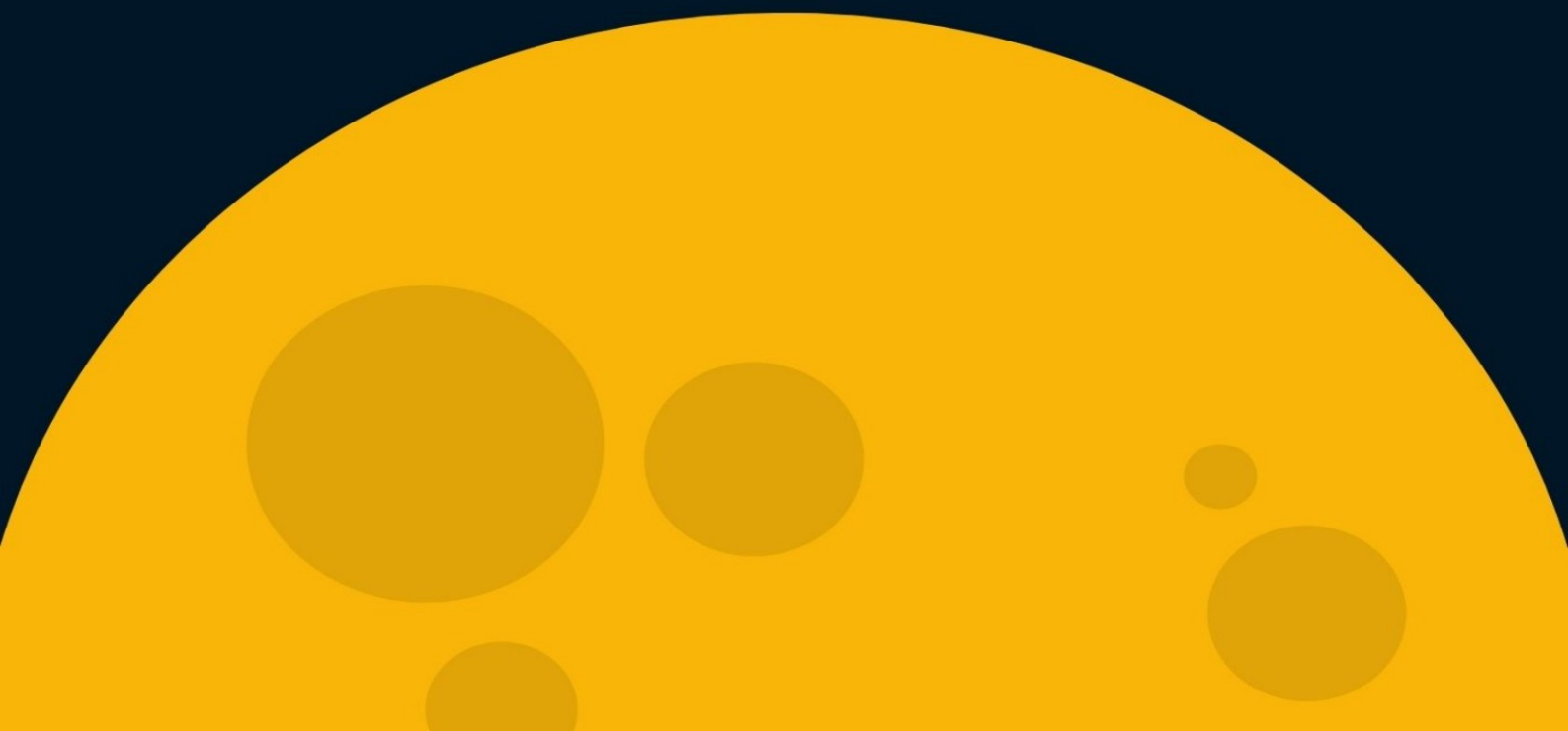
BY BOBBY ILIEV

Introduction to Bash Scripting

FOR DEVELOPERS



Machine Translated by Google

- **Esta versión fue publicada el 01 de febrero de 2021**

Esta es una introducción de código abierto a la guía de secuencias de comandos de Bash que lo ayudará a aprender los conceptos básicos de las secuencias de comandos de Bash y comenzar a escribir secuencias de comandos de Bash increíbles que lo ayudarán a automatizar sus tareas diarias de SysOps, DevOps y Dev. No importa si es un ingeniero, desarrollador o simplemente un entusiasta de Linux DevOps/SysOps, puede usar scripts de Bash para combinar diferentes comandos de Linux y automatizar tareas diarias tediosas y repetitivas para que pueda concentrarse en cosas más productivas y divertidas.

La guía es adecuada para cualquier persona que trabaje como desarrollador, administrador de sistemas o ingeniero de DevOps y quiera aprender los conceptos básicos de las secuencias de comandos de Bash.

Los primeros 13 capítulos se centrarían únicamente en obtener una base sólida de secuencias de comandos de Bash, luego el resto de los capítulos le brindarán algunos ejemplos y secuencias de comandos de la vida real.

Mi nombre es Bobby Iliev y he trabajado como ingeniero DevOps de Linux desde 2014. Soy un ávido amante de Linux y partidario de la filosofía del movimiento de código abierto. Siempre estoy haciendo lo que no puedo hacer para poder aprender a hacerlo, y creo en compartir el conocimiento.

Creo que siempre es esencial mantenerse profesional y rodearse de buenas personas, trabajar duro y ser amable con todos. Tienes que desempeñarte a un nivel consistentemente más alto que los demás. Esa es la marca de un verdadero profesional.

Para obtener más información, visite mi blog en https://bobbyiliev.com, Sígueme en Twitter @bobbyiliev_ y YouTube.

¡Este libro es posible gracias a estas fantásticas empresas!

La base de datos de transmisión para análisis en tiempo real.

Materializar es una base de datos reactiva que ofrece actualizaciones de vista incrementales. Materialise ayuda a los desarrolladores a construir fácilmente con transmisión de datos utilizando SQL estándar.

DigitalOcean es una plataforma de servicios en la nube que ofrece la simplicidad que a los desarrolladores les encanta y en la que las empresas confían para ejecutar aplicaciones de producción a escala.

Proporciona soluciones informáticas, de almacenamiento y de redes de alta disponibilidad, seguras y escalables que ayudan a los desarrolladores a crear software excelente más rápido.

Fundada en 2012 con oficinas en Nueva York y Cambridge, MA, DigitalOcean ofrece precios transparentes y asequibles, una elegante interfaz de usuario y una de las mayores bibliotecas de recursos de código abierto disponibles.

Para obtener más información, visite https://www.digitalocean.com o sigue a @digitalocean en Twitter.

Si es nuevo en DigitalOcean, puede obtener un crédito gratuito de $ 100 y activar sus propios servidores a través de este enlace de referencia aquí:

$ 100 de crédito gratis para DigitalOcean

El DevDojo es un recurso para aprender todo lo relacionado con el desarrollo web y el diseño web. Aprenda en su hora de almuerzo o despiértese y disfrute de una taza de café con nosotros para aprender algo.
nuevo.

Únase a esta comunidad de desarrolladores y todos podemos aprender juntos, construir juntos y crecer juntos.

Únete a DevDojo

Para obtener más información, visite https://www.devdojo.com o sigue a @thedevdojo en Twitter.

Este libro electrónico fue generado por Ibis desarrollado por Mohamed Said.

Ibis es una herramienta PHP que te ayuda a escribir libros electrónicos en Markdown.

La portada de este libro electrónico se creó con Canva.com.

Si alguna vez necesitas crear un gráfico, un póster, una invitación, un logotipo, una presentación, o cualquier cosa que se vea bien, prueba Canva.

¡Bienvenido a esta guía de capacitación básica de Bash! En este **curso intensivo de bash,** aprenderá los **conceptos básicos de Bash** para que pueda comenzar a escribir sus propios scripts de Bash y automatizar sus tareas diarias.

Bash es un lenguaje de comando y shell de Unix. Está ampliamente disponible en varios sistemas operativos y también es el intérprete de comandos predeterminado en la mayoría de los sistemas Linux.

Bash significa Bourne-Again SHell. Al igual que con otros shells, puede usar Bash de forma interactiva directamente en su terminal, y también puede usar Bash como cualquier otro lenguaje de programación para escribir scripts. Este libro lo ayudará a aprender los conceptos básicos de las secuencias de comandos de Bash, incluidas las variables de Bash, la entrada del usuario, los comentarios, los argumentos, las matrices, las expresiones condicionales, los condicionales, los bucles, las funciones, la depuración y las pruebas.

Para escribir scripts Bash, solo necesita una terminal UNIX y un editor de texto como Sublime Text, VS Code o un editor basado en terminal como vim o nano.

Comencemos creando un nuevo archivo con una extensión .sh . Como ejemplo, podríamos crear un archivo llamado devdojo.sh.

Para crear ese archivo, puede usar el comando táctil :

```
tocar devdojo.sh
```

O puede usar su editor de texto en su lugar:

```
nano devdojo.sh
```

Para ejecutar/ejecutar un archivo de script bash con el intérprete de shell bash, la primera línea de un archivo de script debe indicar la ruta absoluta al ejecutable bash:

```
#!/bin/bash
```

Esto también se llama Shebang.

Todo lo que hace Shebang es indicar al sistema operativo que ejecute el script con el ejecutable / bin/bash .

Una vez que hayamos creado nuestro archivo devdojo.sh y hayamos especificado bash shebang en la primera línea, estamos listos para crear nuestro primer script bash Hello World .

Para hacer eso, abra el archivo devdojo.sh nuevamente y agregue lo siguiente después de la línea #!/bin/bash :

```
#!/bin/bash

echo "¡Hola mundo!"
```

Guarda el archivo y cierra.

Después de eso, haga que el script sea ejecutable ejecutando:

```
chmod +x devdojo.sh
```

Después de eso, ejecute el archivo:

```
./devdojo.sh
```

Verá un mensaje "Hello World" en la pantalla.

Otra forma de ejecutar el script sería:

```
bash devdojo.sh
```

Como bash se puede usar de forma interactiva, puede ejecutar el siguiente comando directamente en su terminal y obtendrá el mismo resultado:

```
echo "¡Hola DevDojo!"
```

Armar un script es útil una vez que tiene que combinar varios comandos.

Como en cualquier otro lenguaje de programación, también puede usar variables en Bash Scripting. Sin embargo, no hay tipos de datos y una variable en Bash puede contener números además de caracteres.

Para asignar un valor a una variable, todo lo que necesita hacer es usar el signo = :

```
nombre="DevDojo"
```

Aviso: como nota importante, no puede tener espacios antes y después del signo = .

Después de eso, para acceder a la variable, debe usar $ y hacer referencia a ella como se muestra a continuación:

```
echo $nombre
```

No es necesario envolver el nombre de la variable entre corchetes, pero se considera una buena práctica, y le aconsejo que los use siempre que pueda:

```
echo ${nombre}
```

El código anterior daría como resultado: DevDojo ya que este es el valor de nuestra variable de nombre .

A continuación, actualicemos nuestro script devdojo.sh e incluyamos una variable en él.

