

Clase 19: Shell Scripting

Parte de la flexibilidad de Linux reside en la posibilidad que nos brinda de configurar el entorno de trabajo fácilmente de acuerdo con las necesidades de cada usuario.

Además, si se desea impedir que una cuenta particular se utilice para iniciar sesión, se suele configurar **/bin/false** como shell por defecto para dicho usuario. Este es el caso de las cuentas que se utilizan para ejecutar servicios de red o procesos exclusivamente en segundo plano. El mismo efecto se logra para todos los usuarios si dentro del directorio **/etc** se coloca un archivo llamado **nologin**. Esto deshabilita automáticamente todo intento de inicio de sesión proveniente de cualquier cuenta con excepción de root, y devuelve por pantalla el contenido del archivo a título informativo.

Como si esto fuera poco, usando tan solamente un editor de texto podemos crear nuestros propios programas y ejecutarlos desde la línea de comandos para automatizar tareas.

Configurar el entorno de la shell

Cada usuario posee dentro de su directorio personal una serie de scripts en forma de archivos ocultos que configuran su entorno. Por ejemplo, **~/.bash_profile** (o **/.profile**) y **/.bashrc** almacenan uno o más alias y variables de entorno, mientras que **~/.bash_history** y **~/.bash_logout** guardan el historial de comandos y una lista opcional de acciones a realizar cuando cerramos nuestra sesión de usuario. Estos archivos pueden a su vez ejecutar otros scripts a través del comando **source** o utilizando el punto (**.**) como atajo, tal como observamos en la imagen siguiente. El resultado en ambos casos es el mismo: se lanza el segundo script dentro del mismo entorno que el primero. Como consecuencia, cualquier variable o alias que se establezca en el segundo estarán disponibles en el entorno del primero.

```
# .bash_profile

# Get the aliases and functions
if [ -f ~/.bashrc ]; then
    . ~/.bashrc
fi

# User specific environment and startup programs

PATH=$PATH:/sbin:$HOME/.local/bin:$HOME/bin

export PATH

HISTSIZE=1500
```

Ejecución encadenada de comandos

Hasta ahora hemos hablado de ejecutar comandos uno por uno desde la terminal. Sin embargo, Bash nos permite ejecutar una serie de comandos desde una sola línea, e incluso hacerlo bajo ciertas condiciones. En particular, al escribir

```
comando1; comando2; comando3
```

y presionar Enter, Bash ejecutará comando1, luego comando2 y finalmente comando3 sin detenerse en caso de algún fallo.

Por otro lado,

```
comando1 && comando2
```

hará que comando2 solamente corra en caso de que comando1 haya finalizado correctamente, mientras que

```
comando1 || comando2
```

hará que comando2 se ejecute únicamente en caso de que comando1 falle.

Consideremos los siguientes ejemplos para ilustrar, cuyo resultado podemos ver a continuación:

```
cd; echo "Soy un comando intermedio"; ls
ls -l /usuario && echo "Este mensaje aparecerá únicamente si /usuario existe"
ls -l /usuario || echo "Este mensaje aparecerá en caso de que /usuario no exista"
```

```
[gacanepa@server ~]$ cd; echo "Soy un comando intermedio"; ls
Soy un comando intermedio
almacenamiento.txt  basicos4.txt  ejemplo1.txt  listadescripts.txt  particiones.txt  pr.txt
alumnos.sql         basicos5.txt  ejemplo.sh    login1.txt          primerscript.sh  prueba1
basicos1.txt        capitales.txt enlace-prueba2.txt martinfiirro-cap1.txt proc1.txt         prueba.
basicos2.txt        centos.txt   firstapp      newcookies.txt      procesos          pruebas
basicos3.txt        cnncookies.txt introvim.txt  nohup.out         procesos.txt       prueba.
[gacanepa@server ~]$ ls -l /usuario && echo "Este mensaje aparecerá únicamente si /usuario existe"
ls: cannot access /usuario: No such file or directory
[gacanepa@server ~]$ ls -l /usuario || echo "Este mensaje aparecerá en caso de que /usuario no exista"
ls: cannot access /usuario: No such file or directory
Este mensaje aparecerá en caso de que /usuario no exista
[gacanepa@server ~]$
```

*Un ejemplo clásico del uso de && se encuentra en el proceso de instalar un paquete solamente si la actualización de los repositorios resultó exitosa. En otras palabras, `apt-get update && apt-get install tmux` solamente intentará instalar el paquete **tmux** si la sincronización*

con los repositorios se completa correctamente en Debian o similares. Lo mismo aplica para CentOS o distribuciones similares con yum, por ejemplo.

