Índice

INTRODUCCIÓN

Shell, terminal y Bash

Son tres cosas distintas que se confunden constantemente.

	Qué es
Shell	El programa que **interpreta** las órdenes que escribes y se las pide al sistema. Ejemplos: `bash`, `sh`, `zsh`, `fish`, `powershell`.
Terminal	La **ventana** donde escribes. No entiende ningún comando: solo muestra texto y recoge lo que tecleas. Ejemplos: la terminal de Ubuntu, Windows Terminal, iTerm, Warp.
Bash	Una shell concreta: *Bourne-again shell*. Es la más extendida en Linux y en servidores.

La analogía: la terminal es el teléfono y la shell es la persona que está al otro lado y entiende lo que le dices.

¿Por qué Bash?

- Es **el estándar de facto** en Linux y en la inmensa mayoría de servidores.
- Casi todos los scripts de administración que te vas a encontrar están escritos en Bash.
- Es muy **portable**: lo que aprendas aquí te sirve en cualquier Linux, en macOS y en un servidor remoto por SSH.
- Tiene **muchísima documentación** y una comunidad enorme, así que cualquier duda que tengas ya la ha tenido alguien antes.

> **Nota**
>
> En las últimas versiones de macOS la shell por defecto es `zsh`, no `bash`. A menudo se dice que zsh «es una evolución de bash», pero no es exacto: son dos shells hermanas, desarrolladas por separado a partir de la misma antecesora (la *Bourne shell*). Comparten casi toda la sintaxis, así que este curso te vale igual.

El entorno de este curso

Todos los ejemplos están pensados para **Linux (Ubuntu)**. Si usas Windows, la forma cómoda de tener un Linux de verdad es **WSL** (*Windows Subsystem for Linux*), que es lo que se usa en este curso.

Sistema	Qué usar
Linux	La terminal que ya tienes
macOS	La app Terminal. Escribe `bash` para cambiar de zsh a Bash
Windows	**WSL con Ubuntu**

Para instalar WSL, en PowerShell como administrador:

```
wsl --install -d Ubuntu
```

Después reinicia, abre «Ubuntu» en el menú de inicio y ya tienes una terminal Linux completa.

> **Aviso**
>
> **Git Bash no vale** para seguir el curso entero. Es cómodo para cuatro comandos, pero no trae `man`, y sobre todo **no aplica los permisos de Unix**: un `chmod 750` se queda en otra cosa, así que el capítulo de permisos te dará resultados falsos.

Cómo está organizado

Cada capítulo termina con **ejercicios**. Están puestos para que los hagas tú: no busques la solución antes de haberte peleado un rato con el comando, porque la parte que se queda es justo esa.

Cuando algo no funcione, antes de buscar en internet prueba con `--help` o con `man`, que están explicados en el capítulo siguiente. Aprender a leer la ayuda del propio sistema vale más que cualquier chuleta.

BASH PRIMEROS PASOS

Hola mundo

Primer comando en Bash: echo imprime texto en pantalla.

```
echo "Hola, BASH"
```

Para saber qué shell estás usando:

```
ps -p $$ -o comm= # bash
```

> **Aviso**
>
> Verás a menudo `echo $SHELL` para esto, pero **no es lo mismo**. `$SHELL` guarda la shell que tienes configurada como predeterminada al iniciar sesión, no la que estás ejecutando ahora mismo. Si tu shell por defecto es zsh y escribes `bash` para entrar en Bash, `$SHELL` seguirá diciendo zsh.
>
> `$$` es el número (PID) de la shell actual, así que `ps -p $$` pregunta directamente por el proceso en el que estás.

Saber dónde estás

- `pwd` (*print working directory*) dice en qué directorio estás.
- `ls` (*list*) muestra lo que hay en él.

Con opciones:

- `ls -l` en formato largo: permisos, propietario, tamaño y fecha.
- `ls -a` incluye los archivos **ocultos**, que en Linux son los que empiezan por punto (`.bashrc`).
- `ls -lh` como `-l`, pero con el tamaño en unidades legibles (`4,0K`, `2,3M`) en vez de en bytes.
- `ls -la` las dos cosas: largo y con ocultos.

Moverse: cd

`cd` (*change directory*) cambia de directorio.

Comando	A dónde va
`cd carpeta`	Entra en esa carpeta (tiene que estar aquí dentro)
`cd carpeta/otra/otra`	Baja varios niveles de una vez
`cd ..`	**Sube** un nivel, al directorio que contiene a este
`cd ../..`	Sube dos niveles
`cd ~`	A tu directorio personal (*home*)
`cd` a secas	Lo mismo que `cd ~`
`cd -`	Al **último** directorio en el que estuviste
`cd /`	A la raíz del sistema

> **Consejo**
>
> `cd -` es de los atajos más cómodos que hay: alterna entre dos directorios como el botón «atrás» del navegador. Si estás yendo y viniendo entre dos carpetas lejanas, te ahorra escribir la ruta entera cada vez.

El `.` (un punto) significa «el directorio actual» y el `..` (dos puntos) «el directorio padre». Los verás por todas partes, no solo con `cd`.

El TAB es tu mejor amigo

Esto no es un detalle menor, es **la costumbre que más tiempo te va a ahorrar**. La tecla **TAB** completa automáticamente lo que estés escribiendo: nombres de comandos, de archivos y de directorios.

```
cd Docu      # pulsa TAB aquí y se completa solo: cd Documentos/
```

- Si solo hay **una** posibilidad, la completa entera.
- Si hay **varias**, pulsa TAB **dos veces** y te las lista todas.

Además de ahorrar tiempo, evita erratas: si TAB no completa, es que ese archivo no se llama como tú creías, o no estás donde creías.

Atajos de teclado imprescindibles

La terminal tiene atajos que se usan constantemente. Estos son los que merece la pena memorizar desde el primer día:

Atajo	Qué hace
`Ctrl + C`	**Corta** el comando que se está ejecutando
`Ctrl + L`	Limpia la pantalla (igual que `clear`, pero más rápido)
`Ctrl + R`	**Busca** en el historial: empieza a escribir y va encontrando
`Ctrl + A`	Lleva el cursor al **principio** de la línea
`Ctrl + E`	Lleva el cursor al **final** de la línea
`Ctrl + U`	Borra desde el cursor hasta el principio
`Ctrl + K`	Borra desde el cursor hasta el final
`Ctrl + W`	Borra la **palabra** anterior
`Ctrl + D`	Cierra la sesión (equivale a `exit`)
`↑` / `↓`	Recorre los comandos anteriores

> **Consejo**
> Si te quedas con uno, que sea `Ctrl + R`. Pulsa `Ctrl + R`, escribe un trozo de un comando que usaste hace días y aparece entero. Vuelve a pulsar `Ctrl + R` para ir viendo coincidencias anteriores, y Enter para ejecutarlo.

Ruta absoluta y relativa

La ruta **relativa** hace referencia al directorio actual (pwd). No empieza por / y es más corta y práctica cuando sabes dónde estás.

La ruta absoluta indica la ubicación completa de un archivo o directorio en el sistema de archivos del sistema operativo. Siempre empieza por / para representar la raíz del sistema y no depende de dónde estés en ese momento.

Por ejemplo, este comando te llevaría a la ruta indicada independientemente de dónde te encuentres.

```
cd /home/user/Docs
```

Otros comandos básicos

Comando	Qué muestra
`whoami`	Tu nombre de usuario
`date`	La fecha y la hora actuales
`cal`	El calendario del mes
`uptime`	Cuánto lleva encendido el sistema
`hostname`	El nombre del equipo
`uname -a`	Información del kernel y del sistema
`clear`	Limpia la pantalla

> **Aviso**
>
> `clear` **no borra nada**: solo desplaza el contenido fuera de la vista. Puedes seguir subiendo con la rueda del ratón para ver lo anterior.

Anatomía de un comando

Todos los comandos siguen la misma estructura:

```
comando [opciones] [argumentos]
ls -l Documentos
│   │  └─ argumento: sobre qué actúa
│   └──── opción: cómo se comporta
└──────── comando: qué se ejecuta
```

- El **comando** es lo que quieres ejecutar (`ls`).
- Las **opciones** cambian su comportamiento y suelen empezar por guion (`-l`). Las cortas se pueden juntar: `-l -a` es lo mismo que `-la`.
- Los **argumentos** son los datos sobre los que actúa (un archivo, una carpeta).

Muchas opciones tienen una forma corta y otra larga que hacen lo mismo: `-a` y `--all`, `-h` y `--human-readable`. La larga se entiende mejor cuando la lees dentro de un script.

Ayuda y documentación

Un gran número de comandos y clientes de la terminal soportan la opción –help o -h para mostrar la ayuda.

```
ls --help
python --help
python -h
```

El comando `man` abre el manual de usuario completo de un comando.

```
man ls
```

> **Aviso**
>
> En **Git Bash para Windows** `man` no está disponible, porque no incluye las páginas del manual. En WSL/Ubuntu funciona con normalidad; si te dice que no encuentra la página, instálalas con `sudo apt install man-db`.
>
> **Consejo**
>
> Para salir del manual se pulsa `q`. Dentro se navega con las flechas o la barra espaciadora, y se busca escribiendo `/palabra` y pulsando Enter.

La diferencia entre las dos ayudas: `--help` te da un resumen rápido en pantalla, y `man` abre el manual completo, con todas las opciones explicadas y ejemplos al final. Cuando `--help` no te aclara algo, prueba con `man`.

Ejercicios

1. En la terminal, muestra el directorio en el que estás ahora.
2. Cambia al directorio Documentos (o Documents) de tu sistema.
3. Vuelve al mismo directorio (Documentos o Documents), pero esta vez usando una ruta absoluta completa.
4. Sube un nivel en la jerarquía de directorios.
5. Lista el contenido del directorio actual en formato simple, luego largo y finalmente incluyendo archivos ocultos.
6. Consulta el manual de algún comando.
7. Consulta la ayuda de algún comando.

8. Muestra tu nombre de usuario, la fecha y la hora actuales y el calendario de este mes.
9. Regresa al directorio donde comenzaste en el primer ejercicio.
10. Limpia la pantalla.
11. Escribe `cd Doc` y pulsa TAB. ¿Qué pasa? Ahora escribe solo `cd D` y pulsa TAB dos veces.
12. Muévete a dos carpetas alejadas entre sí y usa `cd -` para ir y venir entre ellas.
13. Pulsa `Ctrl + R` y escribe `ls`. Localiza un `ls` que hayas escrito antes y ejecútalo desde ahí.
14. Escribe un comando largo sin ejecutarlo y practica: `Ctrl + A` para ir al principio, `Ctrl + E` al final y `Ctrl + U` para borrarlo entero.

GESTIÓN DE ARCHIVOS

Sistema de archivos Unix

Los sistemas de archivos Unix están organizados por un único árbol jerárquico que empieza por /, llamado raíz.

Directorios por defecto más típicos

- / Raíz del sistema.
- /home Directorios personales de los usuarios.
- /etc Archivos de configuración del sistema.
- /bin Programas básicos.
- /usr Programas del usuario.
- /var Datos variables del sistema (registros, logs, colas…).
- /tmp Archivos temporales.

Puedes **explorar** un directorio sin encontrarte en él haciendo referencia a su ruta absoluta o relativa.

Manipulación

Creación de un archivo

- touch nombre_archivo Crea un nuevo archivo en el directorio actual.

Creación de un directorio

- mkdir nombre_carpeta Crea un nuevo directorio en el directorio actual.
- mkdir dir/nombre_carpeta Crea un nuevo directorio en el directorio seleccionado.

Eliminación de un directorio vacío

- rmdir nombre_carpeta Elimina un directorio vacío (solo funciona si la carpeta está vacía).

Copia de un archivo o directorio

- cp nombre_archivo copia_archivo Copia un archivo a otro en el directorio (como siempre, puede definirse otro directorio de destino).
- cp -r nombre_carpeta nombre_carpeta_copia Copia recursiva de todos los archivos y subdirectorios (no preserva atributos especiales como permisos, propietarios, marcas de tiempo o enlaces simbólicos). Se usa cuando solo quieres el contenido, no una copia exacta.
- cp -a nombre_carpeta nombre_carpeta_copia Copia recursiva exacta.

Movimiento o renombramiento de un archivo o directorio

- mv nombre_archivo dir Mueve un archivo a un directorio.
- mv nombre_carpeta dir Mueve un directorio a otro.
- mv nombre_carpeta_o_archivo nuevo_nombre Renombra el directorio o archivo.

Eliminación de archivos o directorios

- rm nombre_archivo Elimina un archivo.
- rm -r nombre_carpeta Elimina un directorio y todo su contenido de manera recursiva.
- rm -ri nombre_carpeta Modo de eliminación recursiva con confirmación interactiva.

> **Precaución**
>
> El comando `rm` **no envía nada a la papelera**. Lo que borras, borrado queda. Cuidado con lo que escribes.
>
> La opción `-f` (*force*) de `rm -rf` es especialmente peligrosa: no pide confirmación y ni siquiera avisa si el directorio no existía. Antes de lanzar un `rm` con comodines, prueba el mismo patrón con `ls` para ver exactamente qué vas a borrar.

Comodines

Los comodines permiten actuar sobre **muchos archivos a la vez** sin escribirlos uno a uno. Funcionan con cualquier comando, no solo con `ls`.

Comodín	Significa
`*`	Cero o más caracteres cualesquiera
`?`	**Exactamente un** carácter
`[abc]`	Un carácter que sea `a`, `b` o `c`
`[0-9]`	Un carácter que sea un dígito
`[!a]`	Un carácter que **no** sea `a`

Ejemplos:

```
ls *.md              # todos los .md
ls 03*               # los que empiezan por 03
ls 03*.txt           # empiezan por 03 y acaban en .txt
ls ?????*            # los que tienen 5 caracteres o más
ls informe[0-9].txt  # informe1.txt, informe2.txt... pero no informe10.txt
rm ?.txt             # los de un solo carácter de nombre: a.txt, 1.txt
```

> **Precaución**
>
> Antes de un `rm` con comodines, **lanza el mismo patrón con `ls`**. Es un segundo de trabajo y te enseña exactamente qué vas a borrar:

```
ls a????         # primero miro
rm a????         # y si es lo que quiero, ahora sí
```

Expansión de llaves

Las llaves no son un comodín: no buscan archivos existentes, sino que **generan texto**. Son comodísimas para crear cosas:

```
touch nota{1..5}.txt         # crea nota1.txt ... nota5.txt
mkdir -p proyecto/{src,doc,test}   # crea las tres carpetas de golpe
cp notas.txt notas.txt.bak         # lo de siempre...
cp notas.txt{,.bak}                # ...abreviado
```

Buscar archivos

`tree` muestra la estructura en forma de árbol:

```
tree            # el árbol desde aquí
tree -a         # incluyendo los ocultos
tree -L 2       # solo dos niveles de profundidad
```

`find` busca archivos por lo que le digas. Es más potente de lo que parece:

```
find . -name "notas.txt"      # por nombre exacto, desde aquí
find . -iname "*.LOG"         # sin distinguir mayúsculas
find . -type d                # solo directorios
find . -type f -name "*.sh"   # solo archivos, y que acaben en .sh
find . -size +10M             # los que ocupan más de 10 MB
find . -mtime -7              # modificados en los últimos 7 días
```

Y puede **ejecutar algo** sobre lo que encuentra:

```
find . -name "*.tmp" -delete            # borrarlos
find . -name "*.sh" -exec chmod +x {} \;   # darles permiso de ejecución
```

En `-exec`, las llaves `{}` son «cada archivo encontrado», y el `\;` marca dónde acaba el comando.

> **Precaución**
> `find` con `-delete` o `-exec rm` borra sin preguntar. Ejecuta **siempre primero la búsqueda a secas** para ver la lista, y solo después añade la acción.

Comprimir y descomprimir

Un archivo comprimido junta muchos archivos en uno y ocupa menos. En Linux lo habitual es `tar`:

```
# Comprimir una carpeta
tar -czf copia.tar.gz mi_carpeta

# Ver qué hay dentro sin extraer nada
tar -tzf copia.tar.gz

# Extraer
tar -xzf copia.tar.gz
```

Las letras de `tar` se entienden mejor sabiendo qué significan:

Letra	Significa
`c`	*create*: crear un archivo nuevo
`x`	*extract*: extraer
`t`	*list*: ver el contenido

Letra	Significa
`z`	comprimir con gzip (el `.gz` del nombre)
`f`	el siguiente argumento es el **nombre del archivo**

También existe `zip`, más habitual si vas a compartir con alguien que use Windows:

```
zip -r copia.zip mi_carpeta
unzip copia.zip
```

Enlaces simbólicos

Un enlace simbólico es un **acceso directo**: un archivo que apunta a otro sitio.

```
ln -s /ruta/muy/larga/al/archivo.txt atajo.txt
```

Al listarlo con `ls -l` se reconoce por la `l` del principio y la flecha:

```
lrwxrwxrwx 1 nica nica 9 Jul 30 15:10 atajo.txt -> archivo.txt
```

Si borras el enlace **no borras el original**. Pero si borras el original, el enlace se queda apuntando al vacío (*enlace roto*).

Ejercicios

1. Crea un directorio.
2. Elimina el directorio que acabas de crear.
3. Copia un archivo en el directorio actual y fuera de Este.
4. Mueve un archivo del directorio actual.
5. Cambia el nombre del archivo que acabas de mover.
6. Lista todos los archivos de un tipo usando un comodín.
7. Elimina un directorio de manera recursiva (cuidado con lo que vas a borrar).
8. Elimina todos los archivos de un mismo tipo (cuidado con lo que vas a borrar).
9. Utiliza el comando tree.
10. Busca un archivo concreto en el directorio actual utilizando find.

COMANDOS AVANZADOS

Lectura de archivos

- `cat archivo` muestra todo el contenido de golpe. Útil para archivos pequeños; con uno grande te llenará la pantalla.
- `less archivo` muestra el contenido **paginado**, que es lo que quieres para archivos largos. Se sale pulsando `q`.
- `more archivo` es como `less`, pero con menos posibilidades (por ejemplo, no deja volver hacia atrás). Hoy se usa `less`.
- `head archivo` muestra las **primeras** 10 líneas.
- `head -n 20 archivo` para elegir cuántas.
- `tail archivo` muestra las **últimas** 10 líneas.
- `tail -n 20 archivo` para elegir cuántas.
- `tail -f registro.log` deja el archivo abierto y va **mostrando lo nuevo según se escribe**. Es la forma habitual de vigilar un log en directo. Se corta con `Ctrl + C`.

> **Consejo**
> Para desplazarte por la paginación es habitual usar flechas, barra espaciadora, scroll o PgUp/PgDown.

Búsqueda: grep

`grep` busca un texto dentro de archivos, o dentro de lo que le llegue de otro comando. **Devuelve las líneas enteras** en las que aparece.

```
grep "error" registro.log        # líneas que contienen "error"
```

Las opciones que más se usan:

Opción	Qué hace
`-i`	No distingue mayúsculas de minúsculas
`-r`	Busca dentro de todo un directorio, recursivamente
`-n`	Muestra el número de línea de cada resultado
`-v`	**Al revés**: muestra las líneas que NO contienen el texto
`-c`	En vez de las líneas, cuenta cuántas hay
`-l`	Solo dice en qué archivos aparece, no muestra las líneas

Opción	Qué hace
`-w`	Solo la palabra entera (`grep -w red` no encuentra «redes»)

Y se combinan como cualquier otra opción:

```
grep -rn "TODO" .            # dónde y en qué línea, por todo el proyecto
grep -ic "error" *.log       # cuántos errores por archivo, sin distinguir
                             # mayúsculas
grep -v "^#" config.txt      # el archivo sin sus líneas de comentario
```

> **Consejo**
>
> Ese `^#` del último ejemplo es una **expresión regular**: el `^` significa «al principio de la línea». Con `grep` bastan tres símbolos para el 90% de los casos:

- `^` principio de línea
- `$` final de línea
- `.` un carácter cualquiera

Recuento: wc

`wc` (*word count*) cuenta lo que le eches.

Opción	Cuenta
(ninguna)	líneas, palabras y bytes
`-l`	líneas
`-w`	palabras
`-c`	bytes
`-m`	caracteres (distinto de bytes si hay tildes o eñes)

```
wc -l notas.txt              # cuántas líneas tiene
wc -lw notas.txt             # líneas y palabras
```

Los tres canales de un comando

Antes de redirigir nada hay que saber **qué** se redirige. Todo comando en Linux tiene tres canales abiertos, y cada uno tiene un número:

Nº	Nombre	Qué es
0	*stdin*	Por donde **entra** la información
1	*stdout*	Por donde **sale** el resultado normal
2	*stderr*	Por donde salen los **mensajes de error**

La clave está en que **el resultado y los errores salen por sitios distintos**, aunque los veas mezclados en pantalla. Eso es justo lo que permite separarlos:

```
ls /carpeta_que_no_existe > salida.txt
```

El archivo `salida.txt` queda **vacío** y el error sigue apareciendo en pantalla. Porque `>` redirige solo el canal 1, y el error va por el 2.

Redirecciones

Símbolo	Qué hace
`>`	Manda la salida a un archivo, **sobrescribiéndolo**
`>>`	Manda la salida a un archivo, **añadiendo al final**
`<`	Coge la entrada de un archivo
`2>`	Manda **solo los errores** a un archivo
`2>&1`	Manda los errores **al mismo sitio que la salida**
`&>`	Manda las dos cosas a la vez (atajo de `> archivo 2>&1`)

```
echo "texto" > notas.txt          # crea o sobrescribe notas.txt
echo "más texto" >> notas.txt     # añade una línea al final
sort < notas.txt                  # ordena el contenido del archivo

ls /noexiste 2> errores.txt       # solo el error va al archivo
ls /noexiste &> todo.txt          # salida y error, los dos al archivo
```

> **Precaución**
>
> `>` **borra el archivo antes de escribir**, sin avisar. Si te equivocas y escribes `>` donde querías `>>`, te has cargado el contenido anterior. Cuando dudes, usa `>>`.

El agujero negro: /dev/null

`/dev/null` es un archivo especial que **se traga todo lo que le mandes**. Sirve para silenciar lo que no te interesa:

```
comando 2>/dev/null     # ejecuta y no me enseñes los errores
comando >/dev/null      # ejecútalo y no me enseñes el resultado
comando &>/dev/null     # ejecútalo en silencio absoluto
```

> **Aviso**
>
> Ojo con el orden en `2>&1`: significa «manda el canal 2 **donde está ahora** el canal 1». Por eso va **después** de la redirección:

- `comando > archivo 2>&1` guarda las dos cosas en el archivo. Correcto.
- `comando 2>&1 > archivo` NO hace lo mismo: cuando se lee `2>&1` el canal 1 todavía apunta a la pantalla, así que los errores se quedan en pantalla y solo la salida va al archivo.

