Maîtriser les conditions en bash

Le scripting bash est souvent déroutant. La syntaxe est parfois assez éloignée de ce à quoi nous sommes habitués dans d'autres langages. C'est d'ailleurs pourquoi beaucoup l'évitent au maximum. Pourtant, tôt ou tard, on a tous besoin d'écrire un petit *.sh* pour gérer un service ou automatiser un comportement sur un serveur.

Sous Linux et Unix, il existe plusieurs interpréteurs de commandes ou shell. Les fonctions supportées par l'un ou l'autre peuvent varier. Ainsi, nous parlons ici de bash, alias *bourne again shell*, le successeur de sh, le *shell* historique. Ce dernier est 100% compatible avec sh, en revanche, la réciproque n'est pas vraie. Nous verrons justement cela dans les conditions.

Anatomie d'un `if`

Prenons les choses dans l'ordre. En bash, pas d'indentation ou d'accolades, un `if` se démarque par des mots clefs de début et de fin.

Le `if` teste uniquement le code de retour de la commande (`0` étant le code de succès). Il est néanmoins souvent accompagné d'une commande de test.

Cette commande est `test`, ou `[`. **Notez bien que contrairement aux langages de la famille C, les crochets `[]` utilisés pour les tests sont bien une commande et non une structure de langage**.

```
if [ -f config.php ]
then
    echo 'action if config.php exists'
else
    echo 'action if config.php does not exist'
fi
```

```
# Équivalent à
```

```
test -f config.php
then
    echo 'action if config.php exists'
else
    echo 'action if config.php does not exist'
fi
```

```
# un si simple

if [ test ]
then
    echo "le teste est passé"
fi # on ferme toujours la condition par fi

# et avec un else if et un else
if [ test ]
then
    echo "le teste est passé"
elif [ test2 ]
then
    echo "le test2 est passé"
else
    echo "les deux tests ont échoué"
fi
```

Évidemment, vous pouvez tout à fait omettre le `elif`, le `else` ou les deux. Vous pouvez également mettre plusieurs `elif` les uns à la suite des autres.

Vous noterez bien que la condition est **toujours entourée d'un espace** après le crochet d'ouverture et avant le crochet de fermeture.

Dans les exemples ci-dessus, chaque mot clef est sur sa propre, ligne. Il est néanmoins possible de **placer le then sur la même ligne que la condition en utilisant un point-virgule**.

```
if [ test ]; then
    echo "le teste est passé"
fi
```

Par ailleurs, il est évidemment tout à fait possible d'inverser la condition avec le classique opérateur `!`.

```
if [ ! test ]; then
    echo "le teste n'est pas passé"
fi
```

Les syntaxes conditionnelles

Nous l'avons dit en introduction, bash est une évolution de sh. À ce titre, il comprend les conditions classiques de sh, mais il intègre aussi une syntaxe plus avancée, laquelle n'est pas rétro-compatible.

Conditions à simples crochets

Ce sont les conditions compatibles avec sh. Elles fonctionnent très bien dans la majorité des cas. On dénombre trois grandes familles de conditions : les conditions basées sur des **fichiers ou dossier**, celles basées sur des **chaînes de caractères** et enfin celles concernant des **valeurs arithmétiques**.

Conditions sur les fichiers

Permet de vérifier si un fichier ou un répertoire existe, sa nature etc. Nous verrons les différentes valeurs de conditions plus loin.

```
if [ -f ceci_est_un_fichier ]
then
    echo "il s'agit bien d'un fichier"
fi
```

Conditions sur les strings

Comme dans tous programme qui se respecte, on a toujours à traiter des chaînes de caractères.

```
if [ -z "$variable_vide" ]
then
    echo "la variable est bien vide"
fi

# comparaison

if [ "$variable" = "ta mère" ]
then
    echo "la variable contient la chaîne ta mère"
fi
```

Vous noterez qu'il n'y a qu'un signe égal pour la compatibilité POSIX. Le double, "==" fonctionne aussi. Mais il est suggéré d'utiliser le simple signe. Cela évitera des confusions.

Vous remarquez peut-être que la variable est entourée de guillemets. Bien que ce ne soit pas une obligation, ça vous évitera des bugs si elle contient des espaces ou des retours à la ligne par exemple. Par ailleurs, dans le cas de l'utilisation de `test`, les guillemets sont obligatoires.

