

(/.) / [Developers Community \(/develop/\)](#) / [Infra \(/develop/infra/\)](#)

Authors:



David

Much of this developer documentation provides historical context but may not reflect the current state of the project.

If you see outdated content please navigate to the page footer and click "**Report an issue on GitHub**".

It is **not** user documentation and should not be treated as such.

[User Documentation is available here. \(/documentation/\)](#)

NOTE: for the latest version of this doc, see <http://ovirt-infra-docs.readthedocs.org/en/latest/> (<http://ovirt-infra-docs.readthedocs.org/en/latest/>).

Infra bash scripts style guide

These are some coding guidelines in order to have a reference when submitting patches. It's based on the [Bash Hackers guidelines](http://wiki.bash-hackers.org/scripting/style) (<http://wiki.bash-hackers.org/scripting/style>).

This is not an enforcement, it's meant to be just a reference, of course, compliance is preferred.

Some good code layout helps you to read your own code after a while. And of course it helps others to read the code too.

Basic principles

- First robustness
- Second readability/maintainability
- Third portability

That is why we allow and encourage bashisms, if that improves the robustness or the readability of the script.

Indenting and line breaks

Use 4 spaces per indent

To indent, use 4 spaces per indentation level, similar to python indentation.

Whenever possible use the 79 chars max line.

Avoid hard-tabs when possible. I can imagine only one case where they're useful: Indenting here-documents.

```
# No:
my_func() {
tab>echo 'bla'
}
my_func2() {
    echo 'bla'
}
```

```
# Yes:
my_func() {
    echo 'bla'
}
```

If you have to split a command, use one option per line, with extra indent

```
my_command \
    --opt1 opt1_arg1 \
    --opt2 \
    arg1 \
    arg2
```

Break sequences pre-operator, same indent

```
command1 \
|| command2 \
&& command3 \
| command4
```

Break before redirection, same indent

```
command \  
1> outfile \  
2> errorlog
```

Compound commands: basic layout

```
HEAD_KEYWORD parameters; BODY_BEGIN  
    BODY_COMMANDS  
BODY_END
```

Long predicates

Avoid long predicates as much as possible, but if you have to:

```
HEAD_KEYWORD command1 \  
OPERATOR command2 \  
OPERATOR commandN \  
BODY_BEGIN  
    BODY_COMMANDS  
BODY_END
```

Or if you have one command with long options:

```
HEAD_KEYWORD command1 \  
    --myoption \  
    param1 \  
    param2 \  
BODY_BEGIN  
    BODY_COMMANDS  
BODY_END
```

But try to use a function or storing the return code if able:

```
test_something() {
    command1 \
    OPERATOR command2 \
    OPERATOR command3
    return $?
}

HEAD_KEYWORD test_something; BODY_BEGIN
    BODY_COMMANDS
BODY_END

command1 \
    --myoption \
    param1 \
    param2
command1_res=$?

HEAD_KEYWORD [[ "$command1_res" == 0 ]]; BODY_BEGIN
    BODY_COMMANDS
BODY_END
```

if/then/elif/else

```
if ...; then
    ...
elif ...; then
    ...
else
    ...
fi
```

for

```
for f in /etc/*; do
    ...
done
```

while/until

```
while [[ "$answer" != [YyNn] ]]; do
    ...
done
```

case

```
case $input in
    hello) echo "You said hello";;
    bye)
        echo "You said bye"
        if foo; then
            bar
        fi
    ;;
    *)
        echo "You said something weird..."
    ;;
esac
```

Syntax and coding guidelines

Basic structure

The basic structure of a script simply reads:

```
#!/SHEBANG
GLOBAL_CONFIGURATION_CONSTANTS
FUNCTION_DEFINITIONS
if [[ "$0" =~ /bash$ ]]; then
    PARSING_OPTIONS
    VERIFYING_OPTIONS
    SIMPLE_MAIN_CODE
fi
```

Shebang: use **/bin/bash -e**

If possible (I know it's not always possible!), use a shebang. Be careful with `/bin/sh`: The argument that "on Linux `/bin/sh` is a Bash" **is a lie** (and technically irrelevant) The shebang serves two purposes for me:

- it specifies the interpreter when the script file is called directly: If you code for Bash, specify *bash*!
- it documents the desired interpreter (so: use *bash* when you write a Bash-script, use *sh* when you write a general Bourne/POSIX script, ...)

Whenever able, use the `-e` flag, that will make sure your script fails if any of the commands fail:

```
#!/bin/bash -e
```

If you don't really care about one of the commands failing (or returning `!= 0`) you can use this:

```
mycommand || :
```

Use `[[]]` and not `[]`

Prefer the bash keyword `[[]]` to the old `test` command, it's behavior is a lot more predictable, as it handles spaces as expected.

```
# No
if [ -e "$my_var" ]; then
    ...
fi
```

```
# Yes
if [[ -e "$my_var" ]]; then
    ...
fi
```

Declare all globals at the start of the script

Even if empty or inheriting from the env, declare all the globals for better visibility at the top of the script.

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

MY_VAR1="${ENV_VAR:-default value1}"
MY_VAR2='default value2'
```

Use dotted/prefixed names in libraries

That way it's a lot easier to debug and maintain all the scripts.

```
### mylib.sh ####  
#/usr/bin/env bash -e  
  
mylib_MYVAR1=""  
  
mylib.func1() {  
    ...  
}
```

Use caps and underscores for globals

Also if it could collide with a reserved var (http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap08.html#tag_08), add a prefix like MY_ :

```
MY_GLOBAL="whatever"
```

Use at least 3 chars for any variable

```
# No  
m="whatever"
```

```
# Yes  
msg="whatever"
```

Prefer keywords to Builtins

```
# No  
my_var="$(echo "$my_var" | cut -d' ' -f1)"
```

```
# Yes  
my_var="${my_var%% *}"
```

If not 101% sure, quote all variable expansions

If not, they will undergo path expansion.

