Unidad de trabajo UD-3

Práctica 03 – Primeros comandos

Contenido

[1 Cambiar el nombre de la rama Master a Main 2](#_Toc120617096)

[1.1 Cambiar el nombre de una rama 2](#_Toc120617097)

[1.2 Configurar que por defecto la rama principal de un nuevo proyecto se llame main 2](#_Toc120617098)

[2 Archivo Readme.md y comando log 3](#_Toc120617099)

[2.1 Ver los commits realizados 4](#_Toc120617100)

[3 Adds y commits con visual studio code 5](#_Toc120617101)

[4 Diferentes formas de agregar archivos al escenario. 7](#_Toc120617102)

[4.1 Añadir archivos sueltos 7](#_Toc120617103)

[4.2 Añadir archivos usando el comodín \* 7](#_Toc120617104)

[4.3 Carpetas vacías 8](#_Toc120617105)

[4.4 Archivo .gitkeep 9](#_Toc120617106)

[4.5 Añadir una carpeta y todo su contenido 10](#_Toc120617107)

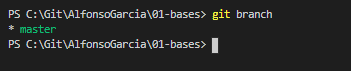
# Cambiar el nombre de la rama Master a Main

En general, una **rama de desarrollo** (“Git Branch”) es una bifurcación del estado del código que crea un nuevo camino para la evolución del mismo.

El comando git branch nos indica en que rama estamos trabajando:

git branch

**(Pega aquí una captura de la ejecución del comando)**



## Cambiar el nombre de una rama

Para cambiar el nombre de una rama usamos el siguiente comando:

git branch -m master main

El -m indica que se va a cambiar de nombre a una rama.

A continuación, ponemos el nombre de la rama a renombrar (en este caso master).

Finalmente indicamos el nuevo nombre (en este caso main).

**(Pega aquí una captura de la ejecución del comando)**

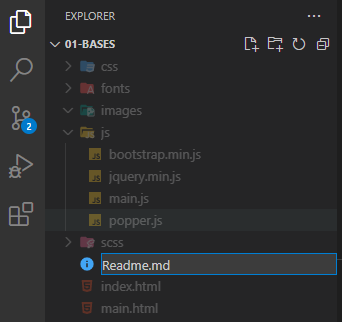


## Configurar que por defecto la rama principal de un nuevo proyecto se llame main

git config --global init.defaultBranch main

# 2 Archivo Readme.md y comando log

Vamos a crear un archivo nuevo en nuestro proyecto, al que llamaremos Readme.md:

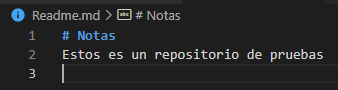


Pulsamos en el botón de nuevo archivo:

Y escribimos el nombre en el recuadro que aparece:

Un archivo README contiene información acerca de otros archivos en un directorio. Es una forma de documentación de software. En Github es habitual agregar un archivo README a un repositorio para comunicar información importante sobre el proyecto.

Escribimos algo de contenido en el archivo:



Le damos seguimiento al archivo:

git add Readme.md

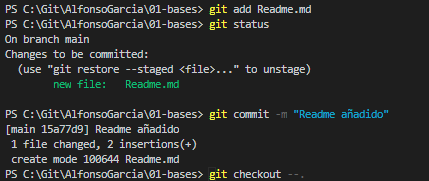
Hacemos commit:

git commit -m “Readme añadido”

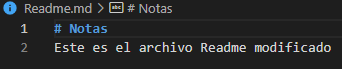
Borramos el archivo desde el visual estudio y como hemos hecho commit previamente podemos recuperarlo con:

git checkout--.

**(Pega aquí una captura de la ejecución de los comandos anteriores)**



Modificamos el archivo Readme:



Hacemos un segundo commit con una versión nueva del comando checkout:

git commit -am “Readme modificado”

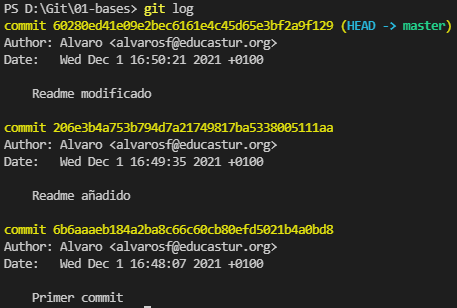
Este comando con –a sólo funciona si ya le estamos dando seguimiento al archivo, pero si estuviera marcado con la U de “untracked” es decir sin seguimiento no funcionaria.

**(Pega aquí una captura de la ejecución del comando)**



## Ver los commits realizados

Vamos a ver los commits que tenemos hechos hasta el momento, para ello vamos a utilizar el siguiente comando:

git log

**(Pega aquí una captura de la ejecución del comando)**

Vemos los tres commits que hemos hecho hasta ahora. La etiqueta head indica la última versión del repositorio:

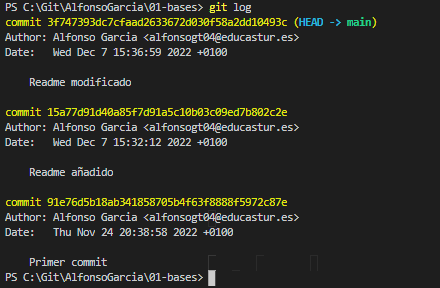
Identificador (hash) del commit

Autor del commit

Fecha y hora del commit

Texto identificativo del commit

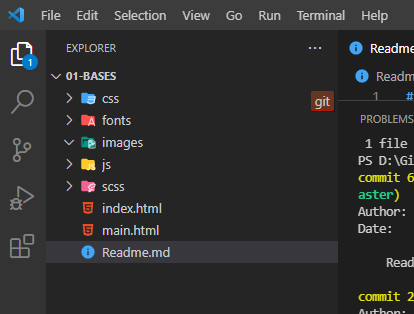
(**Pega aquí una captura de tus commits realizados**)



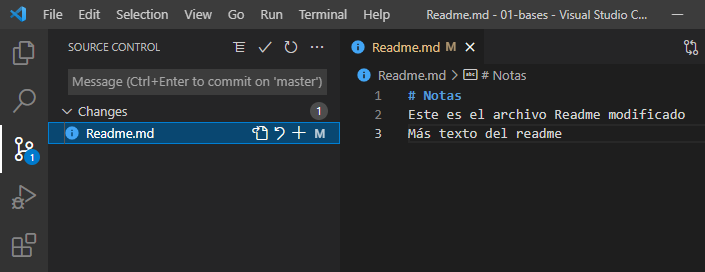
# 

# Adds y commits con visual studio code

En Visual studio code tenemos un apartado para realizar operaciones de git:

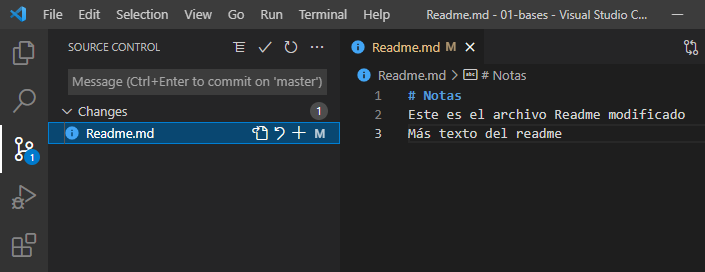


Con este icono abrimos el archivo:



Con este otro deshacemos los cambios realizados en el archivo:

Para añadir los cambios al escenario usamos este botón:



Y para hacer un commit escribimos el mensaje en este campo. Y a continuación utilizamos la combinación Ctrl+Enter o pulsamos en el botón de commit.

