CPPUNIT_LIB_PATH="${LIBRARIES_PATH}/cppunit-1.12.1"
CPPUNIT_PREBUILT_PATH="${LIBRARIES_PATH}/cppunit-prebuilt/${PLATFORM_DIR}"
CPPUNIT_INCLUDE_PATH="${CPPUNIT_LIB_PATH}/include"
```

# Chapter 4

# Paths

## 4.1 Current directory

### Rule 4.1

Do use an example script to get current directory path.

Why? In this example it checks operating system and gets current directory, to enable scripts to operate in cross-platform environment.

```bash
#!/bin/bash
DARWIN="Darwin"
LINUX="Linux"
if [[ $(uname -s) != "${DARWIN}" && $(uname -s) != "${LINUX}" ]]; then
  CURRENT_DIR=$(pwd -W)
else
  CURRENT_DIR=$(pwd)
fi
```

## 4.2 Path of the executable

### Rule 4.2

Do use the example script to get the path of the executable.

Why? Example gets the relative path, which is necessary to get a full paths. (See Relative Paths)

```bash
#!/bin/bash

EXECUTABLE_DIR=$(dirname $0);
```

## 4.3 Paths concatination

### Rule 4.3

Do use double quotes **""** for path concatination.

Why? In order to avoid problems when path contains files/folders with space, the double quotes is always used in path concatination.

```
STYLE_GUIDES_DIR="Dox/Style Guides"
SHELL_STYLE_GUIDE_PATH="${STYLE_GUIDES_DIR}/SG-0003 Shell Style"
```

## 4.4 Relative paths

### Rule 4.4

Do use *relative path* relative to the executable path.

Do add *executable path* to the *relative path*.

Why? Adding executable path to the relative path gets the absolute path.

Why? Using absolute path, there is no need to resolve any problems of path.

```
#!/bin/bash

DARWIN="Darwin"
LINUX="Linux"
EXECUTABLE_DIR=$(dirname $0)

if [[ ! "${EXECUTABLE_DIR}" = "/*" ]]; then
  if [[ $(uname -s) != "${DARWIN}" && $(uname -s) != "${LINUX}" ]]; then
    CURRENT_DIR=$(pwd -W)
  else
    CURRENT_DIR=$(pwd)
  fi
  EXECUTABLE_DIR="${CURRENT_DIR}/${EXECUTABLE_DIR}"
fi

TEMPLATES_DIR="${EXECUTABLE_DIR}/../Templates"

for file in $(ls "${TEMPLATES_DIR/*.html}"); do
  base=${file##*/}
  name=${base%.*}

  echo ${base}
  echo ${name}
done
```

## 4.5 Path in expression

### Rule 4.5

Do use double quotes "" for the path in expressions.

Why? When defined path has a space, without quotes path does not work correctly.

```
DATABASE="${EXECUTABLE_DIR}/Database Sqlite/screens.sqlite"
SOURCE="${EXECUTABLE_DIR}/Database Sqlite/screens.xml"
TEMP="${EXECUTABLE_DIR}/Database Sqlite/screens.csv"
REMOVE_DB_IF_NOT_UP_TO_DATE="${EXECUTABLE_DIR}/Database Sqlite/remove_screens_db_if_not_up_to_date.make"

if [[ ! -e "${SOURCE}" ]]; then
  echo "${SOURCE} does not exist. Aborting."
  exit 1
fi

if [[ -e "${DATABASE}" ]; then
  make -f "${REMOVE_DB_IF_NOT_UP_TO_DATE}"
  if [[ -e "${DATABASE}" ]]; then
    echo "Database $(basename ${DATABASE}) exists and is up to date."
    exit 0
  fi
fi
```