# Guía Completa sobre Shell Scripting en Bash

Bash (Bourne Again SHell) es el intérprete de comandos predeterminado en la mayoría de las distribuciones Linux y macOS. El shell scripting te permite automatizar secuencias de comandos de Bash, creando programas que pueden ejecutar tareas complejas, administrar archivos, interactuar con el sistema y mucho más.

## 1. Conceptos Fundamentales

### ¿Qué es un Shell Script?

Un shell script es simplemente un archivo de texto que contiene una secuencia de comandos que el intérprete Bash puede ejecutar. Es una forma de agrupar comandos para ejecutarlos juntos.

### El Shebang (#!)

La primera línea de un script Bash casi siempre debe ser el "shebang", que le indica al sistema operativo qué intérprete usar para ejecutar el archivo.

```
#!/bin/bash
# O a veces: #!/usr/bin/env bash (más portable)
```

- #!/bin/bash: Especifica la ruta directa al ejecutable de Bash.
- #!/usr/bin/env bash: Usa el comando env para buscar bash en el PATH del sistema, lo que lo hace un poco más flexible si bash está instalado en una ubicación no estándar.

### Permisos de Ejecución

Para poder ejecutar un script directamente (ej. ./mi_script.sh), necesita tener permisos de ejecución. Se otorgan con el comando chmod:

```
chmod +x mi_script.sh
```

### Ejecutar un Script

Hay varias formas de ejecutar un script:

1. **Directamente (con permisos):** ./mi_script.sh (si está en el directorio actual) o /ruta/completa/a/mi_script.sh. Requiere el shebang y permisos de ejecución.
2. **Pasándolo al intérprete:** bash mi_script.sh. No requiere permisos de ejecución ni shebang (aunque es buena práctica incluirlo). El script se ejecuta en una *subshell*.
3. **Usando source o . (punto):** source mi_script.sh o . mi_script.sh. Ejecuta los comandos del script en la *shell actual*, no en una subshell. Esto es útil si el script define variables o funciones que quieres que estén disponibles en tu sesión de terminal actual después de

que el script termine.

# 2. Variables

## Definición y Acceso

- Se definen sin $ y sin espacios alrededor del =.
- Se acceden usando $ antes del nombre, preferiblemente entre llaves {} para evitar ambigüedades.

```
#!/bin/bash

# Definición (sin espacios!)
nombre="Mundo"
edad=30

# Acceso
echo "Hola, ${nombre}!" # Usar llaves es más seguro
echo "Tienes $edad años." # Aquí sin llaves funciona, pero es menos robusto

# Ejemplo de ambigüedad sin llaves:
sufijo="_prueba"
echo "Variable: $nombresufijo" # Bash busca una variable llamada 'nombresufijo' (no existe)
echo "Variable: ${nombre}${sufijo}" # Correcto: imprime "Mundo_prueba"
```

## Tipos y Quoting (Comillas)

Bash no tiene tipos de datos estrictos como otros lenguajes; todo se trata inicialmente como una cadena. Las comillas son cruciales:

- **Comillas Dobles ("):** Permiten la **sustitución** de variables ($variable) y la **sustitución de comandos** (`comando` o $(comando)). Interpretan algunos caracteres especiales como \n (nueva línea con echo -e o printf).
- **Comillas Simples ('):** Tratan todo el texto **literalmente**. No hay sustitución de variables ni de comandos.
- **Sin Comillas:** Peligroso si el valor contiene espacios o caracteres especiales. Bash divide la cadena en palabras basándose en espacios y puede interpretar caracteres como * o ?. **¡Evítalo para valores de variables!**

```
#!/bin/bash

usuario=$(whoami) # Sustitución de comando
directorio_actual=`pwd` # Forma antigua de sustitución de comando

mensaje_dobles="Usuario: $usuario | Directorio: ${directorio_actual}"
mensaje_simples='Usuario: $usuario | Directorio: ${directorio_actual}' # Literal
```

```
echo "Con dobles: $mensaje_dobles"
echo "Con simples: $mensaje_simples"

archivo_con_espacios="Mi Documento.txt"
# echo $archivo_con_espacios # MAL: echo interpreta "Mi" y "Documento.txt" como argumentos separados
echo "$archivo_con_espacios" # BIEN: echo recibe "Mi Documento.txt" como un solo argumento
```

