Guía rápida de Bash/Shell

Terminal y línea de comandos con más de 200 comandos y ejercicios

mouredev pro

mouredev.pro/recursos

Índice

INTRODUCCIÓN

Como desarrolladores, trabajamos en un entorno radicalmente transformado por la Inteligencia Artificial. Los asistentes de IA generan bloques de código complejos, optimizan algoritmos e incluso proponen arquitecturas completas a partir de lenguaje natural. Con herramientas tan potentes a nuestra disposición, es tentador pensar en desplazar tecnologías fundamentales como la línea de comandos o la shell de Bash a las categorías de "legacy".

Sin embargo, la realidad de la infraestructura moderna nos cuenta otra historia. Paradójicamente, el auge de la IA hace que el dominio de Bash sea más importante que nunca.

La razón reside en una distinción fundamental: La IA sobresale en la creación de código, pero Bash sigue siendo el rey indiscutible en la ejecución, orquestación y control de ese código en el mundo real.

Vivimos en la era de la generación de código, pero el código no tiene valor hasta que se despliega y opera de manera fiable. La IA es muy buena definiendo el "qué" (la lógica de la aplicación), pero la infraestructura exige un control absoluto sobre el "dónde" y el "cómo" (el entorno de ejecución).

En la actualidad, piensa en la IA como un copiloto, pero tú sigues siendo el piloto, y Bash es tu panel de control. Dominar la shell no es mirar al pasado, es asegurar tu control sobre la infraestructura que potencia el futuro. Es el superpoder que convierte el potencial estático generado por la IA en sistemas vivos, resilientes y eficientes en producción.

Aquí es donde Bash brilla como la lengua nativa del desarrollo moderno.

CONFIGURACIÓN

Shell, Terminal y Bash

- **Shell**: Programa que interpreta órdenes (comandos) que el usuario escribe.
    - Ejemplos: bash, sh, zsh, csh, powershell...
- **Terminal**: Programa que proporciona una interfaz gráfica o de texto para comunicarse con la shell.
    - Warp, Xterm, terminal del sistema por defecto...
- **Bash** (Bourne-again shell): La shell más popular en entornos Unix (Linux y macOS)

Historia de Bash

Wikipedia
https://es.wikipedia.org/wiki/Bash

¿Por qué Bash?

- Es el estándar Unix/Linux/macOS (aunque en las últimas versiones de macOS es zsh, una evolución de Bash).
- La gran mayoría de scripts y sistemas de administración de servidores están escritos en Bash.
- Gran portabilidad.
- Más documentación y comunidad.

Bash en Windows

Git for Windows (no soporta todos los comandos de Bash)

Instalación
https://gitforwindows.org

WSL - Windows Subsystem for Linux (soporte completo a Bash)

Debes instalarlo como administrador si tienes Windows para seguir el curso.

Instalación
https://learn.microsoft.com/es-es/windows/wsl/install

```shell
wsl --install
```

- Reinicia
- Finaliza la configuración de Linux

Ya podrás usar Bash en la terminal.

> **Nota:**
>
> El directorio raíz de Windows se encuentra en `/mnt/c`

Warp

La terminal y entorno de desarrollo con IA que utilizaremos durante el curso, ya que nos ayudará a interactuar con la Shell de una manera más cómoda y avanzada. Recomiendo su instalación (aunque puedas usar otra terminal).

Importante:

Descarga Warp 2
https://mouredev.link/warp

PRIMEROS PASOS

Hola mundo

Primer comando en Bash: `echo` imprime texto en pantalla.

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

Cómo mostrar la Shell que estamos utilizando.

```shell
echo $SHELL
echo $0
```

Comandos de Orientación

- `pwd` Imprime el directorio actual.
- `ls` Lista archivos en el directorio actual.
    - Con opciones:
    - `ls -l`
      Muestra los archivos en formato largo.
    - `ls -a`
      Muestra todos los archivos, incluidos los ocultos.
    - `ls -lh`
      Como `-l` pero el tamaño de los archivos aparece en formato *"human-readable"*.

