**COMANDOS DE GIT**

**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 config:** para ver las posibles configuraciones
* **git config --list:** para ver la lista de configuraciones hechas
* **git config --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
* **git show nombre\_del\_archivo.txt:** Nos permite observar los cambios que hemos realizado sobre un documento.
* **git log nombre\_del\_archivo.txt:** Este comando me permite ver la información de todos los commit realizados.
* **git diff versión1 version2:** Este comando permite ver una comparación entre dos commit para poder visualizar los cambios que se han hecho en los archivos.
* **git push:** envía a otro repositorio remoto lo que estamos haciendo
* **git pull:** traer repositorio remoto
* **git checkout:** traer cambios realizados

**¿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.

**¿Qué es git init?**

git init es 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:

**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.

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

Estos son los comandos más importantes que debes conocer:

**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).

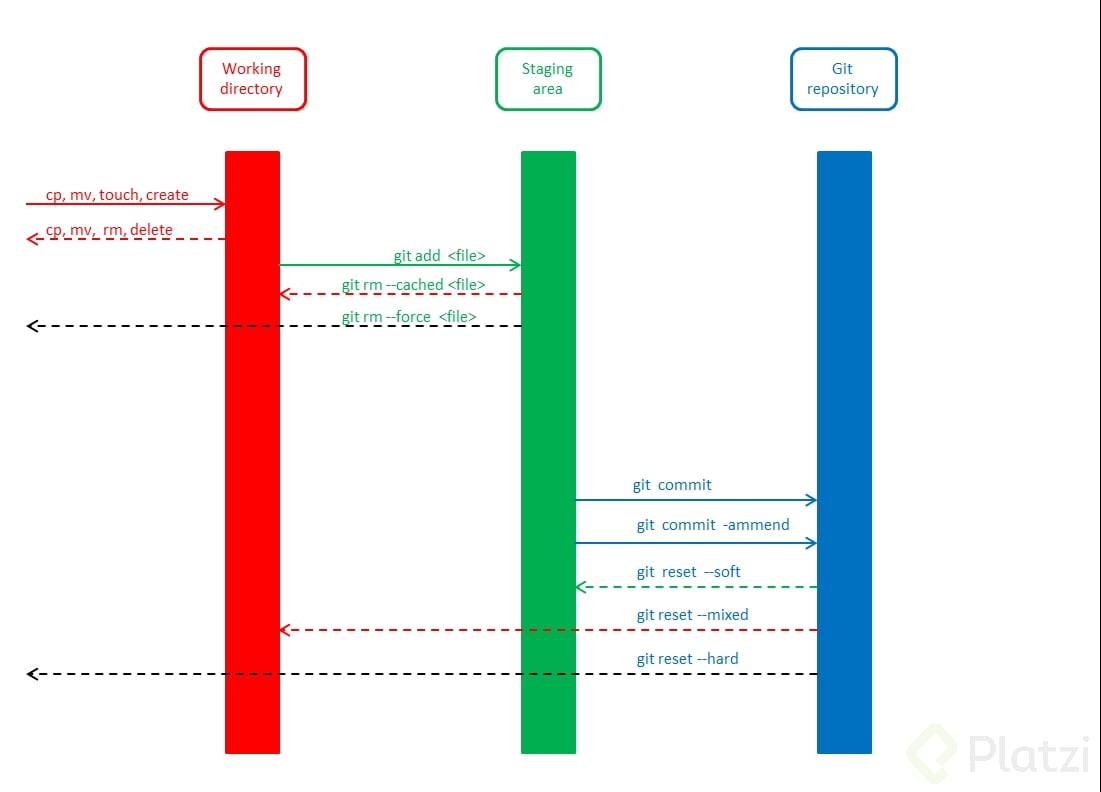
**git reset HEAD:** Nos ayuda a sacar archivos del estado staged para devolverlos a su estado anterior. Si los archivos venían de unstaged, vuelven allí. Y lo mismo se venían de untracked.

**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"**). En caso de que no queramos realizar el git add podemos hacerlo directamente en el commit agregando el argumento -a, y para agregar el mensaje ponemos -am.

**Git rm**

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

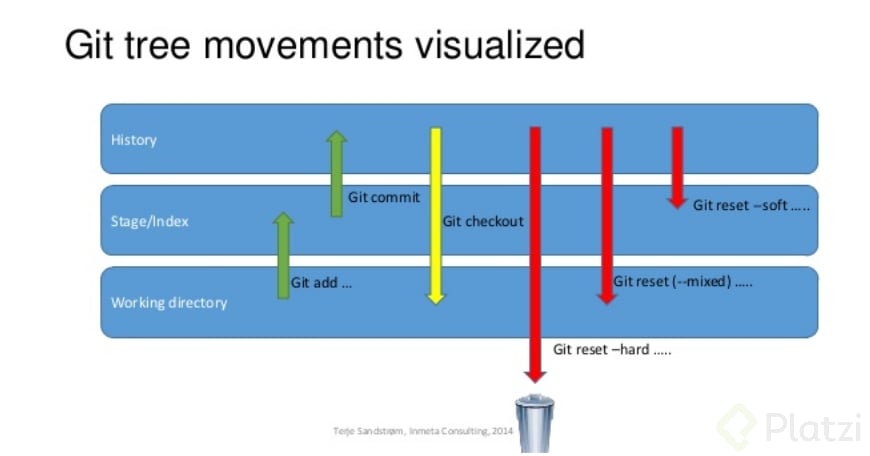
* **git rm --cached:** mueve los archivos que le indiquemos al estado untracked.
* **git rm --force:** elimina los archivos de git y del disco duro. Git guarda el registro de la existencia de los archivos, por lo que podremos recuperarlos si es necesario (pero debemos usar comandos más avanzados).



**COMO REGRESAR EN EL TIEMPO UN REPOSITORIO.**

Para regresar en el tiempo un código podemos usar dos comandos:

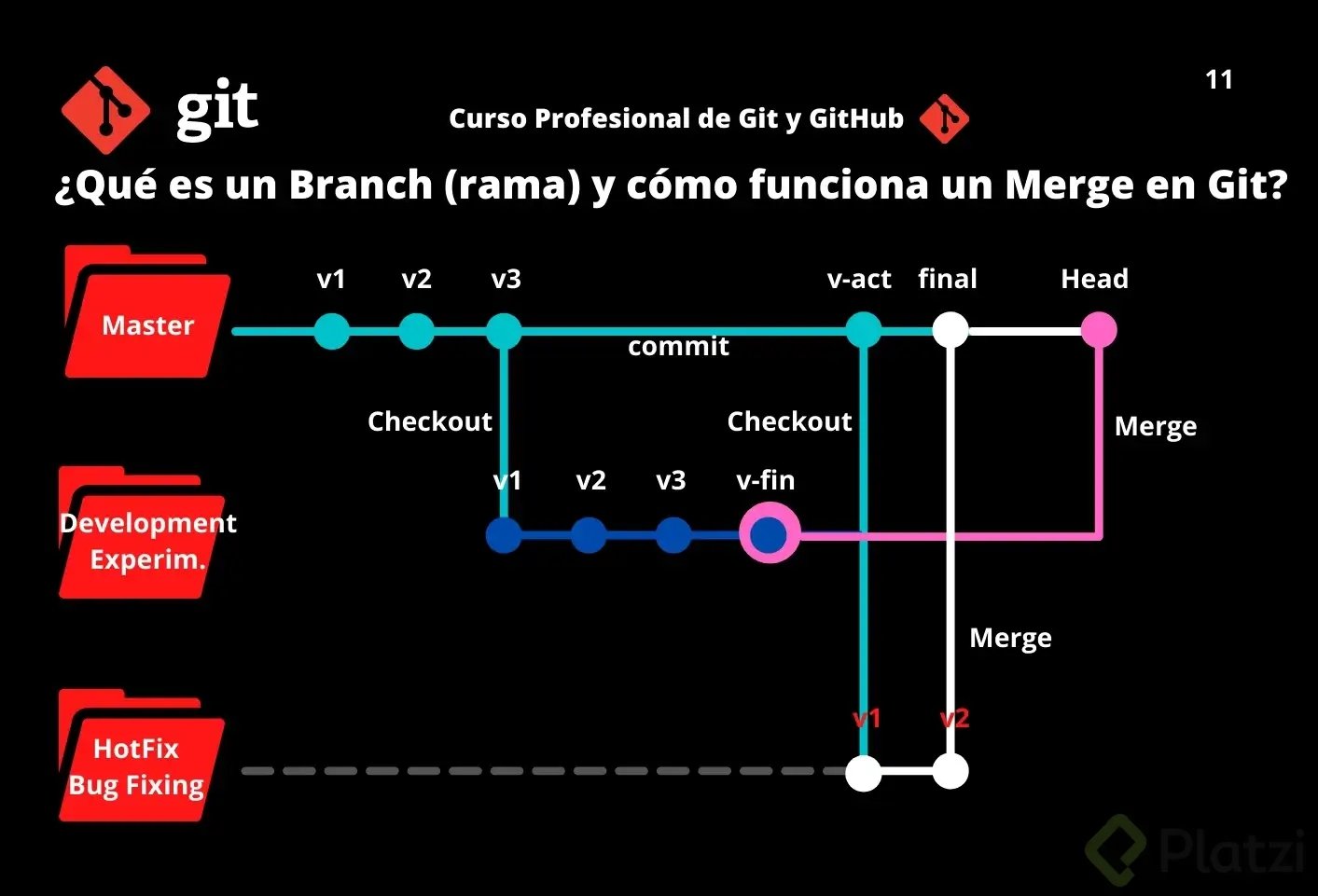
* **reset:** Este comando nos permite setear nuestro repositorio a una versión antigua borrando todos los cambios posteriores al commit que hayamos seleccionado. Existen dos tipos de reset que podemos hacer:
  + **git reset -–hard:** acompañado del argumento –hard, borra toda la información posterior al commit incluyendo lo que tengamos en el área staging, un borrado permanente sin opción de recuperar nada.
  + **git reset –-soft:** acompañado del argumento –soft, mantiene allí los archivos del área de staging para que podamos aplicar nuestros últimos cambios pero desde un commit anterior.
  + **git reset --mixed:** Borra todo, exactamente todo. Toda la información de los commits y del área de staging se elimina del historial.
  + **git reset HEAD:** El comando git reset saca archivos del área de staging sin borrarlos ni realizar otras acciones. Esto impide que los últimos cambios en estos archivos se envíen al último commit. Con este comando puedes cancelar los cambios que ya habías agregado, para que puedas revisarlos, modificarlos o deshacerlos antes de confirmarlos con un commit.
* **checkout:** Este comando 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. La ventaja de este es que nos permite regresar a una versión anterior sin causar daños a nuestra rama master, y también nos deja regresar a la rama master.



**¿Cuál es la diferencia entre git rm y git reset Head?**

La diferencia principal entre git rm y git reset HEAD radica en que git rm elimina archivos del repositorio y de la historia del proyecto, mientras que git reset saca los cambios del área de preparación y los mueve del espacio de trabajo, sin afectar la historia del repositorio.

**BRANCH (RAMA)**



Una rama en git es la creación de una nueva versión del código a partir de cualquier commit, las ramas se dividen en tres:

* **Master:** Esta es la rama principal (ahora debe llamarse main)
* **Development:** En esta rama es en la que los desarrolladores realizan experimentos del código.
* **Hotfix:** Esta es la rama en la que se buscan las soluciones a posibles bugs encontrados en la línea principal, permitiendo buscar soluciones sin dañar el código original.
* **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. También podemos usar git checkout -b rama para que se nos redirija inmediatamente a la nueva ram creada.

Cuando creamos una nueva rama esta se crea a partir del lugar en el que estamos ubicados, siendo una copia exacta de esta rama a partir del último commit realizado.

Una vez que creamos una nueva rama y hacer git show podemos ver que el ultimo commit esta ligado a dos ramas, la principal y la nueva, pero una vez hemos realizado un nuevo commit solo apunta a una rama.

Cada rama nos permite crear nuevas versiones del código con el fin de solucionar bugs u optimizar el código, podemos hacer funciones entre ramas, generalmente con la rama principal.

* **Merge:** merge es el nombre del comando que nos permite fusionar dos ramas, la fusión se realiza de la rama master con alguna otra. Cuando hacemos merge ambas ramas se fusionan creando una nueva en la que se guardan todos los cambios realizados en ambas, creando así una nueva versión de la rama master.

Hacer merge puede llegar a generar conflictos entre ramas.

En caso de que la misma parte del código haya cambiado 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*.

Para evitarnos problemas de perder información debemos realizar un commit antes de hacer un checkout, en caso de estar en repositorio remoto usar los comandos básicos para mantenerlo actualizado **(git fetch, git push y git pull)**.

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

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.

**Como solucionar problemas de Merge:**

Al hacer merge pueden llegar a surgir algunos problemas, como por ejemplo que en ambas ramas se le hayan realizado cambios a las mismas líneas de código, en este caso se creará un nuevo estado conocido como unmerged. Para solucionar este error solo debemos cambiar nosotros mismo las partes del código que generan conflicto eligiendo cual queremos que sea la que se guarde (vs code nos da la opción de seleccionar cual queremos que se quede y elimina la otra automáticamente), y luego solo tenemos que volver hacer el commit.

En caso de querer eliminar una rama podemos usar los siguientes comandos:

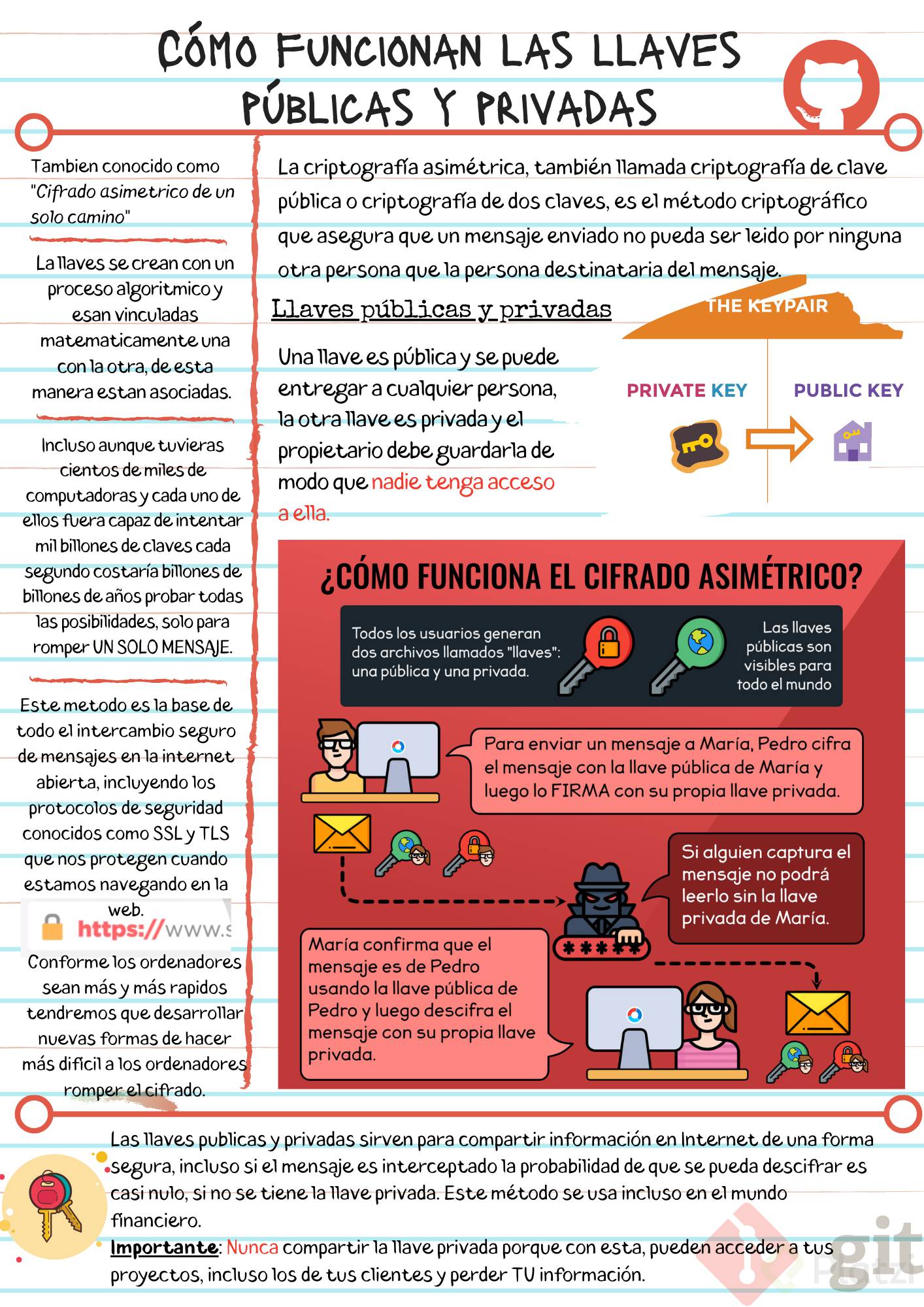
* **git branch -d rama\_eliminar:** Eliminara la rama que hallamos indicado.
* **git branch -D rama\_eliminar:** Si la rama tiene cambios que aún no se han fusionado con otras ramas, Git mostrará un mensaje de advertencia. Para esto se usa la D en mayúscula eliminando así la rama sin fusionar los cambios.
* **git push origin --delete rama\_elimar:** este comando eliminará la rama que seleccionemos del repositorio remoto en el que estemos.

**REPOSITORIOS REMOTOS**

En git también podemos trabajar con repositorios remotos alojados en servidores como Github, GitLab, entre otros. Git nos permite crear una copia de dicho repositorio en nuestro almacenamiento local y también nos permite enviar nuestros cambios a dicho repositorio.

* **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.

**LLAVES PUBLICAS Y LLAVES PRIVADAS**



* **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**

**CONECTANDO GIT CON GITHUB**

Luego de generar nuestra clave ssh debemos dirigirnos a nuestra cuenta de github y vinvular la llave publica, para esto:

* Entramos a la sección de configuración

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

Nos dirigimos al apartado de “ssh and gpg keys”

Texto

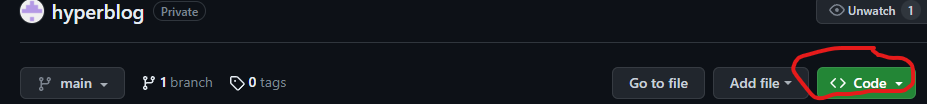
Descripción generada automáticamente

* Clickeamos en “new ssh”, asignamos un título para esta llave y pegamos el contenido de nuestra llave pública.

**IMPORTANTE**: Para cada dispositivo que usemos es necesario que creemos una nueva llave, debido a que si intentamos pasar nuestra llave privada de un lugar a otro esta se podría ver comprometida.

Una vez hemos vinculada la llave, procedemos a crear nuestro repositorio en Github para conectarlo nuestro repositorio local.

Una vez tenemos nuestro repositorio debemos buscar el link https de este.



Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

Copiamos el link y nos dirigimos a git:

* Guardar la URL del repositorio de GitHub con el nombre de origin
* git remote add origin URL
* Verificar que la URL se haya guardado correctamente:
* git remote
* git remote -v
* Al agregar la url a git nos dará un error debido a que en el repositorio de girhub tenemos un archivo, por lo que debemos 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 para combinar las historias de los cambios:
* git pull origin main --allow-unrelated-histories
* Por último, ahora sí podemos hacer git push para guardar los cambios de nuestro repositorio local en GitHub:
* git push origin main
* 

**IMPORTANTE:** Cuando vayamos a hacer **push** primero debemos hacer un **pull** por si se han realizado cambios en el repositorio remoto.

* Para guardar ramas en github utilizamos el comando:
* git push origin nombre-de-la-rama
* No es necesario que enviemos todas nuestras ramas al repo remoto, sola hace falta enviar las más importantes.

**TAGS Y VERSIONES**

En Git, las etiquetas o **Git tags** tienen un papel importante al asignar versiones a los commits más significativos de un proyecto.

* **Creación de etiquetas en Git**

Para crear una etiqueta, ejecuta el siguiente comando:

* git tag -a tagname -m "descripción" hash
* **Uso compartido de etiquetas**

Compartir etiquetas requiere un enfoque explícito al usar el comando **git push**. Por defecto, las etiquetas no se envían automáticamente. Para enviar etiquetas específicas, utiliza:

* git push origin tagname

Para enviar varias etiquetas a la vez, usa:

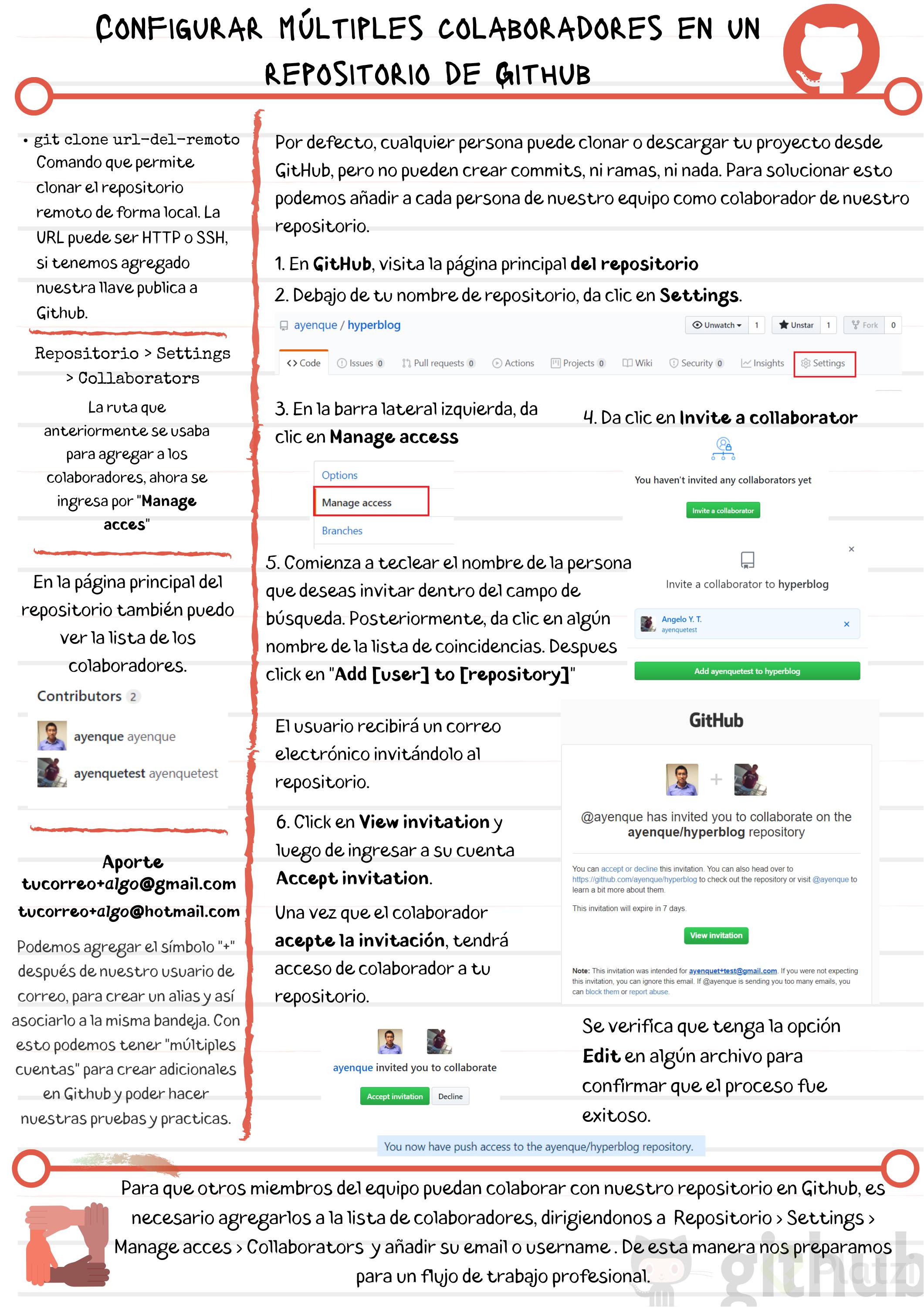
* git push origin --tags
* **Eliminación de etiquetas**

Para eliminar una etiqueta, usa el siguiente comando:

* git **tag** **-d** tagname

Sin embargo, esto no implica que se eliminen del repositorio remoto, por lo cual el comando para eliminar la etiqueta del repositorio remoto es la siguiente:

* git push origin :refs/tags/tagname



**PULL REQUEST**

En un ambiente profesional la rama main se bloque para que solo algunos puedan realizar cambios en esta. Para hacer un merge a la rama main primero se debe pasar por lo que se conoce como pull request, que básicamente es un es una petición en la que los administradores del repositorio (los DevOps), quienes se encargan de validar que el nuevo código no cause problemas con la rama principal y deciden si aceptan el merge o no.

Los pull request no son una función de Git, son una función de Github.

**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

**El flujo del pull request es el siguiente:**

1. Se trabaja en una rama paralela los cambios que se desean (git checkout -b <rama>)
2. Se hace un commit a la rama (git commit -am '<Comentario>')
3. Se suben al remoto los cambios (git push origin <rama>)
4. En GitHub se hace el pull request comparando la rama master con la rama del fix.
5. Uno, o varios colaboradores revisan que el código sea correcto y dan feedback (en el chat del pull request)
6. 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)
7. Se aceptan los cambios en GitHub
8. Se hace merge a master desde GitHub

Importante: Cuando se modifica una rama, también se modifica el pull request.

**FORKS**

En Github, cuando no hacemos parte de los colaboradores directos de un proyecto, pero aun así queremos realizar aportes a estos, podemos generar lo que se llama fork. Un fork es la creación de una nueva división de un proyecto que prodremos utilizar como un nuevo repositorio propio, este repositorio es una copia casi exacta del original en el que podremos hacer cambios. Nosotros no podemos hacer cambio al proyecto original, pero como github sabe de donde se ha creado este fork, nos permite la creación de pull request desde el repositorio creado por el fork al repositorio original.