Como podemos apreciar, en el primer ejemplo todos los comandos de la lista se ejecutaron uno a continuación del otro, mientras que en los dos siguientes eso depende del estado de salida del comando con el que se inicia la lista. Recordemos que el estado de salida o *exit status* de un comando será 0 si finalizó correctamente o un número diferente caso contrario, lo cual podemos verificar a través de `echo $?` en cualquier momento.

Shell scripting

Una de las maneras de descubrir la potencia de la línea de comandos en Linux es mediante la creación de shell scripts, los cuales son archivos ejecutables de texto plano que contienen comando tras comando en líneas sucesivas. Dichos comandos son interpretados por una shell, un programa que los toma como entrada y actúa como interfaz para que el kernel los ejecute.

Primeros pasos

La costumbre al presentar un nuevo lenguaje de programación es mostrar el clásico mensaje *Hola mundo* por pantalla. Tomemos un curso diferente para hacer algo un poco más interesante, útil, y que nos permita presentar conceptos básicos.

Haciendo uso de un editor de texto, escribamos lo siguiente dentro de `primerscript.sh`:

```
#!/bin/bash

# La fecha actual
FECHA=$(date +%d/%m/%Y)

# Cantidad de sesiones abiertas
SESIONES=$(who | wc -l)

# Mensaje
echo "Hoy es $FECHA. Hay $SESIONES sesiones abiertas."
```

Hagamos algunas aclaraciones:

- La primera línea del script siempre debe indicar cuál es la shell con la que deseamos ejecutar el mismo. En la mayoría de las ocasiones elegiremos a Bash. La ruta absoluta a la shell debe ir precedida por los caracteres `#!`.

- Salvo la primera línea del script, todas aquellas que comiencen con un signo numeral (#) son comentarios informativos. Los usaremos para explicar el propósito de los comandos que siguen, con el propósito de aclarar el funcionamiento de nuestro script a otra persona que lo examine en el futuro (y para recordárnoslo a nosotros mismos también!). Estas líneas no son interpretadas por la shell durante la ejecución del script.
- Para evitar tener que repetir un mismo comando en varias partes del script, y para facilitar su lectura también, a menudo queremos guardar el resultado en una *variable*. De esa manera, si deseamos cambiar el comando en el futuro, solamente tendremos que hacerlo en el lugar en donde declaramos la variable. En el ejemplo de arriba guardamos la salida de los comandos que indican la fecha actual y la cantidad de sesiones de usuario abiertas en las variables FECHA y SESIONES, respectivamente. Luego de indicar el nombre deseado para la variable, colocamos los signos = \$ y encerramos el comando entre paréntesis.
- Finalmente, el comando echo nos servirá para mostrar un mensaje por pantalla en el que mezclamos texto de nuestra elección con el valor de FECHA y SESIONES. Cuando deseamos utilizar en otra parte del script el valor guardado en una variable, lo hacemos anteponiendo el signo \$ al nombre de la variable. A esta altura podemos darnos cuenta de que es mucho más amigable leer "Hoy es \$FECHA" que "Hoy es \$(date +%d/%m/%Y)".

Cerremos el archivo guardando los cambios y demos permisos de ejecución al archivo recién creado:

```
chmod +x primerscript.sh
```

Como paso final, ejecutemos el script anteponiendo un punto y una barra al nombre del script (esto es necesario para indicar que **primerscript.sh** es un archivo ejecutable presente en el directorio actual, donde por lo general el sistema no espera encontrar archivos de ese tipo):

```
./primerscript.sh
```

```
[gacanepa@server ~]$ chmod +x primerscript.sh
[gacanepa@server ~]$ ./primerscript.sh
Hoy es 04/09/2018. Hay 2 sesiones abiertas.
[gacanepa@server ~]$
```

Argumentos posicionales

A menudo queremos aplicar una acción a diferentes objetos, por lo que para evitar tener que modificar el script para cada caso en particular, podemos hacer uso de parámetros – o sea, valores que le pasamos al script al momento de ejecutarlo. Dentro

del script, el primer parámetro se llama mediante **\$1**, el segundo como **\$2**, y así sucesivamente.

