**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).

**RECURSOS**

**MARCADORES**

Le indicaremos a 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”

El comando **git show** nos muestra los cambios que han existido sobre un archivo y es muy útil para detectar cuándo se produjeron ciertos cambios, qué se rompió y cómo lo podemos solucionar. Pero podemos ser más detallados.

Si queremos ver la diferencia entre una versión y otra, no necesariamente todos los cambios desde la creación del archivo, podemos usar el comando **git diff commitA commitB.**

Recuerda que puedes obtener el ID de tus **commits** con el comando **git log**.

Comandos para analizar cambios en GIT

• **git init:** inicializar el repositorio

• **git add nombre\_de\_archivo.extensión**: agregar el archivo al repositorio

• **git commit -m “Mensaje”:** Agregamos los cambios para el repositorio

• **git add:** Agregar los cambios de la carpeta en la que nos encontramos agregar todo

• **git status:** visualizar cambios

• **git log nombre\_de\_archivos.extensión**: histórico de cambios con detalles

• **git push:** envía a otro repositorio remoto lo que estamos haciendo

• **git pull:** traer repositorio remoto

• **ls:** listado de carpetas en donde me encuentro. Es decir, como emplear dir en windows.

• **pwd**: ubicación actual

• **mkdir:** make directory nueva carpeta

• **touch archivo.extensión**: crear archivo vacío

• **cat archivo.extensión**: muestra el contenido del archivo

• **history:** historial de comandos utilizados durante esa sesión

• **rm archivo.extensión**: Eliminación de archivo

• **comando --help:** ayuda sobre el comando

• **git checkout:** traer cambios realizados

• **git rm --cached archivo.extensión:** se utiliza para devolver el archivo que se tiene en ram. Cuando escribimos git add, lo devuelve a estado natural mientras está en staging.

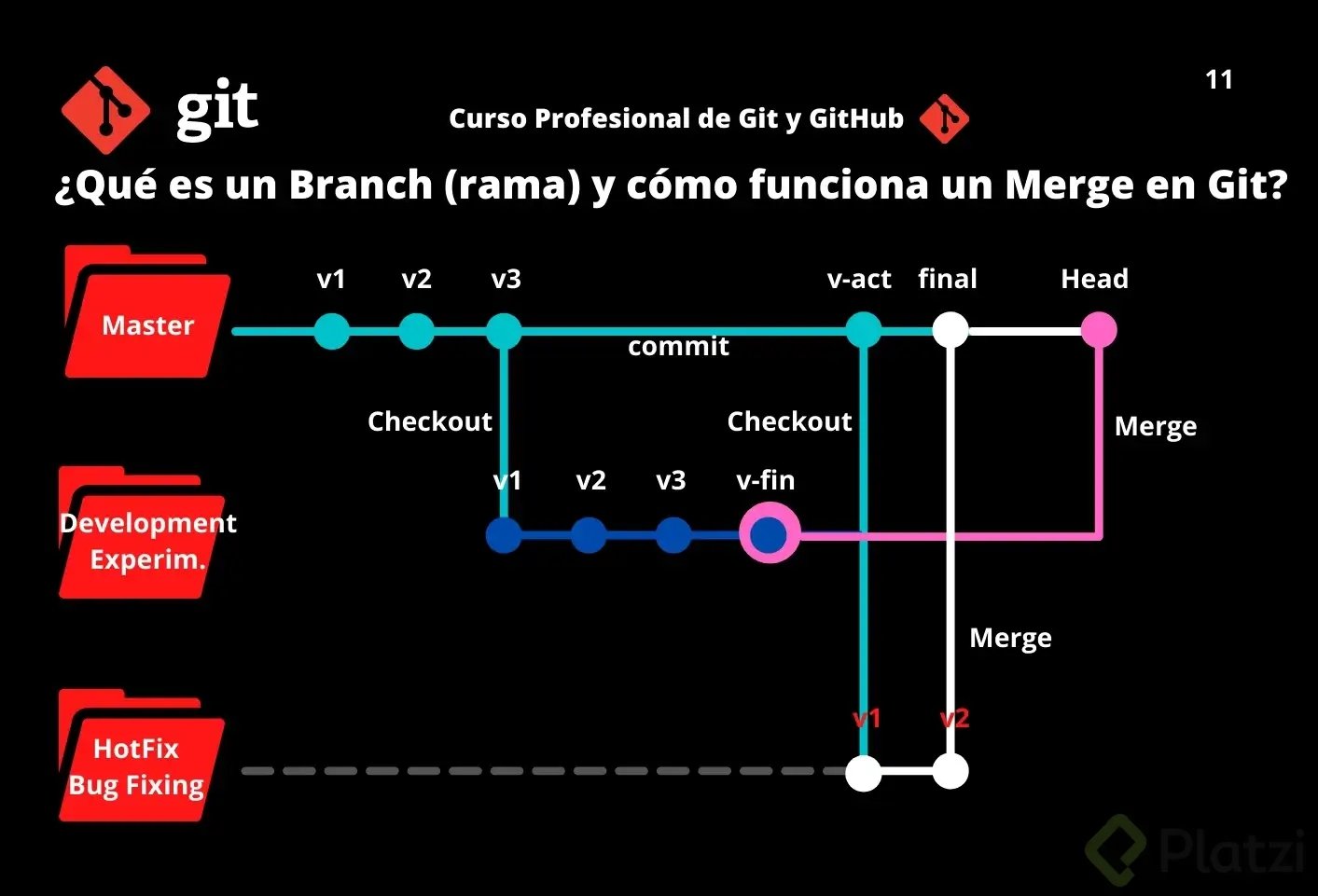
• **git config --list**: muestra la lista de configuración de git

