Design patterns or best practices for shell scripts [closed]

Asked 16 years, 3 months ago Modified 1 year, 11 months ago

Viewed 81k times

186

As it currently stands, this question is not a good fit for our Q&A format. We expect answers to be supported by facts, references, or expertise, but this question will likely solicit debate, arguments, polling, or extended discussion. If you feel that this question can be improved and possibly reopened, visit the help center for guidance.

Closed 12 years ago.

Does anyone know of any resources that talk about best practices or design patterns for shell scripts (sh, bash etc.)?

design-patterns **bash** **shell**

Share

Improve this question

Follow

edited Feb 14, 2017 at 15:47

nbro

16k ● 34 ● 119 ● 212

asked Sep 17, 2008 at 0:00

user14437
3,190 ● 5 ● 22 ● 13

3 I just wrote up a little article on template pattern in BASH last night. See what you think. – quickshiftin Jan 28, 2014 at 16:02

8 Answers

Sorted by: Highest score (default)

242

I wrote quite complex shell scripts and my first suggestion is "don't". The reason is that is fairly easy to make a small mistake that hinders your script, or even make it dangerous.

That said, I don't have other resources to pass you but my personal experience. Here is what I normally do, which is overkill, but tends to be solid, although *very* verbose.

Invocation

make your script accept long and short options. be careful because there are two commands to parse options, getopt and getopts. Use getopt as you face less trouble.

```
CommandLineOptions__config_file=""
CommandLineOptions__debug_level=""

getopt_results=`getopt -s bash -o c:d:: --long config_
"$@"`

if test $? != 0
```

```
then
    echo "unrecognized option"
    exit 1
fi

eval set -- "$getopt_results"

while true
do
    case "$1" in
        --config_file)
            CommandLineOptions__config_file="$2";
            shift 2;
            ;;
        --debug_level)
            CommandLineOptions__debug_level="$2";
            shift 2;
            ;;
        --)
            shift
            break
            ;;
        *)
            echo "$0: unparseable option $1"
            EXCEPTION=$Main__ParameterException
            EXCEPTION_MSG="unparseable option $1"
            exit 1
            ;;
    esac
done

if test "x$CommandLineOptions__config_file" == "x"
then
    echo "$0: missing config_file parameter"
    EXCEPTION=$Main__ParameterException
    EXCEPTION_MSG="missing config_file parameter"
    exit 1
fi
```

Another important point is that a program should always return zero if completes successfully, non-zero if something went wrong.

Function calls

You can call functions in bash, just remember to define them before the call. Functions are like scripts, they can only return numeric values. This means that you have to invent a different strategy to return string values. My strategy is to use a variable called RESULT to store the result, and returning 0 if the function completed cleanly. Also, you can raise exceptions if you are returning a value different from zero, and then set two "exception variables" (mine: EXCEPTION and EXCEPTION_MSG), the first containing the exception type and the second a human readable message.

When you call a function, the parameters of the function are assigned to the special vars $0, $1 etc. I suggest you to put them into more meaningful names. declare the variables inside the function as local:

```
function foo {
   local bar="$0"
}
```

Error prone situations

In bash, unless you declare otherwise, an unset variable is used as an empty string. This is very dangerous in case of typo, as the badly typed variable will not be reported, and it will be evaluated as empty. use

```
set -o nounset
```

to prevent this to happen. Be careful though, because if you do this, the program will abort every time you evaluate an undefined variable. For this reason, the only way to check if a variable is not defined is the following:

```
if test "x${foo:-notset}" == "xnotset"
then
    echo "foo not set"
fi
```