Comandos de Navegación

Para cambiar de directorio utilizamos el comando `cd`.

`dir` hace referencia al directorio (carpeta) actual a la que nos queremos desplazar.

- `cd dir`
  Te desplaza al directorio inferior indicado (significa *"ir a un nivel inferior concreto"*).
- `cd dir/dir/dir`
  Para desplazarse varios directorios inferiores a la vez.
- `cd ..`
  Te desplaza al directorio superior al actual (significa *"un nivel arriba"*).
- `cd ../../../../`
  Para desplazarse varios niveles superiores a la vez.
- `cd .`
  Hace referencia al directorio actual.
- `cd ~`
  Te desplaza directamente a tu directorio personal (home).

> **Importante:**
>
> Utiliza la **tabulación (TAB)** como ayuda para completar automáticamente comandos o visualizar archivos o directorios sugeridos.

```shell
cd TAB
```

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 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 donde te encuentres.

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

Otros comandos básicos

- `whoami`
  Muestra el usuario actual.
- `cal`
  Muestra calendario.
- `date`
  Muestra la fecha y hora actuales.
- `uptime`
  Tiempo encendido del sistema.
- `hostname`
  Nombre del equipo.
- `uname`
  Información del kernel.
- `uname -a`
  Información del kernel/sistema.
- `clear`
  Limpia la pantalla (no borra).

Anatomía del comando

```Shell
comando [opciones] [argumentos]
```

- `comando` es lo que quieres ejecutar (`ls`).
- `opciones` modifican el comportamiento (`-l`)
- `argumentos` son los datos sobre los que actúa (`archivo.txt`, `directorio/`)

Ayuda y documentación

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

```Shell
python --help
python -h
```

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

```Shell
man ls
```

> **Consejo:**
>
> Para salir del manual pulsamos la tecla **"q"**.

Ejercicios primeros pasos

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, hora actuales y el calendario de este mes.

9. Regresa al directorio donde comenzaste en el primer ejercicio.

10. Limpia la pantalla.

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.

```shell
ls /
```

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 (sólo 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 sólo 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.

Cuidado:

⚠️ El comando `rm` No se envía a la papelera. Cuidado con lo que se borra.

✋ La opción `f` (force) en `rm -rf` es muy peligrosa ya que no pide confirmación ni muestra errores si el directorio no existe.

Wildcard (comodines)

Los comodines permiten trabajar con varios archivos de forma rápida. Se pueden combinar entre ellos. Se pueden combinar con diferentes comandos.

- `*`
  Cero o más caracteres.
- `?`
  Exactamente un caracter.

Ejemplos:

- `ls *.md`
  Muestra todos los archivos con la extensión *md*.
- `ls *.txt`
  Muestra todos los archivos con la extensión *txt*.
- `ls 03*`
  Muestra todos los archivos que comienzan por *03*.
- `ls 03*.txt`
  Muestra todos los archivos que comienzan por *03* y tienen la extensión *txt*.
- `ls ?????*`
  Muestra todos los archivos que tienen 5 o más caracteres.
- `rm ?.txt`
  Elimina todos los archivos con un nombre de un único caracter y la extensión *txt*.
- `rm a????`
  Elimina todos los archivos que comiencen por *a* y tengan 5 caracteres.

Consejo:

Puedes realizar combinaciones de todo tipo con comandos y comodines.

Listados avanzados

- `tree`
  Muestra un árbol de directorios y archivos.
  - `tree -a`
    Muestra también los directorios y archivos ocultos.
- `find . -name "nombre"`
  Encuentra archivos por nombre en el directorio actual.
  - `find dir -name "*.log"`
    Encuentra archivos por criterio de búsqueda (todos los *log*, por ejemplo) en el directorio especificado.

Nota:

El comando `tree` no está instalado por defecto. Ten en cuenta cómo hacerlo según tu sistema operativo y el gestor de paquetes empleado (por ejemplo *apt* o *homebrew*, entre otros).