Nuevamente, puede abrir el archivo devdojo.sh con su editor de texto favorito, estoy usando nano aquí para abrir el archivo:

```
nano devdojo.sh
```

Agregando nuestra variable de nombre aquí en el archivo, con un mensaje de bienvenida. Nuestro archivo ahora se ve así:

```
#!/bin/bash

nombre="DevDojo"

echo "Hola $nombre"
```

Guárdelo y ejecute el archivo usando el siguiente comando:

```
./devdojo.sh
```

Vería el siguiente resultado en su pantalla:

```
Hola DevDojo
```

Aquí hay un resumen del script escrito en el archivo:

- #!/bin/bash - Al principio, especificamos nuestro shebang.
- name=DevDojo - Luego, definimos una variable llamada nombre y le asignamos un valor.
- echo "Hola $nombre" - Finalmente, mostramos el contenido de la variable en la pantalla como un mensaje de bienvenida usando echo

También puede agregar múltiples variables en el archivo como se muestra a continuación:

```
#!/bin/bash

nombre="DevDojo"
saludo="Hola"

echo "$saludo $nombre"
```

Guarda el archivo y ejecútalo de nuevo:

```
./devdojo.sh
```

Vería el siguiente resultado en su pantalla:

```
Hola DevDojo
```

Tenga en cuenta que no necesariamente necesita agregar un punto y coma ; al final de cada línea. Funciona en ambos sentidos, ¡un poco como otro lenguaje de programación como JavaScript!

Con el script anterior, definimos una variable y mostramos el valor de la variable en la pantalla con el echo $name.

Ahora sigamos adelante y pidamos al usuario una entrada en su lugar. Para hacerlo nuevamente, abra el archivo con su editor de texto favorito y actualice el script de la siguiente manera:

```
#!/bin/bash

echo "¿Cuál es tu nombre?" leer
nombre

echo "Hola $nombre" echo
"¡Bienvenido a DevDojo!"
```

Lo anterior le pedirá al usuario que ingrese y luego almacenará esa entrada como una cadena/texto en una variable.

Luego podemos usar la variable e imprimirles un mensaje.

La salida del script anterior sería:

- Primero ejecute el script:

```
./devdojo.sh
```

- Luego, se le pedirá que ingrese su nombre:

```
¿Cuál es su nombre?
Poli
```

- Una vez que haya escrito su nombre, simplemente presione enter y obtendrá el siguiente resultado:

```
Hola, Bobby.
¡Bienvenido a DevDojo!
```

Para reducir el código, podríamos cambiar la primera declaración de eco con read -p, el comando de lectura usado con el indicador -p imprimirá un mensaje antes de solicitar al usuario su entrada:

```
#!/bin/bash

read -p "¿Cuál es tu nombre?"          nombre

echo "Hola $nombre" echo
"¡Bienvenido a DevDojo!"
```

¡Asegúrate de probar esto tú mismo también!

Al igual que con cualquier otro lenguaje de programación, puede agregar comentarios a su script. Los comentarios se utilizan para dejarte notas a través de tu código.

Para hacer eso en Bash, debe agregar el símbolo # al comienzo de la línea.
Los comentarios nunca se mostrarán en la pantalla.

Aquí hay un ejemplo de un comentario:

```
# Este es un comentario y no aparecerá en la pantalla
```

Avancemos y agreguemos algunos comentarios a nuestro script:

```
#!/bin/bash

# Preguntar al usuario por su nombre

read -p "¿Cuál es tu nombre?" nombre

# Saludar al usuario echo
"Hola $nombre" echo "¡Bienvenido
a DevDojo!"
```

Los comentarios son una excelente manera de describir algunas de las funciones más complejas directamente en sus scripts para que otras personas puedan orientarse en su código con facilidad.

Puede pasar argumentos a su script de shell cuando lo ejecuta. Para pasar un argumento, solo necesita escribirlo justo después del nombre de su script. Por ejemplo:

```
./devdojo.com tu_argumento
```

En el script, podemos usar $1 para hacer referencia al primer argumento que especificamos.

Si pasamos un segundo argumento, estaría disponible como $2 y así sucesivamente.

Vamos a crear un script corto llamado arguments.sh como ejemplo:

```
#!/bin/bash

echo "El argumento uno es $1"
echo "El argumento dos es $2"
echo "El argumento tres es $3"
```

Guarde el archivo y hágalo ejecutable:

```
chmod +x argumentos.sh
```

Luego ejecute el archivo y pase **3** argumentos:

```
./arguments.sh perro gato pájaro
```

La salida que obtendrías sería:

```
Argumento uno es perro
El argumento dos es el gato.
El argumento tres es pájaro.
```

Para hacer referencia a todos los argumentos, puede usar $@:

```
#!/bin/bash

echo "Todos los argumentos: $@"
```

Si ejecuta el script de nuevo:

```
./arguments.sh perro gato pájaro
```

Obtendrá el siguiente resultado:

```
Todos los argumentos: perro gato pájaro
```

Otra cosa que debe tener en cuenta es que $0 se usa para hacer referencia al script en sí.

Esta es una excelente manera de autodestruir el archivo si lo necesita o simplemente obtener el nombre del script.

Por ejemplo, creemos un script que imprima el nombre del archivo y lo elimine después de eso:

```
#!/bin/bash

echo "El nombre del archivo es: $0 y se borrará automáticamente".

rm-f $0
```

Debe tener cuidado con la eliminación automática y asegurarse de tener una copia de seguridad de su secuencia de comandos antes de eliminarla automáticamente.

Si alguna vez ha hecho algo de programación, probablemente ya esté familiarizado con las matrices.

Pero en caso de que no sea un desarrollador, lo principal que debe saber es que, a diferencia de las variables, las matrices pueden contener varios valores bajo un mismo nombre.

Puede inicializar una matriz asignando valores divididos por espacios y encerrados entre ().
Ejemplo:

```
my_array=("valor 1" "valor 2" "valor 3" "valor 4")
```

Para acceder a los elementos de la matriz, debe hacer referencia a ellos por su índice numérico.

> **Aviso:** tenga en cuenta que debe usar corchetes.

- Acceda a un solo elemento, esto daría como resultado: valor 2

```
echo ${mi_matriz[1]}
```

- Esto devolvería el último elemento: valor 4

```
echo ${mi_matriz[-1]}
```

- Al igual que con los argumentos de la línea de comandos, el uso de @ devolverá todos los argumentos de la matriz, de la siguiente manera: valor 1 valor 2 valor 3 valor 4

```
echo ${mi_matriz[@]}
```

- Anteponer a la matriz un signo de numeral (#) daría como resultado el número total de elementos en la matriz, en nuestro caso es 4:

```
echo ${#mi_matriz[@]}
```

Asegúrese de probar esto y practicarlo en su extremo con diferentes valores.

Revisemos el siguiente ejemplo de cortar una cadena en Bash:

```
#!/bin/bash

letras=( "A""B""C""D""E" ) echo $
{letras[@]}
```

Este comando imprimirá todos los elementos de una matriz.

Producción:

```
$ ABCD
```

Veamos algunos ejemplos más:

- Ejemplo 1

```
#!/bin/bash

letras=( "A""B""C""D""E" ) b=$
{letras:0:2} echo "${b}"
```

Este comando imprimirá una matriz desde el índice inicial 0 hasta el 2, donde 2 es exclusivo.

```
$ AB
```

- Ejemplo 2

```
#!/bin/bash

letras=( "A""B""C""D""E" ) b=$
{letras::5} echo "${b}"
```

Este comando imprimirá desde el índice base 0 a 5, donde 5 es exclusivo y el índice inicial está predeterminado en 0 .

```
$ ABCD
```

- Ejemplo 3

```
#!/bin/bash

letras=( "A""B""C""D""E" ) b=${letras:3}
echo "${b}"
```

Este comando imprimirá desde el índice inicial 3 hasta el final de la matriz inclusive.

```
$ DE
```

En informática, las declaraciones condicionales, las expresiones condicionales y las construcciones condicionales son características de un lenguaje de programación, que realizan diferentes cálculos o acciones dependiendo de si una condición booleana especificada por el programador se evalúa como verdadera o falsa.

En Bash, las expresiones condicionales son utilizadas por el comando compuesto [[y los comandos integrados [para probar los atributos del archivo y realizar comparaciones aritméticas y de cadenas.

Aquí hay una lista de las expresiones condicionales de Bash más populares. No tienes que memorizarlos de memoria. ¡Simplemente puede volver a consultar esta lista cuando la necesite!

- Verdadero si el archivo existe.

```
[[ -a ${archivo} ]]
```

- Verdadero si el archivo existe y es un archivo especial de bloque.

```
[[ -b ${archivo} ]]
```

- Verdadero si el archivo existe y es un archivo especial de caracteres.

```
[[ -c ${archivo} ]]
```

- Verdadero si el archivo existe y es un directorio.

```
[[ -d ${archivo} ]]
```

- Verdadero si el archivo existe.

```
[[ -e ${archivo} ]]
```

- Verdadero si el archivo existe y es un archivo normal.

```
[[ -f ${archivo} ]]
```

- Verdadero si el archivo existe y es un enlace simbólico.

```
[[ -h ${archivo} ]]
```

- Verdadero si el archivo existe y es legible.

```
[[ -r ${archivo} ]]
```

- Verdadero si el archivo existe y tiene un tamaño mayor que cero.

```
[[ -s ${archivo} ]]
```

- True si el archivo existe y se puede escribir.

```
[[ -w ${archivo} ]]
```

- Verdadero si el archivo existe y es ejecutable.

```
[[ -x ${archivo} ]]
```

- Verdadero si el archivo existe y es un enlace simbólico.

```
[[ -L ${archivo} ]]
```

- True si la variable de shell varname está configurada (se le ha asignado un valor).

```
[[ -v ${varnombre} ]]
```

True si la longitud de la cadena es cero.

```
[[ -z ${cadena} ]]
```

True si la longitud de la cadena no es cero.

```
[[ -n ${cadena} ]]
```

- True si las cadenas son iguales. = debe usarse con el comando de prueba para la conformidad con POSIX. Cuando se usa con el comando [[, este realiza la coincidencia de patrones como se describe arriba (Comandos compuestos).

```
[[ ${cadena1} == ${cadena2} ]]
```

- True si las cadenas no son iguales.

```
[[ ${cadena1} != ${cadena2} ]]
```

- Verdadero si cadena1 se ordena lexicográficamente antes que cadena2.

```
[[ ${cadena1} < ${cadena2} ]]
```

- Verdadero si cadena1 se ordena después de cadena2 lexicográficamente.

```
[[ ${cadena1} > ${cadena2} ]]
```

- Devuelve verdadero si los números son **iguales**

```
[[ ${arg1} -eq ${arg2} ]]
```

- Devuelve verdadero si los números **no son iguales**

```
[[ ${arg1} -ne ${arg2} ]]
```

- Devuelve verdadero si arg1 es **menor que** arg2

```
[[ ${arg1} -lt ${arg2} ]]
```

- Devuelve verdadero si arg1 es **menor o igual que** arg2

```
[[ ${arg1} -le ${arg2} ]]
```

- Devuelve verdadero si arg1 es **mayor que** arg2

```
[[ ${arg1} -gt ${arg2} ]]
```

- Devuelve verdadero si arg1 es **mayor o igual que** arg2

```
[[ ${arg1} -ge ${arg2} ]]
```

Como nota al margen, arg1 y arg2 pueden ser números enteros positivos o negativos.

