# clase 3: Instalando Git y GitBash en Windows

# clase 4: Instalando Git en OSX

# clase 5: Instalando Git en Linux

# clase 6: Editores de código, archivos binarios y de texto plano

https://atom.io/

https://www.sublimetext.com/ es de paga

https://code.visualstudio.com/ el que usaremos

# clase 7: Introducción a la terminal y línea de comandos

# clase 8: Crea un repositorio de Git y haz tu primer commit

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”

# clase 9: Analizar cambios en los archivos de tu proyecto con Git

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

# clase10: ¿Qué es el staging y los repositorios? Ciclo básico de trabajo en Git

Para iniciar un repositorio, o sea, activar el sistema de control de versiones de Git en tu proyecto, solo debes ejecutar el comando git init.

Este comando se encargará de dos cosas: primero, crear una carpeta .git, donde se guardará toda la base de datos con cambios atómicos de nuestro proyecto; y segundo, crear un área que conocemos como Staging (en ram), que guardará temporalmente nuestros archivos (cuando ejecutemos un comando especial para eso) y nos permitirá, más adelante, guardar estos cambios en el repositorio (también con un comando especial).

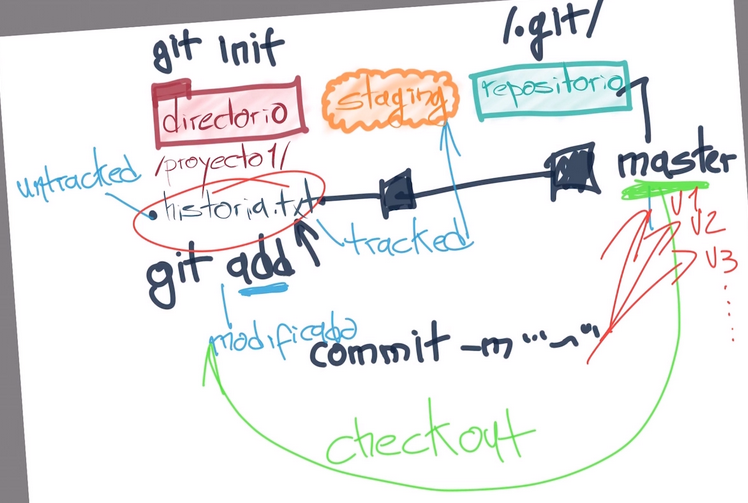
**Ciclo de vida o estados de los archivos en Git**:

Cuando trabajamos con Git nuestros archivos pueden vivir y moverse entre 4 diferentes estados (cuando trabajamos con repositorios remotos pueden ser más estados, pero lo estudiaremos más adelante):

* **Archivos Tracked**: son los archivos que viven dentro de Git, no tienen cambios pendientes y sus últimas actualizaciones han sido guardadas en el repositorio gracias a los comandos git add y git commit.
* **Archivos Staged**: son archivos en Staging. Viven dentro de Git y hay registro de ellos porque han sido afectados por el comando git add, aunque no sus últimos cambios. Git ya sabe de la existencia de estos últimos cambios, pero todavía no han sido guardados definitivamente en el repositorio porque falta ejecutar el comando git commit.
* **Archivos Unstaged**: entiéndelos como archivos *“Tracked pero Unstaged”*. Son archivos que viven dentro de Git pero no han sido afectados por el comando git add ni mucho menos por git commit. Git tiene un registro de estos archivos, pero está desactualizado, sus últimas versiones solo están guardadas en el disco duro.
* **Archivos Untracked**: son archivos que NO viven dentro de Git, solo en el disco duro. Nunca han sido afectados por git add, así que Git no tiene registros de su existencia.  
  Recuerda que hay un caso muy raro donde los archivos tienen dos estados al mismo tiempo: staged y untracked. Esto pasa cuando guardas los cambios de un archivo en el área de Staging (con el comando git add), pero antes de hacer commit para guardar los cambios en el repositorio haces nuevos cambios que todavía no han sido guardados en el área de Staging (en realidad, todo sigue funcionando igual pero es un poco divertido).

**Comandos para mover archivos entre los estados de Git**:

* **git status**: nos permite ver el estado de todos nuestros archivos y carpetas.
* **git add**: nos ayuda a mover archivos del Untracked o Unstaged al estado Staged. Podemos usar git nombre-del-archivo-o-carpeta para añadir archivos y carpetas individuales o git add -A para mover todos los archivos de nuestro proyecto (tanto Untrackeds como unstageds).
* **git reset HEAD**: nos ayuda a sacar archivos del estado Staged para devolverlos a su estado anterior. Si los archivos venían de Unstaged, vuelven allí. Y lo mismo se venían de Untracked.
* **git commit**: nos ayuda a mover archivos de Unstaged a Tracked. Esta es una ocasión especial, los archivos han sido guardados o actualizados en el repositorio. Git nos pedirá que dejemos un mensaje para recordar los cambios que hicimos y podemos usar el argumento -m para escribirlo (git commit -m "mensaje").
* **git rm**: este comando necesita alguno de los siguientes argumentos para poder ejecutarse correctamente:  
  - git rm --cached: Mueve los archivos que le indiquemos al estado Untracked.  
  - git rm --force: Elimina los archivos de Git y del disco duro. Git guarda el registro de la existencia de los archivos, por lo que podremos recuperarlos si es necesario (pero debemos usar comandos más avanzados).\
* checkout: para traer los datos del repositorio a mi directorio local.



# ¿Qué es el staging?

El staging es el lugar donde se guardan temporalmente los cambios, para luego ser llevados definitivamente al repositorio. El repositorio es el lugar donde se guardan todos los registros de los cambios realizados a los archivos.

Para iniciar un repositorio, o sea, activar el sistema de control de versiones de Git en tu proyecto, solo debes ejecutar el comando git init.

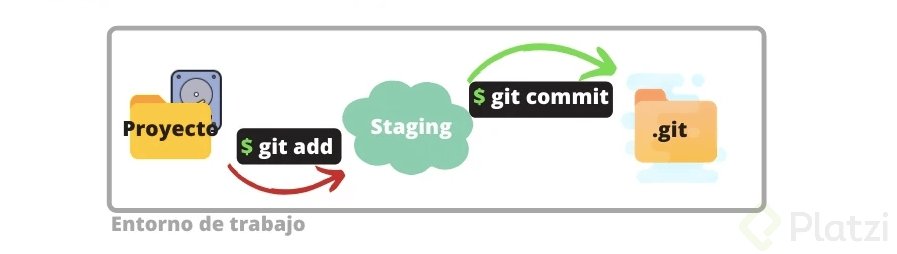
## ¿Qué es el área de staging?

El área de staging se puede ver como un limbo donde nuestros archivos están por ser enviados al repositorio o ser regresados a la carpeta del proyecto.

## ¿Qué es git init?

git inites el comando que activa git en nuestro proyecto creando un espacio en memoria RAM llamado staging y una carpeta .git.

Este comando se encargará de dos cosas: primero, crear una carpeta .git, donde se guardará toda la base de datos con cambios atómicos de nuestro proyecto; segundo, crear un área que conocemos como **staging**, que guardará temporalmente nuestros archivos (cuando ejecutemos un comando especial para eso) y nos permitirá, más adelante, guardar estos cambios en el repositorio (también con un comando especial).



## Cómo funciona el staging y el repositorio: ciclo básico de trabajo en git:

El flujo de trabajo básico en git es algo así:

1. Modificas una serie de archivos en tu directorio de trabajo.
2. Preparas los archivos, añadiéndolos a tu área de preparación (staging).
3. Confirmas los cambios (commit), lo que toma los archivos tal y como están en el área de preparación y almacena esa copia instantánea de manera permanente en tu directorio de git.

Veamos a detalle las 3 secciones principales que tiene un proyecto en git.

### Working directory

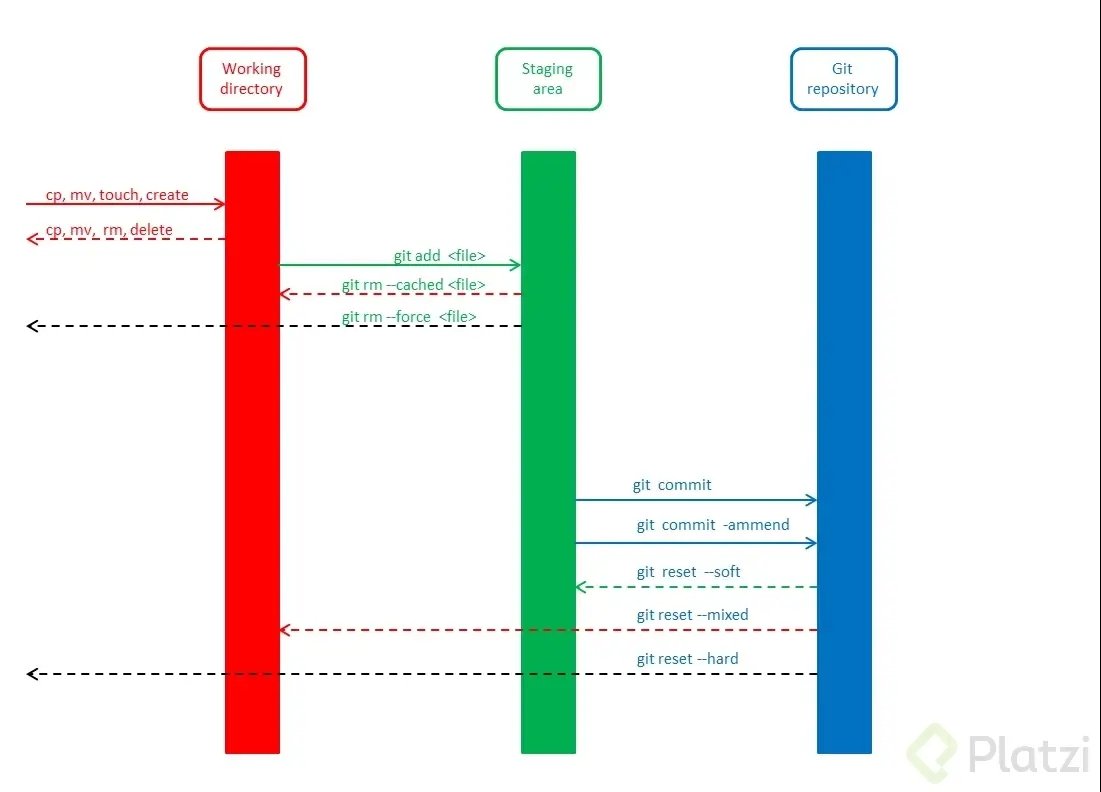
El working directory es una copia de una versión del proyecto. Estos archivos se sacan de la base de datos comprimida en el directorio de git y se colocan en el disco para que los puedas usar o modificar.

### Staging area

Es un área que almacena información acerca de lo que va a ir en tu próxima confirmación. A veces se le denomina índice (index).

### .git directory (repository)

En el repository se almacenan los metadatos y la base de datos de los objetos para tu proyecto. Es la parte más importante de git (carpeta .git) y es lo que se copia cuando clonas un repositorio desde otra computadora.



## ****Ciclo de vida o estados de los archivos en git****

Cuando trabajamos con git, nuestros archivos pueden vivir y moverse entre 4 diferentes estados (cuando trabajamos con repositorios remotos pueden ser más estados, pero lo estudiaremos más adelante):

### **Archivos tracked**

Son los archivos que viven dentro de git, no tienen cambios pendientes y sus últimas actualizaciones han sido guardadas en el repositorio gracias a los comandos git add y git commit.

### **Archivos staged**

Son archivos en staging. Viven dentro de git y hay registro de ellos porque han sido afectados por el comando git add, aunque no sus últimos cambios. Git ya sabe de la existencia de estos últimos cambios, pero todavía no han sido guardados definitivamente en el repositorio porque falta ejecutar el comando git commit.

### **Archivos unstaged**

Entiéndelos como archivos “tracked pero unstaged”. Son archivos que viven dentro de git pero no han sido afectados por el comando git add ni mucho menos por git commit. Git tiene un registro de estos archivos, pero está desactualizado, sus últimas versiones solo están guardadas en el disco duro.

### **Archivos untracked**

Son archivos que NO viven dentro de git, solo en el disco duro. Nunca han sido afectados por git add, así que git no tiene registros de su existencia.

Recuerda que hay un caso muy raro donde los archivos tienen dos estados al mismo tiempo: staged y untracked. Esto pasa cuando guardas los cambios de un archivo en el área de staging (con el comando git add), pero antes de hacer commit para guardar los cambios en el repositorio haces nuevos cambios que todavía no han sido guardados en el área de staging.

## ****Comandos para mover archivos entre los estados de Git****

Estos son los comandos más importantes que debes conocer:

### Git status

**git status** nos permite ver el estado de todos nuestros archivos y carpetas.

### Git add

**git add** nos ayuda a mover archivos del untracked o unstaged al estado staged. Podemos usar git nombre-del-archivo-o-carpeta para añadir archivos y carpetas individuales o git add -A para mover todos los archivos de nuestro proyecto (tanto untrackeds como unstageds).

### G**it reset HEAD**

Nos ayuda a sacar archivos del estado staged para devolverlos a su estado anterior. Si los archivos venían de unstaged, vuelven allí. Y lo mismo se venían de untracked.

### G**it commit**

Nos ayuda a mover archivos de unstaged a tracked. Esta es una ocasión especial, los archivos han sido guardados o actualizados en el repositorio. Git nos pedirá que dejemos un mensaje para recordar los cambios que hicimos y podemos usar el argumento m para escribirlo (git commit -m "mensaje").

### G**it rm**

Este comando necesita alguno de los siguientes argumentos para poder ejecutarse correctamente:

* git rm --cached: mueve los archivos que le indiquemos al estado untracked.
* git rm --force: elimina los archivos de git y del disco duro. Git guarda el registro de la existencia de los archivos, por lo que podremos recuperarlos si es necesario (pero debemos usar comandos más avanzados).

# clase11: ¿Qué es un Branch (rama) y cómo funciona un Merge en Git?

Git es una base de datos muy precisa con todos los cambios y crecimiento que ha tenido nuestro proyecto. Los commits son la única forma de tener un registro de los cambios. Pero las ramas amplifican mucho más el potencial de Git.

