**¿Por qué usar un sistema de control de versiones como Git?**

Un [sistema de control de versiones como Git](https://platzi.com/clases/1557-git-github/20215-que-es-git/) nos ayuda a guardar el historial de cambios y crecimiento de los archivos de nuestro proyecto.

En realidad, los cambios y diferencias entre las versiones de nuestros proyectos pueden tener similitudes, algunas veces los cambios pueden ser solo una palabra o una parte específica de un archivo específico. Git está optimizado para guardar todos estos cambios de forma atómica e incremental, o sea, aplicando cambios sobre los últimos cambios, estos sobre los cambios anteriores y así hasta el inicio de nuestro proyecto.

* El comando para iniciar nuestro repositorio, o sea, indicarle a Git que queremos usar su sistema de control de versiones en nuestro proyecto, es git init.
* El comando para que nuestro repositorio sepa de la existencia de un archivo o sus últimos cambios es git add. Este comando no almacena las actualizaciones de forma definitiva, únicamente las guarda en algo que conocemos como “Staging Area” (área de montaje o ensayo).
* El comando para almacenar definitivamente todos los cambios que por ahora viven en el staging area es git commit. También podemos guardar un mensaje para recordar muy bien qué cambios hicimos en este commit con el argumento -m "Mensaje del commit".
* Por último, si queremos mandar nuestros commits a un servidor remoto, un lugar donde todos podamos conectar nuestros proyectos, usamos el comando git push.

## Comandos básicos de git

* git init: inicializa un repositorio de GIT en la carpeta donde se ejecute el comando.
* git add: añade los archivos especificados al área de preparación (staging).
* git commit -m “commit description”: confirma los archivos que se encuentran en el área de preparación y los agrega al repositorio.
* git commit -am “commit description”: añade al staging area y hace un commit mediante un solo comando. (No funciona con archivos nuevos)
* git status: ofrece una descripción del estado de los archivos (untracked, ready to commit, nothing to commit).
* git rm (. -r, filename) (–cached): remueve los archivos del index.
* git config --global user.email [tu@email.com](mailto:tu@email.com): configura un email.
* git config --global [user.name](http://user.name) : configura un nombre.
* git config --list: lista las configuraciones.

## Analizar cambios en los archivos de un proyecto Git

* git log: lista de manera descendente los commits realizados.
* git log --stat: además de listar los commits, muestra la cantidad de bytes añadidos y eliminados en cada uno de los archivos modificados.
* git log --all --graph --decorate --oneline: muestra de manera comprimida toda la historia del repositorio de manera gráfica y embellecida.
* git show filename: permite ver la historia de los cambios en un archivo.
* git diff : compara diferencias entre en cambios confirmados.

## Volver en el tiempo con branches y checkout

* git reset --soft/hard: regresa al commit especificado, eliminando todos los cambios que se hicieron después de ese commit.
* git checkout : permite regresar al estado en el cual se realizó un commit o branch especificado, pero no elimina lo que está en el staging area.
* git checkout – : deshacer cambios en un archivo en estado modified (que ni fue agregado a staging)

## git rm y git reset

### **git rm**:

Este comando nos ayuda a eliminar archivos de Git sin eliminar su historial del sistema de versiones. Esto quiere decir que si necesitamos recuperar el archivo solo debemos “viajar en el tiempo” y recuperar el último commit antes de borrar el archivo en cuestión.

git rm no puede usarse por sí solo, así nomás. Se debe utilizar uno de los flags para indicar a Git cómo eliminar los archivos que ya no se necesitan en la última versión del proyecto:

* git rm --cached : elimina los archivos del área de Staging y del próximo commit, pero los mantiene en nuestro disco duro.
* git rm --force : elimina los archivos de Git y del disco duro. Git siempre guarda todo, por lo que podemos acceder al registro de la existencia de los archivos, de modo que podremos recuperarlos si es necesario (pero debemos aplicar comandos más avanzados).