Pipes

El símbolo `|` conecta la **salida de un comando con la entrada del siguiente**, sin pasar por ningún archivo. Es lo que hace de Bash una herramienta potente: comandos pequeños que se encadenan.

```
cat notas.txt | grep "bash" | wc -l
```

Se lee de izquierda a derecha: muestra el archivo, quédate con las líneas que contienen «bash», y cuenta cuántas son.

Algunos encadenados que se usan a diario:

```
ls -l | grep "^d"                    # solo los directorios
history | grep git                   # qué comandos de git he usado
ps aux | grep node                   # procesos de node en marcha
cat registro.log | tail -n 50        # las últimas 50 líneas
```

> **Consejo**
> Los pipes solo conectan el canal 1. Si quieres que también pasen los errores, usa `|&` (o `2>&1 |`).

Variables de entorno

Puedes definir variables temporalmente (locales) o que existan en todo momento (globales).

Variable de shell

Vive **solo en la terminal donde la creaste**. Si desde ahí lanzas otro programa, ese programa no la ve.

- nombre_variable=“valor asociado a la variable” Almacena el valor de un texto en una variable.
- echo $nombre_variable Muestra el valor de la variable.

Variable de entorno

Se crea con `export`. La diferencia es que **la heredan los programas que lances desde esa terminal**: pasa a formar parte de su *entorno*.

> **Aviso**
> «Exportar» una variable **no la hace permanente**. Cuando cierras la terminal desaparece igual que las demás. Lo único que consigues con `export` es que los programas hijos puedan verla.

> Para que exista siempre hay que escribirla en un archivo de configuración, como se explica un poco más abajo.

Se ve muy claro con un ejemplo:

```
saludo="hola"           # variable de shell
export despedida="adiós"  # variable de entorno

# bash -c lanza una terminal hija:
bash -c 'echo "saludo=[$saludo] despedida=[$despedida]"'
# saludo=[] despedida=[adiós]
```

La hija no ve `saludo`, pero sí `despedida`.

Algunas variables de entorno que ya existen

```
echo $HOME     # /home/nica     -> tu directorio personal
echo $USER     # nica           -> tu nombre de usuario
echo $PWD      # /home/nica     -> el directorio en el que estás
echo $PATH     # /usr/local/bin:/usr/bin:/bin: ...
```

`$PATH` merece una parada. Es **la lista de carpetas donde Bash busca los programas**, separadas por dos puntos. Cuando escribes `ls`, Bash recorre esas carpetas por orden hasta encontrar un ejecutable llamado `ls`.

Por eso, para lanzar un script tuyo que está en el directorio actual hay que escribir `./script.sh` y no `script.sh` a secas: el directorio actual **no está** en el `PATH`, y si lo estuviera sería un problema de seguridad (bastaría con dejar un archivo llamado `ls` en una carpeta para engañarte).

Crear una variable de entorno

```
export NOMBRE_VARIABLE="su valor"
```

Que sobreviva al cierre de la terminal

Hay que escribir esa línea en un archivo de configuración, que Bash lee solo al arrancar. Verás mencionados cuatro:

Archivo	Cuándo se lee
`~/.bashrc`	Cada vez que abres una terminal normal
`~/.bash_profile`	Solo al iniciar sesión (por ejemplo, por SSH)
`~/.bash_login`	Alternativa al anterior, si aquel no existe
`~/.profile`	El más antiguo; lo leen también otras shells

> **Consejo**
>
> No te compliques: **usa** `~/.bashrc`. Es el que se lee al abrir una terminal, que es el 95% de los casos. Los otros tres existen por razones históricas.

```
# Añadir la línea al final del archivo
echo 'export MI_VARIABLE="hola"' >> ~/.bashrc

# Aplicar los cambios sin cerrar la terminal
source ~/.bashrc
```

`source` (o su atajo `.`) hace que Bash **relea** el archivo en la sesión actual. Si no, tendrías que cerrar y volver a abrir la terminal.

Ejercicios

1. Muestra todo el contenido de un archivo.
2. Muestra el contenido paginado de un archivo.
3. Muestra las 15 primeras líneas de un archivo.
4. Muestra las 15 últimas líneas de un archivo.
5. Busca una palabra en un archivo.
6. Cuenta las líneas de un archivo.
7. Redirige una salida y guárdala en un archivo.
8. Añade una nueva salida al archivo anterior.
9. Encadena 3 comandos con pipes.
10. Crea una variable de shell y muéstrala.
11. Lanza `ls` sobre una carpeta que no exista redirigiendo con `>` a un archivo. ¿Qué contiene el archivo? ¿Dónde ha ido a parar el mensaje de error? Repítelo con `2>` y luego con `&>`.
12. Busca una palabra en un archivo mostrando el número de línea, y después al revés: muestra las líneas que **no** la contienen.
13. Cuenta cuántas veces aparece una palabra en un archivo, encadenando `grep` y `wc`.
14. Muestra tu `$PATH` y cuenta cuántas carpetas tiene (pista: están separadas por `:`, y `tr ':' '\n'` las pone una por línea).

EDITORES BÁSICOS

Tarde o temprano hay que editar un archivo sin salir de la terminal: un script, un archivo de configuración, el `crontab`. Para eso están estos dos editores, que vienen en casi cualquier Linux.

nano

Es el fácil. Se abre, escribes, guardas y sales.

```
nano notas.txt      # lo abre; si no existe, lo crea
```

Puedes empezar a escribir directamente y moverte con las flechas, como en cualquier editor. **Los atajos aparecen listados abajo**, en las dos últimas líneas de la pantalla, así que no hace falta memorizarlos.

En esa lista, el símbolo `^` significa **Ctrl**. Es decir, `^O` es `Ctrl + O`.

Atajo	Qué hace
`Ctrl + O`	Guardar (luego Enter para confirmar el nombre)
`Ctrl + X`	Salir (si hay cambios sin guardar, pregunta)
`Ctrl + W`	Buscar texto
`Ctrl + K`	Cortar la línea
`Ctrl + U`	Pegar la línea
`Ctrl + G`	Ayuda

Si no lo tuvieras instalado:

```
sudo apt install nano      # Ubuntu / WSL / Debian
brew install nano          # macOS
```

Web de nano

vim

- vim nombre_archivo abre el archivo (si no existe, lo crea).
- Editor más avanzado, potente y personalizable, pero con una curva de aprendizaje pronunciada.

Los modos: la clave para entender vim

Lo que desconcierta de vim es que **tiene modos**. En un editor normal, si pulsas una tecla, se escribe esa letra. En vim depende de en qué modo estés:

Modo	Para qué sirve	Cómo se entra
Normal	Moverse y dar órdenes. Es el modo inicial	`Esc`
Inserción	Escribir texto, como en cualquier editor	`i`
Comando	Guardar, salir, buscar	`:`

Por eso, cuando abres vim y empiezas a teclear, pasan cosas raras: estás en modo *normal* y cada letra se interpreta como una orden.

> **Consejo**
>
> Si te pierdes, **pulsa** `Esc`. Te devuelve siempre al modo normal, desde donde puedes salir con `:q!`. Con eso ya no te quedas nunca atrapado, que es el miedo clásico de la primera vez.

Lo mínimo para sobrevivir

Estando en modo **normal**:

Tecla	Qué hace
`i`	Pasar a modo inserción y empezar a escribir
`Esc`	Volver a modo normal
`h` `j` `k` `l`	Mover el cursor (izquierda, abajo, arriba, derecha)
`dd`	Borrar la línea entera
`yy`	Copiar la línea
`p`	Pegar
`u`	Deshacer
`/texto`	Buscar

Y los comandos que empiezan por `:`

Comando	Qué hace
`:w`	Guardar
`:q`	Salir
`:wq`	Guardar y salir
`:q!`	Salir **sin guardar**, descartando los cambios

Las flechas también funcionan para moverse. El `hjkl` existe porque permite no apartar las manos de la fila central del teclado, pero al principio no te pelees con eso.

Web de vim

> **Consejo**
> vim exige tiempo para aprender como funciona en profundidad, pero, una vez dominado, es extremadamente versátil.

Otros editores (Neovim / Emacs)

> **Importante**
> vim es frecuentemente utilizado como editor de código/scripting. Otras opciones son Neovim o Emacs.

Ejercicios

nano

1. Crea un archivo llamado nota.txt y escribe tres líneas de texto.
2. Abre nota.txt, añade una línea al final y guarda los cambios.
3. Abre un archivo nuevo llamado recordatorio.txt, escribe algo y sal sin guardar.
4. Busca una palabra específica en un archivo existente.
5. Corta una línea y pégala en otra parte.

vim

6. Crea apuntes.txt, entra en modo inserción y escribe una frase.
7. Mueve el cursor sin usar las flechas.
8. Borra una línea completa y deshaz el cambio.
9. Copia una línea y pégala debajo.
10. Guarda los cambios y sal.

ADMINISTRACIÓN DEL SISTEMA

En Unix y Linux, cada archivo y cada directorio lleva pegada una lista de permisos que dice **quién puede leerlo, modificarlo o ejecutarlo**. Es la base de la seguridad del sistema, y también el motivo de la mitad de los «permission denied» que te vas a encontrar.

Tipos de permiso

Hay tres permisos, y cada uno se puede escribir de dos formas: con una letra (modo **simbólico**) o con un número (modo **octal**).

Permiso	Letra	Número
Lectura	`r` (*read*)	4
Escritura	`w` (*write*)	2
Ejecución	`x` (*execute*)	1

Los números no son arbitrarios: son 4, 2 y 1 precisamente para que **cada combinación dé una suma distinta**. Así, un solo dígito del 0 al 7 describe sin ambigüedad cualquier mezcla de los tres permisos.

Tipos de usuario

Los permisos se dan por separado a tres grupos de gente:

Letra	Quién es
`u`	*user*: el propietario del archivo
`g`	*group*: el grupo al que pertenece el archivo
`o`	*others*: todos los demás
`a`	*all*: los tres anteriores a la vez

Ver permisos

- `ls -l nombre_archivo` para un archivo.
- `ls -ld nombre_directorio` para un directorio.

> **Consejo**
>
> Fíjate en la `d` de `ls -ld`. Sin ella, `ls -l` sobre un directorio te muestra **lo que hay dentro**, no el directorio en sí. La `d` le dice «no entres, háblame de la carpeta».

Cómo se lee una línea de ls -l

```
-rw-r--r-- 1 nica nica 5 Jul 30 15:10 notas.txt
```

De izquierda a derecha:

Trozo	Qué es
`-rw-r--r--`	Tipo de archivo y permisos
`1`	Número de enlaces
`nica`	Propietario

Trozo	Qué es
`nica`	Grupo
`5`	Tamaño en bytes
`Jul 30 15:10`	Fecha de última modificación
`notas.txt`	Nombre

Para ver el tamaño en unidades legibles, añade `-h`: `ls -lh`.

Anatomía de los permisos

Modo simbólico

Ese primer trozo, `-rwxrwxrwx`, son en realidad **cuatro bloques pegados**:

```
-   rwx   rwx   rwx
│    │     │     └── otros
│    │     └──────── grupo
│    └────────────── propietario
└─────────────────── tipo de archivo
```

Cada bloque de tres dice, **siempre en ese orden**, si hay permiso de lectura, de escritura y de ejecución.

> **Aviso**
> Cuando un permiso **no** está concedido, en su sitio aparece un guion. El orden `rwx` no cambia nunca, así que la posición te dice de qué permiso se trata:

- `r--` → solo lectura
- `r-x` → lectura y ejecución, pero no escritura
- `rw-` → lectura y escritura, pero no ejecución

> **Aviso**
> La ausencia de un permiso (permiso no otorgado) se representa con -.
> Ejemplos: r– indicará que solo tiene permisos de lectura, en cambio, r-x indicará que tiene permisos de lectura y ejecución. Siempre se conserva el orden rwx.

Modo octal

777

- Está formado por 3 dígitos, representando de izquierda a derecha los bloques de [usuario][grupo][otros].
- Cada número es el resultado de sumar el valor asociado al tipo de permiso:

Número	Suma	Permisos	Se ve como
7	4 + 2 + 1	lectura + escritura + ejecución	`rwx`
6	4 + 2	lectura + escritura	`rw-`
5	4 + 1	lectura + ejecución	`r-x`
4	4	solo lectura	`r--`
3	2 + 1	escritura + ejecución	`-wx`
2	2	solo escritura	`-w-`
1	1	solo ejecución	`--x`
0	—	ninguno	`---`

Así, **777** quiere decir que todo el mundo puede hacer de todo.

Y **764** significa que el *propietario* lo puede todo (4+2+1 = 7), el *grupo* puede leer y escribir (4+2 = 6), y los *demás* solo leer (4).

> **Consejo**
> Los dos valores que verás el 90% de las veces son **644** para archivos (`rw-r--r--`: el dueño edita, los demás solo leen) y **755** para directorios y scripts (`rwxr-xr-x`: el dueño lo puede todo, los demás entran y leen).

Tipos de archivo más habituales

Es el primer carácter de la línea, antes de los permisos:

Carácter	Qué es
`-`	archivo normal
`d`	directorio
`l`	enlace simbólico
`b`	dispositivo de bloque (un disco)
`c`	dispositivo de carácter (un terminal)
`s`	socket
`p`	tubería con nombre (*pipe*)

Los permisos en directorios significan otra cosa

Esta es la parte que más confunde, y casi nunca se explica: `r`, `w` y `x` **no quieren decir lo mismo en un archivo que en una carpeta**.

Permiso	En un archivo	En un directorio

Permiso	En un archivo	En un directorio
`r`	ver su contenido	**listar los nombres** que contiene
`w`	modificarlo	**crear y borrar** archivos dentro
`x`	ejecutarlo como programa	**entrar** (`cd`) y acceder a lo de dentro

Lo raro es la `x`: en una carpeta no significa «ejecutar», significa **atravesar**. Sin ella no puedes entrar ni leer nada de lo que hay dentro, aunque sepas cómo se llama.

Se ve mejor probándolo. Con un directorio que tiene `r` pero **no** `x`:

```
mkdir sinx && echo "secreto" > sinx/dato.txt
chmod 600 sinx          # rw- : lectura y escritura, pero sin x

ls sinx                 # dato.txt          <- sí veo el nombre
cat sinx/dato.txt       # Permission denied  <- pero no puedo leerlo
cd sinx                 # Permission denied  <- ni entrar
```

Y al revés, con `x` pero **sin** `r`:

```
mkdir solox && echo "dato" > solox/fichero.txt
chmod 100 solox         # --x : solo atravesar

ls solox                # Permission denied  <- no puedo listar
cat solox/fichero.txt   # dato               <- pero sí leer, si sé el nombre
```

> **Consejo**
>
> Por esto los directorios llevan **755** y los archivos **644**. Un directorio sin `x` no sirve para nada: no puedes entrar. Si alguna vez te encuentras con un «permission denied» raro al hacer `cd`, mira la `x`.

Modificación de permisos

Se usa el comando `chmod` (*change mode*).

Modo simbólico

Se dice a quién, qué se hace y qué permiso:

```
chmod [quién][+ - =][permiso] archivo
```

- `+` concede un permiso, `-` lo quita, `=` deja exactamente ese y ningún otro.

```
chmod u+x script.sh      # el propietario ya puede ejecutarlo
chmod u-x script.sh      # se lo quitamos otra vez
chmod go-w notas.txt     # grupo y otros dejan de poder escribir
chmod a+r notas.txt      # todo el mundo puede leerlo
chmod u=rw notas.txt     # el propietario tiene EXACTAMENTE rw, sin x
```

La ventaja de esta forma es que **solo toca lo que le dices**; el resto de permisos se queda como estaba.

Modo octal

Aquí se dan los tres bloques de golpe, con un dígito cada uno:

```
chmod 753 archivo    # u=rwx  g=r-x  o=-wx
chmod 642 archivo    # u=rw-  g=r--  o=-w-
chmod 644 notas.txt  # lo normal para un archivo
chmod 755 script.sh  # lo normal para un script o un directorio
```

Ojo: el modo octal **reescribe los tres bloques enteros**. No sirve para retocar un permiso suelto sin tocar los demás.

Cambiar permisos a toda una carpeta

La opción `-R` (recursiva) aplica el cambio al directorio y a todo lo que tiene dentro:

```
chmod -R 755 mi_carpeta
```

> **Precaución**
> Con `-R` es fácil hacer un estropicio. `chmod -R 777` sobre una carpeta deja que **cualquier usuario del sistema** modifique o borre todo lo que hay dentro. Es una mala idea aunque lo veas recomendado en foros para «arreglar» un permission denied: casi siempre lo que hacía falta era un `755`, o cambiar el propietario.

Cambiar propietario y grupo

Se usa `chown` (*change owner*):

```
chown nica notas.txt          # cambia el propietario
chown nica:desarrollo notas.txt   # cambia propietario y grupo
chown -R nica mi_carpeta      # a toda una carpeta
```

> **Aviso**
> `chown` casi siempre necesita permisos de administrador, así que se usa con `sudo` (lo tienes explicado justo debajo). Tiene sentido: si cualquiera pudiera regalar sus archivos a otro usuario, o quedarse con los ajenos, los permisos no servirían de nada.

El superusuario: root y sudo

Hay tareas que un usuario normal no puede hacer: instalar programas, tocar la configuración del sistema o cambiar el propietario de un archivo. Para eso existe **root**, el usuario administrador, que se salta todos los permisos.

Trabajar siempre como root es peligroso: no hay red de seguridad, y un comando mal escrito puede cargarse el sistema. Lo normal es trabajar como usuario corriente y pedir permisos de administrador **solo para el comando que lo necesita**, con `sudo`:

```
sudo apt update              # actualizar la lista de paquetes
sudo chown nica notas.txt    # cambiar el propietario de un archivo
sudo nano /etc/hosts         # editar un archivo del sistema
```

La primera vez te pedirá **tu propia contraseña** (no la de root), y durante unos minutos no te la volverá a pedir.

> **Precaución**
>
> Antes de poner `sudo` delante de algo, léelo dos veces. Con `sudo` no hay confirmaciones ni papelera: el sistema da por hecho que sabes lo que haces.
>
> Y desconfía de cualquier página que te diga que pegues un `sudo` con un comando largo que no entiendes. Es la forma más habitual de que alguien te reviente el equipo.

Para ver qué usuario eres en cada momento:

```
whoami        # nica
sudo whoami   # root
```

Máscara de permisos

La máscara son los permisos que se le darán **por defecto** a los archivos y directorios nuevos que crees.

- `umask` a secas muestra la máscara actual.
- `umask 022` la cambia.

> **Precaución**
>
> Cambiar la máscara con `umask` es **temporal**: solo afecta a la terminal en la que lo escribes. Al cerrarla, se pierde.
>
> Para que sea permanente hay que añadir la línea a tu `~/.bashrc`, igual que con las variables de entorno.

Cómo se calculan los permisos con la máscara

La máscara indica **qué permisos hay que quitar** a los permisos de partida que el sistema daría a lo nuevo que crees.

Los permisos de partida son:

- **777** para un directorio (`rwx` para todos).

- **666** para un archivo. Sin ejecución: dar permiso de ejecución por defecto a todos los archivos sería peligroso.

Si la máscara es `0022`:

- El **primer dígito** son los permisos especiales (setuid, setgid y sticky bit). Un `0` significa «ninguno activado». Los tres dígitos siguientes son los de siempre: usuario, grupo y otros.
- `0` → al usuario no le quito ningún permiso.
- `2` → al grupo le quito la escritura.
- `2` → a otros les quito la escritura.

Resultado:

- Directorios: de `777` quito `022` → **755** `[rwxr-xr-x]`
- Archivos: de `666` quito `022` → **644** `[rw-r--r--]`

> **Aviso**
>
> Verás mucho por ahí que la máscara «se resta». **No es una resta**: es quitar permisos. Con `umask 022` el resultado coincide con una resta y por eso cuela, pero se rompe en cuanto la máscara pide quitar un permiso que no estaba puesto.
>
> Pruébalo: con `umask 027`, la resta diría `6 - 7 = -1` para el bloque de «otros», que no significa nada. Lo que pasa de verdad es que a `rw-` le quitas `rwx` y te quedas sin nada, es decir `0`. El archivo sale con permisos **640**, no con un número negativo.

Compruébalo tú en la terminal:

```
# Los paréntesis lanzan una subshell: así la máscara no se te queda puesta
(umask 027; touch prueba.txt; stat -c '%a %n' prueba.txt)
# 640 prueba.txt
```

Ejercicios

1. Crea un archivo y visualiza sus permisos.
2. Otorga permisos de ejecución solo al propietario en modo simbólico.
3. Cambia sus permisos a 644.
4. Elimina los permisos para el grupo.
5. Haz que solo pueda ejecutarse por el propietario.
6. Crea una carpeta y dale permisos para que solo el usuario pueda acceder.
7. Cámbiale el propietario a otro usuario de tu sistema (si existe y tienes permisos).

8. Consulta la máscara de permisos actual y calcula qué permisos por defecto tendrán los nuevos archivos.
9. Cambia la máscara, crea un archivo y consulta los permisos por defecto del archivo.
10. Utiliza un comando como superusuario con `sudo`, y comprueba con `whoami` la diferencia entre ejecutarlo con y sin `sudo`.
11. Crea una carpeta con un archivo dentro. Quítale a la carpeta el permiso `x` y comprueba qué puedes y qué no puedes hacer: ¿ves el nombre del archivo?, ¿puedes leerlo?, ¿puedes entrar en la carpeta?
12. Mira los permisos de tu propio directorio personal con `ls -ld ~` e interpreta lo que ves: ¿quién es el propietario?, ¿puede entrar alguien más?

PROCESOS Y ALIAS

Monitorización

- ps muestra los procesos asociados a la terminal actual
- ps aux muestra la lista de todos los procesos en ejecución del sistema.
- a procesos de todos los usuarios.
- u muestra el nombre de usuario y detalles.
- x muestra procesos no asociados a la terminal.
- top monitor de procesos interactivo en tiempo real.
- htop versión mejorada de top (requiere instalación del CLI).
- free -h muestra información sobre la memoria (solo disponible en Linux).
- df -h uso de disco (disk free).
- du -sh * tamaño de todos los archivos y directorios actuales (disk usage).

Importante

```
La opción -h suele estar asociada a mostrar la información en
formato "human-readable", para facilitar su lectura.
```

> **Consejo**
>
> El **PID** es el número que identifica de forma única a cada proceso. Es lo que necesitas para poder pararlo después.

> Para salir de `top` o `htop` se pulsa `q`, o `Ctrl + C`.