Ejercicios gestión de archivos

1. Crea un directorio.
2. Elimina el directorio que acabas de crear.
3. Copia un archivo en el directorio actual y fuera de éste.
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`
  Muestra todo el contenido del archivo en pantalla (útil para archivos pequeños).
- `less`
  Permite ver archivos largos, paginando su contenido (sales pulsando `q`).
- `more`
  Similar a less, pero como menos funcionalidades (por ejemplo, no puedes desplazarte hacia atrás).
- `head`
  Muestra las primeras 10 líneas de un archivo por defecto.
  - `head -n 20`
    Para especificar el número de líneas.
- `tail`
  Muestra las últimas 10 líneas de un archivo por defecto.
  - `tail -n 20`
    Para especificar el número de líneas.
  - `tail -f file.log`
    Muy útil para ver logs en tiempo real mientras crecen.

Consejo:

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

Búsqueda y recuento

Búsqueda: `grep` busca patrones dentro de archivos o de la salida de otros comandos.

- `grep "texto a buscar" nombre_archivo`
  Busca un texto en un archivo (retorna las filas en las que aparece el texto).
  - `grep -i "texto a buscar" nombre_archivo`
    Busca el texto ignorando mayúsculas/minúsculas.
  - `grep -r "texto a buscar" dir`
    Busca el texto recursivamente en directorios

Recuento: `wc` cuenta líneas, palabras y bytes/caracteres.

- `wc nombre_archivo`
  Muestra el recuento de líneas, palabras y bytes/caracteres.
  - `wc -l nombre_archivo`
    Muestra el recuento de líneas.
  - `wc -w nombre_archivo`
    Muestra el recuento de palabras.
  - `wc -c nombre_archivo`
    Muestra el recuento de bytes/caracteres.

Como siempre, las opciones se pueden combinar: `wc -lw nombre_archivo`.

Redirecciones y pipes

Redirecciones

- `>`
  Redirige la salida y sobrescribe el archivo.
- `>>`
  Redirige la salida y añade al final del archivo.
- `<`
  Toma la entrada desde un archivo.

Ejemplos:

- `echo "texto" > archivo.txt`
  Sobrescribe el contenido del archivo con la salida del comando (en este caso, imprimir un texto).
- `echo "texto" >> archivo.txt`
  Añade al contenido del archivo la salida del comando (en este caso, imprimir un texto).
- `sort < archivo.txt`
  Usa el archivo como entrada y ejecuta el comando con su contenido (en este caso, ordenar).

Pipes

- `|`
  Encadena comandos.

Ejemplo:

- `cat nombre_archivo | grep "texto a buscar" | wc -w`
  Muestra el contenido de un archivo, busca un texto en ese contenido y realiza el recuento de palabras resultantes de la búsqueda.

Variables de entorno

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

Variable local

Las variables locales sólo viven en la sesión actual.

- `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 global

Las variables globales viven más allá de la sesión (en todos los programas lanzados desde esa terminal de la sesión).

Algunas variables globales ya existentes:

- `echo $HOME` Muestra la ruta del directorio home del usuario.
- `echo $PATH` Muestra una lista de rutas separadas conocidas por el sistema por defecto.

Creación de una variable global:

- `export NOMBRE_VARIABLE="valor asociado a la variable"`

Creación de una variable global permanente:
Para ello debes agregar la línea de la exportación a tu archivo de configuración de la shell. Los archivos de configuración más habituales creados en tu directorio de usuario son:

- `~/.bash_profile`
- `~/.bash_login`
- `~/.profile`
- `~/.bashrc`

Ejercicios comandos avanzados

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.
10. Crea una variable local y muéstrala.

EDITORES BÁSICOS

Nota:

Para modificar archivos directamente desde la terminal.

nano

- `nano nombre_archivo` abre el archivo (si no existe, lo crea).
- Editor fácil para principiantes.
- Los comandos aparecen listados en la parte inferior del editor: `^ = Ctrl/Cmd (en macOS)`.
- Puedes comenzar a escribir directamente y desplazarte con las flechas.