### **git reset**

Con git reset volvemos al pasado sin la posibilidad de volver al futuro. Borramos la historia y la debemos sobreescribir.

* git reset --soft: Vuelve el branch al estado del commit especificado, manteniendo los archivos en el directorio de trabajo y lo que haya en staging considerando todo como nuevos cambios. Así podemos aplicar las últimas actualizaciones a un nuevo commit.
* git reset --hard: Borra absolutamente todo. Toda la información de los commits y del área de staging se borra del historial.
* git reset HEAD: No borra los archivos ni sus modificaciones, solo los saca del área de staging, de forma que los últimos cambios de estos archivos no se envíen al último commit. Si se cambia de opinión se los puede incluir nuevamente con git add.

## Ramas o Branches en git

Al crear una nueva rama se copia el último commit en esta nueva rama. Todos los cambios hechos en esta rama no se reflejarán en la rama master hasta que hagamos un merge.

* git branch : crea una nueva rama.
* git checkout : se mueve a la rama especificada.
* git merge : fusiona la rama actual con la rama especificada y produce un nuevo commit de esta fusión.
* git branch: lista las ramas generadas.

# ¿Qué es Git?

**Git es un sistema de control de versiones distribuido** que te permite registrar los cambios que haces en tus archivos y volver a versiones anteriores si algo sale mal. Fue diseñado por Linus Torvalds para garantizar la eficiencia y confiabilidad del mantenimiento de versiones de aplicaciones que tienen un gran número de archivos de código fuente.

* Git está optimizado para guardar cambios de forma incremental.
* Permite contar con un historial, regresar a una versión anterior y agregar funcionalidades.
* Lleva un registro de los cambios que otras personas realicen en los archivos.

Git fue diseñado para operar en un entorno Linux. Actualmente, es multiplataforma, es decir, es compatible con Linux, MacOS y Windows. En la máquina local se encuentra Git, se utiliza bajo la terminal o línea de comandos y tiene comandos como merge, pull, add, commit y rebase, entre otros.

## Para qué proyectos sirve Git

Con Git se obtiene una mayor eficiencia usando archivos de texto plano, ya que con archivos binarios no puede guardar solo los cambios, sino que debe volver a grabar el archivo completo ante cada modificación, por mínima que sea, lo que hace que incremente demasiado el tamaño del repositorio.

“Guardar archivos binarios en el repositorio de Git no es una buena práctica, únicamente deberían guardarse archivos pequeños (como logos) que no sufran casi modificaciones durante la vida del proyecto. Los binarios deben guardarse en un CDN”.

[Cómo usar git stash](https://platzi.com/clases/1557-git-github/19984-stash/)

## Características de Git

Git te ayuda a trabajar de manera más organizada y colaborativa en proyectos de desarrollo de software. Estas son algunas de sus principales características:

### Control de versiones

Git almacena la información como un conjunto de archivos. Te permite llevar un registro de los cambios que haces en tus archivos, lo que significa que siempre puedes volver a versiones anteriores si algo sale mal.

### Ramificación

Puedes crear ramas en tu proyecto, lo que te permite trabajar en diferentes características o aspectos del mismo sin afectar el trabajo de los demás.

### Colaboración

En Git, varias personas pueden trabajar en diferentes aspectos del proyecto al mismo tiempo.

### Seguridad

No existen cambios, corrupción en archivos o cualquier alteración sin que Git lo sepa. Git cuenta con 3 estados en los que es posible localizar archivos: Staged, Modified y Committed.