# 4 Diferentes formas de agregar archivos al escenario.

Cerramos en Visual Studio Code nuestro proyecto actual que era el de la carpeta 01-bases. Para ello vamos al menú File -> Close Folder o pulsamos la combinación Ctrl + K y luego F.

Descomprimimos el archivo 02-bases.zip y copiamos la carpeta 02-bases a la carpeta donde habíamos copiado la carpeta 01-bases.

Arrastramos esta carpeta a visual studio code como hicimos con 01-bases

Nos situamos en la carpeta 02-bases e inicializamos el repositorio:

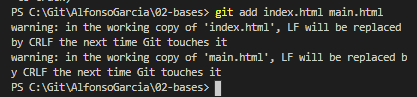
git init

Vamos a ver cómo hacer commits de grupos de archivos en vez de commit de todos los archivos como hicimos en apartados anteriores.

## Añadir archivos sueltos

git add index.html main.html

**(Pega aquí una captura de la ejecución del comando)**



Podemos añadir archivos sueltos simplemente escribiendo sus nombres separados por espacios.

## Añadir archivos usando el comodín \*

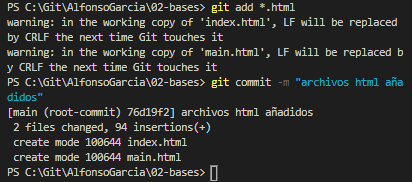
Podemos añadir todos los archivos html de la siguiente forma:

git add \*.html

A continuación hacemos commit de estos archivos

git commit -m “archivos html añadidos”

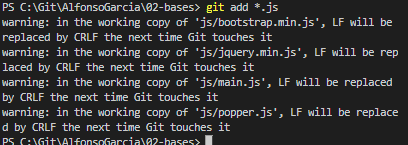
**(Pega aquí una captura de la ejecución de los comandos anteriores)**



Si ahora intentamos añadir todos los archivos .js

git add \*.js

**(Pega aquí una captura de la ejecución del comando)**



Esto no va a funcionar porque en el directorio actual no hay archivos .js , los archivos .js están en la carpeta js por lo que vamos a tener que indicarlo a la hora de ejecutar el comando:

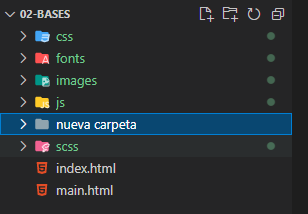
git add js/\*.js

**(Pega aquí una captura de la ejecución del comando)**

(**Nota**: Parece que en la versión actual si ponemos add \*.js si que busca en las carpetas a partir de la carpeta actual y añade todos los archivos .js)

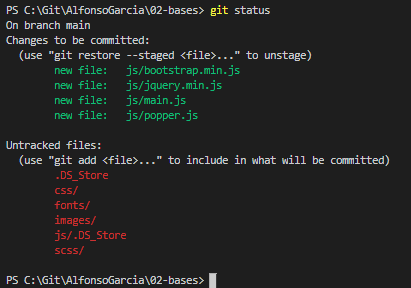
## Carpetas vacías

Git no hace seguimiento a las carpetas vacías. Vamos a crear una carpeta nueva para verlo.



Si hacemos ahora git status:

**(Pega aquí una captura de la ejecución del comando)**

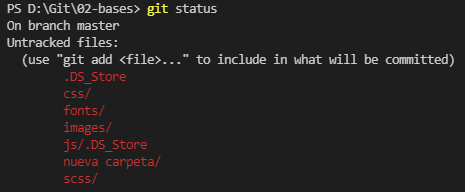


Vemos que git ni la muestra en la lista de archivos de los que no se hace seguimiento.

Si queremos que git la tenga en cuenta tenemos que crear un archivo dentro:



Y si ahora hacemos git status, vemos que ahora si aparece la nueva carpeta:



## Archivo .gitkeep

Cuando queramos añadir al menos un archivo a una carpeta vacía para que git la tenga en cuenta, en vez de crear cualquier archivo existe un archivo con un nombre especial que podemos crear: el archivo **.gitkeep**

Este archivo está especialmente pensado para realizar esta función y ocupa un espacio muy pequeño.

(**Nota:** Renombramos “nueva carpeta” como “nuevacarpeta”)



Vemos además que visual studio code le pone un icono específico al archivo con este nombre.

Añadimos este archivo con el siguiente comando:

git add nuevacarpeta/\*.gitkeep

**(Pega aquí una captura de la ejecución del comando)**



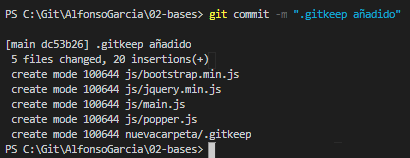
En algunas versiones este comando puesto así podría fallar y podríamos usar esta otra opción:

**(Pega aquí una captura de la ejecución del comando)**

Hacemos commit:

git commit -m “gitkeep añadido”

**(Pega aquí una captura de la ejecución del comando)**



Si borramos la carpeta y restauramos con:

git checkout -- .

**(Pega aquí una captura de la ejecución del comando)**



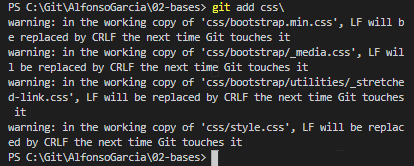
Vemos que se restaura tanto el archivo .gitkeep como la carpeta que lo contenía.

## Añadir una carpeta y todo su contenido

Si queremos añadir por ejemplo todo el contenido de la carpeta css (tanto archivos como directorios que contiene) podemos hacerlo con el siguiente comando:

git add css\

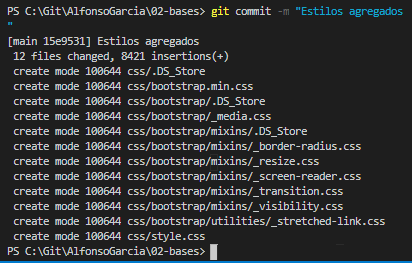
**(Pega aquí una captura de la ejecución del comando)**



Hacemos commit:

git commit -m “Estilos agregados”

**(Pega aquí una captura de la ejecución del comando)**



Práctica 04

Contenido

[1 Creando alias para nuestros comandos 2](#_Toc121820549)

[2 Cambios en los archivos 2](#_Toc121820550)

[2.1 Comparar cambios en archivos con visual studio code 4](#_Toc121820551)

[3 Actualizar mensajes de commits y deshacer commits 5](#_Toc121820552)

[3.1 Actualizar el mensaje de un commit 5](#_Toc121820553)

[3.2 Borrar un commit 5](#_Toc121820554)

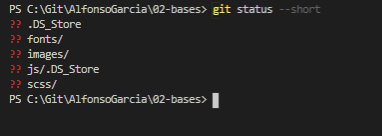
[4 Preparando un repositorio para viajes en el tiempo 5](#_Toc121820555)

# Creando alias para nuestros comandos

Para mostrar el estado de los archivos del repositorio con una descripción corta podemos usar el siguiente comando:

git status --short

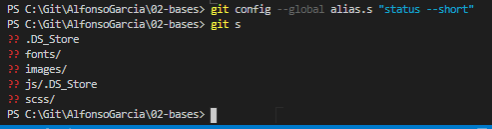
**(Pega aquí una captura de la ejecución del comando)**



Podemos crear un alias para este comando de forma que podamos ejecutarlo escribiendo menos texto:

git config --global alias.s “status --short”

**(Pega aquí una captura de la ejecución del comando)**



Como vemos para crear un alias usamos la configuración global y ponemos alias. y después del punto ponemos el texto que servirá de alias. A continuación escribimos entre comillas el comando que queremos abreviar sin poner git delante.