Al igual que con otros lenguajes de programación, puede usar las condiciones AND & OR :

```
[[ test_case_1 ]] && [[ test_case_2 ]] # Y [[ test_case_1 ]]
|| [[ test_case_2 ]] # O
```

- devuelve verdadero si el comando fue exitoso sin ningún error

```
ps -eq 0 ]]
```

- devuelve verdadero si el comando no tuvo éxito o tuvo errores

```
ps -gt 0 ]]
```

En la última sección, cubrimos algunas de las expresiones condicionales más populares. Ahora podemos usarlos con declaraciones condicionales estándar como declaraciones if, if-else y switch case .

El formato de una sentencia if en Bash es el siguiente:

```
if [[ alguna_prueba ]]
entonces <comandos>

fi
```

Aquí hay un ejemplo rápido que le pedirá que ingrese su nombre en caso de que lo haya dejado vacío:

```
#!/bin/bash

# Ejemplo de instrucción Bash if

read -p "¿Cuál es tu nombre?" nombre

if [[ -z ${name} ]] luego
echo "¡Por favor ingrese
      su nombre!"
fi
```

Con una declaración if-else , puede especificar una acción en caso de que la condición en la declaración if no coincida. Podemos combinar esto con las expresiones condicionales de la sección anterior de la siguiente manera:

```
#!/bin/bash

# Ejemplo de instrucción Bash if

read -p "¿Cuál es tu nombre?"                nombre

if [[ -z ${name} ]] luego
echo "¡Por favor ingrese
        su nombre!" else echo "Hola ${nombre}"


fi
```

Puede usar la declaración if anterior con todas las expresiones condicionales de los capítulos anteriores:

```
#!/bin/bash

admin="devdojo"

read -p "¿Ingrese su nombre de usuario?"   nombre de usuario

# Comprobar si el nombre de usuario proporcionado es el administrador

if [[ "${nombre de usuario}" == "${admin}" ]] ; luego repite
        "¡Eres el usuario administrador!" else echo "¡NO eres el
usuario administrador!"

fi
```

Aquí hay otro ejemplo de una declaración if que verificaría su Usuario actual ID y no le permitiría ejecutar el script como usuario raíz :

```
#!/bin/bash

si (( $EUID == 0)); luego echo "Por
        favor, no ejecute como root" exit

fi
```

Si coloca esto encima de su secuencia de comandos, se cerrará en caso de que el EUID sea 0 y no se ejecute el resto de la secuencia de comandos. Esto se discutió en el foro de la comunidad DigitalOcean .

También puede probar múltiples condiciones con una declaración if . En este ejemplo, queremos asegurarnos de que el usuario no sea el usuario administrador ni el usuario raíz para garantizar que el script no pueda causar demasiado daño. Usaremos el operador o en este ejemplo, señalado por ||. Esto significa que cualquiera de las condiciones debe ser verdadera. Si usamos el operador and de && , entonces ambas condiciones deberían ser verdaderas.

```
#!/bin/bash

admin="devdojo"

read -p "¿Ingrese su nombre de usuario?"        nombre de usuario

# Comprobar si el nombre de usuario proporcionado es el administrador

si [[ "${nombre de usuario}" != "${admin}" ]] || [[ $EUID != 0 ]] ; luego repite "No eres
el administrador ni el usuario raíz, ¡pero ten cuidado!" else echo "¡Usted es el usuario
        administrador! Esto podría ser muy



destructivo!" fi
```

Como en otros lenguajes de programación, puede usar una declaración de caso para simplificar condicionales complejos cuando hay varias opciones diferentes. Entonces, en lugar de usar algunas declaraciones if y if-else , podría usar una declaración de un solo caso .

La sintaxis de la declaración del caso Bash se ve así:

```
case $alguna_variable en

    patrón_1)
        comandos
        ;;

    patrón_2| patrón_3)
        comandos
        ;;

    *)
       comandos predeterminados
        ;;
esac
```

Un resumen rápido de la estructura:

- Todas las declaraciones de casos comienzan con la palabra clave case .
- Al igual que la palabra clave case , debe especificar una variable o una expresión seguida de la palabra clave in .
- Después de eso, tiene sus patrones de casos , donde necesita usar) para identificar el final del patrón.
- Puede especificar varios patrones divididos por una tubería: |.
- Después del patrón, especifica los comandos que desea que se ejecuten en caso de que el patrón coincida con la variable o la expresión que ha especificado.

- Todas las cláusulas deben terminarse agregando ;; al final.
- Puede tener una declaración predeterminada * como el patrón.
- agregando una Para cerrar la declaración del caso , use la palabra clave esac (caso escrito al revés).

Aquí hay un ejemplo de una declaración de caso Bash :

```
#!/bin/bash

read -p "Ingrese el nombre de la marca de su auto: " coche

caso $ coche en

    Tesla)
        echo -n "La fábrica de automóviles de ${car} está en EE. UU."
        ;;

    BMW | Mercedes | Audi | Porsche) echo
        -n "La fábrica de automóviles de ${car} está en Alemania".
        ;;

    Toyota | Mazda | Mitsubishi | Subaru)
        echo -n "La fábrica de automóviles de ${car} está en Japón".
        ;;

    *)
       echo -n "${car} es una marca de coche desconocida"
        ;;

esac
```

Con este script, le pedimos al usuario que ingrese el nombre de una marca de automóvil como Telsa, BMW, Mercedes, etc.

Luego, con una declaración del caso , verificamos el nombre de la marca y si coincide con alguno de nuestros patrones, y si es así, imprimimos la ubicación de la fábrica.

Si el nombre de la marca no coincide con ninguna de las declaraciones de nuestro caso , imprimimos un mensaje predeterminado: una marca de automóvil desconocida.

¡Te aconsejaría que pruebes y modifiques el guión y juegues un poco con él para que puedas practicar lo que acabas de aprender en los últimos dos capítulos!

Para obtener más ejemplos de declaraciones de casos de Bash , asegúrese de consultar el capítulo 16, donde crearíamos un menú interactivo en Bash usando una declaración de casos para procesar la entrada del usuario.

Como con cualquier otro idioma, los bucles son muy convenientes. Con Bash puede usar bucles for , while y till .

Aquí está la estructura de un bucle for:

```
para var en ${lista} hacer

        tus_comandos
hechos
```

Ejemplo:

```
#!/bin/bash

usuarios="devdojo bobby tony"

para usuario en ${usuarios}
hacer
        echo "${usuario}"
hecho
```

Un resumen rápido del ejemplo:

- Primero, especificamos una lista de usuarios y almacenamos el valor en una variable llamada $usuarios.
- Después de eso, comenzamos nuestro bucle for con la palabra clave for
- Luego definimos una nueva variable que representaría cada elemento de la lista que damos. En nuestro caso, definimos una variable llamada usuario, que representaría a cada usuario de la variable $usuarios .
- Luego especificamos la palabra clave in seguida de nuestra lista que recorreremos En la siguiente
- línea, usamos la palabra clave do , que indica lo que haremos para cada iteración del ciclo Luego especificamos los comandos que queremos ejecutar Finalmente, cerramos el ciclo con la palabra
- clave done
-

También puede usar for para procesar una serie de números. Por ejemplo, aquí hay una forma de pasar del 1 al 10:

```
#!/bin/bash

para num en {1..10} hacer

        echo ${num}
hecho
```

La estructura de un bucle while es bastante similar a la del bucle for :

```
mientras [tu_condición] hacer

        tus_condiciones
hechas
```

Aquí hay un ejemplo de un ciclo while :

```
#!/bin/bash

contador=1
mientras que [[ $contador -le 10 ]]
hacer
        echo $contador
        ((contador++))
hecho
```

Primero, especificamos una variable de contador y la configuramos en 1, luego dentro del ciclo, agregamos el contador usando esta declaración aquí: ((contador++)). De esa manera, nos aseguramos de que el ciclo se ejecute solo 10 veces y no se ejecute para siempre. El ciclo se completará tan pronto como el contador llegue a 10, ya que esto es lo que hemos establecido como condición: while [[$counter -le 10]].

Vamos a crear un script que le pida al usuario su nombre y no permita una entrada vacía:

```
#!/bin/bash

read -p "¿Cuál es tu nombre?" nombre

mientras que [[ -z ${nombre} ]]
hacer
        echo "Su nombre no puede estar en blanco. ¡Por favor ingrese un nombre válido!"
read -p "¿Ingresa tu nombre de nuevo?" hecho
                                                nombre



echo "Hola ${nombre}"
```

Ahora, si ejecuta lo anterior y simplemente presiona enter sin proporcionar información, el bucle se ejecutará nuevamente y le pedirá su nombre una y otra vez hasta que realmente proporcione alguna información.

La diferencia entre los bucles till y while es que el bucle till ejecutará los comandos dentro del bucle hasta que la condición se vuelva verdadera .

Estructura:

```
hasta que [tu_condición] lo haga

    tus_comandos
hechos
```

Ejemplo:

```
#!/bin/bash

cuenta=1
hasta que [ $cuenta -gt 10 ] haga

    echo $contar
    ((contar++))
hecho
```

Al igual que con otros idiomas, también puede usar continuar y romper con sus scripts bash:

- continue le dice a su script bash que detenga la iteración actual del bucle y comience la siguiente iteración.

La sintaxis de la sentencia continue es la siguiente:

```
continuar [n]
```

El argumento [n] es opcional y puede ser mayor o igual que 1. Cuando se proporciona [n], se reanuda el ciclo envolvente n-ésimo. continuar 1 es equivalente a continuar.

```
#!/bin/bash

para yo en 1 2 3 4 5 hacer

        if [ $i –eq 2 ] luego
        echo “omitiendo el
                número 2” continuar fi echo “I es igual
        a $i” hecho
```

También podemos usar el comando continuar de manera similar al comando romper para controlar múltiples bucles.

- break le dice a su script bash que finalice el ciclo de inmediato.

La sintaxis de la instrucción break toma la siguiente forma:

```
romper [n]
```

[n] es un argumento opcional y debe ser mayor o igual que 1. Cuando se proporciona [n], se sale del ciclo envolvente n-ésimo. break 1 es equivalente a break.

Ejemplo:

```
#!/bin/bash

num=1
mientras que [ $num –lt 10 ]
hacer
        si [ $num –eq 5 ]
        entonces rompa

        fi
        ((num++))
hecho echo
"Bucle completado"
```

También podemos usar el comando de interrupción con múltiples bucles. Si queremos salir del ciclo de trabajo actual, ya sea interno o externo, simplemente usamos la pausa, pero si estamos en el ciclo interno y queremos salir del ciclo externo, usamos la pausa 2.

Ejemplo:

```
#!/bin/bash

para (( a = 1; a < 10; a++ )) hacer

        echo “bucle externo: $a” for
        (( b = 1; b < 100; b++ )) do

                si [ $b –gt 5 ]
                entonces romper 2

                fi
        echo “Bucle interno: $b ”
        hecho hecho
```

El script bash comenzará con a=1 y se moverá al ciclo interno y cuando llegue a b=5, romperá el ciclo externo. Podemos usar break only en lugar de break 2, para romper el bucle interno y ver cómo afecta la salida.

