**TUTORIAL BÁSICO Y COMANDOS ÚTILES DE GIT**

1. **Instalar GIT**

Descargar e instalar GIT desde la página oficial <https://git-scm.com/downloads>, una vez instalado abrimos el GIT BASH.

1. **Verificar versión de GIT**

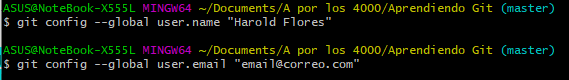
Para saber la versión de Git instalada utilizamos el comando git --version



1. **Configurar nombre y correo de contribuidor de GIT**

Para comenzar debemos identificarnos con un nombre y un correo, esto se lo hace una sola vez y es útil para trabajar en equipo, ya que se podrá ver quién hizo los cambios del código.

* git config --global user.name “[Nombre]”
* git config --global user.email “[Correo]” (No es necesario que sea un correo válido)



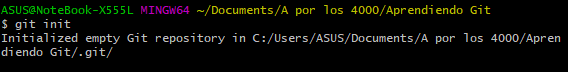
1. **Visualizar nombre y correo de contribuidor de GIT**

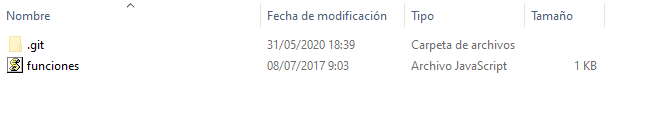
Con el comando git config –-global -l puedo ver el usuario y correo actual del repositorio local



1. **Creación de un repositorio local**

Primero nos ubicamos en la carpeta que queremos tener nuestro repositorio local, ya dentro de la carpeta seleccionada ejecutamos el comando git init el cual inicializa el repositorio local creando una carpeta .git oculta.





1. **Arquitectura básica de GIT**

Para entender de manera general a Git, hablaremos de sus 3 estados o áreas que se manejan.

WORKING DIRECTORY

Git add

STAGING AREA

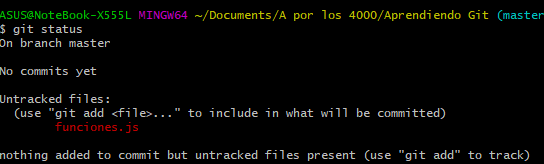
Git commit

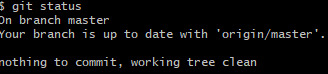
REPOSITORY

El working directory es el estado inicial, en donde se encuentran todos los archivos de la carpeta en donde se inicializó el git, pero que aún no han sido incrustados en el repositorio. El staging area es un estado intermedio de GIT en donde se encuentran los archivos escogidos para ser incrustados en el repositorio, pueden ser todos los archivos o los que yo elija específicamente. Por último el repository es el estado final, en donde ya se encuentra guardada una versión de los archivos.

1. **Verificar en que estado se encuentran los archivos del repositorio.**

git status muestra el estado actual de los archivos. Si los nombres de los archivos están en rojo, es porque aún no han sido añadidos al repositorio, es decir que siguen en el working directory. Si los nombres de los archivos están en verde significa que ya se encuentran añadidos al staging area. Finalmente si aparece el mensaje Nothing to commit, working tree clean, significa que todos los archivos ya han sido subidos al repositorio.

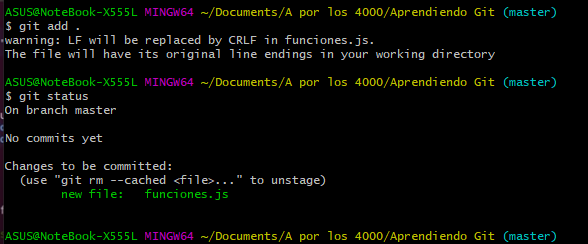




1. **Pasar de estado Working Directory – Staging Area**

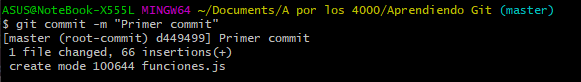
Para pasar de estado (Working directory – Staging Area), debemos ejecutar el comando git add .