El siguiente script (**backupbasico.sh**) espera dos parámetros:

- Directorio a comprimir
- Nombre deseado para el tarball resultante

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

```
# El directorio a comprimir es el primer parámetro  
DIRECTORIO=$1
```

```
# Nombre del tarball  
ARCHIVO=$2
```

```
# Comprimir los contenidos de $DIRECTORIO  
tar czf $ARCHIVO $DIRECTORIO/*
```

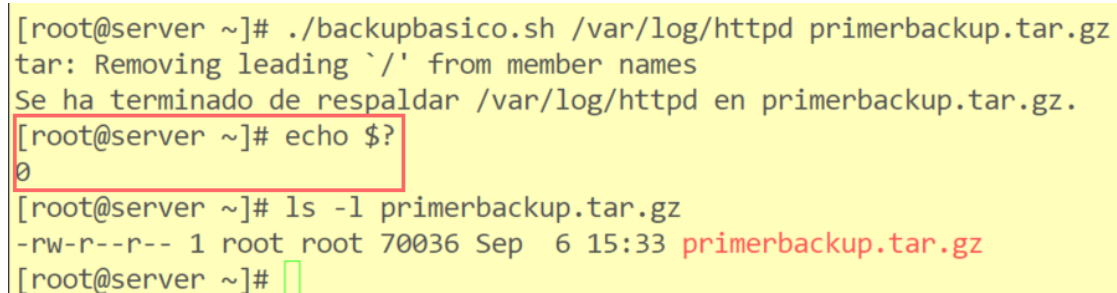
```
# Mostrar mensaje por pantalla  
echo "Se ha terminado de respaldar $DIRECTORIO en $ARCHIVO."
```

Démosle permiso de ejecución al script:

```
chmod +x backupbasico.sh
```

Y ejecutémoslo pasando **/var/log/httpd** y **primerbackup.tar.gz** como parámetros:

```
./backupbasico.sh /var/log/httpd primerbackup.tar.gz
```



```
[root@server ~]# ./backupbasico.sh /var/log/httpd primerbackup.tar.gz  
tar: Removing leading '/' from member names  
Se ha terminado de respaldar /var/log/httpd en primerbackup.tar.gz.  
[root@server ~]# echo $?  
0  
[root@server ~]# ls -l primerbackup.tar.gz  
-rw-r--r-- 1 root root 70036 Sep  6 15:33 primerbackup.tar.gz  
[root@server ~]#
```

Como podemos ver en la imagen de arriba, el script se ejecutó correctamente ya que su estado de salida es igual a 0.

Condicionales

La forma más elemental de control de flujo en cualquier lenguaje de programación es el uso de una estructura **if / else**. En shell scripting se implementa de la siguiente manera:

```
if [ CONDICIÓN ]; then
    [ ACCIÓN ]
else
    [ ACCIÓN ALTERNATIVA ]
fi
```

La indentación del código, aunque no es estrictamente requerida por Bash, ayuda a la lectura del script.

Entre las posibles condiciones que podríamos querer chequear están las siguientes, aunque hay [más alternativas](#):

- Existencia de un archivo o directorio
- Finalización correcta del último comando ejecutado
- Verificación de archivos vacíos

Para ejemplificar, tomaremos el script que creamos más arriba (**backupbasico.sh**) y lo modificaremos para que cumpla las siguientes condiciones:

- Antes de ejecutarse, chequear que el directorio a comprimir exista. Si no existe, finalizar con código de error 10 (una elección de código de error arbitraria).
- Si la compresión resulta exitosa, mostrar un mensaje por pantalla que así lo indique. De otra manera, mostrar un mensaje descriptivo y finalizar con código de error 20 (otra elección arbitraria).
- Verificar que el tarball no esté vacío (es decir, que su tamaño sea distinto de 0 B). Mostrar mensajes de confirmación o de error según corresponda (finalizar con error 30 en este último escenario).

El script resultante:

```
#!/bin/bash

# El directorio a comprimir es el primer parámetro
DIRECTORIO=$1

# Nombre del tarball
ARCHIVO=$2

# Chequear existencia de directorio
if [ -d $DIRECTORIO ]; then
    # Comprimir los contenidos de $DIRECTORIO
    tar czf $ARCHIVO $DIRECTORIO/*
```

```

        if [ $? -eq 0 ]; then
            if [ -s $ARCHIVO ]; then
                # Mostrar mensaje por pantalla
                echo "Se ha terminado de respaldar $DIRECTORIO en
$ARCHIVO."
            else
                # Indicar error por archivo vacío
                echo "El archivo resultante ($ARCHIVO) está vacío
."
            fi
        else
            # Indicar error y finalizar
            echo "La compresión de $DIRECTORIO en $ARCHIVO no finaliz
ó correctamente."
            exit 30
        fi
    else
        # Indicar si el directorio no existe y finalizar con código de er
ror
        echo "El directorio $DIRECTORIO no existe."
        exit 10
    fi