Primer y segundo plano

Cuando lanzas un comando, la terminal se queda ocupada hasta que termina. Eso es el **primer plano** (*foreground*). Si el comando tarda mucho, puedes mandarlo al **segundo plano** (*background*) y seguir trabajando.

```
sleep 300 &        # el & al final lo lanza directamente en segundo plano
```

Y si ya lo has lanzado y se te ha quedado ocupada la terminal:

Acción	Cómo
Suspender lo que está corriendo	`Ctrl + Z`
Ver los trabajos de esta terminal	`jobs`
Reanudarlo en **segundo** plano	`bg %1`
Traerlo al **primer** plano	`fg %1`

El `%1` es el número que te da `jobs`, no el PID.

```
sleep 300          # se queda ocupada la terminal
# pulsas Ctrl + Z  -> [1]+  Detenido    sleep 300
jobs               # [1]+  Detenido    sleep 300
bg %1              # sigue corriendo, pero ya te devuelve el prompt
fg %1              # lo trae de vuelta al primer plano
```

> **Consejo**
>
> `sleep 300` («no hagas nada durante 300 segundos») es perfecto para practicar todo esto sin riesgo de romper nada.
>
> **Aviso**
>
> Un proceso en segundo plano **muere cuando cierras la terminal**. Si necesitas que sobreviva, se lanza con `nohup`:

```
nohup ./mi_script.sh &
```

> La salida se guarda en un archivo llamado `nohup.out`.

Terminar procesos

`kill` no significa exactamente «matar»: lo que hace es **mandar una señal** a un proceso. Las dos que importan:

Comando	Señal	Qué hace
`kill PID`	`TERM` (15)	**Pide** al proceso que termine. Este puede

Comando	Señal	Qué hace
		guardar lo que tenga, cerrar archivos y salir ordenadamente
`kill -9 PID`	`KILL` (9)	Lo **fulmina** de inmediato. No puede negarse, pero tampoco puede limpiar nada

> **Precaución**
>
> Usa siempre `kill` a secas primero, y deja `kill -9` como último recurso. Con `-9` el proceso no puede guardar su trabajo ni cerrar bien lo que tenía abierto, así que puede dejar archivos a medias o bloqueos sueltos.

Si no sabes el PID, hay atajos que buscan por nombre:

```
pkill firefox       # termina los procesos que se llamen así
killall firefox     # equivalente en la mayoría de sistemas
pgrep -l firefox    # solo mira: lista PID y nombre, sin tocar nada
```

Historial

Bash guarda todo lo que escribes, así que rara vez hace falta volver a teclear un comando largo.

Comando	Qué hace
`history`	Lista el historial numerado
`!!`	Repite el **último** comando
`!10`	Ejecuta el comando número 10 del historial
`!ls`	Repite el último que empezaba por `ls`
`Ctrl + R`	**Busca** en el historial escribiendo un trozo

> **Consejo**
>
> `sudo !!` es un clásico: ejecutas algo, te dice «permission denied», y con `sudo !!` lo repites con permisos sin volver a escribirlo.
>
> Y para el día a día, `Ctrl + R` es más cómodo que `history | grep`.
>
> **Aviso**

> Cuidado al escribir ! en una cadena de texto, ya que puede intentar realizar una referencia al historial y producir un error. Para evitarlo, puedes escapar el signo utilizando la barra invertida
> antes de !. También puedes usar comillas simples ' en vez de dobles " para crear la cadena de texto.

Alias

Un alias es un **apodo** que le pones a un comando, normalmente para no repetir opciones que usas siempre.

```
alias ll='ls -l'          # ahora "ll" hace "ls -l"
alias la='ls -la'
alias ..='cd ..'

alias                     # ver todos los alias definidos
unalias ll                # eliminar uno
```

Fíjate en que **no hay espacios alrededor del** `=`, y en que el comando va entre comillas simples si lleva espacios u opciones.

> **Precaución**
> Un alias creado así **solo vive en esa terminal**. Para tenerlo siempre, añádelo a tu `~/.bashrc`:

```
echo "alias ll='ls -l'" >> ~/.bashrc
source ~/.bashrc
```

Un uso muy recomendable es ponerle red de seguridad a los comandos peligrosos:

```
alias rm='rm -i'      # pedirá confirmación antes de borrar
alias cp='cp -i'
alias mv='mv -i'
```

> **Aviso**
> Acostumbrarse a que `rm` siempre pregunte tiene un riesgo: el día que uses otro equipo donde ese alias no existe, `rm` borrará sin avisar. Ten claro que la protección es tuya, no del sistema.

Ejercicios

1. Muestra todos los procesos del sistema.
2. Muestra el monitor interactivo de procesos.
3. Utiliza el comando free de manera correcta.

4. Lanza sleep 100 en la terminal, suspéndelo, mándalo al segundo plano y tráelo al primer plano.
5. Lanza un proceso como sleep y terminarlo usando su PID.
6. Consulta el espacio en disco.
7. Consulta el historial.
8. Repite el ultimo comando.
9. Crea y prueba un alias.
10. Elimina el alias que acabas de crear.

SCRIPTING

Definición

Un script es un archivo de texto que contiene comandos que la shell ejecutará secuencialmente para permitir así automatizar tareas repetitivas.

Convenciones en Bash

1. **Extensión** `.sh`. Es opcional (a Bash le da igual), pero ayuda a saber de un vistazo qué es cada archivo.
2. **Primera línea** `#!/bin/bash`, el llamado *shebang*. Le dice al sistema con qué intérprete tiene que ejecutar el archivo. Sin él, el sistema usa la shell que le parezca, y tu script puede comportarse distinto.
3. **Permisos de ejecución** con `chmod +x script.sh`.

Hay dos formas de lanzar un script:

```
./script.sh        # necesita el permiso de ejecución
bash script.sh     # funciona aunque no lo tenga
```

> **Consejo**
>
> Verás también `#!/usr/bin/env bash` como shebang. Es más portable: en vez de dar por hecho que Bash está en `/bin/bash`, lo busca en el `PATH`. En Linux los dos funcionan; si tus scripts van a correr también en macOS, mejor la segunda.

Cuando algo no funciona

Antes de volverte loco mirando el código, Bash trae sus propias herramientas:

```
bash -x script.sh       # muestra cada línea ya expandida antes de ejecutarla
bash -n script.sh       # solo comprueba la sintaxis, sin ejecutar nada
```

También puedes activar el modo detallado desde dentro del script, poniendo `set -x` en la línea a partir de la cual quieres ver el detalle.

> **Consejo**
>
> Y una recomendación que te ahorrará horas: instala **ShellCheck**, que revisa scripts y te avisa de los fallos típicos (comillas que faltan, variables mal escritas, comparaciones dudosas).

```
sudo apt install shellcheck
shellcheck script.sh
```

Ejemplo básico

```
#!/bin/bash

# Mi primer script
echo "Hola, este es mi primer script en Bash"
date
echo "Tu directorio actual es: $(pwd)"
```

- #!/bin/bash → shebang
- # Mi primer script → comentario (documenta, no se ejecuta)
- echo, date comandos en Bash
- "$(pwd)" interpolación de comandos

Variables

```
name="Nica"
echo "Hola $name"
```

- name="Nica" → definición de la variable
- "$name" → interpolación de la variable

Aritmética básica

Bash trata todo como texto por defecto. Para que haga cuentas hay que pedírselo con `$(( ))`:

```
a=5
b=3

suma=$((a + b))
echo "La suma es $suma"            # La suma es 8
echo "El doble de a es $((a * 2))"
```

Dentro de `$(( ))` **no hace falta el** `$` delante de las variables: `$((a + b))` funciona igual que `$(($a + $b))`.

> **Aviso**
>
> Sin `$(( ))` no hay cuenta que valga:

```
suma=a+b
echo "$suma"       # a+b   <- texto literal, no 8
```

Verás también `let suma=a+b`, que es la forma antigua. Hace lo mismo, pero **usa** `$(( ))`: es lo estándar en Bash moderno y se lee mejor.

Lectura de datos

`read` pide datos al usuario mientras el script se ejecuta.

```
#!/bin/bash

read -r -p "¿Cuál es tu nombre? " nombre
echo "Hola, $nombre"

read -r -p "¿Cuál es tu edad? " edad
echo "Tu edad es $edad"

read -r -s -p "Contraseña: " clave
echo
echo "Has escrito ${#clave} caracteres"
```

Opción	Qué hace
`-p`	Muestra el texto de la pregunta en la misma línea
`-s`	**Oculta** lo que se escribe (para contraseñas)
`-r`	No interpreta las barras invertidas como escape

> **Consejo**
>
> **Usa siempre** `read -r`. Sin la `-r`, si el usuario escribe una ruta de Windows como `C:\Users\nica`, Bash se come las barras y guarda `C:Usersnica`. Es un fallo silencioso y molesto de encontrar.
>
> No hay ningún caso práctico en el que quieras `read` sin `-r`.

Argumentos y parámetros

Los scripts pueden recibir argumentos desde la línea de comandos.

```
#!/bin/bash
echo "El nombre del script es: $0"
echo "El primer parámetro es: $1"
echo "El segundo parámetro es: $2"
echo "Número de parámetros: $#"
echo "Todos los argumentos: $@"
```

Ejecución con argumentos: ./script.sh argumento1 argumento2

Variable	Qué contiene
`$0`	El nombre del script (`./script.sh`)
`$1`, `$2`, `$3`…	El primer, segundo, tercer argumento…
`$#`	Cuántos argumentos se han pasado
`"$@"`	**Todos** los argumentos, cada uno por separado
`$*`	Todos los argumentos, pero pegados en una sola cadena

> **Aviso**
>
> Entre `"$@"` y `$*` hay una diferencia que importa cuando algún argumento lleva espacios. Con `"$@"` cada argumento se mantiene entero; con `$*` se juntan todos y luego se parten por los espacios.
>
> **Usa siempre `"$@"`, con comillas.** Es lo correcto en el 99% de los casos, y por el mismo motivo que se explica en el capítulo de comillas.

Un patrón que conviene coger desde el principio: **comprobar que te han pasado lo que esperabas** antes de seguir.

```
#!/bin/bash

if [ $# -lt 2 ]; then
    echo "Uso: $0 <origen> <destino>" >&2
    exit 1
fi

origen="$1"
destino="$2"
echo "Copiando de $origen a $destino"
```

- El `>&2` manda el mensaje al canal de errores, que es donde deben ir.
- `exit 1` termina el script indicando que hubo un problema.

> **Nota**
>
> Tienes el script completo un poco más arriba, en este mismo apartado.

Ejercicios

1. Crea un script que imprima en pantalla: Hola mundo desde Bash.

2. Crea un script que muestre la fecha y el directorio actual.
3. Crea un script que guarde tu nombre en una variable y lo muestre en pantalla.
4. Crea un script que declare dos variables numéricas, las sume, reste y multiplique, mostrando el resultado de cada operación.
5. Crea un script que pida tu nombre con read y lo muestre.
6. Crea un script que pida dos números al usuario y muestre su suma.
7. Crea un script con tres argumentos que muestre el primero y el tercero.
8. Crea un script con argumentos que muestre el número total.
9. Crea un script que reciba dos números como argumentos y muestre su suma, resta, multiplicación y división.
10. Crea un script que cree un archivo de texto y guarde tu nombre en su interior.

Posibles correcciones (ten en cuenta que pueden variar dependiendo de tu sistema operativo y directorios existentes)

1. Crea un script que imprima en pantalla: Hola mundo desde Bash.

```
#!/bin/bash

# Nombre: hola_mundo.sh

# Descripción: Script que imprime "Hola mundo desde Bash"
echo "Hola mundo desde Bash"
Crear y ejecutar el script:

# Crear el archivo
nano hola_mundo.sh

# Copiar el contenido del script anterior

# Guardar: Ctrl + O, Enter, Ctrl + X

# Dar permisos de ejecución
chmod +x hola_mundo.sh

# Ejecutar el script
./hola_mundo.sh

# O ejecutar sin permisos de ejecución
bash hola_mundo.sh
```

2. Crea un script que muestre la fecha y el directorio actual.

```
#!/bin/bash

# Nombre: fecha_directorio.sh

# Descripción: Muestra la fecha actual y el directorio de trabajo
echo "=== Información del Sistema ==="
echo "Fecha y hora actual:"
date
echo ""
echo "Directorio actual:"
pwd

# Alternativa con formato más personalizado
    echo ""
    echo "=== Información Detallada ==="
    echo "Fecha: $(date '+%d/%m/%Y')"
    echo "Hora: $(date '+%H:%M:%S')"
    echo "Directorio: $(pwd)"
    Crear y ejecutar el script:
    nano fecha_directorio.sh
    chmod +x fecha_directorio.sh
    ./fecha_directorio.sh
```

3. Crea un script que guarde tu nombre en una variable y lo muestre en pantalla.

```
#!/bin/bash

# Nombre: mi_nombre.sh

# Descripción: Guarda un nombre en una variable y lo muestra

# Declarar la variable
nombre="Nica"

# Mostrar el contenido de la variable
echo "Mi nombre es: $nombre"

# Alternativa con más información
echo "Hola, mi nombre es ${nombre}"
echo "Bienvenido/a, ${nombre}!"

# También puedes usar comillas simples para texto literal
echo 'El nombre guardado es:' "$nombre"
Crear y ejecutar el script:
nano mi_nombre.sh
chmod +x mi_nombre.sh
./mi_nombre.sh
```

4. Crea un script que declare dos variables numéricas, las sume, reste y multiplique, mostrando el resultado de cada operación.

```
#!/bin/bash

# Nombre: operaciones.sh

# Descripción: Realiza operaciones matemáticas básicas

# Declarar dos variables numéricas
num1=15
num2=5
echo "=== Operaciones Matemáticas ==="
echo "Primer número: $num1"
echo "Segundo número: $num2"
echo ""

# Suma
suma=$((num1 + num2))
echo "Suma: $num1 + $num2 = $suma"

# Resta
resta=$((num1 - num2))
echo "Resta: $num1 - $num2 = $resta"

# Multiplicación
# OJO: el nombre de la variable va SIN tilde. Bash solo admite letras
# sin acentos, números y guion bajo en los nombres de variable.
multiplicacion=$((num1 * num2))
echo "Multiplicación: $num1 * $num2 = $multiplicacion"

# Bonus: División y módulo
division=$((num1 / num2))
modulo=$((num1 % num2))
echo ""
echo "=== Operaciones adicionales ==="
echo "División: $num1 / $num2 = $division"
echo "Módulo: $num1 % $num2 = $modulo"
Crear y ejecutar el script:
nano operaciones.sh
chmod +x operaciones.sh
./operaciones.sh
```

5. Crea un script que pida tu nombre con read y lo muestre.

```
#!/bin/bash

# Nombre: pedir_nombre.sh

# Descripción: Pide el nombre al usuario y lo muestra

# Pedir el nombre al usuario
echo "Por favor, introduce tu nombre:"
read nombre

# Mostrar el nombre
echo "Hola, $nombre! Bienvenido/a."

# Alternativa con prompt en la misma línea

# read -p "Introduce tu nombre: " nombre

# echo "Hola, $nombre!"

# Versión más completa
echo ""
echo "Encantado de conocerte, ${nombre}!"
Crear y ejecutar el script:
nano pedir_nombre.sh
chmod +x pedir_nombre.sh
./pedir_nombre.sh
```

6. Crea un script que pida dos números al usuario y muestre su suma.

```
#!/bin/bash

# Nombre: suma_numeros.sh

# Descripción: Pide dos números al usuario y muestra su suma

# Pedir el primer número
echo "Introduce el primer número:"
read num1

# Pedir el segundo número
echo "Introduce el segundo número:"
read num2

# Calcular la suma
suma=$((num1 + num2))

# Mostrar el resultado
    echo ""
    echo "La suma de $num1 + $num2 = $suma"

    # Alternativa con read -p (prompt en la misma línea)

    # read -p "Introduce el primer número: " num1

    # read -p "Introduce el segundo número: " num2

    # suma=$((num1 + num2))

    # echo "La suma es: $suma"
    Crear y ejecutar el script:
    nano suma_numeros.sh
    chmod +x suma_numeros.sh
    ./suma_numeros.sh
```

7. Crea un script con tres argumentos que muestre el primero y el tercero.

```bash
#!/bin/bash

# Nombre: argumentos_1_3.sh

# Descripción: Muestra el primer y tercer argumento

# Verificar que se pasaron al menos 3 argumentos
if [ $# -lt 3 ]; then
    echo "Error: Se necesitan 3 argumentos"
    echo "Uso: $0 argumento1 argumento2 argumento3"
    exit 1
fi

# Mostrar el primer y tercer argumento
echo "El primer argumento es: $1"
echo "El tercer argumento es: $3"

# Información adicional
echo ""
echo "=== Información adicional ==="
echo "El segundo argumento (no mostrado) es: $2"
echo "Total de argumentos recibidos: $#"
Crear y ejecutar el script
nano argumentos_1_3.sh
chmod +x argumentos_1_3.sh
./argumentos_1_3.sh primero segundo tercero

# Ejemplo con diferentes valores
./argumentos_1_3.sh Hola Mundo Bash
```

8. Crea un script con argumentos que muestre el número total.

```bash
#!/bin/bash

# Nombre: contar_argumentos.sh

# Descripción: Muestra el número total de argumentos recibidos

# Mostrar el número total de argumentos
echo "Número total de argumentos: $#"

# Información adicional
if [ $# -eq 0 ]; then
    echo "No se recibieron argumentos"
elif [ $# -eq 1 ]; then
    echo "Se recibió 1 argumento: $1"
else
    echo "Se recibieron $# argumentos"
    echo "Todos los argumentos: $@"
fi

# Listar cada argumento
echo ""
echo "=== Lista de argumentos ==="
contador=1
for arg in "$@"; do
    echo "Argumento $contador: $arg"
    ((contador++))
done
Crear y ejecutar el script:
nano contar_argumentos.sh
chmod +x contar_argumentos.sh

# Probar con diferentes cantidades de argumentos
./contar_argumentos.sh
./contar_argumentos.sh uno
./contar_argumentos.sh uno dos tres
./contar_argumentos.sh Hola Mundo desde Bash
```

9. Crea un script que reciba dos números como argumentos y muestre su suma, resta, multiplicación y división.

```bash
#!/bin/bash

# Nombre: calculadora.sh

# Descripción: Realiza operaciones matemáticas con dos números como
argumentos

# Verificar que se recibieron 2 argumentos
if [ $# -ne 2 ]; then
    echo "Error: Se necesitan exactamente 2 números"
    echo "Uso: $0 numero1 numero2"
    exit 1
fi

# Guardar los argumentos en variables
num1=$1
num2=$2
echo "=== CALCULADORA ==="
echo "Primer número: $num1"
echo "Segundo número: $num2"
echo ""

# Realizar las operaciones
suma=$((num1 + num2))
resta=$((num1 - num2))
multiplicacion=$((num1 * num2))
echo "Suma: $num1 + $num2 = $suma"
echo "Resta: $num1 - $num2 = $resta"
echo "Multiplicación: $num1 * $num2 = $multiplicacion"

# División (con verificación de división por cero)
if [ $num2 -eq 0 ]; then
    echo "División: No se puede dividir por cero"
else
    division=$((num1 / num2))
    modulo=$((num1 % num2))
    echo "División: $num1 / $num2 = $division"
    echo "Módulo: $num1 % $num2 = $modulo"
fi
```

Crear y ejecutar el script:

```bash
nano calculadora.sh
chmod +x calculadora.sh

# Probar con diferentes números
./calculadora.sh 20 5
./calculadora.sh 100 7
./calculadora.sh 15 0
```

10. Crea un script que cree un archivo de texto y guarde tu nombre en su interior.

```
#!/bin/bash

# Nombre: crear_archivo_nombre.sh

# Descripción: Crea un archivo de texto y guarda un nombre en

# Definir el nombre del archivo
archivo="mi_nombre.txt"

# Definir el nombre a guardar
nombre="Nica"

# Crear el archivo y guardar el nombre
echo "$nombre" > "$archivo"

# Confirmar que se creó
echo "Archivo '$archivo' creado exitosamente"
echo "Contenido guardado: $nombre"

# Mostrar el contenido del archivo
echo ""
echo "=== Contenido del archivo ==="
cat "$archivo"

# Bonus: Añadir más información al archivo
echo "" >> "$archivo"
echo "Fecha de creación: $(date)" >> "$archivo"
echo "Directorio: $(pwd)" >> "$archivo"
echo ""
echo "=== Contenido completo del archivo ==="
cat "$archivo"
Crear y ejecutar el script:
nano crear_archivo_nombre.sh
chmod +x crear_archivo_nombre.sh
./crear_archivo_nombre.sh

# Verificar que se creó el archivo
ls -l mi_nombre.txt
cat mi_nombre.txt
```

Versión alternativa con input del usuario:

```bash
#!/bin/bash

# Nombre: crear_archivo_nombre_input.sh

# Descripción: Pide el nombre al usuario y lo guarda en un archivo

# Pedir el nombre al usuario
read -p "Introduce tu nombre: " nombre

# Definir el nombre del archivo
archivo="nombre_usuario.txt"

# Crear el archivo y guardar el nombre
echo "$nombre" > "$archivo"
echo "Tu nombre ha sido guardado en '$archivo'"
echo "Contenido: $(cat $archivo)"
```

COMILLAS Y EXPANSIÓN

Este es el capítulo más importante del curso, y el que casi nadie explica a tiempo. La mayoría de los scripts que fallan de forma rara —que funcionan con un archivo pero no con otro, o que se rompen cuando algo está vacío— fallan por lo que se cuenta aquí.

La idea de fondo es sencilla: **antes de ejecutar nada, Bash reescribe la línea que has escrito**. Ese paso intermedio se llama *expansión*, y si no sabes que existe, el comportamiento de Bash parece caprichoso.

Qué hace Bash antes de ejecutar

Cuando pulsas Enter, Bash no ejecuta tu línea tal cual. Primero la transforma, en este orden:

1. **Sustituye las variables** por su valor: `$nombre` → `Nica`
2. **Ejecuta las sustituciones de comandos**: `$(pwd)` → `/home/nica`
3. **Parte el resultado en palabras** por los espacios *(word splitting)*
4. **Expande los comodines**: `*.txt` → `notas.txt informe.txt`
5. Y **entonces** ejecuta el comando con las palabras resultantes

Los pasos 3 y 4 son los que dan sorpresas, porque ocurren **después** de meter el valor de la variable. Bash no ve `$nombre`: ve lo que había dentro.

> **Consejo**
>
> Cuando no entiendas qué está haciendo un script, ponle `bash -x` delante: `bash -x script.sh`. Te muestra cada línea *ya expandida*, justo antes de ejecutarla. Es la mejor forma de ver este capítulo en acción.

Los tres tipos de comillas

Bash tiene tres formas de escribir texto, y hacen cosas distintas.

Forma	Qué hace con `$variable`	Qué hace con los espacios
Sin comillas	La sustituye	**Parte el texto en palabras**
`"comillas dobles"`	La sustituye	Los respeta: es **una sola palabra**
`'comillas simples'`	**No la sustituye**, es texto literal	Los respeta

Un ejemplo que lo enseña todo de golpe:

```
nombre="Nica de Tomás"

echo $nombre     # Nica de Tomás   -> tres palabras
echo "$nombre"   # Nica de Tomás   -> una sola palabra
echo '$nombre'   # $nombre         -> literal, no sustituye nada
```

En pantalla las dos primeras se ven casi igual, y por eso el error pasa desapercibido. Pero para Bash son cosas muy distintas: en la primera, `echo` recibe **tres** argumentos; en la segunda, **uno**.

Con `echo` da igual. Con casi todo lo demás, no.