## Variables Especiales Importantes

Bash proporciona variables automáticas útiles:

- $0: Nombre del script.
- $1, $2, ... $9: Argumentos posicionales pasados al script.
- ${10}, ${11}, ...: Argumentos posicionales más allá del 9 (requieren llaves).
- $#: Número total de argumentos posicionales.
- $*: Todos los argumentos posicionales como una sola cadena (separados por el primer carácter de la variable especial IFS).
- $@: Todos los argumentos posicionales como palabras separadas. **Generalmente preferido sobre $*** cuando se necesita pasar argumentos a otro comando, especialmente si contienen espacios. "$@" expande cada argumento como una cadena separada y entrecomillada.
- $?: Código de salida (estado) del último comando ejecutado (0 significa éxito, cualquier otro valor indica un error).
- $$: ID del proceso (PID) de la shell actual.
- $!: PID del último comando ejecutado en segundo plano.
- $_: Último argumento del comando anterior.

```
#!/bin/bash
# Script: mostrar_args.sh
# Ejecutar como: ./mostrar_args.sh arg1 "arg con espacios" arg3

echo "Nombre del script: $0"
echo "Primer argumento: $1"
echo "Segundo argumento: $2"
echo "Número de argumentos: $#"
echo "Todos los argumentos (\$*): $*"
echo "Todos los argumentos (\$@): $@"

echo "--- Iterando con \$* ---"
for arg in "$*"; do # Trata todos los argumentos como UNA sola cadena
  echo "Arg: '$arg'"
done
```

```
echo "--- Iterando con \$@ ---"
for arg in "$@"; do # Trata cada argumento como una cadena SEPARADA (correcto)
  echo "Arg: '$arg'"
done

ls /ruta/inexistente > /dev/null 2>&1 # Comando que fallará
echo "Código de salida de ls: $?" # Imprimirá un valor distinto de 0

echo "PID de este script: $$"
```

# 3. Entrada y Salida

## Salida: echo y printf

- echo: Simple, ampliamente usado. Útil para mensajes rápidos.
    - echo "Mensaje"
    - echo -n "Sin nueva línea al final"
    - echo -e "Interpretando\nescapes\tcomo este" (no portable, depende de la versión de echo)
- printf: Más potente y **portable**, similar al printf de C. Permite formatear la salida. Es **preferible a echo -e**.
    - printf "Hola, %s. Tienes %d años.\n" "$nombre" "$edad"
    - %s para cadenas, %d para enteros, %f para flotantes, \n para nueva línea, \t para tabulador.

```
#!/bin/bash

nombre="Ana"
puntos=150

echo "--- Usando echo ---"
echo "Jugador: $nombre"
echo "Puntuación: $puntos"

echo "--- Usando printf ---"
printf "Jugador: %s\nPuntuación: %d\n" "$nombre" "$puntos"
printf "Puntuación formateada: [%10d]\n" "$puntos" # Rellena con espacios a la izquierda hasta 10 caracteres
printf "Nombre formateado: [%-15s]\n" "$nombre" # Alinea a la izquierda en 15 caracteres
```

## Entrada: read

Lee una línea de la entrada estándar (normalmente el teclado) y la asigna a una o más variables.

- read nombre_variable: Lee toda la línea en nombre_variable.
- read var1 var2: Divide la entrada por espacios y asigna a var1, var2, etc. (el resto va a la última variable).
- read -p "Mensaje: " variable: Muestra un mensaje (prompt) antes de leer.
- read -s variable: Lectura silenciosa (ej. para contraseñas).
- read -r variable: Modo "raw", no interpreta backslashes (recomendado generalmente).

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

```
printf "Introduce tu nombre: "
read nombre

read -p "Introduce tu edad: " edad
read -sp "Introduce tu contraseña: " contrasena # -s para silencioso
echo # Añadir nueva línea después de la entrada silenciosa

printf "\nHola, %s (%d años).\n" "$nombre" "$edad"
# ¡No imprimas la contraseña en un script real!
# printf "Tu contraseña tiene %d caracteres.\n" "${#contrasena}"
```

# 4. Sustitución de Comandos

Permite capturar la salida de un comando y usarla, típicamente asignándola a una variable.