# Cambios en los archivos

Creamos un nuevo repositorio que se llame 03-instalaciones y lo abrimos en visual studio.

Hacemos el **git init** para inicializar el repositorio y creamos un archivo que se llame instalaciones.md con el siguiente contenido:

Lo agregamos al stage:

git add .

Y hacemos commit:

git commit -m “Instalaciones agregado”

Deberíamos ver algo así en nuestro visual studio code:

Modificamos el archivo instalaciones y lo dejamos así:

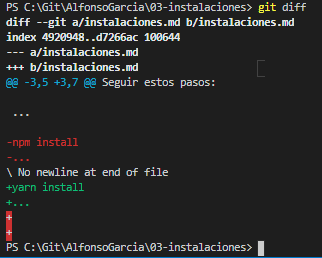
(Hemos modificado una línea y añadido 3 líneas vacías debajo de la segunda línea con …)

Guardamos los cambios, pero no hacemos add ni commit.

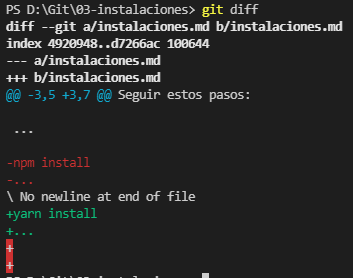
Vamos a comparar las modificaciones realizadas en este archivo, para ellos usamos el siguiente comando:

git diff

**(Pega aquí una captura de la ejecución del comando)**



La versión a del archivo instalaciones nos dice que tiene menos cosas que la versión b del mismo archivo



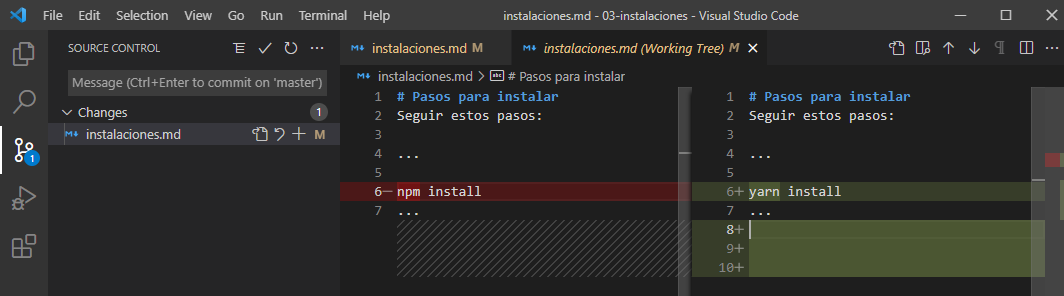
Las líneas en color rojo y precedidas por un - nos indican cosas la primera versión no tiene respecto al segundo y las cosas en verde precedidas por un + nos indican cosas que la segunda versión tiene y la primera no.

El comando git diff nos compara las modificaciones de los archivos que no están en el escenario. Si queremos usar el comando con un archivo que hemos añadido al escenario usaremos lo siguiente:

git diff --staged

## Comparar cambios en archivos con visual studio code

Desde la opción de source control de visual studio podemos comparar las modificaciones realizadas en un archivo de formas más visual:



Hacemos click en el archivo del cual queremos ver las diferencias

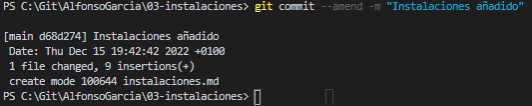
# Actualizar mensajes de commits y deshacer commits

## Actualizar el mensaje de un commit

Para cambiar el mensaje del último commit realizado usaremos el siguiente comando:

git commit --amend -m “Instalaciones añadido”

**(Pega aquí una captura de la ejecución del comando)**



## Borrar un commit

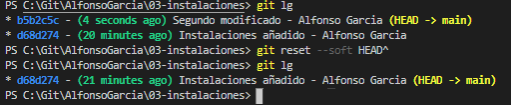
Modificamos el archivo instalaciones.md (escribimos cualquier cosa) y hacemos commit:

git reset -m “Segundo commit”

Para borrar el último commit realizado usaremos el siguiente comando:

git reset --soft HEAD^

**(Pega aquí una captura de la ejecución del comando)**

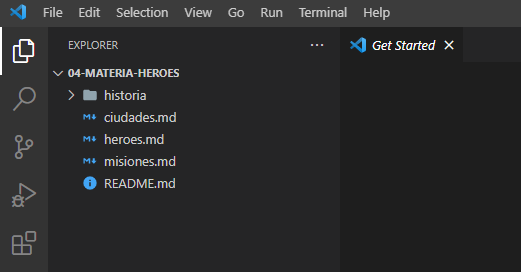


Esto eliminará el commit y nos dejará el repositorio como estaba antes de hacer dicho commit. Es decir si teníamos archivos modificados volverán a aparecer como modificados. El mantener los cambios se debe al parámetro --soft.

# Preparando un repositorio para viajes en el tiempo

Descomprimimos el archivo Materia-Heroes y colocamos la carpeta Materia-Heroes en la carpeta donde estemos creando nuestros repositorios.

Renombramos la carpeta como: 04-Materia-Heroes y arrastramos la carpeta a visual estudio code. Deberíamos tener algo así:



Vamos a ir ejecutando los siguientes comandos:

Inicializamos el repositorio:

git init

Añadimos el archivo README.md

Hacemos el commit de este archivo:

Añadimos el archivo misiones.md

git add misiones.md

Hacemos el commit de este archivo:

git commit -m “misiones.md agregado

Añadimos el archivo heroes.md

git add héroes.md

Hacemos el commit de este archivo:

git commit -m “héroes.md agregado”

Añadimos el archivo ciudades.md

git add ciudades.md

Hacemos el commit de este archivo:

git commit -m “ciudades.md agregado”

Agregamos la carpeta historia

git add /historia

Hacemos el commit de los archivos agregados:

git commit -m “carpeta historia agregada”

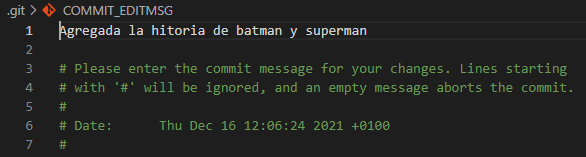
Cambiamos el texto del último commit:

git commit --amend

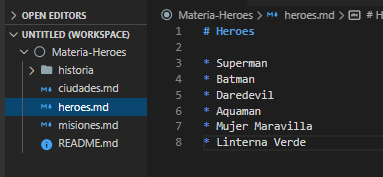
**(Pega aquí una captura de la ejecución del comando)**



Al ejecutar el comando sin poner al final –m “nuevo texto para el commit” se abrirá el editor y podremos cambiar en él el texto del commit:



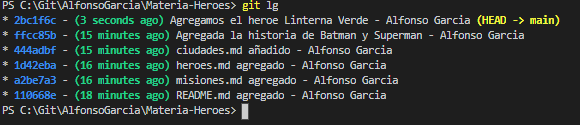
Editamos el archivo héroes.md y agregamos al héroe Linterna Verde:



Hacemos un nuevo commit:

git commit -am “añadimos el héroe linterna verde”

Ejecutamos el comando git lg para ver un listado de los commits que hemos realizado:

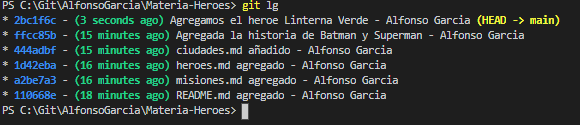


# Viajes en el tiempo, resets y reflogs

Vamos a añadir el héroe Robin en el archivo heroes.md:

En vez de hacer un nuevo commit para añadir este cambio queremos que haya un solo commit con la agragación de Robin y Linterna Verde.