El problema de verdad: nombres con espacios

Imagina que tienes un archivo llamado `mis notas.txt`, con un espacio:

```
archivo="mis notas.txt"

rm $archivo      # MAL: Bash lo lee como  rm "mis" "notas.txt"
                 # Intenta borrar DOS archivos que no existen

rm "$archivo"    # BIEN: borra el archivo "mis notas.txt"
```

Sin comillas, Bash parte el valor por el espacio y `rm` recibe dos argumentos. Con comillas, recibe uno. El script "funciona" mientras tus archivos no tengan espacios, y se rompe el día que alguno los tiene.

> **Precaución**
>
> Este fallo es especialmente peligroso con `rm`, `mv` y `cp`. Un script sin comillas puede borrar algo que no querías, o quejarse de archivos que sí existen.

El problema silencioso: variables vacías

Este es peor, porque no da error: da un resultado **falso**.

```
nombre=""

if [ -n $nombre ]; then
    echo "El nombre existe"
fi
```

Ese script imprime «El nombre existe», aunque el nombre esté vacío. ¿Por qué? Porque Bash sustituye `$nombre` por nada, y la línea que se ejecuta de verdad es:

```
if [ -n ]; then
```

Y `[ -n ]` con un solo argumento no comprueba nada: `[` simplemente ve un texto (`-n`) que no está vacío y responde «verdadero». La comprobación se ha evaporado.

Con comillas funciona como esperas:

```
nombre=""

if [ -n "$nombre" ]; then     # se convierte en:  [ -n "" ]  -> falso
    echo "El nombre existe"
fi
```

La regla práctica

No hace falta memorizar los casos. Basta con una norma:

> **Importante**
>
> **Pon siempre comillas dobles alrededor de las variables:** `"$archivo"`, `"$nombre"`, `"$1"`, `"$@"`.
>
> Es más fácil ponerlas siempre que acordarse de cuándo hacen falta. No estorban nunca, y te ahorran la clase entera de errores de este capítulo.

Usa comillas simples solo cuando quieras que el texto salga **tal cual**, sin que Bash toque nada:

```
echo 'El precio es $50'          # El precio es $50
echo "El precio es $50"          # El precio es 0   <- ¡ojo!
```

En el segundo caso Bash intentó sustituir la variable `$5` (que está vacía) y dejó el `0` suelto.

Sustitución de comandos con $(…)

Sirve para meter la salida de un comando dentro de otra cosa:

```
echo "Hoy es $(date '+%d/%m/%Y')"
echo "Estás en $(pwd)"

archivos=$(ls *.txt | wc -l)
echo "Hay $archivos archivos de texto"
```

Verás también la forma antigua con comillas invertidas, `` `pwd` ``. Hace lo mismo, pero **usa siempre** `$(...)`: se puede anidar y se distingue mucho mejor de las comillas normales.

Y aquí también hacen falta las comillas dobles, por lo mismo de antes:

```
ruta=$(pwd)
cd "$ruta"      # funciona aunque la ruta tenga espacios
```

Las llaves ${variable}

Las llaves marcan dónde acaba el nombre de la variable. Sin ellas, Bash no sabe dónde cortar:

```
fichero="informe"

echo "$fichero_final.txt"     # .txt   -> busca la variable "fichero_final",
                              #           que no existe, y solo queda el .txt
echo "${fichero}_final.txt"   # informe_final.txt
```

Si detrás de la variable viene una letra, un número o un guion bajo, necesitas las llaves. En el resto de casos son opcionales.

Escapar caracteres con la barra invertida

La barra invertida quita el significado especial al carácter siguiente:

```
echo "Cuesta \$50"            # Cuesta $50   (el \ evita la sustitución)
echo "Comillas: \"hola\""     # Comillas: "hola"
touch mi\ archivo.txt         # crea un archivo llamado "mi archivo.txt"
```

Resumen

Escribes	Bash entiende
`$var`	El valor, **partido** por espacios y con comodines expandidos
`"$var"`	El valor, **entero**, como una sola palabra
`'$var'`	El texto literal `$var`
`${var}`	El valor, marcando dónde acaba el nombre
`$(comando)`	La salida del comando
`\$`	Un símbolo de dólar normal y corriente

Ejercicios

1. Crea una variable `saludo` con el valor `Hola desde Bash` y muéstrala con `echo` de las tres formas: sin comillas, con dobles y con simples. Explica qué diferencia ves y cuál no se aprecia en pantalla.
2. Crea un archivo con un espacio en el nombre (`touch "mi archivo.txt"`). Guarda el nombre en una variable e intenta borrarlo sin comillas. ¿Qué error da? Ahora hazlo con comillas.
3. Escribe un script que reciba un argumento y compruebe con `[ -n ... ]` si está vacío. Pruébalo primero sin comillas y ejecútalo **sin pasarle nada**. ¿Qué responde? Añade las comillas y vuelve a probarlo.
4. Usa `$(...)` para guardar en una variable el número de archivos del directorio actual y muéstralo en una frase.
5. Crea una variable `base` con el valor `copia` y consigue que `echo` imprima `copia_seguridad.txt` usando llaves.
6. Haz que `echo` imprima literalmente `El total es $100` de dos maneras distintas: con comillas simples y con `\`.
7. Coge cualquier script de un capítulo anterior y ejecútalo con `bash -x`. Localiza en la salida una línea donde se vea una variable ya sustituida.

LÓGICA

Operadores aritméticos

Sirven para hacer cuentas. Se usan dentro de `$(( ))`.

- `+` suma
- `-` resta
- `*` multiplicación
- `/` división entera
- `%` módulo

> **Aviso**
> La división de Bash es **entera**: `$((7 / 2))` da `3`, no `3.5`. Si necesitas decimales, hay que recurrir a `bc` o a `awk`.

Operadores de comparación

Estos **no** son aritméticos: no calculan, sino que responden verdadero o falso. Son los que se usan dentro de un `if`.

Comparar números

- `-eq` igual
- `-ne` distinto
- `-gt` mayor
- `-lt` menor
- `-ge` mayor o igual
- `-le` menor o igual

Comparar cadenas de texto

- `=` igual
- `!=` distinto
- `-z` cadena vacía
- `-n` cadena no vacía
- `>` mayor en orden alfabético (solo dentro de `[[ ]]`)
- `<` menor en orden alfabético (solo dentro de `[[ ]]`)

Combinar condiciones

- `&&` AND (Y): se tienen que cumplir las dos.
- `||` OR (O): basta con que se cumpla una.
- `!` NOT (NO): invierte la condición.

Comprobar archivos

- `-e` existe
- `-f` es un archivo regular (contiene datos legibles o binarios y no es un directorio)
- `-d` es un directorio
- `-r` tiene permisos de lectura

- `-w` tiene permisos de escritura
- `-x` es ejecutable
- `-s` existe y no está vacío

Los dos tipos de corchetes

Habrás visto las comprobaciones escritas de dos formas. No son lo mismo.

`[ ]` es en realidad **un comando**, no parte del lenguaje. Por eso hay que dejar espacios a los lados: `[ -f "$archivo" ]` funciona, `[-f "$archivo"]` da error de «orden no encontrada». Y por eso también es tan sensible a las comillas.

`[[ ]]` sí es parte de Bash, y es **más seguro y más cómodo**:

```
archivo=""

[ -n $archivo ]      # error silencioso: da verdadero (ver cap. de comillas)
[[ -n $archivo ]]    # correcto: da falso, aunque olvides las comillas
```

Además `[[ ]]` permite cosas que `[ ]` no:

```
# Comparar alfabéticamente
[[ "$a" < "$b" ]]

# Comprobar con un patrón
[[ "$fichero" == *.txt ]]

# Comprobar con una expresión regular
[[ "$edad" =~ ^[0-9]+$ ]] && echo "Es un número"

# Y y O dentro de los mismos corchetes
[[ $num -ge 18 && -n "$nombre" ]]
```

> **Consejo**
>
> **Usa `[[ ]]` siempre que escribas para Bash.** Es más difícil equivocarse. `[ ]` solo hace falta si necesitas que el script funcione también con `sh` en sistemas muy antiguos.
>
> Aun así, **sigue poniendo comillas**: `[[ ]]` te salva de las variables vacías, pero acostumbrarse a las comillas te salva en todos los demás sitios.

En el resto del capítulo verás `[ ]` porque es lo que aparece en la mayoría de scripts que te vas a encontrar por ahí, y hay que saber leerlo.

Condicionales

Permiten ejecutar comandos solo si se cumplen ciertas condiciones.

if

```
if [ condición ]; then
    comando_si_verdadero
fi
```

if con else

```
if [ condición ]; then
    comando_si_verdadero
else
    comando_por_defecto
fi
```

if con elif y else

```
if [ condición ]; then
    comando_si_verdadero
elif [ condición ]; then
    comando_si_verdadero
fi
```

> **Aviso**
>
> Puedes encadenar tantos `elif` como quieras, pero en cuanto **se cumple la primera condición, las demás ya no se evalúan**. El orden importa: pon primero las condiciones más concretas y deja las generales para el final.

```
if [ condición ]; then
    comando_si_verdadero
elif [ condición ]; then
    comando_si_verdadero
elif [ condición ]; then
    comando_si_verdadero
else
    comando_por_defecto
fi
```

case

Cuando hay que comparar **la misma variable contra muchos valores**, encadenar `elif` queda largo y farragoso. Para eso está `case`, que es lo que se usa normalmente para los menús.

```
case variable in
    valor_1) comando_si_es_valor_1;;
    valor_2) comando_si_es_valor_2;;
    valor_3) comando_si_es_valor_3;;
    *)       comando_por_defecto;;
esac
```

- Cada opción termina en **doble punto y coma** (`;;`).
- El `*)` del final es el «en cualquier otro caso», como el `else` del `if`. Conviene ponerlo siempre para que el script no se quede callado ante una entrada que no esperabas.

Un ejemplo real:

```
#!/bin/bash
read -r -p "Elige una opción (a/b/c): " opcion

case "$opcion" in
    a) echo "Elegiste A";;
    b) echo "Elegiste B";;
    c) echo "Elegiste C";;
    *) echo "Opción no válida";;
esac
```

Todo junto

```
#!/bin/bash

num=25
name="Nica"

# Comparar números
if [ $num -ge 10 ]; then
    echo "Número mayor o igual que 10"
elif [ $num -eq 0 ]; then
    echo "Número igual a 0"
else
    echo "Condición por defecto"
fi

# Comprobar que una cadena no está vacía
if [ -n "$name" ]; then
    echo "El nombre existe"
fi

# Dos condiciones a la vez
if [ $num -ge 18 ] && [ -n "$name" ]; then
    echo "Mayor de edad"
fi

# Comprobar que un archivo existe
if [ -e "./script.sh" ]; then
    echo "El archivo existe"
fi
```

- `$num -ge 10` → comprueba que la variable *num* sea *mayor o igual que 10*.
- `$num -eq 0` → comprueba que la variable *num* sea *igual a 0*.
- `-n "$name"` → comprueba que la cadena asociada a *name* no está *vacía*. Fíjate en las **comillas**: sin ellas, la comprobación daría verdadero siempre, incluso con la variable vacía. Está explicado en el capítulo *Comillas y expansión*.
- `[ $num -ge 18 ] && [ -n "$name" ]` → comprueba que *num* sea *mayor o igual que 18* y que *name* no está *vacío*.
- `-e "./script.sh"` → comprueba que el *script existe*.

Bucles

Un bucle repite un trozo de código varias veces. Bash tiene tres, y la diferencia está en cuándo paran.

for

Recorre **una lista de valores** que ya conoces de antemano: números, nombres de archivo, líneas... Se ejecuta una vez por cada elemento.

```
for i in 1 2 3 4 5
do
    echo "Número: $i"
done
for name in *.sh
do
    echo "Archivo: $name"
done
```

- `for i in 1 2 3 4 5` recorre el listado de números definidos, almacenando el valor de cada iteración en la variable *i*.
- `for name in *.sh` recorre el listado de archivos que cumplan el criterio de la extensión, almacenando el valor de cada iteración en la variable `name`.

while

Se repite **mientras** la condición sea verdadera. Se usa cuando no sabes de antemano cuántas vueltas hará.

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

- `while [ $count -le 5 ]` se ejecuta el bucle mientras el valor de la variable *count* sea *menor o igual que 5*.
- `((count++))` incrementa el valor de *count* en 1 tras cada ejecución del bucle.

until

Es el `while` al revés: se repite **hasta que** la condición se cumpla.

```
count=1
until [ $count -gt 10 ]
do
    echo "Contador: $count"
    ((count++))
done
```

- `until [ $count -gt 10 ]` → se ejecuta el bucle hasta el valor de la variable *count* sea *mayor que 10*.
- `((count++))` incrementa el valor de *count* en 1 tras cada ejecución del bucle.

Control de bucles

```
for i in 1 2 3 4 5
do
    if [ $i -eq 3 ]; then
        continue
    elif [ $i -eq 4 ]; then
        break
    fi
    echo "Número: $i"
done
```

- `continue` salta la iteración y pasa a la siguiente.
- `break` sale del bucle actual.

Script de ejemplo con bucles

```
#!/bin/bash

# for
for i in 1 2 3 4 5
do
   echo "Número: $i"
done
for name in *.sh
do
   echo "Archivo: $name"
done

# while
count=1
while [ $count -le 5 ]
do
   echo "Contador: $count"
   ((count++))
done

# until
count=1
until [ $count -gt 10 ]
do
   echo "Contador: $count"
   ((count++))
done

# Control de bucles
for i in 1 2 3 4 5
do
   if [ $i -eq 3 ]; then
       continue
   elif [ $i -eq 4 ]; then
       break
   fi
       echo "Número: $i"
done
```

Funciones

Una función es un trozo de script al que le pones nombre para poder reutilizarlo. Sirve para no repetir el mismo código diez veces.

Función simple

```
saludar() {
    echo "Hola desde la función"
}

saludar
```

- `saludar() { ... }` es la **definición**: describe qué hace, pero no la ejecuta.

- `saludar` a secas es la **llamada**: ahí sí se ejecuta.

La definición tiene que ir **antes** de la llamada en el script. Bash lee de arriba abajo: si llamas a una función que aún no ha leído, dirá que no existe.

Función con parámetros

```
saludar_a() {
    echo "Hola $1"
}

saludar_a Nica
```

Dentro de la función, `$1` es el primer argumento **que le pasas a la función**, `$2` el segundo, y así.

> **Aviso**
>
> Ojo con esto, que confunde mucho: dentro de una función, `$1` **ya no es el primer argumento del script**, sino el de la función. En cambio `$0` sigue siendo el nombre del script.
>
> Si dentro de una función necesitas un argumento del script, pásaselo tú: `mi_funcion "$1"`.

Lo que devuelve una función

Aquí hay una trampa que conviene entender bien desde el principio.

```
comprobar_edad() {
    if [ "$1" -ge 18 ]; then
        return 0        # 0 = todo bien
    else
        return 1        # distinto de 0 = algo pasa
    fi
}

comprobar_edad 20
echo $?                 # 0
```

> **Precaución**
>
> `return` **no devuelve un resultado**, devuelve un **código de salida**: un número entre 0 y 255 que dice si la función fue bien o mal. Y al revés de lo que uno esperaría, **0 significa éxito** y cualquier otro número significa error.
>
> Por eso `return 1` no está «devolviendo un uno»: está diciendo «he fallado».

Si lo que quieres es que la función te devuelva **un dato** (una suma, un texto, una ruta), no se usa `return`: se usa `echo` y se recoge con `$(...)`.

```bash
sumar() {
    echo $(( $1 + $2 ))
}

resultado=$(sumar 5 3)
echo "La suma es $resultado"     # La suma es 8
```

Resumiendo:

Quieres...	Usa	Se recoge con
Decir si fue bien o mal	`return 0` / `return 1`	`$?`
Devolver un dato	`echo "$valor"`	`$(mi_funcion)`

Variables globales y locales (Ámbito)

- Variables **globales**: visibles en todo el script.
- Variables **locales**: visibles solo dentro de la función (`local`).

```bash
#!/bin/bash
nombre="Nica"                  # global: visible en todo el script
saludar() {
    local mensaje="hola"       # local: solo existe aquí dentro
    echo "Dentro:  $mensaje, $nombre"
}
saludar
echo "Fuera:   $mensaje, $nombre"
```

Al ejecutarlo se ve la diferencia:

```
Dentro:  hola, Nica
Fuera:   , Nica
```

- `local` sirve para definir variables locales.
- Dentro de la función se ven **las dos** variables: la local y la global.
- Fuera, `$mensaje` sale **vacío**: la variable local dejó de existir en cuanto terminó la función. `$nombre`, que es global, sigue ahí.

> **Precaución**
>
> Declara siempre con `local` las variables que solo uses dentro de una función. Si te olvidas del `local`, la variable pasa a ser global sin que te des cuenta y puedes machacar otra que se llame igual.

Script de ejemplo con funciones

```
#!/bin/bash

# Función
my_function() {
   echo "Hola desde la función"
}
my_function

# Función con parámetros
my_function_with_params() {
   echo "Hola $1"
}
my_function_with_params Nica

# Función con retorno
my_function_with_return() {
   return 1
}
my_function_with_return
echo $?

# Variables globales y locales
name=Nica # global
my_function_2() {
   local msj=", mundo" # local
   echo "Hola $msj $name"
}
echo "Hola desde fuera $msj"
```

my_function_2

Manejo básico de errores

Códigos de salida

```
0 Éxito
!=0 Error

#!/bin/bash
cp file.txt ../Course
if [ $? -ne 0 ]; then
    echo "Error al copiar el archivo"
fi
cp file.txt ../Course intenta realizar la copia.
```

- $? -ne 0 accede al código de salida y comprueba si es distinto de cero (error).

Atajos

Bash permite encadenar comandos según si el anterior fue bien o mal, sin escribir un `if` entero:

- `comando1 || comando2` ejecuta el segundo **solo si el primero falla**.
- `comando1 && comando2` ejecuta el segundo **solo si el primero fue bien**.

```
cp file.txt ../Course || echo "Otra vez se ha producido el error"
cp script.sh ../Course && echo "No se ha producido un error"
```

- || se ejecuta el *echo* si se produce un error en el *cp*.
- && se ejecuta el *echo* si *cp* se realiza con éxito.

Script de ejemplo con errores

```
#!/bin/bash
cp file.txt ../prueba-wsl
if [ $? -ne 0 ]; then
   echo "Error al copiar el archivo"
fi
cp file.txt ../prueba-wsl || echo "Otra vez se ha producido el error, fallo al copiar el archivo"
cp bucles.sh ../prueba-wsl && echo "No se ha producido un error, fichero copiado correctamente"
```

Ejercicios a realizar en la parte de lógica

1. Crea un script que pida un número y muestre si es positivo, negativo o cero usando if, elif y else.
2. Pide al usuario dos números y muestra cuál es mayor o si son iguales.
3. Crea un script que muestre un menú con tres opciones y ejecute la opción correspondiente según la elección del usuario.
4. Muestra todos los números del 1 al 10 usando un bucle for.
5. Crea un script que pida números al usuario hasta que introduzca el número 0. Al final, muestra cuántos números ha introducido en total.
6. Haz un script que muestre los números del 1 al 10, saltando el 5 y deteniéndose en el 8.
7. Crea una función saludar que reciba un nombre como argumento y muestre: Hola , bienvenido al script.
8. Crea una función que reciba dos números, calcule su suma y la devuelva usando return. Muestra el resultado en el script principal.
9. Intenta copiar un archivo que no exista y muestra un mensaje de error si el comando falla, usando $? o ||.

10. Crea un script con un comentario inicial con autor, fecha, descripción y un bucle for que liste todos los archivos .sh en el directorio actual.

Soluciones a los ejercicios

Pueden variar según tu sistema y los directorios que tengas.