Conditions arithmétiques

On va ici comparer des entiers. Supérieur, inférieur, égal…

```
if [ $num -lt 1 ]
then
    echo "la variable est inférieure à 1"
fi

if [ $num -eq 100 ]
then
    echo "la variable est égale à 100"
fi
```

OR et AND

Il est possible d'utiliser le ET et le OU logique avec les simples crochets avec respectivement les arguments `-o` et `-a`.

```
if [ $num -lt 1 -o $num -eq 100 ]
then
    echo "la variable est inférieure à 1 ou égale à 100"
fi

if [ $num -lt 1 -a $num -eq 100 ]
then
    echo "Ceci ne sera jamais appelé"
elif
then
    echo "Mais ça oui !"
fi
```

Conditions à doubles crochets

Ces conditions permettent tout ce qu'offrent les conditions à simples crochets et plus. **En revanche, elles ne sont pas compatibles avec sh**. Elles prennent la forme suivante.

```
if [[ -z "$variable_vide" ]]
then
    echo "la variable est bien vide"
fi
```

Ces conditions améliorées proposent l'usage du wildcard comme en bash ainsi que des expressions régulières. Ainsi, il est possible d'avoir des conditions comme cela :

```
if [[ "$variable" ==  *superman* ]]
then
    echo "la variable contient le mot superman"
fi


if [[ "$variable" ==  superman* ]]
then
    echo "la variable commence par le mot superman"
fi


if [[ "$variable" ==  [sS]uperman* ]]
then
    echo "la variable commence par le mot superman ou Superman"
fi


if [[ "$email" =~ "b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+.[A-Za-z]{2,4}b" ]]
then
    echo "la variable contient un email valide"
fi
```

On a donc accès aux expressions régulières, ce qui est assez confortable. En revanche, le caractère "*" n'est pas interprété dans les conditions sur les fichiers.

```
if [[ -a *.sh ]]
then
    echo "le fichier *.sh existe"
fi

# la syntaxe classique recherche un fichier en .sh, n'importe lequel
if [ -a *.sh ]
then
    echo "un fichier en *.sh existe"
fi
```

Avec la syntaxe classique, la condition vaut `true` s'il y a un fichier en *.sh*, `false` s'il n'y en a pas, mais une erreur est retournée s'il y en a plusieurs…

La syntaxe améliorée permet aussi l'usage des opérateurs classiques de conditions tels que `||` et `&&`. Par ailleurs, il est possible d'omettre les guillemets autour des variables car les espaces ne posent plus problème.

La syntaxe double parenthèses

Cette syntaxe dédiée aux comparaisons arithmétiques peut sembler plus familière car elle autorise les opérateurs plus connus dans d'autres langages : `==`, `!=`, `<` et `>`. Elle supporte aussi les opérateurs `&&` et `||`.

```
if (( $a != 0 ))
then
    echo "$a est non nul"
fi

# on peut également utiliser les ternaires
(( $a != 0 )) && echo "$a est non nul" || echo "$a est null"
```

La syntaxe simplifiée

Pour les *one-liners*, on peut tout mettre sur une seule ligne.

```
if [ ! -f config.php ]; then echo 'action if config is not created'; else echo
'action if config exists already'; fi
```

On peut aussi utiliser les ternaires pour plus de concision.

```
# Avec test
test ! -f config.php && echo 'action if config is not created' || echo 'action if
config exists already'

# Ou juste sans le if
[ ! -f config.php ] && echo 'action if config is not created' || echo 'action if
config exists already'

# Si l'on veut juste exécuter une commande en cas de succès
[ ! -f config.php ] && echo 'action if config is not created' || :
```

Au cas où vous vous posiez la question, l'opérateur `:` est un `builtin` du shell (sh et tous ses successeur). Il signifie que rien ne sera fait et retourne 0 (code de succès). Sans cette dernière partie,

en cas d'échec de la condition, votre commande retournerait un code d'erreur, ce qui a de grande chance de faire planter votre script.

Par ailleurs, notez bien que dans le cas où vous utilisez la syntaxe simplifiée avec le `||`, la partie juste à droite – soit `echo "ce fichier contient du php"` – doit retourner toujours vrai. Sans quoi le "ou" s'exécuterait aussi.