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

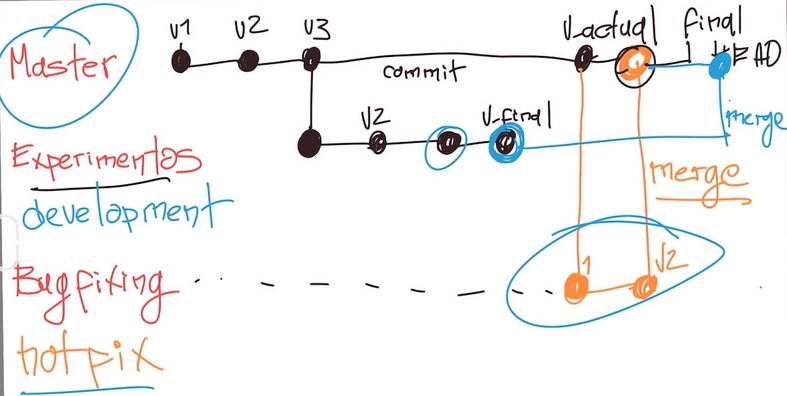
Crear una nueva rama se trata de copiar un commit (de cualquier rama), pasarlo a otro lado (a otra rama) y continuar el trabajo de una parte específica de nuestro proyecto sin afectar el flujo de trabajo principal (que continúa en la rama master o la rama principal).

Los equipos de desarrollo tienen un estándar: Todo lo que esté en la rama master va a producción, las nuevas features, características y experimentos van en una rama “development” (para unirse a master cuando estén definitivamente listas) y los issues o errores se solucionan en una rama “hotfix” para unirse a master tan pronto como sea posible.

Crear una nueva rama lo conocemos como **Checkout**. Unir dos ramas lo conocemos como **Merge**.

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

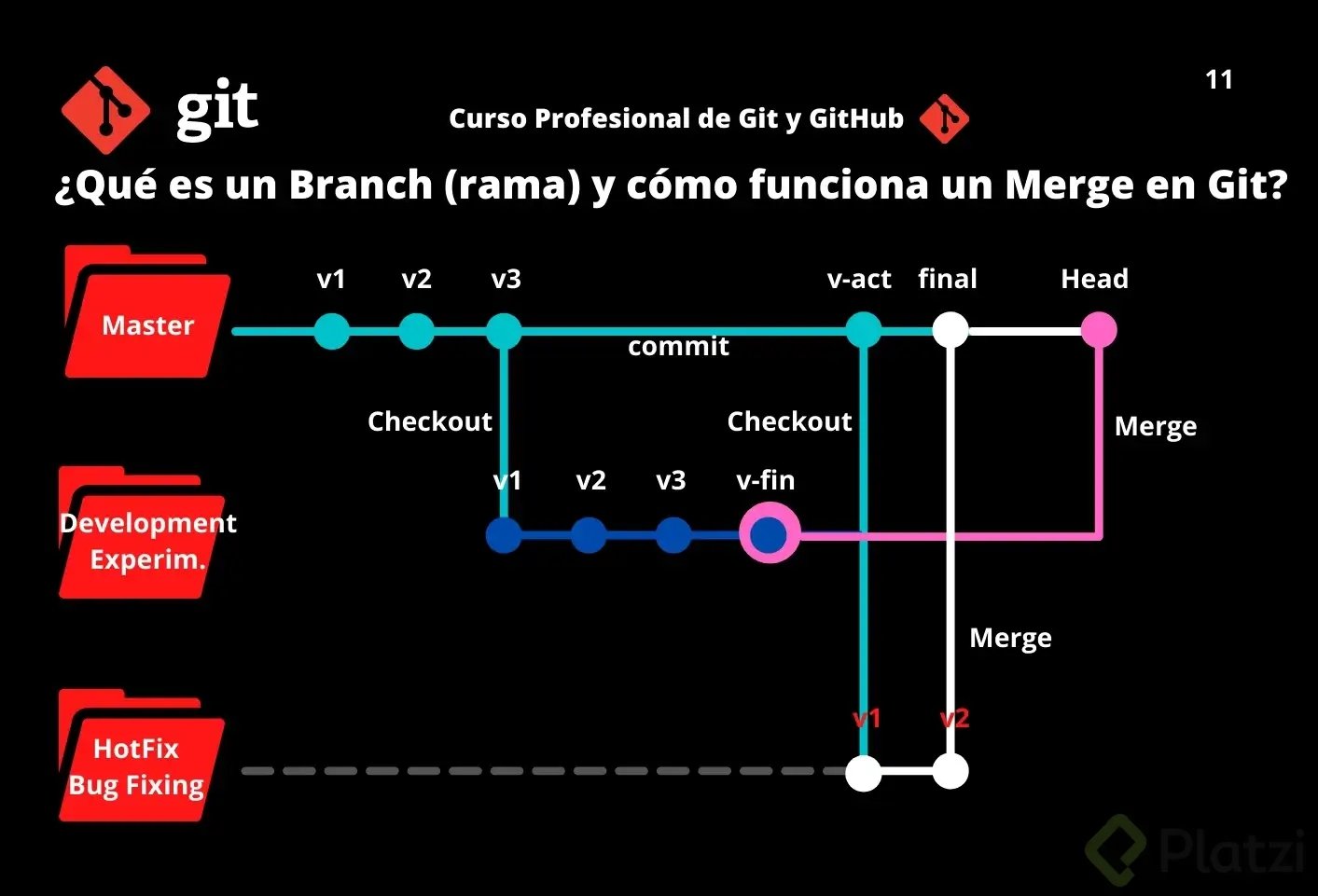
Solo ten en cuenta que combinar estas ramas (sí, hacer “merge”) puede generar conflictos. Algunos archivos pueden ser diferentes en ambas ramas. Git es muy inteligente y puede intentar unir estos cambios automáticamente, pero no siempre funciona. En algunos casos, somos nosotros los que debemos resolver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



# ¿Qué es branch (rama) y cómo funciona un Merge en Git?

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”](https://platzi.com/clases/1557-git-github/19939-funcion-de-ramas-con-git-mer-7/)) 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.

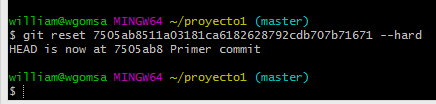
# clase 12: Volver en el tiempo en nuestro repositorio utilizando reset y checkout

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.

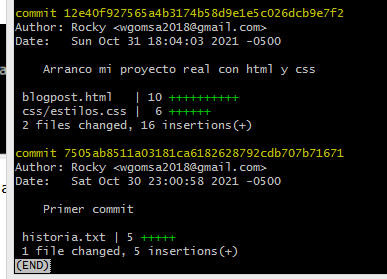
También hay una forma de hacerlo un poco más “ruda”: usando el comando git reset. En este caso, no solo “volvemos en el tiempo”, sino que borramos los cambios que hicimos después de este commit.

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.

tener cuidado de usar el reset porque podemos perder todos nuestros cambios



git log --stat : para ver cambios especificos y en que archivos. nos movemos con las flechas arriba o abajo y para salir con la letra q.



si queremos situarnos en alguna linea de tiempo, en este caso en el primer commit pero de uno de los archivos.

git checkout id\_commit nombre\_archivo

git checkout 7505ab8511a03181ca6182628792cdb707b71671 historia.txt

y para volver al state actual

git checkout rama\_actual nombre\_archivo

git checkout master historia.txt

si modificamos ese archivo y le comiteamos ya se habra cambiado y estaremos en el status actual. lo vemos con git log

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.

