

Introduction aux scripts BASH

1 La programmation shell

- Un script bash est fichier de type texte contenant une suite de commandes shell, exécutable par l'interpréteur (ici le programme `/bin/bash`), comme une commande unique. Un script peut être lancé en ligne de commande, comme dans un autre script.
- Mais il s'agit bien plus qu'un simple enchainement de commande : on peut définir des variables et utiliser des structures de contrôle, ce qui lui confère le statut de langage de programmation interprété et complet.
- Le langage **bash** gère notamment :
 - la gestion des entrées-sorties et de leur redirection
 - des variables définies par le programmeur et des variables systèmes
 - le passage de paramètres
 - des structures conditionnelles et itératives
 - des fonctions internes

1.1 Saisie du script

- Utiliser vi
- Les lignes commençant par le caractère dièse `#` sont des commentaires. En insérer abondamment !
- Le script doit débiter par l'indication de son interpréteur écrite sur la première ligne : `#!/bin/bash`.

En fait si le shell par défaut est bash, cette ligne est superflue

Exemple

```
#!/bin/bash

# script bonjour

# affiche un salut à l'utilisateur qui l'a lancé

# la variable d'environnement $USER contient le nom de login

echo ---- Bonjour $USER ----

# l'option -n empêche le passage à la ligne

# le ; sert de séparateur des commandes sur la ligne

echo -n "Nous sommes le " ; date

# recherche de $USER en début de ligne dans le fichier passwd

# puis extraction de l'uid au 3ème champ, et affichage

echo "Ton numéro d'utilisateur est " $(grep "^$USER" /etc/passwd | cut -d: -f3)
```

1.2 Exécution du script

- Il est indispensable que le fichier script ait la permission **x** (soit exécutable). Lui accorder cette permission pour tous ses utilisateurs avec `chmod` :

`chmod a+x bonjour`
- Pour lancer l'exécution du script, taper `./bonjour`, `./` indiquant le chemin, ici le répertoire courant. Ou bien indiquer le chemin absolu à partir de la racine. Ceci dans le cas où le répertoire contenant le script n'est pas listé dans le PATH
- Si les scripts personnels sont systématiquement stockés dans un répertoire précis, par exemple `/home/bin`, on peut ajouter ce chemin dans le PATH.

Pour cela, il suffit d'ajouter la ligne suivante dans `/etc/skel/.bash_profile`, qui est recopié dans chaque répertoire dont le rôle est d'affiner le profil personnel du shell de chaque utilisateur.

```
# bash_profile
```

```
.....
```

#user specific environment and statup programs

PATH=\$PATH:\$HOME/bin

- Mais on peut plus simplement s'initier au langage Bash, directement en dialoguant avec l'interpréteur.

Si on entre une instruction incomplète en ligne de commande, l'interpréteur passe à la ligne suivante en affichant le prompt **>** et attend la suite de l'instruction (pour quitter Ctrl-C).

1.3 Mise au point, débogage

Exécution en mode "trace" (-x) et en mode "verbeux" (-v) **sh -x ./bonjour**

Pour aider à la mise au point d'un script, on peut insérer des lignes temporaires : **echo \$var** pour afficher la valeur de la variable, **exit 1** pour forcer l'arrêt du script à cet endroit

1.4 Passage de paramètres

On peut passer des arguments à la suite du nom du script, séparés par des espaces. Les valeurs de ces paramètres sont récupérables dans le script grâce aux paramètres de position **\$1, \$2 ..** mais, contrairement aux langages de programmation classiques, ils ne peuvent pas être modifiés.

Exemple

```
#!/bin/bash

# appel du script : ./bonjour nom prenom

if [ $# = 2 ]

then

echo "Bonjour $2 $1 et bonne journée !"

else

echo "Syntaxe : $0 nom prenom"

fi
```

2 Entrées-sorties

Ce sont les voies de communication entre le programme bash et la console :

- **echo**, affiche son argument texte entre guillemets sur la sortie standard, c-à-d l'écran.