- **Forma moderna y preferida: $(comando)** (se puede anidar).
- Forma antigua: `comando` (más difícil de leer y anidar).

```
#!/bin/bash

fecha_actual=$(date "+%Y-%m-%d %H:%M:%S")
num_archivos=$(ls -1 | wc -l) # Cuenta los archivos en el directorio actual
# Forma antigua: num_archivos=`ls -1 | wc -l`

echo "Fecha y hora: $fecha_actual"
echo "Número de archivos/directorios aquí: $num_archivos"

# Anidado
info_sistema=$(uname -a)
echo "Info del sistema: $(echo "$info_sistema" | cut -d' ' -f1,3)" # Obtiene campos 1 y 3
```

# 5. Aritmética

Bash no es ideal para matemáticas complejas, pero maneja enteros.

- **Forma moderna: $((expresion))**
- Comando expr: Más antiguo, requiere espacios alrededor de los operadores y escapar caracteres especiales. expr 5 * \( 10 + 2 \)
- Comando let: Similar a $((...)), modifica variables directamente. let x=5+3
- bc: Para aritmética de precisión arbitraria (incluyendo flotantes).

```
#!/bin/bash

a=10
b=3

# Usando $((...)) - Preferido
suma=$((a + b))
resta=$((a - b))
multiplicacion=$((a * b)) # No necesita escapar * aquí
division=$((a / b))     # División entera
resto=$((a % b))
potencia=$((a ** b))    # Exponenciación (Bash >= 4)

echo "Suma: $suma"
```

```
echo "Resta: $resta"
echo "Multiplicación: $multiplicacion"
echo "División entera: $division"
echo "Resto: $resto"
echo "Potencia: $potencia"

# Incremento/Decremento
let a++ # a ahora es 11
((b--)) # b ahora es 2
echo "a incrementado: $a"
echo "b decrementado: $b"

# Con números flotantes usando bc
num1=10.5
num2=3.2
suma_flotante=$(echo "$num1 + $num2" | bc)
echo "Suma flotante: $suma_flotante"

# Comparación aritmética dentro de (())
if (( a > b )); then
  echo "$a es mayor que $b"
fi
```

# 6. Estructuras Condicionales

## if, elif, else

La estructura básica para tomar decisiones.
if comando_o_test; then
  # Código si el comando_o_test tiene éxito (código de salida 0)
elif otro_comando_o_test; then
  # Código si el comando anterior falla y este tiene éxito
else
  # Código si todos los anteriores fallan
fi # Marca el final del if

## Comandos de Test (test, [ ], [[ ]])

Se usan dentro de if para evaluar condiciones.

- **test expresion** o **[ expresion ]** (¡OJO con los espacios alrededor de [ y ])
    - Forma clásica y portable (POSIX).
    - Menos características que [[ ]].
    - Requiere comillas en las variables y escapar operadores como < y >.
- **[[ expresion ]]** (¡OJO con los espacios!)
    - Forma **moderna y preferida en Bash**. Más robusta y con más funcionalidades.
    - No realiza división de palabras ni expansión de nombres de archivo dentro, haciendo más seguro el uso de variables sin comillas (aunque ponerlas sigue siendo buena práctica).
    - Soporta operadores como == para coincidencia de patrones (globbing), && (AND lógico), || (OR lógico) y expresiones regulares con =~.

**Operadores Comunes (dentro de [ ] o [[ ]]):**

- **Comparación de Cadenas:**
    - str1 = str2 (igual) (== también funciona en [[ ]])
    - str1 != str2 (distinto)
    - -z str (cadena vacía)
    - -n str (cadena no vacía)
    - str1 < str2 (menor que, orden lexicográfico - ¡escapar < en [ ]!)
    - str1 > str2 (mayor que - ¡escapar > en [ ]!)
- **Comparación Numérica:**
    - num1 -eq num2 (equal)
    - num1 -ne num2 (not equal)
    - num1 -lt num2 (less than)
    - num1 -le num2 (less or equal)