## 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

# clase 13: Git reset vs. Git rm

Git reset y git rm son comandos con utilidades muy diferentes, pero aún así se confunden 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 usarlo solo en caso de emergencia. Recuerda que debemos usar alguna de estas dos opciones:

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.

* 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.

# ¿Por qué esto es importante?

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.

# clase 14: Flujo de trabajo básico con un repositorio remoto

revisar el documento adjunto : clase14 flujo trabajo remoto.pdf

para traer los cambios de un repositorio remoto a area de trabajo local se usa:

git fetch : de repositorio remoto a repositorio local

y con git merge: realizamos el merge de mi repositorio local a directorio trabajo

pero con git pull reemplaza a estos 2 comandos

Cuando empiezas a trabajar en un entorno local, el proyecto vive únicamente en tu computadora. Esto significa que no hay forma de que otros miembros del equipo trabajen en él.

Para solucionar esto, utilizamos los **servidores remotos**: un nuevo estado que deben seguir nuestros archivos para conectarse y trabajar con equipos de cualquier parte del mundo.

Estos servidores remotos pueden estar alojados en GitHub, GitLab, BitBucket, entre otros. Lo que van a hacer es guardar el mismo repositorio que tienes en tu computadora y darnos una URL con la que todos podremos acceder a los archivos del proyecto. Así, el equipo podrá descargarlos, hacer cambios y volverlos a enviar al servidor remoto para que otras personas vean los cambios, comparen sus versiones y creen nuevas propuestas para el proyecto.

Esto significa que debes aprender algunos nuevos comandos

## Comandos para trabajo remoto con GIT

* **git clone url\_del\_servidor\_remoto**: Nos permite descargar los archivos de la última versión de la rama principal y todo el historial de cambios en la carpeta .git.
* **git push**: Luego de hacer git add y git commit debemos ejecutar este comando para mandar los cambios al servidor remoto.
* **git fetch**: Lo usamos para traer actualizaciones del servidor remoto y guardarlas en nuestro repositorio local (en caso de que hayan, por supuesto).
* **git merge**: También usamos el comando git merge con servidores remotos. Lo necesitamos para combinar los últimos cambios del servidor remoto y nuestro directorio de trabajo.
* **git pull**: Básicamente, git fetch y git merge al mismo tiempo.

Adicionalmente, tenemos otros comandos que nos sirven para trabajar en proyectos muy grandes:

* **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 título 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 hechos 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

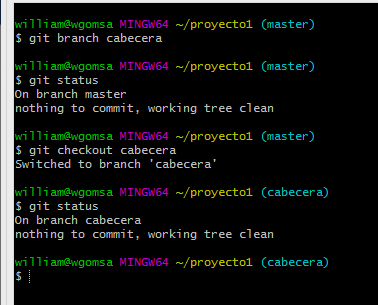
# clase 15: Introducción a las ramas o branches de Git

Las ramas son la forma de hacer cambios en nuestro proyecto sin afectar el flujo de trabajo de la rama principal. Esto porque queremos trabajar una parte muy específica de la aplicación o simplemente experimentar.

La cabecera o HEAD representan la rama y el commit de esa rama donde estamos trabajando. Por defecto, esta cabecera aparecerá en el último commit de nuestra rama principal. Pero podemos cambiarlo al crear una rama (git branch rama, git checkout -b rama) o movernos en el tiempo a cualquier otro commit de cualquier otra rama con los comandos (git reset id-commit, git checkout rama-o-id-commit).

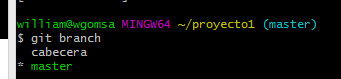
vamos a crear una rama cabecera: git branch cabecera, si le damos status seguimos en la master

y para movernos a esa rama : git checkout cabecera



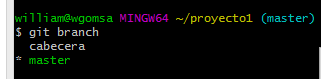
ver las ramas y en cual estoy

git branch



# clase 16: Fusión de ramas con Git merge

git branch : permite ver las ramas que tengo y en que rama estoy



para hacer merge de rama1 a rama2, debo situarme en la rama2 y colocar git merge rama1

en este ejemplo voy hacer el merge de la rama cabecera a la master. me ubico en la master y luego hago el merge desde cabecera

git merge cebecera

saldra para poner un nombre, porque el merge en si es un commit

El comando git merge nos permite crear un nuevo commit con la combinación de dos ramas o [branches](https://platzi.com/clases/1557-git-github/19947-que-es-un-branch-rama-y-como-funciona-un-merge-en-/) (la rama donde nos encontramos cuando ejecutamos el comando y la rama que indiquemos después del comando).

## Cómo usar Git merge

En este ejemplo, vamos a crear un nuevo commit en la rama master combinando los cambios de una rama llamada cabecera:

git checkout **master**

**git** merge cabecera

Otra opción es crear un nuevo commit en la rama cabecera combinando los cambios de cualquier otra rama:

git checkout cabecera

git **merge** cualquier-otra-rama

Asombroso, ¿verdad? Es como si Git tuviera superpoderes para saber qué cambios queremos conservar de una rama y qué otros de la otra. El problema es que no siempre puede adivinar, sobre todo en algunos casos donde dos ramas tienen actualizaciones diferentes en ciertas líneas en los archivos. Esto lo conocemos como un **conflicto**.

Recuerda que al ejecutar el comando git checkout para cambiar de rama o commit puedes perder el trabajo que no hayas guardado. **Guarda siempre tus cambios antes de hacer git checkout**.

## Comandos básicos de GitHub

* git init: crear un repositorio.
* git add: agregar un archivo a staging.
* git commit -m “mensaje”: guardar el archivo en git con un mensaje.
* git branch: crear una nueva rama.
* git checkout: moverse entre ramas.
* git push: mandar cambios a un servidor remoto.
* git fetch: traer actualizaciones del servidor remoto y guardarlas en nuestro repositorio local.
* git merge: tiene dos usos. Uno es la fusión de ramas, funcionando como un commit en la rama actual, trayendo la rama indicada. Su otro uso es guardar los cambios de un servidor remoto en nuestro directorio.
* git pull: fetch y merge al mismo tiempo.

## Comandos para corrección en GitHub

* git checkout “codigo de version” “nombre del archivo”: volver a la última versión de la que se ha hecho commit.
* git reset: vuelve al pasado sin posibilidad de volver al futuro, se debe usar con especificaciones.
* git reset --soft: vuelve a la versión en el repositorio, pero guarda los cambios en staging. Así, podemos aplicar actualizaciones a un nuevo commit.
* git reset --hard: todo vuelve a su versión anterior
* git reset HEAD: saca los cambios de staging, pero no los borra. Es lo opuesto a git add.
* git rm: elimina los archivos, pero no su historial. Si queremos recuperar algo, solo hay que regresar. se utiliza así:  
  git rm --cached elimina los archivos en staging pero los mantiene en el disco duro.  
  git rm --force elimina los archivos de git y del disco duro.

## Comandos para revisión y comparación en GitHub

* git status: estado de archivos en el repositorio.
* git log: historia entera del archivo.
* git log --stat: cambios específicos en el archivo a partir de un commit.
* git show: cambios históricos y específicos hechos en un archivo.
* git diff “codigo de version 1” “codigo de version 2”: comparar cambios entre versiones.
* git diff: comparar directorio con staging.