La validation d'une commande echo provoque un saut de ligne.

```
echo "Bonjour à tous !"
```

- On peut insérer les caractères spéciaux habituels, qui seront interprétés seulement si l'option **-e** suit

echo

\n (saut ligne), \b retour arrière), \t (tabulation), \a (alarme), \c (fin sans saut de ligne)

```
echo "Bonjour \nà tous !"
```

```
echo -e "Bonjour \nà tous !"
```

```
echo -e "Bonjour \nà toutes \net à tous ! \c"
```

- **read**, permet l'affectation directe par lecture de la valeur, saisie sur l'entrée standard au clavier
`read var1 var2 ...` attend la saisie au clavier d'une liste de valeurs pour les affecter, après la validation globale, respectivement aux variables `var1`, `var2` ..

exemple :

```
echo "Donnez votre prénom et votre nom"
```

```
read prenom nom
```

```
echo "Bonjour $prenom $nom"
```

3 Les variables BASH

3.1 Variables programmeur

De façon générale, elles sont de type texte. On distingue les variables définies par le programmeur et les variables systèmes

- syntaxe : `variable=valeur`

Attention ! le signe = NE DOIT PAS être entouré d'espace(s)

On peut initialiser une variable à une chaîne vide :

```
chaîne_vide=
```

- Si **valeur** est une chaîne avec des espaces ou des caractères spéciaux, l'entourer de " " ou de ' '
- Le caractère \ permet de masquer le sens d'un caractère spécial comme " ou '

```
chaîne="Bonjour à tous"
```

```
echo $chaîne
```

- Référence à la valeur d'une variable : faire précéder son nom du symbole \$
- Pour afficher toutes les variables : **set**
- Pour empêcher la modification d'une variable, invoquer la commande **readonly**
- **Substitution de variable**

Si une chaîne contient la référence à une variable, le shell doit d'abord remplacer cette référence par sa valeur avant d'interpréter la phrase globalement. Cela est effectué par l'utilisation de " ", dans ce cas obligatoire à la place de ' '.

Exemples

```
n=123 ;  
  
echo "la variable \$n vaut $n"  
  
salut="bonjour à tous !"  
  
echo "Alors moi je dis : $salut"  
  
echo 'Alors moi je dis : $salut'  
  
echo "Alors moi je dis : \"$salut\" "  
  
readonly salut  
  
salut="bonjour à tous, sauf à toto"  
  
echo "Alors moi je dis : $salut"
```

- **Variables exportées**

Toute variable est définie dans un shell. Pour qu'elle devienne globale elle doit être exportée par la commande :

export variable

export --> Pour obtenir la liste des variables exportées

- **Opérateur {} dans les variables**

Dans certains cas en programmation, on peut être amené à utiliser des noms de variables dans d'autres variables. Comme il n'y a pas de substitution automatique, la présence de {} force l'interprétation des variables incluses.

exemple :

```
user="/home/stage"  
  
echo $user  
  
u1=$user1  
  
echo $u1  --> ce n'est pas le résultat escompté !  
  
u1=${user}1  
  
echo $u1
```

3.2 Variables d'environnement

Ce sont les variables systèmes dont la liste est consultable par la commande **env** | **less**

Les plus utiles sont \$HOME, \$PATH, \$USER, \$PS1, \$SHELL, \$ENV, \$PWD ..

Exemple *(bien sûr on est pas forcément connecté sous le pseudo toto)*

```
[toto@pxx toto]$ moi=Toto
```

```
[toto@pxx toto]$ p="Je m'appelle $moi"
```

```
[toto@pxx toto]$ echo Aujourd'hui, quel jour sommes nous ? ; read jour
```

```
[toto@pxx toto]$ echo aujourd'hui $jour, $p sous le nom $USER, est connecté à la station $HOSTNAME
```

3.3 Variables prédéfinies spéciales

Elles sont gérées par le système et s'avèrent très utiles dans les scripts. Bien entendu, elles ne sont accessibles qu'en lecture.

Ces variables sont automatiquement affectées lors d'un appel de script suivi d'une liste de paramètres.

Leurs valeurs sont récupérables dans \$1, \$2 ...\$9

\$?	C'est la valeur de sortie de la dernière commande. Elle vaut 0 si la commande s'est déroulée sans pb.
\$0	Cette variable contient le nom du script
\$1 à \$9	Les (éventuels) premiers arguments passés à l'appel du script
\$#	Le nombre d'arguments passés au script
\$*	La liste des arguments à partir de \$1
\$\$	le n° PID du processus courant
\$!	le n° PID du processus fils

3.4 Passage de paramètres

On peut récupérer facilement les compléments de commande passés sous forme d'arguments sur la ligne de commande, à la suite du nom du script, et les utiliser pour effectuer des traitements.

Ce sont les variables système spéciales **\$1**, **\$2** **\$9** appelées paramètres de position.