    - num1 -gt num2 (greater than)
    - num1 -ge num2 (greater or equal)
- **Comprobaciones de Archivos:**
    - -e file (existe, sea archivo o directorio)
    - -f file (existe y es un archivo regular)
    - -d file (existe y es un directorio)
    - -s file (existe y tiene tamaño mayor que cero)
    - -r file (existe y tiene permiso de lectura)
    - -w file (existe y tiene permiso de escritura)
    - -x file (existe y tiene permiso de ejecución)
- **Operadores Lógicos:**
    - ! expr (NOT lógico)
    - expr1 -a expr2 (AND lógico en [ ])
    - expr1 -o expr2 (OR lógico en [ ])
    - expr1 && expr2 (AND lógico en [[ ]])
    - expr1 || expr2 (OR lógico en [[ ]])

```
#!/bin/bash

read -p "Introduce un número: " num
read -p "Introduce una palabra: " palabra

# Usando [[ ]] (preferido)
if [[ -z "$num" ]]; then
  echo "No introdujiste un número."
elif ! [[ "$num" =~ ^[0-9]+$ ]]; then # Comprobar si es un entero positivo con regex
  echo "'$num' no parece ser un número entero positivo."
elif (( num > 100 )); then # Comparación aritmética con (())
  echo "$num es mayor que 100."
elif [[ "$palabra" == "secreto" ]]; then
  echo "La palabra es 'secreto'."
else
  echo "El número es $num (menor o igual a 100) y la palabra es '$palabra'."
fi

archivo="mi_script.sh" # Cambia esto por un archivo/directorio real

if [[ -f "$archivo" && -r "$archivo" ]]; then
  echo "El archivo '$archivo' existe y se puede leer."
elif [[ -d "$archivo" ]]; then
  echo "'$archivo' es un directorio."
else
  echo "'$archivo' no es un archivo regular legible o no existe."
fi
```

## case

Útil para comparar una variable contra múltiples patrones.

```
case "$variable" in
  patron1)
    # Código si $variable coincide con patron1
    ;; # Doble punto y coma para terminar la sección
  patron2|patron3) # Múltiples patrones con |
    # Código si $variable coincide con patron2 O patron3
    ;;
  *.txt) # Patrones de globbing (como en el shell)
    # Código si $variable termina en .txt
    ;;
  *) # Patrón por defecto (opcional)
    # Código si no coincide con ningún patrón anterior
    ;;
esac # Marca el final del case
```

```bash
#!/bin/bash

read -p "Introduce una opción (a, b, c, o salir): " opcion

case "$opcion" in
  a|A)
    echo "Seleccionaste la opción A."
    ;;
  b|B)
    echo "Seleccionaste la opción B."
    ;;
  c|C)
    echo "Seleccionaste la opción C."
    ;;
  salir|SALIR)
    echo "Saliendo..."
    exit 0 # Salir del script con éxito
    ;;
  *)
    echo "Opción inválida: '$opcion'"
    exit 1 # Salir del script con error
    ;;
esac
```

# 7. Bucles

## for

Itera sobre una lista de elementos.
# Forma 1: Iterar sobre una lista explícita
for variable in elemento1 elemento2 "elemento con espacios" elementoN; do
  # Código que usa $variable
done

# Forma 2: Iterar sobre la salida de un comando
for item in $(ls *.txt); do # Cuidado con espacios en nombres de archivo!
  echo "Archivo TXT: $item"
done
# Forma más segura para archivos (usando find y while read):
# find . -maxdepth 1 -name "*.txt" -print0 | while IFS= read -r -d $'\0' file; do
#   echo "Archivo TXT seguro: '$file'"
# done

# Forma 3: Bucle estilo C (aritmético)
for (( i=0; i<5; i++ )); do
  echo "Número: $i"
done

## while

Ejecuta un bloque de código mientras un comando o test tenga éxito (código de salida 0).
while comando_o_test; do
  # Código a ejecutar mientras la condición sea verdadera
done
```bash
#!/bin/bash

contador=0
while (( contador < 5 )); do
  echo "Contador (while): $contador"
  ((contador++))
  # O: let contador=contador+1
done

# Ejemplo común: leer líneas de un archivo
archivo_entrada="lista.txt"
```

```
if [[ ! -f "$archivo_entrada" ]]; then
  echo "Error: El archivo '$archivo_entrada' no existe."
  exit 1
fi

linea_num=1
while IFS= read -r linea || [[ -n "$linea" ]]; do # Lee línea a línea de forma segura
  # IFS= previene eliminar espacios al inicio/final
  # -r previene interpretar backslashes
  # || [[ -n "$linea" ]] maneja el caso de que la última línea no tenga un salto de línea final
  printf "Línea %d: %s\n" "$linea_num" "$linea"
  ((linea_num++))
done < "$archivo_entrada" # Redirige el archivo como entrada estándar para el while
```

## until

Ejecuta un bloque de código hasta que un comando o test tenga éxito. Es lo opuesto a while.

```
until comando_o_test; do
  # Código a ejecutar mientras la condición sea FALSA
done
```

```bash
#!/bin/bash

# Esperar hasta que un archivo exista
archivo_esperado="flag.tmp"
echo "Esperando por el archivo '$archivo_esperado'..."

until [[ -e "$archivo_esperado" ]]; do
  printf "."
  sleep 2 # Esperar 2 segundos
done

echo "\n¡Archivo '$archivo_esperado' encontrado!"
rm "$archivo_esperado" # Limpiar
```

## Control de Bucles: break y continue

- break: Sale inmediatamente del bucle actual (for, while, until).
- continue: Salta el resto de la iteración actual y pasa a la siguiente.

```
#!/bin/bash

for i in {1..10}; do
  if (( i == 3 )); then
```

```
  continue # Salta la impresión del 3
 fi
 if (( i == 8 )); then
  break # Sale del bucle cuando i es 8
 fi
 echo "Valor: $i"
done
# Salida: 1, 2, 4, 5, 6, 7
```

# 8. Funciones

Permiten agrupar código reutilizable.

```
# Forma 1 (preferida)
nombre_funcion() {
 # Código de la función
 # Argumentos: $1, $2, ..., $#, $@, $* (locales a la función)
 # Variables locales: usar 'local'
 local variable_local="Soy local"
 echo "Dentro de la función: $1"
 # Valor de retorno: código de salida (0=éxito, >0=error)
 return 0 # O return 1, etc.
}

# Forma 2 (estilo Bourne)
function nombre_funcion_alt {
 # Código...
}

# Llamar a la función
nombre_funcion "argumento1" "argumento 2"
codigo_salida=$? # Capturar el código de salida de la función

if (( codigo_salida == 0 )); then
 echo "La función terminó con éxito."
else
 echo "La función terminó con error (código $codigo_salida)."
fi

# Para devolver un valor (distinto del código de salida),
# la función puede imprimirlo y la llamada capturarlo:
obtener_valor() {
 local resultado="Valor calculado"
```

```
  echo "$resultado" # Imprime el valor a la salida estándar
}

mi_valor=$(obtener_valor)
echo "Valor obtenido de la función: $mi_valor"
```

**Importante sobre return:** Solo devuelve un código de estado numérico (0-255). Para devolver cadenas u otros datos, imprime el valor dentro de la función y captúralo fuera con $(...). Usa local para las variables dentro de las funciones para evitar colisiones con variables globales.

# 9. Arrays

Bash soporta arrays indexados (numéricamente, empezando en 0) y asociativos (clave-valor, Bash >= 4).

## Arrays Indexados

```
# Definición
mi_array=("manzana" "banana" "naranja con espacio")
otro_array=([2]="dos" [0]="cero" [1]="uno") # Índices explícitos

# Acceso a elementos
echo ${mi_array[0]}      # manzana
echo ${mi_array[1]}      # banana
echo "${mi_array[2]}"    # naranja con espacio (¡comillas importantes!)

# Acceso a todos los elementos
echo "${mi_array[@]}"    # manzana banana naranja con espacio (como palabras separadas)
echo "${mi_array[*]}"    # manzana banana naranja con espacio (como una sola palabra)

# Obtener los índices
echo "${!mi_array[@]}"   # 0 1 2

# Número de elementos
echo ${#mi_array[@]}     # 3

# Añadir elemento
mi_array+=("uva")
echo "${mi_array[@]}"    # manzana banana naranja con espacio uva

# Iterar
for fruta in "${mi_array[@]}"; do
  echo "Fruta: $fruta"
done
```