Si hacemos un git status, los nombres de los archivos se cambian a color verde. Hay que tomar en cuenta que el punto “.” al final del git add significa que se añaden **todos** los archivos de la carpeta al staging área, si se desea añadir archivos específicos, se especifica el nombre del archivo, ejemplo: git add index.html y solo el index será añadido al staging área.



1. **Pasar de estado Staging Area - Repository**

El comando git commit -m “[Mensaje]” se utiliza para terminar de añadir los archivos del staging area al repositorio. El mensaje dentro de las comillas sirve para identificar el cambio realizado.

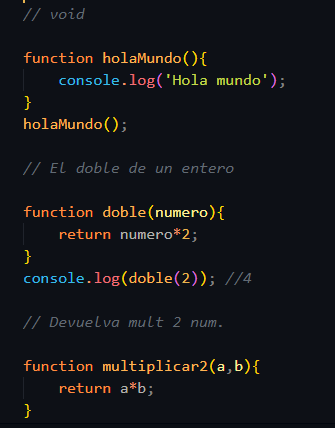
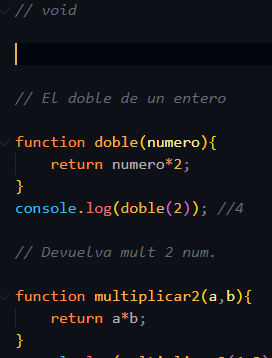


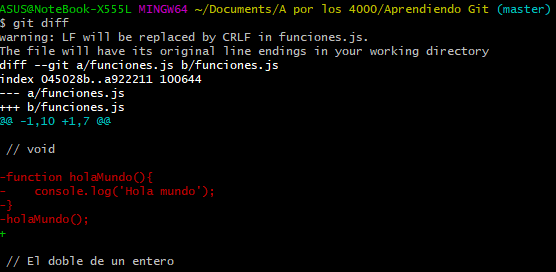


1. **Verificar cambios entre el repositorio actual y el último commit**

El comando git diff sirve para comparar la situación actual del repositorio con la del último commit. Es decir, muestra si faltan o han sido modificados archivos.

* Como ejemplo voy a borrar la función holaMundo del archivo funciones.js y comprobar los cambios mediante el comando git diff



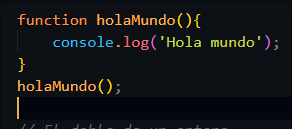


1. **Restaurar archivos del último commit**

El comando git checkout . sirve para restaurar los archivos al último commit en caso de haber modificado los archivos por error.

* Como se ve en el ejemplo después de borrar la función holaMundo del archivo añadido al repositorio, ejecute el comando git checkout . y restauró la función holaMundo automáticamente debido a que estaba presente en mi último commit.

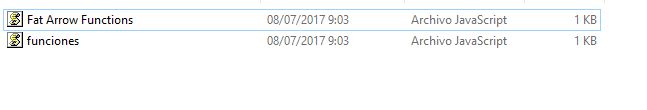




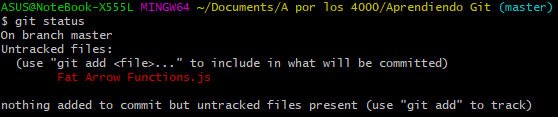
1. **Quitar archivos del Staging Area**

El comado git reset sirve para quitar los archivos del staginig área y devolveros al working directory, esto es útil en caso de equivocarnos en agregar archivos al staging area que no queremos añadirlos al repositorio.

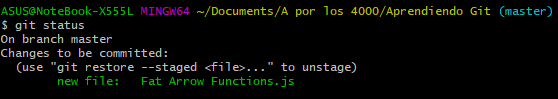
* Como ejemplo se añadió el nuevo archivo Fat Arrow Functions.js a la carpeta en donde se encuentra mi repositorio.



* Con la ayuda del comando git status se puede observar que el archivo Fat arrow functions.js se encuentra en color rojo debido a que es un nuevo cambio.

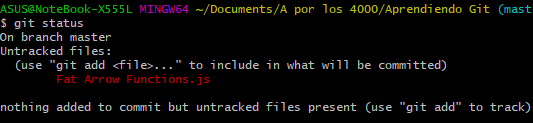


* Al hacer el git add . el archivo pasará al staging área cambiando su color a verde



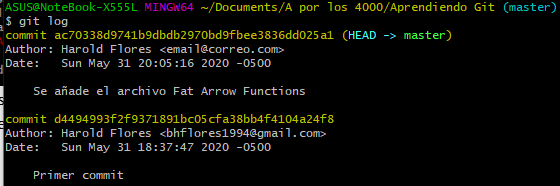
* Finalmente para sacar el archivo del staging area utilizamos el comando git reset [NombreDelArchivo], devolviéndolo al working directory.





1. **Historial de commits**

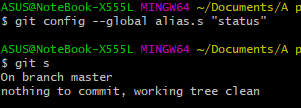
El comando git log muestra un histórico de los commits realizados incluyendo su id-hash, la fecha, el autor y el correo del usuario que lo realizó.



1. **Alias para comandos**

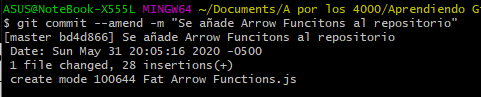
Los alias sirven para establecer un nombre corto o cualquier nombre a un comando con el fin de ahorrar tiempo o facilitar su escritura.

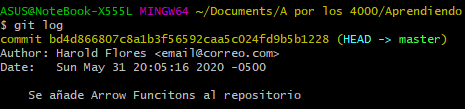
* git config –global alias.[Alias] “[Comando]”



1. **Modificar mensaje de un commit**

Si se comete algún error al poner el mensaje del commit puedo cambiarlo mediante el comando git commit –amend -m “[Nuevo mensaje]”



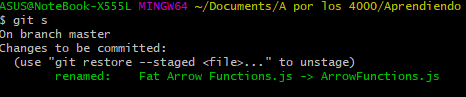


1. **Cambio de nombres a archivos dentro del repositorio**

Si se desea cambiar el nombre de un archivo que ya fue puesto en el repositorio, puedo hacerlo de dos maneras, la primera cambiando el archivo manualmente desde el explorador de archivos y después ejecutar los comandos git add . y git commit , pero una buena práctica es mediante el comando git mv [Nombreantiguo] [Nombrenuevo] debido a que se sabe que archivos fueron renombrados (renamed en el git status). Cabe recalcar que los cambios se reflejan automáticamente en el explorador de archivos. Finalmente también es necesario terminar de añadir los cambios al repositorio mediante los comandos git add . y git commit

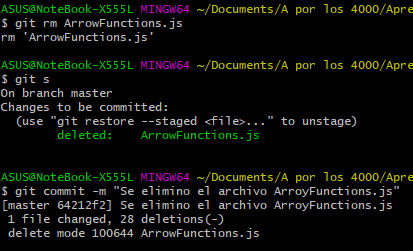






1. **Eliminar un archivo del repositorio**

Al igual que el caso anterior, si debemos borrar un archivo, podemos eliminarlo y volver a hacer el commit, sin embargo para llevar una control de la versión, se recomienda usar el comando git rm Archivo



1. **Recuperar archivos de anteriores commits**

Si por error eliminé un archivo o deseo recuperarlo, puedo volver a recuperarlo a través del comando git reset --hard [commitid], en donde el commitid es el el id del commit al que queremos llegar después del hard.





1. **Obtener el id o hash del commit**

Podemos obtener el id o hash del commit tras ejecutar los comandos git log o git reflog, en donde el id son la combinación de números y letras presentadas en color amarillo-

* + **Git log**

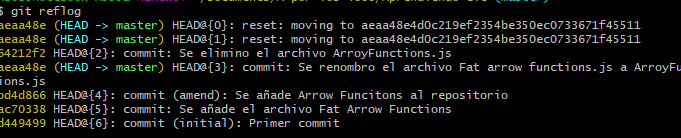


* + **Git reflog**



1. **Git reflog**

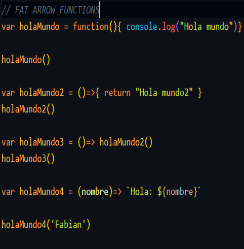
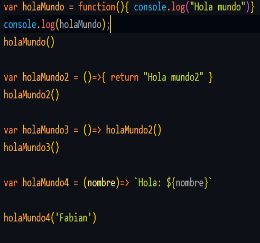
Después de volver a un commit anterior, el comando git log no mostrara los cambios realizados después de ese commit, sin embargo con el comando git reflog sí se puede visualizar todo el histórico de commits realizados en el repositorio.

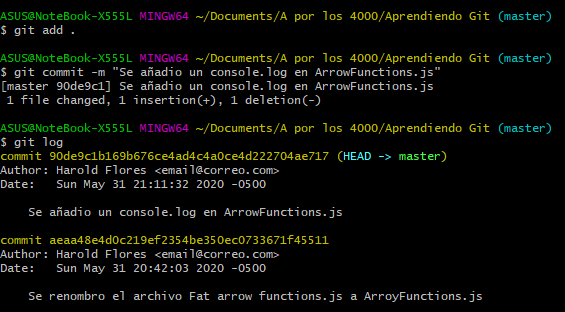


1. **Ubicarse en otro commit manteniendo los cambios del último commit**

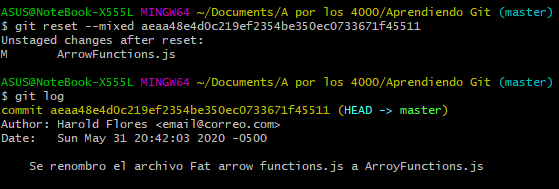
Los comandos git reset –-mixed [Commitid] y git reset –-soft [commitid] , sirve para ubicarse en otro commit, con la característica de que los cambios del último commit se van a mantener. La diferencia entre el git reset mixed y git reset soft radica en que el primero (mixed) tendrá todos los nuevos archivos dentro del working directory en cambio el segundo (soft) tendrá agregado los archivos al staging area.

* Como ejemplo voy a modificar mi archivo ArrowFunctions.js y haré un commit.

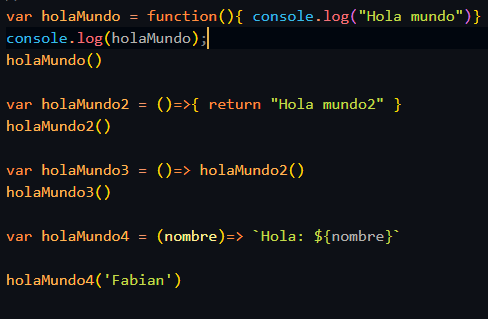




* Ahora volveré al commit de renombrar el archivo haciendo uso del git reset –mixed [Commitid]



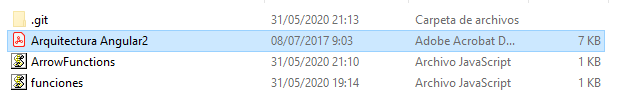
* Como se puede ver volvimos al último commit, sin embargo, los cambios realizados en el anterior commit se mantienen.



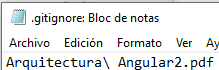
1. **Archivo .gitignore**

El archivo .gitignore permite excluir archivos que no queremos que se carguen en nuestro repositorio, pueden ser configuraciones, dependencias o cualquier tipo de archivos que queremos que git ignore al momento de subirlo al repositorio.

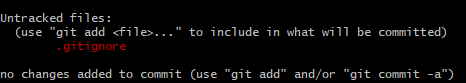
* Como ejemplo voy a cargar un nuevo archivo a mi carpeta donde inicialice el git.



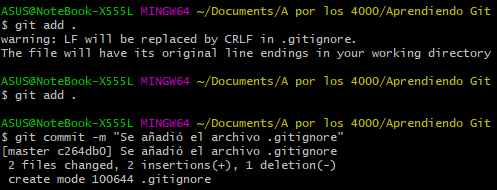
* Ahora voy a crear el archivo .gitignore y añadiré el archivo que no quiero que sea tomado en cuenta en este caso Arquitectura Angular2



* Al hacer un git status solo tomara como nuevo al documento .gitignore e ignorará al archivo Arquitectura Angular2



* Por último hacemos el commit y el archivo .gitignore será parte de nuestro repositorio.

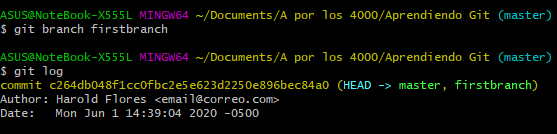


* <https://github.com/github/gitignore> en este link puedes encontrar ejemplos de archivos .gitignore dependiendo de la tecnología que estés usando lo que tendrías que hacer es simplemente quitar el nombre completo (ejemplo java.gitignore) y cambiarlo solo por .gitignore. Esto te ayudará a no sobrecargar el repositorio con información irrelevante

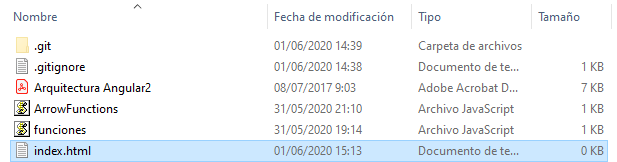
1. **Ramas**

Las ramas son extensiones del proyecto que apuntan a la rama principal, estas sirven para trabajar de manera experimental sobre el proyecto y cuando se está seguro de las modificaciones desarrolladas se las carga a la rama principal (master) a través de un merge.

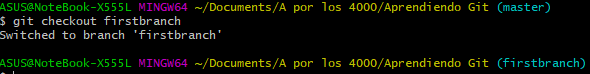
* Comenzaremos creando una nueva rama llamada firstbranch con la ayuda del comando git branch [NombredelaRama]



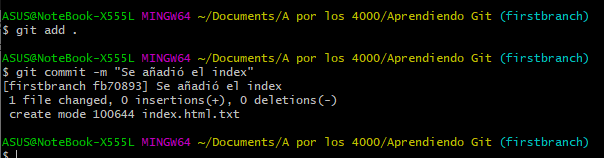
* Observamos que ahora hay dos ramas, la master y la firstbranch, sin embargo el HEAD apunta a master. Ahora procedemos a crear un nuevo archivo de prueba dentro de la carpeta donde se encuentra iniciado el git.



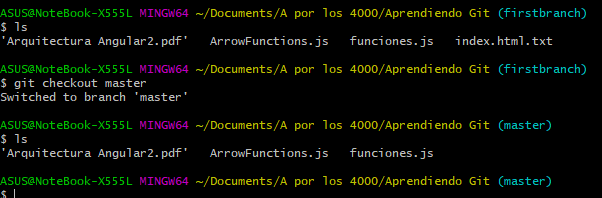
* Ahora nos ubicaremos en la nueva rama usando el comando git checkout [NombredelaRama]



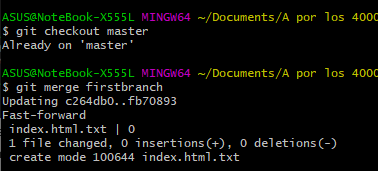
* Una vez ubicados en la nueva rama hacemos el git add . y el git commit para guardar los cambios que se verán reflejados solo en la rama seleccionada.



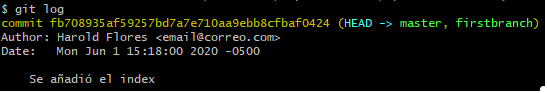
* Los cambios solo se verán reflejados en la rama firstbranch más no en la rama master.



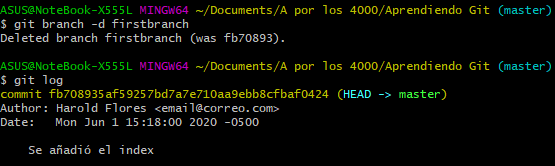
* Una vez que estamos seguros de los cambios realizados en la rama creada, podemos unir los cambios a la rama principal a través de un merge. Para ello primero nos ubicamos en la rama master y hacemos un git merge [NombredelaRama]



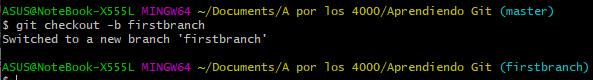
* Aquí ya no es necesario hacer un commit puesto que toma el último commit de la rama creada.



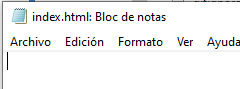
* Si queremos podemos borrar la rama podemos usar el comando git branch -d [NombredelaRama] . Este comando no lo podemos ejecutar desde el interior de la rama que queremos eliminar, por lo que tenemos que ubicarnos en la rama master.



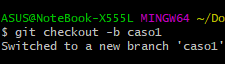
* Un comando opcional para crear la rama y ubicarnos en ella instantáneamente sin la necesidad de usar el checkout, podemos usar el comando git checkout -b [NombredelaRama]



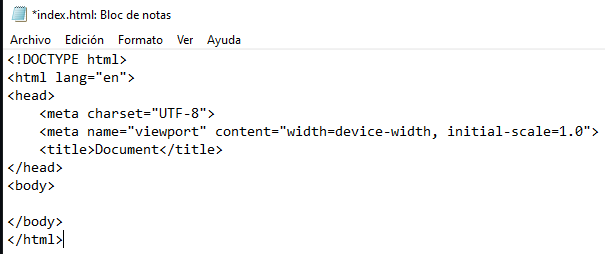
* El comando merge no dará problemas siempre y cuando no haya conflictos entre archivos de la rama principal con el de la rama nueva. Los conflictos se presentan cuando los archivos de la rama principal con la rama creada son diferentes y para solucionarlos toca hacerlo manualmente. A continuación se muestran cuatro casos de modificación de archivos entre las ramas y veremos cuales crean conflictos y cuáles no.
  + - Caso 1: cambios en la nueva rama pero el master se mantiene igual.
      * Comenzamos con el archivo index.html vacío y añadido al repositorio en la rama principal.



* + - * Creamos la nueva rama llamada caso1

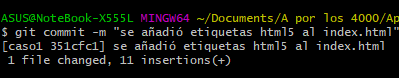


* + - * Modificamos el index.html de la rama caso1



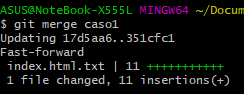
* + - * Guardamos y lo añadimos al repositorio



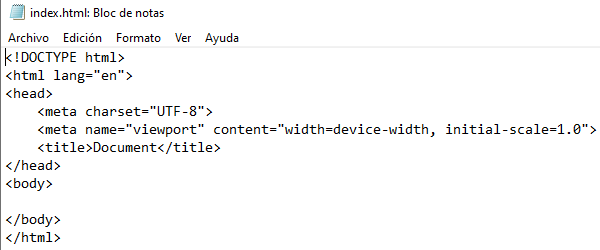


* + - * Volvemos a la rama principal y hacemos merge

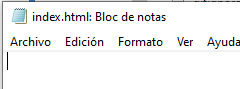




* + - * Como podemos ver no generó conflicto, más bien añadió automáticamente los cambios de la rama caso1 a la rama master.



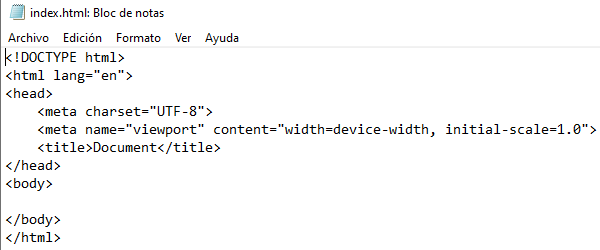
* + - Caso 2: cambio en la rama master pero la nueva rama se mantiene igual
      * Comenzamos con el archivo index.html vacío y añadido al repositorio en la rama principal.

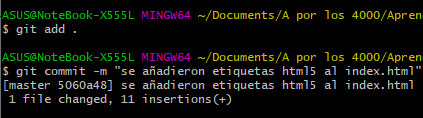


* + - * Creamos la nueva rama llamada caso2.



* + - * Modificamos el archivo index.html de la rama principal, lo guardamos y lo añadimos al repositorio.

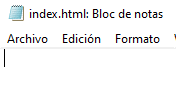




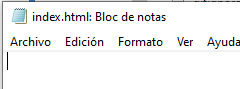
* + - * Hacemos merge con la rama creada que esta con el index.html vacío.



* + - * En este caso no dio ningún conflicto aunque los dos archivos index.html eran distintos, pero al contrario del caso anterior, la rama creada no se llenó con los datos de la rama master se quedó igual a cuando fue creada la rama, en este caso se mantuvo vacío.

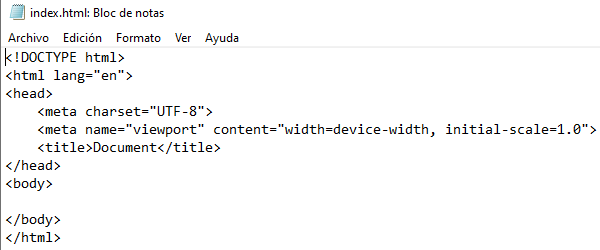


* + - Caso 3: El mismo cambio tanto en la rama master como en la nueva rama
      * Comenzamos con el archivo index.html vacío y añadido al repositorio en la rama principal.

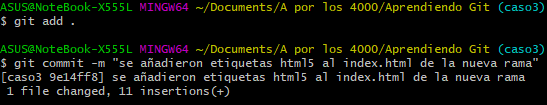


* + - * Creamos la nueva rama y modificamos el index.html



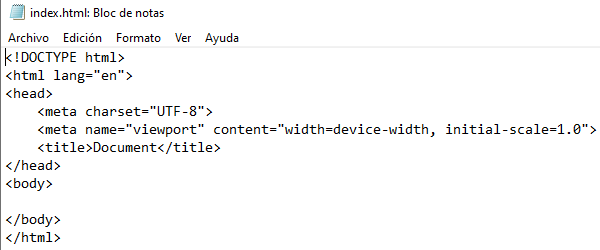


* + - * Lo añadimos al repositorio de la rama creada

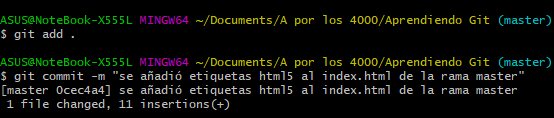


* + - * Cambiamos a la rama master y hacemos el mismo cambio



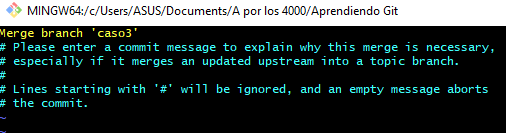


* + - * Añadimos el cambio al repositorio de la rama master.



* + - * Hacemos merge con la rama creada.





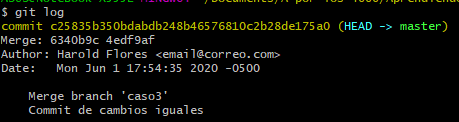
* + - * En este caso aparece este mensaje en donde podemos especificar qué mensaje queremos asignarle al último commit. Al hacer merge se utiliza el ultimo commit de la rama creada, pero en este caso como son los mismos cambios podemos asignar cualquier nombre al commit.

Para salir de la pantalla debemos aplastar esc + :wq y saldremos del editor guardando los cambios.

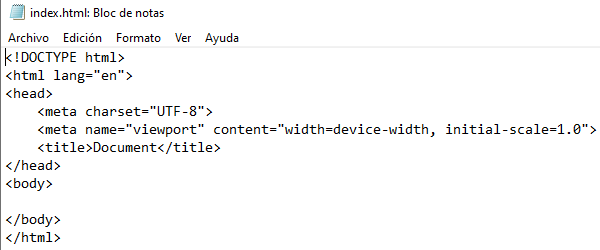
* + - * Podemos ejecutar nuevamente el merge sin generar conflictos



* + - * Se puede observar con git log los commits y si se añadieron líneas en el archivo abierto se lo podrá visualizar.

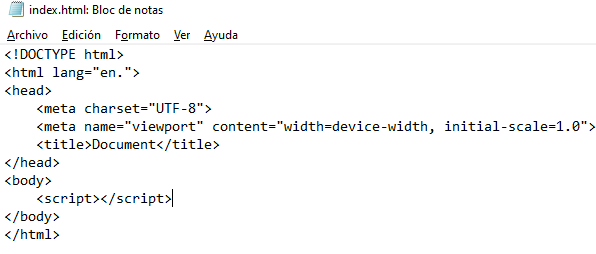


* + - Caso 4: cambio en las mismas líneas de código de la rama master y de la nueva rama.
      * Comenzamos con un archivo index.html con la estructura html5 alojado en la rama principal y añadido al repositorio.

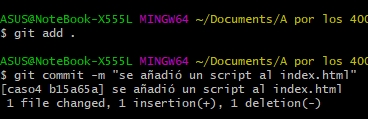


* + - * Creamos la nueva rama y modificamos el index.html añadiéndole una etiqueta script.



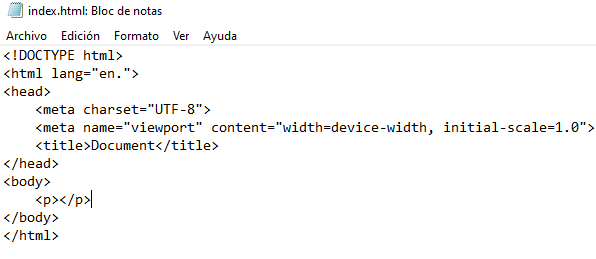


* + - * Guardamos y lo añadimos al repositorio.

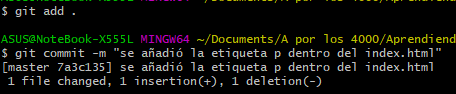


* + - * Regresamos a la rama master y modificamos el index.html, solo que esta vez añadiremos una etiqueta de párrafo.





* + - * Guardamos y lo añadimos al repositorio

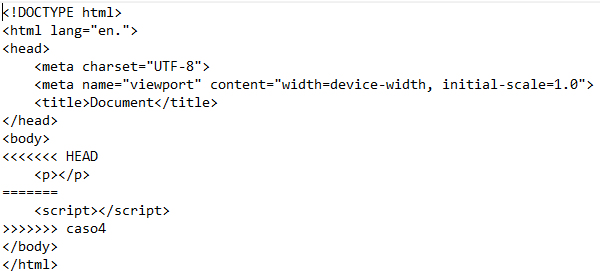


* + - * Realizamos el merge con la rama creada.

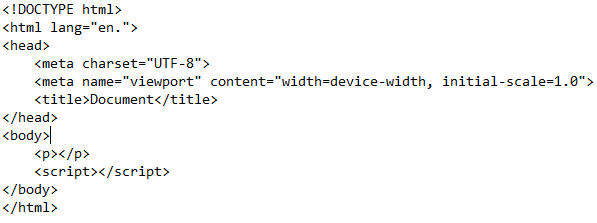


* + - * Aquí podemos ver un conflicto ya que los dos archivos fueron modificados en la misma línea.

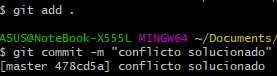
Abrimos el index.html de la rama **master** y observamos símbolos de igual, flechas hacia la derecha y hacia la izquierda, esto simboliza los conflictos entre las ramas y para solucionarlo tenemos que eliminar estos símbolos y letras manualmente.



Cambios:



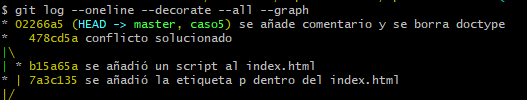
* + - * Guardamos y lo añadimos al repositorio.



* + - * Una vez resuelto podemos verificar con otro merge a la rama y todo estará sin errores.

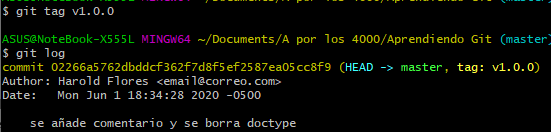


* + - * Hay que tomar en cuenta que así se haya solucionado el conflicto los index de las dos ramas son distintos ya que el index de la rama se mantendrá igual a su último commit.
      * Si se desea observar de una manera más gráfica se puede digitar el comando git log –-oneline –-decorate –-all --graph



1. **Etiquetas**

Para poder versionar nuestro código podemos hacer uso del comando git tag de la siguiente manera:



Así mismo podemos borrarlo mediante git tag -d



También podemos ponerle tag a cualquier commit, previamente creado, mediante la instrucción git tag -a versión [commitid] -m “[Mensaje]”