You can declare variables as readonly:

```
readonly readonly_var="foo"
```

Modularization

You can achieve "python like" modularization if you use the following code:

```
set -o nounset
function getScriptAbsoluteDir {
    # @description used to get the script path
    # @param $1 the script $0 parameter
    local script_invoke_path="$1"
    local cwd=`pwd`

    # absolute path ? if so, the first character is a
    if test "x${script_invoke_path:0:1}" = 'x/'
    then
        RESULT=`dirname "$script_invoke_path"`
    else
        RESULT=`dirname "$cwd/$script_invoke_path"`
    fi
}

script_invoke_path="$0"
```

```
script_name=`basename "$0"`
getScriptAbsoluteDir "$script_invoke_path"
script_absolute_dir=$RESULT

function import() {
    # @description importer routine to get external fu
    # @description the first location searched is the
    # @description if not found, search the module in
$SHELL_LIBRARY_PATH environment variable
    # @param $1 the .shinc file to import, without .sh
    module=$1

    if test "x$module" == "x"
    then
        echo "$script_name : Unable to import unspecif
        exit 1
    fi

    if test "x${script_absolute_dir:-notset}" == "xnot
    then
        echo "$script_name : Undefined script absolute
getScriptAbsoluteDir? Dying."
        exit 1
    fi

    if test "x$script_absolute_dir" == "x"
    then
        echo "$script_name : empty script path. Dying.
        exit 1
    fi

    if test -e "$script_absolute_dir/$module.shinc"
    then
        # import from script directory
        . "$script_absolute_dir/$module.shinc"
    elif test "x${SHELL_LIBRARY_PATH:-notset}" != "xno
    then
        # import from the shell script library path
        # save the separator and use the ':' instead
        local saved_IFS="$IFS"
        IFS=':'
        for path in $SHELL_LIBRARY_PATH
        do
            if test -e "$path/$module.shinc"
```

```
            then
                . "$path/$module.shinc"
                return
            fi
        done
        # restore the standard separator
        IFS="$saved_IFS"
    fi
    echo "$script_name : Unable to find module $module
    exit 1
}
```

you can then import files with the extension .shinc with the following syntax

import "AModule/ModuleFile"

Which will be searched in SHELL_LIBRARY_PATH. As you always import in the global namespace, remember to prefix all your functions and variables with a proper prefix, otherwise you risk name clashes. I use double underscore as the python dot.

Also, put this as first thing in your module

```
# avoid double inclusion
if test "${BashInclude__imported+defined}" == "defined
then
    return 0
fi
BashInclude__imported=1
```

Object oriented programming

In bash, you cannot do object oriented programming, unless you build a quite complex system of allocation of

objects (I thought about that. it's feasible, but insane). In practice, you can however do "Singleton oriented programming": you have one instance of each object, and only one.

What I do is: i define an object into a module (see the modularization entry). Then I define empty vars (analogous to member variables) an init function (constructor) and member functions, like in this example code

```
# avoid double inclusion
if test "${Table__imported+defined}" == "defined"
then
    return 0
fi
Table__imported=1

readonly Table__NoException=""
readonly Table__ParameterException="Table__ParameterEx
readonly Table__MySqlException="Table__MySqlException"
readonly Table__NotInitializedException="Table__NotIni
readonly
Table__AlreadyInitializedException="Table__AlreadyInit

# an example for module enum constants, used in the my
readonly Table__GENDER_MALE="GENDER_MALE"
readonly Table__GENDER_FEMALE="GENDER_FEMALE"

# private: prefixed with p_ (a bash variable cannot st
p_Table__mysql_exec="" # will contain the executed mys

p_Table__initialized=0

function Table__init {
    # @description init the module with the database p
    # @param $1 the mysql config file
    # @exception Table__NoException, Table__ParameterE
```

```
    EXCEPTION=""
    EXCEPTION_MSG=""
    EXCEPTION_FUNC=""
    RESULT=""

    if test $p_Table__initialized -ne 0
    then
        EXCEPTION=$Table__AlreadyInitializedException
        EXCEPTION_MSG="module already initialized"
        EXCEPTION_FUNC="$FUNCNAME"
        return 1
    fi


    local config_file="$1"

      # yes, I am aware that I could put default param
but I am lazy today
      if test "x$config_file" = "x"; then
          EXCEPTION=$Table__ParameterException
          EXCEPTION_MSG="missing parameter config file
          EXCEPTION_FUNC="$FUNCNAME"
          return 1
      fi


    p_Table__mysql_exec="mysql --defaults-file=$config
column-names -e "

    # mark the module as initialized
    p_Table__initialized=1

    EXCEPTION=$Table__NoException
    EXCEPTION_MSG=""
    EXCEPTION_FUNC=""
    return 0

}

function Table__getName() {
    # @description gets the name of the person
    # @param $1 the row identifier
    # @result the name
```

```
    EXCEPTION=""
    EXCEPTION_MSG=""
    EXCEPTION_FUNC=""
    RESULT=""

    if test $p_Table__initialized -eq 0
    then
        EXCEPTION=$Table__NotInitializedException
        EXCEPTION_MSG="module not initialized"
        EXCEPTION_FUNC="$FUNCNAME"
        return 1
    fi

    id=$1

      if test "x$id" = "x"; then
          EXCEPTION=$Table__ParameterException
          EXCEPTION_MSG="missing parameter identifier"
          EXCEPTION_FUNC="$FUNCNAME"
          return 1
      fi

    local name=`$p_Table__mysql_exec "SELECT name FROM
      if test $? != 0 ; then
        EXCEPTION=$Table__MySqlException
        EXCEPTION_MSG="unable to perform select"
        EXCEPTION_FUNC="$FUNCNAME"
        return 1
      fi

    RESULT=$name
    EXCEPTION=$Table__NoException
    EXCEPTION_MSG=""
    EXCEPTION_FUNC=""
    return 0
}
```

