

SHELL

SCRIPTS & EXPRESSIONS RATIONNELLES





VARIABLES

VARIABLES D'ENVIRONNEMENT

Lorsqu'un script est exécuté, un ensemble de variables est directement accessibles proposant des informations sur l'environnement dans lequel le script se déroule : les **VARIABLES D'ENVIRONNEMENT**

Afficher la liste des variables d'environnement :

```
printenv
```

Afficher la valeur d'une variable d'environnement :

```
printenv SHELL  
  
/bin/bash
```

Afficher la valeur d'une variable d'environnement dans un script :

```
echo $SHELL  
  
/bin/bash
```

Définir une variable d'environnement :

```
export NOM_VARIABLE=VALEUR_VARIABLE
```

ATTENTION

export définit une variable d'environnement temporaire qui expirera à la fin de la session courante.

EXEMPLES DE VARIABLES D'ENVIRONNEMENT

\$SECONDS

Fournit le nombre de secondes écoulées depuis le début de l'exécution du script.

\$?

Fournit le **CODE DE RETOUR** de la dernière commande.

\$USER

Contient le nom de l'utilisateur qui exécute le script.

\$EUID

Fournit l'identifiant de l'utilisateur qui exécute le script.

\$PWD

Indique le répertoire de travail courant.

\$OLDPWD

Indique le répertoire de travail précédent.

\$RANDOM

Fonction BASH qui retourne un entier aléatoire compris en 0 et 32767.

\$IFS (Internal Field Separator)

Séparateur utilisé pour décomposer une chaîne de caractères en tableau.

De la même manière que l'on fournit des arguments à une commande, il est possible de passer des paramètres à un script :

```
./mon_script.sh paramètre_1 "paramètre avec espace"
```

\$#

Fournit le nombre de paramètres passés au script.

```
echo $#
```

```
2
```

\$@

Fournit la liste des paramètres transmis séparés par un espace.

```
echo $@
```

```
paramètre_1 paramètre avec espace
```

\$0 ... \$n

Fournit la valeur du i^{ème} paramètre.

```
echo $1
```

```
paramètre_1
```

```
echo $2
```

```
paramètre avec espace
```

```
echo $0
```

```
./mon_script.sh
```

VARIABLES SIMPLES

Pour définir une variable ou en modifier le contenu

```
nom_variable=valeur_variable
```

Exemples

```
utilisateur=dupont
```

```
nomComplet="Jean Dupont"
```

```
entier=933
```

Pour lire le contenu d'une variable

```
$nom_variable
```

Exemples

```
echo $utilisateur
```

```
echo Nom complet : $nomComplet
```

```
echo "Valeur de l'entier : ${entier}"
```

ATTENTION

- Pas d'espace autour du =
- entier est une chaîne de caractères

ATTENTION

- \$ obligatoire pour accéder à une variable en lecture

Pour définir un tableau

```
nom_tableau[indice]=valeur  
# Exemples  
departements[0]="Matériaux"  
departements[1]="Informatique / Electronique"  
departements[2]="Robotique"  
# Autre méthode  
userTypes=("Teacher" "Student")
```

Accéder à un élément du tableau

```
${nom_tableau[indice]}  
# Exemple  
echo ${departement[0]}
```

Afficher la liste des éléments d'un tableau

```
${nom_tableau[@]}  
# Exemple  
echo ${departement[@]}
```

Afficher le nombre d'éléments du tableau

```
${#nom_tableau[@]}  
# Exemple  
echo ${#departement[@]}
```

Le mot clé declare permet de **déclarer une variable**, d'en **définir le type** et de **restreindre sa portée**.

```
# Déclare une variable ne pouvant contenir que des entiers
```

```
declare -i entier=12
```

```
# Déclare un tableau
```

```
declare -a tableau
```

```
# Déclare une constante
```

```
declare -r PI=3.14159
```



BONNE PRATIQUE

La syntaxe `$((expression))` permet de réaliser des opérations arithmétiques sur des variables

```
# Faire la somme des variables entier1 et entier2  
declare -ir entier1=12  
declare -ir entier2=24  
  
echo $entier1 + $entier2 = $((entier1+entier2))
```

En **BASH** seul les entiers sont pris en compte. Utiliser **ZSH** pour les nombres décimaux.