Hacemos un git lg:



Vamos a regresar a un commit anterior usando como referencia el código hash del commit al que queremos volver:

git reset --soft ffcc85b

**(Pega aquí una captura de la ejecución del comando)**



Hacemos un nuevo commit:

git commit -am “agregamos a los héroes linterna verde y robin”

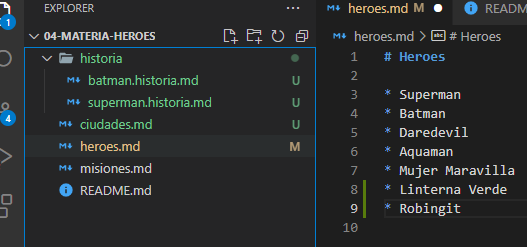
Ahora decidimos que los últimos cambios realizados no están bien y que queremos volver al punto en el que agregamos el archivo de heroes. Y además queremos que los archivos de los que se hubiera hecho commit se queden fuera del stage si es que los habíamos subido. Para ello hacemos un git reset --mixed:

git reset --mixed 1d42eba

**(Pega aquí una captura de la ejecución del comando)**



Si no ponemos una opción del modo de reset (--soft --mixed etc) el tipo de reset que se hace es el --mixed ya que es el modo por defecto.



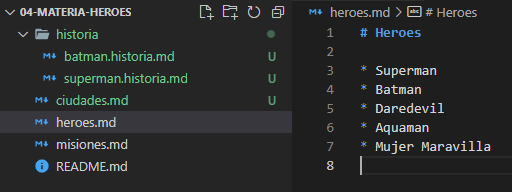
Vemos que los archivos han pasado a estar con la U de untracked es decir sin seguimiento pero que los últimos cambios realizados en heroes.md se han mantenido.

Si queremos hacer un reset que borre los cambios usaremos la opción --hard:

git reset --hard 1d42eba

**(Pega aquí una captura de la ejecución del comando)**





Vemos que el archivo heroes.md ha vuelto al estado del commit al que hemos regresado.

Hacemos un reset hard al commit donde agregamos las misiones:

git reset --hard a2be7a3

**(Pega aquí una captura de la ejecución del comando)**

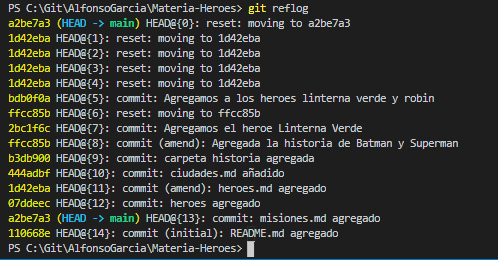


Si hacemos un nuevo git lg:



Vemos que sólo nos quedan dos commits. Si ahora nos damos cuenta de que estaba todo bien y que queremos volver a un commit de los que borramos, todavía es posible ya que git guarda un historial:

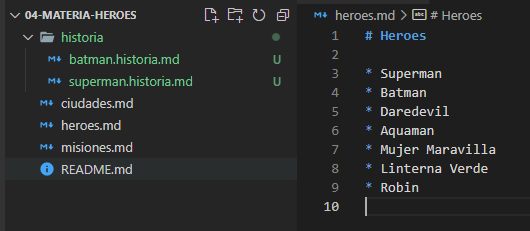
git reflog



Como podemos ver en este historial se guardan tanto los commit como los reset que se fueron realizando. Esto nos permite ver los hash de los commit borrados y si por ejemplo queremos volver al commit en el que agregamos a los héroes Linterna Verde y Robin no tenemos más que ejecutar el siguiente comando:

git reset --hard

Vemos que todo vuelve a ese punto:



Práctica 05

Contenido

[1 Cambiar el nombre y eliminar archivos con git 2](#_Toc124152358)

[2 Cambiar el nombre y eliminar archivos fuera de git 3](#_Toc124152359)

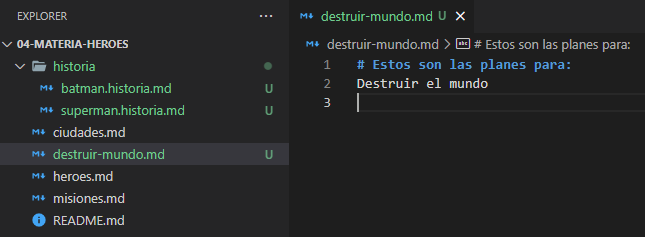
[2.1 Renombrar 3](#_Toc124152360)

[2.2 Borrar archivos 6](#_Toc124152361)

[3 Ignorar archivos que no deseamos 6](#_Toc124152362)

# Cambiar el nombre y eliminar archivos con git

Creamos mediante visual studio un nuevo archivo llamado “destruir-mundo.md”:



Guardamos los cambios del archivo.

Lo subimos al escenario:

git add destruir-mundo.md

Hacemos commit:

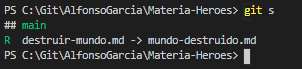
git commit -m “destruir-mundo.md añadido”

Vamos a cambiar el nombre a este archivo:

git mv destruir-mundo.md mundo-destruido.md

Aunque el comando git mv es para mover un archivo al moverlos a la misma ruta con diferente nombre lo que hacemos es renombrarlo.

Al renombrarlo vemos que si usamos el comando git s se muestra el archivo con una R que indica que fue renombrado:



**[Añade aquí una captura con la ejecución del comando git s]**

Visual studio también muestra una R de renombrado:



El archivo renombrado está además ya subido al stage listo para hacer commit:

git commit -m “mundo-destruido.md añadido”

Podemos también borrar un archivo con un comando de git:

Después de ejecutar este comando el archivo queda marcado con la letra D de borrado:

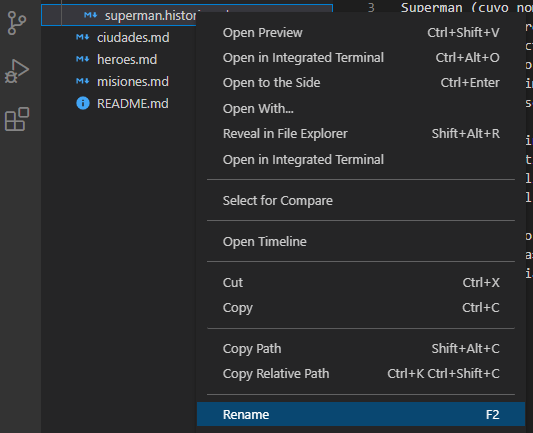
**[Añade aquí una captura con la ejecución del comando anterior]**

Para confirmar el borrado hacemos commit:

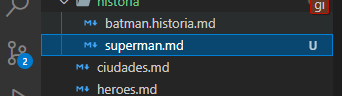
# Cambiar el nombre y eliminar archivos fuera de git

## Renombrar

Vamos a renombrar el archivo “superman.historia.md” desde la interfaz de visual studio. Pulsamos botón derecho sobre el archivo y elegimos la opción Rename:



Lo renombramos como “superman.md”



Vemos que el archivo renombrado queda marcado con la U de untracked (sin seguimiento) en vez de con la R de rename como sucedía en el apartado anterior al renombrar un archivo con el comando git.

