

Git y GitHub

# **Introducción a Git**

## Que es git

Git es un software de control de versiones diseñado por Linus Torvald, pensando en la eficiencia y la coniabilidad del mantenimiento de versiones de aplicaciones cuando estas tienen un gran número de archivos de código fuente. En su lugar GitHub es un forja para alojar proyectos utilizando el sistema de control de versiones Git. GitHub sería la red social de código para los programadores, tu propio curriculum vitae.

## Por qué usar un sistema de control de versiones como Git?

Un sistema de control de versiones como 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 proyecto 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 incrementa, 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, indicar 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 ultimos cambios es ***git add****.* Este comando no almacena las actualizaciones de forma definitiva, solo las guarda en algo que conocemos como “Staging Area” (no te preocupes, lo entenderemos más adelante).
* 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 nuestro commits a un servidor remoto, un lugar donde todos podamos conectar nuestro proyecto, usamos el comando ***git push****.*

## Instalando Git y Git Bash en Windows

Windows y Linux comandos diferentes, graban el enter de formas diferentes y tienen muchas otras diferencias. Cuando instales Git Bash en Windows debes elegir si prefieres trabajar con la forma de Windows o la forma de UNIX (Linux y Mac).

Ten en cuenta que, normalmente, los entornos de desarrollo profesionales tienen personas que usan sistemas operativos diferentes. Esto significa que, si todos podemos usar los mismos comandos, el trabajo resultará más fácil para todos en el equipo.

Los comandos en UNIX son los más comunes entre los equipos de desarrollo. Así que, a menos que trabajes con tecnologías nativas de Microsoft (por ejemplo, .NET), la recomendación es que elijas la opción de la terminal tipo UNIX para obtener una mejor compatibilidad con todo tu equipo.

## Editores de código, archivos binarios y de texto plano

Un editor de código es una herramienta que nos brinda muchas ayudas para escribir codigo, algo asi como un bloc de notas muy avanzado. Los editores más populares son VSCode, Sublime Text y Atom, pero no necesariamente debes usar alguno de estos para continuar con el curso.

Tipos de archivos y sus diferencias:

* **Archivos de Texto (.txt)**: Texto plano normal y sin nada especial. Lo vemos igual sin importar donde 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 muchos más complejos que solo el texto plano. Esto es porque debe guardar todos los estilos de texto y, para esto, usa un código especial un código 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 archivos.*

## Introducción a la terminal y línea de comandos

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

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

Recuerda que Git Bash 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 Carpetas-Importante).*
* **touch**: Nos permite crear archivos (por ejemplo, *touch archivos.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**: Nos permite cambiar ver los archivos de la carpeta donde estamos ahora mismo. Podemos usar uno o más argumentos para ver más informacion sobre esto 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:* Navega 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.
* **! + numero**: Ejecutar algun comando con el numero 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).*

salto de página

# **Comandos básicos en Git**

## Que 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, que guardará temporalmente nuestros archivos (cuando ejecutamos 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):

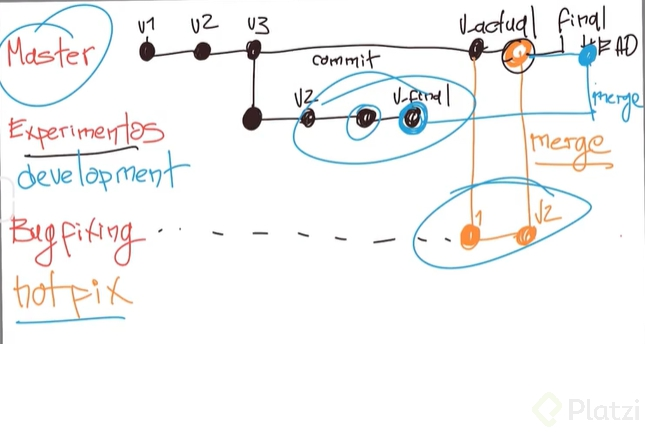
* **Archivos Tracked**: son los archivos que viven dentro de Git, no tienen cambios pendientes y sus últimas actualizaciones han sido guardadas en el repositorio gracias a los comandos git add y git commit.
* **Archivos Staged**: son archivos en Staging. Viven dentro de Git y hay registro de ellos porque 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**: entiendelos 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 Staged. Esta es una ocasión especial, los archivos han sido guardado 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).

## Que es un Branch (rama) y como 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 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 estandar: 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.**

Solo ten en cuenta que combinar estas ramas (si, 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”.

## Crear un repositorio de Git y haz tu primer commit

*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.*

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).

## 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 cuando se produjeron ciertos cambios, que se rompió y como 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 de archivos, podemos usar el comando ***git diff commitA commitB.***

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

## Volver en el tiempo en nuestro repositorio utilizando branches 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:*** 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 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 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 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 que 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.

salto de página

# **Flujo de trabajo básico en Git**

## Flujo de trabajo básico con un repositorio remoto

Por ahora, nuestro proyecto vive únicamente en nuestra computadora. Esto significa que no hay forma de que otros miembros del equipo trabajen en él.

Para solucionar esto están 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 para 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:

* ***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 de 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 repositorios local (en caso de que haya, por supuesto).
* ***git merge:*** También usamos el comando *git fetch* 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.

## 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 cabeza 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 movemos en el tiempo a cualquier otro commit de cualquier otra rama con los comandos (***git reset id, git checkout rama -o -id -commit***).

## Fusión de ramas con Git merge

El comando ***git merge*** nos permite crear un nuevo commit con la combinación de dos ramas (la rama donde nos encontramos cuando ejecutamos el comando y la rama que indicamos después del comando).

