Chapitre 4
Les bases de l'écriture d'un script

1. Forme générale d'un script

Un script est ni plus ni moins qu'un enchaînement de commandes, les unes après les autres, sous la forme d'un fichier texte. Cependant, certaines règles sont à respecter pour qu'un shell puisse comprendre ce que l'on attend de lui.

1.1 Nommage de fichier

Un script shell porte généralement l'extension `.sh` ; cependant, cela n'est pas une obligation : certains scripts portent d'autres extensions, d'autres scripts n'en portent pas du tout.

Il est préférable, de manière générale, de réserver l'extension `.sh` aux scripts compatibles POSIX. De cette manière, on sait que ce script peut être exécuté avec n'importe quel shell qui respecte cette norme. Par conséquent, lorsque l'on développe un script qui utilise des spécificités d'un shell particulier, il vaut mieux utiliser une extension qui indique clairement le script à utiliser ; les shells portant des noms plutôt courts, on utilise simplement leur nom pour l'extension : `.bash` pour bash, `.zsh` pour Zsh...

La plupart des systèmes d'exploitation autorisent les espaces dans les noms de fichiers. Cela est également valable pour les noms des scripts ; de même pour les caractères spéciaux, accentués et autres. Cependant, un script sera amené à être appelé en ligne de commande, son nom tapé à la main. Pour cette raison, il est préférable de garder un nom simple, court, sans caractère spécial, sans accent, comme pour n'importe quel nom de commande car même si un nom peut être tapé sur votre clavier, cela ne veut pas dire qu'il peut l'être sur n'importe quel clavier : pensez simplement aux claviers QWERTY qui ne proposent pas les caractères accentués !

On préférera donc le nom « `update_web_database.sh` » à « `Mettre à jour la base de données des serveurs web.bash` ».

1.2 Mise en forme

Par mesure de cohérence (et parce que c'est plus logique, tout simplement), on place généralement chaque appel à une commande seul sur une ligne ; quand on exécute des commandes à la main, c'est exactement la même chose, une commande par ligne.

Par ailleurs, afin de rendre le script lisible, on peut faire précéder les commandes par autant de caractères d'espacement que l'on veut (cela inclut les tabulations) ; cela sera particulièrement utile dans le cadre de fonctions, de structures conditionnelles ou de boucles. De la même manière, en ligne de commande, les shells interprètent correctement n'importe quelle commande qui est précédée par des espaces, en ignorant simplement ces derniers.

En réalité, on peut aller jusqu'à copier un script et le coller directement dans un terminal : les commandes seront lancées les unes après les autres, de la même manière que si l'on avait exécuté le script lui-même. Conformément à ce qui a été évoqué, un script n'est rien d'autre qu'une transcription de ce que l'on taperait si l'on exécutait manuellement les commandes qu'il appelle.

1.3 Commentaires

Comme avec tout langage de programmation, un shell script peut contenir des commentaires. En l'occurrence, c'est le croisillon (caractère « # ») qui marque le début d'un commentaire ; celui-ci peut se trouver en début de ligne (auquel cas la ligne entière est un commentaire) ou en milieu de ligne (auquel cas le commentaire est précédé d'une commande qui sera exécutée).

Ce langage ne propose par contre pas de méthode pour commenter des blocs entiers (comme ce que l'on peut faire avec « `/* ... */` » en C par exemple) : lorsque l'on souhaite commenter un bloc entier, il faut commenter chacune de ses lignes (cette action est aisée avec la sélection visuelle de Vi ; la plupart des environnements de développement proposent également une fonction de mise en commentaire de bloc).

Les commentaires sont avant tout utilisés pour expliquer le fonctionnement du programme (du script), lorsque celui-ci n'est pas évident. C'est essentiel lorsque l'on souhaite que le programme puisse être repris et compris ultérieurement, que ce soit par soi-même ou par quelqu'un d'autre. On utilise également les commentaires pour désactiver temporairement certaines instructions, lorsque l'on est en train de développer un script par exemple.

Évidemment, un script étant simplement un enchaînement de commandes, ce caractère de commentaire est parfaitement accepté lorsqu'on le tape en ligne de commande. On peut notamment l'utiliser après avoir tapé une ligne de commande que l'on ne doit finalement pas exécuter, mais que l'on souhaite conserver dans l'historique pour la réutiliser plus tard : il suffit alors de retourner en début de ligne pour la commenter entièrement.

1.4 Le shebang

La méthode basique pour lancer un script est de le donner comme argument au shell que l'on souhaite. Pour lancer le script `update_web_database.sh` avec Bash, on exécuterait donc la commande suivante :

```
bash update_web_database.sh
```

Cependant, cette approche mérite d'être simplifiée : et si l'ordinateur pouvait lui-même identifier quel shell utiliser, sans le spécifier explicitement ? C'est le rôle de ce qu'on appelle *shebang*. Ce nom provient des deux caractères qui le caractérisent : le croisillon (« # », *hash* en anglais), et le point d'exclamation (« ! », surnommé *bang* en anglais).