Au-delà du `test` et de la commande `[ ]`, il est possible d'appliquer une condition sur le code de retour d'une commande.

```
if grep -q "php" /var/www/index.php; then echo "ce fichier contient du php"; fi

# sans le if
grep -q "php" /var/www/index.php && echo "ce fichier contient du php"

# il est aussi possible d'avoir un où dans ce cas
grep -q "php" /var/www/index.php && echo "ce fichier contient du php" || echo "ce
fichier ne contient pas de php"
```

Table des conditions

Avant de vous laisser avec une petite liste de conditions, **gardez toujours à l'esprit que les conditions possèdent toujours un espace entre les différents arguments. Ainsi, en bash `1 = 2` vaut `false`, tandis que `1=2` vaut `true`. Le = sans espace est une assignation et non une comparaison** !

Panorama des conditions les plus utiles.

Conditions sur les fichiers

Condition	Vrai si	Détails
`[ -a fichier ]`	"fichier" existe et est un fichier.	–
`[ -b fichier ]`	Le fichier "fichier" existe et est de type block special.	Les block special files sont des fichiers qui représentent des périphériques, souvent des disques dur ou clefs USB. On les trouve principalement dans /dev ex. /dev/sda (disque a) et /dev/sda1 (partition 1 du disque a)
`[ -c fichier ]`	Le fichier "fichier" existe et est de type caractère spécial.	Character special files sont des fichiers spéciaux du kernel, différents des block special files, ils ne sont pas mis en mémoire tampon, buffered, leur effet est immédiat : envoie du caractère au driver correspondant (émission d'un son via la carte son etc). Ce type de fichier concerne aussi les pseudo-devices /dev/null, /dev/zero, /dev/full, /dev/random et /dev/urandom. N'hésitez pas à jeter un œil à ce post Stackexchange pour plus de détails sur ces fichiers spéciaux
`[ -d dossier ]`	dossier existe et est un dossier	–

`[ -e fichier ]`	Même comportement que -a.	–
`[ -f fichier_normal ]`	fichier_normal existe et est un fichier “normal”.	Par normal, on entend que le fichier n’est ni de type block, ni character special.
`[ -h lien_symbolique ]`	lien_symbolique existe et est un lien symbolique	–
`[ -L lien_symbolique ]`	Même comportement que -h.	
`[ -r fichier_lisible ]`	fichier_lisible existe et est lisible du script	–
`[ -s fichier_non_vide ]`	fichier_non_vide existe et possède une taille non nulle	–
`[ -w fichier_writable ]`	fichier_writable existe et le script peut y écrire.	–
`[ -x executable ]`	executable existe et est exécutable depuis le script	Concernant un répertoire, le droit d’exécution permet simplement de lister son contenu.

Conditions sur les chaînes

Nous avons vu le plus complexe. En effet, Linux possède de nombreux types de fichiers et nous avons donc des conditions qui n’ont pas lieu d’exister dans tous les langages.

Les conditions sur les chaînes de caractères sont plus classiques. Les `==` et `!=` se passent d’explications. `<` et `>` permettent de déterminer si un string est avant un autre dans un classement par ordre alphabétique… oui ça peut servir, *who knows*.

Enfin, deux conditions un peu particulière permettent de savoir si un string est vide ou non : `[ -n string_non_vide ]` et `[ -z string_vide ]`.

Conditions sur les entiers

Nous l’avons vu, avec la syntaxe à double parenthèses, rien de sorcier, c’est comme nous en avons l’habitude dans tous les autres langages. En revanche, pour la syntaxe classique, avec les crochets, c’est un peu moins intuitif. Si vous connaissez MongoDB, c’est comme les conditions en Mongo.

- `[ nombre_1 -eq nombre_2 ]` pour *equal*, égal.
- `[ nombre_1 -ne nombre_2 ]` pour *not equal*, non égal.
- `[ nombre_1 -gt nombre_2 ]` pour *greater than*, plus grand que.
- `[ nombre_1 -ge nombre_2 ]` pour *greater than or equal*, plus grand ou égal.
- `[ nombre_1 -lt nombre_2 ]` pour *less than*, plus petit.
- `[ nombre_1 -le nombre_2 ]` pour *less than or equal*.