```

Explicuemos ahora el propósito de los argumentos de cada bloque if / else:

- La opción -d seguida de la ruta a un directorio se utiliza para chequear que el mismo exista.
- Como ya explicamos, \$? nos indica el código de finalización del comando inmediatamente anterior. La opción -eq 0 chequea si el código de finalización es igual a 0.
- La opción -s seguida de la ruta a un archivo verifica que el mismo exista y que su tamaño sea distinto a 0 B.

Veamos a continuación qué sucede y (el código de error asociado) cuando le pasamos al script:

- Un directorio inexistente
- Un directorio vacío

```
[root@server ~]# ./backupbasico.sh /home/yo yo.tar.gz
El directorio /home/yo no existe.
[root@server ~]# echo $?
10
[root@server ~]# mkdir dirvacio
[root@server ~]# ./backupbasico.sh dirvacio/ yo.tar.gz
tar: dirvacio/*: Cannot stat: No such file or directory
tar: Exiting with failure status due to previous errors
La compresión de dirvacio/ en yo.tar.gz no finalizó correctamente.
[root@server ~]# echo $?
30
[root@server ~]#
```

Como podemos apreciar, el uso de códigos de salida personalizados nos permite proveer mensajes de error que van de acuerdo con las situaciones encontradas.

Expresiones test

Como podemos ver, a menudo necesitaremos evaluar condiciones en nuestros scripts. Una forma alternativa de hacerlo es utilizando expresiones **test**, lo que además nos permite realizar operaciones lógicas del tipo **AND** y **OR** entre dos condiciones muy fácilmente.

Todas las expresiones de la lista siguiente se refieren al archivo **prueba1** que debemos haber creado dentro del mismo directorio donde colocaremos el script.

- [-f prueba1] se utiliza para chequear que prueba1 exista y que sea un archivo regular.
- [-e prueba1] o [-a prueba1] verifican que **prueba1** exista. Sin embargo, en este caso la condición se cumplirá incluso cuando se trate de un directorio, enlace simbólico, u otro tipo de archivo.
- [-s prueba1] será verdadero si prueba1 existe y su tamaño es mayor que 0 bytes.
- [-O prueba1] se utiliza para verificar que el usuario actual sea dueño del archivo.

Finalmente,

- [-r prueba1], [-w prueba1], y [-x prueba1] nos ayudarán a determinar si tenemos permisos de lectura, escritura, y ejecución sobre **prueba1**, respectivamente.

Para ver el listado completo de expresiones test, podemos referirnos a la documentación disponible en [The Linux Documentation Project](#).

Para comenzar, crearemos el archivo **prueba1** con una línea de contenido:

```
echo "Soy un archivo de prueba" > prueba1
```

Y también el script **test.sh**:

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

```
ARCHIVO=prueba1
```

```
# Verificación de condiciones
```

```
if [ -f $ARCHIVO -a -s $ARCHIVO ] ; then echo "$ARCHIVO existe, es un archivo regular, y su tamaño es mayor a 0 bytes."; fi
```

```
if [ -r $ARCHIVO -a -w $ARCHIVO ] ; then echo "$ARCHIVO existe y tengo permisos de lectura y escritura."; fi
```

```
if [ ! -x $ARCHIVO -o ! -O $ARCHIVO ]; then echo "No soy el dueño de $ARCHIVO o no tengo permisos de ejecución."; fi
```

Otorguemos el permiso de ejecución y luego corramos el script:

```
chmod +x test.sh
./test.sh
```

Veamos qué sucede cuando ejecutamos el script una primera vez. Luego cambiamos algunas de las condiciones y verifiquemos nuevamente:

```
[gacanepa@server ~]$ echo "Soy un archivo de prueba" > prueba1
[gacanepa@server ~]$ chmod +x test.sh
[gacanepa@server ~]$ ./test.sh
prueba1 existe, es un archivo regular, y su tamaño es mayor a 0 bytes.
prueba1 existe y tengo permisos de lectura y escritura.
[gacanepa@server ~]$ cat /dev/null > prueba1
[gacanepa@server ~]$ chmod u+x prueba1
[gacanepa@server ~]$ ./test.sh
prueba1 existe y tengo permisos de lectura y escritura.
[gacanepa@server ~]$
```

Podemos ver que luego de vaciar el archivo prueba1 con `cat /dev/null > prueba1`, la primera condición no se cumple más. Lo mismo sucede con la tercera condición luego de otorgarle permisos de ejecución al dueño del archivo. Finalmente, las operaciones lógicas **AND** y **OR** se implementan a través de `-a` y `-o`, respectivamente. Es posible implementar el **NOT** al colocar el signo de exclamación `!` delante de la condición que deseamos negar.

