Bash

Linux man pages online | Linux man pages
Bash Reference Manual | Introduction to Bash | BashGuide

Bash es una shell. Suele ser la shell por defecto en Linux.
Otras shells son **Bourne shell** (sh), **PowerShell** (pwsh, Windows), **Z shell**, (zsh, macOS).
Algunos comandos coinciden entre distintas shells, pero no tiene por qué cumplirse siempre.

¿Qué shell estoy usando? `echo $SHELL`. Si es Bash, el resultado será `/bin/bash` o `/usr/bin/bash`.
`which bash` también devuelve la ruta de Bash. Si estamos en otra shell y queremos lanzar Bash, simplemente accedemos a esa ruta.
Esto funciona así en Linux y Mac. En Windows también funciona si se usa un emulador como Git for Windows.

The full syntax for a Bash command is:

```
command [options] [arguments]
```

For example:

```
ls -al /etc
│   │     └ argument: path name
│   │
│   └ options: 'all' flag (a) and 'long form' flag (l)
│
└ command
```

Opciones

Las opciones se prefijan con `--` si son largas o con `-` si son abreviaciones.
Por ejemplo, `ls --all` es lo mismo que a `ls -a`.
Si vamos a poner varias opciones abreviadas se pueden indicar juntas o separadas: `ls -al` es lo mismo que `ls -a -l`.

Obtener ayuda

```
man <command>
<command> --help
apropos
help
```

Por ejemplo:

```
man ls
ls --help
```

Resumen de comandos comunes

`ls`	List files and directories in the current directory
`touch`	Create file
`pwd`	Print working directory
`cat`	Show the content of a text file
`sudo`	Superuser do
`mkdir`	Make directory
`cd`	Change directory
`rmdir`	Remove an empty directory
`rm`	Remove file or directories
`cp`	Copy

Comandos

cat

Lee el contenido de un archivo (con**cat**enate files to standard output).

```
$ cat file.txt
Lorem ipsum
```

Con la flag `-n` (**n**umber) se muestran las líneas numeradas.
Con la flag `-e` se muestra `$` cuando hay un salto de línea.

cd

Cambiar directorio. Recomendable usar `Tab` para autocompletar.
Se puede añadir `/` al final, pero no tiene ningún significado: `cd folder/` es lo mismo que `cd folder`.
Sin embargo, una `/` al principio indica que la ruta se hace desde la raíz.
`.` y `./` equivalen al directorio actual.
Para cambiar a una carpeta ubicada en el directorio actual las siguientes formas son equivalentes:

```
cd folder   =   cd folder/   =   cd ./folder   =   cd ./folder/
```

Cambiar a la carpeta `subfolder`, que está dentro de `folder`, que está en el directorio actual:

```
cd folder/subfolder
```

Si el nombre de la carpeta contiene espacios se usan dobles comillas (`""`) o un character escape (`\`):

```
cd "my folder"
cd my\ folder
```

Subir directorios

`cd ..` = `cd ../`	Cambia al directorio superior
`cd ../..`	Cambia al directorio superior del directorio superior
`cd ../../..`	Cambia al directorio superior del directorio superior del directorio superior
`cd ../folder`	Cambia a la carpeta `folder`, que está en el directorio superior

Directorios home y raíz

`cd ~` = `cd`	Cambia al directorio home (el inicial, pero no el raíz)
`cd /`	Cambia al directorio raíz (no es lo mismo que home)
`cd ~/folder`	Cambia a `folder`, que está en el directorio home
`cd /folder`	Cambia a `folder`, que está en el directorio raíz

Retroceder

`cd -`	Cambia al último directorio visitado

chmod

(**Ch**ange file **mod**e bits). Configura los permisos de lectura (`r`), escritura (`w`) y ejecución (`x`).