Nota:

En macOS, `nano` en realidad abre `pico` por defecto, el editor original en el que está basado nano.

Puedes instalar `nano` con `brew install nano` (recuerda reiniciar la terminal).

Algunos atajos

- Guardar cambios: `Ctrl/Cmd + O`, luego `Enter`
- Salir: `Ctrl/Cmd + X` (si hay cambios sin guardar, preguntará)
- Ayuda: `Ctrl/Cmd + G`
- Buscar texto: `Ctrl/Cmd + W`
- Cortar línea: `Ctrl/Cmd + K`
- Pegar línea: `Ctrl/Cmd + U`

nano
https://www.nano-editor.org

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.

Modos

- Normal: Para desplazarte.
- Inserción: Permite escribir texto.
- Comando: Para ejecutar comandos especiales durante el modo normal.

Algunos atajos y comandos

- Activar modo *inserción*: `i` (si estás en modo *normal*).
- Activar modo *normal*: `Esc` (si estás en modo *inserción*).
- Atajos/Comandos básicos en modo *normal*:
  - `h j k l` o *flechas* para mover el cursor.
  - `:q` para salir.
  - `:wq` para guardar y salir.
  - `:q!` para salir sin guardar.
  - `/texto` para buscar.
  - `dd` para borrar la línea.
  - `yy` para copiar la línea.
  - `p` para pegar.
  - `u` para deshacer.

vim
https://www.vim.org

> **Consejo:**
>
> vim exige tiempo para aprender cómo 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**.

https://neovim.io
https://www.gnu.org/software/emacs

Ejercicios editores básicos

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

Nota:

En Unix/Linux, cada archivo o directorio tiene permisos para controlar quién puede **leer**, **escribir** o **ejecutarlo**.

Tipos de permiso

Modo simbólico:

- `r` lectura
- `w` escritura
- `x` ejecución

Modo octal:

- `4` (`r`) lectura
- `2` (`w`) escritura
- `1` (`x`) ejecución

Tipos de usuario

- `u` usuario propietario
- `g` grupo de usuarios
- `o` otros
- `a` todos

Ver permisos

- En archivos: `ls -l nombre_archivo`
- En carpetas: `ls -ld nombre_directorio`

Anatomía de los permisos

Modo simbólico:

`-rwxrwxrwx`

Está formado por cuatro bloques: `[-][rwx][rwx][rwx]`

De izquierda a derecha:

- `-`
  Tipo de archivo
- `rwx`
  Permisos de usuario
- `rwx`
  Permisos de grupo
- `rwx`
  Permisos para otros usuarios

`rwx` está indicando que ese bloque tiene permisos de lectura (`r`), escritura (`w`) y ejecución (`x`).

> **Cuidado:**
>
> La ausencia de un permiso (permiso no otorgado) se representa con `-`.
>
> Ejemplos: `r--` indicaría que sólo tiene permisos de lectura, en cambio, `r-x` indicaría 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:
  - `7` = 4 + 2 + 1 (lectura + escritura + ejecución) [`rwx`]
  - `6` = 4 + 2 (lectura + escritura) [`rw-`]
  - `5` = 4 + 1 (lectura + ejecución) [`r-x`]
  - `4` (lectura) [`r--`]
  - `3` = 2 + 1 (escritura + ejecución) [`-wx`]
  - `2` (escritura) [`-w-`]
  - `1` (ejecución) [`--x`]

Por lo tanto, `777` quiere decir que todos los usuarios tienen permisos de lectura, escritura y ejecución.

`764` significaría que el *propietario* tiene todos los permisos (4 + 2 + 1 = 7), el *grupo* tendría de lectura y escritura (4 + 2 = 6), y *otros* únicamente de lectura (4).

Tipos de archivos más habituales

- `-` archivo
- `d` directorio
- `l` enlace simbólico
- `b` dispositivo de bloque
- `c` dispositivo de carácter
- `s` socket
- `p` pipe

Modificación de permisos

- Se utiliza el comando `chmod`.