Celles-ci prennent au moment de l'appel du script, les valeurs des chaînes passées à la suite du nom du script (le séparateur de mot est l'espace, donc utiliser si nécessaire des "").

A noter que :

- le nombre d'argument est connu avec **\$#**
- la liste complète des valeurs des paramètres (au delà des 9 premières) s'obtient avec **\$***
- le nom du script rest recopié dans **\$0** (utile pour la programmation récursive)

3.5 La commande shift

Shift est une commande interne qui renomme les paramètres positionnels \$2, \$3, ... en \$1, \$2, ... et perd \$1.

Il n'y a que 9 paramètres de position de \$1 à \$9, et s'il y a davantage de paramètres transmis, comment les récupérer ?

shift effectue un décalage de pas +1 dans les variables \$: \$1 prend la valeur de \$2, etc...

Exemple

```
$ cat shift1

while [ $# -ne 0 ]

do

    echo « il y a $# arguments et » '$1' = $1

    shift

done
```


\$ shift1 a b c d e

il y a 5 arguments et \$1 = a

il y a 4 arguments et \$1 = b

il y a 3 arguments et \$1 = c

il y a 2 arguments et \$1 = d

il y a 1 arguments et \$1 = e

3.6 La commande set

Exemple

a=1 ; b=2 ; c=3

set a b c

echo \$1, \$2, \$3

les valeurs de a, b, c sont récupérées dans \$1, \$2, \$3

4 La commande test

4.1 Généralités

Comme son nom l'indique, elle sert à vérifier des conditions. Ces conditions portent sur des fichiers (le plus souvent), ou des chaînes ou une expression numérique.

Cette commande courante sert donc à prendre des (bonnes) décisions, d'où son utilisation comme condition dans les structures conditionnelles `if.. then ..else`, en quelque sorte à la place de variables booléennes ... qui n'existent pas.

4.2 Syntaxe

- test expression
- [expression] attention aux espaces autour de expression

4.3 Valeur de retour

- Rappels
On sait que toute commande retourne une valeur finale au shell : 0 pour lui indiquer si elle s'est déroulée normalement ou un autre nombre si une erreur s'est produite.
Cette valeur numérique est stockée dans la variable spéciale `$?`
- La commande `test`, de même , retourne 0 si la condition est considérée comme vraie, une valeur différente de 0 sinon pour signifier qu'elle est fausse.

4.4 Tester un fichier

- Elle admet 2 syntaxes (la seconde est la plus utilisée) :

test option fichier

[option fichier]

- Tableau des principales options

option	signification quant au fichier
-e	il existe
-f	c'est un fichier normal
-d	c'est un répertoire
-r -w -x	il est lisible modifiable exécutable
-s	il n'est pas vide

Exemples

[-s \$1]

vrai (renvoie 0) si le fichier passé en argument n'est pas vide

[\$# = 0] le nombre d'arguments

est 0

[-w fichier] le fichier est-il modifiable ?

[toto@p00]\$ [-r "/etc/passwd"] *toto peut-il lire le fichier /etc/passwd ?*

[toto@p00]\$ echo \$? --> 0 (vrai)

[toto@p00]\$ [-r "/etc/shadow"] *toto peut-il lire le fichier /etc/shadow ?*

[toto@p00]\$ echo \$? --> 1 (faux)

[toto@p00]\$ [-r "/etc/shadow"] || echo "lecture du fichier interdite"

4.5 Tester une chaîne

- [option chaîne]

option	signification
-z -n	la chaîne est vide / n'est pas vide
= !=	les chaînes comparées sont identiques différentes

Exemples

[-n "toto"] ; echo \$? *affiche la valeur renvoyée 0*

ch="Bonjour" ; ["\$ch" = "bonjour"] ; echo \$? *affiche 1*

[\$USER != "root"] && echo "l'utilisateur n'est pas le \"root\" !"

4.6 Tester un nombre

- [nb1 option nb2]

Il y a d'abord un transtypage automatique de la chaîne de caractères en nombre

option	signification
-eq -ne	égal différent
-lt -gt	strict. inf strict. sup
-le -ge	inf ou égal sup ou égal

Exemples

a=15 ; ["\$a" -lt 15] ; echo \$?

