

Sistemas Operativos en Red

UD 13. Introducción a Bash Scripting - Parte 2



Autores: Sergi García, Gloria Muñoz

Actualizado Febrero 2025



Licencia



Reconocimiento - No comercial - CompartirIgual (BY-NC-SA): No se permite un uso comercial de la obra original ni de las posibles obras derivadas, la distribución de las cuales se ha de hacer con una licencia igual a la que regula la obra original.

Nomenclatura

A lo largo de este tema se utilizarán diferentes símbolos para distinguir elementos importantes dentro del contenido. Estos símbolos son:

 **Importante**

 **Atención**

 **Interesante**

ÍNDICE

1. Uso del bucle While en Bash	3
2. Uso del bucle For en Bash	5
3. Uso del bucle For como si fuera For-Each en Bash	7
4. Uso de funciones en Bash	9
5. Ejercicios resueltos	12
6. Bibliografía	16

UNIDAD 13. INTRODUCCIÓN A BASH SCRIPTING - PARTE 2

1. USO DEL BUCLE WHILE EN BASH

En Bash, los bucles **while** también evalúan una condición antes de ejecutar un bloque de código. Si la condición es verdadera, el bloque de código se ejecuta y la condición se vuelve a evaluar al final de cada iteración. Esto continúa hasta que la condición se evalúa como falsa. A continuación, te presento cómo se pueden adaptar los ejemplos de bucles **while** a Bash:

Ejemplos de Bucles While en Bash

Contador simple

```
contador=1
while [ $contador -le 5 ]; do
    echo "Contador es: $contador"
    ((contador++))
done
```

Explicación: Este script inicializa un contador en 1 y lo incrementa en cada iteración del bucle **while** hasta que el contador supera 5. En cada iteración, imprime el valor actual del contador.

Espera hasta que un archivo exista

```
while [ ! -f "/tmp/miarchivo.txt" ]; do
    echo "Esperando el archivo..."
    sleep 2
done
echo "Archivo encontrado."
```

Explicación: Este bucle espera hasta que un archivo específico exista en la ubicación dada. Utiliza el comando **test** para verificar la existencia del archivo. Si no existe, el script duerme durante dos segundos antes de verificar nuevamente.

Menú de opciones

```
opcion=""
while [ "$opcion" != "3" ]; do
    echo "1. Opción A"
    echo "2. Opción B"
    echo "3. Salir"
    read -p "Seleccione una opción: " opcion
done
```

Explicación: Este bucle muestra un menú de opciones y permite al usuario seleccionar una opción. El bucle continúa hasta que el usuario selecciona la opción para salir (3).

Validación de entrada

```
while true; do
    read -p "Ingrese su edad (debe ser un número): " edad
```

```
if [[ "$edad" =~ ^[0-9]+$ ]]; then
    break
fi
done
echo "Edad válida: $edad años."
```

Explicación: Solicita la edad del usuario y asegura que sea un número mediante una expresión regular. El bucle persiste hasta que se introduce una entrada válida.

Loop de intentos

```
intentos=0
while [ $intentos -lt 3 ]; do
    read -p "Usuario: " usuario
    read -sp "Contraseña: " contraseña
    echo
    if [ "$usuario" = "admin" ] && [ "$contraseña" = "123" ]; then
        echo "Acceso concedido"
        break
    else
        echo "Acceso denegado. Intente de nuevo."
    fi
    ((intentos++))
done
```

Explicación: Este script permite al usuario intentar iniciar sesión hasta tres veces. Si las credenciales son correctas, el bucle se interrumpe prematuramente.

Suma de números

```
suma=0
numero=1
while [ "$numero" -ne 0 ]; do
    read -p "Introduce un número (0 para salir): " numero
    ((suma+=numero))
done
echo "Suma total: $suma"
```

Explicación: Permite al usuario introducir números continuamente, los cuales se suman a un total. El bucle termina cuando el usuario introduce 0.

Reintentar operación

```
completado=false
while [ "$completado" = false ]; do
    echo "Intentando operación..."
    # Simula que la operación fue exitosa modificando la variable a
    true
    completado=true
```

```
# Puede agregar una sección 'catch' utilizando comandos para
manejar errores
done
echo "Operación completada con éxito."
```

Explicación: Intenta realizar una operación que puede fallar, y si la operación falla, reintentará después de una pausa de un segundo.