# clase17: Resolución de conflictos al hacer un merge

**Git nunca borra nada** a menos que nosotros se lo indiquemos. Cuando usamos los comandos git merge o git checkout estamos cambiando de rama o creando un nuevo commit, no borrando ramas ni commits (recuerda que puedes borrar commits con git reset y ramas con git branch -d).

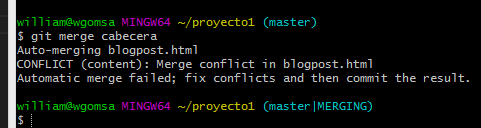
Git es muy inteligente y puede resolver algunos conflictos automáticamente: cambios, nuevas líneas, entre otros. Pero algunas veces no sabe cómo resolver estas diferencias, por ejemplo, cuando dos ramas diferentes hacen cambios distintos a una misma línea.

Esto lo conocemos como **conflicto** y lo podemos resolver manualmente, solo debemos hacer el merge, ir a nuestro editor de código y elegir si queremos quedarnos con alguna de estas dos versiones o algo diferente. Algunos editores de código como VSCode nos ayudan a resolver estos conflictos sin necesidad de borrar o escribir líneas de texto, basta con hundir un botón y guardar el archivo.

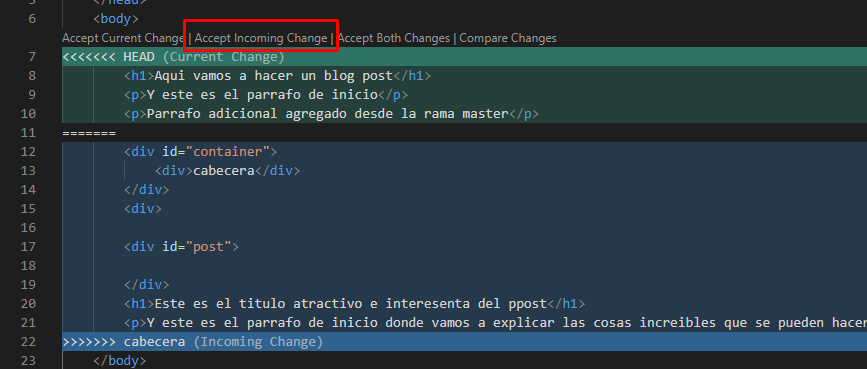
Recuerda que siempre debemos crear un nuevo commit para aplicar los cambios del merge. Si Git puede resolver el conflicto hará commit automáticamente. Pero, en caso de no pueda resolverlo, debemos solucionarlo y hacer el commit.

Los archivos con conflictos por el comando git merge entran en un nuevo estado que conocemos como **Unmerged**. Funcionan muy parecido a los archivos en estado Unstaged, algo así como un estado intermedio entre Untracked y Unstaged, solo debemos ejecutar git add para pasarlos al área de staging y git commit para aplicar los cambios en el repositorio.

tengo conflicto



por lo general se mantienen los cambios entrantes.



luego de corregido le damos add y commit

git add .

git commit -m "Se soluciona el conflicto de las ramas al fusionar"

pero esos 2 comandos se puede hacer en uno solo:

git commit -am "Se soluciona el conflicto de las ramas al fusionar"

## Cómo revertir un merge

Si nos hemos equivocado y queremos cancelar el merge, debemos usar el siguiente comando:

git merge --abort

## Conflictos en repositorios remotos

Al trabajar con otras personas, es necesario utilizar un repositorio remoto.  
­  
-Para copiar el repositorio remoto al directorio de trabajo local, se utiliza el comando git clone <url>, y para enviar cambios al repositorio remoto se utiliza git push.  
-Para actualizar el repositorio local se hace uso del comando git fetch, luego se debe fusionar los datos traídos con los locales usando git merge.

* Para traer los datos y fusionarlos a la vez, en un solo comando, se usa git pull.  
  ­- Para crear commits rápidamente, fusionando git add y git commit -m "", usamos git commit -am "".  
  ­- Para generar nuevas ramas, hay que posicionarse sobre la rama que se desea copiar y utilizar el comando git branch <nombre>.
* Para saltar entre ramas, se usa el comando git checkout <branch>  
  ­- Una vez realizado los cambios en la rama, estas deben fusionarse con git merge.
* El merge ocurre en la rama en la que se está posicionado. Por lo tanto, la rama a fusionar se transforma en la principal.
* Los merges también son commits.
* Los merges pueden generar conflictos, esto aborta la acción y pide que soluciones el problema manualmente, aceptando o rechazando los cambios que vienen.

# clase 18: Cambios en GitHub: de master a main

El escritor Argentino Julio Cortázar afirma que las palabras tienen color y peso. Por otro lado, los sinónimos existen por definición, pero no expresan lo mismo. Feo no es lo mismo que desagradable, ni aromático es lo mismo que oloroso.

Por lo anterior podemos afirmar que los sinónimos no expresan lo mismo, no tienen el mismo “color” ni el mismo “peso”.

Sí, esta lectura es parte del curso profesional de Git & GitHub. Quédate conmigo.

Desde el 1 de octubre de 2020 GitHub cambió el nombre de la rama principal: ya no es “master” -como aprenderás en el curso- sino main.

Este derivado de una profunda reflexión ocasionada por el movimiento #BlackLivesMatter.

La industria de la tecnología lleva muchos años usando términos como master, slave, blacklist o whitelist y esperamos pronto puedan ir desapareciendo.

Y sí, las palabras importan.

Por lo que de aquí en adelante cada vez que escuches a Freddy mencionar “master” debes saber que hace referencia a “main”

# Clase 19: Uso de GitHub

**GitHub** es una plataforma que nos permite guardar repositorios de Git que podemos usar como servidores remotos y ejecutar algunos comandos de forma visual e interactiva (sin necesidad de la consola de comandos).

Luego de crear nuestra cuenta, podemos crear o importar repositorios, crear organizaciones y proyectos de trabajo, descubrir repositorios de otras personas, contribuir a esos proyectos, dar estrellas y muchas otras cosas.

El README.md es el archivo que veremos por defecto al entrar a un repositorio. Es una muy buena práctica configurarlo para describir el proyecto, los requerimientos y las instrucciones que debemos seguir para contribuir correctamente.

Para clonar un repositorio desde GitHub (o cualquier otro servidor remoto) debemos copiar la URL (por ahora, usando HTTPS) y ejecutar el comando git clone + la URL que acabamos de copiar. Esto descargará la versión de nuestro proyecto que se encuentra en GitHub.

Sin embargo, esto solo funciona para las personas que quieren empezar a contribuir en el proyecto.