4.7 Opérations dans une commande test

option	valeur
[expr1 -a expr2]	(and) 0 si les 2 expr sont vraies
[expr1 -o expr2]	(or) 0 si l'une des 2 expr est vraie
[! expr1]	négation

Exemples

Quel résultat ? envisager 2 cas ...

```
f="/home/stage" ; [ -d "$f" -a -x "$f" ] ; echo $?
```

```
note=9; [ $note -lt 8 -o $note -ge 10 ] && echo "tu n'es pas convoqué(e) à l'oral"
```

5 Structures conditionnelles

5.1 Conditionnelle simple : if then else

```
if suite-de-commandes
```

```
then
```

```
# séquence exécutée si suite-de-commandes rend une valeur 0
```

```
bloc-instruction1
```

```
else
```

```
# séquence exécutée sinon
```

```
bloc-instruction2
```

```
fi
```

Attention ! si then est placé sur la 1ère ligne, séparer avec un ;

```
if commande; then
```

```
.....
```

Exemples

1. toto possède t-il un compte ? On teste la présence d'une ligne commençant par toto dans
 /etc/passwd (>/dev/null pour détourner l'affichage de la ligne trouvée)
2. if grep "^toto" /etc/passwd > /dev/null
3. then
4. echo "Toto a déjà un compte"
5. fi
6. Si toto a eu une bonne note, on le félicite
7. note=17
8. if [\$note -gt 16] ---> *test vrai, valeur retournée : 0*
9. then echo "Très bien !"
10. fi
11. Avant d'exécuter un script, tester son existence.

Extrait de \$HOME/.bash_profile

12. if [-f ~/.bashrc]
13. then
14. ~/.bashrc
15. fi

5.2 Conditionnelles imbriquées

Pour imbriquer plusieurs conditions, on utilise la construction :

```
if commande1
then
    bloc-instruction1
elif commande2
then
    bloc-instruction2
else
    # si toutes les conditions précédentes sont fausses
    bloc-instruction3
fi
```

Exemples

1. toto a t-il fait son devoir lisiblement ?
2. fichier=/home/toto/devoir1.html
3. if [-f \$fichier -a -r \$fichier]
4. then
5. echo "je vais vérifier ton devoir."
6. elif [! -e \$fichier]
7. then
8. echo "ton devoir n'existe pas !"
9. else
10. echo "je ne peux pas le lire !"
11. fi
12. Supposons que le script exige la présence d'au moins un paramètre, il faut tester la valeur de \$#, est-elle nulle ?
13. if [\$# = 0]
14. then
15. echo "Erreur, la commande exige au moins un argument .."

```

16. exit 1

17. elif [ $# = 1 ]

18. then

19.  echo "Donner le second argument : "

20.  read arg2

21. fi

```

5.3 Choix multiples

```

case valeur in

    expr1) commandes ;;

    expr2) commandes ;;

    ...

esac

```

Exemples

1. Supposons que le script doive réagir différemment selon l'utilisateur courant; on va faire plusieurs cas selon la valeur de **\$USER**
2. case \$USER in
3. root) echo "Mes respects M le \$USER" ;;
4. jean | stage?) echo "Salut à \$USER" ;;
5. toto) echo "Fais pas le zigo\$USER !" ;;
6. esac
7. Le script attend une réponse *oui/non* de l'utilisateur
8. read reponse
9. case \$reponse in
10. [yYoO]*) ;;
11. [nN]*);;
- esac


```
12. read langue
13. case $langue in
14. francais) echo Bonjour ;;
15. anglais) echo Hello ;;
16. espagnol) echo Buenos Dias ;;
    esac
17. case $param in
18. 0|1|2|3|4|5|6|7|8|9 ) echo $param est un chiffre ;;
19. [0-9]*) echo $param est un nombre ;;
20. [a-zA-Z]*) echo $param est un nom ;;
21. *) echo $param de type non prévu ;;
22. esac
23.
24. Un vrai exemple, extrait du script smb (/etc/rc.d/init.d/smb)
25. # smb attend un paramètre, récupéré dans la variable $1
26. case "$1" in
27.  start)
28.      echo -n "Starting SMB services: "
29.      daemon smbd -D
30.      echo
31.      echo -n "Starting NMB services: "
32.      daemon nmbd -D
33.      ...;;
34.  stop)
35.      echo -n "Shutting SMB services: "
36.      killproc smbd
37.      ....
38.  esac
```

6 Structures itératives

6.1 Boucle for

- Syntaxe

```
for variable [in liste]
do
commandes (utilisant $variable)
done
```

- Fonctionnement

Ce n'est pas une boucle **for** contrôlée habituelle fonctionnant comme dans les langages de programmation classiques (utiliser pour cela une boucle **while** avec une variable numérique).

