Introduction au Bash scripting

David Zarebski

IMIE

2026

[0.0] ⋙ Programme

- Philophie et Spécificités
- Eléments de syntaxe
  - Variables
  - Interpolation de chaînes de caractères
  - Boucles et conditions
  - Piping et redirection
- Parsing de chaînes de caractères
  - Les outils historiques

Philophie et Spécificités

Ecosystème

Pour comprendre les spécicités de `bash`, il faut comprendre l'écosystème dans lequel il s'insère qui n'est autre qu'**UNIX**.

Les outils que nous avons rencontrés au module précédent:

- ne font qu'une chose (mais le font complètement)
- ne capturent pas l'utilisateur (une commande → une sortie, pas de fonctionnement interractif)
- écrivent leur sortie directement dans le terminal

Philosophie UNIX (1978)

Douglas McIlroy (Bell Labs Computing) résume le style de programmation de cet écosystème de la manière suivante

- **interoperabilité** : *Write programs that do one thing and do it well. Write programs to work together. Write programs to handle **text streams**, because that is a universal interface.*
- **simplicité**: *The notion of "intricate and beautiful complexities" is almost an **oxymoron**. Unix programmers vie with each other for "simple and beautiful" honors — a point that's implicit in these rules, but is well worth making overt.*

Bash dans tout cela

Si les programmes sont **simples** et ne **font qu'une chose**, comment réaliser une **opération complexe**?

`bash` (et avant lui `sh`) est ce qui fait la **glue entre ces programmes** simples et autonomes (`ls`, `cat`, ...).

En bash, **tout est une chaine de charactères** (ce qui fait sa beauté et sa laideur de ce **language procédural**) parce que les entrées et sorties des programmes UNIX en sont

Eléments de syntaxe

One script to rule them all

Au fur et à mesure que nous introduirons les **éléments de syntaxe**, nous allons faire évoluer un **script d'analyse système**

Ce script sera assez simple: nous cherchons à identifier l'origine des **certificats d'autorité installés** sur le système

Exercice: Pays d'origine des CA de votre système

Les certificats d'authorité (CA aussi appelé certificats racine) sont des **certificats SSL auto-signés** vivant en `/etc/ssl/certs/`. Tout certificat signé par un de ces certificats est accepté comme légitime et **valide** par votre système.

...

A l'aide de `openssl` analysez l'ensemble des certificats pour afficher le nom de ceux émis par une autorité européenne (resp. française, italienne).
A titre d'exemple, le certificat suivant est émis par *T-Systems Enterprise Services GmbH* (éméteur allemand)

```
openssl x509 -noout -text -in '/etc/ssl/certs/1e1eab7c.0';
...
        Subject: C=DE, O=T-Systems Enterprise Services GmbH, OU=T-Systems Trust
```

Spécifications

Le script fonctionnera avec **deux modes** possibles (**arg 1**):

- `EU`: tous les éméteurs européens
- `<ISO 3166>`: n'afficher que éméteurs du pays désigné par le code ISO 3166 (e.g. *France* = **FR**). Exception si le code est inconnu

Nous allons gérer fait que les certificats puissent ne pas vivre en `/etc/ssl/certs/` (**arg 2**)

- `CERTS_LOCATION`: default `/etc/ssl/certs/`

...

Le script doit:

- logger ses éventuelles erreurs en `/tmp/analysis-err.log`
- les resultats sont triés par ordre alphabétique
- format de lignes: `<nom><tab><ISO 3166>`

Template d'un script

```
#!/bin/bash

COUNTRY=$1

exit 0
```

- shebang (ligne 1): permet au pty de savoir dans quel programme envoyer les lignes suivantes
- stockage des arguments dans des variables locales aux noms explicites
- sortie normale (0 = succès)

Eléments de syntaxe
≫Variables

[2.1] ⋙ Deux types de variables

On distinguera les variables

- d'environnement (que l'on peut voir avec `printenv`): définies dans `/etc/environment`, `/etc/profile`, `.bashrc`
- les **variables locales**, définies comme ceci

```
SOME_VAR=1
echo $SOME_VAR
```