Las funciones son una excelente manera de reutilizar el código. La estructura de una función en bash es bastante similar a la mayoría de los lenguajes:

```
function nombre_función ()
        { tus_comandos
}
```

También puede omitir la palabra clave de función al principio, lo que también funcionaría:

```
nombre_función ()
        { tus_comandos
}
```

Prefiero ponerlo allí para una mejor legibilidad. Pero es una cuestión de preferencia personal.

Ejemplo de "¡Hola mundo!" función:

```
#!/bin/bash

función hola(){ echo
        "¡Función Hola mundo!"
}

Hola
```

Aviso: una cosa a tener en cuenta es que no debe agregar el paréntesis cuando llama a la función.

Pasar argumentos a una función funciona de la misma manera que pasar argumentos a un script:

```
#!/bin/bash

función hola(){ echo
        "¡Hola $1!"
}

hola DevDojo
```

Las funciones deben tener comentarios que mencionen la descripción, las variables globales, los argumentos, los resultados y los valores devueltos, si corresponde.

```
######################################## #
Descripción: Función de saludo # Globales: # Ninguno
# Argumentos: # Argumento de entrada único # Salidas:
# Valor del argumento de entrada # Devuelve: # 0 si
tiene éxito, distinto de cero en caso de error.
######################################## función
hola(){ echo "Hola $1!"




}
```

¡En los próximos capítulos usaremos muchas funciones!

Para depurar sus scripts bash, puede usar -x al ejecutar sus scripts:

```
bash -x ./tu_script.sh
```

O puede agregar set -x antes de la línea específica que desea depurar, set -x habilita un modo del shell donde todos los comandos ejecutados se imprimen en la terminal.

Otra forma de probar sus scripts es usar esta fantástica herramienta aquí:

https://www.shellcheck.net/

Simplemente copie y pegue su código en el cuadro de texto, y la herramienta le dará algunas sugerencias sobre cómo puede mejorar su secuencia de comandos.

También puede ejecutar la herramienta directamente en su terminal:

https://github.com/koalaman/shellcheck

Si te gusta la herramienta, ¡asegúrate de destacarla en GitHub y contribuir!

Como SysAdmin/DevOps, paso gran parte del día en la terminal. Aquí están mis atajos favoritos que me ayudan a hacer tareas más rápido mientras escribo scripts de Bash o simplemente mientras trabajo en la terminal.

Los dos siguientes son particularmente útiles si tiene un comando muy largo.

- Elimine todo, desde el cursor hasta el final de la línea:

```
Ctrl + k
```

- Elimine todo, desde el cursor hasta el comienzo de la línea:

```
Ctrl + tu
```

- Eliminar una palabra hacia atrás desde el cursor:

```
Control + w
```

- Busca en tu historial hacia atrás. Este es probablemente el que más uso. Es realmente útil y acelera mucho mi flujo de trabajo:

```
Ctrl + r
```

- Borre la pantalla, uso esto en lugar de escribir el comando borrar :

```
Ctrl + l
```

- Detiene la salida a la pantalla:

```
Ctrl + s
```

- Habilitar la salida a pantalla en caso de que se detuviera previamente con Ctrl + s:

```
Ctrl + q
```

- Terminar el comando actual

```
Ctrl + c
```

- Lanza el comando actual al fondo:

> Ctrl + z

Los uso regularmente todos los días, y me ahorra mucho tiempo.

Si cree que me he perdido alguno, no dude en unirse a la discusión en el foro de la comunidad de DigitalOcean .

Como desarrollador o administrador del sistema, es posible que deba pasar mucho tiempo en su terminal. Siempre trato de buscar formas de optimizar las tareas repetitivas.

Una forma de hacerlo es escribir scripts bash cortos o crear comandos personalizados, también conocidos como alias. Por ejemplo, en lugar de escribir un comando muy largo cada vez, podría crear un atajo para él.

Comencemos con el siguiente escenario, como administrador del sistema, es posible que deba verificar las conexiones a su servidor web con bastante frecuencia, por lo que usaré el comando netstat como ejemplo.

Lo que normalmente haría cuando accedo a un servidor que tiene problemas con las conexiones al puerto 80 o 443 es verificar si hay algún servicio escuchando en esos puertos y la cantidad de conexiones a los puertos.

El siguiente comando netstat nos mostraría cuántas conexiones TCP en los puertos 80 y 443 tenemos actualmente:

```
planta netstat | grep'80 \|443' | grep -v ESCUCHAR | wc-l
```

Este es un comando bastante largo, por lo que escribirlo cada vez puede llevar mucho tiempo a largo plazo, especialmente cuando desea obtener esa información rápidamente.

Para evitar eso, podemos crear un alias, así que en lugar de escribir el comando completo, podríamos escribir un comando corto en su lugar. Por ejemplo, digamos que queríamos poder escribir conn (abreviatura de conexiones) y obtener la misma información. Todo lo que tenemos que hacer en este caso es ejecutar el siguiente comando:

```
alias conn="netstat -plant | grep '80\|443' | grep -v ESCUCHAR | wc -l"
```

De esa manera estamos creando un alias llamado conn que sería esencialmente un 'atajo' para nuestro comando netstat largo. Ahora, si ejecuta solo conn:

```
control
```

Obtendría el mismo resultado que el comando netstat largo. Puede ser aún más creativo y agregar algunos mensajes de información como este aquí:

```
alias conn="echo 'Total de conexiones en los puertos 80 y 443:' ; netstat
-plant | grep '80\|443' | grep -v LISTEN | wc -l"
```

Ahora, si ejecuta conn , obtendrá el siguiente resultado:

```
Conexiones totales en puerto 80 y 443: 12
```

Ahora, si cierra sesión y vuelve a iniciarla, su alias se perderá. En el siguiente paso, verá cómo hacer que esto sea persistente.

Para que el cambio sea persistente, debemos agregar el comando alias en nuestro archivo de perfil de shell.

De forma predeterminada en Ubuntu, este sería el archivo ~/.bashrc , para otros sistemas operativos, podría ser ~/.bash_perfil. Con tu editor de texto favorito abre el archivo:

```
nano ~/.bashrc
```

Ve al final y agrega lo siguiente:

```
alias conn="echo 'Total de conexiones en los puertos 80 y 443:' ; netstat
-plant | grep '80\|443' | grep -v LISTEN | wc -l"
```

Guardar y luego salir.

De esa manera, ahora, incluso si cierra sesión y vuelve a iniciar sesión, su cambio se mantendrá y podrá ejecutar su comando bash personalizado.

Para enumerar todos los alias disponibles para su shell actual, solo tiene que ejecutar el siguiente comando:

```
alias
```

Esto sería útil en caso de que vea algún comportamiento extraño con algunos comandos.

Esta es una forma de crear comandos bash personalizados o alias bash.

Por supuesto, podría escribir un script de bash y agregar el script dentro de su carpeta /usr/bin , pero esto no funcionaría si no tiene acceso de root o sudo, mientras que con los alias puede hacerlo sin necesidad de root. acceso.

> **Aviso:** esto se publicó inicialmente en DevDojo.com

¡Intentemos juntar lo que hemos aprendido hasta ahora y crear nuestro primer script Bash!

Como ejemplo, escribiremos un script que recopilará información útil sobre nuestro servidor como:

- Uso actual del disco
- Uso actual de la CPU
- Uso actual de RAM
- Verifique la versión exacta del Kernel

Siéntase libre de ajustar la secuencia de comandos agregando o eliminando funciones para que se ajuste a sus necesidades.

Lo primero que debe hacer es crear un nuevo archivo con una extensión .sh . Crearé un archivo llamado status.sh ya que el script que crearemos nos dará el estado de nuestro servidor.

Una vez que haya creado el archivo, ábralo con su editor de texto favorito.

Como aprendimos en el capítulo 1, en la primera línea de nuestro script Bash necesitamos especificar el llamado Shebang:

```
#!/bin/bash
```

Todo lo que hace Shebang es indicar al sistema operativo que ejecute el script con el ejecutable /bin/bash.

A continuación, como se discutió en el capítulo 6, comencemos agregando algunos comentarios para que las personas puedan descubrir fácilmente para qué se usa el script. Para hacer eso justo después del shebang, solo puede agregar lo siguiente:

```
#!/bin/bash

# Script que devuelve el estado actual del servidor
```

Luego, sigamos adelante y apliquemos lo que aprendimos en el capítulo 4 y agreguemos algunas variables que podríamos querer usar a lo largo del script.

Para asignar un valor a una variable en bash, solo tienes que usar el signo = . Por ejemplo, almacenemos el nombre de host de nuestro servidor en una variable para que podamos usarlo más tarde:

```
server_name=$(nombre de host)
```

Al usar $() le decimos a bash que realmente interprete el comando y luego asigne el valor a nuestra variable.

Ahora, si hiciéramos eco de la variable, veríamos el nombre de host actual:

```
echo $nombre_servidor
```

Como ya sabrás después de leer el capítulo 12, para crear una función en bash necesitas usar la siguiente estructura:

```
function nombre_función ()
        { tus_comandos
}
```

Vamos a crear una función que devuelva el uso de memoria actual en nuestro servidor:

```
función memory_check() { echo
        ""
                echo "El uso de memoria actual en ${server_name} es: " free -h echo ""


}
```

Desglose rápido de la función:

- function memory_check() { - así es como definimos la función echo "" - aquí solo
- imprimimos una nueva línea echo "El uso actual de la memoria en ${server_name} es: " aquí imprimimos todo un pequeño mensaje y la variable -
- $server_name } - finalmente así es como cerramos la función
-

Luego, una vez que se ha definido la función, para llamarla, simplemente use el nombre de la función:

```
# Definir la función function
memory_check() { echo ""

        echo "El uso de memoria actual en ${server_name} es: " free -h echo ""


}

# Llamar a la función
memory_check
```

Antes de revisar la solución, lo desafiaría a usar la función de arriba y escribir algunas funciones usted mismo.

Las funciones deben hacer lo siguiente:

- Uso actual del disco
- Uso actual de la CPU
- Uso actual de RAM
- Verifique la versión exacta del Kernel

Siéntase libre de usar Google si no está seguro de qué comandos necesita usar para obtener esa información.

Una vez que esté listo, siéntase libre de desplazarse hacia abajo y ver cómo lo hemos hecho y comparar los resultados.

¡Tenga en cuenta que hay varias formas correctas de hacerlo!

Así es como se vería el resultado final:

```
#!/bin/bash

##
# Script BASH que verifica:
# - Uso de memoria
# - Carga de la CPU
# - Número de conexiones TCP
# - Versión del núcleo ##


server_name=$(nombre de host)

función memory_check() { echo
        ""
                echo "El uso de la memoria en ${server_name} es: "
                free -h echo ""

}

función cpu_check() { eco ""

                echo "La carga de la CPU en ${server_name} es:
        " echo "" uptime echo ""


}

función tcp_check() { eco ""

                echo "Conexiones TCP en ${server_name}: " echo ""

                gato /proc/net/tcp | wc -l eco ""

}

función kernel_check() { eco ""

                echo "La versión del kernel en ${server_name} es: " echo
                ""
```

```
        uname -r
    eco ""
}

función all_checks()
        { memory_check
        cpu_check
        tcp_check
        kernel_check
}

todos_cheques
```

¡Las secuencias de comandos de Bash son increíbles! No importa si es un ingeniero, desarrollador o simplemente un entusiasta de Linux DevOps/SysOps, puede usar scripts de Bash para combinar diferentes comandos de Linux y automatizar tareas diarias aburridas y repetitivas, para que pueda concentrarse en cosas más productivas y divertidas.

