SysDev - Part 2 - Introduction to scripting

Michel FACERIAS

20 décembre 2023

Polytech Montpellier

Université de Montpellier

Table des matières

1 From uniline to *bash* script

A ***bash* script** is a script **interpreted by the current CLI interpreter**.

Remember that **when you will buy new droids, they must speak *bash*** (bocce !)
https://www.youtube.com/watch?v=eUH2_n8jE70 (Ester egg is near second 42 !).

We will use a concept **already seen in SysAdmin** part 2 to introduce the notion of script.

1.1 Uniline (Reminder)

1.1.1 Concept : Uniline

 We recall here that it is possible to pass two (or more) commands in the same command line. You must separate them using **semicolon** (;).
It is called **uniline commands**.

1.1.2 To do : Trying uniline, one more time !

 Try separately `date` and `whoami` commands.
Then type the following dual command and execute it :

```
$ date ; whoami
```

1.2 First script

We are going to transform the uniline in a srcipt.

1.2.1 Concept : What is a script ?

 **A script is** neither more nor less than **a sequence of commands stored in a file**.
Where in an **uniline**, it is the **semicolon** (;) that separate up the sequence of successive instructions, **in a script, this is done with a line feed**.

1.2.2 To do : Write your first script

 Use your favorite editor to create the `first_script` file and fill it with :

```
date
whoami
```

Then run it :

```
$ bash first_script
jeu. 05 janv. 2023 14:57:17 CET
mfacerias
```

Now compare with the result obtained with the uniline.
It's identical, good job ! You have made your first cript !

1.2.3 Concept : Blanks, empty lines and comments

 All **empty lines are ignored**.

The same applies to **everything after # until the end of the current line** and multiple blank characters.

1.2.4 To do : Add blank lines and comments

 Use your favorite editor to modify the `first_script` file and fill it with :

```
# this sis a comment

 date # here, we show the date and time, and we add a space before the command too

  whoami # here, we show the current user, and we add a tab before the comamnd too


# all empty lines above were ignored
```

Then run it :

```
$ bash first_script
jeu. 05 janv. 2023 14:59:17 CET
mfacerias
```

The concept is validated !

1.3 From script to program

To run our script, we **launched an interpreter**, in this case `bash`, with the **path to our script as argument**.

It would be smarter to **be able to launch it by referring to the script itself**.

1.3.1 Concept : Shebang

 The **shebang** is a particular sequence, placed **on the first line** of a script that informs about **wich interpreter** must be used to execute it.

Then, to **make this script executable**, the user who launches the command **must also have the `x` right** (execute) on this script.

 If **no shebang** is used, the script is **interpreted by the current shell** !

1.3.2 To do : First program

 Use your favorite editor to modify the `first_script` file and fill it with :

```
#!/bin/bash

date

whoami
```

Then add rights, and run it :

```
$ chmod u+x firts_script

$ ./first_script
jeu. 05 janv. 2023 14:59:17 CET
mfacerias
```

⚠ The script is launched using a relative path. Here the script is in our working directory. You **MUST** use a **full referenced relative path**, like `./first_script`.

1.3.3 To do : Good practices

 Some says that a *bash* script must be **well named** !

Try this :

```
$ mv first_script first_script.sh

$ ./first_script.sh
```

Your script run as well as before... **Linux ignored the "extension" in file names**. But it remains easyer for the user !

1.4 More about scripts and shebang

The **shebang** is not only used for bash scripts, but it should be whith all interpreted languages.

1.4.1 To do : Python script

 Use your favorite editor to create the `python_script.py` file and fill it with :

```
#!/usr/bin/python3

print("Hello World")
```

Then add rights, and run it :

```
$ chmod u+x python_script.py

$ ./python_script.py
Hello World
```

1.4.2 To do : PHP script

 Use your favorite editor to create the `php_script.php` file and fill it with :

```
#!/usr/bin/php

print("Hello World") ;
```

Then add rights, and run it :

```
$ chmod u+x php_script.php

$ ./php_script.php
Hello World
```

2 Using variables

2.1 User defined variables

2.1.1 Concept : Declaration

To **define a *bash* user variable** use its name and give it a value, like this :

```
a=10 # this is a string !

B=ten # this is a sting too !

value1="this is a string whith spaces" # this is a string again, using double quotes

value2='this is a string whith spaces' # this is definitivly a string, using single quotes

A=20 # A is not a, variable are case sensitives
```

As you can see, bash knows only strings !

2.1.2 Concept : Use

To **use a variable**, just precede it with $.

2.1.3 To do : Variable expansion

Create a new script file called `variables.sh`, and fill it :

```
A=10
a=20
text='Hello World'
echo $text
echo $a
echo "$A"
echo '$A'
echo \$A
b="$a $A"
echo $b
```

Then, run it :

```
$ bash variable
Hello World
20
10
$A
$A
20 10
```

As you can see :
— single quottes block the expension ;
— \force $ to be treated literaly (escaped) ;
— concatenation is easy.

2.2 Environment variables

This variables are pre-defined by the system. Some are usefull...

2.2.1 To do : Enumerate environment variables

 Use the *set* command, and analyse the result.

2.2.2 To do : Use environment variables in a script

 Create a new script file called `hello.sh`. It must show a welcome message dedicated to the current user and display the name of the machine as this :

```
# bash hello.sh
Hello mfacerias, your are using mfa-laptop}
```

2.3 Special variables

2.3.1 To do : Use of special variables

 Create a new script file called `special.sh` and fill it like this :

```
echo "There were $# command line parameters"
echo "They are: $@"
echo "Parameter 1 is: $1"
echo "The script is called: $0"
# any old process so that we can report on the exit status
pwd
echo "pwd returned $?"
echo "This script has Process ID $$"
sleep 3
echo "It has been running for $SECONDS seconds"
echo "Random number: $RANDOM"
echo "This is line number $LINENO of the script"
```

Run it using : `bash special.sh toto titi` and analyse the result.

2.3.2 To do : Shebang or not shebang

 Modify the script file called `special.sh` and fill it like this :

```
#!/bin/bash
echo "There were $# command line parameters"
echo "They are: $@"
echo "Parameter 1 is: $1"
echo "The script is called: $0"
# any old process so that we can report on the exit status
pwd
echo "pwd returned $?"
echo "This script has Process ID $$"
sleep 3
echo "It has been running for $SECONDS seconds"
echo "Random number: $RANDOM"
echo "This is line number $LINENO of the script"
```

Add execution right and run it using : `./special.sh toto titi` and analyse the result, mainly the number of command line parameters.

The "running method" has no influence!!!

2.3.3 To do : Returned value

$? variable is used to know the execution status of the last command you used :

— 0 means that everything is OK ;

— an other value gives, sometime, more information about what was wrong.

Create a script file called `errolevel.sh` and fill it like this :

```
mkdir toto
echo $?
mkdir toto
echo $?
mkdir titi/toto
echo $?
mkdir /root/toto
echo $?
```

Assuming neither `titi` and `toto` directory exists, run it using : `bash errorlevel.sh` and analyse the result.

As you can see, **some commands can not explaine what append** by there returned value.

2.4 Auto fill variable with *stdout*

Let suppose that you want to collect *stdout* of a command to a variable.

2.4.1 To do : Redirect *ls* output to a variable

Create a script file called `myls.sh` and fill it like this :

```
echo "Official ls out is :"
ls

list_of_files=$(ls)

echo "My list of files is"
echo $list_of_files
```

Run it using : `bash myls.sh` and analyse the result.

As you can see, *stdout* is captured, and the result is space-separated, instead of line separated.