> pas d'espace de part et d'autres de `=`

Vos variables peuvent entrer en **conflit** avec les variables d'environement

⇒ n'utilisez pas `HOME`, `USER`, ... ou quelque autre variable existante

[2.1] ⋙ Par défaut tout est une chaines de charactères

Ces deux expressions sont **strictement équivalentes**. Il n'y a pas de notion de `int` ou de `str`

```
SOME_VAR=1
echo $SOME_VAR
SOME_VAR="1"
echo $SOME_VAR
```

...

```
SOME_VAR=$(dig imie.com)
echo $SOME_VAR
```

Il en va de même de la sortie standard d'un programme.
Vous pouvez voir `$()` comme une sous-requête ou *stocke la sortie de l'exécution dans*

Plusieurs chaines peuvent être stockées dans un tableau
Celui ci est **mutable**

> Notez l'absence de `,`

```
TABLEAU=("un" "deux" "trois")
echo ${TABLEAU[1]}
TABLEAU[1]="uno"
echo ${TABLEAU[1]}
```

...

```
CARDINAUX="un,deux,trois"
readarray -d, <<<$CARDINAUX
echo ${MAPFILE[0]}
```

Il est possible de convertir une chaine en tableau à l'aide de `readarray` (entre autre)

```
declare -A CONFIG

CONFIG[host]=127.0.0.1
CONFIG[port]=5432

printf "endpoint: ${CONFIG[host]}:${CONFIG[port]}"
```

Depuis bash 4, il est possible de définir des **tableaux associatifs**

Dans les faits, cette structure est assez peu utilisée car complexe à transposer dans d'autres shells

Eléments de syntaxe
≫Interpolation de chaînes de caractères

```
NAME=Mark

echo "Salut $NAME"
echo "Salut ${NAME}" #OK
echo 'Salut ${NAME}' #KO
```

On utilisera la forme en `${}` pour deux raisons

- elle évite que la chaine ne soit étendue/interprétée (sécurité)
- elle permet une gestion fine des valeurs par défaut et cas

...

```
foobar=un
foo=deux

echo "$foobar"
# VS
echo "${foo}bar"
```

Afin d'éviter ce genre de cas limite, utilisez le pattern `${}` qui vous **évitera des surprises**

Valeurs par defaut

```
unset $NAME
echo "hello ${NAME:-anonymous}"

unset $NAME
echo "hello ${NAME?:Error: NAME}"

NAME=
echo "hello ${NAME?Error: NAME}"
```

Il est possible

- d'attribuer une **valeur par défaut** à une variable templatée.
- de **générer une erreur** si la variable est
  - vide ou non définie (`?:`)
  - non définie (`?`)

Eléments de syntaxe
≫Boucles et conditions

```
IP=127.0.0.1

if [ $IP == 127.0.0.1 ]; then
    echo "localhost"
else
    echo "wan"
fi
```

Le **pattern général** est le suivant:

- `if` `[condition]` `;then`
- opt. `elif` `[condition]` `;then` autre condition
- opt. `else` complémentaire (i.e. tout autre cas)
- `fi` *ferme le conditionel*

L'expression entre `[ ]` est une expression **booléenne**

Puisque `[ ]` attend un booléen, rien n'oblige à ce que ce dernier soit **atomique**. On peut, en effet, combiner **plusieurs expressions booléennes** à l'aide d'**opérateurs booléen** (nécessite `[[]]`)

```
FQDN=some.domain.com

if [[ $FQDN == localhost || $FQDN == andr0m3da ]]; then
    echo 127.0.0.1
elif [[ $FQDN == *"."* && "${FQDN:0:1}" != '.' ]]; then
    echo 'Ask your DNS about it'
else
    echo 'Not a domain name'
fi
```

opérateur	bash
NOT	!
AND	&&
OR	||

Syntaxic sugar

Tant pour alléger la syntaxe que pour fournir une assistance sémantique (car ces chaines sont interprétées comme des chemins), `bash` dispose des opérateurs suivants

chaines

- `-z` : la chaine est-elle **vide**?

chaines interprétées comme des chemins vers des fichiers / dossiers

- `-e` : le **fichier** existe-t-il?
- `-d` : le **dossier** existe-t-il?
- `-f` : est-ce un **fichier régulier**? (i.e. pas un **lien symbolique**)
- `-s` : le fichier est-il non **vide**?