Modo simbólico:

- `chmod [tipo_usuario][+/-][permiso] nombre_archivo/directorio`
- `+` para otorgar un permiso.
- `-` para eliminar un permiso.
- Ejemplos:
    - `chmod u+x nombre_archivo/directorio`
      Otorga permisos de ejecución al usuario propietario.
    - `chmod u-x nombre_archivo/directorio`
      Elimina permisos de ejecución para el usuario propietario.

Modo octal:

- `chmod [permisos_octal] nombre_archivo/directorio`
- Ejemplos:
    - `chmod 753 nombre_archivo/directorio`: u=`rwx`, g=`r-x`, o=`-wx`
    - `chmod 642 nombre_archivo/directorio`: u=`rw-`, g=`r--`, o=`-w-`

Cambiar propietario y grupo

- `chown [usuario] nombre_archivo/directorio` para cambiar el propietario.
- `chown [usuario]:[grupo] nombre_archivo/directorio` para cambiar el propietario y el grupo.

Máscara de permisos

La máscara hace referencia a los permisos por defecto que se le otorgarán a nuevos directorios o archivos.

- `umask`
  Muestra la máscara de permisos por defecto.
- `umask [permisos_octal]`
  Establece los permisos por defecto.
    - Ejemplo: `umask 022`

Cuidado:

La modificación de la máscara es temporal, y sólo afecta a la sesión de la shell actual. Para hacerlo permanente, puedes agregarlo al script de configuración de Bash (como en el caso de variables globales explicado en una lección anterior).

Cálculo de permisos con máscara

La máscara hace referencia a los valores que deben restarse a los permisos máximos razonables de un directorio o archivo.

- Permisos máximos razonables para un directorio: `777`
- Permisos máximos razonables para un archivo: `666` (Sin ejecución, ya que dar esos permisos por defecto a todos los archivos sería peligroso)

Si la máscara de permisos es `0022`, quiere decir lo siguiente:

- Si la máscara tiene 4 dígitos, el primer cero de la izquierda únicamente indica que la representación de los 3 siguientes está en sistema octal.
- 0 al usuario no le quito ningún permiso
- 2 a los grupos le quito el permiso de escritura (2)
- 2 a otros le quito el permiso de escritura (2)

Cálculo de permisos:

- Para directorios: `777` - `022` (bloque a bloque) = `755` [`rwxr-xr-x`]
- Para archivos: `666` - `022` (bloque a bloque) = `644` [`rw-r--r--`]

Esos son los permisos por defecto que se le otorgarían a nuevas carpetas y archivos.

El superusuario

Cuidado:

Algunas acciones requieren privilegios de administrador (**root**).

`sudo [comando]` (solicita la contraseña del superusuario).

Ejemplo: `sudo rm -rf [directorio_protegido]`

Ejercicios administración del sistema

1. Crea un archivo y visualiza sus permisos.
2. Otorga permisos de ejecución sólo al propietario en modo simbólico.
3. Cambia sus permisos a 644.
4. Elimina los permisos para el grupo.
5. Haz que sólo pueda ejecutarse por el propietario.
6. Crea una carpeta y dale permisos para que sólo 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.

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 (sólo 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** hace referencia al identificador único del proceso para acceder a su manipulación.

Para salir del monitor utiliza `Ctrl/Cmd + C`.

Trabajos (jobs)

- `jobs` muestra los procesos actuales de la sesión.
- `Ctrl/Cmd + Z` para suspender un proceso.
- `bg %[número_job]` reanuda el proceso en segundo plano (background).
- `fg %[número_job]` trae el proceso al frente (foreground).

Consejo:

Utiliza `sleep [segundos]` para simular un job que "duerme" el sistema.

Matar procesos

- `kill PID` elimina el proceso asociado a su PID.
- `kill -9 PID` elimina con forzado el proceso asociado a su PID.

Historial

- `history` muestra el historial de comandos.
- `!!` repite el último comando.
- `!` expansión de comandos:
  - `!10` ejecuta el comando número *10*.
  - `!ls` ejecuta el último comando que empezó con `ls`.

Cuidado:

Cuidado al escribir `!` en una cadena de texto, ya que puede intentar realizar un 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 crea atajos para comandos.

- `alias atajo=comando`
- `alias ll='ls -l'` crea el comando `ll` que lanzará `ls -l`.
- `unalias ll` elimina el alias.

> **Cuidado:**
>
> El alias también es temporal. Puedes agregarlo a un script de configuración para hacerlo permanente.

Ejercicios procesos y alias

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 termínalo usando su PID.
6. Consulta el espacio en disco.
7. Consulta el historial.
8. Repite el último 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` (opcional, ya que Bash no la requiere).
2. Primera línea `#!/bin/bash` (shebang), para indicar con qué intérprete ejecutar el script.
3. Permisos de ejecución `chmod +x script.sh` (el archivo debe ser ejecutable para correr directamente).

Ejemplo básico

```shell
#!/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

```Shell
name="Brais"
echo "Hola $name"
```

- `name="Brais"` → definición de la variable
- `"$name"` → interpolación de la variable

Aritmética básica

Aunque Bash trata todo como texto, podemos realizar operaciones numéricas.

```Shell
a=5
b=3
let sum=a+b
echo "La suma es $sum"
sum2=$((a+b))
echo "La suma2 es $sum2"
```

- `let` es el comando que se usa para realizar operaciones aritméticas directamente sobre variables.
- `$(( ))` es más habitual usar expansión aritmética en Bash moderno.

Nota:

Script de ejemplo básico: **script.sh**

```shell
#!/bin/bash

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

# Variables
name="Brais"
echo "Hola $name"

# Aritmética
a=5
b=3
let sum=a+b
echo "La suma es $sum"
sum2=$((a+b))
echo "La suma2 es $sum2"
```

Lectura de datos

Permite solicitar datos interactivos durante la ejecución.

```shell
#!/bin/bash
echo "¿Cuál es tu nombre? "
read name
echo "Hola, $name"
read -p "¿Cuál es tu edad? " age
echo "Tu edad es $age"
read -s pass
echo "Tu contraseña es $pass"
```

- `read` → solicita datos y los almacena en la variable definida.
  - `read -p` → muestra el prompt en la misma línea.
  - `read -s` → entrada oculta para contraseñas.

Nota:

Script de ejemplo de lectura: **read_script.sh**

```shell
#!/bin/bash

echo "¿Cuál es tu nombre? "
read name
echo "Hola, $name"

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

read -s pass
echo "Tu contraseña es $pass"
```

Argumentos y parámetros

Los scripts pueden recibir argumentos desde la línea de comandos.

```shell
#!/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`

- `$0` accede al nombre del script (*script.sh*).
- `$1` accede al argumento en la primera posición (*argumento1*).
- `$2` accede al argumento en la segunda posición (*argumento2*).
- `$#` accede al número total de parámetros (*2*).
- `$@` accede a todos los parámetros (*argumento1 argumento2*).

Nota:

Script de ejemplo con parámetros: **params_script.sh**

```shell
#!/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: $@"
```

Ejercicios scripting

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.

LÓGICA

Operadores

Aritméticos

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

Numéricos

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

Cadenas

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

Lógicos

- `&&` AND (Y)
- `||` OR (O)
- `!` NOT (NO)

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

Condicionales

Permiten ejecutar comandos sólo si se cumplen ciertas condiciones.

if

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

if con else

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

if con elif

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

Cuidado:

Puedes crear tantos elif como quieras, pero, en el momento que se cumple la primera condición, se dejan de evaluar las restantes.

if con elif y else

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

case

Para múltiples condiciones que evalúan un mismo valor o menús.

```shell
case variable in
    valor_variable) comando_si_verdadero;;
    valor_variable) comando_si_verdadero;;
    valor_variable) comando_si_verdadero;;
    ...
    *) comando_por_defecto;;
esac
```

```shell
#!/bin/bash
read -p "Elige una opción (a/b/c): " option
case $option in
    a) echo "Elegiste A";;
    b) echo "Elegiste B";;
    c) echo "Elegiste C";;
    *) echo "Opción no válida";;
esac
```