### Flexibilidad

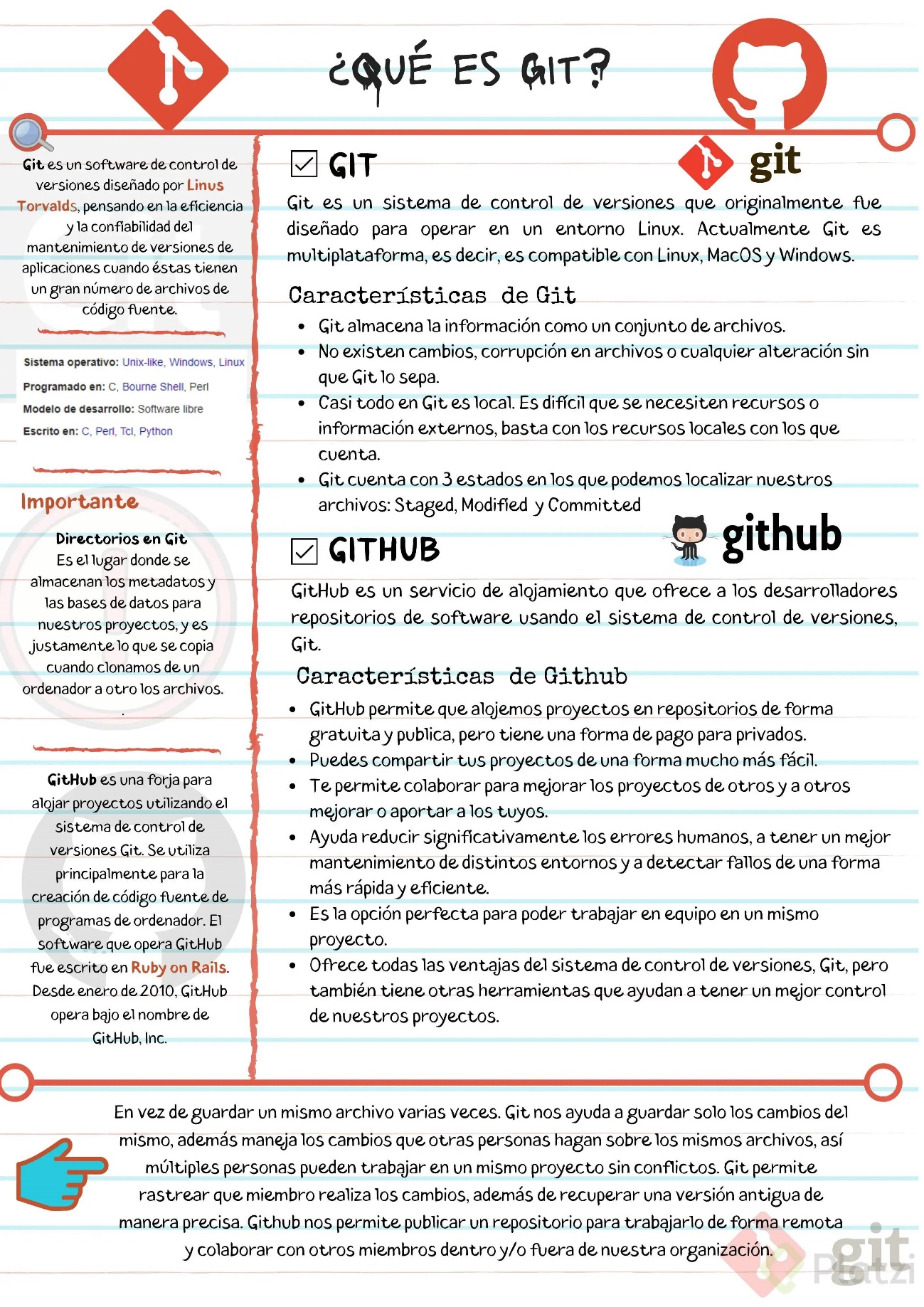
Casi todo en Git es local. Es difícil que se necesiten recursos o información externos, basta con los recursos locales con los que cuenta.

### Comandos

Git tiene una sintaxis de comandos bastante sencilla y fácil de aprender, lo que lo hace accesible incluso para principiantes en programación.