Leer y procesar datos hasta un marcador

```
linea=""
while [ "$linea" != "FIN" ]; do
    read -p "Ingrese texto (FIN para terminar): " linea
    echo "Procesando: $linea"
done
```

Explicación: Permite al usuario ingresar líneas de texto hasta que escriba "FIN". Cada línea se procesa y se muestra de inmediato.

Bucle infinito controlado

```
while true; do
    read -p "Escriba 'salir' para terminar: " entrada
    if [ "$entrada" = "salir" ]; then
        break
    else
        echo "Usted escribió: $entrada"
    fi
done
```

Explicación: Crea un bucle infinito que sólo termina cuando el usuario escribe "salir". De lo contrario, imprime lo que el usuario escribió.

Conteo regresivo

```
inicio=10
while [ $inicio -gt 0 ]; do
    echo $inicio
    sleep 1
    ((inicio--))
done
echo "¡Tiempo terminado!"
```

Explicación: Realiza un conteo regresivo desde 10 hasta 1, mostrando cada número y esperando un segundo entre cada uno.

2. USO DEL BUCLE FOR EN BASH

En Bash, los bucles **for** se utilizan para iterar sobre listas de valores o ejecutar un bloque de código un número definido de veces. A continuación vemos como:

Ejemplos de Bucles For en Bash

Contar de 1 a 10

```
for i in {1..10}; do
    echo "Número: $i"
done
```

Explicación: Este script usa un bucle `for` para contar del 1 al 10. En cada iteración, muestra el valor actual del contador.

Mostrar los caracteres de una cadena

```
cadena="Hello"
for (( i=0; i<${#cadena}; i++ )); do
    echo "${cadena:$i:1}"
done
```

Explicación: Itera sobre cada carácter de la cadena "Hello" y los imprime uno por uno.

Iterar sobre un array

```
colores=("Rojo" "Verde" "Azul")
for (( i=0; i<${#colores[@]}; i++ )); do
    echo "Color $i: ${colores[$i]}"
done
```

Explicación: Recorre un array de colores e imprime cada color con su índice.

Bucle for con decremento

```
for (( i=10; i>=1; i-- )); do
    echo "Cuenta regresiva: $i"
done
```

Explicación: Realiza una cuenta regresiva desde 10 hasta 1, decrementando el valor de `i` en cada iteración.

Tabla de multiplicar

```
n=5
for (( i=1; i<=10; i++ )); do
    producto=$(( n * i ))
    echo "$n x $i = $producto"
done
```

Explicación: Genera la tabla de multiplicar del 5, multiplicando el número 5 por los números del 1 al 10.

Sumar números de un array

```
numeros={1..10}
suma=0
for i in "${numeros[@]"; do
```

```
    suma=$(( suma + i ))  
done  
echo "Suma total: $suma"
```

Explicación: Calcula la suma de los números del 1 al 10 almacenados en un array.

Bucle for con condiciones múltiples

```
for (( i=1, j=10; i<=10 && j>=1; i++, j-- )); do  
    echo "i = $i, j = $j"  
done
```

Explicación: Utiliza dos variables en el bucle **for**. **i** se incrementa de 1 a 10 y **j** se decrementa de 10 a 1 simultáneamente.

Ejecutar un comando para cada elemento de un array

```
commands=('date' 'whoami' 'pwd')  
for cmd in "${commands[@]}"; do  
    echo "Ejecutando: $cmd"  
    $cmd  
done
```

Explicación: Itera sobre un array de comandos de sistema y ejecuta cada uno de ellos, mostrando el comando que se está ejecutando.

Bucle for sin bloque de inicialización

```
i=1  
for (( ; i<=5; i++ )); do  
    echo "Valor de i: $i"  
done
```

Explicación: Comienza con **i** inicializado fuera del bucle y lo incrementa hasta que supera 5. Muestra el valor de **i** en cada iteración.

3. USO DEL BUCLE FOR COMO SI FUERA FOR-EACH EN BASH

En Bash, el bucle **for** se utiliza para iterar sobre listas de elementos, que es funcionalmente similar al bucle **foreach** en PowerShell. A continuación vamos como funciona:

Ejemplos de uso del bucle for como si fuera for-each en Bash

Impresión de elementos de un array

```
nombres=("Ana" "Beto" "Carlos")  
for nombre in "${nombres[@]}"; do  
    echo "$nombre"  
done
```

Explicación: Este script recorre un array de nombres e imprime cada nombre en la consola.