> **Aviso:** esto se publicó inicialmente en DevDojo.com

En este tutorial, le mostraré cómo crear un menú de opción múltiple en Bash para que sus usuarios puedan elegir entre qué acción ejecutar.

Volveríamos a utilizar parte del código del capítulo anterior, así que si aún no lo ha leído, asegúrese de hacerlo.

Comencemos de nuevo repasando la funcionalidad principal del script:

- Comprueba el uso actual del disco
- Comprueba el uso actual de la CPU
- Comprueba el uso actual de RAM
- Comprueba la versión exacta del Kernel

En caso de que no lo tenga a mano, aquí está el script en sí:

```
#!/bin/bash

##
# Script de menú BASH que comprueba:
# - Uso de memoria
# - Carga de la CPU
# - Número de conexiones TCP
# - Versión del núcleo ##


server_name=$(nombre de host)

función memory_check() { echo
        ""
                echo "El uso de la memoria en ${server_name} es: "
                free -h echo ""

}

función cpu_check() { eco ""

                echo "La carga de la CPU en ${server_name} es:
        " echo "" uptime echo ""


}

función tcp_check() { eco ""

                echo "Conexiones TCP en ${server_name}: " echo ""

                gato /proc/net/tcp | wc-l
```

```
        eco ""
}

función kernel_check() { eco ""

                echo "La versión del kernel en ${server_name} es: " echo
                ""
                uname -r
        eco ""
}

función all_checks()
                { memory_check
                cpu_check
                tcp_check
                kernel_check
}
```

Luego construiremos un menú que le permita al usuario elegir qué función ejecutar.

Por supuesto, puede ajustar la función o agregar otras nuevas según sus necesidades.

Para que el menú sea un poco más 'legible' y fácil de comprender a primera vista, agregaremos algunas funciones de color.

Al comienzo de su secuencia de comandos, agregue las siguientes funciones de color:

```
##
# Variables de color
##
verde='\e[32m'
azul='\e[34m'
claro='\e[0m'

##
# Funciones de color
##

ColorVerde(){ echo
        -ne $verde$1$claro
}
ColorAzul(){ echo
        -ne $azul$1$claro
}
```

Puede utilizar las funciones de color de la siguiente manera:

```
echo -ne $(ColorBlue 'Algo de texto aquí')
```

¡ Lo anterior generaría la cadena Some text here y sería azul!

Finalmente, para agregar nuestro menú, crearemos una función separada con un cambio de mayúsculas y minúsculas para nuestras opciones de menú:

```
menu()
{ echo -ne "
Mi primer menú
$(ColorGreen '1)') Uso de memoria $
(ColorGreen '2)') Carga de CPU $
(ColorGreen '3)') Número de conexiones TCP $(ColorGreen
'4)') Versión del kernel $(ColorGreen '5)') Marcar todo $
(ColorGreen '0)') Salir $(ColorBlue 'Elija una opción:') "
leer un caso $a en 1) memory_check ; menu ;; 2)
cpu_check ; menu ;; 3) tcp_check ; menu ;; 4) kernel_check ;
        menu ;; 5) all_checks ; menu ;; 0) exit 0 ;; *)
        echo -e $red"Opción incorrecta."$clear;

Comando Incorrecto;;
        esac
}
```

Primero solo hacemos eco de las opciones del menú con algo de color:

```
                "
eco-ne
Mi primer menú
$(ColorGreen '1)') Uso de memoria
$(ColorGreen '2)') Carga de la CPU
$(ColorGreen '3)') Número de conexiones TCP
$(ColorGreen '4)') Versión del núcleo
$(ColorVerde '5)') Marcar todo
$(ColorVerde '0)') Salir
$(ColorBlue 'Elige una opción:') "
```

Luego leemos la respuesta del usuario y la almacenamos en una variable llamada $a:

```
            leer un
```

Finalmente, tenemos un caso de interruptor que activa una función diferente según el valor de $a:

```
            case $a en 1)
                    memory_check ; menú ;; 2)
                    control_cpu; menú ;; 3) tcp_control;
                    menú ;; 4) comprobación_del_núcleo;
                    menú ;; 5) todos_cheques ; menú ;;
                    0) salir 0 ;; *) echo -e $red"Opción
                    incorrecta."$clear;

Comando Incorrecto;;
            esac
```

Al final, debemos llamar a la función de menú para imprimir el menú:

```
# Llamar a la función del menú
menú
```

Al final, su script se verá así:

```
#!/bin/bash

##
# Script de menú BASH que comprueba:
# - Uso de memoria
# - Carga de la CPU
# - Número de conexiones TCP
# - Versión del núcleo ##


server_name=$(nombre de host)

función memory_check() { echo
        ""
                echo "El uso de la memoria en ${server_name} es: "
                free -h echo ""

}

función cpu_check() { eco ""

                echo "La carga de la CPU en ${server_name} es:
        " echo "" uptime echo ""


}

función tcp_check() { eco ""

                echo "Conexiones TCP en ${server_name}: " echo ""

                gato /proc/net/tcp | wc -l eco ""

}

función kernel_check() { eco ""

                echo "La versión del kernel en ${server_name} es: " echo
                ""
```

```
                uname -r
        eco ""
}

función all_checks()
                { memory_check
                cpu_check tcp_check
                kernel_check

}

##
# Variables de color
##
verde='\e[32m'
azul='\e[34m'
claro='\e[0m'

##
# Funciones de color
##

ColorVerde(){ echo
                -ne $verde$1$claro
}
ColorAzul(){ echo
                -ne $azul$1$claro
}

menú()
{ echo -ne      "
Mi primer menú
$(ColorGreen '1)') Uso de memoria
$(ColorGreen '2)') Carga de la CPU
$(ColorGreen '3)') Número de conexiones TCP
$(ColorGreen '4)') Versión del núcleo
$(ColorVerde '5)') Marcar Todo
$(ColorVerde '0)') Salir
$(ColorBlue 'Elige una opción:') " leer un

                case $a en 1)
                                memory_check ; menú ;; 2)
                                control_cpu; menú ;; 3) tcp_control;
                                menú ;; 4) comprobación_del_núcleo;
                                menú ;; 5) todos_cheques ; menú ;;
```

```
                0) salir 0 ;; *)
                echo -e $red"Opción incorrecta."$clear;
Comando Incorrecto;;
        esac
}

# Llamar a la función del menú
menú
```

Para probar el script, cree un nuevo archivo con una extensión .sh , por ejemplo: menu.sh y luego ejecútelo:

```
menú bash.sh
```

La salida que obtendrías se verá así:

```
Mi primer menú
1) Uso de memoria
2) carga de la CPU
3) Número de conexiones TCP
4) Versión del núcleo
5) Marcar todo
0) Salir
Escoge una opción:
```

Podrá elegir una opción diferente de la lista y cada número llamará a una función diferente del script:

¡Ahora sabe cómo crear un menú Bash e implementarlo en sus scripts para que los usuarios puedan seleccionar diferentes valores!

> **Aviso:** este contenido se publicó inicialmente en DevDojo.com

Cualquier comando que pueda ejecutar desde la línea de comando se puede usar en un script bash.
Los scripts se utilizan para ejecutar una serie de comandos. Bash está disponible de forma predeterminada en los sistemas operativos Linux y macOS.

Tengamos un escenario hipotético en el que necesita ejecutar una secuencia de comandos BASH en varios servidores remotos, pero no desea copiar manualmente la secuencia de comandos en cada servidor, luego iniciar sesión nuevamente en cada servidor individualmente y solo luego ejecutar la secuencia de comandos.

Por supuesto, podría usar una herramienta como Ansible, ¡pero aprendamos cómo hacerlo con Bash!

Para este ejemplo, usaré 3 servidores Ubuntu remotos implementados en DigitalOcean. Si aún no tiene una cuenta de Digital Ocean, puede registrarse en DigitalOcean y obtener $ 100 de crédito gratis a través de este enlace de referencia aquí:

https://m.do.co/c/2a9bba940f39

Una vez que tenga lista su cuenta de Digital Ocean, continúe e implemente 3 gotas.

Seguí adelante y creé 3 servidores Ubuntu:

Pondré las IP de esos servidores en un archivo servers.txt que usaría para hacer un bucle con nuestro script Bash.

Si es nuevo en DigitalOcean, puede seguir los pasos sobre cómo crear un Droplet aquí:

- Cómo crear una gota desde el panel de control de DigitalOcean

También puede seguir los pasos de este video aquí sobre cómo hacer su configuración inicial del servidor:

- Cómo hacer la configuración inicial del servidor con Ubuntu

O incluso mejor, puede seguir este artículo aquí sobre cómo automatizar la configuración inicial de su servidor con Bash:

Automatización de la configuración inicial del servidor con Ubuntu 18.04 con Bash

Con los 3 nuevos servidores en su lugar, podemos continuar y concentrarnos en ejecutar nuestro script Bash en todos ellos con un solo comando.

Reutilizaré el script de demostración del capítulo anterior con algunos cambios leves. Simplemente ejecuta algunas comprobaciones como el uso actual de la memoria, el uso actual de la CPU, la cantidad de conexiones TCP y la versión del kernel.

```
#!/bin/bash

##
# Script BASH que verifica lo siguiente:
# - Uso de memoria
# - Carga de la CPU
# - Número de conexiones TCP
# - Versión del núcleo
##

##
# Comprobación de
memoria ##
server_name=$(nombre de host)

función memory_check() {
        eco "########"
                echo "El uso de memoria actual en ${server_name} es: " free -h echo
                "########"


}


función cpu_check() { echo
        "########"
                echo "La carga actual de la CPU en ${server_name} es: " echo ""
        uptime echo "########"



}

función tcp_check() { echo
        "########"
                echo "Total de conexiones TCP en ${server_name}: " echo ""

                gato /proc/net/tcp | wc -l echo
        "########"
```

```
}

función kernel_check() {
        eco "########"
                echo "La versión exacta del Kernel en ${server_name} es: " echo ""

                uname -r
        eco "########"
}

función all_checks()
                { memory_check
                cpu_check
                tcp_check
                kernel_check
}

todos_cheques
```

Copie el código a continuación y agréguelo en un archivo llamado remote_check.sh. También puede obtener el script desde aquí.

Ahora que tenemos la secuencia de comandos y los servidores listos y que hemos agregado esos servidores en nuestro archivo servers.txt, podemos ejecutar el siguiente comando para recorrer todos los servidores y ejecutar la secuencia de comandos de forma remota sin tener que copiar la secuencia de comandos en cada servidor y conectarse individualmente a cada servidor.

```
para servidor en $(servidores cat.txt) ; haz ssh tu_usuario@${servidor} 'bash -s'
< ./remote_check.sh ; hecho
```

Lo que hace este bucle for es pasar por cada servidor en el archivo servers.txt y luego ejecuta el siguiente comando para cada elemento de la lista:

```
ssh su_usuario@la_ip_del_servidor 'bash -s' < ./remote_check.sh
```

Obtendrías el siguiente resultado:

Este es solo un ejemplo realmente simple de cómo ejecutar un script simple en varios servidores sin tener que copiar el script en cada servidor y sin tener que acceder a los servidores individualmente.

Por supuesto, podría ejecutar un script mucho más complejo y en muchos más servidores.

Si está interesado en la automatización, le recomendaría consultar la página de recursos de Ansible en el sitio web de DigitalOcean:

Recursos de Ansible

> **Aviso:** este contenido se publicó inicialmente en DevDojo

La herramienta de línea de comandos jq es un procesador **JSON** de línea de comandos ligero y flexible. Es excelente para analizar la salida JSON en BASH.

Una de las mejores cosas de jq es que está escrito en C portátil y no tiene dependencias de tiempo de ejecución. Todo lo que necesita hacer es descargar un solo binario o usar un administrador de paquetes como apt e instalarlo con un solo comando.

Para la demostración en este tutorial, usaría una API REST externa que devuelve un simple
Salida JSON llamada QuizAPI:

https://quizapi.io/

Si desea seguir, asegúrese de obtener una clave de API gratuita aquí:

https://quizapi.io/clientarea/settings/token

QuizAPI es gratuito para los desarrolladores.

Hay muchas formas de instalar jq en su sistema. Una de las formas más sencillas de hacerlo es usar el administrador de paquetes según su sistema operativo.

Aquí hay una lista de los comandos que necesitará usar dependiendo de su sistema operativo:

- Instale jq en Ubuntu/Debian:

```
sudo apt-get install jq
```

- Instale jq en Fedora:

```
sudo dnf instalar jq
```

- Instale jq en openSUSE:

```
sudo zypper instalar jq
```

- Instale jq en Arch:

```
sudo pacman -S jq
```

- Instalación en Mac con Homebrew:

```
cerveza instalar jq
```

- Instalar en Mac con MacPort:

```
puerto instalar jq
```

Si está utilizando otro sistema operativo, le recomendaría echar un vistazo al oficial

documentación aquí para más información:

https://stedolan.github.io/jq/download/

Una vez que haya instalado jq, puede verificar su versión actual ejecutando este comando:

```
jq --versión
```

Una vez que haya instalado jq y su clave API QuizAPI, puede analizar la salida JSON de QuizAPI directamente en su terminal.

Primero, cree una variable que almacene su clave API:

```
API_KEY=TU_API_KEY_AQUÍ
```

Para obtener algún resultado de uno de los puntos finales de QuizAPI, puede usar el comando curl:

```
rizo
"https://quizapi.io/api/v1/questions?apiKey=${API_KEY}&limit=1 0"
```

Para obtener un resultado más específico, puede usar el generador de URL de QuizAPI aquí:

https://quizapi.io/api-config

Después de ejecutar el comando curl, el resultado que obtendría se vería así:

Esto podría ser bastante difícil de leer, pero gracias a la herramienta de línea de comandos jq, todo lo que tenemos que hacer es canalizar el comando curl a jq y veríamos una salida JSON bien formateada:

```
curl
"https://quizapi.io/api/v1/questions?apiKey=${API_KEY}&limit=1 0" | jq
```

Tenga en cuenta el | jq al final.

En este caso, la salida que obtendría sería algo como esto:

Ahora, ¡esto se ve mucho mejor! La herramienta de línea de comandos jq formateó la salida para nosotros y

agregó un poco de color agradable!

Digamos que solo queríamos obtener el primer elemento de la salida JSON, para hacer eso solo tenemos que especificar el índice que queremos ver con la siguiente sintaxis:

```
jq.[0]
```

Ahora, si ejecutamos el comando curl nuevamente y canalizamos la salida a jq .[0] así:

```
curl
"https://quizapi.io/api/v1/questions?apiKey=${API_KEY}&limit=1 0" | jq.[0]
```

Solo obtendrá el primer elemento y la salida se verá así:

A veces, es posible que desee obtener solo el valor de una clave específica, digamos en nuestro ejemplo, QuizAPI devuelve una lista de preguntas junto con las respuestas, la descripción, etc., pero ¿qué sucede si desea obtener las preguntas solo sin la información adicional? ?

Esto va a ser bastante sencillo con jq, todo lo que necesita hacer es agregar la clave después del comando jq, por lo que se vería así:

```
jq .[].pregunta
```

Tenemos que agregar .[] ya que QuizAPI devuelve una matriz y al especificar .[] le decimos a jq que queremos obtener el valor de .question para todos los elementos de la matriz.

La salida que obtendría se vería así:

Como puede ver, ahora solo obtenemos las preguntas sin el resto de los valores.

Avancemos y creemos un pequeño script bash que debería generar la siguiente información para nosotros:

- Obtenga solo la primera pregunta de la salida
- Obtenga todas las respuestas para esa pregunta
- Asignar las respuestas a las variables
- Imprime la pregunta y las respuestas.
- Para hacer eso, he creado el siguiente script:

> **Aviso:** asegúrese de cambiar la parte API_KEY con su clave QuizAPI real:

```
#!/bin/bash

##
# Realice una llamada API a QuizAPI y almacene el resultado en una variable ##


salida=$(curl 'https://
 quizapi.io/api/v1/questions?apiKey=API_KEY&limit=10' 2>/dev/null)


##
# Obtener solo la primera pregunta ##

salida=$(echo $salida | jq .[0])

##
# Obtener la pregunta ##

pregunta=$(echo $salida | jq .pregunta)

##
# Obtén las respuestas ##


answer_a=$(echo $salida | jq .answers.answer_a) answer_b=$(echo $salida
| jq .answers.answer_b) answer_c=$(echo $salida | jq .answers.answer_c)
answer_d=$(echo $salida | jq .respuestas.answer_d)


##
# Salida de la pregunta ##


eco "
Pregunta: ${pregunta}

A) ${respuesta_a}
B) ${respuesta_b}
C) ${respuesta_c}
D) ${respuesta_d}
"
```

Si ejecuta el script, obtendrá el siguiente resultado:

Incluso podemos ir más allá al hacer que esto sea interactivo para que podamos elegir la respuesta directamente en nuestra terminal.

Ya existe un script bash que hace esto usando QuizAPI y jq:

Puedes echarle un vistazo a ese script aquí:

- https://github.com/QuizApi/QuizAPI-BASH/blob/master/quiz.sh

La herramienta de línea de comandos jq es una herramienta increíble que le brinda el poder de trabajar con JSON directamente en su terminal BASH.

De esa manera, puede interactuar fácilmente con todo tipo de API REST diferentes con BASH.

Para obtener más información, puede consultar la documentación oficial aquí:

- https://stedolan.github.io/jq/manual/

Y para obtener más información sobre **QuizAPI,** puede consultar la documentación oficial aquí:

- https://quizapi.io/docs/1.0/overview

> **Aviso:** este contenido se publicó inicialmente en DevDojo.com

Alojo todos mis sitios web en **DigitalOcean** Droplets y también uso Cloudflare como mi proveedor de CDN. Uno de los beneficios de usar Cloudflare es que reduce el tráfico general a su usuario y también oculta la dirección IP real de su servidor detrás de su CDN.

Mi característica personal favorita de Cloudflare es su protección DDoS gratuita. Ha salvado mis servidores varias veces de diferentes ataques DDoS. Tienen una API genial que puede usar para habilitar y deshabilitar su protección DDoS fácilmente.

¡Este capítulo va a ser un ejercicio! ¡Lo desafío a que siga adelante y escriba un breve script de bash que habilitaría y deshabilitaría la protección DDoS de Cloudflare para su servidor automáticamente si es necesario!

Antes de seguir esta guía aquí, configure su cuenta de Cloudflare y prepare su sitio web. Si no está seguro de cómo hacerlo, puede seguir estos pasos aquí: Cree una cuenta de Cloudflare y agregue un sitio web.

Una vez que tenga su cuenta de Cloudflare, asegúrese de obtener la siguiente información:

- Una cuenta de Cloudflare
- Clave API de Cloudflare
- ID de zona de Cloudflare

Además, asegúrese de que curl esté instalado en su servidor:

```
rizo --versión
```

Si curl no está instalado, debe ejecutar lo siguiente:

- Para RedHat/CentOS:

```
ñam instalar rizo
```

- Para Debian/Ubuntu

```
apt-get install curl
```

La secuencia de comandos debe monitorear el uso de la CPU en su servidor y, si el uso de la CPU aumenta según el número de vCPU, habilitaría la protección DDoS de Cloudflare automáticamente a través de la API de Cloudflare.

Las características principales del script deben ser:

- Comprueba la carga de la CPU del script en el servidor
- En caso de un pico de CPU, el script activa una llamada API a Cloudflare y habilita la función de protección DDoS para la zona especificada.
- Después de que la carga de la CPU vuelva a la normalidad, el script deshabilitará la opción "Estoy bajo ataque" y la restablecerá a la normalidad.

Ya he preparado un script de demostración que podrías usar como referencia. ¡Pero te animo a que intentes escribir el guión tú mismo primero y solo luego eches un vistazo a mi guión!

Para descargar el script simplemente ejecute el siguiente comando:

```
wget
https://raw.githubusercontent.com/bobbyiliev/cloudflare-ddos-protection/main/
protection.sh
```

Abra el script con su editor de texto favorito:

```
nano protección.sh
```

Y actualice los siguientes detalles con sus datos de Cloudflare:

```
CF_CONE_ID=SU_CF_ZONE_ID
CF_EMAIL_ADDRESS=SU_CF_EMAIL_ADDRESS
CF_API_KEY=TU_CF_API_KEY
```

Después de eso, haga que el script sea ejecutable:

```
chmod +x ~/protección.sh
```

Finalmente, configure 2 trabajos Cron para que se ejecuten cada 30 segundos. Para editar su ejecución de crontab:

```
crontab -e
```

Y agrega el siguiente contenido:

```
* * * * * /ruta-al-script/cloudflare/protection.sh * * * * * (dormir 30; /ruta-al-script/cloudflare/protection.sh)
```

Tenga en cuenta que debe cambiar la ruta al script con la ruta real donde almacenó el script.

Esta es una solución bastante sencilla y económica, una de las desventajas de la secuencia de comandos es que si su servidor no responde debido a un ataque, es posible que la secuencia de comandos no se active en absoluto.

Por supuesto, un mejor enfoque sería usar un sistema de monitoreo como Nagios y, en función de las estadísticas del sistema de monitoreo, puede activar el script, ¡pero este desafío de script podría ser una buena experiencia de aprendizaje!

Aquí hay otro gran recurso sobre cómo usar la API de Discord y enviar notificaciones a su canal de Discord con un script Bash:

Cómo usar Webhooks de Discord para recibir notificaciones sobre el estado de su sitio web en Ubuntu 18.04

> **Aviso:** este contenido se publicó inicialmente en DevDojo

Una de las primeras cosas que normalmente haría en caso de que note un uso elevado de la CPU en algunos de mis servidores Linux sería verificar la lista de procesos con top o htop y, en caso de que note muchos procesos de Apache o Nginx, revisaría rápidamente mis registros de acceso para determinar qué ha causado o está causando el pico de CPU en mi servidor o para averiguar si está sucediendo algo malicioso.