La commande **bc** fournit une calculatrice en ligne mais nécessite une installation du paquet correspondant :

```
echo "1712 * 89 + 18" | bc
```

L'indirection permet d'accéder au contenu d'une variable à partir d'une chaîne de caractères contenant son nom.

Exemple

```
declare -r game="Starfield"
```

```
declare -r variableName="game"
```

```
echo ${!variableName}
```



TESTS

Utilisation de la commande test

```
# Tester l'égalité entre deux valeurs
test 10 = 010
# Afficher le résultat du test
echo $?
# 1
```

Utilisation de l'opérateur []

```
# Tester l'égalité entre deux valeurs
[ 10 -eq 010 ]
# Afficher le résultat du test
echo $?
# 0
```

ATTENTION

- En Shell, **0 = VRAI** et **1 = FAUX**
- L'opérateur = teste l'égalité entre deux chaînes de caractères
- Bien respecter les espaces autour des []

```
# Tester l'égalité
```

```
test "chaine1" = "chaine2"
```

```
# Tester la différence
```

```
test "chaine1" != "chaine2"
```

```
# La chaine est elle vide ?
```

```
test -z "chaine1"
```

```
# La chaine contient elle des caractères
```

```
test -n "chaine1"
```

```
# Tester l'égalité
test nb1 -eq nb2

# Tester la différence
test nb1 -ne nb2

# Tester l'infériorité stricte
test nb1 -lt nb2

# Tester l'infériorité ou l'égalité
test nb1 -le nb2

# Tester la supériorité stricte
test nb1 -gt nb2

# Tester la supériorité ou l'égalité
test nb1 -ge nb2
```

```
# NOT
test ! bool1

# AND
test bool1 -a bool2

# OR
test bool1 -o bool2
```

```
# Exemple
declare -i a=5
declare -i b=10
declare -i c=10

# Teste si a = b ET b = c
[ $a -eq $b -a $b -eq $c ]
echo $?

# Teste si a = b OU b = c
[ $a -eq $b -o $b -eq $c ]
echo $?
```

```
# L'opérateur [[ ]] offre plus de
fonctionnalités et correspond davantage aux
syntaxes traditionnelles
declare -i a=5
declare -i b=10
declare -i c=10

[[ !($a -eq $b) && ($b -eq $c) ]]
echo $?
```

- Le premier test affichera 1 (faux)
- Le second test affichera 0 (vrai)

- Le test affichera 0 (vrai)

```
# Le fichier existe-t-il ?  
test -e nomFichier  
  
# Est-ce un dossier ?  
test -d nomFichier  
  
# Est-ce un fichier ?  
test -f nomFichier  
  
# Est-ce un lien symbolique ?  
test -L nomFichier  
  
# fichier1 est il plus récent que fichier 2 ?  
test fichier1 -nt fichier2  
  
# fichier1 est il plus ancien que fichier 2 ?  
test fichier1 -ot fichier2
```

```
# L'utilisateur courant possède-t-il...  
  
# ... le droit de lecture sur le fichier ?  
test -r nomFichier  
  
# ... le droit d'écriture sur le fichier ?  
test -w nomFichier  
  
# ... le droit d'exécution sur le fichier ?  
test -x nomFichier
```

IF SIMPLE

```
if [[ ! (-e nomFichier) ]]
then
    echo "Le fichier n'existe pas !"
    exit -1
fi
```

IF / ELSE

```
if [[ ! (-e nomFichier) ]]
then
    echo "Le fichier n'existe pas !"
    exit -1
else
    echo "Le fichier existe."
fi
```

IF / ELSE IF / ELSE

```
if [[ ! (-e nomFichier) ]]
then
    echo "Le fichier n'existe pas !"
    exit -1
elif [[ -f nomFichier ]]
then
    echo "C'est un fichier."
else
    echo "Ce n'est pas un fichier."
fi
```

ATTENTION

à ne pas oublier le **then** et le **fi**

```
case $saisieUtilisateur in
  1)      echo "Choix 1" ;;
  2)      echo "Choix 2"
          echo "Excellent choix" ;;
  "end")  echo "Fin du programme"
          exit 0 ;;
  *)      echo "Ce choix n'est pas proposé..." ;;
esac
```