• **git config --list --show-origin:** rutas de acceso a la configuración de git

• **git log archivo.extensión:** muestra la historia del archivo

Una rama o branch es una versión del código del proyecto sobre el que estás trabajando. Estas ramas ayudan a mantener el orden en el control de versiones y manipular el código de forma segura.

En otras palabras, un branch o rama en Git es una rama que proviene de otra. Imagina un árbol, que tiene una rama gruesa, y otra más fina, en la rama más gruesa tenemos los commits principales y en la rama fina tenemos otros commits que pueden ser de hotfix, devlopment entre otros.ㅤ



Clases de branches o ramas en Git

Estas son las ramas base de un proyecto en Git:

**1. Rama main (Master)**

Por defecto, el proyecto se crea en una rama llamada Main (anteriormente conocida como Master). Cada vez que añades código y guardas los cambios, estás haciendo un commit, que es añadir el nuevo código a una rama. Esto genera nuevas versiones de esta rama o branch, hasta llegar a la versión actual de la rama Main.

**2. Rama development**

Cuando decides hacer experimentos, puedes generar ramas experimentales (usualmente llamadas development), que están basadas en alguna rama main, pero sobre las cuales puedes hacer cambios a tu gusto sin necesidad de afectar directamente al código principal.

**3. Rama hotfix**

En otros casos, si encuentras un bug o error de código en la rama Main (que afecta al proyecto en producción), tendrás que crear una nueva rama (que usualmente se llaman bug fixing o hot fix) para hacer los arreglos necesarios. Cuando los cambios estén listos, los tendrás que fusionar con la rama Main para que los cambios sean aplicados. Para esto, se usa un comando llamado Merge, que mezcla los cambios de la rama que originaste a la rama Main.

Todos los commits se aplican sobre una rama. Por defecto, siempre empezamos en la rama Main (pero puedes cambiarle el nombre si no te gusta) y generamos nuevas ramas, a partir de esta, para crear flujos de trabajo independientes.

Cómo crear un branch o rama en Git

El comando git branch permite crear una rama nueva. Si quieres empezar a trabajar en una nueva función, puedes crear una rama nueva a partir de la rama master con git branch new\_branch. Una vez creada, puedes usar git checkout new\_branch para cambiar a esa rama.

Recuerda que todas tus versiones salen de la rama principal o Master y de allí puedes tomar una versión específica para crear otra rama de versiones.

**Cómo hacer merge**

Producir una nueva rama se conoce como Checkout. Unir dos ramas lo conocemos como Merge.

Cuando haces merge de estas ramas con el código principal, su código se fusiona originando una nueva versión de la rama master (o main) que ya tiene todos los cambios que aplicaste en tus experimentos o arreglos de errores.

Podemos generar todas las ramas y commits que queramos. De hecho, podemos aprovechar el registro de cambios de Git para producir ramas, traer versiones viejas del código, arreglarlas y combinarlas de nuevo para mejorar el proyecto.

Solo ten en cuenta que combinar estas ramas (hacer “merge”) puede generar conflictos. Algunos archivos pueden ser diferentes en ambas ramas. Git es muy inteligente y puede intentar unir estos cambios automáticamente, pero no siempre funciona. En algunos casos, somos nosotros los que debemos resolver estos conflictos a mano.

El comando git checkout + ID del commit nos permite viajar en el tiempo. Podemos volver a cualquier versión anterior de un archivo específico o incluso del proyecto entero. Esta también es la forma de crear ramas y movernos entre ellas.

También hay una forma de hacerlo un poco más “ruda”: usando el comando git reset. En este caso, no solo “volvemos en el tiempo”, sino que borramos los cambios que hicimos después de este commit.

Hay dos formas de usar git reset: con el argumento --hard, borrando toda la información que tengamos en el área de staging (y perdiendo todo para siempre). O, un poco más seguro, con el argumento --soft, que mantiene allí los archivos del área de staging para que podamos aplicar nuestros últimos cambios pero desde un commit anterior.