## ¿Qué es un sistema de control de versiones?

El SCV o VCS (por sus siglas en inglés) es un sistema que registra los cambios realizados sobre un archivo o conjunto de archivos a lo largo del tiempo, de modo que puedas llevar el historial del ciclo de vida de un proyecto, comparar cambios a lo largo del tiempo, ver quién los realizó o revertir el proyecto entero a un estado anterior.

Cualquier tipo de archivo que se encuentre en un ordenador puede ponerse bajo control de versiones.  


## ¿En qué se diferencia de Github?

Github es una plataforma de desarrollo colaborativo para alojar proyectos utilizando el sistema de control de versiones Git. Se emplea principalmente para la creación de código fuente de programas de computadora.

Puede considerarse a Github como la red social de código para los programadores y en muchos casos es visto como un curriculum vitae, pues aquí se guarda el portafolio de proyectos de programación.

## Características de Github

* GitHub permite alojar proyectos en repositorios de forma gratuita y pública, pero tiene una forma de pago para privados.
* Puedes compartir fácilmente tus proyectos.
* Permite colaborar para mejorar los proyectos de otros y a otros mejorar o aportar a los tuyos.
* Ayuda a reducir significativamente los errores humanos, a tener un mejor mantenimiento de distintos entornos y a detectar fallos de una forma más rápida y eficiente.
* Es la opción perfecta para poder trabajar en equipo en un mismo proyecto.
* Ofrece todas las ventajas del sistema de control de versiones Git, pero también tiene otras herramientas que ayudan a tener un mejor control de los proyectos.

# Instalando GitBash en Windows

Al implementar **Git Bash**, podrás tener una terminal de Git en el sistema operativo Windows. Esto te ayudará a acceder a tus repositorios al conectarte desde GitHub. Descubramos más acerca de esta integración.

## ¿Qué es Git bash?

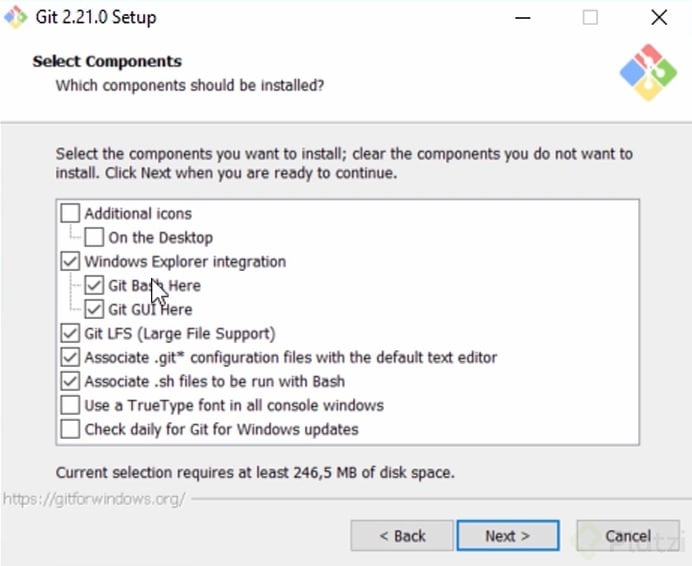
**Git Bash** es una aplicación en línea que une las utilidades de Git y Bash en un sistema operativo de Windows. Mejora la experiencia de la línea de comandos y se comporta como una especie de intérprete para un sistema operativo Unix, pero adaptado para funcionar en Windows.

Además, el programa permite a los usuarios acceder a las herramientas de Git,y realizar tareas de control de versiones a través de comandos de texto en lugar de una interfaz gráfica de usuario. Incluye otros comandos de Unix y herramientas como ssh , scp y rsync

## Instalación de Git bash paso a paso

Para instalar este sistema de control de versiones en Windows, simplemente dirígete al [repositorio de descarga](https://git-scm.com/downloads).

1. Una vez descargado, ejecútalo como cualquier otra aplicación de Windows.
2. Al iniciar el instalador, asegúrate de marcar la opción de instalar Git Bash en el computador. Esto nos permitirá correr comandos de Linux en la consola sin problemas para trabajar con Git.



## ¿Cómo utilizar Git Bash?

Git bash funciona al igual que el Bash estándar en UNIX. Una vez instalada solo tienes que abrirla en tu ordenador y luego podrás inicializar un nuevo repositorio GIT.

1. Navega hasta el directorio que deseas usar como repositorio y ejecuta el comando “git init”. Esto creará un nuevo repositorio Git en el directorio actual.
2. Abre Git Bash en tu ordenador. Esto se puede hacer de varias maneras, como hacer clic con el botón derecho del ratón en una carpeta y seleccionar “Git Bash Here” en el menú contextual.
3. Inicializa un nuevo repositorio Git.

## ¿Cómo navegar por las carpetas en Bash?

El comando de Bash pwd se utiliza para imprimir el directorio de trabajo actual, que es la carpeta o ruta en la que reside la sesión actual de Bash. Es equivalente a ejecutar cd en un terminal DOS de Windows.

Por otro lado, el comando de Bash ls se emplea para enumerar el contenido del directorio de trabajo actual. Este comando es equivalente a ejecutar el comando DIR en un terminal de host de consola de Windows.

Es importante destacar que tanto el host de la consola de Bash como el de Windows tienen un comando cd, que se emplea para cambiar de directorio. Si se invoca cd con el nombre de un directorio adjunto, se cambiará el directorio de trabajo actual de las sesiones del terminal al argumento del directorio especificado.

## Comandos complementarios de Git Bash

Windows y Linux difieren en comandos, en la forma en que registran el “enter” y en muchas otras cosas. Cuando instales Git Bash en Windows, debes elegir entre trabajar con la forma de Windows o la forma de UNIX (Linux y Mac). Algunos comandos que pueden ser familiares son:

* git checkout
* git commit
* git clone
* git pull
* git push

En entornos de desarrollo profesionales, es común que las personas utilicen sistemas operativos diferentes. Si todos podemos usar los mismos comandos, el trabajo resultará más fácil.

Los comandos de UNIX son los más frecuentes en equipos de desarrollo, por lo que, a menos que trabajes con tecnologías nativas de Microsoft (por ejemplo, .NET), es recomendable elegir la opción de la terminal tipo UNIX para una mejor compatibilidad con todo el equipo.

# Editores de código, archivos binarios y de texto plano

Un editor de código o **IDE** es una herramienta que nos brinda muchas ayudas para escribir código, algo así como un bloc de notas muy avanzado. Los editores más populares son VSCode, Sublime Text y Atom, pero no es obligatorio usar alguno de estos para programar. Conoce más a fondo sobre [qué es un IDE](https://platzi.com/blog/que-es-ide-editor-de-texto/).

## Tipos de archivos y sus diferencias:

* **Archivos de Texto (.txt)**: Texto plano normal y sin nada especial. Lo vemos igual sin importar dónde lo abramos, ya sea con el bloc de notas o con editores de texto avanzados.
* **Archivos RTF (.rtf)**: Podemos guardar texto con diferentes tamaños, estilos y colores. Pero si lo abrimos desde un editor de código, vamos a ver que es mucho más complejo que solo el texto plano. Esto es porque debe guardar todos los estilos del texto y, para esto, usa un código especial un poco difícil de entender y muy diferente a los textos con estilos especiales al que estamos acostumbrados.
* **Archivos de Word (.docx)**: Podemos guardar imágenes y texto con diferentes tamaños, estilos o colores. Al abrirlo desde un editor de código podemos ver que es código binario, muy difícil de entender y muy diferente al texto al que estamos acostumbrados. Esto es porque Word está optimizado para entender este código especial y representarlo gráficamente.