BOUCLES

Itérations conditionnées

```
for (( i = 0 ; i < 10 ; i++ ))  
do  
    echo "Itération "$i  
done
```

Parcours d'un intervalle

```
for i in {1..10}  
do  
    echo "Itération "$i  
done
```

Parcours d'un intervalle avec pas

```
for i in {1..10..2}  
do  
    echo "Itération "$i  
done
```

Parcours d'un tableau

```
for jour in lundi mardi mercredi jeudi vendredi samedi dimanche  
do  
    echo $jour  
done
```

Boucle sur condition

```
while [[ $saisie != "fin" ]]  
do  
    read -p "Saisissez fin pour arrêter : " saisie  
done
```

Boucle sur résultat d'une commande

```
dossiers[0]="/bin"  
dossiers[1]="/usr"  
dossiers[2]="/toto"  
dossiers[3]="/home"  
  
iDossier=0  
dossier=${dossiers[$iDossier]}  
  
while ls $dossier  
do  
    echo "ok: "$dossier  
    iDossier=$((iDossier + 1))  
    dossier=${dossiers[$iDossier]}  
done
```



FONCTIONS

```
# Déclaration de la fonction pi
```

```
function pi
```

```
{
```

```
    echo 3.14159
```

```
}
```

```
# Appel de la fonction pi
```

```
pi
```

```
# Syntaxe alternative
```

```
pi()
```

```
{
```

```
    echo 3.14159
```

```
}
```

```
# Appel de la fonction pi
```

```
pi
```

```
# Déclaration d'une fonction
```

```
# prenant deux paramètres
```

```
somme()
```

```
{
```

```
    echo $(( $1 + $2 ))
```

```
}
```

```
# Appel de la fonction somme
```

```
somme 18 24
```

MÉTHODE 1

Récupération du flux de sortie

```
# Déclaration de la fonction
function somme
{
    echo $(( $1 + $2 ))
}

# Récupération du résultat
echo $(somme 24 18)
```

MÉTHODE 2

Récupération du code de retour

```
# Déclaration de la fonction
function somme
{
    return $(( $1 + $2 ))
}

# Appel de la fonction
somme 24 18

# Récupération du code de retour
echo $?
```

ATTENTION

Ne fonctionne qu'avec des valeurs de retour numérique !

MÉTHODE 3

Création d'une variable

```
# Déclaration de la fonction
function somme
{
    resultat=$(( $1 + $2 ))
}

# Appel de la fonction
somme 24 18

# Récupération du résultat
echo $resultat
```



EXPRESSIONS RATIONNELLES

```
enovations@LAPTOP-B3V8QTN0:~$ ls -l
total 0
drwxr-xr-x 1 enovations enovations 512 Oct  8 14:31 images
drwxr-xr-x 1 enovations enovations 512 Oct  8 14:31 logs
-rwxr--r-- 1 enovations enovations  13 Oct  8 14:31 script.sh
-rw-r--r-- 1 enovations enovations   6 Oct  8 14:31 texte.txt
```

Comment trouver les fichiers exécutables ?

Les motifs standards
sont trop limités pour faire
des recherches avancées

```
enovations@LAPTOP-B3V8QTN0:~$ ls -l | grep --color=always "x"
drwxr-xr-x 1 enovations enovations 512 Oct  8 14:31 images
drwxr-xr-x 1 enovations enovations 512 Oct  8 14:31 logs
-rwxr--r-- 1 enovations enovations  13 Oct  8 14:31 script.sh
-rw-r--r-- 1 enovations enovations   6 Oct  8 14:31 textext
```

L'option **--color** de **grep** met en évidence les correspondances identifiées.

Une expression rationnelle permet de décrire des motifs complexes :

```
ls -l | grep -E "^-.{0,6}(..x){1,3}"
```

```
enovations@LAPTOP-B3V8QTN0:~$ ls -l | grep -E "^-.{0,6}(..x){1,3}" --color=always  
-rwxr--r-- 1 enovations enovations 13 Oct  8 14:31 script.sh
```

L'option **-E** de **grep** indique que le motif est donné sous la forme d'une expression rationnelle.