Trapping and handling signals

I found this useful to catch and handle exceptions.

```
function Main__interruptHandler() {
    # @description signal handler for SIGINT
    echo "SIGINT caught"
    exit
}
function Main__terminationHandler() {
    # @description signal handler for SIGTERM
    echo "SIGTERM caught"
    exit
}
function Main__exitHandler() {
    # @description signal handler for end of the progr
    # probably redundant call, we already call the cle
    exit
}

trap Main__interruptHandler INT
trap Main__terminationHandler TERM
trap Main__exitHandler EXIT

function Main__main() {
    # body
}

# catch signals and exit
trap exit INT TERM EXIT

Main__main "$@"
```

Hints and tips

If something does not work for some reason, try to reorder the code. Order is important and not always intuitive.

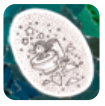
do not even consider working with tcsh. it does not support functions, and it's horrible in general.

Please note: If you have to use the kind of things I wrote here, it means that your problem is too complex to be solved with shell. use another language. I had to use it due to human factors and legacy.

Share Improve this answer

Follow

edited Dec 22, 2022 at 10:29



starball

48.2k ● 28 ● 185 ● 846

answered Apr 10, 2009 at 22:14

Stefano Borini

143k ● 100 ● 308 ● 446

8 Wow, and I thought I was going for overkill in bash... I tend to use isolated functions and abuse subshells (thus I suffer when speed is in any way relevant). No global variables ever, neither in nor out (to preserve remains of sanity). All returns through stdout or file output. set -u/set -e (too bad set -e becomes useless as soon as first if, and most of my code often is in there). Function arguments taken with [local something="$1"; shift] (allows for easy reordering when refactoring). After one 3000 lines bash script I tend to write even smallest scripts in this fashion... – Eugene Aug 22, 2009 at 10:10

small corrections for modularization : 1 you need a return after . "$script_absolute_dir/$module.shinc" to avoid missing warning. 2 you must set IFS="$saved_IFS" before your return on finding module in $SHELL_LIBRARY_PATH – Duff May 2, 2013 at 12:43

"human factors" are the worst ones. Machines don't fight you when you give them something better. – jeremyjjbrown Dec 18, 2014 at 23:20

1 Why `getopt` vs `getopts`? `getopts` is more portable and works in any POSIX shell. Especially since the question

is *shell best practices* instead of specifically bash best practices, I would support POSIX compliance to support multiple shells when possible. – Wimateeka Sep 5, 2017 at 14:41

1 thanks for offering all the advice for shell scripting even though you are being honest :"Hope it helps, although please note. If you have to use the kind of things I wrote here, it means that your problem is too complex to be solved with shell. use another language. I had to use it due to human factors and legacy." – dieHellste Nov 15, 2017 at 10:59

28

Take a look at the Advanced Bash-Scripting Guide for a lot of wisdom on shell scripting - not just Bash, either.

Don't listen to people telling you to look at other, arguably more complex languages. If shell scripting meets your needs, use that. You want functionality, not fanciness. New languages provide valuable new skills for your resume, but that doesn't help if you have work that needs to be done and you already know shell.

As stated, there aren't a lot of "best practices" or "design patterns" for shell scripting. Different uses have different guidelines and bias - like any other programming language.