```
var1="$var2"
for iter in "$@"; do
    ...
done
```

Arrays are there, use them!

```
var1=(
    "elem one"
    "elem two"
)
var1+=( "elem three" )
for iter in "${var1[@]}"; do
    ....
done
```

When expanding arrays, use "\$@"

It expands spaces properly. and unless you really know what you are doing, that's what you expect.

```
# No
for iter in ${my_array[*]}; do
    ...
done
```

```
# Yes
for iter in "${my_array[@]}"; do
    ...
done
```

Don't use `, use \$()

Though being more portable, the backtick is less robust, it's not nestable and it's less readable.

```
# No
my_var=`echo `ls``
```



```
#Yes
my_var="$(echo "$(ls)")"
```

Quote all command expansions "\$()"

Mainly because the result of the command expansion will undergo word splitting and path expansion, and that's usually not wanted. In some cases, like defining a var it will not, but as that's not generic, the safest is to quote it always.

```
# No
# Will work, but just in this case
my_asterisc=$(echo '*')
# Now it will be expanded to all the current dir contents
ls $(echo '*')
```

```
# Yes
# result is the same as without quotes
my_asterisc="$(echo '*')"
```

```
# but now it will only show a file/dir named '*'
ls "$(echo '*')
```

If eval is the answer, surely you are asking the wrong question

Avoid if, unless absolutely necessary, it's usually unnecessary, and it's really easy to break things. Use only if you really have to and you know what you are doing (add a comment too, so future you will remember).

Output: normal > stdout, error + debug > stderr

If the script gives syntax help (- ? or -h or --help arguments), it should go to *STDOUT*, since it's expected output, unless it's a response to a missing or malformed parameter.

Prefer gnu getopt to getopt

It's more robust, implements the more readable gnu long options, plus POSIX, and does not hurt readability too much

```
# No
while getopts a:b:c flag; do
    case $flag in
        a)
            echo "-a used: $OPTARG";
            ;;
        ...
        ?)
            exit;
            ;;
    esac
done
shift $(( OPTIND - 1 ));
```

```
# Yes
local args="$(getopt -o a:b:c -l "ay:,bee:,cee" -n "$0" -- "$@")"
#Bad arguments
if [[ $? -ne 0 ]]; then
    exit 1
fi
eval set -- "$args";
while true; do
    case "$1" in
        -a|--ay)
            shift;
            if [[ -n "$1" ]]; then
                echo "-a used: $1";
                shift;
            fi
            ;;
        ...
        --)
            # end of options
            shift;
            break;
            ;;
    esac
done
```

Always check the input

Never blindly assume anything. If you want the user to input a number, **check the input** for being a number, check for leading zeros, etc... Users will do what they want, not what the program wants. If you have specific format or content needs, **always check the input**

Functions

Define local variables with **local** and one per line

```
local var1 \  
    var2 \  
    var3
```

Or

```
local var1  
local var2  
local var3
```

Define the parameter holder vars first

```
my_func() {  
    local my_param1="${1:?}"  
    local my_param2="${2:?}"  
    local other1 \  
        other2  
    ...  
}
```

Make parameters required or set default

```
my_func() {  
    # required  
    local my_param1="${1:?}"  
    # default  
    local my_param2="${2:-default value}"  
    ...  
}
```

If able, pass all the information as parameters

By maintaining the encapsulation, the reusability, maintainability and the debuggability (if any of those words exist) are greatly increased.

```
# No  
my_func() {  
    ls "$DIR"  
    ...  
}
```

```
# Yes  
my_func() {  
    local my_dir="${1?}"  
    ls "$my_dir"  
    ...  
}
```

Almost no functions should use exit, use return instead

Using exit will end any program that calls the function and will not allow it to properly react on the event of a failure.

```
# No  
my_func() {  
    [[ -e /dummy ]] \  
    || exit 1  
}
```

```
# Yes
my_func() {
    [[ -e /dummy ]] \
    || return 1
}
```

Prefer return statements to implicit return

That helps the debuggability and avoids returning unexpected values.

```
# No
# the return code here is always 1
my_func() {
    [[ -e /dummy ]] \
    && return 1
}
```

```
# Yes
my_func() {
    [[ -e /dummy ]] \
    && return 1
    return 0
}
```

Use lowercase and underscores for vars and function names

[Privacy policy \(/privacy-policy.html\)](#)


• [About \(/community/about.html\)](#)

• [Disclaimers](#)

[\(/general-disclaimer.html\)](#)

```
my_func() {
  © 2013–2022 oVirt
}
my_var="bla"
```

 [Report an issue with this page \(https://github.com/oVirt/ovirt-site/issues/new?labels=content&title=Issue:%20/develop/infra/infra-bash-style-guide.html&template=\)](https://github.com/oVirt/ovirt-site/issues/new?labels=content&title=Issue:%20/develop/infra/infra-bash-style-guide.html&template=)

 [Edit this page \(https://github.com/oVirt/ovirt-site/edit/main/source/develop/infra/infra-bash-style-guide.md\)](https://github.com/oVirt/ovirt-site/edit/main/source/develop/infra/infra-bash-style-guide.md)

Note on portability

If you can imagine a reason where your script is going to be executed on a machine where bash is not available (most common Linux distributions and GNU-based systems have bash as default shell, and can be easily installed on many others), you should use the POSIX standard.