> /DEV/NULL

2020/03/04

DISCLAIMER

- This presentation is not intended to be a complete guide
- I am self-taught in bash, I may show fiddle-with-it approach on some parts

PLAN

1. What is bash?
2. How is it used at Talend?
3. How should I write it?
4. Cheatsheet
5. Scripting advice
6. Pitfalls
7. Bash swiss knife
8. Portability

WHAT IS BASH?

- Free, open source Unix Shell
- Initially written by Brian Fox for GNU Project in 1989
- Replacement for Bourne Shell
- POSIX compliant
- Bash actually means Bourne-Again Shell
- Version installed by default on Mac is 3.2 (2006) for licensing issues
- Version installed by default on most linux distribs is 4.4 (2016)
- Last available version is 5.5 (2019)

Speaker notes

Bourne Shell = sh. Bash copies most features from sh and adds a few of its own like history.

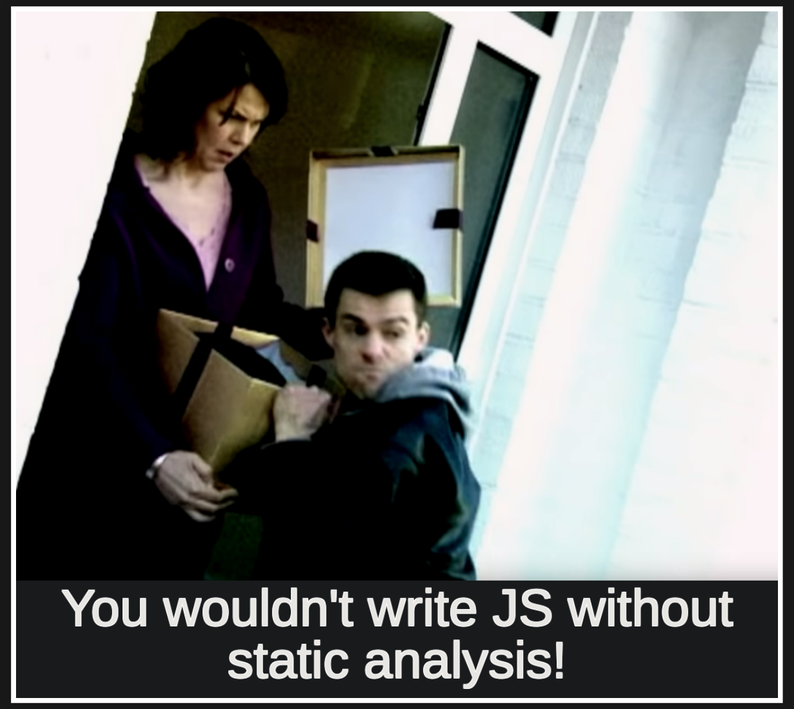
POSIX is a set of standards aimed at making OSes compatible. It defines an API for software compatibility.

Mac users CAN and SHOULD install Bash 4.4. Apple just isn't allowed to do so on the computers they sell.

HOW IS IT USED AT TALEND?

- On our AWS machines
- On CI for builds
- On developer computers for basic tooling
- On the demo stack via SSH

HOW SHOULD I WRITE IT?



LINT AWAY!

Shellcheck to the rescue!

```
vidar@vidarholen ~ $ shellcheck myscript

In myscript line 7:
if (( $n > 3.5 ))
      ^-- Don't use $ on variables in (( )).
          ^-- (( )) doesn't support decimals. Use bc or awk.


In myscript line 16:
[[ $1 == $result ]] && mode=lookup
         ^-- Quote the rhs of = in [[ ]] to prevent glob interpretation.

vidar@vidarholen ~ $
```

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

main $@
     Double quote array expansions to avoid re-splitting elements. [SC2068]
```

SHELLCHECK IN EDITORS

Shellcheck is available on:

- Vim
- Emacs
- Sublime
- Atom
- VSCode
- Most editors via GCC error compatibility

No excuses!

CHEATSHEET

A cheatsheet you can come back to at any moment

A longer course can be found on MIT's website

PROCESSES INPUTS

For programs that read standard input

```
sort <   hello.txt          # Sorts hello.txt
sort <<< "$var"             # Sorts content of variable var
```

PROCESSES OUTPUTS

For all programs that write to standard output

```
printf 'hello' >  out.txt          # Writes standard output (hello) to out.txt
printf 'hello' >> out.txt          # Appends standard output (hello) to out.txt
command 2> err.txt                 # Writes standard error to err.txt
command &> all.txt                 # Writes standard output & error to all.txt
printf 'hello' > /dev/null         # Discards standard output
```

MIXING INPUT/OUTPUT