Application

```
TEST_DIR=/tmp/demo
TEST_FILE="${TEST_DIR}/test"

mkdir -p $TEST_DIR && touch $TEST_FILE

if [[ -d $TEST_DIR && -f $TEST_FILE ]]; then
    echo 'test file exists'
    if [ -s $TEST_FILE ]; then # si le fichier contient quelque chose
        echo "non empty file, leaving"
    else
        echo "empty file, appending"
        echo "coucou" >> $TEST_FILE
    fi
fi
```

```
for w in salut comment ça va; do
    echo $w
done
```

l'`IFS` (*Internal Field Separator*) est par défaux un **espace**
⇒ le pattern passé à `in` est tokenisé en mots séparés par un espace sur lesquels `for` boucle

...

```
DATA=$(dig imie.com)
IFS=$'\n'

for l in $DATA; do
    echo "Ligne: ${l}"
done
 # on revient au séparateur de base
IFS=
```

L'`IFS` peut être **surchargé** au besoin (ici, retour à la ligne `\n`)
Pour les **caractères spéciaux** comme `\n`, la syntaxe est un peu déroutante

[2.3] ⋙ Boucles while

Dans certains cas, il est necessaire de boucler **jusqu'à ce qu'une condition soit satisfaite** grace à `while`

```
CONTINUE=1

while [ $CONTINUE == 1 ]; do
    number=$RANDOM
    if [ $(($number % 5)) == '0' ]; then
        echo "${number} est multiple de 5"
        CONTINUE=0
    else
        echo "${number} n'est pas multiple de 5"
    fi
done
```

Ici, on tire des nombres aléatoires jusqu'à ce que l'on trouve un multiple de 5

`$(())` est la manière dont on désigne une opération arithmétique en `bash`. Toute variable passée sera **interprétée comme un nombre**

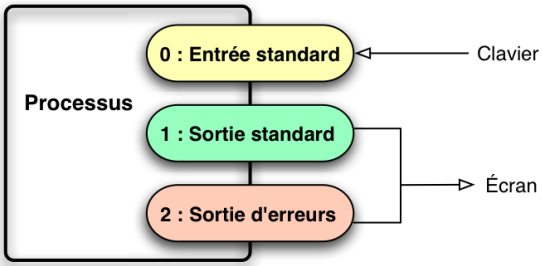
Eléments de syntaxe
≫Piping et redirection

Si les **sorties** et **entrée** de chaque programme sont des chaînes de caractères, il devient possible des '*chainer*'

```
programme_1 | programme_2
```

...

`STDOUT` de **programme_1** est transmis au `STDIN` de **programme_2**



[2.4] ⋙ Piping: Illustration

Dans cet exemple, nous récupérons les headers `-I`

Nous sélectionnons la ligne contenant `server`

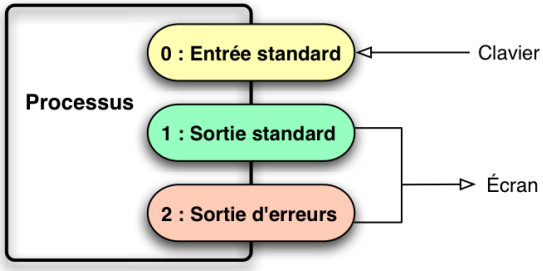
```
curl https://www.imie-paris.fr/ -I | grep server

  % Total    % Received % Xferd  Average Speed  Time    Time    Time   Curren
                                 Dload  Upload  Total   Spent   Left   Speed
  0      0   0      0   0      0      0      0                              0
server: Apache
```