Cette ligne, utilisée dans tous les langages de script (et pas seulement les shell scripts), indique au système d'exploitation que le fichier courant n'est pas qu'un simple fichier texte, mais qu'il s'agit d'un script. Le format de cette ligne est le suivant :

```
#!<commande pour lancer l'interpréteur>
```

Lorsqu'on lance un fichier texte exécutable sur un système d'exploitation de type UNIX alors que ce fichier n'a pas de *shebang*, le shell dans lequel on tente de l'exécuter s'instanciera lui-même pour exécuter ce fichier : le script est exécuté avec le même shell que celui qu'on utilise, qui est certainement le shell par défaut. Par ailleurs, en interface graphique, si l'on clique sur un fichier texte exécutable n'ayant pas de *shebang*, l'environnement risque de simplement ouvrir ce fichier dans un éditeur de texte, ne sachant pas quel interpréteur exécuter.

Par contre, si le fichier texte contient un *shebang*, le système d'exploitation exécutera la commande qui y est mentionnée en lui donnant le chemin du fichier comme premier argument (comme on le ferait manuellement). Notez que la commande indiquée dans le *shebang* n'est pas nécessairement un simple nom d'exécutable : cela peut être une commande avec des arguments.

Voici quelques *shebang* que l'on peut couramment rencontrer :

```
#!/bin/sh
```

... exécution du script avec l'interpréteur shell compatible POSIX par défaut (dans Ubuntu ou Debian par exemple, ce sera *Dash*).

```
#!/bin/sh -x
```

... exécution avec l'interpréteur par défaut, en affichant chaque commande rencontrée sur `stderr` avant son exécution (mais après interprétation des variables : cela permet de s'assurer que les commandes sont bien exécutées comme on l'entend, avec les contenus de variables attendus).

```
#!/bin/awk -f
```

... exécution avec la commande `awk` (qui propose un tout autre langage) : ici, on doit utiliser l'argument `-f` pour pouvoir préciser un fichier à lire.

```
#!/usr/bin/env python2
```

... exécution avec l'interpréteur Python en version 2 (dans ce cas, il s'agit d'un script écrit dans le langage Python, l'extension du fichier est généralement « `.py` »).

La commande env

Dans les exemples ci-dessus, la commande `env` est utilisée pour lancer l'interpréteur Python. Cette commande n'est pas spécifique à Python : on peut l'utiliser pour n'importe quel interpréteur ! En général, on utilise cette formulation pour exécuter l'exécutable dans un nouvel environnement.

L'intérêt de cette commande est alors simplement d'avoir accès à un *path* correct, même si l'environnement dans lequel elle est lancée est mal configuré.

2. Variables et assignation

Lorsque l'on tape des commandes à la main, on utilise rarement des variables, même si cela est possible. Dans un script par contre, la possibilité de créer des variables est beaucoup exploitée. En effet, quel serait l'intérêt d'un script où tout serait écrit dans le marbre, qui ne pourrait toujours exécuter que les mêmes commandes ?

Les exemples ci-après sont repris dans le script `ch4/variables.sh` des données téléchargeables, vous permettant de visualiser les différents cas en direct.

2.1 Assignation d'une valeur

Comme avec n'importe quel langage de programmation, l'assignation d'une valeur à une variable se fait avec le signe d'égalité « = ». Cependant, la syntaxe dans son ensemble est différente de ce que l'on peut trouver dans les autres langages. En effet, le fonctionnement d'un shell suit toujours le même schéma : une instruction, c'est une commande suivie d'arguments. Par conséquent, il n'est notamment pas possible de mettre des espaces autour du signe « = », car de tels espaces signifieraient que le nom de variable est une commande, que son premier argument est le signe d'égalité et son second argument est la valeur qu'on essaie de placer dans la variable :

```
var="val"
```

... ceci est une syntaxe correcte.

```
var = "val"
```

... cette ligne revient à exécuter la commande `var` avec les arguments « = » et « `val` ».

Bien que certains shells acceptent des noms de variables avec caractères spéciaux (notamment les accents), il est préférable de ne pas les utiliser : la meilleure façon d'être en sécurité est de se limiter aux caractères alphabétiques de base, ainsi qu'éventuellement le caractère de soulignement « _ ». Les majuscules et les minuscules, quant à elles, sont correctement différenciées.

2.2 Type de données

Dans l'exemple ci-dessus, la valeur a été explicitement donnée sous forme d'une chaîne de caractères, entourée de guillemets. En réalité, toutes les valeurs sont des chaînes de caractères : il n'existe pas plusieurs types de valeurs en shell script. D'ailleurs, il n'est pas nécessairement indispensable d'encadrer une chaîne de caractères de guillemets ; les deux lignes suivantes sont équivalentes :

```
var="val"
var=val
```