

1. CURSO PROFESIONAL DE GIT Y GITHUB

Introducción a Git / Introducción a la línea de comandos.

`~` : hogar de "Mis documentos"

`pwd`: Nos da la dirección en donde estamos

`cd`: para navegar en carpeta.
`cd [nombre_ruta]`

`ls` : listar archivos en la raíz de linux emulado.

`ls -al`: argumento que mostrará todos los archivos ocultos en forma de lista.

`ls -a` : muestra archivos ocultos de forma desordenada.

`ls -l` : muestra los archivos de la ruta en forma de lista.

`clear`: limpia la consola.

`mkdir`: crear carpeta.
`mkdir [nombre_carpeta]`

`touch`: crea un archivo vacío.
`touch [nombre_archivo.tipo]`

`cat`: muestra rápido el contenido de un archivo.
`cat [nombre_archivo.tipo]`

`history`: nos muestra el historial de comandos.
para repetir un comando dado en history:
`![numero_del_comando]`

`rm`: elimina archivo
`rm [nombre_del_archivo]`

Git Bash es de Linux. No existe la ruta como en Windows: `C:\Users\nombre` este se cambia por: `/C/Users/nombre`

Para cambiar de directorio en Git Bash, se usa el comando `cd /` esto nos envía a la raíz del disco.

Al utilizar el comando `ls` dentro de una ruta, nos mostrará todos los archivos que existen dentro de él.

Para entrar en el disco en el cual se encuentra nuestra partición utilizaremos `cd /c` de esa forma entraremos en el disco C.

Para entrar en la carpeta usuarios se utiliza el siguiente comando `cd Users` cabe destacar que en Windows no importa si se usan mayus. o minusc. mientras se ingrese el `nombre_carpeta` correcta.

Para volver a la carpeta anterior, es necesario usar el comando `cd ..` (esto nos regresa a la unidad C).

Nota: al usar una letra+[TAB], nos agrega la ruta con auto llenado.

`cd` siempre nos va a redireccionar al HOME (`/C/Users/nombre`).

al ver las carpetas dentro del directorio con `ls` hay unas que terminan en un "." o ".." un punto significa en la carpeta que estamos, dos puntos es la carpeta anterior a la que estamos.

para obtener una ayuda del uso de los comandos se puede utilizar el siguiente comando: `[comando] --help`

1. CURSO PROFESIONAL DE GIT Y GITHUB

Comandos básicos de Git / Staging y Repositorios en Git.

`git init`: inicializa git en una carpeta.

`git add`: añade archivos al staging (vive en staging).

`git add .` (añade todos los cambios que hay en la carpeta).

`git add [nombre_archivo.tipo]` (añade cambios dentro del documento)

`commit`: manda el archivo al repositorio

`commit -m "mensaje_commit"`

`git rm`: remueve el archivo que se encuentra en Staging.

Al usar el comando **git init** se crean 2 cosas: un área de staging y se crea también el repositorio.

Staging: es el área en la RAM donde se guardarán los cambios, antes de ser enviados al repositorio.

Repositorio: Es el lugar donde se almacenan todos los cambios que se han realizado.

La carpeta `.git` que se crea al inicializar **git init** dentro de nuestra carpeta, es el Repositorio.

El nombre por defecto de un repositorio siempre será **MASTER**.

MASTER: se encuentran todos los cambios que se hagan dentro del repositorio.

Para subir un archivo, se utiliza el comando **git add**, al usar este comando nuestro archivo estará en **staging**. Este no llega hasta el repositorio hasta realizar un **commit**.

Cada **commit** es una nueva versión de cambios dentro del repositorio.

Los números aleatorios que se generan al hacer un **commit** son el nombre interno en la base de datos de git.

1. CURSO PROFESIONAL DE GIT Y GITHUB

Comandos básicos de Git / ¿Qué son Branch y Merge?

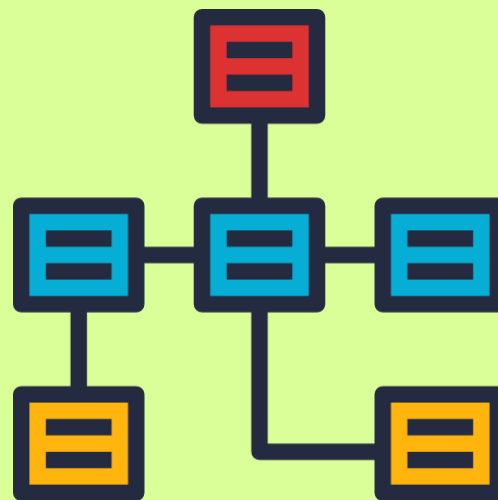
branch: ramas dentro de git.

Merge: unir cambios de una rama a la rama actual o a otra rama.

Siempre se trabaja sobre la rama **MASTER** del repositorio, pero también se pueden crear distintas ramas, para trabajar en paralelo al **MASTER** sin alterar el proyecto original.

Para solucionar bugs de forma rápida se llaman hotfix, a esa rama normalmente le llaman **bugfixing**.

A la rama de experimentos se le llama **development**.



1. CURSO PROFESIONAL DE GIT Y GITHUB

Comandos básicos de Git / Creando Repositorio y commit.

`code`: abre VSCODE.

`code [nombre_archivo.tipo]`

*sirve para abrir un archivo específico.

`git status`: visualiza el estado en que se encuentra el proyecto en git.

`rm --cached`: elimina el proyecto del staging, pero el archivo sigue existiendo.

`git config`: muestra los comandos que existen en git.

`git config --list`: muestra los comandos por defecto que están configurados en nuestro git.

`git config --list --show-origin`: muestra donde está el origen de los datos configurados.

`git config --global user.name [nombre_usuario]`
*sirve para ingresar un usuario en git.

`git config --global user.email [correo_electronico]`
*sirve para ingresar un correo electrónico en git.

`git log`: historial de commits hechos en el proyecto.
`git log [nombre]`

Para crear un repositorio debemos estar en la carpeta en la cual deseamos iniciar un repositorio y utilizar el comando **git init**.

Git Bash, tiene la facilidad de usar el comando **code** para abrir rápidamente **Visual Studio Code**.

git status, nos muestra 3 cosas distintas:

- ◆ Cuando existen cambios dentro del proyecto (los cambios se muestran en color **rojo**).
- ◆ Cuando ya se ha hecho un **git add** al proyecto y se encuentra el archivo en staging (los archivos añadidos se encuentran en color **verde**, esto significa que aún no están los proyectos en el repositorio y que se requiere hacer un **commit**).
- ◆ Cuando no existen cambios dentro del proyecto que añadir. (aquí no se muestra nada pues todos los archivos ya están cargados en el repositorio).

Para poder guardar un **commit** es necesario tener un usuario y correo electrónico dado de alta en git.

Todos los commit mostrados en el comando **git log** contienen un número, ese es su **TAG** de referencia para poder movernos entre versiones.

La versión (**HEAD -> MASTER**) eso significa que es la versión más reciente del proyecto guardado en el repositorio.

1. CURSO PROFESIONAL DE GIT Y GITHUB

Comandos básicos de Git / Analizando cambios en proyecto.

git show (archivo): muestra las modificaciones internas por líneas.

git show [nombre_archivo.tipo]

git diff: sirve para comparar 2 versiones de commit que se elijan.

git diff [código_commit1]
[código_commit2]

*Los Códigos para comparar se obtienen de git log.

ESC + SHIFT + Z + Z: para guardar lo escrito en VIM y volver a git bash.

ESC + i : para insertar texto dentro del editor de texto VIM.

git show es un comando que nos muestra los cambios realizados en el documento en forma de líneas, como se muestra de la siguiente manera:

```
$ git show hey.txt
commit 9618115887e27fe072415e0e1568673ec06e256b (HEAD -> master)
Author: Loreli Cervantes <54919465+Lorelicervantes@users.noreply.github.com>
Date: Sun May 24 15:33:04 2020 -0500
diff --git a/hey.txt b/hey.txt
index 053e4d9..1aa959100644
--- a/hey.txt
+++ b/hey.txt
@@ -1,3 +1,5 @@
 Esta es mi historia.
 Soy Loreli Cervantes y tengo 24 años.
-He nacido 564 veces...
 \ No newline at end of file
+He nacido 564 veces...
+
+De un día a otro.
 \ No newline at end of file
```

Al no añadir ningún mensaje al commit este nos mandará al editor de texto de Vim, para que se añada un mensaje de forma obligatoria.

***UN COMMIT JAMÁS DEBE ENVIARSE VACÍO.**

Para salir del editor y volver al Git Bash, es importante usar un juego de teclas, pues así está definido en Linux ESC + SHIFT + Z + Z para poder salir.

El comando git diff, sirve para comparar versiones, el primer código que se ingresa mostrará la diferencia con el segundo, es decir depende el orden se muestran las diferencias.

```
lorel@LAPTOP-IDHRDSU3 MINGW64 ~/prb (master)
$ git diff 3d180d60884756c18fafd5e3b3a9d742d3867b90 d6409c9ef9b65a256362ae7059a762fcc9d9c8c1
diff --git a/blog.html b/blog.html
new file mode 100644
+++ b/css/style.css
@@ -0,0 +1,4 @@
+body{
+  text-align: center;
+  font-family: 'Franklin Gothic Medium', 'Arial Narrow', Arial, sans-serif;
+}
 \ No newline at end of file
diff --git a/hey.txt b/hey.txt
index 8dc5995..4250d33 100644
--- a/hey.txt
+++ b/hey.txt
@@ -1 +1,5 @@
-Esta es mi historia.
 \ No newline at end of file
+Esta es mi historia.
+Soy Loreli Cervantes y tengo 24 años.
+He nacido 564 veces...
+
+De un día a otro, todo se convierte en color azul.
```

1. CURSO PROFESIONAL DE GIT Y GITHUB

Comandos básicos de Git / Usando Branch y Checkout.

git reset: nos permite volver a una versión anterior.

git reset [código_commit] --hard.
git reset [código_commit] --soft.

git diff: cambios entre repo actual y staging.

git status: nos muestra la vista de staging.

git log --stat: cambios específicos en archivos de un commit a otro mostrado en bits.

q: nos saca del (END) en git bash.

git checkout [#commit]
[nombre_archivo]: nos muestra la versión del commit seleccionado, pero no se almacena a menos que se haga un commit.

git checkout [rama_MASTER]
[nombre_archivo]: nos muestra la versión actual de nuestra rama MASTER del archivo seleccionado.

Contamos con 3 entornos distintos:

- ◆ Directorio de trabajo (nuestra carpeta de proyecto).
- ◆ Staging (Área de memoria RAM donde esperan los documentos a ser enviados al repositorio).
- ◆ Repositorio (donde se guardan todos los cambios que hemos hecho).

Existen 2 tipos de usar el comando **git reset**:

- ◆ git reset [#commit] --hard: nos regresa al estado del commit que seleccionemos, eliminando todos los commits posteriores a la versión seleccionada.
- ◆ git reset [#commit] --soft: nos regresa a la versión del commit seleccionado, pero no elimina lo que se encuentra en staging, aun que se eliminen las versiones posteriores, se guarda el último cambio hecho en staging.

Para volver a una versión anterior se utiliza el comando **checkout** esta nos permite visualizar la versión seleccionada, pero NO SE CAMBIA A ESA VERSIÓN a menos que se realice un commit.

Para volver a la versión actual, solo se ingresa el comando **checkout** seguida de la rama MASTER y el archivo.

con esto se puede modificar el archivo de una versión anterior, sin eliminar o modificar los demás archivos dentro del directorio, es por eso que se selecciona el [nombre_archivo].

1. CURSO PROFESIONAL DE GIT Y GITHUB

Comandos básicos de Git / Git reset vs Git rm.

Traemos de nuevo estos comandos aquí:

`git reset`: nos permite volver a una versión anterior.

`git reset [código_commit] --hard.`

`git reset [código_commit] --soft.`

`git rm --cached`: elimina el proyecto del staging, pero el archivo sigue existiendo.

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. debe utilizarse uno de los siguientes flags:

`git rm --cached`: Elimina los archivos del área de staging y del próximo commit, pero los mantiene en el disco duro.

`git rm --force`: Elimina los archivos de git y del disco duro.

Git reset.

Este comando nos permite 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 sobre escribir, no hay vuelta atrás.

`git reset --hard`: Borramos el historial y los registros de git, pero guardamos los cambios en staging.

`git reset --soft`: Se elimina absolutamente todo.

`git reset HEAD`: Este comando saca los archivos de staging, no para borrarlos, sino para que no se envíen al último commit.

1. CURSO PROFESIONAL DE GIT Y GITHUB

Flujo de trabajo en Git / Flujo con repositorio remoto.

`git clone [url]`: copia del repositorio remoto al repositorio local del proyecto.

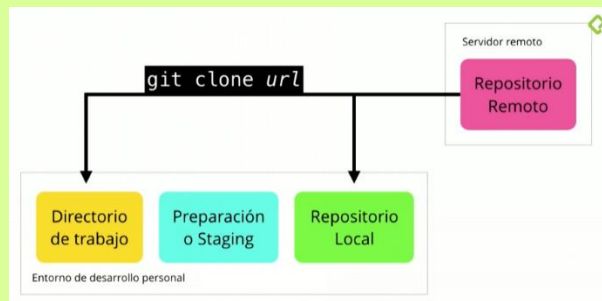
`git push`: envía el guardado en el repositorio local, los envía al repositorio remoto

`git fetch`: actualizaciones del repositorio remoto al repositorio local.

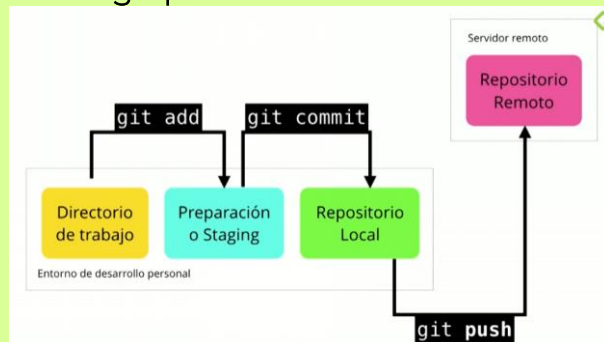
`git pull`: actualiza los cambios hechos en el repositorio remoto y los guarda en el repositorio local y en nuestro directorio de trabajo.

Repositorio remoto: para trabajar con un equipo en un servidor, donde se encuentra la versión del documento. Ejemplo: (GitHUB).

git clone: con el link del repo remoto, trae una copia del master al directorio de carpeta en equipo y crea la base de datos de todos los cambios históricos en el repositorio local.



git push: ultima versión del head en master de nuestro repositorio local sea enviado al repositorio remoto, se utiliza `git push`.



git fetch: sirve para traer actualizaciones hechas en el repositorio remoto hacia el repositorio local.

git pull: funciona `git fetch` y `git merge`, donde además de bajar las actualizaciones al repositorio local, también lo hace en el directorio de nuestro proyecto.



1. CURSO PROFESIONAL DE GIT Y GITHUB

Flujo de trabajo en Git / Introducción a ramas en Git.

git commit -am "mensaje":
esto automáticamente hace el
git add de los cambios, solo a
los archivos que anteriormente
ya les hayamos realizado un git
add.

git commit -a: nos manda a la
consola de VIM para insertar
un add.

git branch [nombre_rama]: nos
crea una rama. No se observa
hasta que se da git show.

git branch cabecera

git checkout [nombre_rama]:
nos envía a la rama
seleccionada.

git checkout cabecera
git checkout MASTER

git status: también nos sirve
para mostrar en que rama
estamos, si en MASTER u en
otra.

git log: nos muestra el historial
de commits.

git branch -D [nombre rama]:
nos permite eliminar una rama

Las **ramas** son formas en las que podemos hacer cambios
sin afectar el MASTER

Las ramas son paralelas y pueden fusionarse.

HEAD <- LUGAR DONDE ESTOY TRABAJANDO

Al crear una rama, no se muestra nada, hasta dar git
show nos dice que el ultimo commit hecho apunta a
master y a cabecera como se ve a continuación:

```
lore1@LAPTOP-IDHRDSU3 MINGW64 ~/prb (master)
$ git show
commit d03d7d83773d76e7f35fc981a226ae8f0644b04b (HEAD -> master, cabecera)
Author: Loreli Cervantes <54919465+Lorelicervantes@users.noreply.github.com>
Date:   Wed May 27 00:11:39 2020 -0500
```

Para movernos del MASTER a la rama creada, en este
caso CABECERA, se utiliza el comando git checkout
como se muestra:

```
lore1@LAPTOP-IDHRDSU3 MINGW64 ~/prb (master)
$ git checkout cabecera
Switched to branch 'cabecera'

lore1@LAPTOP-IDHRDSU3 MINGW64 ~/prb (cabecera)
```

cuando hacemos un commit en una rama distinta a
MASTER, y ponemos el comando git log, nos muestra los
commits, de esta forma:

```
commit 241f206f5b299f4507d572d6aeac863e80822e2f (HEAD -> cabecera)
Author: Loreli Cervantes <54919465+Lorelicervantes@users.noreply.github.com>
Date:   Thu May 28 14:11:45 2020 -0500

    estructura inicial de cabecera

League of Unity
commit d03d7d83773d76e7f35fc981a226ae8f0644b04b (master)
Author: Loreli Cervantes <54919465+Lorelicervantes@users.noreply.github.com>
Date:   Wed May 27 00:11:39 2020 -0500

    Nueva versión del HTML
```

Lo que pasa internamente al crear una rama, es que
copia del MASTER, en la primera imagen vemos como
un commit apunta a (MASTER, CABECERA), lo que
denota que es ahí cuando se crea la rama.

Cuando se realizan cambios, es decir un commit dentro
de la rama, eso ya no se muestra en la rama MASTER.

*El comando git checkout es el más importante, ya que
nos permite movernos entre las diferentes ramas del
proyecto.

1. CURSO PROFESIONAL DE GIT Y GITHUB

Flujo de trabajo en Git / Fusión de ramas con git MERGE.

`git merge [nombre_rama]`: para traer los cambios de esa rama a la rama en la que estás posicionado.

`git branch`: te muestra las ramas que existen en tu proyecto.

Cuando creamos un MERGE, se marca el final de una rama, pues esta se fusiona a la rama MASTER.

*Puede que la rama no se pierda, pero todo el contenido lo guarda en MASTER.

Si realizamos cambios al archivo dentro de cualquier rama y no guardamos cambios, al dar `git checkout`, se pierden todos los cambios realizados en esa rama.

El MERGE siempre ocurre en la rama en la que te encuentras. Ej.

Si tengo la rama `[nombre_rama]`, y la quiero enviar al MASTER, debo posicionarme en la rama MASTER y desde ahí invocar el comando **git merge**.

Al crear un merge en MASTER, internamente se crea un nuevo commit que trae la nueva información de la rama que exporta, se fusionan (el ultimo commit de MASTER y el último commit de `[nombre_rama]`) y es por eso que se crea un nuevo commit.

*A menos que se hayan modificado 2 líneas iguales dentro de la rama MASTER y `[nombre_rama]`, al dar `git merge`, nos creará un conflicto, y no nos permitirá hacer el cambio.

Todos los merge piden un mensaje, ya que se crea un nuevo commit.

Al fusionar una rama al MASTER, se ve de la siguiente forma:

```
lore1@LAPTOP-IDHRDSU3 MINGW64 ~/prb (master)
$ git merge cabecera
Auto-merging blog.html
Merge made by the 'recursive' strategy.
 blog.html      | 3 +++
 css/style.css  | 8 +++++++
 2 files changed, 11 insertions(+)
```

1. CURSO PROFESIONAL DE GIT Y GITHUB

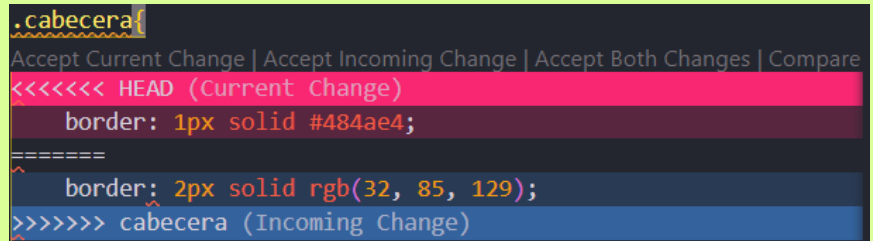
Flujo de trabajo en Git / Solución de conflictos.

Visual Studio Code: nos permite ver los conflictos.

Existen 3 tipos: de cambios, el de la rama actual, la rama de donde exportamos y aceptar ambos cambios.

Cuando 2 usuarios modifican la misma línea de código en distintas ramas, y luego intentan unirlos, se crea un conflicto, ya que git no sabe cuál de las 2 versiones elegir.

Cuando se intenta hacer un merge, marca de esta forma en VSCode:



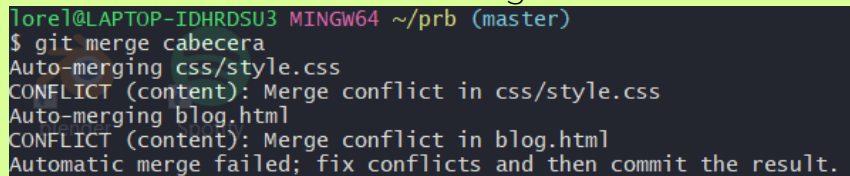
Nos dice que:

<<<<<<En la cabecera actual tenemos esto

=====

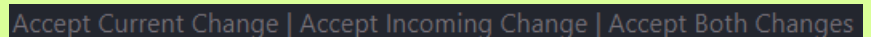
>>>>>>Y en la cabecera esto otro

En consola se muestran de la siguiente forma:



(MASTER|MERGING) nos muestra que estamos en un estado donde no se ha completado el merge, esto está a la espera de solucionar el conflicto seleccionando la versión final y añadiendo un commit.

VSCode, nos permite seleccionar que cambio deseamos hacer para no hacerlo manual, con las siguientes opciones:



1. Aceptar el cambio actual. (rama en que estas)
2. Aceptar el cambio que viene. (rama de la que exportas)
3. Aceptar ambos cambios.

Una vez hecho eso, se debe:

1. Realizar un add y commit de los archivos solucionados con **git commit -am "mensaje"**
2. Revisar con git status que se ha subido el cambio
3. Volver a hacer el merge.

1. CURSO PROFESIONAL DE GIT Y GITHUB

Repositorios Remotos / Uso de GitHub.

New repository: Sirve para crear un repositorio remoto.

READ ME: archivo al entrar al repositorio de github, es una buena práctica hacerlo.

git remote add origin [url_repositorio]: para agregar un origen remoto de nuestros archivos.

git remote: nos muestra un origen remoto,

git remote -v: nos muestra el origen para hacer fetch y para hacer pull.

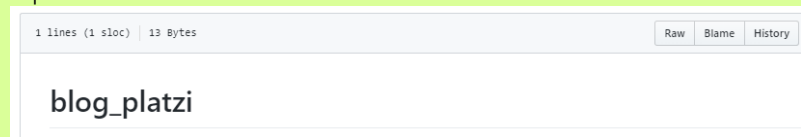
git pull origin [rama_MASTER]: para traer cambios del repositorio remoto a nuestro directorio local.

git pull origin master --allow-unrelated-histories: permite fusionar lo que existe en la rama origin/master con la rama Master local, es decir realiza un MERGE

git push origin [rama_MASTER]: sirve para enviar al origen remoto, todo lo que hay en nuestra rama MASTER.

Al crear un repositorio en GitHub, existe una buena práctica que siempre se debe realizar, que es **inicializar READ ME**.

Dentro del repositorio creado en GitHub nos muestra 3 opciones al dar clic en **READ ME**:



- ◆ Raw: Muestra el código plano de ese archivo.
- ◆ Blame: Muestra quien ha realizado los commits.
- ◆ History: muestra la historia del archivo, similar al hacer un **log**.

El repositorio puede ser de 2 tipos:

- ◆ Público.
- ◆ Privado.

fetch: traer cosas del repositorio remoto.

push: enviar cosas de nuestra rama MASTER al origen del repositorio remoto.

Antes de realizar un push, debemos hacer un pull del repositorio remoto para que puedan almacenarse los cambios en el.

al realizar pull, nos aparece la siguiente información:

```
lorel@LAPTOP-IDHRDSU3 MINGW64 ~/prb (master)
$ git pull origin master
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 605 bytes | 31.00 KiB/s, done.
From https://github.com/Lorelicervantes/blog_platz
 * branch                master       -> FETCH_HEAD
 * [new branch]          master       -> origin/master
fatal: refusing to merge unrelated histories
```

Ya cargada la información podemos realizar un push y se observa de la siguiente manera:

```
lorel@LAPTOP-IDHRDSU3 MINGW64 ~/prb (master)
$ git push origin master
Enumerating objects: 61, done.
Counting objects: 100% (61/61), done.
Delta compression using up to 8 threads
Compressing objects: 100% (49/49), done.
Writing objects: 100% (60/60), 5.68 KiB | 277.00 KiB/s, done.
Total 60 (delta 24), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (24/24), done.
To https://github.com/Lorelicervantes/blog_platz.git
 3dd5292..9707ada master -> master
```

1. CURSO PROFESIONAL DE GIT Y GITHUB

Repositorios Remotos / Uso de GitHub.

New repository: Sirve para crear un repositorio remoto.

READ ME: archivo al entrar al repositorio de github, es una buena práctica hacerlo.

git remote add origin [url_repositorio]: para agregar un origen remoto de nuestros archivos.

git remote: nos muestra un origen remoto,

git remote -v: nos muestra el origen para hacer fetch y para hacer pull.

git pull origin [rama_MASTER]: para traer cambios del repositorio remoto a nuestro directorio local.

git pull origin master --allow-unrelated-histories: permite fusionar lo que existe en la rama origin/master con la rama Master local, es decir realiza un MERGE

git push origin [rama_MASTER]: sirve para enviar al origen remoto, todo lo que hay en nuestra rama MASTER.

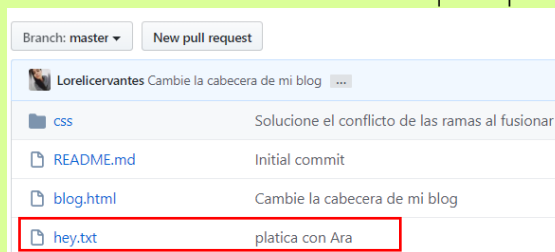
Después de:

1. Crear el repositorio remoto en GitHub.
2. Clonar el repositorio remoto en nuestro repositorio local con git remote add origin [URL].
3. Ingresar nuestras credenciales de identificación de github.
4. Realizar el pull origin master, para traer el repositorio remoto a nuestro repositorio local.
5. Hecho el Merge del origin al master del repositorio local para fusionar los datos.
6. Visualizar que el archivo README.md se encuentra en nuestro repositorio local con ls -al.
7. Realizar un push origin, para enviar todo lo que tenemos en nuestro repositorio local, hacia el repositorio remoto.

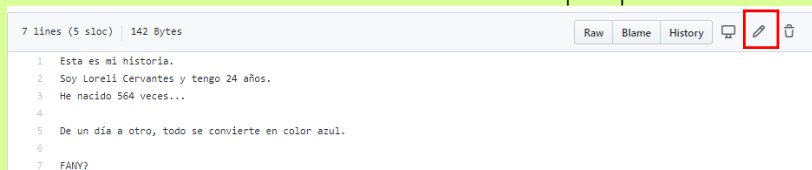
Todo se visualizará en el repositorio remoto en GitHub.

GitHub web, también nos permite realizar modificaciones y commits, de una forma amigable por medio de su interfaz de la siguiente manera:

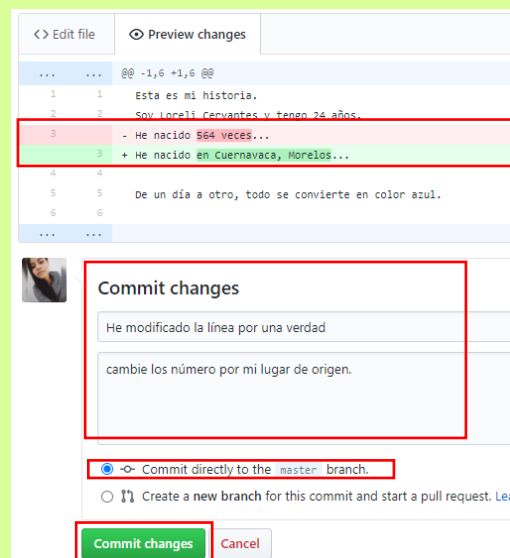
Nos vamos al documento que queremos modificar:



Una vez dentro seleccionamos el lápiz para editar:



Al editar una línea y dar clic en “Preview Changes” nos permite visualizar los cambios y a la vez realizar un commit:



1. CURSO PROFESIONAL DE GIT Y GITHUB

Repositorios Remotos / Llaves públicas y privadas en Git.

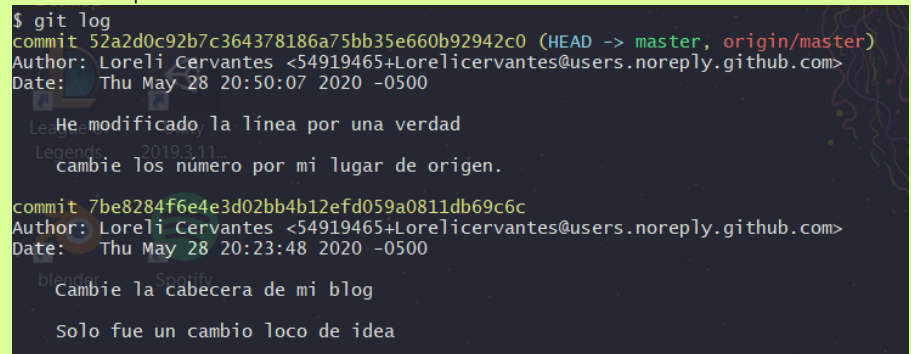
En este apartado, sólo se muestran conceptos básicos acerca de la teoría antes de pasar a la práctica.

Todos los commits realizados en el proyecto local, al fusionarse, también se muestran en el repositorio remoto y viceversa:

En el repositorio Local:



En el repositorio local:



Cifrado simétrico de un solo camino (algoritmo de las llaves públicas y privadas).

Lo que se cifra con la llave pública, solo lo abre la llave privada ya que estas se encuentran vinculadas matemáticamente una con la otra.

Las llaves permiten encriptar mensajes, que solo pueden ser descifrados por la llave pública. Es un cifrado tan poderoso, que así funcionan las finanzas del mundo.

1. CURSO PROFESIONAL DE GIT Y GITHUB

Repositorios Remotos / Configurar llaves privadas SSH local.

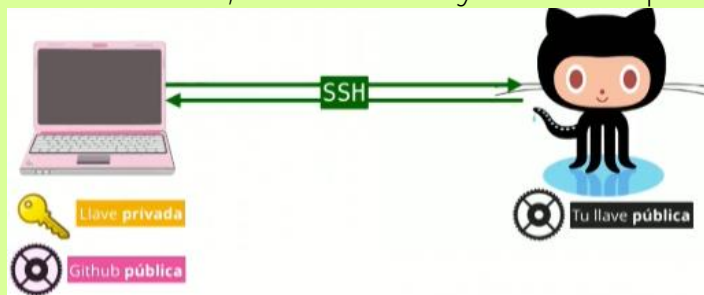
`ssh-keygen -t rsa -b 4096 -C "email"` : especifica algoritmo para usar la llave usando el rsa, que es el más popular, B 4096 es la complejidad de la llave, con C especificamos el correo electrónico al que estará conectado la llave.

Enter passphrase: Nos deja añadir una contraseña con espacios para la seguridad SSL.

Al configurar seguridad SSH en Git y GitHub, la seguridad se vuelve más fuerte y no tendremos que volver a colocar nuestras credenciales de identificación.

Funciona de la siguiente manera: En el repositorio local se crean las llaves privadas y públicas. Al crearlas enviaremos la llave pública a GitHub, para que use la llave privada que tendremos en nuestra computadora, de esa forma podremos conectarnos con github por medio de un protocolo nuevo: **SSH**.

Al enviar la llave publica a Github, Github enviará su propia llave pública con la llave que le enviamos para hacer la conexión, creando así una conexión cifrada de doble camino, entre GitHub y nuestro repositorio local.



A la llave privada que tenemos, podemos añadirle una contraseña para hacerla aún más segura.

Las llaves SSH no son por repositorio, si no por persona.

Para crear la llave SSH se debe hacer:

1. Estar en la carpeta Home de nuestros documentos.
2. Tener el mismo correo en la configuración de Git.
3. Usar el comando para inicializar SSH (lado izq.)
4. Dar Enter para que se guarde en la carpeta que nos muestra.
5. Ingresar una contraseña para tener aún mayor seguridad.

Eso es todo, nos crea el cifrado, nos dice donde se encuentra salvado desde nuestro ordenador, nos da la huella, y una imagen de la llave:

```
$ ssh-keygen -t rsa -b 4096 -C "lorel@lorel.com"
Generating public/private rsa key pair.
Enter file in which to save the key (/c/Users/lorel/.ssh/id_rsa):
Created directory '/c/Users/lorel/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /c/Users/lorel/.ssh/id_rsa
Your public key has been saved in /c/Users/lorel/.ssh/id_rsa.pub
The key fingerprint is:
a1:ad:01:2c:02:10:53:90:80:09:00:01:00:00:00:00 @gmail.com
```


1. CURSO PROFESIONAL DE GIT Y GITHUB

Repositorios Remotos / Configurar llaves privadas SSH local.

`ssh-keygen -t rsa -b 4096 -C "email"` : especifica algoritmo para usar la llave usando el rsa, que es el más popular, B 4096 es la complejidad de la llave, con C especificamos el correo electrónico al que estará conectado la llave.

Enter passphrase: Nos deja añadir una contraseña con espacios para la seguridad SSL.

`eval $(ssh-agent -s)`: con este comando se evalúa que un comando SSH se dispare.

`~` : es una variable que tiene el directorio HOME.

`id_rsa` : esta es la llave privada, la que no se debe compartir con NADIE!!

`id_rsa.pub`: es la llave pública que si podemos compartir.

La imagen se nos muestra de esta forma y ya se encontrará creada:



Se debe revisar que el servidor de llaves SSH se encuentre prendido para ver que estén revisando que las llaves están corriendo y conecte la conexión doble cuando se haga la conexión remota.

Para eso se ejecuta el comando **eval**. el cual nos dará como resultado una línea de comando como esta: **Agent pid #####** (los hashes son números aleatorios del servicio).

Posteriormente, después de verificar que el servidor de cifrado corre bien, debemos agregarla al sistema, para eso veamos los siguientes puntos:

1. Identificar en que carpeta se guardó el `id_rsa`
2. Ingresar el carácter `~` para que nos muestre el directorio.
3. nos dirigimos a la carpeta por medio de este comando `cd ~/.ssh/`
4. podemos ver con el comando `ls -al` que nos encontramos en la carpeta indicada.
5. Para agregar la llave debemos poner el siguiente comando con la ruta donde se encuentra nuestro ssh: `ssh-add ~/.ssh/id_rsa`
6. Si configuraste contraseña, debes añadirla para que pueda ejecutarse.
7. Listo, se mostrará en terminal "Identidad agregada"

1. CURSO PROFESIONAL DE GIT Y GITHUB

Repositorios Remotos / Configuración de GitHub con SSH.

`git remote set-url origin [url_SSH]`: sirve para cambiar la URL, ya que ahora usaremos la de clonar con seguridad SSH.

`git pull`: trae del servidor remoto al repositorio local.

`git push`: envía los cambios del repositorio local al repositorio remoto.

`git diff`: nos muestra las diferencias en cambios realizados.

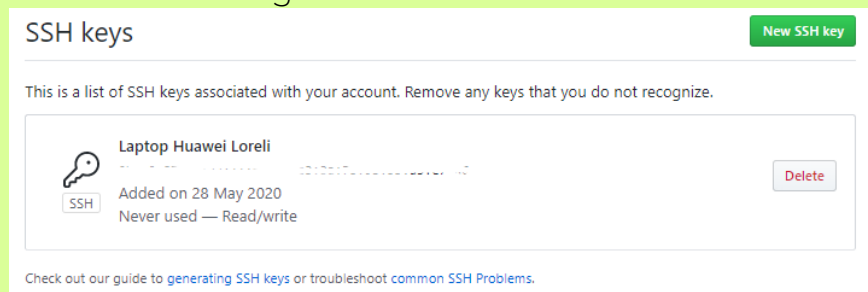
Las llaves ya están creadas en el home de la carpeta de nuestro usuario en nuestra computadora.

*Cada persona debe tener una llave única.

Para configurar GitHub con el SSH, debemos hacer lo siguiente:

1. Copiar lo que hay dentro del archivo `id_rsa.pub`
2. Ir a GitHub > configuración > SSH y GPG keys.
3. Una vez dentro agregaremos una nueva llave SSH
4. Añadimos un título
5. en Key, pegamos la clave que copiamos de `id_rsa.pub`
6. Damos clic en agregar
7. Añadimos nuestra contraseña

Al finalizar nos mandará una pantalla como esta y ya se encontrará configurado.



Ahora podremos clonar nuestro repositorio usando la opción SSH, para ello debemos modificar la URL por la de SSH que nos da en GitHub con el comando:

`git remote set-url origin [url_SSH]`

Eso nos cambiará la URL por una con seguridad de cifrado SSH, al ingresar el comando `git remote -v` observaremos que se han cambiado las URL's.

Después del proceso exitoso, antes de realizar cambios en el servidor local y hacer commit, debemos hacer un **pull al origen** para traer la última versión del servidor remoto:

`git pull origin master`

Al realizar eso, ahora si podremos realizar cambios, commit y un `git push origin master` para enviar esos cambios del repositorio local al servidor remoto de GitHub.

1. CURSO PROFESIONAL DE GIT Y GITHUB

Repositorios Remotos / Tags y versiones en Git y GitHub.

`git log --all`: muestra todos los commits que hay incluso en las ramas.

`git log --all --graph`: nos muestra gráficamente como están las ramas.

`git log --all --graph --decorate -oneline`: muestra lo mismo que graph, pero todo más comprimido.

`alias [nombre_alias] = "comando"`: crea alias para no usar el comando completo.

`git tag -a [nombre_versión] -m "mensaje_commit"`
[hash_commit]: Sirve para crear un TAG

`git tag -a v0.1 -m "Primer bifurcación" d03d7d8`

`git tag`: nos permite ver los TAGS que tenemos.

`git show-ref --tags`: para saber a qué hash está apuntando el tag.

`git push origin --tags`: empuja los tags hacia el origin de GitHub.

`git tag -d nombre_tag`: esto elimina el tag de nuestro directorio local.

`git push origin :refs/tags/nombre_tag`: Elimina los tags que tenemos en GitHub.

Para añadir un alias a algún comando es necesario escribir en la línea de comandos lo siguiente:

`alias [nombre_alias] = "comando"`

Entonces para ejecutar esos comandos, basta solo escribir el **alias** que ingresamos en ese momento.

Un tag sirve para crear un atajo a algún commit en específico.

Usamos el hash de un commit, para ver lo que existe en el de forma rápida y sin tener que ingresar el hash completo.

Los tags son importantes para que los vea alguien, y sepan la versión, eso se observa en GitHub por eso es por lo que se debe enviar el TAG a él, para ello debemos realizar un **git pull origin master** para traer todo lo que tenemos en el origin hacia master.

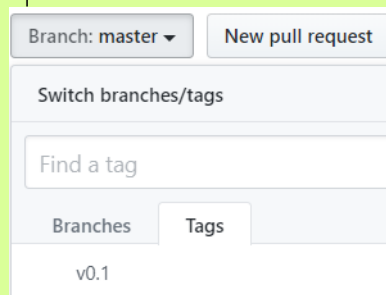
Para enviar los TAGS realizados a GitHub, se utiliza el siguiente comando:

`git push origin --tags`

Dentro del Bash, no nos aparecerá nada más que esto:

```
To github.com:Lorelicervantes/blog_platzi.git
* [new tag]          v0.1 -> v0.1
```

Al revisar en GitHub, en el área de Branch, a un lado podremos ver Tags y ahí nos aparecerá la versión del TAG que hemos subido:



Cuando envías por error un TAG hacia GitHub, la forma de eliminarlo es la siguiente:

`git push origin :refs/tags/nombre_tag`

1. CURSO PROFESIONAL DE GIT Y GITHUB

Repositorios Remotos / Manejo de ramas en GitHub.

REPASO DE COMANDOS RAMAS.

git branch [nombre_rama: te permite crear una rama con el nombre que elijas.

git checkout [nombre_rama]: permite navegar entre ramas.

git branch: nos muestra todas las ramas que existen.

git show-branch: nos muestra las ramas que existen y la historia de ellas.

git show-branch --all: nos muestra cuando se ha utilizado la rama y la divide por colores.

gitk: nos abre un software visual para observar los cambios.

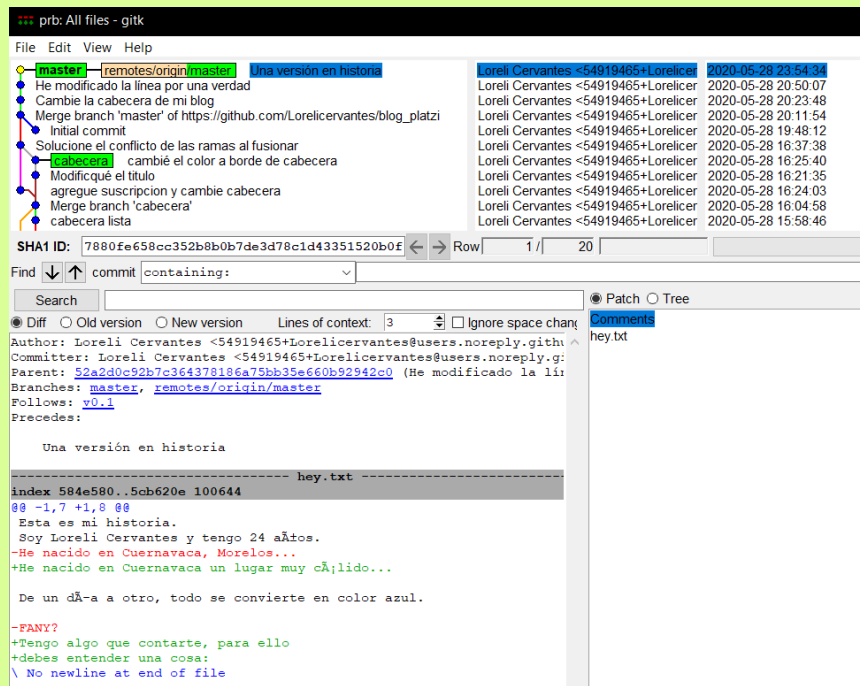
git push origin [nombre_rama]: envía nuestra rama hacia GitHub. *debemos estar dentro de la rama que deseamos enviar.

En GitHub siempre se vera la rama **MASTER** es decir el origen.

Internamente podemos ver un desglose de las ramas que tenemos con los comandos:

git show-branch
git show-branch --all

También existe en git, un software visual que se abre con el comando **gitk** y se ve de la siguiente forma:



****SIEMPRE ANTES DE ENVIAR ARCHIVOS A GITHUB DEBEMOS TRAER LOS DATOS AL REPOSITORIO LOCAL CON git pull origin master.**

Para enviar las ramas que tenemos en el repositorio local hacia GitHub, es necesario seguir estos pasos:

1. posicionarnos en la rama que deseamos subir con **git checkout [nombre_rama]**
2. Usar el comando (estando entro de la rama) **git push origin [nombre_rama]**

Y eso sería todo para enviar una rama hacia GitHub, cabe mencionar que la rama debe estar creada previamente.

1. CURSO PROFESIONAL DE GIT Y GITHUB

Repositorios Remotos / Múltiples colaboradores en GitHub.

`git restore --staged <file>`:
elimina el archivo que hemos
agregado con `git add`.

`git pull origin [nombre_rama]`:
sirve para traer el origen del
repositorio remoto a una rama
en específico.

`git pull origin footer`: nos trae la
rama footer desde GitHub a
nuestro repositorio remoto.

CTRL + F5: Para forzar la
actualización de un sitio.

Para añadir a colaboradores del repositorio en GitHub
debemos:

1. Ir a settings del repositorio.
2. Collaborators, ahí podremos enviarles la invitación por medio de su correo electrónico ligado a GitHub.
3. Si no puede agregarse por correo, también existe la posibilidad de agregarlo por nombre de usuario.

Cualquier persona puede clonar un repositorio si es público, pero no tendrá acceso a realizar push al origen, es decir que si realiza cambios, no podrá subirlos al repositorio a menos que sea colaborador.

****LAS MEJORES PRÁCTICAS DICEN QUE LOS ARCHIVOS BINARIOS NO DEBEN AGREGARSE A LOS REPOSITORIOS.**

El problema con los binarios es el peso, entre más binarios haya, más pesado será nuestro repositorio.

El trabajo en conjunto debe realizarse en distintas ramas por lo cual para fusionar al MASTER, es necesario seguir los pasos:

1. Revisar los cambios de la rama a fusionar antes de hacerlo.
2. Una vez revisado podremos pasar a la rama MASTER
3. Una vez en MASTER podremos hacer el merge
4. Si el merge no se realiza porque hay conflictos, debemos solucionar los conflictos (**IR AL TEMA SOLUCIÓN DE CONFLICTOS**).
5. Después de realizar el merge y observar los cambios, el proyecto del MASTER
6. Se debe hacer un pull del origen al master para ver que no se hayan realizado cambios en el repositorio remoto.
7. Podremos hacer un push origin master, para subir todos los cambios que hemos revisado y realizado en la rama MASTER.

1. CURSO PROFESIONAL DE GIT Y GITHUB

Repositorios Remotos / Flujo de trabajo con Pull Request.

En este apartado se habla de GitHub para realizar Pull Request.

Un **Pull Request**: es un estado intermedio antes de enviar el merge, este nos permite que otros miembros del equipo visualicen los cambios realizados que hemos hecho y al aprobarlos se ejecuta el merge en staging.

Staging Developer: Es una rama donde se encuentra una copia exacta del MASTER final, y se utiliza para realizar pruebas.

El **Pull Request** es una característica única de GitHub, es importante porque permite a personas que no son colaboradores, trabajar en nuestro proyecto o apoyar en una rama.

Las personas que realizan este tipo de cambios, pruebas y acepta los **pull request** son los líderes de equipo o personas con un perfil de **DevOps**.

Para hacer un pull request debemos:

1. Seleccionar New Pull Request
2. Seleccionamos la base, es decir Master y la rama que queremos comparar:
3. Al entrar a la comparación del pull request, nos permite ver los cambios y agregar detalles.
4. Se puede añadir **Reviewers** para que revisen los pull request
5. Se da clic en pull request.

Una vez hecho el pull request, nos mandará a una pantalla donde observaremos la conversación principal del pull request, los commits, los cheks y los archivos modificados. Para ello debemos estar en files changed, estando ahí podemos, seleccionar Review changes y:

- ◆ Aceptar el pull request y hacer un merge.
- ◆ Solicitar modificar esos cambios.
- ◆ Realizar un comentario para hacer feedback.

Si solicitamos Reviewers, estos deben aceptar todos los cambios antes de realizar el merge, debe estar marcado con una palomita verde antes de realizar el merge.

Una vez hecho el merge al MASTER, es posible eliminar el branch, ya que hay branch que se abren solo para solucionar errores

1. CURSO PROFESIONAL DE GIT Y GITHUB

Repositorios Remotos / Creando un Fork en GitHub.

vi [nombre_archivo.extension]:
nos permite utilizar el editor
de texto vim desde consola.

stash them: solución de
errores.

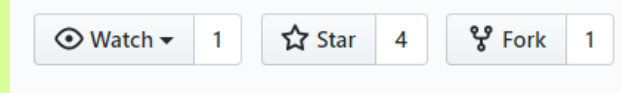
git remote -v: nos permite ver
donde tenemos el proyecto al
que se hace push y pull.

git remote add
[nombre_fuente]
[url_repositorio]: Nos crea una
fuente nueva, visible desde el
comando anterior para traer a
2 puntos un pull o un hacer un
push.

switching the base: nos
cambia a la base del
repositorio original donde
sacamos el fork en GitHub,
para eso antes debemos darle
en compare para ver los
cambios.

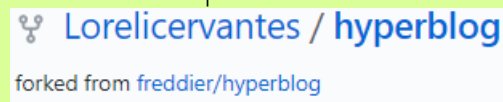
git remote add origin
[url_repositorio]: para agregar
un origen remoto de nuestros
archivos.

Un fork lo puede realizar una persona que **no es colaborador directo del repositorio** pero que si se encuentra interesado en aportar a él.

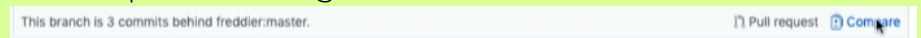


- ◆ Al hacer a un **watch** a un repositorio en GitHub marcamos que nos interesa y podremos ver cambios que se realicen.
- ◆ Al colocar una **star** significa que el proyecto me gusta y es importante.
- ◆ un **fork** es tomar una copia del estado actual proyecto y hacerlo mío, esto solo funciona para repositorio públicos.

AL realizar el fork, el proyecto se pasa nuestro lugar de repositorios, pero en la parte de abajo aparecerá de donde se copió el fork.



Lo que pasa cuando el repositorio de donde realizamos el fork avanza, nosotros debemos actualizar esos cambios. Al entrar al repositorio veremos si se han hecho cambios en el repositorio original:



Nosotros podremos hacer una comparación de lado derecho se encuentra el botón.

El flujo es el siguiente después de tener el fork

1. Crear una fuente y clonar el repositorio original.

```
lorel@LAPTOP-IDHRDSU3 MINGW64 ~/cursos/hyperblog (master)
$ git remote add upstream git@github.com:freddier/hyperblog.git
```

2. git remote -v nos va a permitir ver las fuentes de repositorios remotos que tenemos:

```
lorel@LAPTOP-IDHRDSU3 MINGW64 ~/cursos/hyperblog (master)
$ git remote -v
origin  git@github.com:Lorelicervantes/hyperblog.git (fetch)
origin  git@github.com:Lorelicervantes/hyperblog.git (push)
upstream  git@github.com:freddier/hyperblog.git (fetch)
upstream  git@github.com:freddier/hyperblog.git (push)
```

3. Hacer un **git pull upstream master**. Para traer todo lo que clono **upstream** al **MASTER** del fork.
4. Después de tenerlo ya en nuestro **MASTER**, procedemos a realizar un **git push origin master**, para guardar los cambios del **fork** en GitHub.

En el repositorio en GitHub debe aparecer algo así:

This branch is 1 commit ahead of freddier:master.

1. CURSO PROFESIONAL DE GIT Y GITHUB

Repositorios Remotos / Ignorar archivos con .gitignore.

La sintaxis dentro de git ignore es la siguiente:

* que nos dice que añadirán todos los archivos que tengamos. seguido del `[extensión]`

*.jpg
*.png
*.doc

para ignorar carpetas debe ponerse de la siguiente manera:
`/nombre_carpeta`

sí seleccionamos la carpeta pero no deseamos que todo sea ignorado:
`!/nombre_carpeta/(archivos que no deben ignorarse)`

Si deseamos ignorar de una carpeta un archivo específico:
`/nombre_carpeta/extensión`

Con el símbolo hash # podemos hacer comentarios dentro del .gitignore

.gitignore es la forma en la cual podemos ignorar archivos importantes para que no puedan verlos, es decir que no serán agregados al repositorio. Sirve por si deseamos añadir archivos binarios o archivos importantes debemos:

****evitar que los archivos binarios del contenido sean parte de un repositorio.**

Para usar .gitignore debemos seguir los siguientes pasos:

1. Crear un nuevo archivo .gitignore y guardarlo en la raíz del proyecto.
2. .gitignore es una lista de los archivos que vamos a ignorar.
3. Se guarda y debemos ir a Bash y dar **git status**.
4. Una vez ahí si teníamos archivos con **git add**.
5. Hacer **git commit**.
6. Hacer un **git pull origin master**.
7. Hacer un **git push origin master**.

Las imágenes ya no aparecerán, por lo que debemos subirlas a un servidor y cargar la url para poder visualizarlas.

1. CURSO PROFESIONAL DE GIT Y GITHUB

Repositorios Remotos / Readme.md.

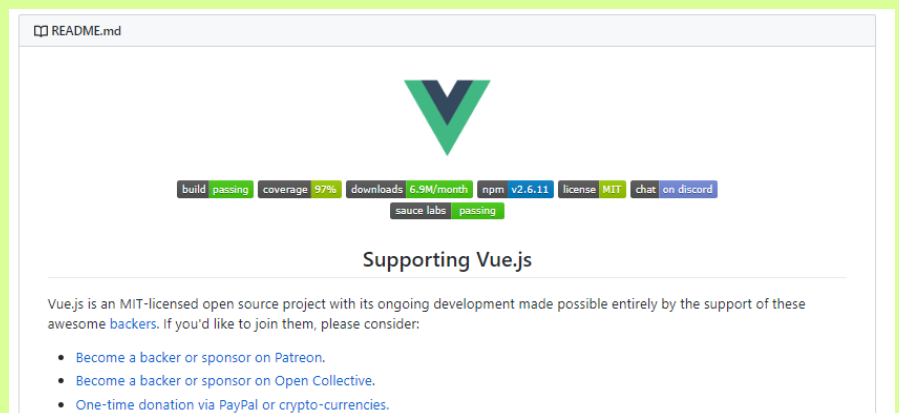
##: Mark Down es un lenguaje intermedio utilizado en Wikipedia y también para crear un readme.md

para salir del editor de texto
Vim: Esc, después :x Enter

Readme.md Existe para mostrarle al mundo de que trata nuestro repositorio y es una buena práctica dentro de nuestros repositorios.

para poder crear nuestro README.md podemos utilizar el siguiente link: <https://dillinger.io/>

Ejemplo de un README.md:



1. CURSO PROFESIONAL DE GIT Y GITHUB

Repositorios Remotos / Git rebase.

`git rebase [nombre_rama]`: Nos permite hacer un rebase, es decir, pegar lo que hemos realizado en la rama.

`git rebase master`: trae todo lo que tenemos en nuestra rama hacia master pegándolo.

Rebase: es tomar una rama entera y pegarla en la historia del MASTER, no con un merge... si no como si se hubiera trabajado en el MASTER todo el tiempo.

**Es muy mala práctica hacer eso en los repositorios remotos, es mejor utilizarla internamente.*

Funciona como un parche.

Para hacer un rebase debemos:

1. Posicionarnos en una rama distinta a MASTER.
2. Crear cambios en esa rama
3. En lugar de hacer un merge, debemos: estando en la rama que queremos enviar a MASTER usar el comando:

`git rebase MASTER`

4. Para unir lo que hay en Master en nuestra rama_prueba.
5. Después debemos posicionarnos en MASTER con **`git checkout master`**, estando en Master debemos usar el siguiente comando:

`git rebase rama_prueba`

6. Y eso nos traerá toda la información que existe en esa rama hacia la rama MASTER.
7. Una vez realizado eso será como si la **rama_prueba**, jamás hubiera existido pues los commits que se hicieron dentro de la **rama_prueba** ahora son parte de la historia del MASTER

La magia de **rebase** radica en que, si se realizan cambios en la rama MASTER, mientras estamos en alguna otra rama, al ejecutar el comando rebase en la **rama_prueba**, traerá primero todos los cambios de MASTER hacia tu rama y lo fusiona.

Es muy útil usar esta información para datos locales, no para datos remotos ya que modifica por completo la historia de donde fue creada Tal rama.

Recapitulando **un rebase** es la forma de realizar cambios silenciosos en otras ramas, como si no existieran y llevarlas al MASTER.

1. CURSO PROFESIONAL DE GIT Y GITHUB

Repositorios Remotos / Git stash.

git stash: nos permite volver a un estado anterior de los cambios realizados, porque almacena los cambios en un lugar temporal.

git stash list: nos permite ver los cambios que tenemos guardados en un lugar temporal.

WIP: Work In Progress.

git stash pop: nos regresa a lo que teníamos dentro del stash (es decir los cambios que hicimos).

git stash branch
nombre_rama: nos manda lo que tenemos en el stash guardado (el cambio que hicimos) a otra rama, la rama que escribamos puede ser una nueva que se crea ahí mismo, o una que ya tengamos.

Git Stash es: nos permite almacenar un cambio en un lugar temporal. Es aquel que toma un cambio y lo guarda hasta que decidamos volver a soltarlo.

Para ejecutarlo hace falta escribir el comando:

git stash

Para soltar el cambio que guardamos es:

git stash pop

git stash también nos sirve por si hemos roto parte del código y hemos guardado cambios dentro del mismo proyecto.

OJO(no hecho git add o git commit -am, eso lo resuelve un git rm o un git reset)

Ya que podemos guardar esos cambios hechos en el stash y utilizar el comando

git stash drop

esto es para borrar los cambios que hay en el stash, es decir aquellos cambios que hicimos que rompieron el código.

como lo dice en la parte lateral izquierda, al ejecutar el comando **git stash** el cambio que recién hicimos lo guarda en el **stash** y nos regresa a lo que teníamos en el commit anterior, pero los cambios los tenemos guardados dentro de ese stash para:

- ◆ Soltarlos en el código para modificarlo.
- ◆ Ponerlos dentro de una rama.
- ◆ Borrarlos del stash, si es que esos cambios no nos sirven.

Después de haber aceptado un stash se debe hacer un **git commit -am "mensaje"** para guardarlo.

1. CURSO PROFESIONAL DE GIT Y GITHUB

Repositorios Remotos / Git clean y Git cherry-pick.

`git clean --dry-run`: es un comando que sirve para simular lo que se va a borrar, sin borrarlo.

`git clean -f`: borra las copias que no nos sirven dentro del proyecto.

`git cherry-pic #commit`: Nos trae a la rama en la que nos encontramos el cambio del commit seleccionado.

`git log --oneline`: nos muestra en una sola línea simplificada la lista de los commits.

Git clean: nos sirve para limpiar de nuestro repositorio a los archivos que por error creamos o que no nos sirven realmente como lo son copias de archivos.

Git las detecta y puede eliminarlas pero para saber que va a eliminar antes de hacerlo se utiliza el comando:

`git clean --dry-run`

git elimina con **git clean -f** los archivos (aquellos que ya hemos revisado con el comando anterior), SOLO no elimina copias de carpetas o de archivos que se encuentren dentro de **.gitignore**

Git cherry-pick: Sirve para traernos cambios específicos de una rama a otra, sin tener que hacer merge utilizando únicamente el commit del cambio que deseamos utilizar.

Para utilizarlo necesitamos:

1. Posicionarnos en la rama de donde queremos sacar el #commit.
2. Realizar un **git log --oneline** para ver el # del commit que deseamos traernos a la otra rama.
3. Una vez visto el # lo copiamos.
4. Nos regresamos a la rama donde queremos traer ese commit.
5. Ejecutamos el comando **git cherry-pic #commit** y listo, tendremos la actualización de ese commit que se encontraba en otra rama, dentro de nuestra otra rama, en este caso **MASTER**.

```
Education-Platzl@Laptop MINGW64 ~/proyecto1 (master)
$ git cherry-pick dca2a24
[master c66c7a5] Credits al team Platzl
Date: Thu Apr 18 15:13:01 2019 -0700
1 file changed, 1 insertion(+)
```

*Los cambios se verán reflejados dentro de nuestro código.

1. CURSO PROFESIONAL DE GIT Y GITHUB

Repositorios Remotos / Reconstruir en Git con amend.

`git commit --amend`: sirve para copiar los cambios que hicimos al commit anterior.

`Git commit --amend`, nos ayuda cuando hemos enviado un commit y nos ha faltado añadir algo, entonces podemos utilizar este comando y todo lo que agreguemos se va a pegar en el commit anterior.

La forma correcta de realizar los cambios es:

1. Al observar que nos ha faltado algo debemos modificarlo y hacer un **git add [archivo.extension]**
2. Eso debe hacerse antes de hacer el commit, ya que el commit se ejecutará en la forma del comando para pegarlo:

`git commit --amend`

Y listo con eso habremos “**enmendado**” la falta que no subimos en el commit anterior. Se le llama enmendar porque es lo que significa amend en inglés.

1. CURSO PROFESIONAL DE GIT Y GITHUB

Repositorios Remotos / Git Reset y Reflog: EMERGENCIAS.

git reflog: nos muestra todos los **head** que hemos tenido en el proyecto a lo largo de nuestro proyecto.

git reset (--HARD o --SOFT) HEAD@{numero}: nos recupera el head de los commits.

Git Reflog, nos muestra TODO, absolutamente toda la historia incluso ramas que hemos eliminado ya que nos muestra todos los head que hemos tenido.

Al mostrarnos todos los head, podemos seleccionar el **HEAD** al que deseemos volver, si es que hemos tenido un error enorme, como lo dice el título, solo se usa en emergencias.

por ejemplo: aquí podemos volver al head donde realicé un cambio mal usamos: HEAD@{numero}

```
86ba281 HEAD@{4}: commit: Agregando el .gitignore
98a4fb6 HEAD@{5}: pull origin master: Fast-forward
d71458c HEAD@{6}: checkout: moving from fix-typo to master
5d9113b (origin/fix-typo) HEAD@{7}: commit: Cambio de nombre al saludo
316371c HEAD@{8}: commit: Fin del commit 7
```

después de eso utilizamos (los comandos ya antes vistos)

git reset (--HARD o --SOFT) HEAD@{numero}

Ese comando nos regresa a la posición donde estuvimos antes, dependiendo de si usamos **Hard** o **Soft** nos podrá guardar lo que tenemos en staging o no.

Aun así si hemos eliminado archivos el **HEAD@{numero}** no nos servirá pues solo nos recupera el Head de los commits no los archivos eliminados.

Para ello demos utilizar la misma función pero esta vez copiando el **#commit** para volver a ese punto, utilizando:

git reset --hard #commit

```
Education-Platzi@Laptop MINGW64 ~/proyecto1 (master)
$ git reset --hard c894560
HEAD is now at c894560 Cambio al tagline y color del footer :
)
```

1. CURSO PROFESIONAL DE GIT Y GITHUB

Repositorios Remotos / Git Grep y Log.

`git grep [palabra]`: busca esa palabra en nuestros archivos en la rama en la cual nos encontremos.

`git grep -n [palabra]`: nos da la línea específica en donde se encuentra esa palabra.

`git grep -c [palabra]`: nos muestra el número de veces que se utiliza esa palabra en nuestros distintos archivos.

`git log -S "palabra"`: busca palabras usadas en commits.

Cuando nuestro proyecto se hace grande, se hace más complicado buscar algunas palabras que hemos utilizado dentro del proyecto, para eso existe el comando:

`git grep [palabra]`

al ejecutar ese código nos mostrará los resultados de la siguiente manera:

```
loreli@LAPTOP-IDHRDSU3 MINGW64 ~/cursos/prb (master)
$ git grep color
css/style.css:      color: #333;
css/style.css:      color: white;
css/style.css:      color: aliceblue;
css/style.css:      background-color: #484ae4;
hey.txt:De un día a otro, todo se convierte en color azul.
```

mostrándonos todos los lugares donde utilice la palabra color.

para conocer la línea de código exacta en donde utilicé esa palabra se usa el comando:

`git grep -n [palabra]`

```
loreli@LAPTOP-IDHRDSU3 MINGW64 ~/cursos/prb (master)
$ git grep -n color
css/style.css:2:      color: #333;
css/style.css:16:     color: white;
css/style.css:26:     color: aliceblue;
css/style.css:27:     background-color: #484ae4;
hey.txt:5:De un día a otro, todo se convierte en color azul.
```

Para saber cuántas veces he usado esa palabra en los archivos usamos el comando:

`git grep -c [palabra]`

```
loreli@LAPTOP-IDHRDSU3 MINGW64 ~/cursos/prb (master)
$ git grep -c color
css/style.css:4
hey.txt:1
```

Para buscar una palabra, no dentro de los archivos del proyecto, sino en los commits se utiliza el comando:

`git log -S "palabra"`

Se nos muestra de la siguiente forma:

```
loreli@LAPTOP-IDHRDSU3 MINGW64 ~/cursos/prb (master)
$ git log -S "cabecera"
commit 7be8284f6e4e3d02bb4b12efd059a0811db69c6c
Author: Loreli Cervantes <54919465+Lorelicervantes@users.noreply.github.com>
Date: Thu May 28 20:23:48 2020 -0500

    Cambie la cabecera de mi blog

    Solo fue un cambio loco de idea

commit 9c12a2287c1838645de677d901dca309088738fb
Author: Loreli Cervantes <54919465+Lorelicervantes@users.noreply.github.com>
Date: Thu May 28 15:58:46 2020 -0500

    cabecera lista
```

1. CURSO PROFESIONAL DE GIT Y GITHUB

Repositorios Remotos / Recursos Colaborativos Git y GitHub.

`git shortlog`: para ver cuántos commits ha hecho cada miembro del equipo.

`git shortlog -sn`: para contar los commits y las personas que han hecho commits.

`git shortlog -sn --all`: nos muestra todos los commits hechos incluso los que se borraron.

`git shortlog -sn --all --no-merges`: Vuelve a mostrar los commits total de cada miembro del equipo eliminando los merges.

`git config --global alias.stats "comando"`: sirve para agregar a la configuración local un alias sin tener que añadirlo cada vez que se abre git.

`git config --global alias.stats "shortlog -sn --all --no-merges"`

`git [comando] --help`: no sabe un manual de cómo funciona el comando

`git blame [archivo.extension]`: nos muestra quien ha hecho los cambios en ese archivo

`Git branch -r`: vemos las ramas remotas que tenemos en el servidor GitHub.

`git branch -a`: se ven todas las ramas (remotas y locales)

Insights. Nos muestra TODO lo que debemos saber del proyecto. Pulsos, contribuidores, comunidad, commits, Frecuencia del código por días, Dependencia del código (librerías a las que sigue el proyecto). Network, Forks, Advertencias.

Gracias a todos los que han descargado esta guía que realicé en mi paso por la ruta, espero le sirva de mucha ayuda. Mi nombre es @Loreli-Cervantes y para mí es un placer ayudarlos.