Podemos comprobar los permisos de un archivo con `ls -l`. Veremos si el usuario/owner, el grupo y others/public tienen permisos para leer, escribir o ejecutar el archivo. Por ejemplo, `-rw-r--r--`, indica que el usuario tiene permiso de lectura y escritura (`rw-`), pero el grupo y others sólo tienen permiso de lectura (`r--`).
Para modificar los permisos usamos `chmod` con `u` para usuario, `g` para grupo y `o` para others.
Por ejemplo, para añadir (`+`) permiso de escritura (`w`) al grupo (`g`):

```
$ chmod g+w file.txt
```

chown

Modifica el propietario de un archivo (**ch**ange **ow**ner).
Los argumentos son el nombre de usuario y el archivo:

```
$ chown Daniel file.txt
```

cp

Copia (**co**p**y**) el primer archivo indicado al segundo:

```
$ cp file.txt file2.txt
```

Con `cp -r` se pueden copiar directorios recursivamente.

date

Muestra la fecha.

```
$ date
Sat Oct 19 23:41:45     2024
```

echo

Como `console.log`. Las variables se prefijan con `$`:

```
$ echo "Hello world"
$ echo $PATH
$ echo $SHELL
```

find

El comando `find` busca archivos desde un directorio de comienzo (por defecto, `.`).
Admite búsquedas por nombre de archivo, tamaño, fecha, permisos y un largo etcétera.
Por ejemplo, para buscar archivos con extensión `.txt`:

```
$ find . -name *.txt
```

Para ejecutar un comando sobre cada resultado se utiliza `-exec <command> {} +`, donde `{}` indica el elemento encontrado y `+` indica el final de la ejecución.

grep

(**G**lobal **R**egular **E**xpression **P**rint). Busca expresiones en el archivo indicado. Admite RegEx.

```
$ grep "hello" file.txt
```

Para buscar recursivamente en varios archivos y directorios se usa la flag `-r`:

```
$ grep -r "hello" .
```

ln

Crea enlaces (**l**i**n**ks) entre archivos.
Por defecto crea enlaces hard. Para crear enlaces blandos o simbólicos se usa la opción `--symbolic` o `-s`.
El archivo de origen tiene que existir, pero el de destino no debería existir antes de ejecutar el comando.

```
$ ln origin.txt target.txt
```

ls

(**L**ist **s**torage / **l**i**s**t directory contents). Muestra una lista de archivos y carpetas del directorio actual.

```
$ ls
```

`ls -l`	(**l**ong)	Con más detalles.
`ls -a`	(**a**ll)	Muestra todos los archivos, incluso los ocultos.
`ls -l folder`		Muestra el contenido de la carpeta `folder` con la opción de más detalles.
`ls */*`		Muestra el contenido del directorio y subdirectorios.

man

Obtiene ayuda sobre un comando.

```
$ man mkdir
```

`man` tiene varias secciones: `1` (herramientas de Unix), `2` (syscalls), `3` (librerías de C).
Por ejemplo: `man 1 cat`, `man 2 write`, `man 3 strftime`. Si no se pone el número saldrá el primer comando que encuentre.
Si no sabemos qué página necesitamos: `apropos <texto>` busca todas las páginas de man que contienen el texto.

mkdir

Crea un nuevo directorio (**m**a**k**e **dir**ectory).

```
$ mkdir folder
```

Si queremos crear subdirectorios hay que añadir la flag `-p` para crear el **p**arent directory. De lo contrario saldrá un error.

```
$ mkdir folder/subfolder/subfolder -p
```

mv

Mueve (**m**o**v**e) uno o varios archivos a un directorio. Es como `cp` pero elimina el archivo original.
El primer argumento indica el archivo y el segundo la ruta de destino.

```
$ mv file.txt ../file.txt
```

Podemos mover varios archivos a la vez al mismo directorio:

```
$ mv index.html styles.css script.js ../
```

pwd

Muestra el directorio actual (**p**rint **w**orking **d**irectory).

```
$ pwd
/c/Users/Daniel
```

read

Guarda lo que escriba el usuario en el teclado en una o varias variables.
Por ejemplo, la siguiente línea crea una variable `age` y guarda lo que escriba el usuario:

```
$ read age
```

Se pueden crear más variables. Por ejemplo, así se crean tres variables (`a` , `b` y `c`). En cada una de ellas se guarda una palabra. Si sobran palabras se guardarán en la última variable:

```
$ read a b c
```

