En esta lección aprenderemos a inicializar un repositorio GIT y a probar (commit) ficheros en el control de versiones.

El control de versiones te permite grabar cambios en un fichero o grupo de ficheros permitiendo volver atrás a una versión específica si se requiere.

Git es un Sistema de Control de Versiones Distribuido (Distributed Version Control System). Esto quiere decir que, en lugar de disponer de una instantánea de los últimos ficheros, disponemos de un espejo (mirror) completo del repositorio en nuestra máquina local. El repositorio mantiene un registro de los cambios, cuándo sucedieron y quién los efectuó. Tener todo el repositorio en nuestra máquina local reduce los retrasos provocados por tráfico de red y te permite seguir trabajando cuando estamos desconectados.

Paso 1 - Git Init

Para almacenar un directorio bajo control de versiones, se necesita crear un repositorio. Con GIT se inicializa un repositorio en el primer nivel del directorio de un proyecto.

Aprovecharemos esta práctica para introducir los settings iniciales de Git.

Tarea

Introducir previamente los settings iniciales de Git que vamos a utilizar.

Posteriormente, iniciaremos un nuevo proyecto, y será necesario crear un nuevo repositorio. Se usará el comando git init para crear el repositorio.

Avanzado

Después de inicializar un repositorio, se crea un nuevo subdirectorio oculto *.git* . Este subdirectorio contiene los metadatos que Git utiliza para almacenar su información. Explorar los contenidos del mismo para ver su contenidos.

Paso 2 - Git Status

Cuando un directorio es parte de un repositorio, se le llama *Working Directory*. Un directorio de trabajo o *working directory* contiene la última versión descargada desde el repositorio junto con cualquier cambio que tenga que ser aprobado. Al estar trabajando en un proyecto, todos los cambios se realizan en el *working directory*.

Podemos ver los cuales de los ficheros han cambiado en el *working directory* y qué cambios van a ser aprobados (*committed*) al repositorio usando el comando git status.

La salida de este comando recibe el nombre de "*working tree status*".

Tarea

En nuestro espacio de trabajo, o directorio scn1, creamos un archivo README.md y un par de archivos más.

A continuación, lanzamos el comando git status

Avanzado

Todos los ficheros que creamos en nuestro espacio aparecen como "*untracked*" o no considerados por Git mientras no le digamos lo contrario. Los detalles para resolver esto se cubren en el siguiente paso.

Paso 3 - Git Add

Para guardar, o aprobar (commit) ficheros en nuestro repositorio GIT, primero se necesita añadirlos al *stagging area* (área de preparación). Git tiene tres áreas: un *working directory*, un *staging area*, y el propio repositorio en sí (directorio oculto .git). Los usuarios "mueven" (también se refiere a ello como "promueven/*promote*") cambios desde el *working directory* al área de preparación o *staging área*, antes de aprobarlos/*commit* hacia el repositorio.

Uno de los enfoques clave con Git consiste en que los *commits* están concentrados, son pequeños y frecuentes. El área de preparación/staging area ayuda a mantener este flujo de trabajo permitiendo promover solo ciertos ficheros cada vez en lugar de obligar a subir todos los cambios del directorio de trabajo.

Tarea

Dentro del directorio en el que estamos, al que previamente hemos hecho git init, creamos un fichero "hello-world.js" con este contenido:

```
console.log("Hello World");
```

Luego, usa el comando git add para agregar *hello-world.js*a la *staging area*.

Avanzado

Importante: si hacemos cualquier otro cambio adicional a ese fichero posterior a la ejecución de *git add* (es decir, después de agregarlo a la *staging area*), el cambio no se reflejará hasta que no lances *git add* otra vez.

Protip

Tal y como se describió en el paso 2, el comando git status te permite ver el estado tanto del *working directory* como de la *staging area* en cualquier momento.

Paso 4 - Git Commit

Una vez que el fichero ha sido añadido a la satging area, se necesita que sea aprobado (commited) hacia el repositorio. El comando git commit -m "commit message" mueve ficheros desde la staging area o área de preparación hacia el repositorio, y almacena fecha/hora, autor y un mensaje de aprobación que puede usarse para agregar información de contexto adicional a los cambios, como por ejemplo un identificador de error, o número identificador de *bug*.

Solo los cambios añadidos a la staging area serán aprobados/*commited* ; cualquier fichero en el *working directory* que no haya sido añadido con git add (es decir, cualquier fichero no *staged*), no se incluirá en el commit de cambios.

Tarea

Usamos el editor nano o vim para crear un fichero README2.md

Le agregamos contenido. Lo añadimos a staging area.