Si hacemos un git status o git s vemos lo siguiente:

**[Añade aquí una captura con la ejecución del comando anterior]**

Para git es como si hubiéramos borrado el archivo original que como vemos queda marcado con una D, y el archivo renombrado lo marca con dos interrogaciones considerando que es un archivo nuevo.

Si ahora subimos los cambios al stage:

git add .

Si ahora hacemos git s

**[Añade aquí una captura con la ejecución del comando git s]**

Vemos que ahora el sólo aparece el archivo renombrado marcado con la R de renamend (renombrado). Al hacer git add . git ha analizado los archivos y se ha dado cuenta de que son el mismo y se ha producido un renombrado.

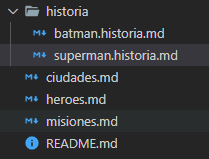
Hacemos commit de los cambios:

Hacemos un git lg para ver nuestro commits:

**[Añade aquí una captura con la ejecución del comando git s]**

Vamos a movernos al commit “Salvar-mundo borrado”:

Observamos que el archivo de historia de superman vuelve a tener su nombre original:



Si queremos deshacer el reset y volver al punto anterior hacemos:

**[Añade aquí una captura con la ejecución del comando git s]**

El comando reflog nos permite ver el commit “Historia de superman renombrada” que no veríamos con git log:



**[Añade aquí una captura con la ejecución del comando git s]**

## Borrar archivos

Borramos el archivo batman.hitoria.md desde visual studio pulsando botón derecho sobre él y eligiendo la opción Delete.

Si ejecutamos git s tras borrarlo:

**[Añade aquí una captura con la ejecución del comando git s]**

Vemos que el archivo aparece marcado con la D de deleted (borrado) pero no está subido al escenario como sucedía cuando borrábamos con un comando git.

Lo subimos al escenario:

git add .

Hacemos commit:

**[Añade aquí una captura con la ejecución del comando git s]**

# Ignorar archivos que no deseamos

Vamos a crear algunos carpetas y archivos de prueba para luego configurar que git no les de seguimiento.

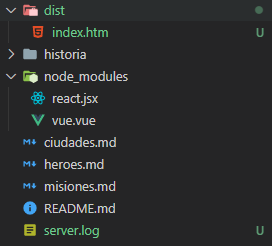
Creamos las carpetas:

* dist
* node\_module

Y los siguientes archivos:

* server.log
* dist/index.html
* node\_modules/react.jsx
* node\_modules/vue.vue

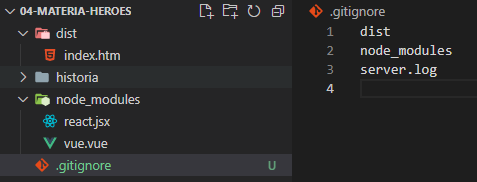
En los archivos podemos poner el texto que queramos o dejarlos vacíos.



Si hacemos git s vemos que git nos indica que los nuevos archivos no tienen seguimiento:

**[Añade aquí una captura con la ejecución del comando git s]**

Ahora lo que queremos es configurar que git ignore estos archivos porque no queremos darles seguimiento. Para ello creamos en la carpeta raíz del proyecto un archivo llamado **.gitignore** y dentro de este archivo escribiremos los nombres de archivos y directorios que queremos que git ignore. Podemos usar comodines para indicar los archivos:



Si después de escribir este archivo y guardar los cambios hacemos nuevamente git s:

**[Añade aquí una captura con la ejecución del comando git s]**

Vemos que efectivamente git está ignorando todos los archivos de las dos carpetas nuevas y el archivo server.log , sólo aparece como archivo sin seguimiento el archivo que hemos creado **.gitignore**

A este archivo si le vamos a dar seguimiento ya que queremos que se guarden las configuraciones que hemos hecho:

**[Añade aquí una captura con la ejecución del comando git s]**

Práctica 06

Contenido

[1 Ramas uniones y conflictos 2](#_Toc124153061)

[1.1 Tipos de merge 2](#_Toc124153062)

[1.2 Merge Fast-forward 3](#_Toc124153063)

[1.3 Merge unión automática 9](#_Toc124153064)

[1.4 Merges manuales (con conflictos) 13](#_Toc124153065)

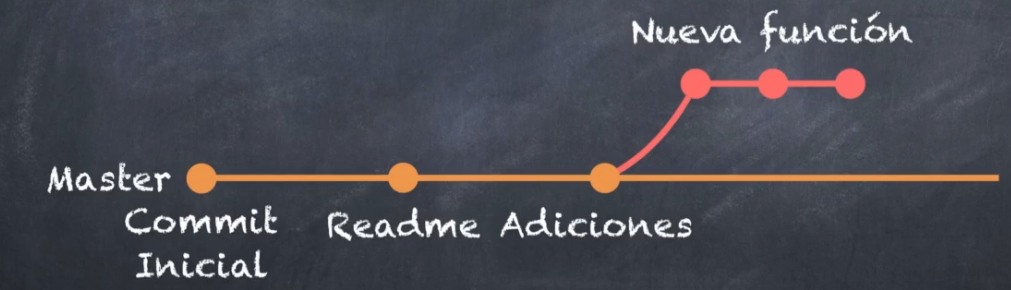
# Ramas uniones y conflictos

Una rama de un proyecto es una copia en la que empezamos a trabajar y a realizar modificaciones y commits. Desde el punto de vista de git podemos ver cada rama como un camino independiente de commits.

Hasta ahora hemos estado trabajando con sólo una rama que era la rama **master** que en un momento dado vimos como renombrar a **main** por ser más políticamente correcto:



Crear una nueva rama se puede ver visualmente como una bifurcación en el camino actual:



En un momento dado puede que queramos que una rama secundaria se integre en la principal a esto se le llama **unión** o **merge:**

****

## Tipos de merge

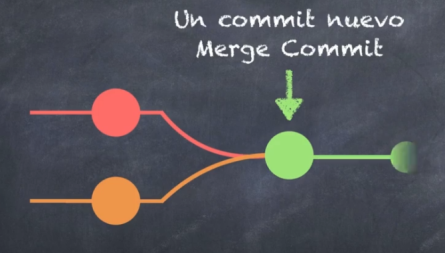
* **Fast-forward**: este tipo de unión se realiza cuando no ha habido cambios en la rama principal y por lo tanto se pueden unir sin problema las dos ramas simplemente agregando los commits de la rama secundaria a la principal:



* **Uniones automáticas**: este tipo de unión se realiza hay cambios en la rama principal pero que no entran en conflicto con los cambios realizados en la rama secundaria, por lo cual git puede realizar la unión de forma automática:

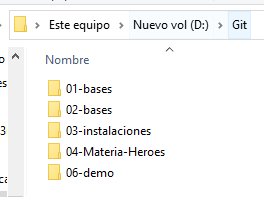


* **Manuales**: en este caso hay modificaciones en la rama principal y en la secundaria que entran en conflicto, por lo que git no puede decidir por sí mismo como hacer la unión y lo que hace es preguntarnos como realizar la unión y a continuación realizar un commit que se denomina **Merge Commit**