Ejemplo condicionales

Shell

```
#!/bin/bash

num=25
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
read -p "Elige una opción (1/2/3): " option
case $option in
    1) echo "Elegiste 1";;
    2) echo "Elegiste 2";;
    3) echo "Elegiste 3";;
    *) echo "Opción no válida";;
esac
name=Brais
if [ -n $name ]; then
    echo "El nombre existe"
fi
if [ $num -ge 18 ] && [ -n $name ]; then
    echo "Mayor de edad"
fi
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 10*.
- `case $option in` → inspecciona el valor de *option* para asignar el caso de ejecución correcto en base a su valor.
- `-n $name` → comprueba que la cadena asociada a *name* no esté *vacía*.
- `[ $num -ge 18 ] && [ -n $name ]` → comprueba que *num* sea *mayor o igual que 18* y que *name* no esté *vacío*.
- `-e "./script.sh"` → comprueba que el *script existe*.

Nota:

Script de ejemplo con condicionales: **conditionals_script.sh**

Shell

```
#!/bin/bash

# if/elif/else
num=25
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

# case
read -p "Elige una opción (1/2/3): " option
case $option in
      1) echo "Elegiste 1";;
      2) echo "Elegiste 2";;
      3) echo "Elegiste 3";;
      *) echo "Opción no válida";;
esac

# if con algunos operadores
name=Brais
if [ -n $name ]; then
      echo "El nombre existe"
fi
if [ $num -ge 18 ] && [ -n $name ]; then
      echo "Mayor de edad"
fi
if [ -e "./script.sh" ]; then
      echo "El archivo existe"
fi
```

Bucles

for

Recorre un listado de valores.

```shell
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 ejecuta mientras la condición sea verdadera.

```shell
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

Se ejecuta hasta que la condición sea verdadera.

```shell
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

```shell
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.

Nota:

Script de ejemplo con bucles: **loops_script.sh**

Shell

```
#!/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

Función simple

```shell
my_function() {
    echo "Hola desde la función"
}
my_function
```

- `my_function()` → definición
- `my_function` → invocación

Función con parámetros

```shell
my_function_with_params() {
    echo "Hola $1"
}
my_function_with_params Brais
```

- `$1` → accede al primer argumento posicional
- `my_function_with_params argumento` → invocación enviando el parámetro

Función con retorno

```shell
my_function_with_return() {
    return 1
}
my_function_with_return
echo $?
```

- `return` → retorna un valor desde la función
- `$?` → accede al valor del retorno fuera de la función tras su invocación

Consejo:

Retorna `0` como código de salida para indicar éxito en la ejecución y otro número para indicar un error.

Variables globales y locales (ámbito)

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

```shell
name=Brais # global
my_function_2() {
    local msj=", mundo" # local
    echo "Hola $msj $name"
}
echo "Hola desde fuera $msj"
my_function_2
```

- `local` → para definir variables locales.
- `echo "Hola $msj $name"` → puede acceder a las 2 variables desde su ámbito.
- `echo "Hola desde fuera $msj"` → no puede acceder a la variable local fuera del ámbito de la función.

Cuidado:

Utiliza `local` como buena práctica para definir variables locales.

Nota:

Script de ejemplo con funciones: **functions_script.sh**

Shell

```
#!/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 Brais

# Función con retorno

my_function_with_return() {
      return 1
}
my_function_with_return
echo $?

# Variables globales y locales

name=Brais # 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

```shell
#!/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

- `|| command` → ejecuta el comando si hay un error.
- `&& command` → ejecuta el comando si no hay un error.

```shell
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.

Nota:

Script de ejemplo con errores: **errors_script.sh**