Use git commit -m “” para aprobar el fichero que hemos creado.

Ahora creamos otros dos ficheros. En ambos introducimos contenido.

Agregamos a staging **SOLO** uno de ellos.

Lanzamos git commit.

Posteriormente, comprobamos que quedan ficheros que no han subido al repositorio.

Avanzado

A cada commit se le asigna un hash SHA-1 que posibilita identificarlo para usos futuros. Obsérvalo.

Paso 5 - Git Ignore

A veces, hay ficheros o directorios que nunca querremos aprobar aunque estén en nuestro *working directory* , como por ejemplo configuración local de desarrollo. Para ignorar estos ficheros, usamos un fichero *.gitignore* lal raíz del repositorio.

El fichero *.gitignore* te permite definir máscaras o *wildcards* para los ficheros que deseamos ignorar; por ejemplo, el *wildcard* “*.tmp” permitiría ignorar a todos los ficheros con extensión* tmp* .

Cualquier fichero que corresponda a un *wildcard* no será mostrado en la salida de un git status y por lo tanto será ignorado cuando lancemos un comando git add.

Tarea

Construye con nano un fichero en la raíz del repositorio que permita ignorar los ficheros *.tmp.

Comprueba que el filtro construido funciona (crea ficheros con y sin extensión .tmp , intenta agregarlos a *staging* y haz commit).

Avanzado

IMPORTANTE: el propio .gitignore debería ser aprobado/*committed* al repositorio para asegurar que las reglas se cumplen en máquinas diferentes, ya que si no, solo se aplicaría a tu repositorio local.

Aún con el .gitignore, ¿existe alguna forma de agregar de forma forzada un tmp?

Paso 1 - Git Status

Tal y como se ha discutido en la lección previa, git status nos permite visualizar los cambios en el *working directory*y en la *staging area* comparando con el repositorio.

Dado que el repositorio actual en el que lanzamos git status muestra que un cambio hecho en nuestro *working directory* y tenemos en él un fichero previamente aprobado (por ejemplo hello-world.js), si lo modificamos, y a la vez creamos otro fichero, commited.js, que ni siquiera ha sido movido aún a *staging area*, ¿qué nos mostrará?

Tarea

Modificamos fichero “hello-world.js” agregándole un comentario.

Creamos un fichero commited.js

Lanzamos

```
git status
```

Paso 2 - Git Diff

El comando git diff permite comparar cambios en el *working directory* contra una versión previamente aprobada. Por defecto, el comando compara el *working directory* y el commit que llamamos *HEAD* (el más reciente).

Si se desea comparar el *working directory* contra una versión más antigua, entonces podemos proporcionar su *hash_id* como parámetro en la llamada al comando; así: git diff .

Comparar contra *commits* mostrará los cambios en todos los ficheros que se hayan modificado. Si deseamos comparar los cambios en un único fichero, podemos pasar su nombre como parámetro:

```
git diff committed.js
```

Incluso ambas funcionalidades, así:

```
git diff <commit> committed.js.
```

Avanzado

Por defecto, la salida está en formato diff.

By default the output is in the combined diff format. The command git difftool will load an external tool of your choice to view the differences.

En nuestro caso, elegiremos tkdiff, descargándola y colocándola donde se ha indicado arriba.

Si queremos vimdiff:

```
git config --global diff.tool vimdiff
```

Si queremos tkdiff:

```
git config --global diff.tool vimdiff
```

Tarea

Hagamos una prueba modificando un fichero respecto de la versión anterior que esté aprobada.

Ejecutemos:

```
git diff git difftool
```

Y veamos las posibilidades que tiene la herramienta.

Paso 3 - Git Add

Anteriormente, vimos que podíamos aprobar un cambio mediante *commit* siempre y cuando previamente hubiésemos agregado el archivo al área de preparación (stage) mediante el comando git add .

Avanzado

Si renombramos (*mv*) o borramos (*rm*) ficheros, se necesita especificar esos ficheros con un git add para que sean registrados como un cambio. Esto a veces resulta confuso.

De forma alternativa, se puede usar git mv y el comando git rm de git para llevar a cabo la acción sobre el fichero, y además incluir el cambio en la *staging area*.

Tarea

Practicamos agregando cambios con el comando git add

Practicamos haciendo renombrado y renombrado sin “git mv” y sin “git rm”. ¿Qué sucede?

Lo solucionamos con git add.

Ahora repetimos el proceso con git mv y el comando git rm. ¿Qué sucede?