```
printf 'hello' | grep 'toto'      # Pipes standard output (hello) to grep
```

VARIABLES

```
name='Toto'                    # /!\ No spaces around = in bash assignations /!\
printf "Hello $name\n"         # Variable is substituted in double quotes, prints 'Hello Toto'
printf 'Hello $name\n'         # Variable is not substituted in simple quotes, prints 'Hello $name'
```

FUNCTIONS

```
functionName() {    # Classic syntax for declaration
  scriptName=$0     # $0 is the script name
  firstArgument=$1  # $n is the script's nth argument
  allArgs=$@        # $@ is all the arguments
  argsNumber=$#     # $# is the number of arguments
}
```

RETURN CODES

```
grep toto <<< 'toto'          # Return code is 0 === success
grep toto <<< 'tata'          # Return code is 1. Any code !== 0 is an error
lastReturnCode=$?             # $? contains the return code of the last command
command && { printf 'OK'; } # The block after && executed if command succeeds
command || { printf 'KO'; } # The block after || executed if command fails
```

SHARE RESULTS

```
# $() allows to retrieve the standard output of a method
greeting="$(printf "Hello $name")"
# Less known, <() puts the standard output of a method in a temp file
greetingFile=<(printf "Hello $name")
# greetingFile is the path to a temporary file where the greeting was written
```

CONDITIONALS

Conditionals in bash are expressions.
Return code 0 = true, any other is false

```
if grep --silent 'toto' <<< 'tata'; then
  # Executed if the grep returned 0
else
  # Executed if the grep returned anything but 0
fi
[[ "$var" = 'toto' ]]                  # Expressions inside double brackets return 0 if true, non-zero otherwise
[[ "$var" = 'toto' || "$var" = 'tata' ]] # Composite conditionals
```

Speaker notes

Everything available inside double brackets is from test command

GLOBBING

In a folder containing

```
.
├── bar
├── img.png
├── img.jpg
├── img.svg
├── foo1
├── foo2
├── foo3
└── foo99
```

```
rm foo?           # Removes foo1, foo2 & foo3
rm foo??          # Removes foo99
rm foo*           # Removes foo1, foo2, foo3 & foo99
rm img.{svg,png} # Removes img.svg & img.png
rm img.*          # Removes img.svg img.jpg & img.png
rm foo{1..2}      # Removes foo1 & foo2
rm foo{*,99}      # Removes ?
# Only foo99!
```

Note: using globbing is often better than ls | grep

REGEX

```
[[ 'toto' =~ (to){3} ]]     # Returns 0, the string matches the regex
regex='(to){3}'
[[ 'toto' =~ $regex ]]      # Returns 1, the string does not match the regex
[[ '(to){3}' =~ "$regex" ]] # Returns 0, the regex is matched as a string /!\
```

PARAMETER SUBSTITUTION - BOURNE

```
# Before starting: $var = ${var}, $1 = ${1}
# Substitutions look like this: ${<varName><subsitutionCharacter><fallbackIfVariableIsUnset>}

# For substitutions below, adding : before the substitution character adds empty case to failure cases
# Otherwise, only unset values fall in failure cases
toto=${var1-Nope}                # fallback value (Nope) if var1 is unset
toto=${1?Missing parameter toto} # error with message if $1 is unset

length=${#var}                   # returns the length of $var
```

PARAMETER SUBSTITUTION - BASH

```
offset=${var:2}                 # returns the value of var, starting with an offset of 2
offset=${var:2:5}               # same but only returns 5 characters from the offset start

replaced=${var/[0-9]/?}         # replaces the first number in $var with ?
replaced=${var//[0-9]/?}        # replaces all numbers in $var with ?

# /!\ The commands below only work with Bash 4+
upperCaseVar=${var^^}
lowerCaseVar=${var,,}
```

More on parameter substitution

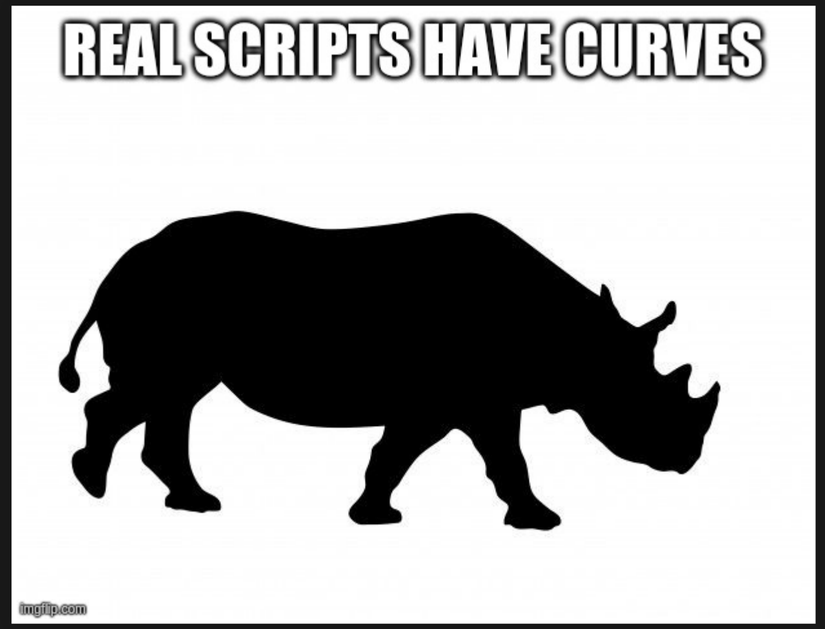
You probably know enough though

ARRAYS

```
declare -a array         # Create indexed array: keys are integers
declare -A array         # Create declarative array: keys are whatever the hell ya want
array=()                 # Create an indexed array without values
array=('titi tata' toto) # Create an indexed array with values

arraySize=${#array[@]}      # Gives the size: 2
array3rdElement=${array[1]} # Gives the second element, toto

array+=(tutu tete)                    # Appends to an array
array[2]=(tyty)                       # Updates array item
for item in "${array[@]}"; do ... done # Loops on an array
sliced=("${array[@]:1:2}")            # Slices array, offset 1, 2 elements

declare -p array     # Log the array in the form: declare -a array=([0]="titi tata" [1]="toto")
printf "${array[@]}" # A simpler version but less clear: titi tata toto
```

SCRIPTING ADVICE

SCRIPT SHAPE



SCRIPT SHAPE

Goals

- Readable
- Usable
- Debuggable

SCRIPT SHAPE

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

set -xe
test -f '/path/to/file' && { source "$_"; }

main() {
  if isHelp; then displayHelpAndReturn; fi

  local arg1="${1?Missing path to lib}"
  local arg2="${2?Missing bla bla}"

  _importedCommand "$arg1"
  command1 "$arg1"
  command2 "$arg2"
}

command1() {
  local arg1="$1"
  ...
}

command2() { ... }

main "$@"
```

Speaker notes

- **Shebang** So the system knows how to execute it. Bash for bash, sh for Bourne!

  You can execute it with `./script.sh` but it requires a `chmod +x ./script.sh` first

- **set -xe** Eq: `set -x; set -e` Only when necessary (ex: CI)
- **Import files** For re-usability. Test the existence before
- **main part** Should be almost literal, this is were you understand what it does at a glance
- **help** For interactive scripts. For CI, comments are enough.
- **args** Get and check your args before doing anything
- **body** Literal, as said earlier
- **Script utilities** To hide unimportant implementation details and EXPLAIN
- **main call** Call the main method, passing all arguments passed to the script

CONVENTIONS

Disclaimer: these are my own, they've helped me a lot tho

Rule	Reason
Use double quotes only when there is a substitution	Makes it easier to spot constants from templated strings
Quote everything unless you have a good reason not to	Minimizes errors due to word splitting
Prefix external methods/constants with _	Makes it easier to spot them and find where they are implemented
Casing: upper-snake-case for constants and global variables. Lower-camel-case otherwise.	Makes them easier to differentiate

Rule	Reason

Use full flags in CLI tools	More explicit, ex:
	`jq -r`
	vs.
	`jq --raw-output`

PITFALLS

DEPRECATED SYNTAXES

```
files=`ls`    # Back-tick syntax is deprecated in bash
files="$(ls)" # The syntax to use
```

Explanation of the back-tick deprecation

```
[ -n "$fileName" ]   # Uses /usr/bin/[
[[ -n "$fileName" ]] # Prefer built-in syntax, can't be messed with
test -n "$fileName"  # This works too
```

Explanation of why you should use double brackets for conditionals

VARIABLE NAMES AND SIDE-EFFECTS

Some variable names are reserved Some are used by other tools In doubt, always use super-precise names for upper-case variables. Also, initialize them or anyone can change the behavior of your script unintentionally.