```shell
#!/bin/bash

cp file.txt ../Course
if [ $? -ne 0 ]; then
     echo "Error al copiar el archivo"
fi

cp file.txt ../Course || echo "Otra vez se ha producido el error"

cp script.sh ../Course && echo "No se ha producido un error"
```

Ejercicios 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.

CRON

Explorador de archivos

Abre el explorador de archivos del sistema en la ruta en la que nos encontremos desde Bash.

- Linux `xdg-open .`
- macOS `open .`
- Windows `explorer .`

Crontab

Cron es un demonio (demon) 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 utiliza tablas llamadas `crontab`. Pueden existir crontab a nivel de usuario y de sistema a nivel global (*/etc/crontab*).

- `crontab -e` edita tu tabla de cron
- `crontab -l` lista tus tareas programadas
- `crontab -r` elimina tu tabla de cron actual

Sintaxis crontab

Cada línea de `crontab` representa una tarea programada.

```Shell
┌──────────── minuto (0-59)
│ ┌────────── hora (0-23)
│ │ ┌──────── día del mes (1-31)
│ │ │ ┌────── mes (1-12)
│ │ │ │ ┌──── día de la semana (0-6, 0=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 (0-6, 0=domingo)

Consejo:

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

Cuidado:

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

Ejemplos de sintaxis

- `30 2 * * *` se ejecuta todos los días a las 2:30 AM
- `*/5 * * * *` se ejecuta cada 5 minutos
- `0 9 * * 1` se ejecuta cada lunes a las 9:00 AM
- `0 0 1 * *` se ejecuta cada inicio de mes a medianoche

Comodines

- `*`
  Cualquier valor (`* * * * *` se ejecutará cada minuto).
- `,`
  Lista de valores (`0 8,12 * * *` se ejecutará a las 8 y a las 12).
- `-`
  Rango de valores (`0 9-17 * * 1-5` se ejecutará cada horas entre las 9 y las 17 inclusive de lunes a viernes).
- `/`
  Frecuencia (`*/10 * * * *` se ejecutará cada 10 minutos).

Ejercicios cron

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.

EXTRAS

Warp 2

> **Importante:**
>
> **Descarga Warp 2**
> **https://mouredev.link/warp**

Sácale partido a esta herramienta mucho más allá de sus características como terminal (dispone de un plan de uso gratuito).

> **Consejo:**
>
> Explora la configuración (settings) de Warp, sus temas (appearance) y atajos (shortcuts) para mejorar la experiencia de uso.

Enlaces de interés

- **Para programar**
  https://www.warp.dev/code
- **Uso de Agentes con IA**
  https://www.warp.dev/agents
- **Como terminal**
  https://www.warp.dev/terminal
- **Warp Drive (MCPs, reglas, prompts, workflows...)**
  https://www.warp.dev/drive
- **Documentación**
  https://docs.warp.dev
- **Warp University (tutoriales)**
  https://www.warp.dev/university

ZSH

Zsh (Z Shell) es un intérprete de comandos moderno y potente que evoluciona a Bash (y soporta todos sus comandos).

- **Instalación**
  https://github.com/ohmyzsh/ohmyzsh/wiki/Installing-ZSH

Oh My Zsh

Oh My Zsh es un framework open source que ayuda a gestionar la configuración de Zsh para sacarle el máximo partido con temas, plugins...

https://ohmyz.sh

> Comenzar a utilizar Zsh puede ser un paso lógico si quieres especializarte en el uso de Bash y la terminal de línea de comandos.

Cuanto más practiques, más cómodo te sentirás utilizando la terminal.

¿Quieres aprender más sobre Bash?

Aquí tienes mis cursos para aprender Bash desde cero.

Curso gratis en YouTube y GitHub:
https://mouredev.link/bash

Curso con extras (lecciones por tema, ejercicios, soporte, comunidad, test y certificado) en el campus de estudiantes de **mouredev pro**:
https://mouredev.pro/cursos/bash-desde-cero

(Utiliza el cupón "PRO" para acceder con un 10% de descuento a todas las suscripciones y cursos del campus).

mouredev pro

Estudia programación y desarrollo de software de manera diferente

mouredev.pro