Suma de números en un array

```
numeros=(1 2 3 4 5)
suma=0
for numero in "${numeros[@]}"; do
    ((suma+=numero))
done
echo "La suma es: $suma"
```

Explicación: Este ejemplo suma todos los números en un array usando un bucle **for**.

Transformación de texto

```
textos=("hola" "mundo" "powershell")
for texto in "${textos[@]}"; do
    texto_mayuscula=$(echo "$texto" | tr '[:lower:]' '[:upper:]')
    echo "$texto_mayuscula"
done
```

Explicación: Convierte cada elemento de un array de cadenas a mayúsculas y las imprime.

Filtrado de datos

```
edades=(18 22 16 29 14)
for edad in "${edades[@]}"; do
    if [ "$edad" -ge 18 ]; then
        echo "Mayor de edad: $edad"
    fi
done
```

Explicación: Filtra y muestra solo las edades que son mayores o iguales a 18 años.

Manipulación de archivos

```
archivos=$(ls /path/to/directory)
for archivo in $archivos; do
    echo "Procesando archivo: $archivo"
done
```

Explicación: Recorre todos los archivos en un directorio específico e imprime el nombre de cada archivo.

Aplicación de descuentos

```
productos=(100 200 150)
descuento=0.1 # 10% de descuento
for precio in "${productos[@]}"; do
    precio_con_descuento=$(echo "$precio - ($precio * $descuento)" |
bc)
    echo "Precio con descuento: $precio_con_descuento"
done
```


Explicación: Aplica un descuento del 10% a cada precio de producto en la lista.

Validación de entradas

```
entradas=("email@example.com" "notanemail" "another@example.com")
for entrada in "${entradas[@]"; do
    if [[ "$entrada" =~
^([A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,4}$)]; then
        echo "$entrada es un email válido"
    else
        echo "$entrada no es un email válido"
    fi
done
```

Explicación: Verifica si cada entrada en un array es un correo electrónico válido según una expresión regular.

Conversión de unidades

```
kilometros=(1 5 10)
for km in "${kilometros[@]"; do
    millas=$(echo "$km * 0.621371" | bc)
    echo "$km km son equivalentes a $millas millas"
done
```

Explicación: Convierte valores en kilómetros a millas utilizando una relación de conversión fija.

Creación de directorios

```
carpetas=("Proyectos" "Documentos" "Imágenes")
for carpeta in "${carpetas[@]"; do
    mkdir -p "/home/usuario/$carpeta"
    echo "Creada carpeta: $carpeta"
done
```

Explicación: Crea directorios basados en los nombres especificados en un array de cadenas.

4. USO DE FUNCIONES EN BASH

En Bash, las funciones también se utilizan para agrupar código que realiza tareas específicas, facilitando la reutilización y mejorando la legibilidad del código. A continuación presentamos algunos ejemplos:

Ejemplos de funciones en Bash

Función simple

```
function saludar {
    echo "¡Hola, mundo!"
}
saludar
```

Explicación: Esta función no toma parámetros y simplemente imprime "¡Hola, mundo!" cada vez que se llama.

Función con parámetros

```
function saludarPersona {  
    local nombre=$1  
    echo "¡Hola, $nombre!"  
}  
saludarPersona "Pedro"
```

Explicación: Esta función toma un parámetro y utiliza este valor para personalizar un mensaje de saludo.

Función que devuelve valor

```
function sumar {  
    local num1=$1  
    local num2=$2  
    echo $((num1 + num2))  
}  
resultado=$(sumar 5 3)  
echo "Resultado: $resultado"
```

Explicación: Esta función toma dos números como parámetros y devuelve su suma.

Función con parámetros opcionales

```
function saludar {  
    local nombre=${1:-"Invitado"}  
    echo "¡Bienvenido, $nombre!"  
}  
saludar  
saludar "Ana"
```

Explicación: Esta función tiene un parámetro que es opcional y tiene un valor por defecto de "Invitado".

Función con validación de parámetros

```
function setEdad {  
    local edad=$1  
    if [[ $edad -ge 1 && $edad -le 150 ]]; then  
        echo "Edad configurada a $edad años."  
    else  
        echo "Error: Edad fuera de rango permitido."  
    fi  
}  
setEdad 25
```

Explicación: Esta función valida que el valor de edad esté entre 1 y 150.