Paso 4 – Diferencias en estado preparado (Staged)

Una vez que los cambios están en la *staging area* no se mostrarán en la salida de git diff

Por defecto, git diff solo comparará el *working diretory* y no el *staging area* .

Para comparar cambios en el *staging area* contra el *commit* previo, debemos proporcionar el parámetro git diff —staged . Esto posibilita asegurarnos de que hemos preparado (*staged*) todos nuestros cambios.

Tarea

- Lanzamos git rm README.md y efectuamos un commit.
- Ahora lanzar “git diff” y “git diff —staged” y ver si hay diferencia.
- Creamos un archivo README.md con nano y agregamos contenido.
- Posteriormente, hemos de lanzar “git diff” y “git diff —staged” y ver si hay diferencia.
- Ahora lanzamos git add de ese fichero recién creado.

Si lanzamos ahora ambos comandos, ¿hay alguna diferencia?

Paso 5 - Git Log

El comando git log permite visualizar el histórico del repositorio y el *commit log* .

Avanzado

El formato de la salida de log es muy flexible. Por ejemplo para sacar cada *commit* en una única línea, el comando es:

```
git log --pretty=format:"%h %an %ar - %s"
```

También funciona así:

```
git log --pretty="%h %an %ar - %s"
```

Podemos encontrar más detalles y opciones usando:

```
git log --help
```

Tarea

Probar la salida del comando git log agregando las opciones descritas.

Paso 6 - Git Show

Mientras que git log te dice el autor del *commit* y el mensaje, para visualizar los cambios efectuados en el *commit* se necesita usar el comando git show .

Al igual que otros comandos, por defecto, “git show” muestra los cambios del *commit* HEAD. Podemos usar el comando git show para visualizar cambios más antiguos.

Tarea

Usar git log para obtener varios números de commit. Lanzar git show con dos de ellos.

Git .gitignore

Ayuda a excluir a los archivos que no quieres que se incluyan en un add, por ejmplo ficheros tmp.

Nano .gitignore

****.tmp***

ver las diferencias en git diff

Ayuda a ver los cambios ejecutados.

En esta lección aprenderemos cómo podemos compartir cambios en nuestro repositorio con otras personas, y combinar sus cambios dentro de nuestro repositorio.

Con Git como DVCS, tenemos nuestro repositorio local conteniendo todos los logs, ficheros y cambios realizados desde que se inicializó el repositorio. Para asegurar que todos están trabajando en la copia más reciente, es necesario compartir los cambios. Cuando compartimos estos cambios con otros repositorios, solo se sincronizarán las diferencias, por lo que es un procedimiento extremadamente rápido.

Paso 1 - Git Remote

Los repositorios remotos permiten compartir cambios desde o hacia nuestro repositorio. Las ubicaciones remotas son generalmente servidores locales, una máquina de un equipo de trabajo o bien un almacén de repositorios en la nube como GitLab o Github.

Los repositorios remotos se añaden usando el comando git remote con un nombre amigable y una ubicación remota; normalmente una conexión HTTPS o SSH (para esto último no hace falta software específico como GitLab).

Ejemplo: **https://github.com/sharkdp/bat**

El nombre amigable permite referenciar la localización en otros comandos. Nuestro repositorio local puede referenciar múltiples repositorios remotos, dependiendo de nuestras necesidades.

Tarea

Trabajaremos en otro directorio. Crearemos la carpeta scn3 y esa será nuestro *working directory*.

Agregaremos el repositorio remoto **https://github.com/sharkdp/bat** usando el comando git remote con el nombre amigable origin.

Formato de llamada:

```
git remote add origin /s/remote-project/1 git fetch origin
```

Avanzado

Si usamos *git clone* que será tratado en una lección futura, cuando se clona el repositorio, se agrega automáticamente como descriptor amigable que lo referencia, el nombre *origin*.

Paso 2 - Git Push

Cuando estamos listos para compartir nuestros *commits*, es necesario hacer un *push* de ellos a un repositorio remoto usando *git push*. Un flujo de trabajo habitual de GIT sería llevar a cabo múltiples *commits* pequeños conforme vamos finalizando tareas, y hacerles un *push* a un repositorio remoto en hitos relevantes, como cuando finalizamos un bloque de trabajo, de manera que aseguramos la sincronización del código dentro de un equipo.

El comando *git push* se acompaña de dos parámetros. El primero es el nombre amigable del repositorio remoto (normalmente *origin*). El segundo es el nombre de la rama (normalmente,

rama *master*). Por defecto, todos los repositorios tienen una rama *master* donde se trabaja con el código.