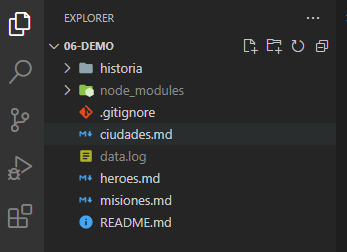


## Merge Fast-forward

Vamos a descomprimir el archivo de recursos 06-demo.zip en la carpeta de git donde estamos subiendo nuestros repositorios. Al final tendremos que tener una nueva carpeta que se llamará 06-demo:



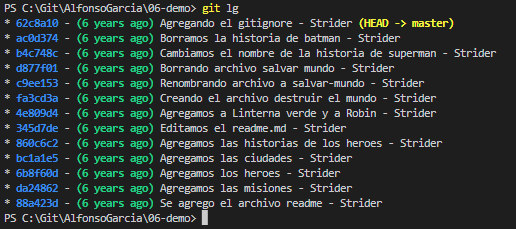
Cerramos la carpeta que tuviéramos abierta en visual studio code y arrastramos esta nueva carpeta 06-demo:



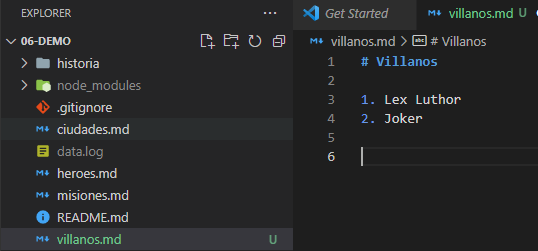
Este repositorio ya tiene la carpeta oculta .git es decir es un repositorio en el que ya se ha inicializado git y además tiene ya una serie de commits realizados y además tiene el archivo .gitignore también creado.

Vamos a ver los commits que tiene el repositorio en este momento:

git lg



Supongamos que estamos pensando en añadir al proyecto un apartado de villanos, aunque no estamos seguros de si al final esa parte se quedará en el proyecto o no. De momento vamos a crear un archivo llamados villanos.md con el texto que se ve en la imagen de más abajo:



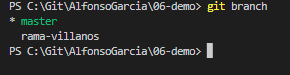
Vamos a crear una nueva rama que se llame rama-villanos:

git branch rama-villanos

Ahora para ver las ramas existentes y en cual estamos ahora mismo ejecutamos el comando:

git branch

**[Añade aquí una captura del resultado de la ejecución del comando anterior]**



El asterisco y el color verde nos indica que estamos en la rama master.

Para cambiar a la nueva rama escribimos:

git checkout rama-villanos

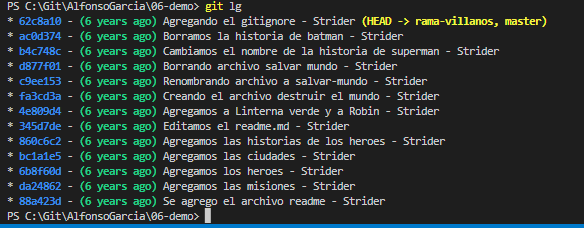
**[Añade aquí una captura del resultado de la ejecución del comando anterior]**



Como vemos un mensaje nos indica que nos hemos movido a la rama rama-villanos.

Si hacemos git lg vemos lo siguiente:

**[Añade aquí una captura del resultado de la ejecución del comando anterior]**



Vemos que la cabecera apunta a la rama rama-villanos y que la rama master está en el mismo punto.

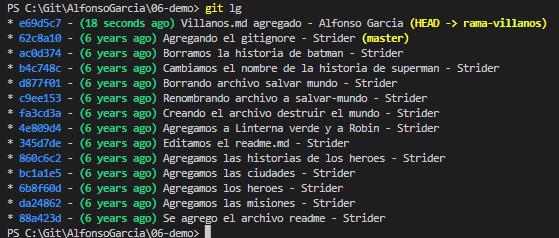
Agregamos el archivo al stage y hacemos commit:

git add .

git commit -m “Villanos.md agregado”

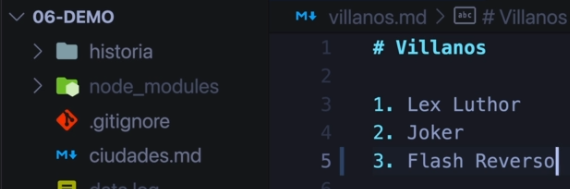
Si hacemos nuevamente git lg:

**[Añade aquí una captura del resultado de la ejecución del comando anterior]**



Vemos que el Head apunta a la rama rama-villanos y que la rama master se ha quedado un commit atrás.

Añadimos un nuevo villano al archivo de villanos:

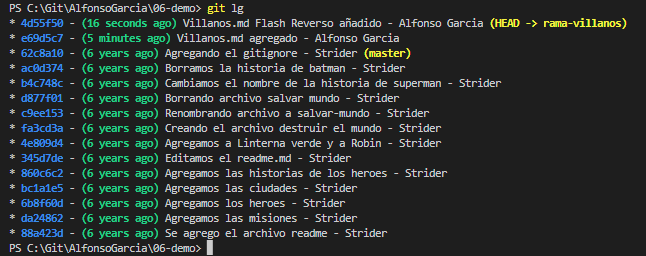


Hacemos un nuevo commit con este cambio usando la forma que sube el archivo al escenario y hace el commit a la vez:

git commit -am “Villanos.md Flash reverso añadido”

Hacemos nuevamente git lg:

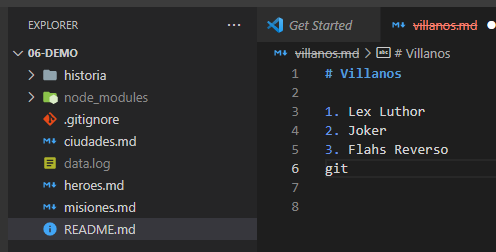
**[Añade aquí una captura del resultado de la ejecución del comando anterior]**



Vemos que la rama master se encuentra ya dos commits por detrás de la rama-villanos.

Si ahora cambiamos a la rama master:

git checkout master



Vemos que el archivo villanos.md desaparece ya que en la rama master no está, y además visual studio code nos indica que la versión que teníamos abierta de villanos.md corresponde a un archivo borrado y por eso nos lo muestra en color rojo y tachado.

Si vuelvo a la rama rama-villanos el archivo volverá a aparecer:

git checkout rama-villanos

Ahora decidimos que los cambios implementados en esta funcionalidad de villanos son correctos y por lo tanto queremos integrarlos en la rama principal que es la rama master. Al hacer la unión de ramas hay que tener en cuenta que rama tenemos seleccionada, ya que la rama seleccionada recibirá los cambios de la otra rama.