Recuerda que debes habilitar la opción de ver la extensión de los archivos, de lo contrario, solo podrás ver su nombre. La forma de hacerlo en Windows es Vista > Mostrar u ocultar > Extensiones de nombre de archivo.

## Conceptos importantes de Git

* Bug: Error en el código
* Repository: Donde se almacena todo el proyecto, el cual puede vivir tanto en local como en remoto. El repositorio guarda un historial de versiones y, más importante, de la relación de cada versión con la anterior para que pueda hacerse el árbol de versiones con las diferentes ramas.
* Fork: Si en algún momento queremos contribuir al proyecto de otra persona, o si queremos utilizar el proyecto de otro como el punto de partida del nuestro. Esto se conoce como “fork”.
* Clone: Una vez se decide hacer un fork , hasta ese momento sólo existe en GitHub. Para poder trabajar en el proyecto, toca clonar el repositorio elegido al computador personal.
* Branch: Es una bifurcación del proyecto que se está realizando para anexar una nueva funcionalidad o corregir un bug.
* Master: Rama donde se almacena la última versión estable del proyecto que se está realizando. La rama master es la que está en producción en cada momento (o casi) y debería estar libre de bugs. Así, si esta rama está en producción, sirve como referente para hacer nuevas funcionalidades y/o arreglar bugs de última hora.
* Commit: consiste en subir cosas a la versión local del repositorio. De esta manera se puede trabajar en la rama de forma local sin tener que modificar ninguna versión en remoto ni tener que tener la última versión remota, cosa muy útil en grandes desarrollos trabajados por varias personas.
* Push: Consiste en enviar todo lo que se ha confirmado con un commit al repositorio remoto. Aquí es donde se une nuestro trabajo con el de los demás.
* Checkout: Acción de descargarse una rama del repositorio GIT local (sí, GIT tiene su propio repositorio en local para poder ir haciendo commits) o remoto.
* Fetch: Actualiza el repositorio local bajando datos del repositorio remoto al repositorio local sin actualizarlo, es decir, se guarda una copia del repositorio remoto en el local.
* Merge: La acción de merge es la continuación natural del fetch. El merge permite unir la copia del repositorio remoto con tu repositorio local, mezclando los diferentes códigos.
* Pull: Consiste en la unión del fetch y del merge, esto es, recoge la información del repositorio remoto y luego mezcla el trabajo en local con esta.
* Diff: Se utiliza para mostrar los cambios entre dos versiones del mismo archivo.

# Introducción a la terminal y línea de comandos

La línea de comandos nos permite interactuar con nuestro computador sin necesidad de utilizar una interfaz gráfica. Sin embargo, los computadores emplean distintos sistemas de archivos y manejan diferentes comandos, dependiendo del sistema operativo que utilicen.

Aprende más sobre: [qué es un comando.](https://platzi.com/clases/2292-terminal/37343-que-es-un-comando/)

## **Diferencias** entre la estructura de archivos de Windows, Mac o Linux

* La ruta principal en Windows es C:\, en UNIX es solo /.
* Windows no hace diferencia entre mayúsculas y minúsculas pero UNIX sí.

Recuerda que GitBash usa la ruta /c para dirigirse a C:\ (o /d para dirigirse a D:\) en Windows. Por lo tanto, la ruta del usuario con el que estás trabajando es /c/Users/Nombre de tu usuario