Cuando `read` se usa sin argumentos, la respuesta del usuario se guardará en `$REPLY` .
La opción `-p` (prompt) permite añadir una frase.

```
$ read -p "Enter your age " age
```

rm

Elimina un archivo (re**m**ove).

```
$ rm file.txt
```

Si añadimos la flag `-i` nos pedirá una confirmación:

```
$ rm -i file.txt
rm: remove regular empty file 'file.txt'?
```

Para borrar directorios no vacíos hace falta la flag `-rf` (**r**ecursive **f**orce). Esto borra recursivamente el directorio y todos los directorios y archivos que hubiera dentro.

```
$ rm -rf folder
```

rmdir

Elimina un directorio vacío.

```
$ rmdir folder
```

ssh-keygen

Genera una clave privada y otra pública. Podemos pulsar `Enter` 3 veces para aceptar la configuración por defecto (recomendable).

```
$ ssh-keygen
```

Hay varios tipos de claves: `ecdsa`, `ecdsa-sk`, `ed25519`, `ed25519-sk` y `rsa`, que se pueden especificar con la opción `-t`. El tipo por defecto es `ed25519`, pero esto puede variar en cada sistema operativo.
Por ejemplo, para crear claves `rsa`:

```
$ ssh-keygen -t rsa
```

Este comando generará dos archivos en el directorio oculto `.ssh`: `id_rsa` e `id_rsa.pub`. El primero es la clave privada (que no se debe compartir) y el segundo es la clave pública.
También puede generarse un archivo `known_hosts`.
Para mostrar el contenido de la clave pública: `cat id_rsa.pub`.
Al final de la clave pública se escribe automáticamente un email. Este email no tiene ningún significado, es un comentario que sirve para identificar la clave. Para personalizar el email o poner otro mensaje:

```
$ ssh-keygen -C "mensaje"
```

su

(**S**ubstitute **u**ser). Permite ejecutar comandos como un usuario distinto. Si no se especifica el usuario se ejecuta la shell como root, es decir, como administrador.

sudo

(**S**uper **u**ser **do**). Normalmente ejecutamos los comandos con privilegios de usuario, pero si los prefijamos con `sudo` podemos acceder a privilegios de administrador. Nos pedirá la contraseña.
El usuario administrador se llama `superuser` o `root`, su user ID es `0` y tiene todos los privilegios posibles.

sudo su

The often-used `sudo su` combination works as follows: first `sudo` asks you for your password, and, if you're allowed to do so, invokes the next command (`su`) as a super-user. Because `su` is invoked by root, it does not require you to enter the target user's password. So, `sudo su` allows you to open a shell as another user (including root), if you're allowed super-user access by the `/etc/sudoers` file.

touch

Crea el archivo indicado. En realidad sirve para modificar las fechas de modificación y acceso de un archivo, pero si el archivo no existe lo crea.

```
$ touch file.txt
```

Se pueden crear varios archivos a la vez, por ejemplo:

```
$ touch index.html styles.css script.js
```

which

Devuelve la ruta de un binary (ejecutable) para saber dónde está el código que ejecutamos.

```
$ which ls
/usr/bin/ls
```

Caracteres genéricos

*

El signo `*` se utiliza como un comodín para cualquier secuencia de caracteres de cualquier longitud, incluso ningún caracter. Por ejemplo:

```
ls file*.txt
```

Sirve para `file.txt`, `file0.txt`, `filea.txt`, `file21351020354.txt`, etc.
También se puede aplicar a las extensiones:

```
ls file.*
```

Sirve para `file.txt`, `file.jpg`, `file.css`, etc.
Se excluyen los archivos con un `.` inicial, que suelen ser archivos ocultos o de configuración.

?

El signo `?` indica cualquier caracter (excepto `.`), pero sólo uno. Por ejemplo:

```
ls file?.txt
```

El comando anterior sirve para `file3.txt`, `files.txt`, `fileZ.txt`, pero no para `file.txt` ni `file33.txt`.
Para indicar más de un caracter ponemos varios signos: dos caracteres se indican con `??`, tres con `???`, etc.

```
ls f?le??.txt
```

Sirve para `file00.txt`, `fole00.txt`, `fuless.txt`, etc.
Una cadena como por ejemplo `????` indica cualquier string de cuatro caracteres.

[]

Los corchetes `[]` indican un caracter dentro de ese conjunto.
Por ejemplo, `file[s0x]123.txt` sólo sirve para `files123.txt`, `file0123.txt` y `filex123.txt`, ninguna opción más.
Se pueden usar varios conjuntos seguidos, por ejemplo, `file[123][xyz].txt`.

Rangos

Los corchetes también sirven para indicar rangos separados por un guión. Por ejemplo, `[0-9]` indica cualquier dígito de `0` a `9` incluidos, `[a-h]` indica cualquier letra minúscula desde `a` hasta `h`.

```
ls file[0-9].txt
```

Sirve para `file0.txt`, `file1.txt`, `file2.txt` … hasta `file9.txt`.
En los mismos corchetes se pueden declarar varios conjuntos. Por ejemplo, `[a-zA-Z]` sirve para letras minúsculas y mayúsculas.

Clases

Algunos conjuntos se pueden expresar en clases:

`[:alnum:]`	`[a-zA-Z0-9]`	Letras minúsculas, mayúsculas y dígitos
`[:alpha:]`	`[a-zA-Z]`	Letras minúsculas y mayúsculas
`[:lower:]`	`[a-z]`	Letras minúsculas
`[:upper:]`	`[A-Z]`	Letras mayúsculas
`[:digit:]`	`[0-9]`	Dígitos del 0 al 9
`[:punct:]`	Signos de puntuación: `!"#$%&'()*+,-./:;<=>?@[]\^_`{}|~`	
`[:space:]`	Espacios, tabulaciones y saltos de línea `\n` y `\r`	
`[:blank:]`	Espacios y tabulaciones	

Otras clases disponibles son `ascii`, `cntrl`, `digit`, `graph`, `print`, `word` y `xdigit`.

Estas clases se usan a su vez dentro de otros corchetes. Por ejemplo:

```
ls [[:digit:]][[:alpha:]]*
```

Sirve para un dígito seguido de una letra seguida de cualquier cadena, por ejemplo: `0a`, `1Ahola`, `2xfile.txt`, etc.

Exclusión

Con `!` o `^` se indica la negación del patrón.
Por ejemplo, `ls [^ab]*` sirve para cualquier archivo que NO comience por `a` o por `b`.

{}

Las llaves `{}` permiten separar patrones. Por ejemplo:

```
ls {fi,im}*
```

Corresponde a todos los archivos que empiecen por `fi` o `im`, como `file.txt`, `img001.jpg`, `file-01.doc`, etc.
Dentro de las llaves se pueden poner otros caracteres genéricos:

```
ls {*ro[sj]*,?a*}
```

Sirve para `banco`, `rosado`, `rojo`.

Las llaves también pueden indicar el número de veces que debe repetirse un patrón.
Por ejemplo, `[[:digit:]]{4}` requiere exactamente 4 dígitos, como `0000`, `1234` o `9999`.

Comillas

Las cadenas de texto entre comillas simples `'string'` se interpretan de manera literal, es decir, si hay un variable dentro no se interpretará como variable.
En cambio, si hay variables entre comillas dobles `"string"` sí que se interpretarán como variables.

```
$ echo 'directorio $HOME'     # directorio $HOME
$ echo "directorio $HOME"     # directorio /c/Users/Daniel
```

Operadores I/O

Redirecciones

>

Por defecto el output de un comando se muestra en la pantalla, pero el operador `>` redirige el output a otro destino. Por ejemplo:

```
ls > listing.txt
```

Este comando hace que el output de `ls` se guarde en el archivo `listing.txt`, en lugar de mostrarse en la pantalla. Si el archivo ya existe se borrará el contenido anterior.
El operador `2>` redirige los posibles mensajes de error (0 es entrada, 1 es salida y 2 es error).

>>

El operador `>>` funciona como `>` pero si el archivo ya existe no sobreescribe, sino que añade al final.

```
ls >> listing.txt
```

<

Por defecto el input es el teclado del usuario, pero el operador `<` permite que el input de un comando sea el contenido de un archivo. Por ejemplo:

```
sort < file.txt
```

Combinar redirecciones

Se pueden combinar varios `>` y `<`. Por ejemplo:

```
sort < file.txt > sorted_file.txt
```

Piping

El operador de piping `|` redirige el output de un proceso a la entrada de otro proceso. De esta manera se pueden encadenar varios procesos seguidos. Por ejemplo:

```
cat file.txt | fmt | pr | lpr
```

The output from `cat file.txt` goes to `fmt`, the output from `fmt` goes to `pr`, and so on.
`fmt` formats the results into a tidy paragraph. `pr` paginates the results. And `lpr` sends the paginated output to the printer. All in a single line!

Editores de texto

Con los comandos `code`, `nano` o `vim` abrimos un editor de texto para modificar los archivos. Esto depende del sistema operativo, distro, etc. `code` funciona si VS Code está instalado, `nano` suele estar instalado por defecto en Linux. `nano` y `vim` se pueden usar desde la propia terminal.

nano

Podemos usar `nano file.txt` para abrir un archivo o crearlo si no existe (en lugar de `touch`).

`Y` / `N`	Confirmar / Cancelar
`Ctrl` + `S`	Guardar archivo
`Ctrl` + `O`	Guardar como…
`Ctrl` + `X`	Cerrar editor
`Ctrl` + `W`	Buscar
`Ctrl` + `6`	Iniciar selección
`Alt` + `6`	Copiar selección
`Ctrl` + `U`	Pegar
`Alt` + `U`	Deshacer
`Ctrl` + `G`	Manual de instrucciones

vim

Vim tiene cuatro modos de edición: Command (`Esc`), Insert (`i`), Visual (`v`) y Command-Line (`:`).
En el modo Command-Line podemos ejecutar órdenes con el prefijo `:`. Por ejemplo:

`:w`	Guardar
`:q`	Cerrar
`:q!`	Cerrar sin guardar cambios
`:x` / `:wq`	Guardar y cerrar

Para cortar, copiar y pegar texto accedemos al modo visual (`v`) y usamos `y` para copiar la selección, `d` para cortar y `p` para pegar.

sed

`sed` (**s**tream **ed**itor) es otro editor de texto pero no es interactivo.
Como admite RegEx se puede usar para sustituir patrones. Hay que indicar un archivo de destino o usar la flag `-i`. Por ejemplo:

```
sed 's/mom/dad/' main.js > main2.js
```

Scripts de Bash

En un archivo de texto se pueden escribir comandos de Bash y que luego se ejecuten línea por línea.
Se pueden guardar con extensión `.sh`. Por ejemplo, `script.sh`.
Pero también se pueden guardar sin extensión, `script`, como veremos más adelante.

De momento no se puede ejecutar. Antes hay que marcarlo como ejecutable:

```
$ chmod +x script.sh
$ sudo chmod +x script.sh
```

Cuando un script es ejecutable se muestra en un color (suele ser verde) en la lista `ls`.

Y para ejecutarlo (si está en el directorio actual):

```
$ ./script.sh
```

Shebang

Normalmente los archivos de scripts de Bash se escriben sin extensión: `script`, en lugar de `script.sh`.
En la primera línea se añade `#!/bin/bash`, un *shebang* o *hashbang* para indicar con qué ejecutable se tiene que interpretar. Normalmente `#` indica comentario, pero en este caso es una excepción.

```
script
#!/bin/bash

echo "Hello World!"
```

Si quisiéramos usar otro intérprete añadiríamos `#!/bin/sh` (Bourne Shell), `#!/usr/bin/pwsh` (PowerShell), `#!/usr/bin/env node` (Node.js), `#!/usr/bin/env python3` (Python), etc.

Variables

Una variable se declara simplemente así: `name="Daniel"`. Las dobles comillas no son obligatorias, pero a veces son convenientes.
Para usar una variable la prefijamos con `$`, por ejemplo, `echo $name`.
Se pueden usar dentro de strings, por ejemplo: `echo "My name is $name."`. Pero en este caso sí hay que usar dobles comillas.

En una variable podemos guardar el resultado de un comando. Para ello envolvemos el comando en `$()`.
Por ejemplo, para guardar el resultado de `ls` en la variable `files` escribimos:

```
files=$(ls)
```

Esto se conoce como subshell.