La *variable* parcourt un ensemble de fichiers donnés par une liste ou bien implicitement et le bloc *commandes* est exécuté pour chaque de ses valeurs.

Les mots-clés **do** et **done** apparaissent en début de ligne (ou après un ;)

- La liste peut être explicite :

```
for nom in jean toto stage1
do
echo "$nom, à bientôt"
done
```

- La liste peut être calculée à partir d'une expression régulière

```
# recopier les fichiers perso. de toto dans /tmp/toto
for fich in /home/toto/*
do
cp $fich tmp/toto
```

done

- Si aucune liste n'est précisée, les valeurs sont prises dans la variable système `$@`, c'est-à-dire en parcourant la liste des paramètres positionnels courants.

```
# pour construire une liste de fichiers dans $@
```

```
cd /home/stagex ; set * ; echo $@
```

```
for nom in $@
```

```
do echo $nom
```

```
done
```

- **Expliquer les 4 exemples suivants**

```
for nom in /home/stage[1-9]
```

```
do
```

```
echo "$nom, tout va bien ?"
```

```
done
```

```
for i in /home/*/* ; do echo $i; done
```

```
for x in /home/st*
```

```
do
```

```
echo $x >> liste-rep-stage.txt
```

```
done
```

```
less liste-rep-stage.txt
```

```
for x in $(grep "^st" /etc/passwd | cut -d: -f6)
```

```
do
```

```
echo $x; echo $x >> $HOME/tmp/liste-rep-stage.txt
```

```
done
```

6.2 Boucle while

while liste-commandes do commandes done	La répétition se poursuit TANT QUE la dernière commande de la liste est vraie (c-à-dire renvoie un <u>code de retour nul</u>)	Voici 2 exemples à comparer echo -e "Entrez un nom de fichier" read fich while [-z "\$fich"] do echo -e "Saisie à recommencer" read fich done while echo -e "Entrez un nom de fichier" read fich [-z "\$fich"] do echo -e "Saisie à recommencer" done
until liste-commandes do commandes done	La répétition se poursuit JUSQU'A CE QUE la dernière commande de la liste devienne vraie	

Exemples à tester

Pour dire bonjour toutes les secondes (arrêt par CTRL-C)

```
while true ;
```

```
do
```

```
echo "Bonjour M. $USER"
```

```
sleep 1
```

```
done
```

Lecture des lignes d'un fichier pour traitement : noter que la redirection de l'entrée de la commande
while .. do .. done est placée à la fin

```
fich=/etc/passwd
```

```
while read ligne
```

```
do
```

```
echo $ligne
```

```
.....
```

```
done < $fich
```

6.3 Sortie et reprise de boucle

break placé dans le corps d'une boucle, provoque une sortie définitive cette boucle.

continue permet de sauter les instructions du corps de la boucle (qui suivent continue) et de "continuer" à l'itération suivante. Pour les boucles **for**, **while** et **until**, **continue** provoque donc la réévaluation immédiate du test de la boucle.

Exemples importants

Boucle de lecture au clavier arrêtée par la saisie de *stop*

```
#!/bin/bash

# syntaxe : lecture.sh

texte=""

while true

do

    read ligne

    if [ $ligne = stop ]

    then break

    else texte="$texte \n$ligne"

    fi

done

echo -e $texte
```

Lecture des lignes d'un fichier

```
fich="/etc/passwd"

grep "^stage" $fich | while true

do

    read ligne

    if [ "$ligne" = "" ] ; then break ; fi

    echo $ligne
```

7 Fonctions

- 2 syntaxes

function nom-fct {

bloc d'instructions

}

nom-fct() {

bloc d'instructions

}

- Exemple

En connexion root, on doit relancer les "démons", si on a modifié un fichier de configuration.

Par exemple /etc/rc.d/init.d/smb contient la commande `daemon smbd -D`, pourtant à l'essai `daemon` est une commande inconnue !