IMPORTS

Imports are realized from cwd (current work directory) in bash!

```
# Bad way
source ./utils.sh # <== Resolves to ~/utils.sh if executed from ~ and /tmp/utils.sh if executed from /tmp!


# Better way
dir="$( cd "$( dirname "${BASH_SOURCE[0]}" )" && pwd )" # Gets script's directory, see one-liners section
source "$dir/utils.sh"
```

SUBSHELLS - WHAT IS IT?

A subshell is a separate instance of the command processor (here, bash)

SUBSHELLS - HOW DO I CREATE ONE?

```
( cd /tmp; ls ) # Anything inside parenthesis happens in a subshell, goes for $() too

# Piping creates a subshell
cat file.txt | while read -r line; do
  # In a subshell here
done

# This does not create a subshell
while read -r line; do
  # Not in a subshell here
done < file.txt
```

Speaker notes

For the attentive reader, the flag -r in read disables escaping of line breaks with \

This prevents read from reading the following code as a single line

```
cat toto.txt \
  | sort
```

SUBSHELLS - WHAT ARE THE IMPACTS?

When in a subshell, you can't modify the state from outside, meaning

```
( cd /tmp; ls ) # I won't be in /tmp after this line, only the subshell got cded

var=toto
cat file.txt | while read -r line; do
  var=tata # NAY! The value is still toto, can't touch dat!
done

while read -r line; do
  var=tutu # That's OK!
done < file.txt
```

SET -E

This one's a tricky one, you need to protect against it.

```
set -e # The script fails on any uncaught errors from now on

# What do you think this'll do?
grep --silent toto <<< 'tata' && { printf 'OK!\n'; }
# It FAILS HAHAHA! Error case is not caught

grep --silent 'a' <<< 'b' || { printf 'OK!\n'; } # The error is caught here
# If in doubt, always use if, it catches all errors
```

BASH SWISS KNIFE

A nice collections of things to know/install

COMMANDS AND ONE-LINERS

Save them somewhere!

```
# Create a temporary file and writes its path on stdout
mktemp -f toto-XXX.txt # The XXX are replaced by random characters

# Retrieve the path to the file being executed
dir="$( cd "$( dirname "${BASH_SOURCE[0]}" )" && pwd )"
# What it does:
# * ${BASH_SOURCE[@]} contains the path to the current script
# * dirname           extracts the containing folder path
# * cd                cds to it
# * pwd               prints the absolute path to the current directory
```

TIPS & TRICKS

```
# Escape any character with \
command | uniq \      # Can improve readibility
        | sort \
        > output.txt
cat ~/File\ from\ a\ windaube\ user\ troll.txt # Or use file names with spaces

# Group outputs with blocks
command1 >  output.txt # Big chance of using a > by mistake later
command2 >> output.txt
command2 >  output.txt # And ruining the begining of the file, oops!

{
  command1; command2; command3
} > output.txt # No risk of error!

# Pushd with auto-popd!
set -e
(
  cd "$folder"
  commandThatMayFailAndWouldNotResetCwdIfInMainShell
)
```

I'll enrich this section as I go along

USEFUL PACKAGES

- curl THE most used CLI HTTP client
- htop process inspector
- fd to replace find
- rg (ripgrep) to replace grep
- tree replaces ls for nested folders
- jq JSON parser (use 1.5+ to keep property order)
- yq YAML equivalent of jq
- xmlstarlet XML parser

PORTABILITY

Making sure it executes properly on all systems by avoiding non-portable commands

Command	Reason	Recommandation
sed	GNU version (Linux) and FreeBSD (Mac) differ on some flags. Syntax hard to read.	For structured formats (JSON etc) use a real parser. Otherwise, use awk
echo	Some flags (-e) don't behave the same on linux and mac	Use printf
readlink	Mac does not have the GNU version of readlink	`resolvedLink="$(cd "$path" && pwd -P)"`
Direct shebangs	Programs are not always installed at the same place	`#!/usr/bin/env python` `# Instead of /usr/bin/python for example`

Shell flags not in shebangs	Some systems will ignore them or crash	`#!/usr/bin/env bash`
		`set -e`
		`# Instead of #!/usr/bin/env bash -e`

Speaker notes

The developer computer thing implies we must make it portable!

NICE SOURCES

- The missing semester of your CS education
- The bash hacker wiki

Speaker notes

Note: the first one is digestible easily, the second one less so.

THANKS FOR YOUR ATTENTION!

Please don't hesitate to slack me any cool/tricky thing you know about bash that's not here!

Slack handle: @cyp