A veces, leer los registros puede ser bastante intimidante, ya que el registro puede ser enorme y revisarlo manualmente puede llevar mucho tiempo. Además, el formato de registro sin formato podría resultar confuso para las personas con menos experiencia.

Al igual que el capítulo anterior, ¡este capítulo será un desafío! Debe escribir un breve script de bash que resuma todo el registro de acceso sin la necesidad de instalar ningún software adicional.

Este script BASH necesita analizar y resumir sus registros de acceso y brindarle información muy útil como:

- Las 20 páginas principales con más solicitudes POST
- Las 20 páginas principales con más solicitudes GET
- Las 20 principales direcciones IP y su ubicación geográfica

Ya he preparado un script de demostración que podrías usar como referencia. ¡Pero te animo a que intentes escribir el guión tú mismo primero y solo luego eches un vistazo a mi guión!

Para descargar el script, puede clonar el repositorio con el siguiente comando:

```
clon de git
https://github.com/bobbyiliev/quick_access_logs_summary.git
```

O ejecute el siguiente comando que descargaría el script en su directorio actual:

```
wget
https://raw.githubusercontent.com/bobbyiliev/quick_access_logs _summary/
master/spike_check
```

La secuencia de comandos no realiza ningún cambio en su sistema, solo lee el contenido de su registro de acceso y lo resume, sin embargo, una vez que haya descargado el archivo, asegúrese de revisar el contenido usted mismo.

Todo lo que tiene que hacer una vez que se haya descargado el script es hacerlo ejecutable y ejecutarlo.

Para hacer eso, ejecute el siguiente comando para hacer que el script sea ejecutable:

```
chmod +x pico_verificar
```

Luego ejecute el script:

```
./spike_check /ruta/a/su/registro_de_acceso
```

Asegúrese de cambiar la ruta al archivo con la ruta real a su registro de acceso. Por ejemplo, si está utilizando Apache en un servidor Ubuntu, el comando exacto se vería así:

```
./spike_check /var/log/apache2/access.log
```

Si usa Nginx, el comando exacto sería casi el mismo, pero con la ruta al registro de acceso de Nginx:

```
./spike_check /var/log/nginx/access.log
```

Una vez que ejecute la secuencia de comandos, puede llevar un tiempo según el tamaño del registro.

La salida que vería debería verse así:

Esencialmente, lo que podemos decir en este caso es que hemos recibido 16 solicitudes POST a nuestro archivo xmlrpc.php, que los atacantes suelen utilizar para intentar explotar los sitios web de WordPress mediante el uso de varias combinaciones de nombre de usuario y contraseña.

En este caso específico, no fue un gran ataque de fuerza bruta, pero nos brinda una indicación temprana y podemos tomar medidas para evitar un ataque mayor en el futuro.

También podemos ver que había un par de direcciones IP rusas accediendo a nuestro sitio, por lo que en caso de que no espere ningún tráfico de Rusia, es posible que también desee bloquear esas direcciones IP.

Este es un ejemplo de un script BASH simple que le permite resumir rápidamente sus registros de acceso y determinar si está ocurriendo algo malicioso.

Por supuesto, es posible que también desee revisar manualmente los registros, ¡pero es un buen desafío intentar automatizar esto con Bash!

> **Aviso:** este contenido se publicó inicialmente en DevDojo

SSMTP es una herramienta que entrega correos electrónicos desde una computadora o un servidor a un host de correo configurado.

SSMTP no es un servidor de correo electrónico en sí mismo y no recibe correos electrónicos ni administra una cola.

Uno de sus usos principales es para reenviar correos electrónicos automatizados (como alertas del sistema) desde su máquina a una dirección de correo electrónico externa.

Necesitaría lo siguiente para poder completar este tutorial con éxito:

- Acceda a un servidor Ubuntu 18.04 como usuario no root con privilegios sudo y un firewall activo instalado en su servidor. Para configurarlos, consulte nuestra Guía de configuración inicial del servidor para Ubuntu 18.04

- Un servidor SMTP junto con el nombre de usuario y la contraseña de SMTP, esto también funcionaría con el servidor SMTP de Gmail, o podría configurar su propio servidor SMTP siguiendo los pasos de este tutorial en [https://www.digitalocean.com/ community/tutorials /how-to-install-and-configure postfix-as-a-send-only-smtp-server-on-ubuntu-16-04](Cómo instalar y configurar Postfix como un servidor SMTP de solo envío en Ubuntu 16.04)

Para instalar SSMTP, primero deberá actualizar su caché apt con:

```
sudo apt actualizar
```

Luego ejecute el siguiente comando para instalar SSMTP:

```
sudo apt instalar ssmtp
```

Otra cosa que necesitarías instalar es mailutils, para hacerlo ejecuta el siguiente comando:

```
sudo apt install mailutils
```

Ahora que tiene ssmtp instalado, para configurarlo para usar su servidor SMTP al enviar correos electrónicos, necesita editar el archivo de configuración de SSMTP.

Con su editor de texto favorito, abra el archivo /etc/ssmtp/ssmtp.conf :

```
sudo nano /etc/ssmtp/ssmtp.conf
```

Debe incluir la configuración de su SMTP:

```
root=postmaster
mailhub=<^>su_host_smtp.com<^>:587
hostname=<^>su_nombre_de_host<^>
AuthUser=<^>su_nombre_de_usuario_gmail@su_host_smtp.com<^>
AuthPass=<^>su_contraseña_gmail<^> FromLineOverride=YES

Usar STARTTLS=YES
```

Guarda el archivo y cierra.

Una vez que haya terminado su configuración, para enviar un correo electrónico simplemente ejecute el siguiente comando:

```
echo "<^>Aquí agrega el cuerpo de tu correo electrónico<^>" | mail -s "<^>Aquí
especifica el asunto de tu correo electrónico<^>"
<^>tu_receptor_correoelectrónico@tudominio.com<^>
```

Puede ejecutar esto directamente en su terminal o incluirlo en sus scripts bash.

Si necesita enviar archivos como archivos adjuntos, puede usar mpack.

Para instalar mpack , ejecute el siguiente comando:

```
sudo apt instalar mpack
```

A continuación, para enviar un correo electrónico con un archivo adjunto, ejecute el siguiente comando.

```
mpack -s "<^>Su Asunto aquí<^>" su_archivo.zip
<^>su_correo_electrónico@su_dominio.com<^>
```

El comando anterior enviaría un correo electrónico a

<^>your_recepient_email@yourdomain.com<^> con <^>your_file.zip<^> adjunto.

SSMTP es una forma excelente y confiable de implementar la funcionalidad de correo electrónico SMTP directamente en scripts bash.

Para obtener más información sobre SSMTP, recomendaría consultar la documentación oficial aquí.

> **Aviso:** este contenido se publicó inicialmente en el foro de la comunidad de DigitalOcean.

No es una situación poco común en la que deberá generar una contraseña aleatoria que puede usar para cualquier instalación de software o cuando se registra en cualquier sitio web.

Hay muchas opciones para lograr esto. Puede usar un administrador/bóveda de contraseñas donde a menudo tiene la opción de generar una contraseña al azar o usar un sitio web que puede generar la contraseña en su nombre.

También puede usar Bash en su terminal (línea de comando) para generar una contraseña que pueda usar rápidamente. Hay muchas maneras de lograrlo y me aseguraré de cubrir algunas de ellas y dejaré que usted elija qué opción es la más adecuada para sus necesidades.

Esta secuencia de comandos está destinada a practicar sus habilidades de secuencias de comandos bash. Puede divertirse mientras realiza proyectos simples con BASH, pero la seguridad no es una broma, así que asegúrese de no guardar sus contraseñas en texto sin formato en un archivo local ni escribirlas a mano en una hoja de papel.

Recomiendo encarecidamente a todos que utilicen proveedores seguros y confiables para generar y guardar las contraseñas.

Permítanme primero hacer un breve resumen de lo que va a hacer nuestro script:

1. Tendremos la opción de elegir la longitud de los caracteres de la contraseña cuando el script es ejecutado.
2. El script generará 5 contraseñas aleatorias con la longitud que se especificó en el paso 1

Necesitaría un terminal bash y un editor de texto. Puede usar cualquier editor de texto como vi, vim, nano o Visual Studio Code.

Estoy ejecutando el script localmente en mi computadora portátil Linux, pero si está usando una PC con Windows, puede acceder a cualquier servidor de su elección y ejecutar el script allí.

Aunque la secuencia de comandos es bastante simple, tener algunos conocimientos básicos de secuencias de comandos BASH lo ayudará a comprender mejor la secuencia de comandos y cómo funciona.

Uno de los grandes beneficios de Linux es que puedes hacer muchas cosas usando diferentes métodos. Cuando se trata de generar una cadena aleatoria de caracteres, tampoco es diferente.

Puede usar varios comandos para generar una cadena aleatoria de caracteres. Cubriré algunos de ellos y proporcionaré algunos ejemplos.

- Usando el comando de fecha . El comando de fecha generará la fecha y la hora actuales. Sin embargo, también manipulamos aún más la salida para usarla como contraseña generada aleatoriamente. Podemos codificar la fecha usando md5, sha o simplemente ejecutarla a través de base64. Estos son algunos ejemplos:

```
fecha | md5sum
94cb1cdecfed0699e2d98acd9a7b8f6d -
```

usando sha256sum:

```
fecha | sha256sum
30a0c6091e194c8c7785f0d7bb6e1eac9b76c0528f02213d1b6a5fbcc76cef
f4-
```

usando base64:

```
fecha | base64
0YHQsSDRj9C90YMgMzAgMTk6NTE6NDggRUVUIDIwMjEK
```

- También podemos usar openssl para generar bytes pseudoaleatorios y ejecutar la salida a través de base64. Un ejemplo de salida será:

```
abre ssl rand -base64 10
9+soM9bt8mhdcw==
```

Tenga en cuenta que es posible que openssl no esté instalado en su sistema, por lo que es probable que deba instalarlo primero para poder usarlo.

- La forma preferida es usar el generador de números pseudoaleatorios - /dev/urandom ya que está diseñado para la mayoría de los propósitos criptográficos. También necesitaríamos manipular la salida usando tr para traducirla. Un comando de ejemplo es:

```
tr -cd '[:alnum:]' < /dev/urandom | doblar -w10 | cabeza -n 1
```

Con este comando, tomamos la salida de /dev/urandom y la traducimos con tr mientras usamos todas las letras y dígitos e imprimimos la cantidad deseada de caracteres.