Une expression rationnelle décrit une succession attendue de caractères :

- **bon** 'b' suivi de 'o' suivi de 'n'
- **ab** 'a' suivi de 'b'
- **13** '1' suivi de '3'

Il est possible de préciser où chercher le motif dans la chaîne de caractères :

- **^Bonjour** ^ signifie que le motif doit se trouver au début de la chaîne de caractères.

 Bonjour Madame OK
 Madame **Bonjour** PAS OK
- **txt\$** \$ signifie que le motif doit se trouver à la fin de la chaîne de caractères.

 fichier.**txt** OK
 fichier.txt.**old** PAS OK
- **^utilisateur\$** La combinaison de ^ et de \$ indique que la chaîne de caractères doit correspondre à la totalité du motif.

 utilisateur OK
 L'utilisateur PAS OK

PLUSIEURS MOTIFS POSSIBLES

L'opérateur **|** permet d'indiquer plusieurs motifs pouvant correspondre à la recherche :

- **jpg|bmp|png** recherchera les motifs "**jpg**" ou "**bmp**" ou "**png**"

```
index.html  
image1.bmp  
image2.jpg  
text.txt  
pngviewer.exe
```

L'opérateur [] indique que l'on cherche un caractère parmi ceux proposés :

- **[abc]** Le prochain caractère attendu peut être un 'a', un 'b' ou un 'c'
- **[a-z]** Le prochain caractère attendu est un lettre minuscule
- **[a-zA-Z]** Le prochain caractère attendu est une lettre (minuscule ou majuscule)
- **[0-9]** Le prochain caractère attendu est un chiffre
- **[^0-9]** Le prochain caractère attendu **ne doit pas être** un chiffre

- `.` Le prochain caractère attendu peut être n'importe quel caractère
- `\w` Le prochain caractère attendu est une lettre (minuscule ou majuscule)
- `\d` Le prochain caractère attendu est un chiffre
- `\n` Le prochain caractère attendu est un retour à la ligne
- `\t` Le prochain caractère attendu est une tabulation
- `\.` Le prochain caractère attendu est un point
- `\\` Le prochain caractère attendu est un '\'

Pour ne pas être interprétés et faire partie du motif, les caractères suivants doivent être précédés d'un '\' :

`( ) { } [ ] \ . + - * ? | , ^ $`

- $a\{2\}$ Deux caractères 'a' doivent se suivre.
- $a\{1,2\}$ Entre 1 et 2 caractères 'a' doivent se suivre.
- $a\{,2\}$ Au maximum 2 caractères 'a' doivent se suivre. Il peut ne pas y en avoir.
- $a\{2, \}$ Au moins 2 caractères 'a' doivent se suivre. Il peut y en avoir plus.
- a^* Plusieurs caractères 'a' peuvent se suivre. Il peut ne pas y en avoir.
- a^+ Plusieurs caractères 'a' doivent se suivre; Il doit y en avoir au moins un.
- $a?$ Il peut y avoir un caractère 'a' ou aucun

```
enovations@LAPTOP-B3V8QTN0:~$ ls -l
total 0
drwxr-xr-x 1 enovations enovations 512 Oct  8 14:31 images
drwxr-xr-x 1 enovations enovations 512 Oct  8 14:31 logs
-rwxr--r-- 1 enovations enovations  13 Oct  8 14:31 script.sh
-rw-r--r-- 1 enovations enovations   6 Oct  8 14:31 texte.txt
```

Comment trouver les fichiers exécutable ?

- Ce ne doit pas être un dossier
- Le fichier doit avoir au moins un type d'utilisateur possédant les droits d'exécution
- Les droits se trouvent en début de ligne

[^] - . { 0 , 6 } (. . x) { 1 , 3 }

Le motif recherché
est en début de ligne

Ce n'est pas un
dossier = le premier
caractère de la ligne
doit être un tiret

Il y a 3 types d'utilisateurs,
chacun disposant de 3 droits :
rwx

Au maximum, les deux premiers
types d'utilisateurs peuvent ne
pas avoir les droits d'exécution.