En Bash hay variables de entorno por defecto: `$USER` o `$USERNAME`, `$SHELL`, `$PWD`, `$HOME`, etc.
Si ejecutamos `env` se mostrarán todas las variables de entorno.

read

Con `read` guardamos en una variable lo que introduzca el usuario a mano.

```
echo "Enter your age:"
read age
```

En la variable `age` se habrá guardado lo que haya escrito el usuario.

Operaciones

Las operaciones se prefijan con `expr` para indicar a Bash que evalúe una expresión.

```
$ expr 30 + 10
$ expr 30 - 10
$ expr 30 / 10
$ expr 30 \* 10
```

El símbolo de multiplicación `*` se pone con un escape character `\*` porque el asterisco significa "todo".

Operadores

`-eq`	Equal to
`-ne`	Not equal to
`!`	Negación
`-gt`	Greater than
`-ls`	Less than

Condiciones

Un bloque condicional se escribe entre `if` e `fi` (`if` escrito al revés). Por ejemplo:

```
mynum=200

if [ $mynum -eq 200 ]; then
  echo "True"
else
  echo "False"
fi
```

Los corchetes tienen que ver con algo del comando `test` y no es necesario usarlos siempre.

Bucle for

Un bucle `for` recorre una lista de valores y aplica una serie de comandos por cada valor.

```
for <variable> in <lista de valores>
do
    <comandos>
done
```

La lista de valores puede consistir en los nombres de archivo en ese directorio. Por ejemplo:

```
for file in *.jpg
do
   echo "imagen jpg: $file"
```

```
    done
```

Bucle while

Ejecuta una serie de comandos mientras una condición sea `true`. Por ejemplo:

```
    ((i=0))

    while ((i++ < 10))
    do
        echo "\$i=$i"
    done
```

Ejemplos de scripts

Crear carpetas

Un bucle para crear una carpeta por cada línea de texto en un archivo `files.txt`.

```
#! /bin/bash

LINE=1

while read -r CURRENT_LINE
  do
  mkdir $CURRENT_LINE
  ((LINE++))
done < "files.txt"
```

Crear carpeta por cada archivo

Un bucle para crear una carpeta por cada archivo con extensión html en ese directorio. Luego mueve el archivo a la carpeta creada.

```
#! /bin/bash

for file in ./*.html; do
  mkdir "${file%.*}" && mv "$file" "${file%.*}"
done
```

Lo mismo, en una línea:

```
for file in ./*.html; do mkdir "${file%.*}" && mv "$file" "${file%.*}"; done
```

Mover archivos según su extensión

Crea tres carpetas `photos`, `docs` y `scripts` y mueve los archivos según su extensión a una de esas tres carpetas (los `.jpg` van a `photos`, `.doc` a `docs`, `.sh` a `scripts`).

```
mkdir photos docs scripts
mv *.jpg photos
mv *.doc docs
mv *.sh scripts
```

Configuración de Bash

Los archivos `~./bashrc` y `~./bash_profile` son ambos archivos de configuración.
También podemos encontrar `profile`, `bash.bashrc`, `bash_login`, etc.

.bashrc

Variables de entorno
En el archivo `.bashrc` podemos añadir variables de entorno. Por ejemplo, podemos modificar la variable de entorno `PATH` para referirnos a ejecutables adicionales. Normalmente se escriben solas cuando instalamos un programa, pero a veces puede ser necesario añadirlas a mano.

alias
Podemos añadir alias, es decir, referencias a comandos que usamos con frecuencia. Por ejemplo:

```
alias ll='ls -alF'
alias la='ls -A'
alias l='ls -CF'
```

$PS1 y $PS2
Las variables `$PS1` y `$PS2` (**P**rompt **S**tring) configuran el aspecto del prompt principal y el de espera. También se pueden usar para cambiar el tema de color.

```
PS1='\u@\h \W$ '
```

`\a`	Emite un bip
`\d`	Fecha
`\h` y `\H`	Nombre del equipo (y nombre completo con dominio)
`\t`	Hora
`\u`	Usuario
`\w` y `\W`	Ruta o nombre del directorio

Para reiniciar la terminal con el nuevo aspecto: `source ~/.bashrc`.