**Repasa qué es branch**

**Cómo usar Git Reset**

Para volver a commits previos, borrando los cambios realizados desde ese commit, podemos utilizar:

**git reset --soft [SHA 1]:** elimina los cambios hasta el staging area

**git reset --mixed [SHA 1**]: elimina los cambios hasta el working area

**git reset --hard [SHA 1]**: regresa hasta el commit del [SHA-1]

**Donde el SHA-1 es el identificador del commit**

**Git reset y git rm** son comandos con utilidades muy diferentes, pero se pueden confundir muy fácilmente.

**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.

**Recuerda que git rm** no puede usarse así nomás. Debemos usar uno de los **flags** para indicarle a Git cómo eliminar los archivos que ya no necesitamos en la última versión del proyecto:

**git rm --cached:** Elimina los archivos de nuestro repositorio local y del área de **staging**, pero los mantiene en nuestro disco duro. Básicamente le dice a Git que deje de **trackear** el historial de cambios de estos archivos, por lo que pasaran a un estado **untracked.**

**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 usar comandos más avanzados).

**git reset**

Este comando nos ayuda a volver en el tiempo. Pero no como git checkout que nos deja ir, mirar, pasear y volver. Con git reset volvemos al pasado sin la posibilidad de volver al futuro. Borramos la historia y la debemos sobreescribir. No hay vuelta atrás.

Este comando es muy peligroso y debemos emplearlo solo en caso de emergencia. Recuerda que debemos usar alguna de estas dos opciones:

**Hay dos formas de utilizar git reset: con el argumento --hard,** borrando toda la información que tengamos en el área destaging (y perdiendo todo para siempre). O, un poco más seguro, **con el argumento --soft,** que mantiene allí los archivos del área de **staging** para que podamos aplicar nuestros últimos cambios pero desde un **commit anterior.**

**git reset --soft:** Borramos todo el historial y los registros de Git pero guardamos los cambios que tengamos en **Staging,** así podemos aplicar las últimas actualizaciones a un nuevo commit.

**git reset --hard:** Borra todo. Todo todito, absolutamente todo. Toda la información de los commits y del área de staging se borra del historial.

¡Pero todavía falta algo!

**git reset HEAD:** Este es el comando para sacar archivos del área de **staging.** No para borrarlos ni nada de eso, solo para que los últimos cambios de estos archivos no se envíen al último commit, a menos que cambiemos de opinión y los incluyamos de nuevo en **staging con git add, por supuesto.**

**La relevancia de estos comandos**

**Imagina el siguiente caso:**

Hacemos cambios en los archivos de un proyecto para una nueva actualización. Todos los archivos con cambios se mueven al área de staging con el comando git add. Pero te das cuenta de que uno de esos archivos no está listo todavía. Actualizaste el archivo, pero ese cambio no debe ir en el próximo commit por ahora.

**¿Qué podemos hacer?**

Bueno, todos los cambios están en el área de Staging, incluido el archivo con los cambios que no están listos. Esto significa que debemos sacar ese archivo de Staging para poder hacer commit de todos los demás.

¡Al usar git rm lo que haremos será eliminar este archivo completamente de git! Todavía tendremos el historial de cambios de este archivo, con la eliminación del archivo como su última actualización. Recuerda que en este caso no buscábamos eliminar un archivo, solo dejarlo como estaba y actualizarlo después, no en este commit.

En cambio, si usamos git reset HEAD, lo único que haremos será mover estos cambios de Staging a Unstaged. Seguiremos teniendo los últimos cambios del archivo, el repositorio mantendrá el archivo (no con sus últimos cambios, pero sí con los últimos en los que hicimos commit) y no habremos perdido nada.

Conclusión: Lo mejor que puedes hacer para salvar tu puesto y evitar un incendio en tu trabajo es conocer muy bien la diferencia y los riesgos de todos los comandos de Git.

­­--Algunos comandos que pueden ayudar cuando colaboren con proyectos muy grandes de github:

git log --oneline - Te muestra el id commit y el título del commit.

git log --decorate- Te muestra donde se encuentra el head point en el log.

git log --stat - Explica el número de líneas que se cambiaron brevemente.

git log -p- Explica el número de líneas que se cambiaron y te muestra que se cambió en el contenido.

git shortlog - Indica que commits ha realizado un usuario, mostrando el usuario y el titulo de sus commits.

git log --graph --oneline --decorate y

git log --pretty=format:"%cn hizo un commit %h el dia %cd" - Muestra mensajes personalizados de los commits.

git log -3 - Limitamos el número de commits.

git log --after=“2018-1-2” ,

git log --after=“today” y

git log --after=“2018-1-2” --before=“today” - Commits para localizar por fechas.

git log --author=“Name Author” - Commits realizados por autor que cumplan exactamente con el nombre.

git log --grep=“INVIE” - Busca los commits que cumplan tal cual está escrito entre las comillas.

git log --grep=“INVIE” –i- Busca los commits que cumplan sin importar mayúsculas o minúsculas.

git log – index.html- Busca los commits en un archivo en específico.

git log -S “Por contenido”- Buscar los commits con el contenido dentro del archivo.

git log > log.txt - guardar los logs en un archivo txt