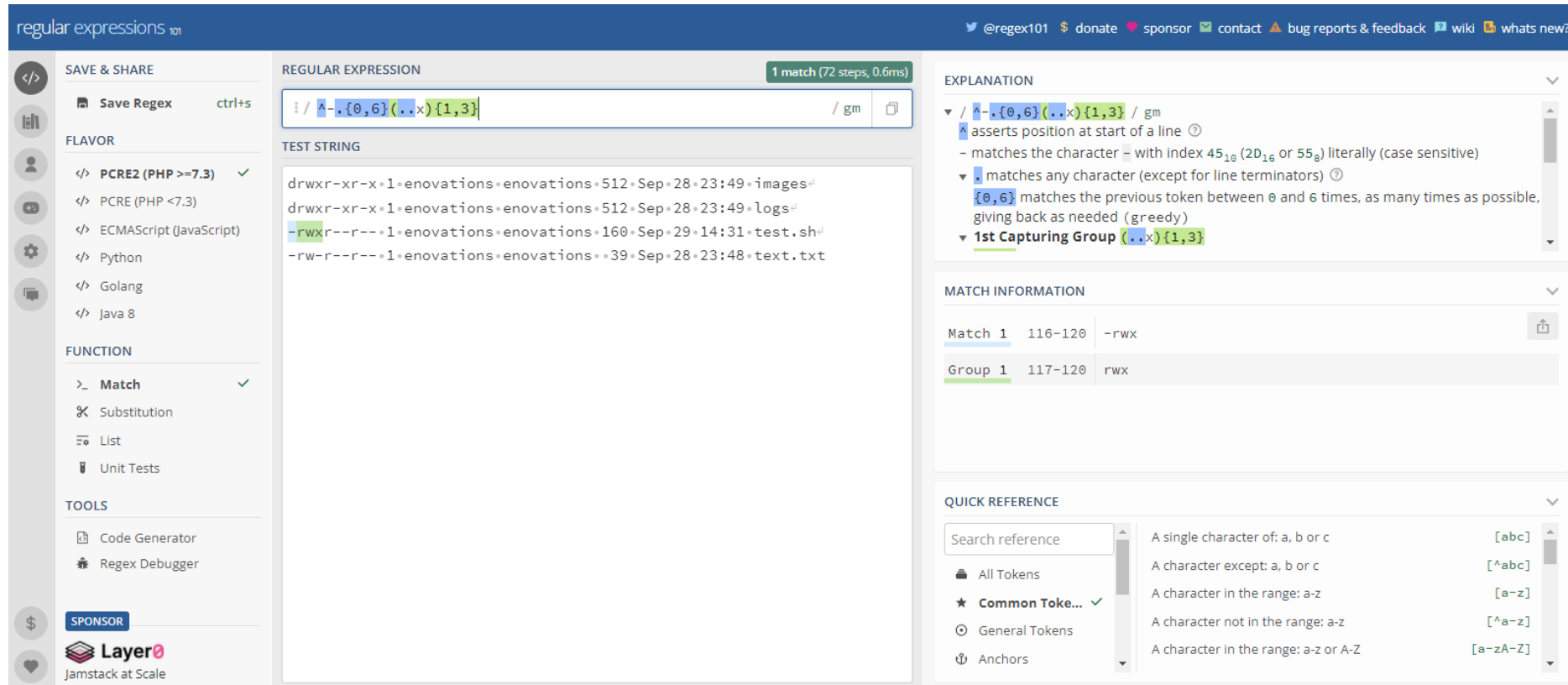
Donc au maximum, il peut y
avoir 6 caractères quelconques
qui se suivent.

Au moins 1 type d'utilisateurs
doit avoir les droits d'exécution.

Au maximum, 3 peuvent les
avoir.

Les droits en lecture ou en
écriture ne nous intéressent pas.

Les expressions rationnelles proposent de nombreuses autres fonctionnalités.