## Cómo conectar un repositorio de GitHub a nuestro documento local

Si queremos conectar el repositorio de GitHub con nuestro repositorio local, que creamos usando el comando git init, debemos ejecutar las siguientes instrucciones:

* 1. Guardar la URL del repositorio de GitHub con el nombre de origin

git remote add origin URL

* 1. Verificar que la URL se haya guardado correctamente:

git remote

git remote -v

* 1. Traer la versión del repositorio remoto y hacer merge para crear un commit con los archivos de ambas partes. Podemos usar git fetch y git merge o solo git pull con el flag --allow-unrelated-histories:

git pull origin **master** **--allow-unrelated-histories**

* 1. Por último, ahora sí podemos hacer git push para guardar los cambios de nuestro repositorio local en GitHub:

git push origin **master**

## Cómo autenticarte en GitHub 2022

Antes de empezar debemos renombrar la rama ‘máster’ a ‘main’, este es el nuevo estándar en GitHub, para esto:

1. Primero nos posicionamos en la rama a la que queremos cambiarle el nombre.
2. Ejecutamos el siguiente comando: git branch -M main

**Pasos para crear un token de acceso personal.**

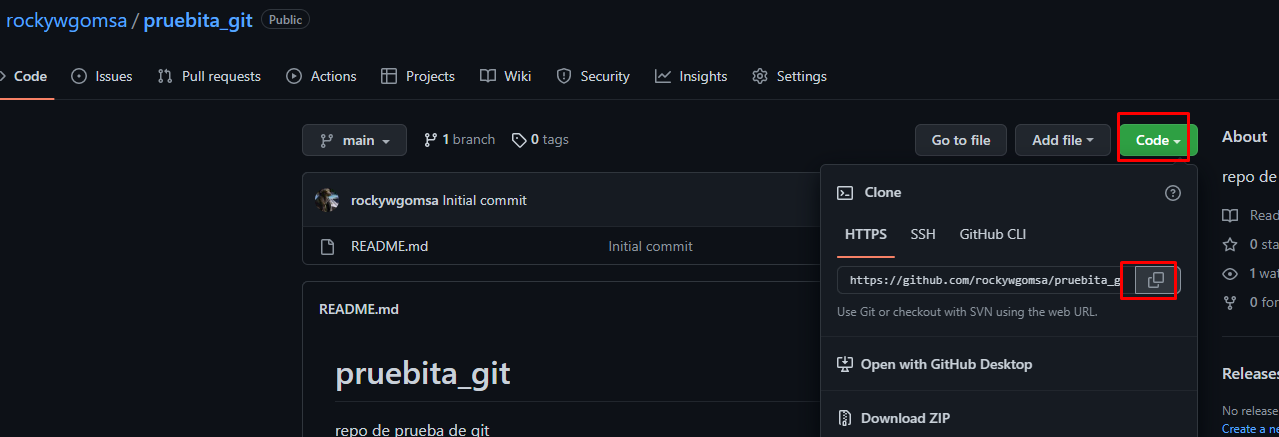
**Desde el 2022 GitHub** ya no deja hacer el push con la contraseña del propio GitHub, para esto tenemos que crear un token, y este token es la contraseña que vamos a colocar cuando nos pida clave

Seguir la secuencia: Ingresamos a nuestra cuenta de GitHub.

1. Buscamos Settings
2. Click en Developer settings
3. Click en Personal access tokens
4. Click en Generate new token aquí se puede colocar un nombre, la fecha de expiración.
5. Tildar en repo y luego click en el botón verde Generate token

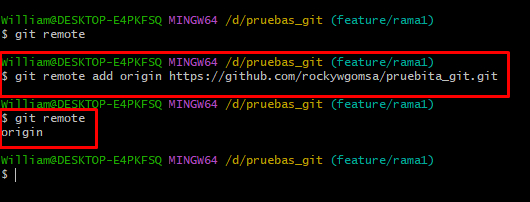
Subir mi repo a github, ejemplo de prueba\_git:

Copiar la url:



Agregamos esa url:

git remote add origin <https://github.com/rockywgomsa/pruebita_git.git>



# Clase 20 : Cómo funcionan las llaves públicas y privadas

Las **llaves públicas y privadas**, conocidas también como cifrado asimétrico de un solo camino, sirven para mandar mensajes privados entre varios nodos con la lógica de que firmas tu mensaje con una llave pública vinculada con una llave privada que puede leer el mensaje.

Las llaves públicas y privadas nos ayudan a cifrar y descifrar nuestros archivos de forma que los podamos compartir sin correr el riesgo de que sean interceptados por personas con malas intenciones.

## Cómo funciona un mensaje cifrado con llaves públicas y privadas

* 1. Ambas personas deben crear su llave pública y privada.
  2. Ambas personas pueden compartir su llave pública a las otras partes (recuerda que esta llave es pública, no hay problema si la “interceptan”).
  3. La persona que quiere compartir un mensaje puede usar la llave pública de la otra persona para cifrar los archivos y asegurarse que solo puedan ser descifrados con la llave privada de la persona con la que queremos compartir el mensaje.
  4. El mensaje está cifrado y puede ser enviado a la otra persona sin problemas en caso de que los archivos sean interceptados.
  5. La persona a la que enviamos el mensaje cifrado puede emplear su llave privada para descifrar el mensaje y ver los archivos.

Nota: puedes compartir tu llave pública, pero nunca tu llave privada.

# Clase 21: Configura tus llaves SSH en local

En este ejemplo, aprenderemos **cómo configurar nuestras llaves SSH en local**.

## Cómo generar tus llaves SSH

### 1. Generar tus llaves SSH\*\*

Recuerda que es muy buena idea proteger tu llave privada con una contraseña.

ssh-keygen -t rsa -**b** 4096 -C "tu@email.com"

### 2. Terminar de configurar nuestro sistema.

**En Windows y Linux**:

* Encender el “servidor” de llaves SSH de tu computadora:

eval $(ssh-agent -s)

* Añadir tu llave SSH a este “servidor”:

ssh-**add** ruta-donde-guardaste-tu-**llave-privada**

**En Mac**:

* Encender el “servidor” de llaves SSH de tu computadora:

eval "$(ssh-agent -s)"

Si usas una versión de OSX superior a Mac Sierra (v10.12), debes crear o modificar un archivo “config” en la carpeta de tu usuario con el siguiente contenido (ten cuidado con las mayúsculas):  
Host \*

AddKeysToAgent **yes**

UseKeychain **yes**

IdentityFile ruta-donde-guardaste-tu-llave-privada

* Añadir tu llave SSH al “servidor” de llaves SSH de tu computadora (en caso de error puedes ejecutar este mismo comando pero sin el argumento -K):

ssh-**add** -K ruta-donde-guardaste-tu-**llave-privada**

# clase 22: Conexión a GitHub con SSH

La creación de las SSH es necesario solo una vez por cada computadora. Aquí conocerás **cómo conectar a GitHub usando SSH**.

Luego de crear nuestras llaves SSH podemos entregarle la llave pública a GitHub para comunicarnos de forma segura y sin necesidad de escribir nuestro usuario y contraseña todo el tiempo.

Para esto debes entrar a la [Configuración de Llaves SSH en GitHub](https://github.com/settings/keys), crear una nueva llave con el nombre que le quieras dar y el contenido de la llave pública de tu computadora.

Ahora podemos actualizar la URL que guardamos en nuestro repositorio remoto, solo que, en vez de guardar la URL con HTTPS, vamos a usar la URL con SSH:

ssh

git remote **set**-**url** origin **url**-ssh-del-repositorio-en-github

## Comandos para copiar la llave SSH:

-**Mac**:

pbcopy < ~/.ssh/id\_rsa.pub

* **Windows (Git Bash)**:

clip < ~/.ssh/id\_rsa.pub

* **Linux (Ubuntu)**:

cat ~/.ssh/id\_rsa.pub

# clase 23: Tags y versiones en Git y GitHub

Los tags o etiquetas nos permiten asignar versiones a los commits con cambios más importantes o significativos de nuestro proyecto.