Primero comenzamos el guión con el tinglado. Lo usamos para decirle al sistema operativo qué intérprete usar para analizar el resto del archivo.

```
#!/bin/bash
```

Luego podemos continuar y pedirle al usuario alguna entrada. En este caso nos gustaría saber cuántos caracteres debe tener la contraseña:

```
# Solicitar al usuario la longitud de la
contraseña clear printf "\n" read -p
"¿Cuántos caracteres desea que tenga
la contraseña?" pass_length printf "\n"
```

Genere las contraseñas y luego imprímalas para que el usuario pueda usarlas.

```
# ¡Aquí es donde ocurre la magia!
# Genere una lista de 10 cadenas y córtela al valor deseado proporcionado
por el usuario

para i en {1..10}; do (tr -cd '[:alnum:]' < /dev/urandom | fold -w${pass_longt} | head
-n 1); hecho

# Imprime las cadenas
printf "$pass_output\n" printf
"Adiós, ${USUARIO}\n"
```

```
#!/bin/bash
#=======================================
# Generador de contraseñas con opción de inicio de
sesión #=======================================

# Pregunte al usuario por la longitud de la
cadena clear printf "\n" read -p "¿Cuántos
caracteres desea que tenga la contraseña?"
pass_lenght printf "\n"



# ¡Aquí es donde ocurre la magia!
# Genere una lista de 10 cadenas y córtela al valor deseado proporcionado
por el usuario

para i en {1..10}; do (tr -cd '[:alnum:]' < /dev/urandom | fold -w${pass_longt} | head
-n 1); hecho

# Imprime las cadenas
printf "$pass_output\n" printf
"Adiós, ${USUARIO}\n"
```

Así es más o menos cómo puede usar un script bash simple para generar contraseñas aleatorias.

:advertencia: **Como ya se mencionó, asegúrese de usar contraseñas seguras para asegurarse de que su cuenta esté protegida. Además, siempre que sea posible, use la autenticación de 2 factores, ya que esto proporcionará una capa adicional de seguridad para su cuenta.**

Si bien el script funciona bien, espera que el usuario proporcione la entrada solicitada. Para evitar cualquier problema, deberá realizar algunas comprobaciones más avanzadas en la entrada del usuario para asegurarse de que el script seguirá funcionando bien, incluso si la entrada proporcionada no coincide con nuestras necesidades.

alex georgiev

Un superusuario de Linux debe tener un buen conocimiento de las tuberías y la redirección en Bash. Es un componente esencial del sistema y, a menudo, es útil en el campo de la administración del sistema Linux.

Cuando ejecuta un comando como ls, cat, etc., obtiene algo de salida en la terminal. Si escribe un comando incorrecto, pasa un indicador incorrecto o un argumento de línea de comando incorrecto, obtiene un resultado de error en la terminal. En ambos casos, se le proporciona un texto. Puede parecerle "solo un mensaje de texto", pero el sistema trata este texto de manera diferente. Este identificador se conoce como descriptor de archivo (fd).

En Linux, hay 3 descriptores de archivos, **STDIN** (0); **STDOUT** (1) y **STDERR** (2).

- **STDIN** (fd: 0): Gestiona la entrada en el terminal.
- **STDOUT** (fd: 1): Gestiona el texto de salida en el terminal.
- **STDERR** (fd: 2): Gestiona el texto de error en el terminal.

Tanto *las canalizaciones* como *las redirecciones* redireccionan los flujos (descriptor de archivo) del proceso que se está ejecutando. La principal diferencia es que las *redirecciones* tratan con el flujo de archivos, enviando el flujo de salida a un archivo o enviando el contenido de un archivo dado al flujo de entrada del proceso.

Por otro lado, una tubería conecta dos comandos enviando el flujo de salida del primero al flujo de entrada del segundo. sin ninguna redirección especificada.

Cuando ingresa un texto de entrada para un comando que lo solicita, en realidad está ingresando el texto en el descriptor del archivo **STDIN .** Ejecute el comando cat sin argumentos de línea de comandos. Puede parecer que el proceso se ha detenido, pero en realidad es un gato que pide **STDIN.** cat es un programa simple e imprimirá el texto pasado a **STDIN.**
Sin embargo, puede extender el caso de uso al redirigir la entrada a los comandos que toman **STDIN.**

Ejemplo con gato:

```
gato << EOF
¡Hola Mundo!
¿Cómo estás?
fin de semana
```

Esto simplemente imprimirá el texto proporcionado en la pantalla del terminal:

```
Hola Mundo!
¿Cómo estás?
```

Lo mismo se puede hacer con otros comandos que toman entrada a través de STDIN. Me gusta, wc:

```
wc -l << EOF ¡Hola
Mundo!
¿Cómo estás?
fin de semana
```

El indicador -l con wc cuenta el número de líneas. Este bloque de código bash imprimirá el número de líneas en la pantalla del terminal:

```
2
```

El texto normal sin error en la pantalla de su terminal se imprime a través del descriptor de archivo **STDOUT .** El **STDOUT** de un comando se puede redirigir a un archivo, de tal manera que la salida del comando se escribe en un archivo en lugar de imprimirse en la pantalla del terminal. Esto se hace simplemente con la ayuda de los operadores > y >> .

Ejemplo:

```
echo "¡Hola mundo!" > archivo.txt
```

El siguiente comando no imprimirá "Hello World" en la pantalla del terminal, sino que creará un archivo llamado file.txt y escribirá la cadena "Hello World" en él. Esto se puede verificar ejecutando el comando cat en el archivo file.txt .

```
archivo gato.txt
```

Sin embargo, cada vez que redirija el **STDOUT** de cualquier comando varias veces al mismo archivo, eliminará los contenidos existentes del archivo para escribir los nuevos.

Ejemplo:

```
echo "¡Hola mundo!" > archivo.txt echo "¿Cómo
estás?" > archivo.txt
```

Al ejecutar cat en el archivo file.txt :

```
archivo gato.txt
```

Solo obtendrás el "¿Cómo estás?" cadena impresa.

```
¿Cómo estás?
```

Esto se debe a que se ha sobrescrito la cadena "Hello World". Este comportamiento puede ser

evitado usar el operador >> .

El ejemplo anterior se puede escribir como:

```
echo "¡Hola mundo!" > archivo.txt echo "¿Cómo
estás?" >> archivo.txt
```

Al ejecutar cat en el archivo file.txt , obtendrá el resultado deseado.

```
Hola Mundo!
¿Cómo estás?
```

Alternativamente, el operador de redirección para **STDOUT** también se puede escribir como 1>. Me gusta,

```
echo "¡Hola mundo!" 1> archivo.txt
```

El texto de error en la pantalla del terminal se imprime a través de **STDERR** del comando.
Por ejemplo:

```
ls--hola
```

daría un mensaje de error. Este mensaje de error es el **STDERR** del comando.

STDERR se puede redirigir usando el operador 2> .

```
ls --hola 2> error.txt
```

Este comando redirigirá el mensaje de error al archivo error.txt y lo escribirá en él.
Esto se puede verificar ejecutando el comando cat en el archivo error.txt .

También puede usar el operador 2>> para **STDERR** al igual que usó >> para **STDOUT.**

Los mensajes de error en Bash Scripts pueden ser indeseables a veces. Puede optar por ignorarlos redirigiendo el mensaje de error al archivo /dev/null . /dev/null es un pseudodispositivo que toma texto y luego lo descarta inmediatamente.

El ejemplo anterior se puede escribir de la siguiente manera para ignorar el texto de error por completo:

```
ls --hola 2> /dev/null
```

Por supuesto, puede redirigir **STDOUT** y **STDERR** para el mismo comando o secuencia de comandos.

```
./install_package.sh > salida.txt 2> error.txt
```

Ambos pueden ser redirigidos al mismo archivo también.

```
./install_package.sh > archivo.txt 2> archivo.txt
```

También hay una manera más corta de hacer esto.

```
./install_package.sh > archivo.txt 2>&1
```

Hasta ahora hemos visto cómo redirigir **STDOUT, STDIN** y **STDOUT** hacia y desde un archivo. Para concatenar la salida del programa *(comando)* como la entrada de otro programa *(comando)* puede usar una barra vertical |.

Ejemplo:

```
ls | grep ".txt"
```

Este comando enumerará los archivos en el directorio actual y pasará la salida al comando grep , que luego filtrará la salida para mostrar solo los archivos que contienen la cadena ".txt".

Sintaxis:

```
[tiempo [-p]] [!] comando1 [ | o |& comando2 ] ...
```

También puede construir cadenas arbitrarias de comandos conectándolos para lograr un resultado poderoso.

Este ejemplo crea una lista de todos los usuarios que poseen un archivo en un directorio determinado, así como la cantidad de archivos y directorios que poseen:

```
ls -l /proyectos/bash_scripts | cola -n +2 | sed 's/\s\s*/ /g' | cortar -d '
                ' -f 3 | ordenar | uniq -c
```

Producción:

```
8 ana
34 harry
37 tina 18
ryan
```

El símbolo << se puede usar para crear un archivo temporal [heredoc] y redirigir desde él en la línea de comando.

```
COMANDO << EOF
        ContenidoDeDocumento
        ...
        ...
fin de semana
```

Tenga en cuenta aquí que EOF representa el delimitador (final del archivo) del heredoc. De hecho, podemos usar cualquier palabra alfanumérica en su lugar para indicar el inicio y el final del archivo. Por ejemplo, este es un heredoc válido:

```
gato <<palabra aleatoria1
        Este script imprimirá estas líneas en la terminal.
        Tenga en cuenta que cat puede leer desde la entrada estándar.
Usando este heredoc, podemos crear un archivo temporal con estas líneas como
        su contenido y canalizarlo a cat. palabra aleatoria1
```

Efectivamente, parecerá que el contenido del heredoc se canaliza al comando.
Esto puede hacer que la secuencia de comandos sea muy limpia si es necesario canalizar varias líneas a un programa.

Además, podemos adjuntar más tuberías como se muestra:

```
gato << palabra aleatoria1 | WC
        Este script imprimirá estas líneas en la terminal.
        Tenga en cuenta que cat puede leer desde la entrada estándar. Usando
este heredoc, podemos crear un archivo temporal con estas líneas como su contenido
        y canalizarlo a cat. palabra aleatoria1
```

Además, las variables predefinidas se pueden usar dentro de heredocs.

Herestrings son bastante similares a heredocs pero usan <<<. Estos se utilizan para cadenas de una sola línea que deben canalizarse a algún programa. Esto parece más limpio que heredocs ya que no tenemos que especificar el delimitador.

```
wc <<<"esta es una manera fácil de pasar cadenas al stdin de un programa (aquí wc)"
```

Al igual que heredocs, herestrings puede contener variables.

Descripción del operador	
>	Guardar la salida en un archivo
>>	Agregar salida a un archivo
<	Leer la entrada de un archivo
2>	Redirigir mensajes de error
|	Enviar la salida de un programa como entrada a otro programa
<<	Canaliza varias líneas en un programa de forma limpia
<<<	Canaliza una sola línea en un programa limpiamente

¡Felicidades! ¡Acabas de completar la guía básica de Bash!

Si lo encuentra útil, ¡asegúrese de iniciar el proyecto en GitHub!

Si tiene alguna sugerencia de mejora, asegúrese de contribuir con solicitudes de incorporación de cambios o problemas abiertos.

En esta introducción al libro de secuencias de comandos de Bash, solo cubrimos los conceptos básicos, ¡pero aún tiene suficiente en su haber para comenzar a redactar algunas secuencias de comandos increíbles y automatizar las tareas diarias!

¡Como siguiente paso, intente escribir su propio guión y compártalo con el mundo! ¡Esta es la mejor manera de aprender cualquier nuevo lenguaje de programación o secuencias de comandos!

En caso de que este libro lo haya inspirado a escribir algunos guiones geniales de Bash, asegúrese de twittear al respecto y etiquetar a @bobbyiliev_ para que podamos comprobarlo!

¡Felicitaciones nuevamente por completar este libro!