The screenshot shows the regex101.com interface. The top bar includes the site name "regular expressions 101" and navigation links like "@regex101", "donate", "sponsor", "contact", "bug reports & feedback", "wiki", and "whats new?". The left sidebar contains sections for "SAVE & SHARE", "FLAVOR" (with options like PCRE2, PCRE, ECMAScript, Python, Golang, Java 8), "FUNCTION" (with options like Match, Substitution, List, Unit Tests), and "TOOLS" (with options like Code Generator, Regex Debugger). The main area is divided into three panels: "REGULAR EXPRESSION" (showing the regex `/^-\{0,6\}(\.\.x){1,3}/gm`), "TEST STRING" (showing a list of file permissions and paths), and "EXPLANATION" (providing a detailed breakdown of the regex components). The "MATCH INFORMATION" panel shows a single match for the regex, and the "QUICK REFERENCE" panel provides a search bar and a list of common tokens with their corresponding regex patterns.

REGULAR EXPRESSION 1 match (72 steps, 0.6ms)

`/^-\{0,6\}(\.\.x){1,3}/gm`

TEST STRING

drwxr-xr-x+1+enovations+enovations+512+Sep+28+23:49+images+
drwxr-xr-x+1+enovations+enovations+512+Sep+28+23:49+logs+
-rwxr--r--+1+enovations+enovations+160+Sep+29+14:31+test.sh+
-rw-r--r--+1+enovations+enovations+39+Sep+28+23:48+text.txt

EXPLANATION

- `/` asserts position at start of a line
- `^` matches the character `^` with index 45₁₀ (2D₁₆ or 55₈) literally (case sensitive)
- `-` matches any character (except for line terminators)
- `{0,6}` matches the previous token between 0 and 6 times, as many times as possible, giving back as needed (greedy)
- 1st Capturing Group** `(\.\.x){1,3}`

MATCH INFORMATION

Match	Start	End	Match
1	116	120	-rwx

QUICK REFERENCE

Token	Pattern
A single character of: a, b or c	<code>[abc]</code>
A character except: a, b or c	<code>[^abc]</code>
A character in the range: a-z	<code>[a-z]</code>
A character not in the range: a-z	<code>[^a-z]</code>
A character in the range: a-z or A-Z	<code>[a-zA-Z]</code>

<https://regex101.com>



TÂCHES PLANIFIÉES

La commande **crontab** permet de gérer les tâches planifiées de l'utilisateur

Lister les tâches planifiées de l'utilisateur

```
crontab -l
```

Modifier les tâches planifiées de l'utilisateur

```
crontab -e
```

Supprimer toutes les tâches planifiées de l'utilisateur

```
crontab -r
```

ATTENTION

Aucune confirmation ne sera demandée !

PLANIFIER UN TÂCHE

Une tâche est définie par 6 paramètres :

- **Minute** Minute [0-59] à laquelle doit s'exécuter la tâche
- **Heure** Heure [0-23] à laquelle doit s'exécuter la tâche
- **Jour du mois** Jour du mois [1-31] durant lequel doit s'exécuter la tâche
- **Mois** Mois [1-12] durant lequel doit s'exécuter la tâche
- **Jour de la semaine** Jour de la semaine [0-6] durant lequel doit s'exécuter la tâche (*0 = dimanche*)
- **commande** La commande à exécuter

VALEURS POSSIBLES

Exemple pour les minutes :

- **13** La tâche s'exécutera à la 13^{ème} minute
- ***** La tâche s'exécutera toutes les minutes
- **1-5** La tâche s'exécutera aux minutes 1, 2, 3, 4 et 5
- ***/10** La tâche s'exécutera toutes les 10 minutes
- **3,8,26** La tâche s'exécutera les minutes 3, 8 et 26

Tâche exécutée à chaque minute

```
* * * * * /var/cron/save.sh
```

Tâche exécutée tous les jours à 23h00

```
0 23 * * * /var/cron/save.sh
```

Tâche exécutée tous les 1^{ers} du mois à 23h00

```
0 23 1 * * /var/cron/save.sh
```

Tâche exécutée tous mercredis à 23h00

```
0 23 * * 3 /var/cron/save.sh >> /var/cron/save.log 2>>&1
```

RECOMMANDATIONS

- Donner le chemin absolu des scripts à exécuter
- Rediriger les sorties vers des fichiers de log



LA PROCHAINE FOIS

Pas de prochaine fois