## **Comandos básicos en la terminal**

* **pwd**: Nos muestra la ruta de carpetas en la que te encuentras ahora mismo.
* **mkdir**: Nos permite crear carpetas (por ejemplo, mkdir Carpeta-Importante).
* **touch**: Nos permite crear archivos (por ejemplo, touch archivo.txt).
* **rm**: Nos permite borrar un archivo o carpeta (por ejemplo, rm archivo.txt). Mucho cuidado con este comando, puedes borrar todo tu disco duro.
* **cat**: Ver el contenido de un archivo (por ejemplo, cat nombre-archivo.txt).
* **ls**: Nos permite cambiar ver los archivos de la carpeta donde estamos ahora mismo. Podemos usar uno o más argumentos para ver más información sobre estos archivos (los argumentos pueden ser -- + el nombre del argumento o - + una sola letra o shortcut por cada argumento).  
  - ls -a: Mostrar todos los archivos, incluso los ocultos.  
  - ls -l: Ver todos los archivos como una lista.
* **cd**: Nos permite navegar entre carpetas.  
  - cd /: Ir a la ruta principal:  
  - cd o cd ~: Ir a la ruta de tu usuario  
  - cd carpeta/subcarpeta: Navegar a una ruta dentro de la carpeta donde estamos ahora mismo.  
  - cd .. (cd + dos puntos): Regresar una carpeta hacia atrás.  
  - Si quieres referirte al directorio en el que te encuentras ahora mismo puedes usar cd . (cd + un punto).
* **history**: Ver los últimos comandos que ejecutamos y un número especial con el que podemos repetir su ejecución.
* **! + número**: Ejecutar algún comando con el número que nos muestra el comando history (por ejemplo, !72).
* **clear**: Para limpiar la terminal. También podemos usar los atajos de teclado Ctrl + L o Command + L.

Todos estos comandos tiene una función de autocompletado, o sea, puedes escribir la primera parte y presionar la tecla Tab para que la terminal nos muestre todas las posibles carpetas o comandos que podemos ejecutar. Si presionas la tecla Arriba puedes ver el último comando que ejecutamos.

Recuerda que podemos descubrir todos los argumentos de un comando con el argumento --help (por ejemplo, cat --help).

# Crea un repositorio de Git y haz tu primer commit

Le indicaremos a [Git](https://platzi.com/clases/1557-git-github/20215-que-es-git/) que queremos crear un nuevo repositorio para utilizar su sistema de control de versiones. Solo debemos posicionarnos en la carpeta raíz de nuestro proyecto y ejecutar el comando:

git init

Recuerda que al ejecutar este comando (y de aquí en adelante) vamos a tener una nueva carpeta oculta llamada .git con toda la base de datos con cambios atómicos en nuestro proyecto.

Recuerda que Git está optimizado para trabajar en equipo, por lo tanto, debemos darle un poco de información sobre nosotros. No debemos hacerlo todas las veces que ejecutamos un comando, basta con ejecutar solo una sola vez los siguientes comandos con tu información:

git config --global user.email "tu@email.com"

git config --global user.name "Tu Nombre"

Existen muchas otras configuraciones de Git que puedes encontrar ejecutando el comando git config --list (o solo git config para ver una explicación más detallada).

Si quieres ver los archivos ocultos de una carpeta puedes habilitar la opción de Vista > Mostrar u ocultar > Elementos ocultos (en Windows) o ejecutar el comando ls -a.

## Comandos para iniciar tu repositorio con Git

* git init: para inicializar el repositorio git y el staged
* git add nombre\_del\_archivo.txt: enviar el archivo al staged
* git status: ver el estado, si se requiere agregar al starget o si se requiere commit
* git conf: para ver las posibles configuraciones
* git conf --list: para ver la lista de configuraciones hechas
* git conf --list --show-origin: para mostrar las configuraciones y sus rutas
* git rm --cached nombre\_del\_archivo.txt: para eliminar el archivo del staged(ram)
* git rm nombre\_del\_archivo.txt: para eliminar del repositorio

Si por algún motivo te equivocaste en el nombre o email que configuraste al principio, lo puedes modificar de la siguiente manera:  
git config --global --replace-all user.name “Aquí va tu nombre modificado”  
O si lo deseas eliminar y añadir uno nuevo  
git config --global --unset-all user.name :Elimina el nombre del usuario  
git config --global --add user.name “Aquí va tu nombre”