# Crear un nuevo commit en la rama master combinando

# los cambios de la rama cabecera:

git checkout master

git merge cabecera

# 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 cambios queremos conservar de una rama y que otros de la otra. El problema es que no siempre puede adivinar, sobretodo 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 haya guardado. Guarda tus cambios antes de hacer ***git checkout.***

## Solución 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 como 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 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.

salto de página

# Trabajando con repositorios remotos en GitHub

## 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 descargara 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. Si queremos conectar el repositorio de GitHub con nuestro repositorio local, el que creamos con ***git init***, debemos ejecutar las siguientes instrucciones:

# Primero: Guardar la URL del repositorio de GitHub

# con el nombre de origin

git remote add origin URL

# Segundo: Verificar que la URL se haya guardado

# correctamente:

git remote

git remote -v

# Tercero: 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 el git pull con el flag --allow-unrelated-histories:

git pull origin master --allow-unrelated-histories

# Por último, ahora sí podemos hacer git push para guardar

# los cambios de nuestro repositorio local en GitHub:

git push origin master

## Cómo funcionan las llaves públicas y privadas

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

La forma de hacerlo es la siguiente:

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 usar su llave privada para descifrar el mensaje y ver los archivos.

Puedes compartir tu llave pública pero nunca tu llave privada.

## Configura tus llaves SSH en local

**Primer paso: 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](mailto:tu@email.com)"

**Segundo paso:** 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**

## Conexión a GitHub con SSH

Luego de crear nuestra 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:

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

## 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 :refs/tags/nombre-del-tag***

## Manejo de ramas en GitHub

Puedes trabajar con ramas que nunca envias a GitHub, así como pueden haber ramas importantes en GitHub que nunca usas en el repositorio local. Lo importante que aprendas a manejarlas para trabajar profesionalmente.

* 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 usando el comando ***gitk***

## 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, ni nada.

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 proyecto.

Solo debemos entrar a la configuración de colaboradores de nuestro proyecto (*repositorio > settings > collaborators*) y añadir el email o username de los nuevos colaboradores.

salto de página

# **Flujos de trabajo profesionales**

## Flujo de trabajo profesional: Haciendo merge de desarrollo a master

Flujo profesional: Cada quien trabaja en su determinado branch y un delegado confirma los cambios al máster una vez terminados los cambios

Nota: GitHub soporta archivos binarios aunque tarda en actualizarse y no es una buena práctica

## 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 codigo 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 a el servidor de producción con el ya antes mencionado *merge request.*

## Como crear un buen README.md y sintaxis de markdown

El README es el archivo en el cual hacemos la descripción del proyecto, ya sea open source o privados es importante tener un buen README.Este archivo se escribe con formato markdown

**Markdown**

Es un formato de escritura que permite la generación de contenido fácil y rápido, permite generar una salida (por lo general) en formato HTML sin necesidad de aprender a profundidad HTML. Es ampliamente utilizado por su facilidad de generar texto enriquecido.

**Encabezados:**

Lo utilizamos para resaltar una parte importante, títulos, subtítulos, etc. Se utiliza el símbolo # para demarcar el inicio de un encabezado.

# Encabezado nivel 1

## Encabezado nivel 2

### Encabezado nivel 3

#### Encabezado nivel 4

##### Encabezado nivel 5

###### Encabezado nivel 6

**Párrafos:**

En formato Markdown escribirlos no es tan distinto a escribir en un texto plano, automáticamente se reconoce que es un párrafo, por ejemplo: JavaScript es un lenguaje muy poderoso.

**Itálicas y negritas**

Hat partes en las que necesitamos hacer énfasis en ciertas palabras, lo común es que utilicemos itálicas y negritas para resaltarlas, en Markdown debemos hacer lo siguiente:

\*\*Esto es una negrita\*\*

\*Esto es una itálica\*

\*\*\_Esto es una negrita con itálica\_\*\*

**Citas**

Se utilizan para mostrar referencias a otros autores, en markdown hacemos:

> Esto es una cita

Podemos poner citas con varios párrafos

> Este es el primer párrafo

>

> Este es el último párrafo

**Código**

Es esencial que en los README podamos escribir código, esto para especificar la instalación o partes que debemos resaltar de nuestro proyecto. Hay dos formas en las que podemos resaltar código, dentro de un párrafo o en una sección completa, tal cual estamos haciendo en esta clase.

Esto es un pedazo de código dentro de un párrafo *console.log(“Hola Mundo”)*

Para insertar código lo que hacemos es dejar un tabulación y automaticamente lo reconoce como código si no podemos utilizar ``para crear el bloque, así:

var **name** = 'Escuela de Javascript'

console.log(**name**)

**Cómo escribir un buen README**

No hay un estándar sobre cómo escribir un buen README, cada proyecto es diferente y depende de cada uno. Pero hay ciertas partes que si o si debería contener un buen README.

1. Nombre: Especificamos como se llama nuestro proyecto.
2. Descripción: Es donde diremos para qué exactamente es el proyecto, que problemas resuelve y cualquier información relevante.
3. Instalación: Muestra los pasos específicos para instalar el proyecto. Por lo general se muestra un pedazo del código necesario para la instalación.
4. Cómo usar: Describe rápidamente casos de uso en los cuales se puede usar el proyecto, además de mostrar funcionalidades.
5. Cómo contribuir: Si es un proyecto open source se describe acá la forma en la que deberían crearse las contribuciones.
6. Licencia: Muestra la licencia que tiene el proyecto. En formato markdown podemos escribir cada uno de los ítems de esta manera: