Comandos para mover archivos entre los estados de Git:

* $ git init - Inicializa un repositorio local.
* $ 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).
* $ 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.
* $ git reset. 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 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. PELIGRO

Mis anotaciones

#To config the user

$ git config --global [user.name](http://user.name/) “johan7perez”  
$ git config --global user.email ["johan7perez@gmail.com](mailto:%22johan7perez@gmail.com)"

#To see the current config  
$ git config --list

#To see where the git configuration is stored  
$ git config --list --show-origin

#To start a new empty project with git  
$ git init

#To see the status of the current project  
$ git status

#To add a new file/modificated file to the staging area  
$ git add

#To remove a new file/modificated file from the staging area (cached, remove the file from the ram memory also)  
$ git rm --cached

#To see the history of a file  
$ git show

#To show the log of the commit or a file, stat is to see the changes at a bytes level  
$ git log [–stat]

#To confirm the changes that we have in the staging area  
$ git commit -m “Message”

#To see the differences of two files / if we do not pass any flag it will compare the current directory and the staging area  
$ git diff [commit file code] [another commit file code]

#To back in the time, if it’s hard will delete everithing made, but if it’s not, will treat the file as a modificated/untracked file  
$ git reset [commit code] [–hard]

#To see back in the pass and see the status of a file / for leave the file in the original state just use ‘git checkout master’  
$ git checkout [commit code] [file name]

**git init:** crea un área en memoria RAM que se llama staging en donde se agregan todos los cambios del proyecto y se crea un repositorio.  
º **git add:** una vez realizados los cambios, se agregan a staging con este comando y el archivo o proyecto pasa a vivir al staging.  
º **git commit -m**: agrega el archivo o proyecto a la carpeta repositorio llamada master.  
º **git config:** muestra todas las configuraciones que tiene git.  
º **git config --list:** nos permite ver toda la configuración por defecto de nuestro git.  
º \*\* git config --list --show --orgin: **permite ver en donde están las configuraciones guardadas.  
º git config --global: permite hacer cambios a nuestro git, como cambiar todos los usuarios.  
º git config --global**[**user.name**](http://user.name/)**nombreUser: permite cambiar el nombre del usuario.  
º git config --global user.email Email: permite cambiar el correo del usuario.  
º git log: muestra toda la historia del archivo o proyecto modificado, muestran los datos de quien lo modifico.  
º git show: muestra los cambios que hemos realizado al archivo.  
º git diff: permite comparar entre las versiones de los archivos modificados.  
º git reset (aquí va la cadena de la version) --hard: vuelve todo a la versión anterior.  
º git reset (aquí va la cadena de la version) --soft: vuelve todo a la versión anterior, pero lo que tenemos en staging sigue ahi.  
º git clone url: permite traer los datos desde un servidor remoto.  
º git push: envia la version lista al repositorio remoto.  
º** git fetch:\*\* nos permite traer una actualización después de clonarlo, pero no lo copea en los archivos. Para poder copiarlo en los archivos tenemos que fusionar la ultima versión que esta en el repositorio local con la versión actual.  
º **git merge:** nos permite hacer la fusión de las versiones de los archivos de las ramas de master.  
º **git pull:** copia el repositorio local, la bd de cambios y copia el directorio.  
**git commit -am:** automáticamente realiza un git add de los cambios pero solo funciona con archivos a los que ya anteriormente se les hizo add. PARA LOS ARCHIVOS NUEVOS NO SE UTILIZA ESTE COMANDO.

**CREAR RAMAS EN MASTER**  
1 º git branch nombreRama: crea la rama  
2 º git checkout nombreRama: este nos permite mover a la rama creada.

**FUSIONAR MASTER CON OTRA RAMA**  
Esto es ara fusionar los cambios de una rama a master.  
1º Si estamos en otra rama nos tenemos que mover a master con el comando **git checkout master.**  
2º **git merge nombreRama:** fusiona a master los cambios realizados en otra rama.

<h1>Curso de Git en Platzi</h1>

Comandos para mover archivos entre los estados de Git:

* $ git init - Inicializa un repositorio local.
* $ 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).
* $ 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.
* $ git reset. 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 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. PELIGRO puede ser peligroso
* $ 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 indiquemos después del comando).

Principales

Repositorio sin REDME

$ git init - Inicializa un repositorio local

$ git remote add origin (url)

$ 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

$ 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 push (origin) (master) (Sube Cambios)

Repositorio con REDME

$ git init - Inicializa un repositorio local

$ git remote add origin (url)

$ git remote -v(es verval)

$ git pull origin master --allow-unrelated-histories fusiona lo que viene con lo que tengo

$ git pull origin master (Descarga cambios)

$ git status

$ git commit

$ git push (origin) (master) (Sube Cambios)

Unirse a un repositorio(Clonar)

Se debe dirigir a la carpeta donde se quiera clonar

git clone url

Al clonar el repositorio automáticamente traer todos los archivos que contiene

Para agregar cosas al repositorio:

git status

git add .

git commit -m “Comit texto”

git push -u origin master

Nota: al momento de hacer push si el usuario no esta agregado al repositorio dará error

Otros

$ git log muestra información y historial de los commits

$ git diff: permite comparar entre las versiones de los archivos modificados

$ 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 reset nombredelcomit --hard vuelve al estado del commit deseado borra lo nuevo para volver al ultimo commit.

$ 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)

$ 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

$ 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. PELIGRO puede ser peligroso

$ 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 indiquemos después del comando)

CREAR RAMAS EN MASTER  
1 º $ git branch nombreRama: crea la rama  
2 º $ git checkout nombreRama: este nos permite mover a la rama creada.

$ git commit -am “Mensaje” Para crear commits dentro de la rama

FUSIONAR MASTER CON OTRA RAMA  
Esto es para fusionar los cambios de una rama a master.  
1º Si estamos en otra rama nos tenemos que mover a master con el comando git checkout master.  
2º git merge nombreRama: fusiona a master los cambios realizados en otra rama.

Si da error se deben hacer los cambios luego se hace un comit se suben los cambios y luego se vuleve a hacer el merge.

En gitlab

Se debe dirigir a la carpeta donde se quiera clonar

git clone url

Se dirige a la carpeta creada por git

git add .

git commit -m “Comit texto”

git push -u origin master

Para clonar un proyecto compartido

Se va a la carpeta donde se quiera guardar

git clone url

cd nombreproyecto

git checkout develop

git pull

Llaves públicas y privadas

En la carpeta base que se abre al abrir el gitbach

ssh-keygen -t rsa -b 4096 -C “[youremail@example.com](mailto:youremail@example.com)”

luego se le asigna una contraseña

Posterior se verifica que el archivo .ssh se creo

Se copia la llave publica

Se dirige a la consola

Se introduce el comando eval $(ssh-agent - s) que verifica que el servidor ssh este activo

ssh-add ~/.ssh/id\_rsa Agregar la llave privada al sistema o servidor

Luego se copia la llave publica, nos dirigimos al github en la sección de serttings profile buscamos las llaves ssh se crea una nueva y se pega la llave publica

Posteriormente nos dirigimos al repositorio deseado y copiamos el ssh

En consola escribimos el comando al proyecto:

Git remote -v visualizamos la url del proyecto

Git remote set-url origin proyectosshcopiadoDelGitHub

Nota:

Si quieres eliminar las todo lo anterior se debe dirigir a la carpeta creada .ssh eliminarla y posteriormente ir a github y eliminar de igual manera la llave

Tags y versiones de git

Git log muestra los commits

Git log –all muestra todos los commits

Git log –all –graph –decorate –oneline muestra toda la historia del proyecto de manera bonita

Alias nombreAlias=”Comando que deasea guardar con alisas”

Ejemplo: alias arbolito=” git log –all –graph –decorate –oneline”

Para la creación de un tag

Git tag -a versionDelTag -m “Mensaje deseado del tag” nombreDelCommit

Ejemplo: git tag -a v0.1 -m "Resultado de las primeras clases del curso" 9f98e8f

git show-ref –tags ver los tags creados

git status

git pull origin master

git push origin –tags mandar a git los tags creados

git tag -d Dormido eliminar tag

git pull origin master

git push origin :refs/tags/Dormido eliminar tag definitivamente

Branch

Git show-branch

Git show-branch –all

Gitk abre toda la historia de las ramas con toda la informacios de los cambios

Subir ramas a github

Git Branch

Git pull

Git push origin cabecera

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.

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

El flujo del pull request es el siguiente:

* 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

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

Forks o Bifurcaciones

Es una característica única de GitHub en la que se crea una copia exacta del estado actual de un repositorio directamente en GitHub, éste repositorio podrá servir como otro origen y se podrá clonar (como cualquier otro repositorio), en pocas palabras, lo podremos utilizar como un git cualquiera.

Un fork es como una bifurcación del repositorio completo, tiene una historia en común, pero de repente se bifurca y pueden variar los cambios, ya que ambos proyectos podrán ser modificados en paralelo y para estar al día un colaborador tendrá que estar actualizando su fork con la información del original.

Al hacer un fork de un poryecto en GitHub, te conviertes en dueñ@ del repositorio fork, puedes trabajar en éste con todos los permisos, pero es un repositorio completamente diferente que el original, teniendo alguna historia en común.

Los forks son importantes porque es la manera en la que funciona el open source, ya que, una persona puede no ser colaborador de un proyecto, pero puede contribuír al mismo, haciendo mejor software que pueda ser utilizado por cualquiera.

Al hacer un fork, GitHub sabe que se hizo el fork del proyecto, por lo que se le permite al colaborador hacer pull request desde su repositorio propio.

Trabajando con más de 1 repositorio remoto

Cuando trabajas en un proyecto que existe en diferentes repositorios remotos (normalmente a causa de un fork) es muy probable que desees poder trabajar con ambos repositorios, para ésto puedes crear un remoto adicional desde consola.

git remote add <nombre\_del\_remoto> <url\_del\_remoto>

git remote upstream <https://github.com/freddier/hyperblog>

Al crear un remoto adicional podremos, hacer pull desde el nuevo origen (en caso de tener permisos podremos hacer fetch y push)

git pull <remoto> <rama>

git pull upstream master

Éste pull nos traerá los cambios del remoto, por lo que se estará al día en el proyecto, el flujo de trabajo cambia, en adelante se estará trabajando haciendo pull desde el upstream y push al origin para pasar a hacer pull request.

git pull upstream master

git push origin master

Agregando mi nombre al repositorio de platzi

Se hace un fork del repositorio

Se clona el repositorio creado en mis repositorios

Se hacen los cambios

Se hace commit

Se hace push

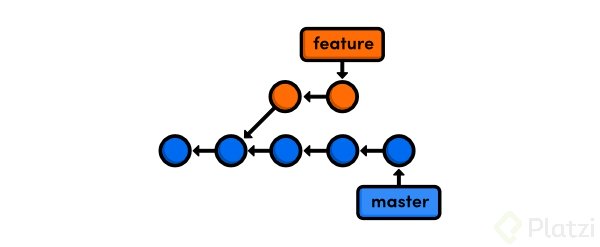
En github crearmos un pull request

Git ignore

Se crea como un archivo .gitignore y dentro de agregan las extensiones de los archivos que no se deben subir al git.

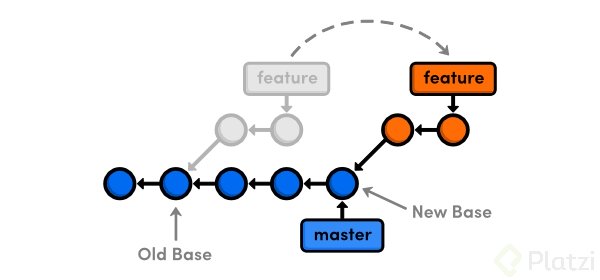
REDME

<https://pandao.github.io/editor.md/en.html> Esta url redirecciona a una pagina donde es facil crear el formato de un redme.

Rebase  
Es el proceso de mover o combinar una secuencia de confirmaciones en una nueva confirmación base. La reorganización es muy útil y se visualiza fácilmente en el contexto de un flujo de trabajo de ramas de funciones. El proceso general se puede visualizar de la siguiente manera.  
  
Para hacer un rebase en la rama feature de la rama master, correrías los siguientes comandos:

git checkout feature

git rebase **master**

Esto trasplanta la rama feature desde su locación actual hacia la punta de la rama master:  


Ahora, falta fusionar la rama feature con la rama master

git checkout **master**

**git** rebase feature

**# No reorganices el historial público**

Nunca debes reorganizar las confirmaciones una vez que se hayan enviado a un repositorio público. La reorganización sustituiría las confirmaciones antiguas por las nuevas y parecería que esa parte del historial de tu proyecto se hubiera desvanecido de repente.

## Stashed:

El stashed nos sirve para guardar cambios para después, Es una lista de estados que nos guarda algunos cambios que hicimos en Staging para poder cambiar de rama sin perder el trabajo que todavía no guardamos en un commit

Ésto es especialmente útil porque hay veces que no se permite cambiar de rama, ésto porque porque tenemos cambios sin guardar, no siempre es un cambio lo suficientemente bueno como para hacer un commit, pero no queremos perder ese código en el que estuvimos trabajando.

El stashed nos permite cambiar de ramas, hacer cambios, trabajar en otras cosas y, más adelante, retomar el trabajo con los archivos que teníamos en Staging pero que podemos recuperar ya que los guardamos en el Stash.

## git stash

El comando git stash guarda el trabajo actual del Staging en una lista diseñada para ser temporal llamada Stash, para que pueda ser recuperado en el futuro.

Para agregar los cambios al stash se utiliza el comando:

git stash

Podemos poner un mensaje en el stash, para asi diferenciarlos en git stash list por si tenemos varios elementos en el stash. Ésto con:

git stash save "mensaje identificador del elemento del stashed"

## Obtener elelmentos del stash

El stashed se comporta como una [Stack](https://es.wikipedia.org/wiki/Pila_(inform%C3%A1tica)" \t "_blank) de datos comportándose de manera tipo [LIFO](https://es.wikipedia.org/wiki/LIFO) (del inglés Last In, First Out, «último en entrar, primero en salir»), así podemos acceder al método pop.

El método **pop** recuperará y sacará de la lista el **último estado del stashed** y lo insertará en el **staging area**, por lo que es importante saber en qué branch te encuentras para poder recuperarlo, ya que el stash será **agnóstico a la rama o estado en el que te encuentres**, siempre recuperará los cambios que hiciste en el lugar que lo llamas.

Para recuperar los últimos cambios desde el stash a tu staging area utiliza el comando:

git stash pop

Para aplicar los cambios de un stash específico y eliminarlo del stash:

git stash pop stash@{<num\_stash>}

Para retomar los cambios de una posición específica del Stash puedes utilizar el comando:

git stash apply stash@{<num\_stash>}

Donde el <num\_stash> lo obtienes desden el git stash list

## Listado de elementos en el stash

Para ver la lista de cambios guardados en Stash y así poder recuperarlos o hacer algo con ellos podemos utilizar el comando:

git stash list

Retomar los cambios de una posición específica del Stash || Aplica los cambios de un stash específico

## Crear una rama con el stash

Para crear una rama y aplicar el stash mas reciente podemos utilizar el comando

git stash branch <nombre\_de\_la\_rama>

Si deseas crear una rama y aplicar un stash específico (obtenido desde git stash list) puedes utilizar el comando:

git stash branch nombre\_de\_rama stash@{<num\_stash>}

Al utilizar estos comandos **crearás una rama** con el nombre <nombre\_de\_la\_rama>, te pasarás a ella y tendrás el **stash especificado** en tu **staging area**.

## Eliminar elementos del stash

Para eliminar los cambios más recientes dentro del stash (el elemento 0), podemos utilizar el comando:

git stash drop

Pero si en cambio conoces el índice del stash que quieres borrar (mediante git stash list) puedes utilizar el comando:

git stash drop stash@{<num\_stash>}

Donde el <num\_stash> es el índice del cambio guardado.

Si en cambio deseas eliminar todos los elementos del stash, puedes utilizar:

git stash clear

Consideraciones:

* El cambio más reciente (al crear un stash) **SIEMPRE** recibe el valor 0 y los que estaban antes aumentan su valor.
* Al crear un stash tomará los archivos que han sido modificados y eliminados. Para que tome un archivo creado es necesario agregarlo al Staging Area con git add [nombre\_archivo] con la intención de que git tenga un seguimiento de ese archivo, o también utilizando el comando git stash -u (que guardará en el stash los archivos que no estén en el staging).
* Al aplicar un stash este no se elimina, es buena práctica eliminarlo.

Git Clean

git clean –dry-run simula y muestra los archivos que va a eliminar con el comando

git clean -f borra los archivos basura

git clean -df borra los archivos basura y también las carpetas.

Git cherry-pick:

1. Crear una nueva rama y pasarse para esa rama.
2. Realizar los cambios en el código y realizar todos los commits necesarios.
3. Antes de ejecutar el comando git cherry-pick, se debe anotar el id del commit que queremos traer al master.
4. Para ello vamos a la rama donde esta el commit requerido (git checkout nombre\_rama), ejecutamos el comando git log --oneline,  
   copiamos el id del commit y nos regresamos a la rama master (git  
   checkout master).
5. En la rama master, ejecutamos: git cherry-pick id\_commit.
6. Git agregará al master el código requerido del commit indicado.
7. Después, de ser necesario, se ejecutan los comandos git pull  
   origin master y git push origin master para subir los cambios al  
   repositorio de github en la nube.
8. Si se necesitan traer todos los cambios de la rama de donde tomé el commit, se ejecuta desde master, git merge nombre\_rama.
9. De haber conflictos al hacer merge, se resuelven dejando las líneas de código necesarias o utilizando las opciones de Vscode.

Git amend:

Se utiliza cuando se hace un commit pero necesita agregarle algo mas a ese mismo commit

Se hacen los cambios olvidados y luego se usa el comando

git add -A Para hacer uso de ammend los archivos deben de estar en staging

git commit --amend Remendar último commit.

Git Reset y Reflog

git reflog muestra todo el historial de lo que ha pasado

git reset --hard eff544f vuelve al estado de un commit en especifico esto se usa en caso de que la cagaste y quieres recuperar archivos borrados o volver hacia atrás en el tiempo

Git con Grep y log

git grep color -->use la palabra color

git grep la --> donde use la palabra la

git grep -n color–> en que lineas use la palabra color

git grep -n platzi --> en que lineas use la palabra platzi

git grep -c la --> cuantas veces use la palabra la

git grep -c paltzi --> cuantas veces use la palabra platzi

git grep -c “<p>”–> cuantas veces use la etiqueta <p>

git log -S “cabecera” --> cuantas veces use la palabra cabecera en

todos los commits.

grep–> para los archivos

log --> para los commits.

Git otros comandos

git shortlog -sn = muestra cuantos commit han hecho cada miembros del equipo.

git shortlog -sn --all = muestra cuantos commit han hecho cada miembros del equipo hasta los que han sido eliminado

git shortlog -sn --all --no-merge = muestra cuantos commit han hecho cada miembros quitando los eliminados sin los merges

git blame ARCHIVO = muestra quien hizo cada cosa linea por linea

git COMANDO --help = muestra como funciona el comando.

git blame ARCHIVO -Llinea\_inicial,linea\_final= muestra quien hizo cada cosa linea por linea indicándole desde que linea ver ejemplo -L35,50

\*\*git branch -r \*\*= se muestran todas las ramas remotas

git branch -a = se muestran todas las ramas tanto locales como remotas

**Común WORKFLOW**

La manera en como normalmente se trabaja dentro de un proyecto es haciendo una nueva Branch por cada tarea o ticket

Al aceptar una tarea por realizar se crea una nueva Branch con el siguiente comando:

Git checkout -b mynewbranch

Ahí se trabaja y luego cuando la tarea fue terminada se verifica que los cambios no afecten a la rama madre para poder hacer un merge request

Esto lo hacemos haciendo un merge de la rama madre en la rama donde trabajamos con el siguiente comando

Git merge ramamadre

Ahí en el editor se abreran los conflictos en caso de que los haya o si todo esta bien no mostrara error

Posterior a el paso anterior verificamos que el gitignore contenga los datos que no deben ser subidos a git

Y por ultimo se hace un commit y luego un push