## Arrays Asociativos (Bash >= 4)

Necesitan ser declarados explícitamente con declare -A.

```
#!/bin/bash

# Declaración obligatoria
declare -A capitales
```

```
# Asignación
capitales["España"]="Madrid"
capitales["Francia"]="París"
capitales["Italia"]="Roma"

# Acceso
echo "La capital de Francia es ${capitales["Francia"]}"

# Acceso a todos los valores
echo "Valores: ${capitales[@]}"

# Acceso a todas las claves (índices)
echo "Claves: ${!capitales[@]}"

# Número de elementos
echo "Número de capitales: ${#capitales[@]}"

# Añadir/Modificar
capitales["Alemania"]="Berlín"
capitales["Francia"]="Paris" # Modifica

# Iterar sobre claves
for pais in "${!capitales[@]}"; do
  echo "País: $pais, Capital: ${capitales[$pais]}"
done
```

# 10. Manipulación de Cadenas

Bash ofrece varias formas de manipular cadenas sin llamar a herramientas externas como sed o awk (aunque estas son más potentes para tareas complejas).

- **Longitud:** ${#variable}
- **Subcadena:** ${variable:offset:longitud} (offset empieza en 0)
- **Reemplazo (primera ocurrencia):** ${variable/patron/reemplazo}
- **Reemplazo (todas las ocurrencias):** ${variable//patron/reemplazo}
- **Eliminar prefijo más corto:** ${variable#patron} (patrón es un glob)
- **Eliminar prefijo más largo:** ${variable##patron}
- **Eliminar sufijo más corto:** ${variable%patron}
- **Eliminar sufijo más largo:** ${variable%%patron}
- **Mayúsculas/Minúsculas (Bash >= 4):**
    - ${variable^^} (Todo a mayúsculas)
    - ${variable,,} (Todo a minúsculas)
    - ${variable^} (Primera letra a mayúsculas)
    - ${variable,} (Primera letra a minúsculas)

```
#!/bin/bash

archivo="/home/usuario/documentos/reporte_final.txt.gz"
nombre_completo="Juan Pérez Gómez"

# Longitud
echo "Longitud de archivo: ${#archivo}"

# Subcadena
echo "Usuario: ${archivo:6:7}" # usuario

# Reemplazo
echo "Cambiando home: ${archivo/\/home/\/disco_externo}"
echo "Quitando todos los espacios: ${nombre_completo// /_}"

# Eliminar prefijo (ruta)
nombre_archivo=${archivo##*/} # reporte_final.txt.gz
echo "Nombre de archivo: $nombre_archivo"

# Eliminar sufijo (extensión)
base_nombre=${nombre_archivo%.*} # reporte_final.txt
echo "Nombre base: $base_nombre"
base_nombre_sin_gz=${nombre_archivo%%.*} # reporte_final (elimina .txt.gz)
echo "Nombre base sin gz: $base_nombre_sin_gz"
```

```
# Mayúsculas/Minúsculas
echo "Mayúsculas: ${nombre_completo^^}"
echo "Minúsculas: ${nombre_completo,,}"
echo "Primera Mayúscula: ${nombre_completo^}"
```

# 11. Operaciones con Archivos y Directorios

Además de los tests -f, -d, etc., puedes usar comandos estándar:

- touch archivo: Crea un archivo vacío o actualiza su fecha de modificación.
- mkdir directorio: Crea un directorio. mkdir -p a/b/c crea directorios padres si no existen.
- cp origen destino: Copia archivos o directorios (cp -r para directorios).
- mv origen destino: Mueve o renombra archivos/directorios.
- rm archivo: Elimina un archivo. rm -r directorio elimina un directorio y su contenido (¡con cuidado!). rm -f fuerza la eliminación sin preguntar.
- ln -s objetivo enlace: Crea un enlace simbólico.
- cat archivo: Muestra el contenido de un archivo.
- grep patron archivo: Busca líneas que coincidan con un patrón en un archivo.
- find directorio -name patron -print: Busca archivos/directorios.

**Redirección:**

- comando > archivo: Redirige la salida estándar (stdout) a archivo (sobrescribe).
- comando >> archivo: Redirige stdout a archivo (añade al final).
- comando < archivo: Usa archivo como entrada estándar (stdin) para comando.
- comando 2> archivo_error: Redirige la salida de error (stderr) a archivo_error.
- comando > archivo_salida 2>&1: Redirige stdout a archivo_salida y stderr al mismo sitio que stdout.
- comando &> archivo_completo: Forma corta (Bash) para redirigir stdout y stderr al mismo archivo.
- comando1 | comando2: Tubería (pipe). La salida estándar de comando1 se convierte en la entrada estándar de comando2.

```
#!/bin/bash

DIR_TEMP="mi_temp_dir"
ARCHIVO_LOG="mi_app.log"
ARCHIVO_ERR="mi_app.err"

# Crear directorio temporal si no existe
if [[ ! -d "$DIR_TEMP" ]]; then
  mkdir "$DIR_TEMP"
  echo "Directorio temporal '$DIR_TEMP' creado."
fi
```

```
# Ejecutar un comando y redirigir salidas
echo "--- Ejecutando comando ---" > "$ARCHIVO_LOG" # Sobrescribir log
date >> "$ARCHIVO_LOG" # Añadir fecha al log

# Simular un comando que puede dar salida o error
ls /etc/passwd /etc/nonexistent_file >> "$ARCHIVO_LOG" 2>> "$ARCHIVO_ERR"
codigo_salida=$?

echo "--- Fin del comando ---" >> "$ARCHIVO_LOG"

if (( codigo_salida == 0 )); then
  echo "Comando ejecutado con éxito. Ver '$ARCHIVO_LOG'."
else
  echo "Comando falló (código $codigo_salida). Ver '$ARCHIVO_LOG' y '$ARCHIVO_ERR'."
  echo "--- Errores ---"
  cat "$ARCHIVO_ERR" # Mostrar errores
fi

# Limpieza (opcional)
# read -p "¿Eliminar directorio temporal y logs? (s/N): " respuesta
# if [[ "$respuesta" =~ ^[sS]$ ]]; then
#   rm -r "$DIR_TEMP" "$ARCHIVO_LOG" "$ARCHIVO_ERR"
#   echo "Limpieza realizada."
# fi
```

# 12. Manejo de Errores

- **Código de Salida ($?):** Siempre comprueba el código de salida de comandos importantes si su fallo puede afectar el resto del script.
- **set -e:** Hace que el script termine **inmediatamente** si cualquier comando (simple) falla (devuelve un código de salida distinto de 0). ¡Útil pero puede ser drástico! No se aplica a comandos dentro de if, while, until, ||, && o !.
- **set -u:** Trata el uso de variables no definidas como un error y termina el script. Ayuda a detectar typos en nombres de variables.
- **set -o pipefail:** Hace que el código de salida de una tubería (|) sea el del último comando de la tubería que falló (distinto de cero), en lugar de ser siempre el del último comando. Muy recomendable junto con set -e.
- **trap:** Permite ejecutar un comando cuando el script recibe una señal del sistema (ej. EXIT, INT, TERM). Útil para limpieza.

```
#!/bin/bash

# Opciones recomendadas al inicio para scripts robustos:
set -e  # Salir si un comando falla
set -u  # Salir si se usa una variable no definida
set -o pipefail # El código de salida de una pipe es el del último comando fallido

# Función de limpieza a ejecutar al salir
limpieza() {
  echo "Ejecutando limpieza..."
  rm -f archivo_temporal*.tmp
  echo "Limpieza completada."
}

# Registrar la función de limpieza para la señal EXIT (salida normal o por error)
trap limpieza EXIT
# También puedes atrapar otras señales como INT (Ctrl+C) o TERM
# trap 'echo "Interrumpido!"; exit 1' INT TERM

echo "Creando archivo temporal..."
TMP_FILE="archivo_temporal_$$.tmp"
date > "$TMP_FILE"

echo "Procesando datos..."
# Simular un comando que podría fallar en una tubería
cat "$TMP_FILE" | grep "patron_inexistente" | sort > /dev/null
```

```
# Sin 'set -o pipefail', el código de salida sería 0 (de sort) aunque grep falle.
# Con 'set -o pipefail', el código de salida será el de grep (distinto de 0).
# Con 'set -e' y 'set -o pipefail', el script terminará aquí si grep falla.

echo "Este mensaje no se mostrará si 'set -e' está activo y grep falla."

# La función 'limpieza' se llamará automáticamente al salir (normalmente o por error)

exit 0 # Salida explícita con éxito
```

# 13. Conceptos Avanzados

- **Redirección de Procesos (<(comando) y >(comando)):** Trata la salida o entrada de un comando como si fuera un archivo. Útil para comparar salidas de comandos con diff <(comando1) <(comando2).
- **Here Documents (<<DELIMITADOR):** Proporcionan múltiples líneas de entrada a un comando directamente en el script.
- **Here Strings (<<< "cadena"):** Proporcionan una sola cadena como entrada a un comando (Bash).
- **Debugging (set -x):** Imprime cada comando que se va a ejecutar (con variables expandidas) en la salida de error. Muy útil para depurar. Se desactiva con set +x. También se puede lanzar el script con bash -x mi_script.sh.
- **Señales y kill:** Enviar señales a procesos (ej. kill -TERM $pid, kill -KILL $pid).
- **Subshells ((...)):** Los comandos dentro de paréntesis se ejecutan en una subshell. Los cambios en variables o directorio dentro de (...) no afectan a la shell padre.

```
#!/bin/bash
# Ejemplo de Here Document y Here String

# Here Document para pasar texto a 'cat'
cat << EOF
Este es un texto
de múltiples líneas
que se pasará como entrada estándar.
La variable HOME es: $HOME
EOF
# EOF es el delimitador, puede ser cualquier palabra

# Here String para pasar una cadena a 'wc' (contar palabras)
wc -w <<< "Esta cadena tiene cinco palabras"

# Comparar la salida de dos comandos usando redirección de procesos
echo "--- Comparando directorios (simulado) ---"
```

```
# diff <(ls -1 /etc) <(ls -1 /usr/bin) # Ejemplo real (puede ser largo)
diff <(echo -e "a\nb\nc") <(echo -e "a\nx\nc")

# Ejemplo de subshell
echo "Directorio actual: $(pwd)"
(cd /tmp && echo "Dentro de la subshell: $(pwd)") # Cambia de directorio solo en la subshell
echo "Directorio después de la subshell: $(pwd)" # Sigue siendo el original
```

# 14. Buenas Prácticas

- **Comentarios:** Usa # para explicar qué hace tu código, especialmente las partes complejas.
- **Shebang:** Inclúyelo siempre.
- **Permisos:** Asegúrate de que sean los correctos.
- **Comillas:** Usa comillas dobles (") alrededor de las expansiones de variables ("$variable", "$@", "$(comando)") para manejar correctamente espacios y caracteres especiales. Evita usar variables sin comillas.
- **Llaves en Variables:** Usa ${variable} en lugar de $variable para evitar ambigüedades.
- **set -e -u -o pipefail:** Considera usarlos al inicio para scripts más robustos.
- **Comprobación de Errores:** Verifica el código de salida ($?) de comandos críticos si no usas set -e.
- **Funciones:** Úsalas para organizar y reutilizar código. Declara variables locales con local.
- **Nombres de Variables:** Usa nombres descriptivos en minúsculas o snake_case (por convención, las mayúsculas se reservan para variables de entorno).
- **Portabilidad:** Si necesitas que tu script funcione en diferentes shells (no solo Bash) o sistemas Unix más antiguos, cíñete a las características POSIX (ej. usa [ ] en lugar de [[ ]], expr o $((..)) en lugar de (()) para aritmética si es necesario, evita arrays asociativos y expansiones específicas de Bash).
- **Herramientas Externas:** No reinventes la rueda. Usa herramientas potentes como grep, sed, awk, find, jq (para JSON) cuando sea apropiado. Bash es bueno para "pegar" comandos, no tanto para manipulación compleja de texto o datos.
- **Idempotencia:** Si es posible, haz que tu script se pueda ejecutar múltiples veces sin causar efectos secundarios no deseados.

# 15. Conclusión

El shell scripting en Bash es una habilidad increíblemente útil para la automatización de tareas, administración de sistemas y desarrollo. Aunque su sintaxis puede parecer peculiar al principio, dominar sus fundamentos (variables, condiciones, bucles, funciones, manejo de errores) y conocer las herramientas estándar de Unix te permitirá escribir scripts potentes y eficientes. ¡La práctica constante es la clave!