Si hacemos git branch:

**[Añade aquí una captura del resultado de la ejecución del comando anterior]**



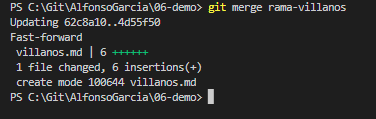
Vemos que ahora mismo está seleccionada la rama rama-villanos y de hacer la unión ahora los cambios de la rama master pasarían a la rama-villanos y lo que queremos es agregar los cambios de la rama rama-villanos en la rama master. Por lo tanto cambiamos de rama:

git checkout master

Hacemos la unión o merge:

git merge rama-villanos

**[Añade aquí una captura del resultado de la ejecución del comando anterior]**

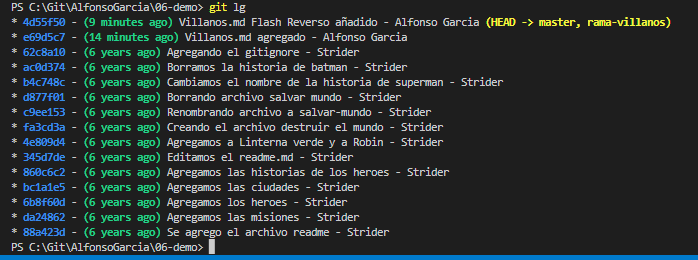


Vemos que se ha realizado un merge fast-forward que es la mejor opción porque implica que no ha habido conflictos y git ha podido hacer la unión sin ningún tipo de problemas.

También nos indica que al hacer el merge se incluyó el archivo villanos.md y 7 modificaciones.

Si hacemos una vez más git lg:

**[Añade aquí una captura del resultado de la ejecución del comando anterior]**



Vemos que después del merge las dos ramas están en el mismo punto

Como hemos terminado el desarrollo que estábamos haciendo en la rama rama-villanos y ya hemos unido su funcionalidad con la rama master, no necesitamos ya la rama rama-villanos. Vamos por lo tanto a borrar esta rama:

git branch -d rama-villanos

**[Añade aquí una captura del resultado de la ejecución del comando anterior]**



Si hubiera cambios en una rama que queremos borrar que no han sido unidos a otra rama, git nos avisará por si estamos borrando la rama por error. En ese caso podríamos forzar el borrado de la rama añadiendo el parámetro –f:

## Merge unión automática

Vamos a trabajar en unas modificaciones sobre el archivo de villanos y para ello vamos a crear una nueva rama. En el apartado anterior vimos como crear una rama y posteriormente movernos a ella usando por tanto dos comandos distintos. Vamos a ver ahora como crear una rama y movernos a la misma con un solo comando:

git checkout -b rama-villanos

**[Añade aquí una captura del resultado de la ejecución del comando anterior]**



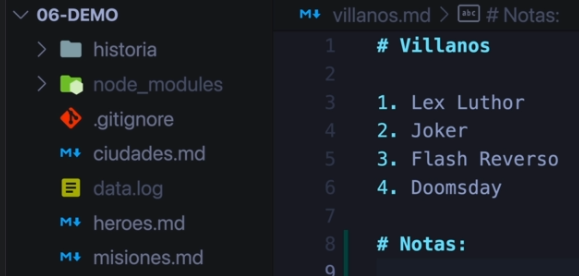
A continuación modificamos el archivo villanos.md y guardamos los cambios:



Hacemos commit de estos cambios:

git commit -am “Villanos.md Doomsday añadido”

Hacemos otra modificación en el archivo villanos.md:

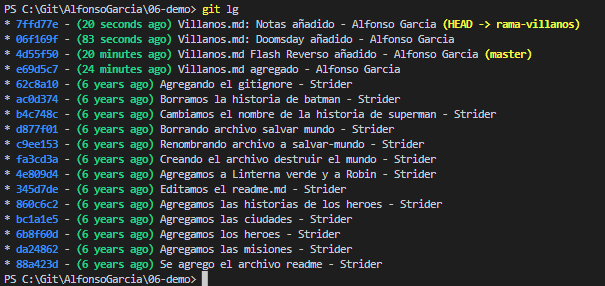


Y realizamos un nuevo commit:

git commit -am “Notas agregado”

Hacemos git lg:

**[Añade aquí una captura del resultado de la ejecución del comando anterior]**



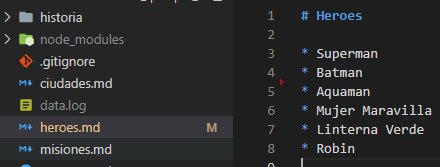
Vemos que la rama master está dos commits por detrás de rama-villanos.

Ahora nos piden que hagamos un cambio en la rama master ya que hay un error en el archivo de heroes ya que que Daredevil es un héroe de Marvel y queremos tener sólo heroes del universo DC en ese archivo.

Nos cambiamos a la rama master:

git checkout master

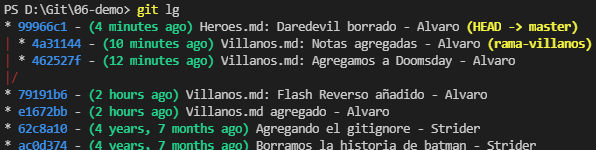
Borramos Daredevil del archivo de heroes y guardamos los cambios:



Hacemos commit de este cambio:

git commit -am “Heroes.md : Daredevil borrado”

Si hacemos git lg:

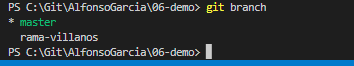


Vemos que la cabecera está en la rama master y que la rama rama-villanos está un commit por detrás. Además visualmente vemos que hay dos ramas.

Vamos a realizar la unión de estas dos ramas:

Queremos unir los cambios de rama-villanos a la rama master por lo que comprobamos si estamos en ella con git branch:

**[Añade aquí una captura del resultado de la ejecución del comando anterior]**

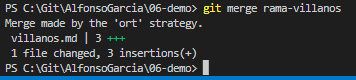


En esta ocasión ya nos encontrábamos en la rama master ya que estuvimos trabando en ella modificando el archivo de heroes.

Hacemos el merge:

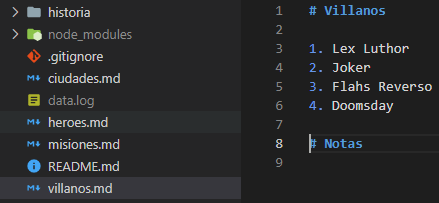
git merge rama-villanos

**[Añade aquí una captura del resultado de la ejecución del comando anterior]**

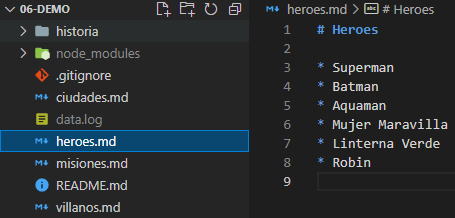


Vemos que git indica que ha realizado la unión usando la estrategia ‘recursive’ que es a la que estamos llamando en este apartado unión automática, en la cual ha habido cambios en la rama master pero no hay conflictos y git ha podido hacer la unión sin preguntarnos nada.

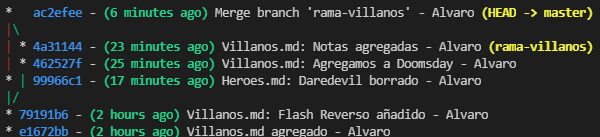
Después de hacer la unión podemos ver que tenemos el archivo de villanos con los 4 villanos y el apartado de notas:



Y que en el archivo héroes no tenemos a Daredevil:



Si hacemos una vez más git lg:



Vemos los dos caminos de commits y como se uniroen en el último commit realizado al llevar a cabo la unión de las dos ramas.