Función que itera sobre un array

```
function mostrarColores {  
    local colores=("$@")  
    for color in "${colores[@]}"; do  
        echo "Color: $color"  
    done  
}  
misColores=("Rojo" "Verde" "Azul")  
mostrarColores "${misColores[@]}"
```

Explicación: Esta función recibe un array de colores como parámetro y utiliza un bucle para imprimir cada color.

Función con confirmación

```
function eliminarArchivo {  
    local ruta=$1  
    read -p "¿Está seguro que desea eliminar el archivo? (s/n) "  
confirm  
    if [[ $confirm == 's' ]]; then  
        rm $ruta  
        echo "Archivo eliminado."  
    else  
        echo "Operación cancelada."  
    fi  
}  
eliminarArchivo "/path/to/miarchivo.txt"
```

Explicación: Esta función solicita confirmación antes de eliminar un archivo.

Función con procesamiento pipeline

```
function getNumerosPares {  
    local numeros=("$@")  
    for num in "${numeros[@]}"; do  
        if ((num % 2 == 0)); then  
            echo $num  
        fi  
    done  
}  
array={1..10}  
pares=$(getNumerosPares "${array[@]}")  
echo "Números pares: ${pares[@]}"
```

Explicación: Esta función filtra y devuelve solo los números pares de un array de números.

5. EJERCICIOS RESUELTOS

Ejercicio 1: Promedio de notas

Enunciado:

Desarrolla un script en Bash que permita al usuario ingresar las notas de un estudiante hasta que decida detenerse e imprima el promedio de estas notas.

Solución:

```
function calcularPromedio {  
    local notas=()  
    local nota  
    local suma=0  
    local promedio  
  
    while true; do  
        read -p "Introduce una nota (o 'fin' para terminar): " nota  
        if [[ $nota == "fin" ]]; then  
            break  
        fi  
        notas+=($nota)  
        suma=$(echo "$suma + $nota" | bc)  
    done  
  
    if [[ ${#notas[@]} -gt 0 ]]; then  
        promedio=$(echo "scale=2; $suma / ${#notas[@]}" | bc)  
        echo "El promedio de las notas es: $promedio"  
    else  
        echo "No se introdujeron notas."  
    fi  
fi  
  
calcularPromedio
```

Explicación:

Este script utiliza una función `calcularPromedio` que pide al usuario ingresar notas continuamente hasta que ingrese 'fin'. Las notas se acumulan en un array y luego se calcula el promedio. Este ejercicio aplica un bucle `while` para la recogida de datos y operaciones aritméticas para el cálculo del promedio.

Ejercicio 2: Conteo de elementos pares en un array

Enunciado:

Crea un script en Bash que defina un array de números y utilice un bucle `for` para contar cuántos números pares hay en el array, mostrando este conteo al final.

Solución:

```
function contarPares {
    local numeros=({1..20})
    local conteoPares=0

    for numero in "${numeros[@]"; do
        if ((numero % 2 == 0)); then
            ((conteoPares++))
        fi
    done

    echo "Hay $conteoPares números pares en el array."
}

contarPares
```

Explicación:

En este script, se define un array de números del 1 al 20. Utiliza un bucle **for** para iterar a través de cada número en el array y un condicional **if** para verificar si el número es par. Si es par, incrementa un contador. Al final del bucle, el script imprime el número total de elementos pares encontrados.

Ejercicio 3: Lista de tareas pendientes**Enunciado:**

Escribe un script en Bash que permita al usuario introducir una lista de tareas pendientes y luego iterar sobre la lista para imprimirlas numeradas.

Solución:

```
function listarTareas {
    local tareas=()
    local tarea

    while true; do
        read -p "Introduce una tarea (o 'fin' para terminar): " tarea
        if [[ $tarea == "fin" ]]; then
            break
        fi
        tareas+=("$tarea")
    done

    for i in "${!tareas[@]"; do
        echo "$((i+1)). ${tareas[i]}"
    done
}
```

```
}  
  
listarTareas
```

Explicación:

Este script recopila tareas en un array utilizando un bucle **while** hasta que el usuario escribe 'fin'. Luego, utiliza un bucle **for** para imprimir cada tarea con su número correspondiente. Este ejercicio muestra cómo combinar bucles **while** y **for** para la entrada y procesamiento de datos en una lista simple y numerada.

Ejercicio 4: Gestión de habitaciones de hotel**Enunciado:**

Crea un programa para verificar si las habitaciones del hotel están disponibles (están vacías) o no (hay un huésped en la habitación). Además, las habitaciones disponibles pueden estar limpias o no limpias. Nuestro hotel tiene 100 habitaciones. En un bucle, el programa pedirá un número de habitación y luego mostrará un menú con esas opciones:

- Verificar si una habitación está disponible o no.
- Verificar si una habitación disponible está limpia o no.
- Configurar no disponible, una habitación limpia y disponible.
- Configurar disponible y no limpio para una habitación no disponible.
- Configurar limpiada, una habitación no limpiada.

Se desea que la información del programa sea persistente. Pista: pueden ayudarte los comandos "touch" para crear ficheros y "mkdir" para crear directorios y "rm" para eliminarlos.

Solución:

```
#!/bin/bash  
  
# Directorio base para las habitaciones  
base_dir="./hotel_rooms"  
  
# Crear el directorio base si no existe  
mkdir -p "$base_dir"  
  
# Función para verificar la disponibilidad de la habitación  
check_availability() {  
    local room_number=$1  
    local room_file="$base_dir/${room_number}"  
  
    if [ ! -e "$room_file" ]; then  
        echo "La habitación $room_number está disponible y limpia."  
    else  
        local status=$(<"$room_file")  
        echo "La habitación $room_number está $status."  
    fi  
}
```

```
fi
}

# Función para configurar el estado de la habitación
set_status() {
    local room_number=$1
    local status=$2
    local room_file="${base_dir}/${room_number}"

    case $status in
        "ocupada")
            echo "ocupada y no limpia" > "$room_file"
            ;;
        "disponible")
            rm -f "$room_file"
            ;;
        "limpiar")
            echo "disponible y limpia" > "$room_file"
            ;;
    esac
}

# Ciclo principal del programa
while true; do
    echo "1. Verificar si una habitación está disponible o no."
    echo "2. Configurar habitación como no disponible."
    echo "3. Configurar habitación como disponible y no limpia."
    echo "4. Configurar habitación como limpia."
    echo "5. Salir."
    read -p "Seleccione una opción: " option
    case $option in
        1)
            read -p "Ingrese el número de habitación: " room_number
            check_availability $room_number
            ;;
        2)
            read -p "Ingrese el número de habitación a marcar como
ocupada: " room_number
            set_status $room_number "ocupada"
            ;;
        3)
            read -p "Ingrese el número de habitación a marcar como
disponible: " room_number
```

```
        set_status $room_number "disponible"
        ;;
4)
    read -p "Ingrese el número de habitación a marcar como
limpia: " room_number
    set_status $room_number "limpiar"
    ;;
5)
    echo "Saliendo del programa."
    break
    ;;
*)
    echo "Opción no válida. Intente de nuevo."
    ;;
esac
done
```

Explicación:

El programa que proponemos utiliza un enfoque basado en archivos para representar el estado de cada habitación en un hotel. Cada habitación tendrá un archivo en un directorio específico, y la presencia o ausencia de un archivo, junto con su contenido, indicará si la habitación está disponible o no y si está limpia o no.

Directorio Base: El script crea un directorio base donde cada archivo representa una habitación. La ausencia de un archivo indica que la habitación está disponible y limpia.

Verificación de Disponibilidad: Se verifica si existe el archivo correspondiente a cada habitación. Si no existe, la habitación está disponible y limpia; si existe, se lee el contenido del archivo para saber si la habitación está ocupada y no limpia o disponible pero no limpia.

Configuración del Estado: Se puede cambiar el estado de cada habitación modificando o eliminando el archivo correspondiente. Esto permite marcar la habitación como ocupada (creando un archivo con el estado "ocupada y no limpia"), disponible ("disponible y limpia"), o limpiar una habitación disponible.

Persistencia: Los estados de las habitaciones se almacenan en archivos dentro del sistema de archivos del servidor, lo que asegura que la información se mantenga persistente entre ejecuciones del programa.

6. BIBLIOGRAFÍA

- Bash Scripting Guide <https://www.tldp.org/LDP/Bash-Beginners-Guide/html/>
- Advanced Bash-Scripting Guide <https://www.tldp.org/LDP/abs/html/>
- Bash Guide for Beginners
<https://www.tldp.org/LDP/Bash-Beginners-Guide/html/Bash-Beginners-Guide.html>
- GNU Bash manual <https://www.gnu.org/software/bash/manual/>