Reportons nous au début du fichier, le script /etc/rc.d/init.d/functions y est appelé. Celui-ci contient la fonction : `daemon() {`

- passage d'arguments

Le mécanisme est le même que vis à vis d'un script : les paramètres \$1, \$2, ... \$9 sont récupérées dans le corps de la fonction

- variables locales

Dans le corps de la fonction, on peut définir et utiliser des variables déclarées locales, en les introduisant avec le mot-clé **local**

8 Commandes diverses

8.1 Calcul sur les entiers relatifs

Ne pas confondre la syntaxe `$((expression arithmétique))` avec la substitution de commande `$(commande)`

Les priorités sont gérées par un parenthésage habituel

```
echo $((30+2*10/4))
```

```
echo $(( (30+2) * (10-7) /4 ))
```

8.2 tr

- Cette commande de filtre permet d'effectuer des remplacements de caractères dans une chaîne.
Pour une étude plus complète voir le **chapitre sur les filtres**.
- Par exemple pour transformer une chaîne en minuscules

```
chaîne="Bonjour, comment allez VOUS aujourd'hui ?"
```

```
echo $chaîne | tr 'A-Z' 'a-z'
```

- Pour permettre l'utilisation de la commande **set** (voir ci-dessous), il est nécessaire que le **séparateur de champ sur une ligne soit l'espace, et non pas par exemple :**

Exemple : créer un fichier passwd.txt qui introduit un espace à la place de ":" dans une **copie** de

```
/etc/passwd
```

```
cat passwd | tr ":" " " > passwd.txt
```


8.3 set

Cette **commande interne** est très pratique pour séparer une ligne en une liste de mots, chacun de ces mots étant affecté à une variable positionnelle. Le caractère de séparation est l'espace.

soit une chaine ch qui contient une liste de mots

```
c="prof eleve classe note"
```

set va lire chaque mot de la liste et l'affecter aux paramètres de position

```
set $c ; echo $1 $2 $3 $4
```

```
shift ; echo $1 $2 $3 $4
```

Le langage bash est inadapté aux calculs numériques.

Mais si vraiment on veut calculer (sur des entiers) ..

Exemple :

calcul des premières factorielles (attention, il y a rapidement un dépassement de capacité)

```
k=1 ; p=1
```

```
while [ $k -le 10 ]
```

```
do
```

```
echo "$k! = " $((p=$p * $k)) ; k= $k+1
```

```
done
```

8.4 eval

Cette commande ordonne l'interprétation par le shell de la chaîne passée en argument. On peut ainsi construire une chaîne que l'appel à **eval** permettra d'exécuter comme une commande !

Exemple 1

```
message="Quelle est la date d'aujourd'hui ?
```

```
set $message
```

```
echo $# ---> le nombre de mots est 6
```

```
echo $4 ---> affiche la chaîne "date"
```

```
eval $4 ---> interprète la chaîne "date" comme une commande, donc ...
```

Il est souvent pratique de construire une chaîne dont la valeur sera égale au libellé d'un enchaînement de commandes (par ;). Pour faire exécuter ces commandes contenues dans la chaîne, on la passe comme argument de la commande **eval**

exemple 2

double quote " " obligatoires sinon le ; est un séparateur de commandes)

```
liste="date;who;pwd"
```

```
eval $liste
```

```
---> exécute bien les 3 commandes
```

8.5 Application aux scripts cgi

La soumission d'un formulaire HTML à la passerelle CGI est un mécanisme qui aboutit à la récupération par le script (Bash, Perl ,etc..) d'une chaîne qui contient la requête sous un format particulier.

Voici un exemple

- Le professeur Toto a rempli un formulaire sur le WEB pour recevoir un spécimen, soient 2 champs de texte nommés en HTML **nom** et **prenom**, et a coché la case nommée **prof**

Voici la chaîne supposée nommée *requete* qui a été transmise :

```
nom=toto&prenom=jules&prof=on
```

- Cette chaîne contient toute l'information relative au formulaire saisi par l'utilisateur. Il s'agit toujours de la traiter pour en récupérer les couples (var, valeur) où var sont les noms donnés aux composants de formulaire et valeur les chaînes saisies ou exprimant une sélection.
- En Bash le découpage de \$chaîne, puis les affectations des variables doit être fait "à la main"

```
requete="nom=toto&prenom=jules&prof=on"
```

*# le filtre **tr** va remplacer dans la chaîne \$requete qu'il reçoit, tous les caractères & par ;*

```
commande=$( echo $requete | tr '&' ';')
```

```
echo $commande ---> nom=toto;prenom=jules;prof=on
```

```
eval $commande ---> exécute le ligne de commande, donc effectue les affectations !
```

```
echo $prenom $nom
```

```
[ $prof = "on" ] && echo "$prenom $nom est professeur"
```