Cas d'usage

Les `STDOUT` et `STDERR` d'un programme peuvent servir à autre chose qu'alimenter le `STDIN` d'un autre programme. Il sera parfois nécessaire

- d'agréer `STDOUT` et `STDERR` dans un même flux
- d'écrire ces flux dans un fichier
- de les envoyer à travers un réseau ou un socket



On se sert pour cela de redirections `>`, `>>`

Redirection des sorties dans un fichier

```
out='je suis une ligne'

echo $out > /tmp/out.log
echo $out > /tmp/out.log
# une seule ligne

echo $out >> /tmp/out.log
echo $out >> /tmp/out.log
# deux lignes
```

- Par défaut, `>` redirigera `STDOUT` dans un fichier en **écrasant son contenu**
- `>>` fait de même mais en **ajoutant au contenu existant**

...

Réduire au silence les erreurs

```
find /etc -writable
# output illisible

find /etc -writable 2>/dev/null
```

Ce pattern est très utile pour n'afficher que les résultats sans les alertes
Je veux identifier les fichiers sur lesquels peux écrire dans `/etc`
`STDERR` (2) est donc redirigé vers `/dev/null`

...

Aggrégation STDOUT STDERR

```
command 2>&1
```

Il est parfois nécessaire d'envoyer `STDOUT` et `STDERR` dans le même flux

Parsing de chaînes de caractères

Parsing de chaînes de caractères
≫Les outils historiques

[3.1] ⋙ Les REGEX seront vos meilleures amies

Puisque toutes les **entrées** et **sorties** de nos programmes sont des **chaînes de caractères**, pouvoir

- les filtrer
- les transformer
- en selectionner des portions

constitue une part importante du scripting `bash`.

Les **outils de base** permettent de réaliser ces opérations mais

- leurs syntaxes respectives peuvent être déroutantes (vieux outils)
- le comportement des REGEX diffère quelque peu de vos habitudes (match affiché par défaut, syntaxes *à la* `perl`)

Nous nous en sommes beaucoup servi pour **filtrer les lignes** d'une sortie

```
cat /etc/ssh/sshd_config
  | grep Subsystem

grep -ri "password" /etc

grep -riE "(P|p)assword\s?=" /etc
```

- mode simple dans un pipe: tout ce qui contient la chaîne
- recherche d'une chaine dans tous les fichiers trouvés **recursivement**
- **regex** Match: option `-E`

`grep` n'est, par défaut, **pas sensible à la casse**

Fonctionnement général

La commande de `sed` la plus utilisée est la suivante (mais beaucoup d'autres opérations sont possibles, *Cf* `man sed`)

```
sed 'f/<texte à trouver>/<texte à remplacer>/'
```

...

A la différence de `grep`, `sed` permet de n'afficher que les **groupes capturés** d'une expression régulière (option `-E`)

```
sed -nE 's/^NAME="(.*)"/\1/p' /etc/os-release
```

testeur et debugger en ligne

Utilisation commune: édition d'un fichier de configuration

Nous voulons désactiver la possibilité de se connecter comme `root` via `ssh`. Pour se faire

- trouvons les lignes commençant par **PermitRootLogin** (eventuellement précédées de `#`)
- et inscrivons *PermitRootLogin no* à la place

```
# état actuel de la conf
cat /etc/ssh/sshd_config | grep PermitRootLogin

sed -E 's/#?PermitRootLogin.*/PermitRootLogin no/' /etc/ssh/sshd_config
```

il suffirait alors d'utiliser l'option `-i` pour écrire dans le fichier

Dans le cas de données structurées, `awk` reste l'outil le plus commode

```
awk '{print $<Le numéro de la colonne que l on veut>}'
```

...

Pour peu que l'on spécifie le **séparateur** (`-F`), il devient facile de d'afficher le nom et le `HOME` de chaque utilisateur du système

```
awk -F: '{print "HOME de " $1 " : " $6}' /etc/passwd
```