Share Improve this answer

Follow

answered Sep 17, 2008 at 2:44

jtimberman

8,258 ● 2 ● 45 ● 37

10 Note that for scripts of even slight complexity, that is NOT a best practice. Coding is not about just getting something to work. It is about building it quickly, easily, and it being reliable, reusable, and easy to read and maintain (especially for others). Shell scripts do not scale well to any level. More robust languages are much simpler for projects with any logic. – Sierra Dec 9, 2011 at 13:26

20

shell script is a language designed to manipulate files and processes. While it's great for that, it's not a general purpose language, so always try to glue logic from existing utilities rather than recreating new logic in shell script.

Other than that general principle I've collected some common shell script mistakes.

Share Improve this answer

Follow

answered Sep 17, 2008 at 23:57

pixelbeat

31.7k ● 9 ● 53 ● 62

13

Know when to use it. For quick and dirty gluing commands together it's okay. If you need to make any more than few non-trivial decisions, loops, anything, go for Python, Perl, and **modularize**.

The biggest problem with shell is often that end result just looks like a big ball of mud, 4000 lines of bash and growing... and you can't get rid of it because now your

whole project depends on it. Of course, **it started at 40 lines** of beautiful bash.

Share Improve this answer

Follow

answered Sep 17, 2008 at 19:30

Paweł Hajdan

18.5k ● 9 ● 52 ● 65

12

There was a great session at OSCON this year (2008) on just this topic:
http://assets.en.oreilly.com/1/event/12/Shell%20Scripting%20Craftsmanship%20Presentation%201.pdf

Share Improve this answer

Follow

edited Apr 10, 2009 at 22:50

Jonathan Leffler

752k ● 145 ● 946 ● 1.3k

answered Sep 17, 2008 at 4:57

Fhoxh

778 ● 3 ● 17

10

use set -e so you don't plow forward after errors. Try making it sh compatible without relying on bash if you want it to run on not-linux.

Share Improve this answer

Follow

answered Sep 17, 2008 at 0:05

user10392

1,392 ● 8 ● 12

9

Easy: use python instead of shell scripts. You get a near 100 fold increase in readablility, without having to complicate anything you don't need, and preserving the ability to evolve parts of your script into functions, objects, persistent objects (zodb), distributed objects (pyro) nearly without any extra code.

Share Improve this answer

Follow

answered Sep 17, 2008 at 0:02

Joao S O Bueno

8 you contradict yourself by saying "without having to complicate" and then listing the various complexities you think add value, while in most cases are abused into ugly monsters instead of used to simplify problems and implementation. – Evgeny Zislis Feb 8, 2010 at 18:39

3 this implies a great drawback, your scripts will not be portable on systems where python is not present – astropanic Aug 25, 2011 at 21:48

1 I realize this was answered in '08 (it is now two days before '12); however, for those looking at this years later, I would caution anyone against turning their back on languages like Python or Ruby as it is more likely that it is available and if not, it is a command (or couple clicks) away from being installed. If you need further portability, think about writing your program in Java as you will be hard-pressed to find a machine that doesn't have a JVM available. – Wil Moore III Dec 29, 2011 at 21:14

@astropanic pretty much all Linux ports with Python nowadays – Pithikos Mar 7, 2017 at 10:31

@Pithikos, sure, and fiddle with the hassle of python2 vs python3. Nowadays I writte all my tools with go, and can't be

happier. – astropanic Mar 7, 2017 at 11:18

7

To find some "best practices", look how Linux distro's (e.g. Debian) write their init-scripts (usually found in /etc/init.d)

Most of them are without "bash-isms" and have a good separation of configuration settings, library-files and source formatting.

My personal style is to write a master-shellscript which defines some default variables, and then tries to load ("source") a configuration file which may contain new values.

I try to avoid functions since they tend to make the script more complicated. (Perl was created for that purpose.)

To make sure the script is portable, test not only with #!/bin/sh, but also use #!/bin/ash, #!/bin/dash, etc. You'll spot the Bash specific code soon enough.

Share Improve this answer

Follow

answered Sep 17, 2008 at 20:33

Willem

918 ● 5 ● 10

Highly active question. Earn 10 reputation (not counting the association bonus) in order to answer this question. The reputation requirement helps protect this question from spam and non-answer activity.