Este nuevo repositorio no se actualizará con los nuevos cambios que se realicen en el original, sino que debemos agregar un nuevo remote en nuestro repositorio local para que así podamos hacer pull de estos cambios.

git remote add <nombre\_del\_remoto> <url\_del\_remoto> git remote upstream https://github.**com**/freddier/hyperblog

**DEPLOY:** Un deploy es el proceso que permite enviar a un servidor uno o varios archivos, este servidor puede ser uno de prueba, uno de desarrollo o uno de producción.

**.GITIGNORE**

No todos los archivos que utilicemos en nuestros proyectos se deberían subir a un repositorio. Esto se debe a que existen archivos que no todas las personas deberían ver, además de que hay archivos que ralentizan procesos de desarrollo (por ejemplos los archivos binarios).

Para que no se suban archivos no deseados a un repositorio debemos agregar un archivo llamado .gitignore en nuestro repositorio, en este archivo vamos a poner nuestras reglas para los archivos que no se deben subir.

**REGLAS:**

* Ignorar un archivo en específico: nombre\_del\_archivo
* Ignorar todos los archivos con una extensión específica: \*.extension
* Ignorar todos los archivos en un directorio específico: nombre\_del\_directorio/
* Ignorar todos los archivos en un directorio específico y sus subdirectorios: nombre\_del\_directorio/\*\*
* Ignorar un patrón de caracteres específico en cualquier lugar del repositorio: patron\_de\_caracteres/
* Ignorar todas las extensiones de un tipo de archivo específico en cualquier lugar del repositorio: \*.extension
* Ignorar todos los archivos y directorios en la raíz del repositorio excepto uno o varios:

/\*

!nombre\_del\_archivo\_permitido

* Ignorar archivos generados automáticamente, como archivos de compilación o registros:

\*.log

\*.exe

\*.class

* Ignorar directorios de dependencias o módulos:

/node\_modules/

/vendor/

* Ignorar archivos que contienen información sensible o configuración local:

config.ini

secrets.json

**README.md**

Contar con un archivo README.md en nuestro repositorio es una excelente práctica, ya que en este archivo podemos dar un contexto e introducción sobre nuestro repositorio.

La siguiente pagina nos sirve de ayuda para crear un buen readme, en ella tenemos muchas opciones que nos permiten escribir código automáticamente además de mostrar cómo se verá nuestro readmi. <https://pandao.github.io/editor.md/en.html>

**GITHUB PAGES**

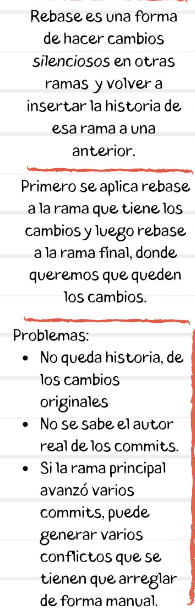
Github cuenta con un servicio de hosting gratis llamado github pages, que nos permite la creación de un sitio web a partir de algún repositorio que tengamos alojado en github.

Para crear dicho sitio web lo que debemos hacer es lo siguiente:

* Crear un repositorio publico donde alojemos nuestro proyecto.
* Asegurarnos de que nuestro html principal lleve como nombre “index” para que pueda ser detectado como la página principal.
* Dirigirnos a la sección de settings en nuestro repositorio y entrar donde dice “pages”.
* Configuramos la rama de la que queremos que se muestre nuestra página.

**GIT REBASE**

rebase es un comando de git que nos permite combinar el contenido de dos ramas reescribiendo la historia del repositorio.



**IMPORTANTE:** Usar rebase es una mala práctica, esto se debe a que al reescribir la historia de nuestro repositorio pueden llegar a surgir errores que son demasiado difíciles de corregir, y mas si se usa en ramas compartidas.

Si se quiere usar rebase es mejor solo hacerlo de forma local

**COMANDOS A TENER EN CUENTA:**

* **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