1. Crea un script que pida un número y muestre si es positivo, negativo o cero usando if, elif y else.

```
#!/bin/bash

# Nombre: positivo_negativo.sh

# Descripción: Verifica si un número es positivo, negativo o cero

# Pedir un número al usuario
read -r -p "Introduce un número: " numero

# Verificar si es positivo, negativo o cero
if [ $numero -gt 0 ]; then
    echo "El número $numero es positivo"
elif [ $numero -lt 0 ]; then
    echo "El número $numero es negativo"
else
    echo "El número es cero"
fi
```

2. Pide al usuario dos números y muestra cuál es mayor o si son iguales.

```
#!/bin/bash

# Nombre: comparar_numeros.sh

# Descripción: Compara dos números y muestra cuál es mayor

# Pedir dos números al usuario
read -r -p "Introduce el primer número: " num1
read -r -p "Introduce el segundo número: " num2

# Comparar los números
if [ $num1 -gt $num2 ]; then
    echo "El número $num1 es mayor que $num2"
elif [ $num1 -lt $num2 ]; then
    echo "El número $num2 es mayor que $num1"
else
    echo "Los números son iguales: $num1 = $num2"
fi
```

3. Crea un script que muestre un menú con tres opciones y ejecute la opción correspondiente según la elección del usuario.

```bash
#!/bin/bash

# Nombre: menu.sh

# Descripción: Muestra un menú con tres opciones

# Mostrar el menú
echo "=== MENÚ PRINCIPAL ==="
echo "1. Mostrar fecha y hora"
echo "2. Listar archivos del directorio"
echo "3. Mostrar información del usuario"
echo "======================"
read -r -p "Elige una opción (1-3): " opcion

# Ejecutar la opción elegida
case $opcion in
    1)
    echo ""
    echo "Fecha y hora actual:"
    date ;;
2)
echo ""
echo "Archivos en el directorio actual:"
ls -lh ;;
3)
echo ""
echo "Información del usuario:"
echo "Usuario: $USER"
echo "Directorio home: $HOME"
echo "Shell actual: $SHELL" ;; *)
echo "Opción inválida. Por favor, elige 1, 2 o 3." ;;
esac
```

4. Muestra todos los números del 1 al 10 usando un bucle for.

```bash
#!/bin/bash

# Nombre: bucle_for_1_10.sh

# Descripción: Muestra los números del 1 al 10 con bucle for
echo "Números del 1 al 10:"

# Bucle for con secuencia
for i in {1..10}
do
    echo $i
done

# Alternativa 1: usando seq
echo ""
echo "=== Alternativa con seq ==="
for i in $(seq 1 10)
do
    echo $i
done

# Alternativa 2: estilo C
echo ""
echo "=== Alternativa estilo C ==="
for ((i=1; i<=10; i++))
do
    echo $i
done
```

Crear y ejecutar el script:

```bash
nano bucle_for_1_10.sh
chmod +x bucle_for_1_10.sh
./bucle_for_1_10.sh
```

5. Crea un script que pida números al usuario hasta que introduzca el número 0. Al final, muestra cuántos números ha introducido en total.

```bash
#!/bin/bash

# Nombre: contar_numeros.sh

# Descripción: Pide números hasta que el usuario introduzca 0

# Inicializar contador
contador=0
echo "Introduce números (0 para terminar):"

# Bucle while infinito
while true
do
    read -r -p "Número: " numero

    # Verificar si es 0 para salir
    if [ $numero -eq 0 ]; then
        break
    fi

    # Incrementar contador
    ((contador++))
done

# Mostrar el resultado
    echo ""
    echo "Has introducido $contador números en total."
    Versión alternativa:

    #!/bin/bash
    contador=0
    numero=1 # Inicializar con valor diferente de 0
    echo "Introduce números (0 para terminar):"
    while [ $numero -ne 0 ]
    do
        read -r -p "Número: " numero
        if [ $numero -ne 0 ]; then
            ((contador++))
        fi
    done
    echo "Has introducido $contador números en total."
```

6. Haz un script que muestre los números del 1 al 10, saltando el 5 y deteniéndose en el 8.

```bash
#!/bin/bash

# Nombre: continue_break.sh

# Descripción: Muestra números del 1 al 10, salta el 5 y se detiene
en el 8
echo "Números del 1 al 10 (saltando el 5 y deteniéndose en el 8):"
for i in {1..10}
do

    # Saltar el número 5
    if [ $i -eq 5 ]; then
        continue
    fi

    # Mostrar el número
    echo $i

    # Detenerse después del 8
    if [ $i -eq 8 ]; then
        break
    fi
done
echo "Bucle terminado."
```

Versión con más explicación:

```bash
#!/bin/bash

# continue_break_explicado.sh
echo "=== Bucle con continue y break ==="
echo "Saltaremos el 5 (continue) y nos detendremos en el 8 (break)"
echo ""
for i in {1..10}
do

    # continue: salta a la siguiente iteración
    if [ $i -eq 5 ]; then
        echo "[Saltando el $i]"
        continue
    fi
    echo "Número: $i"

    # break: sale del bucle completamente
    if [ $i -eq 8 ]; then
        echo "[Deteniéndose en el $i]"
        break
    fi
done
echo ""
echo "Fin del bucle."
```

7. Crea una función saludar que reciba un nombre como argumento y muestre: Hola , bienvenido al script.

```
#!/bin/bash

# Nombre: funcion_saludar.sh

# Descripción: Función que saluda con el nombre proporcionado

    # Definir la función
    saludar() {
        nombre=$1
        echo "Hola $nombre, bienvenido al script."
    }
```

- Llamar a la función con diferentes nombres

saludar “Nica”

saludar “Marta”

saludar “Leo”

```
# Pedir nombre al usuario y saludar
echo ""
read -r -p "Introduce tu nombre: " mi_nombre
saludar "$mi_nombre"
```

Versión con validación:

```
#!/bin/bash

# funcion_saludar_validacion.sh

# Función con validación
saludar() {
    if [ -z "$1" ]; then
        echo "Error: No se proporcionó un nombre"
        return 1
    fi
    nombre=$1
    echo "Hola $nombre, bienvenido al script."
}

# Probar la función
saludar "Nica"
saludar "" # Sin argumento
saludar "Leo"
```

8. Crea una función que reciba dos números, calcule su suma y la devuelva usando return. Muestra el resultado en el script principal.

```bash
#!/bin/bash

# Nombre: funcion_suma.sh

# Descripción: Función que suma dos números y devuelve el resultado

    # Función para sumar
    sumar() {
        local num1=$1
        local num2=$2
        local suma=$((num1 + num2))

        # return solo puede devolver valores entre 0-255

        # Por eso usamos echo para devolver el resultado
            echo $suma
        }

        # Llamar a la función y capturar el resultado
            resultado=$(sumar 5 3)
            echo "La suma de 5 + 3 = $resultado"
            resultado=$(sumar 10 20)
            echo "La suma de 10 + 20 = $resultado"

            # Pedir números al usuario
            read -r -p "Introduce el primer número: " a
            read -r -p "Introduce el segundo número: " b
            resultado=$(sumar $a $b)
            echo "La suma de $a + $b = $resultado"
            Versión alternativa usando variable global:

            #!/bin/bash

            # funcion_suma_return.sh

            # Nota: return en bash solo devuelve códigos de estado (0-255)

            # Para devolver valores, usamos variables globales o echo

            # Versión con variable global
            sumar_global() {
                local num1=$1
                local num2=$2
                RESULTADO=$((num1 + num2))
            }
            echo "=== Usando variable global ==="
            sumar_global 15 25
            echo "Resultado: $RESULTADO"

            # Versión con echo (recomendada)
            sumar_echo() {
                local num1=$1
                local num2=$2
                echo $((num1 + num2))
            }
            echo ""
            echo "=== Usando echo (capturando con \$()) ==="
```

```
resultado=$(sumar_echo 7 8)
echo "Resultado: $resultado"
```

9. Intenta copiar un archivo que no exista y muestra un mensaje de error si el comando falla, usando $? o ||.

```
#!/bin/bash

# Nombre: manejo_errores.sh

# Descripción: Manejo de errores al copiar archivos
echo "=== Método 1: Usando \$? ==="

# Intentar copiar un archivo que no existe
cp archivo_que_no_existe.txt destino.txt

# Verificar el código de salida
if [ $? -ne 0 ]; then
    echo "Error: No se pudo copiar el archivo"
else
    echo "Archivo copiado exitosamente"
fi
echo ""
echo "=== Método 2: Usando || (OR) ==="

# Si el comando falla (retorna no-cero), ejecuta lo que está después
de || cp otro_archivo_inexistente.txt destino2.txt || echo "Error: El archivo no
existe"
echo ""
echo "=== Método 3: Usando && (AND) ==="

# Si el comando tiene éxito (retorna 0), ejecuta lo que está después
de && cp archivo_real.txt copia.txt && echo "Archivo copiado con éxito" || echo
"Error al copiar"
echo ""
echo "=== Método 4: Manejo completo ==="
archivo_origen="archivo_inexistente.txt"
archivo_destino="destino.txt"
if [ ! -f "$archivo_origen" ]; then
    echo "Error: El archivo '$archivo_origen' no existe"
    exit 1
fi
cp "$archivo_origen" "$archivo_destino"
if [ $? -eq 0 ]; then
    echo "Archivo copiado exitosamente"
else
    echo "Error al copiar el archivo"
fi
Versión simplificada:

#!/bin/bash

# manejo_errores_simple.sh
echo "Intentando copiar un archivo que no existe..."

# Usando ||
cp archivo_inexistente.txt destino.txt || {
    echo "Error: No se pudo copiar el archivo"
    echo "El archivo probablemente no existe"
}
echo ""
echo "Script finalizado."
```

10. Crea un script con un comentario inicial con autor, fecha, descripción y un bucle for que liste todos los archivos .sh en el directorio actual.

```bash
#!/bin/bash

# Nombre: listar_scripts.sh

# Descripción: Este script lista todos los archivos con extensión .sh
# que se encuentran en el directorio actual
echo "=== Listado de archivos .sh en el directorio actual ==="
echo "Directorio: $(pwd)"
echo ""

# Contador de archivos
contador=0

# Bucle for para listar archivos .sh
for archivo in *.sh
do

    # Verificar si existen archivos .sh
    if [ -e "$archivo" ]; then
        ((contador++))
        echo "$contador. $archivo"

        # Información adicional del archivo
        tamanio=$(ls -lh "$archivo" | awk '{print $5}')
        echo " Tamaño: $tamanio"

        # Verificar si tiene permisos de ejecución
        if [ -x "$archivo" ]; then
            echo " Permisos: Ejecutable "
        else
            echo " Permisos: No ejecutable"
        fi
        echo ""
    else
        echo "No se encontraron archivos .sh en este directorio"
        break
    fi
done

# Resumen
if [ $contador -gt 0 ]; then
    echo "Total de archivos .sh encontrados: $contador"
else
    echo "Total de archivos .sh encontrados: 0"
fi
echo ""
echo "Fecha de ejecución: $(date '+%d/%m/%Y %H:%M:%S')"
```

Versión simplificada:

```bash
#!/bin/bash

# Script: listar_scripts_simple.sh

# Descripción: Lista archivos .sh en el directorio actual
echo "Archivos .sh en el directorio actual:"
echo ""
for archivo in *.sh
do
    if [ -f "$archivo" ]; then
        echo "- $archivo"
    fi
done
```

CRON

Abrir el explorador de archivos

A veces viene bien saltar de la terminal a la ventana de archivos, en la carpeta en la que estás. El punto significa «aquí».

Sistema	Comando
Linux	`xdg-open .`
macOS	`open .`
Windows (desde WSL)	`explorer.exe .`

Crontab

Cron es un demonio (daemon) del sistema que permite ejecutar tareas programadas automáticamente. Siempre está corriendo en segundo plano. De esta forma, una vez configurado, el sistema ejecuta las tareas sin que el usuario tenga que intervenir.

> **Nota**
> Cron usa unas tablas llamadas **crontab**. Hay una por usuario (la tuya, la que editas con `crontab -e`) y otra general del sistema, en `/etc/crontab`, que solo puede tocar el administrador.

Comando	Qué hace
`crontab -e`	Edita tu tabla de tareas
`crontab -l`	Lista tus tareas programadas
`crontab -r`	**Borra tu tabla entera**

> **Precaución**

> Cuidado con `crontab -r`: borra **todas** tus tareas de golpe y sin preguntar. Y está peligrosamente cerca de `crontab -e` en el teclado.
>
> Antes de tocar nada, guarda una copia:

```
crontab -l > ~/copia-crontab.txt
```

Sintaxis crontab

Cada línea de crontab representa una tarea programada.

┌──────────────── minuto (0-59)

│ ┌────────────── hora (0-23)

│ │ ┌──────────── día del mes (1-31)

│ │ │ ┌────────── mes (1-12)

│ │ │ │ ┌──── día de la semana (0-7, donde 0 y 7 son domingo)

│ │ │ │ │

-
  -
    -
      -
        - comando_a_ejecutar

Cada * representa estas unidades de tiempo (de izquierda a derecha):

- minuto (0-59)
- hora (0-23)
- día del mes (1-31)
- mes (1-12)
- día de la semana (0-7, donde 0 y 7 son domingo)

> **Consejo**
>
> Ejecuta primero de manera manual el comando para comprobar que funciona y que tienes los permisos adecuados.

Precaución

```
Si el comando a ejecutar es un script, utiliza su ruta absoluta.
```

Ejemplos de sintaxis

Línea	Cuándo se ejecuta
`30 2 * * *`	Todos los días a las 2:30
`*/5 * * * *`	Cada 5 minutos
`0 9 * * 1`	Todos los lunes a las 9:00
`0 0 1 * *`	El día 1 de cada mes, a medianoche
`0 8 * * 1-5`	De lunes a viernes a las 8:00

Comodines

Símbolo	Significa	Ejemplo
`*`	Cualquier valor	`* * * * *` cada minuto
`,`	Lista de valores	`0 8,12 * * *` a las 8 y a las 12
`-`	Rango	`0 9-17 * * *` cada hora de 9 a 17
`*/n`	Cada n unidades	`*/10 * * * *` cada 10 minutos

> **Consejo**
>
> Para no equivocarte con la sintaxis, escribe la línea en crontab.guru: te la traduce a lenguaje normal mientras la escribes.

Antes de programar nada

Cron es una fuente inagotable de «pues a mí me funcionaba». Estas tres costumbres evitan casi todos los problemas:

1. **Prueba el comando a mano primero.** Si no funciona en tu terminal, tampoco va a funcionar en cron.
2. **Usa rutas absolutas siempre**, tanto para el script como para los archivos que toque. Cron no arranca en el directorio que tú crees.
3. **Guarda la salida en un archivo** mientras depuras, o no verás ningún error:

```
*/5 * * * * /home/nica/mi_script.sh >> /home/nica/cron.log 2>&1
```

> **Aviso**
>
> Cron ejecuta las tareas con un entorno **casi vacío**: un `PATH` mínimo y sin tus variables ni tus alias del `.bashrc`. Por eso un script que va perfecto en tu terminal puede fallar en cron diciendo que no encuentra un comando.
>
> La solución es usar rutas absolutas también para los comandos, o definir el `PATH` al principio del crontab:

```
PATH=/usr/local/bin:/usr/bin:/bin
```

Ejercicios

1. Crea un script que muestre la fecha y hora actual en un archivo. Programa su ejecución cada minuto con una cron.
2. Crea un script que muestre “Hola desde cron” en un archivo. Configura cron para ejecutarlo cada 5 minutos.
3. Escribe un script de backup que comprima una carpeta en un archivo con fecha. Programa su ejecución cada día a las 2:00 AM.
4. Haz un script que borre archivos .log de una carpeta temporal. Programa su ejecución todos los domingos a la medianoche.
5. Programa un script que escriba la hora actual, ejecutándose cada hora de 9 a 17 (horario laboral).
6. Programa un script que muestre “Hoy toca practicar” en un archivo de log solo los lunes, miércoles y viernes a las 8:00 AM.
7. Modifica uno de los scripts para que su salida y errores se guarden en cron.log.
8. Programa un script que muestre “Sistema OK” y lo ejecute cada 10 minutos.
9. Crea un script que genere un archivo con la fecha actual. Prográmalo para ejecutarse el primer día de cada mes a medianoche.
10. Configura un script que escriba “Probando cron” en un archivo. Después, buscar información para revisar los logs del sistema y confirmar su ejecución.

Posibles correcciones

(ten en cuenta que pueden variar dependiendo de tu sistema operativo y directorios existentes)

1. Crea un script que muestre la fecha y hora actual en un archivo. Programa su ejecución cada minuto con una cron.

```
#!/bin/bash

# Nombre: fecha_hora.sh

# Descripción: Guarda la fecha y hora actual en un archivo

# Definir el archivo de salida
archivo="$HOME/fecha_hora.log"

# Escribir la fecha y hora en el archivo
echo "Fecha y hora: $(date '+%d/%m/%Y %H:%M:%S')" >> "$archivo"
```

Pasos para configurar:

```
# 1. Crear el script
nano ~/fecha_hora.sh

# 2. Dar permisos de ejecución
chmod +x ~/fecha_hora.sh

# 3. Probar el script
./fecha_hora.sh
cat ./fecha_hora.log

# 4. Editar crontab
crontab -e

# 5. Añadir esta línea al crontab:
* * * * * $HOME/fecha_hora.sh

# 6. Verificar que se añadió correctamente
crontab -l

# 7. Esperar 1-2 minutos y verificar el archivo
tail -f ~/fecha_hora.log
```

Explicación del cron:

-
    -
        -
            -
                - comando

| | | | |

| | | | └──── Día de la semana (0-7, donde 0 y 7 = domingo)

| | | └────── Mes (1-12)

| | └──────── Día del mes (1-31)

| └────────── Hora (0-23)

└──────────────────── Minuto (0-59)

- = cada minuto/hora/día/mes/día de semana

2. Crea un script que muestre "Hola desde cron" en un archivo. Configura cron para ejecutarlo cada 5 minutos.

```
#!/bin/bash

# Nombre: hola_cron.sh

# Descripción: Escribe un mensaje en un archivo cada 5 minutos

# Archivo de salida
archivo="$HOME/hola_cron.log"

# Escribir mensaje con fecha
echo "[$(date '+%Y-%m-%d %H:%M:%S')] Hola desde cron" >> "$archivo"
Configurar crontab:

# 1. Crear y preparar el script
nano ~/hola_cron.sh
chmod +x ~/hola_cron.sh

# 2. Editar crontab
crontab -e

# 3. Añadir esta línea (ejecutar cada 5 minutos):
*/5 * * * * $HOME/hola_cron.sh

# 4. Verificar
crontab -l

# 5. Monitorear el archivo
tail -f ~/hola_cron.log
```

Explicación

/5 * * = Cada 5 minutos

/10 * * = Cada 10 minutos

/15 * * = Cada 15 minutos

/30 * * = Cada 30 minutos

3. Escribe un script de backup que comprima una carpeta en un archivo con fecha. Programa su ejecución cada día a las 2:00 AM.

```bash
#!/bin/bash

# Nombre: backup_diario.sh

# Descripción: Realiza backup comprimido de una carpeta

# Configuración
carpeta_origen="$HOME/documentos"
carpeta_destino="$HOME/backups"
fecha=$(date '+%Y%m%d_%H%M%S')
nombre_backup="backup_${fecha}.tar.gz"

# Crear carpeta de destino si no existe
mkdir -p "$carpeta_destino"

# Archivo donde se registra lo que pasa
log_file="$HOME/backups/backup.log"

# Realizar backup
tar -czf "$carpeta_destino/$nombre_backup" -C "$HOME" "documentos" 2>/dev/null

# OJO: $? hay que comprobarlo AQUI, en la linea siguiente al comando.
# Si entre medias metes cualquier otra cosa (aunque sea una asignacion),
# $? pasa a ser el resultado de ESA otra cosa y no el del tar.
if [ $? -eq 0 ]; then
    echo "[$(date '+%Y-%m-%d %H:%M:%S')] Backup exitoso: $nombre_backup" >>
"$log_file"
else
    echo "[$(date '+%Y-%m-%d %H:%M:%S')] Error en backup" >> "$log_file"
fi

# Opcional: Eliminar backups antiguos (más de 7 días)
find "$carpeta_destino" -name "backup_*.tar.gz" -mtime +7 -delete
```

Configurar crontab:

```bash
# 1. Crear el script
nano ~/backup_diario.sh
chmod +x ~/backup_diario.sh

# 2. Crear carpetas necesarias
mkdir -p ~/documentos ~/backups

# 3. Probar el script manualmente
~/backup_diario.sh
ls -lh ~/backups/

# 4. Editar crontab
crontab -e
```

- 5. Añadir para ejecutar a las 2:00 AM diariamente:

0 2 * * * $HOME/backup_diario.sh

```
# 6. Verificar
crontab -l
```

Explicación del cron:

0 2 * * * = A las 2:00 AM todos los días

30 3 * * * = A las 3:30 AM todos los días

0 0 * * * = A medianoche (00:00) todos los días

4. Haz un script que borre archivos .log de una carpeta temporal. Programa su ejecución todos los domingos a la medianoche.

```
#!/bin/bash

# Nombre: limpiar_logs.sh

# Descripción: Elimina archivos .log de carpeta temporal

# Carpeta a limpiar
carpeta_temporal="$HOME/temp_logs"

# Archivo de registro
log_limpieza="$HOME/limpieza.log"

# Crear carpeta si no existe
mkdir -p "$carpeta_temporal"

# Contar archivos antes de eliminar
cantidad_antes=$(find "$carpeta_temporal" -name "*.log" -type f | wc -l)

# Eliminar archivos .log
find "$carpeta_temporal" -name "*.log" -type f -delete

# Contar archivos después
cantidad_despues=$(find "$carpeta_temporal" -name "*.log" -type
f | wc -l)
eliminados=$((cantidad_antes - cantidad_despues))

# Registrar la acción
echo "[$(date '+%Y-%m-%d %H:%M:%S')] Archivos .log eliminados: $eliminados" >>
"$log_limpieza"
Configurar crontab:

# 1. Crear el script
nano ~/limpiar_logs.sh
chmod +x ~/limpiar_logs.sh

# 2. Crear carpeta temporal y algunos archivos de prueba
mkdir -p ~/temp_logs
touch ~/temp_logs/test{1..5}.log

# 3. Probar el script
~/limpiar_logs.sh
cat ~/limpieza.log

# 4. Editar crontab
crontab -e

# 5. Añadir para ejecutar domingos a medianoche:
```

0 0 * * 0 $HOME/limpiar_logs.sh

```
# 6. Verificar
crontab -l
```

Explicación días de la semana:

0 0 * * 0 = Domingo a medianoche

0 0 * * 1 = Lunes a medianoche

0 0 * * 6 = Sábado a medianoche

0 0 * * 1-5 = lunes a viernes a medianoche

0 0 * * 0,6 = Fines de semana a medianoche

Días: 0 o 7 = Domingo, 1 = Lunes, 2 = Martes, …, 6 = Sábado

5. Programa un script que escriba la hora actual, ejecutándose cada hora de 9 a 17 (horario laboral).

```
#!/bin/bash

# Nombre: horario_laboral.sh

# Descripción: Registra la hora durante el horario laboral

# Archivo de salida
archivo="$HOME/horario_laboral.log"

# Escribir hora actual
echo "Hora registrada: $(date '+%H:%M:%S - %d/%m/%Y')" >> "$archivo"
Configurar crontab:

# 1. Crear el script
nano ~/horario_laboral.sh
chmod +x ~/horario_laboral.sh

# 2. Probar
~/horario_laboral.sh
cat ~/horario_laboral.log

# 3. Editar crontab
crontab -e

# 4. Añadir para ejecutar cada hora de 9 AM a 5 PM:
```

0 9-17 * * * $HOME/horario_laboral.sh

```
# El rango 9-17 SI incluye las 17:00, así que son 9 ejecuciones:
# 9, 10, 11, 12, 13, 14, 15, 16 y 17.
# Si quisieras llegar hasta las 18:00, tendrías que poner 9-18.

# 5. Verificar
crontab -l
```

Explicación de rangos de hora:

0 9-17 * * * = Cada hora de 9:00 a 17:00

0 9-17 * * 1-5 = Cada hora de 9 a 17, solo lunes a viernes

30 9-17 * * * = A los 30 minutos de cada hora (9:30, 10:30, etc.)

0 8-18/2 * * * = Cada 2 horas de 8:00 a 18:00 (8, 10, 12, 14, 16, 18)

0 9,12,15,17 * * * = A las 9, 12, 15 y 17 horas

6. Programa un script que muestre “Hoy toca practicar” en un archivo de log solo los lunes, miércoles y viernes a las 8:00 AM.

```
#!/bin/bash

# Nombre: practicar.sh

# Descripción: Recordatorio para practicar lunes, miércoles y viernes

# Archivo de log
archivo="$HOME/recordatorios.log"

# Obtener el día de la semana
dia_semana=$(date '+%A')

# Escribir mensaje
echo "[$(date '+%Y-%m-%d %H:%M:%S')] - $dia_semana - Hoy toca practicar" >>
"$archivo"
Configurar crontab:

# 1. Crear el script
nano ~/practicar.sh
chmod +x ~/practicar.sh

# 2. Probar
~/practicar.sh
cat ~/recordatorios.log

# 3. Editar crontab
crontab -e

# 4. Añadir para lunes (1), miércoles (3) y viernes (5) a las 8:00
AM:
```

0 8 * * 1,3,5 $HOME/practicar.sh

```
# 5. Verificar
crontab -l
```

Explicación de múltiples das:

0 8 * * 1,3,5 = Lunes, miércoles y viernes a las 8:00 AM

0 8 * * 1-5 = Lunes a viernes (toda la semana laboral) a las 8:00 AM

0 8 * * 0,6 = Sábado y domingo a las 8:00 AM

0 8 * * 2,4 = Martes y jueves a las 8:00 AM

30 14 * * 1,3,5 = Lunes, miércoles y viernes a las 14:30

7. Modifica uno de los scripts para que su salida y errores se guarden en cron.log.

```
#!/bin/bash

# Nombre: fecha_hora_logs.sh

# Descripción: Guarda fecha/hora con manejo de salida y errores
archivo="$HOME/fecha_hora_output.log"

# Simulando algunas operaciones que podrían generar salida y errores
echo "=== Inicio de ejecución ==="
echo "Fecha y hora: $(date '+%d/%m/%Y %H:%M:%S')" >> "$archivo"

# Verificar si el archivo se creó correctamente
if [ $? -eq 0 ]; then
    echo "Registro exitoso"
else
    echo "Error al escribir en el archivo" >&2
fi
```

Configurar crontab con redirección:

```
# 1. Crear el script
nano ~/fecha_hora_logs.sh
chmod +x ~/fecha_hora_logs.sh

# 2. Editar crontab
crontab -e

# 3. OPCIÓN 1: Redirigir salida estándar y errores al mismo archivo
* * * * * $HOME/fecha_hora_logs.sh >> $HOME/cron.log 2>&1

# 4. OPCIÓN 2: Redirigir a archivos separados
* * * * * $HOME/fecha_hora_logs.sh >> $HOME/cron_output.log
2>> $HOME/cron_error.log

# 5. OPCIÓN 3: Solo errores a archivo, salida descartada
* * * * * $HOME/fecha_hora_logs.sh > /dev/null 2>> $HOME/cron.log

# 6. Verificar
crontab -l

# 7. Ver los logs
tail -f ~/cron.log
```

Explicación de redirecciones:

comando >> archivo.log 2>&1

>> = Añadir salida estándar al archivo

2>&1 = Redirigir errores (2) a donde va salida estándar (1)

comando > salida.log 2> error.log

> = Sobrescribir salida estándar

2> = Sobrescribir errores

comando &>> archivo.log

```
# &>> = Añadir tanto salida como errores (bash 4+)
comando > /dev/null 2>&1

# Descartar toda salida y errores
```

8. Programa un script que muestre “Sistema OK” y lo ejecute cada 10 minutos.

```
#!/bin/bash

# Nombre: sistema_ok.sh

# Descripción: Monitoreo simple del sistema

# Archivo de log
log_file="$HOME/sistema_monitor.log"

# Obtener información del sistema
carga=$(uptime | awk -F'load average:' '{print $2}' | awk
'{print $1}')
memoria_libre=$(free -h 2>/dev/null | grep Mem | awk '{print
$4}' || echo "N/A")

# Escribir estado
    echo "[$(date '+%Y-%m-%d %H:%M:%S')] Sistema OK - Carga: $carga - Memoria
libre: $memoria_libre" >> "$log_file"
    Versión simplificada:

    #!/bin/bash

    # sistema_ok_simple.sh
    archivo="$HOME/sistema_ok.log"
    echo "[$(date '+%Y-%m-%d %H:%M:%S')] Sistema OK" >> "$archivo"
    Configurar crontab:

    # 1. Crear el script (versión simple)
    nano ~/sistema_ok.sh
    chmod +x ~/sistema_ok.sh

    # 2. Probar
    ~/sistema_ok.sh
    cat ~/sistema_ok.log

    # 3. Editar crontab
    crontab -e

    # 4. Añadir para ejecutar cada 10 minutos:
    */10 * * * * $HOME/sistema_ok.sh

    # 5. Verificar
    crontab -l

    # 6. Monitorear
    tail -f ~/sistema_ok.log
```

Otros intervalos útiles:

*/10 * * * = Cada 10 minutos

*/15 * * * = Cada 15 minutos

0,10,20,30,40,50 * * * * = Cada 10 minutos (alternativa)

0-59/10 * * * * = Cada 10 minutos (alternativa)

9. Crea un script que genere un archivo con la fecha actual. Prográmalo para ejecutarse el primer da de cada mes a medianoche.

```
#!/bin/bash

# Nombre: reporte_mensual.sh

# Descripción: Genera archivo de reporte mensual

# Directorio de reportes
dir_reportes="$HOME/reportes_mensuales"
mkdir -p "$dir_reportes"

# Generar nombre de archivo con fecha
fecha=$(date '+%Y-%m-%d')
mes_anio=$(date '+%Y-%m')
archivo="$dir_reportes/reporte_${mes_anio}.txt"

# Crear contenido del reporte
{
    echo "======================================"
    echo "REPORTE MENSUAL"
    echo "======================================"
    echo "Fecha de generación: $fecha"
    echo "Hora: $(date '+%H:%M:%S')"
    echo ""
    echo "Sistema: $(uname -s)"
    echo "Usuario: $USER"
    echo "Hostname: $(hostname)"
    echo ""
    echo "Directorio home: $HOME"
    echo "Espacio en disco:"
    df -h "$HOME" | tail -1
    echo ""
    echo "======================================"
} > "$archivo"

# Log de ejecución
log_file="$HOME/reportes_mensuales/ejecucion.log"
echo "[$(date '+%Y-%m-%d %H:%M:%S')] Reporte mensual generado: $archivo" >>
"$log_file"
Configurar crontab:

# 1. Crear el script
nano ~/reporte_mensual.sh
chmod +x ~/reporte_mensual.sh

# 2. Probar manualmente
~/reporte_mensual.sh
ls -lh ~/reportes_mensuales/
cat ~/reportes_mensuales/reporte_*.txt

# 3. Editar crontab
crontab -e
```

- ## 4. Añadir para ejecutar el día 1 de cada mes a medianoche:

0 0 1 * * $HOME/reporte_mensual.sh

```
# 5. Verificar
crontab -l
```

Explicación de cron mensual:

0 0 1 * * = Día 1 de cada mes a medianoche

0 0 15 * * = Día 15 de cada mes a medianoche

0 0 1,15 * * = Día 1 y 15 de cada mes a medianoche

0 0 1 1 * = El 1 de enero a medianoche (anualmente)

0 0 1 */3 = Primer día cada 3 meses (trimestral)

Para el **último día del mes** no hay atajo: la `L` que verás en algunos sitios es de Quartz (el planificador de Java) y **el cron de Linux y macOS no la entiende**; da error de sintaxis. Lo que se hace es lanzarlo todos los días candidatos y comprobar dentro si mañana es día 1:

```
55 23 28-31 * * [ "$(date -d tomorrow +\%d)" = "01" ] && $HOME/cierre_mes.sh
```

> **Precaución**
>
> Fíjate en el `\%d`, con barra invertida. **Dentro de un crontab, el símbolo `%` no es un porcentaje: significa «salto de línea»**, y todo lo que venga detrás del primero se le pasa al comando como si lo hubieras tecleado.
>
> Por eso una línea como esta **no funciona** en el crontab:
> `* * * * * echo "$(date +%Y-%m-%d)" >> ~/registro.log`
>
> Hay que escapar cada `%` con `\`, o —mucho mejor y más legible— **meter el comando dentro de un script `.sh` y llamar al script desde el crontab**. Dentro del `.sh` el `%` se comporta con normalidad.
>
> Es la causa número uno de «mi cron no funciona y no sé por qué».

10. Configura un script que escriba “Probando cron” en un archivo. Después, buscar información para revisar los logs del sistema y confirmar su ejecución.

```
#!/bin/bash

# Nombre: probar_cron.sh

# Descripción: Script de prueba para verificar cron

# Archivo de salida
archivo="$HOME/prueba_cron.log"

# Escribir mensaje
echo "[$(date '+%Y-%m-%d %H:%M:%S')] Probando cron - Ejecución exitosa" >>
"$archivo"
```

- ## Escribir también en salida estándar para que aparezca en logs de

  cron

- echo “Cron ejecutado correctamente a las $(date ‘+%H:%M:%S’)”

Configurar y verificar:

```
# 1. Crear el script
nano ~/probar_cron.sh
chmod +x ~/probar_cron.sh

# 2. Probar manualmente
~/probar_cron.sh
cat ~/prueba_cron.log

# 3. Editar crontab
crontab -e

# 4. Añadir para ejecutar cada 5 minutos (para prueba rápida):
*/5 * * * * $HOME/probar_cron.sh

# 5. Verificar que se añadió
crontab -l
```

Revisar logs del sistema para confirmar ejecución:

En Linux:

Ver logs de cron (varias ubicaciones según distribución)

> sudo tail -f /var/log/cron
> sudo tail -f /var/log/cron.log

> sudo tail -f /var/log/syslog | grep CRON

Buscar ejecuciones de tu script

> grep probar_cron /var/log/syslog
> grep CRON /var/log/syslog | grep $USER

Ver logs con journalctl (systemd)

> sudo journalctl -u cron.service -f
> sudo journalctl -u cron.service –since “10 minutes ago”

En macOS:

Ver logs de cron en macOS

log show –predicate ‘process == “cron”’ –last 1h

log show –predicate ‘eventMessage contains “probar_cron”’ –last 30m

```
# Logs del sistema
tail -f /var/log/system.log | grep cron

# Verificar que cron está corriendo
    ps aux | grep cron
```

Comandos útiles para debugging:

```
# Ver tu crontab actual
crontab -l

# Ver el archivo de salida del script
cat ~/prueba_cron.log

# Ver en tiempo real las nuevas líneas
tail -f ~/prueba_cron.log

# Verificar permisos del script
ls -l ~/probar_cron.sh

# Verificar que cron está activo

# Linux:
sudo systemctl status cron

# o
sudo service cron status

# macOS:
sudo launchctl list | grep cron

# Ver todas las tareas cron del sistema (root)
sudo crontab -l

# Editar variables de entorno en crontab (si necesario)
crontab -e

# Añadir al inicio:

# PATH=/usr/local/bin:/usr/bin:/bin

    # SHELL=/bin/bash
```

Troubleshooting command:

```
# Si cron no ejecuta el script:

# 1. Verificar que el script tiene permisos de ejecución
chmod +x ~/probar_cron.sh

# 2. Usar rutas absolutas en crontab

# Correcto: /home/usuario/probar_cron.sh

# Incorrecto: ~/probar_cron.sh

# 3. Redirigir salida y errores para debug
*/5 * * * * $HOME/probar_cron.sh >> $HOME/cron_debug.log
2>&1

# 4. Verificar que el servicio cron está corriendo
    sudo systemctl start cron # Linux

    # En macOS cron siempre está activo

    # 5. Verificar sintaxis del crontab

    # Después de editar, si hay errores, crontab te avisará
```

PROCESAR TEXTO

En el capítulo 4 aprendiste a **encontrar** líneas con `grep`. Este capítulo va del paso siguiente: **transformarlas**. Quedarte con una columna, ordenar, contar cuántas veces se repite algo, sustituir texto.

Es lo que convierte un pipe en una respuesta. Con `grep` sabes *qué líneas hablan de errores*; con lo de aquí sabes *qué usuario dio más errores y cuántos*.

El texto como filas y columnas

Casi todo en Unix es texto plano en líneas: los registros del sistema, la salida de `ls -l`, un CSV exportado de una hoja de cálculo, la respuesta de un programa. Y todas las herramientas de este capítulo comparten la misma idea:

- Leen **una línea cada vez**.
- La parten en **campos** (columnas) por un separador.
- Escriben el resultado en la salida estándar, para que lo recoja el siguiente comando del pipe.

Por eso encajan unas con otras. Ninguna hace mucho por sí sola; juntas hacen casi cualquier cosa.

Para todos los ejemplos vamos a usar el mismo archivo, `ventas.csv`. Créalo para poder ir probando:

```
cat > ventas.csv << 'FIN'
fecha,vendedor,producto,unidades,precio
2026-07-01,Nica,teclado,3,25
2026-07-01,Marta,raton,5,12
2026-07-02,Leo,monitor,1,180
2026-07-02,Nica,raton,2,12
2026-07-03,Marta,teclado,4,25
2026-07-03,Nica,monitor,2,180
2026-07-04,Leo,raton,7,12
2026-07-04,Nica,teclado,1,25
FIN
```

> **Nota**
>
> Ese `cat > archivo << 'FIN'` es un *heredoc*: escribe en el archivo todo lo que venga hasta encontrar la palabra `FIN` sola en una línea. Es la forma cómoda de crear un archivo de varias líneas desde la terminal.

Quedarse con una columna: cut

`cut` recorta trozos de cada línea. Su uso más habitual es por campos:

- `-d` indica el **d**elimitador (el separador).
- `-f` indica qué **f**ield (campo) quieres, empezando por 1.

```
cut -d',' -f2 ventas.csv
```

Y sale:

```
vendedor
Nica
Marta
Leo
Nica
Marta
Nica
Leo
Nica
```

Puedes pedir varios campos, o un rango:

```
cut -d',' -f2,4 ventas.csv      # el 2 y el 4
cut -d',' -f3- ventas.csv       # del 3 hasta el final
cut -d',' -f-2 ventas.csv       # desde el principio hasta el 2
```

También sabe cortar por posición de carácter, con `-c`:

```
cut -c1-10 ventas.csv
```

Y sale:

```
fecha,vend
2026-07-01
2026-07-01
```

Fíjate en que la **primera línea es la cabecera** y se cuela en todos los resultados. Para quitarla, `tail -n +2` («desde la línea 2 en adelante»):

```
tail -n +2 ventas.csv | cut -d',' -f2
```

Ese `tail -n +2` va a aparecer mucho en el capítulo.

Dónde falla cut

`cut` corta por **un** carácter separador. Si el separador son espacios y hay un número variable de ellos, se descoloca. El caso típico es `ls -l`, que alinea las columnas rellenando con espacios:

```
ls -l
```

Y sale:

```
total 200
-rw-r--r-- 1 ndetomas ndetomas 200000 Jul 31 15:02 grande.bin
-rw-r--r-- 1 ndetomas ndetomas      5 Jul 31 15:02 pequeno.txt
```

El tamaño parece la quinta columna, así que uno diría `cut -d' ' -f5`:

```
ls -l | cut -d' ' -f5
```

Y sale:

```
200000
```

El primer archivo sale bien y el segundo sale **vacío**. Porque en la línea de `pequeno.txt` el tamaño va precedido de varios espacios, y para `cut` cada espacio abre un campo nuevo: el quinto campo de esa línea es un espacio en blanco.

Hay dos soluciones. La primera, comprimir los espacios repetidos con `tr -s` antes de cortar (lo verás dentro de un momento). La segunda, usar `awk`, que trata los espacios seguidos como un único separador y acierta a la primera:

```
ls -l | awk '{print $5}'
```

Y sale:

```
200000
5
```

> **Consejo**
>
> Regla rápida: si el separador es un carácter fijo (una coma, dos puntos, un tabulador), `cut`. Si son espacios, `awk`.

Ordenar: sort

`sort` ordena las líneas. Por defecto, **alfabéticamente**:

```
tail -n +2 ventas.csv | cut -d',' -f2 | sort
```

Y sale:

```
Leo
Leo
Marta
Marta
Nica
Nica
Nica
Nica
```

Y ahí está la trampa más famosa de `sort`: alfabéticamente, el `1` de `100` va antes que el `9`.

```
printf '10\n9\n100\n2\n' | sort
```

Y sale:

```
10
100
2
9
```

Con `-n` (numérico) ordena como esperas:

```
printf '10\n9\n100\n2\n' | sort -n
```

Y sale:

```
2
9
10
100
```

Las opciones que se usan de verdad:

Opción	Qué hace
`-n`	Ordena como números, no como texto
`-r`	Al revés (de mayor a menor)
`-u`	Quita los repetidos (unique)
`-t`	Indica el separador de campos
`-k`	Ordena por un campo concreto, no por la línea entera
`-h`	Entiende tamaños tipo `4K`, `2M`, `1G`

Para ordenar el CSV por unidades vendidas, el separador es la coma (`-t','`) y el campo es el cuarto:

```
tail -n +2 ventas.csv | sort -t',' -k4,4n
```

Y sale:

```
2026-07-02,Leo,monitor,1,180
2026-07-04,Nica,teclado,1,25
2026-07-02,Nica,raton,2,12
2026-07-03,Nica,monitor,2,180
2026-07-01,Nica,teclado,3,25
2026-07-03,Marta,teclado,4,25
2026-07-01,Marta,raton,5,12
2026-07-04,Leo,raton,7,12
```

> **Nota**
> Se escribe `-k4,4n` y no `-k4n` por un motivo: `-k4` significa «desde el campo 4 **hasta el final de la línea**». `-k4,4` significa «solo el campo 4». La `n` pegada al final aplica el orden numérico a ese campo. Con `-k4n` a veces sale bien por casualidad y otras no, según lo que haya en las columnas siguientes.

Contar repetidos: uniq

`uniq` quita líneas repetidas. Pero con una condición que sorprende a todo el mundo la primera vez: **solo compara líneas contiguas**.

```
printf 'Nica\nMarta\nNica\n' | uniq
```

Y sale:

```
Nica
Marta
Nica
```

No ha quitado nada, porque los dos `Nica` no estaban pegados. Por eso `uniq` casi siempre va detrás de un `sort`:

```
printf 'Nica\nMarta\nNica\n' | sort | uniq
```

Y sale:

```
Marta
Nica
```

Lo interesante viene con `-c`, que además **cuenta** cuántas veces aparecía cada línea:

```
tail -n +2 ventas.csv | cut -d',' -f2 | sort | uniq -c
```

Y sale:

```
      2 Leo
      2 Marta
      4 Nica
```

Opción	Qué hace
`-c`	Antepone el número de repeticiones

Opción	Qué hace
`-d`	Solo muestra las líneas que estaban repetidas
`-u`	Solo las que aparecían una única vez
`-i`	No distingue mayúsculas de minúsculas

La receta del «top»

Esta combinación de tres comandos es de las más útiles que vas a aprender, y se usa igual para vendedores, para IPs de un registro o para tipos de error:

```
... | sort | uniq -c | sort -rn
```

Se lee así: **ordena** (para que los iguales queden juntos), **cuenta** los grupos, y **vuelve a ordenar** por ese recuento de mayor a menor.

```
tail -n +2 ventas.csv | cut -d',' -f2 | sort | uniq -c | sort -rn
```

Y sale:

```
      4 Nica
      2 Marta
      2 Leo
```

Añade `| head -10` al final y tienes un «top 10».

Sustituir y borrar caracteres: tr

`tr` (*translate*) trabaja **carácter a carácter**, no con palabras. Recibe dos listas y cambia cada carácter de la primera por el que le toca en la segunda.

```
echo "hola nica" | tr 'a-z' 'A-Z'
```

Y sale:

```
HOLA NICA
```

Sus tres usos habituales:

```
echo "1.234.567" | tr -d '.'              # -d borra: 1234567
echo "hola     mundo" | tr -s ' '         # -s comprime: hola mundo
head -2 ventas.csv | tr ',' '\n'          # una columna por línea
```

`tr -s ' '` es la solución al problema de `cut` con `ls -l` que veíamos antes:

```
ls -l | tr -s ' ' | cut -d' ' -f5
```

Ahora sí devuelve los dos tamaños, porque los espacios repetidos se han convertido en uno solo.

> **Precaución**

> `tr` no entiende de palabras. `tr 'raton' 'RATON'` no cambia la palabra «raton»: cambia cada `r` por `R`, cada `a` por `A`, y así con cada letra, **en cualquier parte del texto**. Para sustituir palabras, `sed`.

Sustituir texto: sed

`sed` (*stream editor*) edita el texto según va pasando. Tiene un lenguaje entero detrás, pero con cuatro cosas cubres casi todo.

Sustituir

La orden `s` es la que se usa el 90% de las veces:

```
sed 's/raton/ratón/' ventas.csv
```

Se lee: **sustituye** `raton` por `ratón`. Las barras solo separan las partes.

Por defecto cambia **una vez por línea**. Con la `g` final (*global*), todas:

```
echo "uno dos uno dos" | sed 's/uno/UNO/'
echo "uno dos uno dos" | sed 's/uno/UNO/g'
```

Y sale:

```
UNO dos uno dos
UNO dos UNO dos
```

Borrar líneas

La orden `d` borra:

```
sed '1d' ventas.csv          # borra la línea 1 (la cabecera)
sed '/Leo/d' ventas.csv      # borra las líneas que contengan "Leo"
```

Mostrar solo lo que interesa

`-n` le dice a `sed` que no imprima nada por su cuenta, y la orden `p` imprime lo que le pidas. Juntos son un `grep`:

```
sed -n '/Nica/p' ventas.csv
sed -n '2,4p' ventas.csv     # solo las líneas 2 a 4
```

Editar el archivo de verdad

Todo lo anterior muestra el resultado por pantalla y **no toca el archivo**. Para modificarlo está `-i` (*in place*).

> **Precaución**
>
> `sed -i` reescribe el archivo sin preguntar y sin papelera. Si la sustitución estaba mal, el original ya no existe.

> Dos costumbres que evitan el disgusto: **ejecuta siempre primero sin** `-i` para ver qué saldría, y usa `-i.bak`, que guarda una copia del original antes de tocarlo.

```
sed 's/raton/ratón/g' ventas.csv        # 1. mirar el resultado
sed -i.bak 's/raton/ratón/g' ventas.csv # 2. aplicarlo, con copia de seguridad
```

Después de eso tienes el archivo cambiado y un `ventas.csv.bak` con el original intacto.

Columnas de verdad: awk

`awk` es el más potente de todos, y en realidad es un lenguaje de programación completo. Pero se puede usar como una herramienta más, aprendiendo cuatro cosas.

Su idea básica: por cada línea, `awk` la parte en campos y les pone nombre.

Variable	Qué es
`$1`, `$2`, `$3`…	El primer campo, el segundo, el tercero…
`$0`	La línea entera
`NF`	El número de fields (cuántas columnas tiene la línea)
`NR`	El número de registro: por qué línea va

Por defecto separa por espacios (y le dan igual los espacios repetidos, de ahí que acertara con `ls -l`). Con `-F` le indicas otro separador:

```
awk -F',' '{print $2}' ventas.csv
awk -F',' '{print $2, $4}' ventas.csv
```

Y sale:

```
vendedor unidades
Nica 3
Marta 5
Leo 1
```

La coma dentro del `print` mete un espacio entre los dos valores.

Filtrar

Antes de las llaves puedes poner una **condición**, y `awk` solo aplica el bloque a las líneas que la cumplen:

```
awk -F',' '$4 > 3 {print $2, $3, $4}' ventas.csv
```

Y sale:

```
vendedor producto unidades
Marta raton 5
Marta teclado 4
Leo raton 7
```

La cabecera se ha colado. No es un error de `awk`: al comparar el texto `unidades` con el número `3`, `awk` los compara como texto, y `unidades` alfabéticamente va después de `3`. Se arregla exigiendo además que no sea la primera línea:

```
awk -F',' 'NR > 1 && $4 > 3 {print $2, $3, $4}' ventas.csv
```

Y sale:

```
Marta raton 5
Marta teclado 4
Leo raton 7
```

También puedes filtrar por texto, entre barras, como en `grep`:

```
awk -F',' '/Leo/ {print $0}' ventas.csv
```

Calcular

Aquí es donde `awk` se separa del resto: sabe hacer cuentas.

```
awk -F',' 'NR > 1 {print $3, $4 * $5}' ventas.csv
```

Y sale:

```
teclado 75
raton 60
monitor 180
raton 24
teclado 100
monitor 360
raton 84
teclado 25
```

Y con el bloque `END`, que se ejecuta **una sola vez al terminar**, puedes acumular totales:

```
awk -F',' 'NR > 1 {total += $4} END {print "Total unidades:", total}' ventas.csv
```

Y sale:

```
Total unidades: 25
```

> **Nota**
>
> No hace falta declarar `total` ni ponerlo a cero: `awk` supone que una variable nueva usada en una suma empieza valiendo 0. Es una de las pocas cosas en las que es más cómodo que Bash.

Todo junto

Vamos con una pregunta de verdad: **¿qué vendedor ha ingresado más dinero?**

No hay un comando que lo responda. Hay que construirlo por tramos, y la forma de hacerlo es **ir añadiendo un comando cada vez y mirar la salida**.

Primero, quitar la cabecera:

```
tail -n +2 ventas.csv
```

Ahora, por cada línea, el vendedor y lo que facturó (unidades por precio):

```
tail -n +2 ventas.csv | awk -F',' '{print $2, $4 * $5}'
```

Y sale:

```
Nica 75
Marta 60
Leo 180
Nica 24
Marta 100
Nica 360
Leo 84
Nica 25
```

Falta sumar por vendedor. `awk` puede guardar totales separados usando el nombre como índice, y volcarlos al final:

```
tail -n +2 ventas.csv \
  | awk -F',' '{suma[$2] += $4 * $5} END {for (v in suma) print suma[v], v}'
```

Y sale:

```
Marta 160
Leo 264
Nica 484
```

Solo queda ordenar de mayor a menor:

```
tail -n +2 ventas.csv \
  | awk -F',' '{suma[$2] += $4 * $5} END {for (v in suma) print suma[v], v}' \
  | sort -rn
```

Y sale:

```
484 Nica
264 Leo
160 Marta
```

Fíjate en dos detalles del resultado. El primero, que se imprime `suma[v], v` (primero el número y luego el nombre) precisamente **para poder ordenar después**: `sort -rn` mira el principio de la línea. El segundo, que la barra invertida al final de la línea permite partir un pipe largo en varias líneas para que se lea mejor.

Cuándo usar cuál

Herramienta	Úsala para
`cut`	Sacar columnas cuando el separador es un carácter fijo

Herramienta	Úsala para
`sort`	Ordenar, y para preparar la entrada de `uniq`
`uniq`	Contar repeticiones (siempre después de `sort`)
`tr`	Cambiar o borrar **caracteres** sueltos, comprimir espacios
`sed`	Sustituir o borrar **texto** línea a línea
`awk`	Columnas separadas por espacios, filtros con condiciones y cuentas

La regla práctica: empieza por lo simple. Si `cut` te vale, usa `cut`. Cuando necesites una condición o una suma, pasa a `awk`.

Ejercicios

1. Muestra la lista de productos vendidos, sin repetir y ordenada alfabéticamente.
2. Averigua cuál es el producto que más veces aparece en el archivo, usando la receta del «top».
3. Calcula el total de dinero facturado en todo el archivo (unidades × precio, sumando todas las líneas).
4. Saca las ventas del día 2026-07-03 mostrando solo vendedor y producto, separados por un guion en vez de por una coma.

Soluciones

1. Cortar la columna del producto, quitar la cabecera y ordenar sin repetidos:

```
tail -n +2 ventas.csv | cut -d',' -f3 | sort -u
```

2. La misma columna, pero con la receta completa:

```
tail -n +2 ventas.csv | cut -d',' -f3 | sort | uniq -c | sort -rn | head -1
```

3. Es la suma del capítulo, pero multiplicando antes:

```
awk -F',' 'NR > 1 {total += $4 * $5} END {print "Facturado:", total}' ventas.csv
```

4. Se filtra por la fecha y se imprime con el separador que quieras. La variable `OFS` (*output field separator*) es lo que `awk` pone entre los valores del `print`:

```
awk -F',' -v OFS=' - ' '/2026-07-03/ {print $2, $3}' ventas.csv
```

También vale sin `OFS`, escribiendo el guion a mano:

```
awk -F',' '/2026-07-03/ {print $2 " - " $3}' ventas.csv
```

SCRIPTS QUE NO SE ROMPEN

El capítulo 9 te explicó **por qué** fallan los scripts: comillas, expansión y palabras que se parten donde no debían. Este capítulo va de lo que se hace con eso: conseguir que un script, cuando falle, **falle pronto y se note**, en vez de seguir adelante haciendo daño.

Es la diferencia entre un script que usas tú una tarde y uno que dejas programado en cron y no vuelves a mirar.

El script que hace daño en silencio

Mira este script de cuatro líneas. Su idea es entrar en una carpeta de trabajo y borrar los archivos temporales:

```
#!/bin/bash
cd /carpeta/que/no/existe
rm -f *.tmp
echo "Limpieza terminada"
```

Guárdalo como `malo.sh`. Ahora crea una carpeta con cosas dentro y ejecútalo **desde ahí**:

```
mkdir zona && cd zona
touch a.tmp b.tmp datos.txt
ls
bash ../malo.sh
echo "código de salida: $?"
ls
```

Y sale:

```
a.tmp
b.tmp
datos.txt
../malo.sh: line 2: cd: /carpeta/que/no/existe: No such file or directory
Limpieza terminada
código de salida: 0
datos.txt
```

Lee eso despacio, porque han pasado tres cosas y las tres son malas:

1. El `cd` **falló**. Lo dijo, pero nadie lo escuchó.
2. Como el `cd` no funcionó, el script seguía en tu carpeta. Y el `rm -f *.tmp` **se ejecutó ahí**: ha borrado `a.tmp` y `b.tmp`, que no era lo que querías.
3. El script terminó diciendo «Limpieza terminada» y devolviendo **código 0**, es decir, *éxito*. Si esto estuviera en cron, no te enterarías nunca.

Un error se convirtió en un borrado en el sitio equivocado, y encima se informó como si todo hubiera ido bien. Todo el capítulo va de evitar esto.

Las tres líneas mágicas: set -euo pipefail

La primera medida cabe en una línea, y es la que más problemas evita:

```
#!/bin/bash
set -euo pipefail
```

Son tres opciones independientes. Conviene entender qué hace cada una por separado, porque ninguna es magia.

-e: parar al primer error

Con `set -e`, si un comando devuelve un código distinto de cero, el script **termina ahí mismo**.

```
#!/bin/bash
set -e
cd /carpeta/que/no/existe
rm -f *.tmp
echo "Limpieza terminada"
```

Al ejecutarlo:

```
bueno.sh: line 3: cd: /carpeta/que/no/existe: No such file or directory
código de salida: 1
```

El `rm` no llegó a ejecutarse, no se borró nada, y el script devolvió 1. Con una sola línea, el desastre de antes no ocurre.

-u: variable sin definir es un error

Por defecto, una variable que no existe se sustituye por **nada**, en silencio. Y eso, combinado con rutas, es peligroso:

```
#!/bin/bash
destino="/tmp/copia_seguridad"
echo "Borrando $destino/$carpeta"
```

Al ejecutarlo:

```
Borrando /tmp/copia_seguridad/
```

La variable `carpeta` no existía (a lo mejor la escribiste `Carpeta` en otro sitio), así que la ruta se quedó a medias. Cambia ese `echo` por un `rm -rf` e imagina el resultado.

Con `set -u`, Bash se planta:

```
u2.sh: line 4: carpeta: unbound variable
código de salida: 1
```

> **Consejo**
> `set -u` es el que más erratas caza, porque los nombres de variable mal escritos no dan ningún síntoma hasta que es tarde.

-o pipefail: que un pipe no mienta

Esta es la más sutil. Cuando encadenas comandos con `|`, Bash toma como código de salida **el del último comando**, no el de todos. Así que si falla el primero pero el último va bien, el pipe entero parece un éxito:

```
#!/bin/bash
set -e
cat /fichero/inexistente | wc -l
echo "El pipe fue bien, sigo adelante"
```

Al ejecutarlo:

```
cat: /fichero/inexistente: No such file or directory
0
El pipe fue bien, sigo adelante
código de salida: 0
```

`cat` falló, pero `wc -l` contó cero líneas y terminó tan contento. Ni siquiera `set -e` lo detiene, porque para Bash el pipe fue bien.

Con `pipefail`, el pipe falla si **cualquiera** de sus comandos falla:

```
#!/bin/bash
set -e -o pipefail
cat /fichero/inexistente | wc -l
echo "El pipe fue bien, sigo adelante"
```

Al ejecutarlo:

```
cat: /fichero/inexistente: No such file or directory
0
código de salida: 1
```

Como en este curso todo se hace a base de pipes (y el capítulo 12 va entero de eso), `pipefail` importa más de lo que parece.

Cuándo -e no te salva

Aquí viene la parte honesta, que casi nadie cuenta: `set -e` **tiene agujeros**. Si crees que te protege siempre, te confiarás justo donde no debes.

`set -e` **no** detiene el script en estos cuatro casos:

1. Dentro de la condición de un if. Es lógico: un `if` existe precisamente para comprobar si algo falla.

```
#!/bin/bash
set -e
if grep "nada" /fichero/inexistente; then
    echo "encontrado"
else
    echo "no encontrado, pero el script SIGUE VIVO"
fi
```

2. A la izquierda de un `&&`.

```
#!/bin/bash
set -e
false && echo "esto no se ve"
echo "y aquí seguimos, aunque false ha fallado"
```

3. Cuando tú mismo capturas el fallo con `||`. Esto no es un agujero, es lo que quieres: le estás diciendo a Bash que ya te ocupas tú.

```
cat /fichero/inexistente || echo "lo he capturado yo"
```

4. Dentro de una función que se llama desde un `if`. Este es el peligroso de verdad:

```
#!/bin/bash
set -e
comprobar() {
    false
    echo "OJO: esta línea se ejecuta"
}
if comprobar; then echo "fue bien"; else echo "falló"; fi
```

Al ejecutarlo:

```
OJO: esta línea se ejecuta
fue bien
```

El `false` no detuvo la función, la función siguió, y encima devolvió *éxito* (porque el código de salida de una función es el de su última orden, que era el `echo`). Dentro de un `if`, `set -e` queda desactivado en **todo** lo que haya debajo.

> **Precaución**
>
> Moraleja: `set -euo pipefail` es la primera línea de defensa, no la única. Cuando algo es importante de verdad —borrar, copiar sobre otra cosa, subir a un servidor— compruébalo tú explícitamente en vez de dar por hecho que el script se habría parado.

Valores por defecto

Un script útil suele tener parámetros opcionales. La expansión de parámetros resuelve eso sin escribir un `if` por cada uno.

Escritura	Qué hace

Escritura	Qué hace
`${var:-valor}`	Usa `valor` si `var` está vacía o no existe. **No la modifica**
`${var:=valor}`	Igual, pero además **guarda** ese valor en `var`
`${var:?mensaje}`	Si está vacía, aborta el script con ese mensaje
`${var:+otro}`	Al revés: usa `otro` solo **si** `var` tiene algo

```
destino="${1:-/tmp/copias}"      # si no pasan argumento, /tmp/copias
echo "Voy a copiar a $destino"
```

El tercero es el que conviene conocer para las cosas que **no** son opcionales. En vez de escribir la comprobación, la exiges:

```
#!/bin/bash
: "${API_TOKEN:?hace falta definir API_TOKEN}"
echo "el token es $API_TOKEN"
```

Al ejecutarlo:

```
tok.sh: line 2: API_TOKEN: hace falta definir API_TOKEN
código de salida: 1
```

Y con la variable puesta, funciona:

```
API_TOKEN=abc123 bash tok.sh
```

Y sale:

```
el token es abc123
```

> **Nota**
> Los dos puntos sueltos del principio (`:`) son un comando de Bash que no hace nada. Se usa aquí solo como percha para colgar la expansión: nos interesa el efecto (comprobar y abortar), no imprimir nada.

Recortar cadenas

La misma sintaxis de las llaves sirve para trocear texto sin llamar a `sed` ni a `cut`. Es más rápido y no monta un pipe por cada archivo.

Escritura	Qué hace
`${#var}`	Cuántos caracteres tiene
`${var%patron}`	Quita el patrón por el **final** (el trozo más corto)
`${var%%patron}`	Igual, pero el trozo más largo

Escritura	Qué hace
`${var#patron}`	Quita el patrón por el **principio** (el más corto)
`${var##patron}`	Igual, pero el más largo
`${var/viejo/nuevo}`	Sustituye la **primera** aparición
`${var//viejo/nuevo}`	Sustituye **todas**

Regla para acordarse de cuál es cuál: en un teclado, `#` está a la izquierda del `$` y quita por la izquierda; `%` está a la derecha y quita por la derecha.

```
ruta="/home/nica/documentos/informe.txt"
nombre="informe.txt"

echo "${#nombre}"         # 11
echo "${nombre%.txt}"     # informe
echo "${nombre%.*}"       # informe   (cualquier extensión)
echo "${ruta##*/}"        # informe.txt   (solo el archivo)
echo "${ruta%/*}"         # /home/nica/documentos   (solo la carpeta)

frase="uno dos uno dos"
echo "${frase/uno/UNO}"  # UNO dos uno dos
echo "${frase//uno/UNO}" # UNO dos UNO dos
```

El uso típico es renombrar en bloque:

```
for f in *.txt; do
    mv -- "$f" "${f%.txt}.md"
done
```

Fíjate en las comillas de `"${f%.txt}.md"`: sin ellas, un archivo llamado `mis apuntes.txt` volvería a romper el script, exactamente como en el capítulo 9. Y el `--` le dice a `mv` que lo que viene después son nombres de archivo, aunque empiecen por guion.

Arrays

Un array es una variable que guarda **una lista** de valores en vez de uno solo. Es lo que necesitas cuando manejas «los servidores», «los archivos a copiar» o «las carpetas a revisar».

```
servidores=("web1" "base de datos" "correo")

echo "${#servidores[@]}"    # cuántos hay: 3
echo "${servidores[0]}"     # el primero: web1  (se empieza a contar en 0)

servidores+=("respaldo")    # añadir uno al final
```

Para recorrerlo, `for` con `"${array[@]}"`. **Y las comillas no son opcionales:**

```
for s in "${servidores[@]}"; do
    echo "[$s]"
done
```

Y sale:

```
[web1]
[base de datos]
[correo]
```

Quítalas y vuelve el word splitting del capítulo 9: el elemento «base de datos» se parte en tres.

```
for s in ${servidores[@]}; do
    echo "[$s]"
done
```

Y sale:

```
[web1]
[base]
[de]
[datos]
[correo]
```

> **Precaución**
>
> `"${array[@]}"` con comillas y con `@` es **la única forma correcta** de recorrer un array. Con `*` en vez de `@` se junta todo en una sola cadena, y sin comillas se parte por los espacios. Apréndete la forma buena y no le des más vueltas.

Si además necesitas la posición de cada elemento, `"${!array[@]}"` da los índices:

```
for i in "${!servidores[@]}"; do
    echo "$i -> ${servidores[$i]}"
done
```

Limpiar al salir: trap

Muchos scripts crean archivos temporales. El problema es qué pasa con ellos cuando el script muere a mitad, o cuando pulsas Ctrl+C.

Primero, la forma correcta de crear un temporal es `mktemp`, que se inventa un nombre único (dos ejecuciones a la vez no se pisan):

```
tmp="$(mktemp)"
```

Y ahora el problema. Este script no llega a borrar su temporal:

```
#!/bin/bash
set -euo pipefail
tmp="$(mktemp)"
echo "datos" > "$tmp"
cat /fichero/inexistente     # aquí muere
rm -f "$tmp"                 # esta línea no se ejecuta nunca
```

El temporal se queda en `/tmp` para siempre. Repite eso cada hora en cron y en un mes tienes miles.

`trap` resuelve esto: registra una orden para que Bash la ejecute **pase lo que pase** cuando el script termine.

```
#!/bin/bash
set -euo pipefail
tmp="$(mktemp)"
trap 'rm -f "$tmp"' EXIT

echo "datos" > "$tmp"
cat /fichero/inexistente
```

Ahora el script sigue fallando (eso está bien, queremos que falle), pero el temporal desaparece igualmente.

La señal `EXIT` significa «cuando el script termine, por el motivo que sea»: final normal, `exit`, error con `set -e` o Ctrl+C. Por eso es la que se usa el 99% de las veces.

Señal	Cuándo salta
`EXIT`	Siempre que el script termina, sea como sea
`INT`	Ctrl+C
`TERM`	Cuando alguien lo mata con `kill`
`ERR`	Cuando un comando falla (útil para mensajes de diagnóstico)

> **Nota**
>
> El `trap` se escribe entre **comillas simples** a propósito. Así la orden se guarda tal cual y `$tmp` se sustituye cuando el trap se ejecuta, no cuando se registra. Con comillas dobles funcionaría también en este caso, pero se rompe en cuanto la variable cambie durante el script.

Comprobar antes de hacer

Ya tienes las piezas. Falta el hábito: **validar al principio y salir con un mensaje claro**. Este patrón de tres líneas es el que uso en todos los scripts:

```
morir() {
    echo "Error: $1" >&2
    exit 1
}

[ "$#" -eq 2 ] || morir "uso: $(basename "$0") ORIGEN DESTINO"
[ -d "$1" ]    || morir "el origen '$1' no es una carpeta"
```

Dos detalles importantes:

- El `>&2` manda el mensaje al **canal de errores** (el capítulo 4 explicaba los tres canales). Así, si alguien redirige la salida del script a un archivo, los errores siguen viéndose en pantalla.
- El `exit 1` devuelve un código de error. Es lo que permite que quien llame a tu script —otro script, o cron— se entere de que falló.

```
Error: uso: val.sh ORIGEN DESTINO
código de salida: 1
```

ShellCheck

ShellCheck es un programa que lee tus scripts y te avisa de los fallos **antes** de ejecutarlos. Detecta justo lo de este capítulo y el 9: comillas que faltan, variables mal escritas, comparaciones que no hacen lo que crees.

```
sudo apt install shellcheck
shellcheck mi-script.sh
```

Vamos a pasárselo a un script con los fallos típicos:

```
#!/bin/bash
origen=$1
destino=$2

for archivo in $(ls $origen); do
    cp $origen/$archivo $destino
done

echo "Copiados $numero archivos"
```

Este script *parece* correcto y funciona mientras las carpetas no tengan espacios. Esto es lo que dice ShellCheck:

```
In copiar.sh line 5:
for archivo in $(ls $origen); do
               ^-----------^ SC2045 (error): Iterating over ls output is fragile.
Use globs.
                     ^-----^ SC2086 (info): Double quote to prevent globbing and
word splitting.

In copiar.sh line 6:
    cp $origen/$archivo $destino
       ^-----^ SC2086 (info): Double quote to prevent globbing and word splitting.
               ^------^ SC2086 (info): Double quote to prevent globbing and word
splitting.
                        ^------^ SC2086 (info): Double quote to prevent globbing
and word splitting.

In copiar.sh line 9:
echo "Copiados $numero archivos"
               ^-----^ SC2154 (warning): numero is referenced but not assigned.
```

Ha encontrado las tres cosas: que recorrer la salida de `ls` es frágil (mejor un comodín), las cinco variables sin comillas del capítulo 9, y una variable `numero` que nunca se asigna —una errata que habría impreso «Copiados archivos» sin más—. Además te da la dirección de una página que explica cada aviso con detalle.

Si usas VS Code, hay una extensión que lo va marcando mientras escribes, que es todavía más cómodo.

No sustituye a entender lo que hace tu script, pero encuentra en dos segundos cosas que de otra forma descubres dentro de un mes, un domingo, cuando la tarea de cron lleva semanas fallando en silencio.

El script del principio, arreglado

Volvamos al script de la primera sección y apliquemos todo:

```
#!/bin/bash
set -euo pipefail

morir() {
    echo "Error: $1" >&2
    exit 1
}

[ "$#" -eq 1 ] || morir "uso: $(basename "$0") CARPETA"
carpeta="$1"
[ -d "$carpeta" ] || morir "'$carpeta' no es una carpeta"

cd "$carpeta"

archivos=( *.tmp )
if [ ! -e "${archivos[0]}" ]; then
    echo "No hay temporales que borrar en $(pwd)"
    exit 0
fi

echo "Voy a borrar ${#archivos[@]} archivos de $(pwd)"
rm -f -- "${archivos[@]}"
echo "Limpieza terminada"
```

Compara los dos comportamientos:

```
--- sin argumento ---
Error: uso: limpiar.sh CARPETA
código de salida: 1
--- carpeta inexistente ---
Error: '/no/existe' no es una carpeta
código de salida: 1
--- carpeta sin temporales ---
No hay temporales que borrar en /home/nica/pruebas/vacia
código de salida: 0
--- carpeta con temporales ---
Voy a borrar 2 archivos de /home/nica/pruebas/zona
Limpieza terminada
código de salida: 0
```

Nunca borra en la carpeta equivocada, dice **dónde** va a borrar y **cuántos** archivos antes de hacerlo, distingue «no había nada que borrar» (que no es un error) de «me has dado una carpeta que no existe» (que sí lo es), y funciona con nombres que llevan espacios.

> **Consejo**
>
> El `archivos=( *.tmp )` seguido de `[ ! -e "${archivos[0]}" ]` resuelve una trampa clásica: cuando un comodín no encuentra nada, Bash deja el `*.tmp` literal en vez de una lista vacía. Comprobar si el primer elemento existe de verdad es la forma de distinguir «no hay archivos» de «hay uno que se llama así».

Ejercicios

1. Coge un script cualquiera de los capítulos anteriores, añádele `set -euo pipefail` y ejecútalo. Si deja de funcionar, averigua por qué: casi siempre es una variable sin definir que llevaba ahí desde el principio.
2. Escribe un script que reciba un nombre de archivo y muestre por separado su nombre sin extensión y su extensión, usando solo expansión de parámetros (nada de `cut` ni `sed`).
3. Escribe un script que reciba una o más carpetas como argumentos, las guarde en un array y muestre cuántos archivos hay en cada una. Debe funcionar con carpetas cuyo nombre tenga espacios.
4. Añade a ese script un archivo temporal creado con `mktemp` y un `trap` que lo borre al salir. Comprueba que el temporal desaparece incluso si interrumpes el script con Ctrl+C.

Soluciones

1. El fallo más habitual es `set -u` protestando por `$1` cuando el script se ejecuta sin argumentos. Se arregla dándole un valor por defecto:

```
archivo="${1:-}"        # ahora está definida, aunque esté vacía
```

2. Para el nombre, `%.*` quita desde el último punto hacia la derecha. Para la extensión, `##*.` quita todo hasta el último punto, que es justo lo contrario. Con `copia.tar.gz` dan `copia.tar` y `gz`:

```
#!/bin/bash
set -euo pipefail

[ "$#" -eq 1 ] || { echo "uso: $(basename "$0") ARCHIVO" >&2; exit 1; }

archivo="$1"
echo "nombre:    ${archivo%.*}"
echo "extensión: ${archivo##*.}"
```

3. Los argumentos se meten en un array con `("$@")`, con comillas:

```bash
#!/bin/bash
set -euo pipefail

[ "$#" -ge 1 ] || { echo "uso: $(basename "$0") CARPETA..." >&2; exit 1; }

carpetas=("$@")

for c in "${carpetas[@]}"; do
    if [ -d "$c" ]; then
        cuantos=$(find "$c" -maxdepth 1 -type f | wc -l)
        echo "$c: $cuantos archivos"
    else
        echo "$c: no es una carpeta" >&2
    fi
done
```

4. Basta con añadir dos líneas al principio. El `trap` se registra **justo después** de crear el temporal, no antes (si no, aún no existe la variable) ni mucho después (si no, hay un hueco en el que puede morir sin limpiar):

```bash
#!/bin/bash
set -euo pipefail

tmp="$(mktemp)"
trap 'rm -f "$tmp"' EXIT

for c in "$@"; do
    find "$c" -maxdepth 1 -type f >> "$tmp"
done

echo "Total de archivos: $(wc -l < "$tmp")"
```

BASH EN EL MUNDO REAL

Hasta ahora todo ha pasado dentro de tu máquina. Este capítulo es el que saca tus scripts a la red: entrar en un servidor, copiar archivos a él y hablar con servicios web.

Son las cuatro herramientas que aparecen en cualquier trabajo real con servidores: `ssh` para entrar, `rsync` para copiar, `curl` para pedir cosas por HTTP y `jq` para entender lo que te contestan.

Entrar en otra máquina: ssh

SSH (*Secure Shell*) abre una terminal en **otro ordenador**. Escribes en el tuyo, pero los comandos se ejecutan allí. Todo lo que va y viene está cifrado.

```bash
ssh nica@203.0.113.45
```

Es `usuario@máquina`. La máquina puede ser una IP o un nombre de dominio. Si el servidor no usa el puerto estándar (el 22), se indica con `-p`:

```
ssh -p 2222 nica@203.0.113.45
```

Para salir, `exit` o Ctrl+D, como en cualquier terminal.

La primera vez que te conectas a un servidor, SSH te avisa de que no lo conoce y te enseña su huella digital. Si dices que sí, la guarda en `~/.ssh/known_hosts` y no vuelve a preguntar. Si algún día te avisa de que la huella **ha cambiado** en un servidor que ya conocías, para y averigua por qué: suele ser que reinstalaron el servidor, pero es exactamente la misma señal que daría alguien suplantándolo.

Ejecutar un comando sin quedarte dentro

Esto es lo que más se usa en scripts. Si pones un comando entre comillas después del destino, SSH lo ejecuta allí, te devuelve la salida y cierra la conexión:

```
ssh nica@203.0.113.45 "df -h /"
ssh nica@203.0.113.45 "systemctl status nginx"
```

Y como la salida vuelve a tu terminal, se puede meter en un pipe o guardar en una variable, igual que cualquier otro comando:

```
libre=$(ssh nica@203.0.113.45 "df -h / | awk 'NR==2 {print \$5}'")
echo "El disco del servidor está al $libre"
```

> **Consejo**
>
> Fíjate en el `\$5` con barra invertida. Dentro de comillas dobles, tu Bash local sustituiría `$5` antes de enviar nada (capítulo 9). La barra invertida le dice «este dólar es para el otro lado». Si el comando remoto es complicado, suele ser más limpio usar comillas simples por fuera.

Claves en vez de contraseña

Escribir la contraseña cada vez es incómodo, y en un script directamente no sirve: un script no puede teclearla. La solución son las **claves SSH**.

Una clave SSH son dos archivos que van juntos:

- La **clave privada**, que se queda en tu ordenador y **no sale de ahí jamás**.
- La **clave pública**, que copias al servidor.

La analogía habitual: la pública es un candado abierto y la privada es la única llave que lo abre. Puedes repartir candados por todos los servidores que quieras; mientras solo tú tengas la llave, solo tú entras.

Se generan con `ssh-keygen`:

```
ssh-keygen -t ed25519 -C "nica@portatil"
```

`-t ed25519` elige el tipo de clave (el recomendado hoy) y `-C` añade un comentario para que luego sepas de qué máquina era. Te preguntará dónde guardarla (acepta lo que propone) y si quieres una contraseña para la clave.

El resultado son dos archivos:

```
-rw------- 1 nica nica 399 Jul 31 15:10 id_ed25519
-rw-r--r-- 1 nica nica  95 Jul 31 15:10 id_ed25519.pub
```

Mira los permisos, que son los del capítulo 6: la privada es `600` (solo tú puedes leerla) y la pública `644` (la puede leer cualquiera). SSH **se niega a usar** una clave privada si tiene permisos más abiertos, así que si algún día te da un error raro de permisos, ya sabes.

La pública es una sola línea de texto:

```
ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIDqvdWwQ8GwSgqt/jXlhpLWy7k24Q55goNr0JX1ak80y
nica@portatil
```

Y se instala en el servidor con un comando que lo hace todo:

```
ssh-copy-id nica@203.0.113.45
```

Te pedirá la contraseña **esa última vez**, y a partir de ahí entras sin ella. Lo que hace por dentro es añadir tu clave pública al archivo `~/.ssh/authorized_keys` del servidor.

> **Precaución**
>
> La clave **privada** (el archivo sin `.pub`) no se comparte nunca: ni por correo, ni en un repositorio de Git, ni copiándola a otro servidor. Quien la tenga entra donde tú entras. Si alguna vez se te escapa, borra la pública correspondiente del `authorized_keys` de todos los servidores y genera otra.
>
> Lo que se copia siempre es el archivo `.pub`.

El archivo ~/.ssh/config

Recordar puertos, IPs y usuarios es insufrible. El archivo `~/.ssh/config` te deja ponerle un mote a cada servidor:

```
Host vps
    HostName 203.0.113.45
    User nica
    Port 2222
    IdentityFile ~/.ssh/id_ed25519

Host *
    ServerAliveInterval 60
```

Con eso, esto:

```
ssh -p 2222 nica@203.0.113.45
```

se convierte en esto:

```
ssh vps
```

Y el mote funciona en todo lo demás: `scp archivo vps:/ruta`, `rsync ... vps:`, `ssh vps "comando"`. El bloque `Host *` aplica a todos los servidores; ese `ServerAliveInterval 60` manda una señal cada minuto para que la conexión no se caiga sola cuando te distraes.

Puedes comprobar qué configuración le sale a un destino **sin conectarte**, con `ssh -G`:

```
ssh -G vps | grep -E '^(hostname|user|port|identityfile) '
```

Y sale:

```
user nica
hostname 203.0.113.45
port 2222
identityfile ~/.ssh/id_ed25519
```

> **Nota**
>
> El archivo `config` debe tener permisos `600`, igual que la clave privada: `chmod 600 ~/.ssh/config`. Si no, SSH lo ignora.

Copiar archivos: scp

`scp` (*secure copy*) copia archivos por SSH. La sintaxis es la de `cp`, pero con `usuario@máquina:ruta` en el lado remoto.

```
scp informe.txt vps:/home/nica/documentos/     # subir
scp vps:/var/log/nginx/error.log .             # bajar
scp -r web/ vps:/var/www/                      # una carpeta entera
```

Igual que `cp`, necesita `-r` para carpetas. Y el puerto se indica con `-P` **mayúscula** (en `ssh` era `-p` minúscula, que es una de esas incoherencias que hay que sufrir; con el `~/.ssh/config` te olvidas del tema).

> **Precaución**
>
> Los dos puntos son el error número uno de `scp`. Si escribes `scp informe.txt vps` **sin** los dos puntos, `scp` no entiende «al servidor vps»: entiende «copia el archivo aquí mismo con el nombre `vps`». No da error, y te quedas esperando un archivo que nunca llegó.
>
> Regla: si no hay `:` en algún lado, no estás copiando a ninguna parte.

Si los dos puntos van sin ruta detrás (`vps:`), se copia a la carpeta personal del usuario, que suele ser lo que quieres.

Copiar bien: rsync

`scp` copia todo, siempre. `rsync` compara origen y destino y **copia solo lo que ha cambiado**. Para subir una web de cien archivos de los que has tocado dos, la diferencia es abismal.

```
rsync -avz web/ vps:/var/www/detomas/
```

Las opciones que se usan siempre:

Opción	Qué hace
`-a`	Modo archivo: recursivo y conservando permisos y fechas
`-v`	Cuenta lo que va haciendo
`-z`	Comprime durante el envío (útil por red, inútil en local)
`-n` o `--dry-run`	**Ensayo**: dice lo que haría, sin hacer nada
`--delete`	Borra en el destino lo que ya no está en el origen
`--exclude`	Se salta lo que le digas (`--exclude '.git'`)
`-P`	Muestra progreso y permite reanudar archivos grandes

La barra final, que confunde a todo el mundo

Esta es **la** particularidad de `rsync`. Una barra al final del **origen** cambia el resultado:

```
rsync -av web/ destino/    # copia el CONTENIDO de web dentro de destino
rsync -av web  destino/    # copia la CARPETA web dentro de destino
```

Con barra queda `destino/index.html`. Sin barra queda `destino/web/index.html`.

La forma de acordarse: la barra significa «lo de dentro de». Y la forma de no equivocarse nunca es la de la sección siguiente.

Ensayar antes: –dry-run

`--dry-run` (o `-n`) hace que `rsync` te cuente exactamente lo que haría, sin tocar nada:

```
rsync -av --dry-run web/ vps:/var/www/detomas/
```

Y sale:

```
sending incremental file list
./
nuevo.txt

sent 178 bytes  received 23 bytes  402.00 bytes/sec
total size is 18  speedup is 0.09 (DRY RUN)
```

Ese `(DRY RUN)` del final es la confirmación de que no se ha hecho nada.

> **Precaución**
>
> `--delete` borra en el destino todo lo que no esté en el origen. Es lo que quieres para dejar una web idéntica a tu copia local, y es lo que vacía un servidor entero si te equivocas de carpeta de origen o de barra final.
>
> **Con `--delete`, el `--dry-run` no es opcional.** Ejecútalo siempre primero y lee la lista de lo que va a borrar.

```
rsync -av --delete --dry-run web/ vps:/var/www/detomas/
```

Y sale:

```
sending incremental file list
deleting index.html
./
nuevo.txt
```

Ahí se ve claramente qué va a desaparecer antes de que desaparezca.

Hablar con una web: curl

`curl` hace peticiones HTTP desde la terminal. En su forma más simple, descarga una página y la escupe por pantalla:

```
curl https://detomas.net
```

Las opciones que de verdad se usan:

Opción	Qué hace
`-s`	Callado: sin barra de progreso ni estadísticas
`-o archivo`	Guarda la respuesta en un archivo en vez de imprimirla
`-L`	Sigue las redirecciones
`-I`	Solo las cabeceras, sin el contenido
`-w`	Imprime datos de la petición con un formato que tú eliges
`--max-time`	Se rinde pasados N segundos
`-H`	Añade una cabecera
`-X`	Cambia el método (`POST`, `PUT`, `DELETE`…)
`-d`	Envía datos en el cuerpo de la petición

Ver solo las cabeceras

```
curl -sI https://detomas.net | head -4
```

Y sale:

```
HTTP/2 200
x-powered-by: PHP/8.1.34
content-type: text/html; charset=UTF-8
date: Fri, 31 Jul 2026 13:11:12 GMT
```

Seguir redirecciones

Muchas URLs no responden directamente, sino que te mandan a otra. Sin `-L`, `curl` se queda en la redirección:

```
curl -s -o /dev/null -w '%{http_code} -> %{redirect_url}\n'
https://detomas.net/cursos
```

Y sale:

```
301 -> https://detomas.net/cursos/
```

Con `-L` la sigue hasta el final:

```
curl -sL -o /dev/null -w '%{http_code} final: %{url_effective}\n'
https://detomas.net/cursos
```

Y sale:

```
200 final: https://detomas.net/cursos/
```

Comprobar si una web está viva

Esta combinación es la que vas a usar en scripts. `-o /dev/null` tira el contenido (no nos interesa) y `-w '%{http_code}'` imprime solo el código de estado:

```
curl -s -o /dev/null -w '%{http_code}\n' https://detomas.net
```

Y sale:

```
200
```

Los códigos que hay que reconocer: **200** todo bien, **301** y **302** redirección, **404** no existe, **500** el servidor ha reventado, **000** ni siquiera se pudo conectar (esto último es de `curl`, no de HTTP).

Metido en una variable, ya es una comprobación:

```
codigo=$(curl -sL -o /dev/null -w '%{http_code}' --max-time 10
https://detomas.net)
if [ "$codigo" != "200" ]; then
    echo "La web responde $codigo" >&2
fi
```

Enviar datos: POST y webhooks

Hasta aquí solo hemos **pedido** cosas. Para enviarlas se usa `POST`:

```
curl -X POST https://ejemplo.com/webhook \
     -H "Content-Type: application/json" \
     -d '{"evento":"web-caida","url":"https://detomas.net","codigo":500}'
```

Las tres piezas:

- `-X POST` es el método: «te traigo datos», en vez de «dame algo».
- `-H "Content-Type: application/json"` avisa de que lo que mandas es JSON. Sin esa cabecera, muchos servicios no saben interpretarlo.
- `-d` es el contenido. Va entre **comillas simples** para que Bash no toque las comillas dobles de dentro del JSON.

Si el JSON es largo, se puede leer de un archivo con `@`:

```
curl -X POST https://ejemplo.com/webhook \
     -H "Content-Type: application/json" \
     -d @aviso.json
```

Esto es exactamente lo que necesita un **webhook**: una URL que se queda esperando a que alguien le mande datos para arrancar un proceso. Es la forma de que un script de Bash dispare un flujo de n8n.

> **Precaución**
>
> Los webhooks suelen llevar un token o una clave en la URL. **No lo escribas dentro del script**, sobre todo si el script va a un repositorio de Git. Guárdalo en una variable de entorno y exígela con lo del capítulo 13:

```
WEBHOOK="${WEBHOOK_URL:?define WEBHOOK_URL antes de ejecutar}"
```

> Y ejecuta el script así: `WEBHOOK_URL="https://..." ./mi-script.sh`

Leer JSON sin volverse loco: jq

Casi todo lo que devuelve una API es JSON. Y el JSON **no se procesa con** `grep`: no tiene una estructura de líneas, los espacios dan igual, un valor puede ocupar varias líneas o estar todo en una. Lo que `grep` te devuelve es texto que luego tienes que volver a trocear a mano.

`jq` entiende la estructura. Se instala con `sudo apt install jq`.

Vamos a usar este archivo, `servidores.json`:

```
cat > servidores.json << 'FIN'
{
  "empresa": "detomas.net",
  "servidores": [
    { "nombre": "web",    "ip": "203.0.113.10", "activo": true,  "carga": 0.42 },
    { "nombre": "n8n",    "ip": "203.0.113.11", "activo": true,  "carga": 0.87 },
    { "nombre": "backup", "ip": "203.0.113.12", "activo": false, "carga": 0.05 }
  ]
}
FIN
```

Lo básico

Un punto solo significa «todo», y sirve para **formatear** JSON apelmazado:

```
jq . servidores.json
```

Un nombre detrás del punto saca ese campo:

```
jq '.empresa' servidores.json      # "detomas.net"  (con comillas)
jq -r '.empresa' servidores.json   # detomas.net    (crudo, sin comillas)
```

`-r` (*raw*) es imprescindible cuando el valor va a acabar en una variable de Bash: sin él te llevas las comillas dentro.

Recorrer listas

`[]` recorre todos los elementos de una lista:

```
jq -r '.servidores[].nombre' servidores.json
```

Y sale:

```
web
n8n
backup
```

Y `|` dentro de `jq` encadena operaciones, igual que el pipe de Bash pero dentro del propio `jq`:

```
jq -r '.servidores[] | .nombre + " " + .ip' servidores.json
```

Y sale:

```
web 203.0.113.10
n8n 203.0.113.11
backup 203.0.113.12
```

Filtrar

`select()` se queda solo con lo que cumpla la condición:

```
jq -r '.servidores[] | select(.activo == true) | .nombre' servidores.json
```

Y sale:

```
web
n8n
```

Y con `\(...)` puedes construir el texto de salida como quieras:

```
jq -r '.servidores[] | select(.carga > 0.5) | "\(.nombre): carga \(.carga)"'
servidores.json
```

Y sale:

```
n8n: carga 0.87
```

Filtro	Qué hace
`.`	Todo, formateado
`.campo`	Un campo
`.[]`	Recorre una lista
`.[0]`	El primer elemento
`length`	Cuántos hay
`select(cond)`	Filtra
`keys`	Los nombres de los campos
`-r`	Sin comillas

El combo: curl y jq

Aquí es donde todo encaja. Una petición, y del resultado sacas solo el dato que te interesa:

```
primero=$(jq -r '.servidores[0].nombre' servidores.json)
echo "El primero se llama: $primero"
```

Y sale:

```
El primero se llama: web
```

Con `curl` es lo mismo, cambiando el archivo por un pipe:

```
curl -s https://una-api.com/estado | jq -r '.estado'
```

El propio `curl` sabe emitir JSON con sus estadísticas, así que puedes probar el combo ahora mismo contra cualquier web:

```
curl -s -o /dev/null -w '%{json}' https://detomas.net \
  | jq '{codigo: .http_code, segundos: .time_total, tam: .size_download}'
```

Y sale:

```
{
  "codigo": 200,
  "segundos": 0.089304,
  "tam": 40180
}
```

Un script de verdad

Juntamos todo el capítulo y el anterior: un script que comprueba si tu web responde y, si no, avisa por webhook.

```
#!/bin/bash
set -euo pipefail

URL="${1:-https://detomas.net}"
WEBHOOK="${WEBHOOK_URL:?define WEBHOOK_URL antes de ejecutar}"

codigo=$(curl -sL -o /dev/null -w '%{http_code}' --max-time 10 "$URL" || true)

if [ "$codigo" = "200" ]; then
    echo "OK: $URL responde $codigo"
    exit 0
fi

echo "AVISO: $URL responde $codigo" >&2

respuesta=$(curl -s -X POST "$WEBHOOK" \
    -H "Content-Type: application/json" \
    -d "{\"evento\":\"web-caida\",\"url\":\"$URL\",\"codigo\":\"$codigo\"}")

echo "Aviso enviado. Respuesta: $(echo "$respuesta" | jq -r '.recibido')"
exit 1
```

Ejecutándolo:

```
--- la web responde bien ---
OK: https://detomas.net responde 200
código de salida: 0

--- la web no responde ---
AVISO: https://detomas.net/no-existe-999 responde 404
Aviso enviado. Respuesta: true
código de salida: 1

--- el dominio ni siquiera existe ---
AVISO: https://noexisteestedominio-xyz123.net responde 000
Aviso enviado. Respuesta: true
código de salida: 1

--- falta el webhook ---
vigilar.sh: line 5: WEBHOOK_URL: define WEBHOOK_URL antes de ejecutar
código de salida: 1
```

Repasa lo que hay ahí dentro, porque es casi el curso entero:

- `set -euo pipefail` y `${WEBHOOK_URL:?...}` del capítulo 13.

- `${1:-...}` para que el argumento sea opcional.
- El `|| true` es para el caso de que `curl` ni siquiera pueda conectar. Cuando eso pasa, `curl` imprime `000` **y además** devuelve un código de error, que con `set -e` habría matado el script antes de poder avisar. El `|| true` se come ese error y deja que el `000` llegue a la variable.
- `>&2` manda el aviso al canal de errores, del capítulo 4.
- `exit 1` para que cron sepa que hubo un problema.
- Y las comillas de `"$URL"` y `"$codigo"`, del capítulo 9.

Añádele una línea de crontab (capítulo 11) y tienes una vigilancia de tu web funcionando cada cinco minutos:

```
*/5 * * * * WEBHOOK_URL="https://n8n.detomas.net/webhook/xxx"
/home/nica/bin/vigilar.sh
```

> **Nota**
>
> El `\"` dentro del `-d` es necesario porque el JSON va entre comillas dobles para que se sustituyan `$URL` y `$codigo`. Cuando el JSON crece, esto se hace ilegible y es mejor construirlo con `jq -n`:

```
cuerpo=$(jq -n --arg url "$URL" --arg cod "$codigo" \
         '{evento: "web-caida", url: $url, codigo: $cod}')
```

> Además, así `jq` se encarga de escapar los caracteres raros.

Ejercicios

1. Genera un par de claves SSH nuevo con `ssh-keygen` guardándolo en `~/pruebas/clave_test`, y mira los permisos de los dos archivos que crea.

2. Escribe una entrada en `~/.ssh/config` para un servidor inventado y comprueba con `ssh -G` que la configuración sale como esperas. (Sin conectarte a ninguna parte.)

3. Copia una carpeta a otro sitio con `rsync`, primero con `--dry-run`. Después añade un archivo al origen, bórrale otro, y vuelve a ejecutarlo con `--delete --dry-run` para ver qué haría.

4. Escribe un script que reciba una lista de URLs como argumentos y muestre el código de estado de cada una, marcando con un aviso las que no devuelvan

Soluciones

1. El `-N ''` pone la clave sin contraseña, para no tener que teclear nada:

```
mkdir -p ~/pruebas
ssh-keygen -t ed25519 -C "prueba" -f ~/pruebas/clave_test -N ''
ls -l ~/pruebas/
```

La privada sale como `-rw-------` (600) y la pública como `-rw-r--r--` (644).

2. Escribe el bloque en `~/.ssh/config` y consulta:

```
ssh -G miservidor | grep -E '^(hostname|user|port) '
```

`ssh -G` resuelve la configuración y la imprime **sin abrir ninguna conexión**, así que es seguro probar con servidores que no existen.

3. El ciclo completo:

```
rsync -av --dry-run origen/ destino/    # ver qué haría
rsync -av origen/ destino/              # hacerlo
touch origen/nuevo.txt && rm origen/viejo.txt
rsync -av --delete --dry-run origen/ destino/   # ver qué borraría
```

En la última salida aparece una línea `deleting viejo.txt`. **Esa** es la línea que hay que leer siempre antes de quitar el `--dry-run`.

4. Un bucle sobre los argumentos, con la comprobación de `curl`:

```
#!/bin/bash
set -euo pipefail

[ "$#" -ge 1 ] || { echo "uso: $(basename "$0") URL..." >&2; exit 1; }

fallos=0

for url in "$@"; do
    codigo=$(curl -sL -o /dev/null -w '%{http_code}' --max-time 10 "$url" || true)
    if [ "$codigo" = "200" ]; then
        echo "OK    $codigo  $url"
    else
        echo "AVISO $codigo  $url"
        fallos=$((fallos + 1))
    fi
done

echo "---"
echo "Revisadas $# URLs, $fallos con problemas"
[ "$fallos" -eq 0 ]
```

La última línea es un detalle útil: `[ "$fallos" -eq 0 ]` es la última orden del script, así que su resultado es el código de salida. Si hubo fallos, devuelve distinto de cero y cron se entera.