Bucles y toma de decisiones

Hasta ahora hemos tratado casos en los que no hay repetición en las acciones a realizar. Cuando deseamos repetirlas debemos recurrir al uso de bucles como explicaremos a continuación.

El bucle for

Este tipo de bucle se utiliza para realizar una determinada acción *un número definido de veces que se conoce de antemano*. La sintaxis es la siguiente:

```
for VARIABLE in elemento1 elemento2... elementoN; do
    acción
done
```

Por ejemplo, supongamos que quisiéramos renombrar una lista de archivos colocándoles **.bkp** al final:

```
for ARCHIVO in archivo1.conf archivo2.conf archivo3.conf; do
    mv $ARCHIVO $ARCHIVO.bkp
done
```

En este ejemplo, nombramos (de manera arbitraria) nuestra variable como **ARCHIVO**. La acción que deseamos realizar se llevará a cabo sobre cada archivo de la lista. Durante la primera iteración, **archivo1.conf** será renombrado a **archivo1.conf.bkp**. Lo mismo sucederá con **archivo2.conf** y **archivo3.conf** en la segunda y tercera iteraciones, respectivamente.

El bucle while

Si no conocemos inicialmente el número de elementos del conjunto, utilizaremos el bucle **while**, el que nos permite ejecutar una acción mientras (del inglés *while*) se cumpla una cierta condición. La sintaxis de este bucle se muestra a continuación:

```
while CONDICIÓN; do
    ACCIÓN
done
```

Mientras se cumpla **CONDICIÓN**, se llevará a cabo **ACCIÓN** (que puede consistir en una serie de comandos o en uno individual). En la siguiente sección ilustraremos el uso del bucle **while** al mismo tiempo que presentaremos una estructura adicional.

Uso de case

Un caso clásico del control de flujo de un programa o script se da en el caso cuando existen dos o más alternativas viables, y debemos elegir una basada en una condición. Por ejemplo, cuando se nos presenta un menú numerado por pantalla. En tal caso, el flujo del programa a partir de ese punto quedará dado por nuestra elección de las opciones disponibles.

Para ilustrar, supongamos que queremos examinar una a una las cuentas de usuario con acceso a Bash y elegir si deseamos agregar un recordatorio para quitarlo.

Guardemos el siguiente contenido dentro del script **ejemplocase.sh**:

```
#!/bin/bash

# En este archivo guardaremos la lista de tareas
ARCHIVO_TAREAS=tareas.txt

# En este otro tendremos la lista de usuarios con acceso a Bash
USUARIOS_BASH=usuariosBash.txt

# Si el archivo de tareas existe, borrarlo primero
if [ -f $ARCHIVO_TAREAS ]; then
    rm $ARCHIVO_TAREAS
fi

# Guardar la lista de usuarios con acceso a Bash en usuariosBash.txt
grep bash /etc/passwd | cut -d: -f1 > $USUARIOS_BASH

# Examinar cuenta por cuenta
while read CUENTA
do
    # Mostrar un mensaje por cada usuario. Leer la respuesta desde la terminal
    read -p "Quitar acceso a Bash a $CUENTA? [s/n]: " QUITAR < /dev/tty
    case $QUITAR in
        [sS])
            # Si la respuesta es Sí [sS], agregar mensaje al archivo
            echo "Quitar acceso a Bash a $CUENTA" >> $ARCHIVO_TAREAS
            ;;
        [nN])
            # De otra manera, mostrar mensaje por pantalla
            echo "Elegió no quitar el acceso a Bash a $CUENTA."
            ;;
    esac
done < $USUARIOS_BASH
```

y le daremos permisos de ejecución:

```
chmod +x ejemplocase.sh
```

Antes de ejecutarlo, aclaremos algunas cosas primero:

- La lista de usuarios con acceso a Bash se toma del primer campo del archivo **/etc/passwd** y se guarda en **usuariosBash.txt** utilizando **grep** y **cut**.