## Comandos para trabajar con etiquetas:

* Crear un nuevo tag y asignarlo a un commit: **git tag -a nombre-del-tag id-del-commit**.
* Borrar un tag en el repositorio local: **git tag -d nombre-del-tag**.
* Listar los tags de nuestro repositorio local: **git tag** o **git show-ref --tags**.
* Publicar un tag en el repositorio remoto: **git push origin --tags**.
* Borrar un tag del repositorio remoto: git tag -d nombre-del-tag y **git push origin :refs/tags/nombre-del-tag**.

Para generar un comando complejo con varios comandos de una forma optimizada, utilizamos conjuntos de sentencias conocidas como alias.

## Cómo aregar un alias solo para git

* Para un proyecto:

git config alias.arbolito "log **--**all **--**graph **--**decorate **--**oneline"

* Global:

git config **--**global alias.arbolito "log **--**all **--**graph **--**decorate **--**oneline"

* Para correrlo:

git arbolito

para ver en forma de árbol. Ejecuta, pero a ese se le puede asignar un alias.

git log --all --graph --decorate –oneline

creando un alias:

alias arbolito="git log --all --graph --decorate --oneline"

# clase 24: Manejo de ramas en GitHub

Si no te funciona el comando gitk es posible no lo tengas instalado por defecto.  
Para instalar gitk debemos ejecutar los siguientes comandos:  
sudo apt-get update  
sudo apt-get install gitk

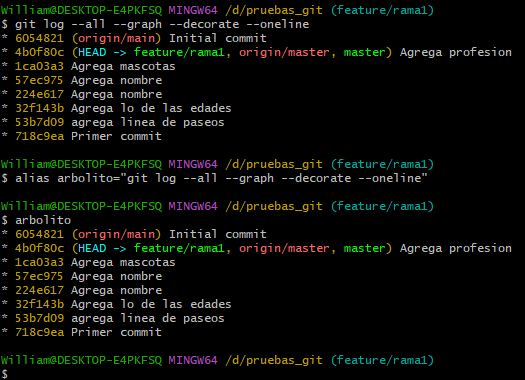
Las ramas nos permiten hacer cambios a nuestros archivos sin modificar la versión principal (master). Puedes trabajar con ramas que nunca envías a GitHub, así como pueden haber ramas importantes en GitHub que nunca usas en el repositorio local. Lo crucial es que aprendas a manejarlas para trabajar profesionalmente.

Si, estando en otra rama, modificamos los archivos y hacemos commit, tanto el historial(git log) como los archivos serán afectados. La ventaja que tiene usar ramas es que las modificaciones solo afectarán a esa rama en particular. Si luego de “guardar” los archivos(usando commit) nos movemos a otra rama (git checkout otraRama) veremos como las modificaciones de la rama pasada **no aparecen** en la otraRama.

## Comandos para manejo de ramas en GitHub

* Crear una rama:  
  git branch branchName
* Movernos a otra rama:  
  git checkout branchName
* Crear una rama en el repositorio local:  
  git branch nombre-de-la-rama o git checkout -b nombre-de-la-rama.
* Publicar una rama local al repositorio remoto:  
  git push origin nombre-de-la-rama.

Recuerda que podemos ver gráficamente nuestro entorno y flujo de trabajo local con Git utilizando el comando gitk. Gitk fue el primer visor gráfico que se desarrolló para ver de manera gráfica el historial de un repositorio de Git.



# Clase 24: Manejo de ramas en GitHub

Si no te funciona el comando gitk es posible no lo tengas instalado por defecto.  
Para instalar gitk debemos ejecutar los siguientes comandos:  
sudo apt-get update  
sudo apt-get install gitk

Las ramas nos permiten hacer cambios a nuestros archivos sin modificar la versión principal (master). Puedes trabajar con ramas que nunca envías a GitHub, así como pueden haber ramas importantes en GitHub que nunca usas en el repositorio local. Lo crucial es que aprendas a manejarlas para trabajar profesionalmente.

Si, estando en otra rama, modificamos los archivos y hacemos commit, tanto el historial(git log) como los archivos serán afectados. La ventaja que tiene usar ramas es que las modificaciones solo afectarán a esa rama en particular. Si luego de “guardar” los archivos(usando commit) nos movemos a otra rama (git checkout otraRama) veremos como las modificaciones de la rama pasada **no aparecen** en la otraRama.

## Comandos para manejo de ramas en GitHub

* Crear una rama:  
  git branch branchName
* Movernos a otra rama:  
  git checkout branchName
* Crear una rama en el repositorio local:  
  git branch nombre-de-la-rama o git checkout -b nombre-de-la-rama.
* Publicar una rama local al repositorio remoto:  
  git push origin nombre-de-la-rama.

Recuerda que podemos ver gráficamente nuestro entorno y flujo de trabajo local con Git utilizando el comando gitk. Gitk fue el primer visor gráfico que se desarrolló para ver de manera gráfica el historial de un repositorio de Git.

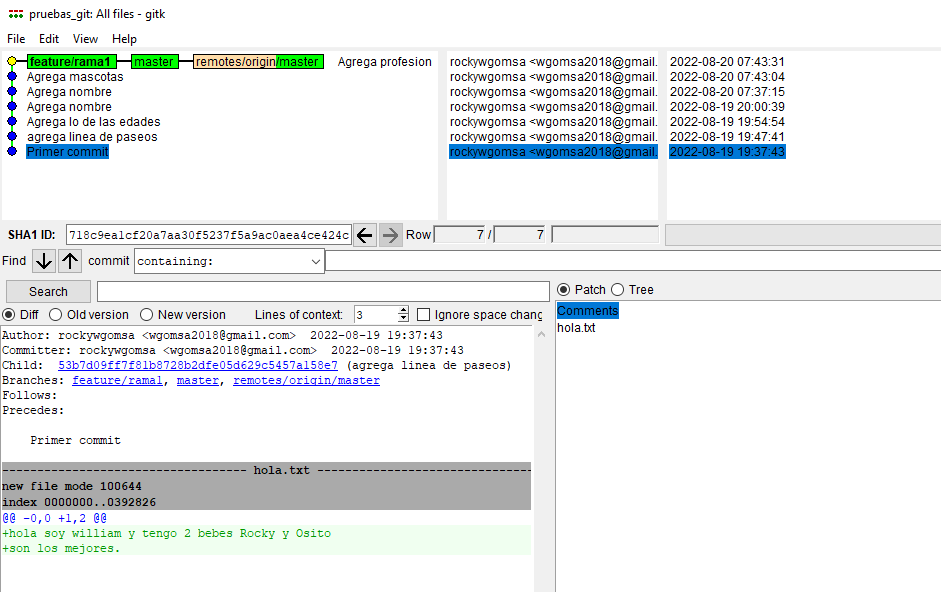
Ver todas las ramas con su historia:

git show-branch

git show-branch –all

ver en modo grafico:

gitk



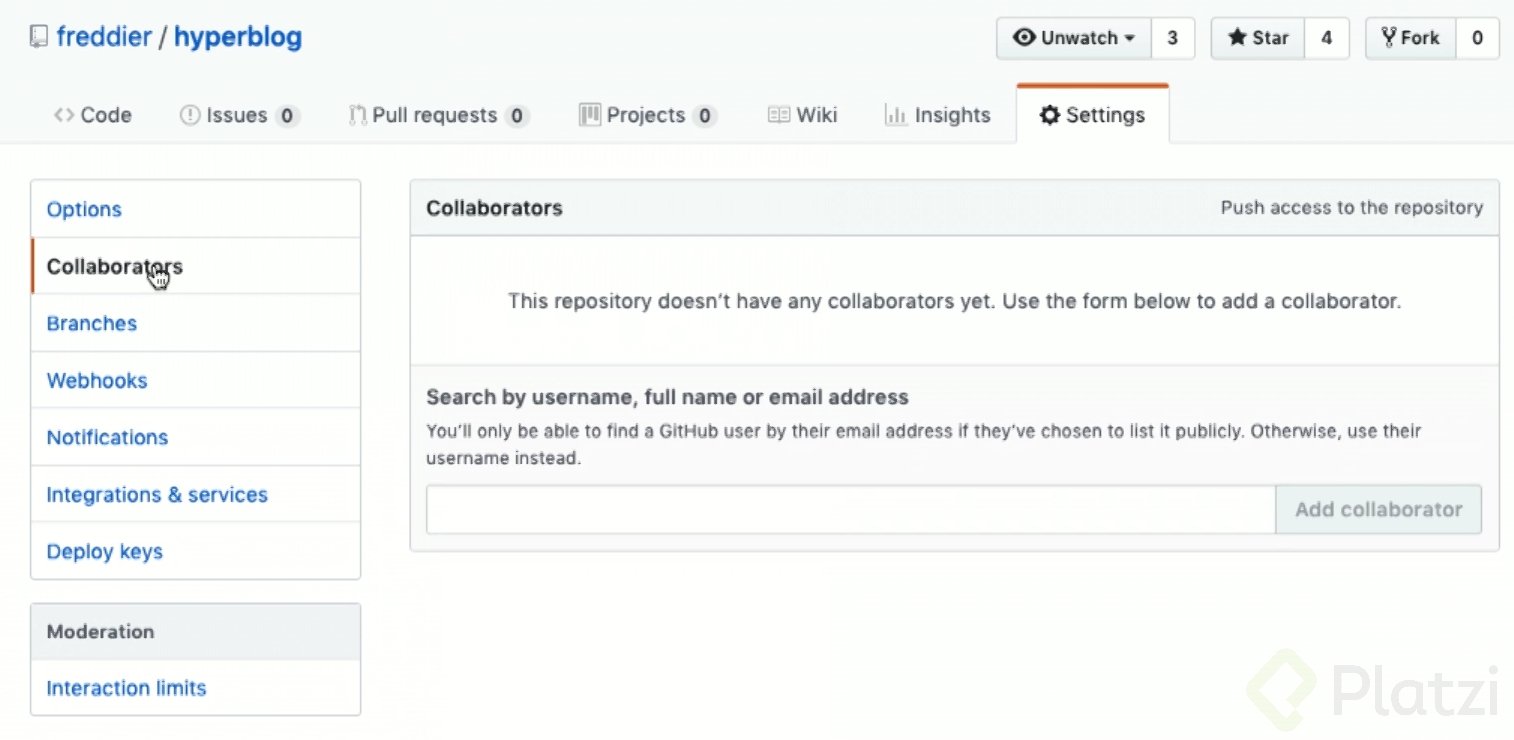
# Clase 25: Configurar múltiples colaboradores en un repositorio de GitHub

Por defecto, cualquier persona puede clonar o descargar tu proyecto desde GitHub, pero no pueden crear commits, ni ramas. Esto quiere decir que pueden copiar tu proyecto pero no colaborar con él. Existen varias formas de solucionar esto para poder aceptar contribuciones. Una de ellas es añadir a cada persona de nuestro equipo como colaborador de nuestro repositorio.

## Cómo agregar colaboradores en Github

* Solo debemos entrar a la configuración de colaboradores de nuestro proyecto. Se encuentra en:  
  Repositorio > Settings > Collaborators

Ahí, debemos añadir el email o username de los nuevos colaboradores.



Si, como colaborador, agregaste erróneamente el mensaje del commit, lo puedes cambiar de la siguiente manera:

* Hacer un commit con el nuevo mensaje que queremos, esto nos abre el editor de texto de la terminal:  
  git commit —amend
* Corregimos el mensaje
* Traer el repositorio remoto  
  git pull origin master
* Ejecutar el cambio  
  git push --set-upstream origin master

# Clase 26: Flujo de trabajo profesional: Haciendo merge de ramas de desarrollo a master

Para poder desarrollar software de manera óptima y ordenada, necesitamos tener un flujo de trabajo profesional, que nos permita trabajar en conjunto sin interrumpir el trabajo de otros desarrolladores. Una buena práctica de flujo de trabajo sería la siguiente:

* 1. Crear ramas
  2. Asignar una rama a cada programador
  3. El programador baja el repositorio con git pull origin master
  4. El programador cambia de rama
  5. El programador trabaja en esa rama y hace *commits*
  6. El programador sube su trabajo con git push origin #nombre\_rama
  7. El encargado de organizar el proyecto baja, revisa y unifica todos los cambios

# Clase 27: Flujo de trabajo profesional con Pull requests

En un entorno profesional normalmente se bloquea la rama **master**, y para enviar código a dicha rama pasa por un code review y luego de su aprobación se unen códigos con los llamados merge request.

Para realizar pruebas enviamos el código a servidores que normalmente los llamamos staging develop (servidores de pruebas) luego de que se realizan las pruebas pertinentes tanto de código como de la aplicación estos pasan al servidor de producción con el ya antes mencionado merge request.

Los PR (pull requests) son la base de la colaboración a proyectos Open Source, si tienen pensando colaborar en alguno es muy importante entender esto y ver cómo se hace en las próximas clases. Por lo general es forkear el proyecto, implementar el cambio en una nueva rama, hacer el PR y esperar que los administradores del proyecto hagan el merge o pidan algún cambio en el código o commits que hiciste.

## Proceso de un pull request para trabajo en producción:

* Un pull request es un estado intermedio antes de enviar el merge.
* El pull request permite que otros miembros del equipo revisen el código y así aprobar el merge a la rama.
* Permite a las personas que no forman el equipo, trabajar y colaborar con una rama.
* La persona que tiene la responsabilidad de aceptar los pull request y hacer los merge tienen un perfil especial y son llamados DevOps

# Clase 28: Utilizando Pull Requests en GitHub

**Pull request** es una funcionalidad de Github (en Gitlab llamada merge request y en Bitbucket push request), en la que un colaborador pide que revisen sus cambios antes de hacer merge a una rama, normalmente master (ahora conocida como main).

Al hacer un pull request, se genera una conversación que pueden seguir los demás usuarios del repositorio, así como autorizar y rechazar los cambios.

## Cómo se realiza un pull request

* Se trabaja en una rama paralela los cambios que se desean git checkout -b <rama>.
* Se hace un commit a la rama git commit -am '<Comentario>'.
* Se suben al remoto los cambios git push origin <rama>.
* En GitHub se hace el pull request comparando la rama master con la rama del fix.
* Uno, o varios colaboradores revisan que el código sea correcto y dan feedback (en el chat del pull request).
* El colaborador hace los cambios que desea en la rama y lo vuelve a subir al remoto (automáticamente jala la historia de los cambios que se hagan en la rama, en remoto).
* Se aceptan los cambios en GitHub.
* Se hace merge a master desde GitHub.

falta