## Merges manuales (con conflictos)

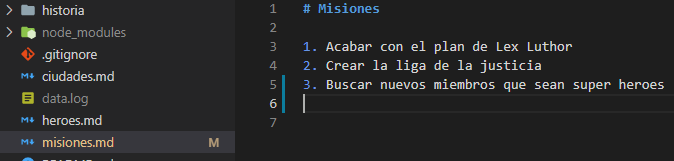
Vamos a crear una nueva rama que se va a llamar rama-conflicto:

git checkout -b rama-conflicto

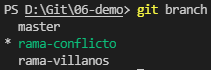
**[Añade aquí una captura del resultado de la ejecución del comando anterior]**



Vamos a modificar el archivo misiones.md y guardar los cambios:



Comprobamos que estamos en la rama rama-conflicto:



Si nos aparece la rama-villanos del apartado anterior como en la imagen la borramos ya que para este apartado no la necesitamos:

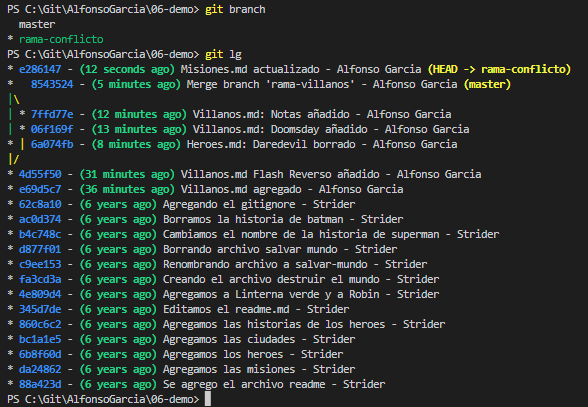


Hacemos commit de los cambios realizados en el archivo misiones.md:

git commit -am “Misiones.md actualizado”

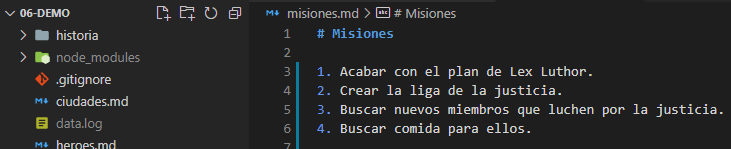
Miramos el estado actual:

**[Añade aquí una captura del resultado de la ejecución del comando anterior]**



Vamos a regresar a la rama master y realizar cambios en el mismo archivo villanos.md:





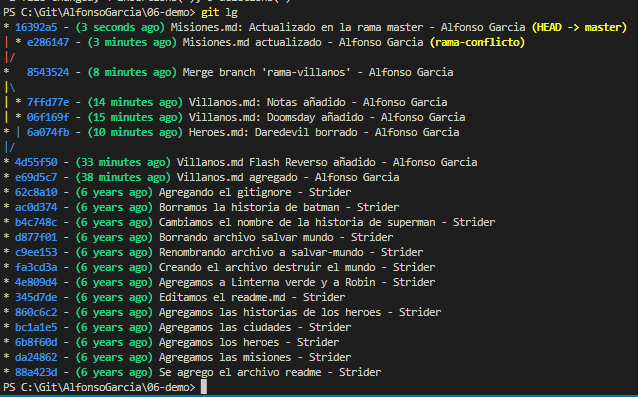
(Nota: se ha cambiado la línea 3 y añadido la línea 4, además **se han añadido puntos en todas las líneas**)

Hacemos commit de estos cambios:

git commit -am “Misiones.md: Actualizado en la rama master”

Comprobamos el estado actual:

**[Añade aquí una captura del resultado de la ejecución del comando anterior]**



Ahora mismo tenemos dos ramas que tienen el mismo archivo modificado, con modificaciones diferentes en cada una de las ramas.

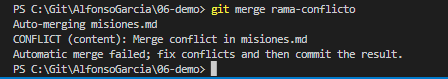
Vamos a realizar la unión o merge. Nos aseguramos que estamos en la rama master ya que queremos traer los cambios de la rama rama-conflicto a la rama master:



Realizamos el merge:

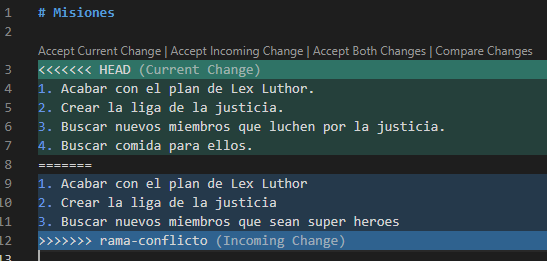
git merge rama-conflicto

**[Añade aquí una captura del resultado de la ejecución del comando anterior]**



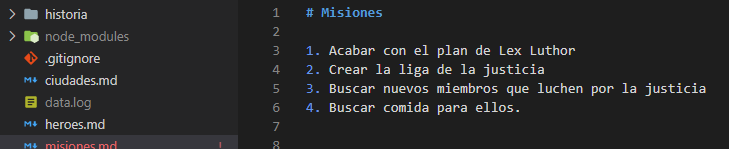
Git nos muestra un error. Nos dice que ha intentado hacer el merge automático (Auto-merging) pero que no es posible realizarlo porque hay conflictos que hay que resolver de manera manual.

Si abrimos el archivo misiones.md veremos algo así (si no nos aparece algo parecido puede deberse a que lo teníamos abierto con anterioridad, en ese caso debemos cerrarlo y volverlo a abrirlo):



Vemos las dos versiones del archivo. En color verde nos indica que es la versión actual ya que es la de la rama en la que estamos. En color azul nos marca la versión de la rama rama-conflicto y nos dice que es la versión entrante (incoming).

La forma de resolver el conflicto de forma manual es comparar visualmente ambas versiones e ir editando el archivo hasta dejar la versión que queramos. Por ejemplo podemos dejar una mezcla de las dos versiones con las 4 líneas pero quitando los puntos:



Guardamos los cambios.

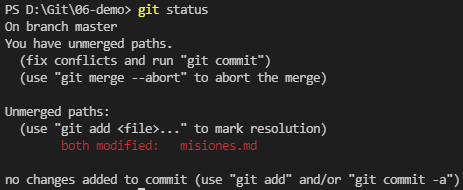
Si ahora hacemos git s:

**[Añade aquí una captura del resultado de la ejecución del comando anterior]**



Vemos que git nos marca el conflicto en la unión de las dos versiones del fichero con dos U en color rojo.

Si lo vemos con la versión git status:



Vemos que git nos da más información y nos dice que tenemos un conflicto que tenemos que resolver y hacer luego commit. También nos dice que podemos deshacer el merge usando git merge –abort

Como ya hemos arreglado el conflicto de forma manual editando el fichero con las dos versiones sólo nos falta hacer el commit:

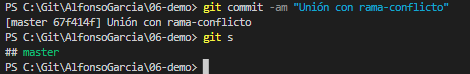
git cmmit -am “Unión con rama-conflicto”

**[Añade aquí una captura del resultado de la ejecución del comando anterior]**

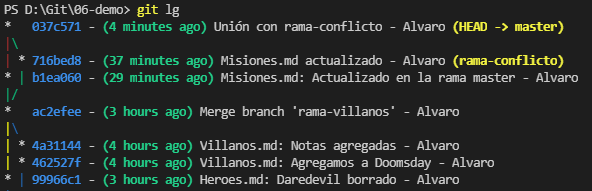


Si hacemos git s nuevamente vemos que ya se resolvió el conflicto:

**[Añade aquí una captura del resultado de la ejecución del comando anterior]**



Y si miramos el estado actual:



Vemos que ya tenemos el commit en el que hemos unido la rama master con la rama rama-conflicto.

Como ya no necesitamos la rama rama-conflicto la borramos:

git branch -d rama-conflicto

**[Añade aquí una captura del resultado de la ejecución del comando anterior]**



# Práctica 7.