- El bucle **while** lee línea por línea del archivo **usuariosBash.txt** y guarda el contenido de la misma (que es diferente en cada iteración) en la variable **CUENTA**. En este caso, `read CUENTA` es la condición a evaluar. Mientras se lee el archivo (desde la primera hasta la última línea), esta condición se cumple.
- Con el comando `read -p` podemos mostrar un mensaje por pantalla y guardar la respuesta del usuario en la variable que colocamos luego del mensaje. En este caso dicha variable es **QUITAR**.
- Al colocar `< /dev/tty`, estamos indicando que debemos esperar por una respuesta del usuario mediante la terminal que controla el proceso actual (el script en ejecución).
- A continuación, con la palabra clave **case** evaluamos la entrada del usuario. Si dicha entrada consiste en una letra **s** o **S**, se guarda un mensaje en el archivo **tareas.txt**. De otra manera (**n** o **N**), se muestra un mensaje por pantalla.
- Cada opción evaluada por **case** debe finalizar en dos puntos y coma (`;;`), y el bloque debe cerrarse con la palabra clave **esac**.

Veamos el resultado de ejecutar **ejemplocase.sh**:

```
[gacanepa@server ~]$ ./ejemplocase.sh
Quitar acceso a Bash a root? [s/n]: n
Elegió no quitar el acceso a Bash a root.
Quitar acceso a Bash a gacanepa? [s/n]: n
Elegió no quitar el acceso a Bash a gacanepa.
Quitar acceso a Bash a postgres? [s/n]: s
Quitar acceso a Bash a fulano? [s/n]: s
[gacanepa@server ~]$ cat tareas.txt
Quitar acceso a Bash a postgres
Quitar acceso a Bash a fulano
[gacanepa@server ~]$ cat usuariosBash.txt
root
gacanepa
postgres
fulano
[gacanepa@server ~]$
```

Cuando se necesita realizar una acción en repetidas ocasiones dentro de un script, lo más conveniente es escribir una función. Luego podemos utilizarla las veces que sea necesario a fin de duplicar código innecesariamente al mismo tiempo que facilitamos lectura del script, como explicaremos en la siguiente sección.

Funciones

El concepto de función es el mismo que en los lenguajes de programación tradicionales. En palabras simples, una función es un conjunto de instrucciones a

llevarse a cabo. En nuestro caso, dichas instrucciones consisten en comandos que se ejecutarán de manera secuencial al invocar la función. El utilizar funciones en shell scripts nos permite reutilizar el código cuando lo necesitemos y de modificarlo en un solo lugar si se desea hacer algún cambio.

Para empezar, veamos cómo declarar una función en un script. Cualquiera de las dos alternativas mostradas a continuación es válida para declarar una función llamada **Ahora**:

```
function() Ahora{
comando1
comando2
}

o

Ahora() {
comando1
comando2
}
```

En este caso, comando1 y comando2 representan dos comandos cualesquiera, aunque podemos utilizar cuantos deseemos.

Aunque el siguiente ejemplo es un tanto trivial, servirá para ilustrar el uso de funciones en shell scripts. Supongamos que tenemos un script que realiza ciertas tareas y al terminar cada una guarda en un archivo de texto o muestra por pantalla la hora de finalización. El siguiente script (**ejemplofuncion.sh**) comienza con la declaración de la función Ahora. Esta función, al ser invocada con un argumento (como veremos más abajo), hará una pausa de 5 segundos y mostrará la fecha y hora actuales por pantalla 3 veces:

```
#!/bin/bash

# Declaramos la funcion
Ahora() {
    FECHA_HORA=$(date +%d-%m-%Y %H:%M:%S')
    sleep 5
    echo "Tarea $1 completa - $FECHA_HORA"
}

# Primera tarea: desayuno
Ahora desayuno

# Segunda tarea
```

Ahora almuerzo

Tercera tarea
Ahora cena

Ejecutemos el script luego de darle los permisos correspondientes:

```
chmod +x ejemplofuncion.sh  
./ejemplofuncion.sh
```

```
[gacanepa@server ~]$ chmod +x ejemplofuncion.sh  
[gacanepa@server ~]$ ./ejemplofuncion.sh  
Tarea desayuno completa - 06-09-2018 16:31:49  
Tarea almuerzo completa - 06-09-2018 16:31:54  
Tarea cena completa - 06-09-2018 16:31:59  
[gacanepa@server ~]$
```

Tal como observamos arriba, la función **Ahora** nos permite reutilizar código y facilita la lectura o el análisis del script.