Tarea

Haremos *push* de los *commits* de la rama *master* a la ubicación remota (*origin*).

Para este ejercicio, vamos crearnos un usuario en gitlab. Crearemos un nuevo proyecto y nos vincularemos a él. Hacemos un pull. Posteriormente, crearemos un archivo (por ejemplo README2.md), y lo aprobaremos *commit* en nuestro master. Por último, lo subiremos al repositorio de Gitlab

Paso 3 - Git Pull

El comando *git push* permite subir los cambios a un repositorio remoto, mientras que *git pull*funciona de forma inversa. El comando *git pull* permite sincronizar cambios de un repositorio remoto en nuestra versión local.

Los cambios desde el repositorio remoto son automáticamente fusionados (*merge*) en la rama en la que estamos trabajando en el momento de lanzar el comando.

Tarea

- Crear un nuevo repositorio llamado scn3-pull.
- Descargar (*pull*) los cambios de una ubicación remota **https://github.com/sharkdp/bat** a nuestra rama *master*.
- Borrar los archivos siguientes appveyor.yml Cargo.lock Cargo.toml LICENSE-APACHE LICENSE-MIT README.md
- Agregar un archivo LEEME.txt
- Aprobar los cambios.
- Lanzar un “ls” para ver los archivos de la rama *master*.
- Hacer un *checkout* para cambiarnos a *origin/master*.
- Lanzar un “ls” para ver los archivos de la rama *origin/master*.
- Conmutarnos de nuevo a la rama master y volver a lanzar un comando “ls”

En el siguiente paso exploraremos qué cambios se han efectuado.

IMPORTANTE: Podemos crear y cambiar de rama con un mismo comando.

```
git checkout -b nombre-rama
```

Esto nos capacita para descargar una versión de repositorio, y luego versionar

Paso 4 - Git Log

Tal y como se ha descrito en la lección previa, podemos usar *git log* para ver el histórico del repositorio. El comando *git show* permitirá ver los cambios realizados en cada *commit*.

En esta prueba, vamos a usar el repositorio anterior, que tenemos vinculado al proyecto BAT y vamos a filtrar los commits que lleven la cadena Fix o fix. Al más reciente de los cuales, le lanzaremos un *git show* <*hashid*> para visualizar los cambios.

Avanzado

Usar el modificador –pretty para llevar a cabo la consulta de commits en el repositorio remoto de BAT

Git clone:
Git init (crea repositorio)
Git fech (compreuba)
Git pull (descarga todo)

Una de las ventajas principales cuando usamos un sistema de control de versiones es la capacidad de deshacer cambios y volver a una versión previa. Git proporciona enfoques poderosos y control sobre la gestión de los repositorios y su histórico. Exploraremos esas posibilidades en esta lección.

Notas previas:

hace falta crear un repositorio git con algunos commits previos

Paso 1 - Git Checkout

Cuando trabajamos con Git, una lección habitual es deshacer cambios en nuestro *working directory*. El comando *git checkout* reemplazará todo en el *working directory* a la última versión aprobada (*commited*).

Si deseamos reemplazar todos los ficheros hacia el *working directory*, usamos el carácter punto (“.”) para referenciar al directorio actual. En otro caso, se proporciona una lista de directorios/ficheros separados por espacios.

Como sucede que *git checkout* acepta tanto nombres de rama, como ficheros (o path de ellos), se puede usar el indicador “—” para forzar a Git a interpretar que cualquier cosa que venga después de “—”, es un nombre de fichero (o todos los ficheros); en otro caso, se interpretará como el nombre de una rama.

Se ilustra en este ejemplo la diferencia:

Supongamos que tengo un archivo llamado *master* pero también una rama llamada *master* .

El comando

```
git checkout master
```

Haría un checkout de la rama, pero el comando:

```
git checkout -- master
```

chequearía el fichero *master*.

El comando

```
git checkout HEAD -- .
```

forzaría a reemplazar TODOS LOS ARCHIVOS (de ahí el "—") lo que teníamos en el último commit a nuestro *working directory* (que es el ".").

Si tenemos algo en staging y queremos eliminarlo de staging, entonces usamos:

```
git reset HEAD fichero
```

Avanzado

Git checkout, no toca el área de preparación o stagging rea. Para eso tenemos un git reset.

Tarea

Usar *git checkout* para limpiar o deshacer los cambios del *working directory*.

Pasos:

- Creamos un directorio de trabajo, entramos en él y lo inicializamos.
- Creamos el README.md, le agregamos contenido, lo incluimos en staging y lo aprobamos.
- Creamos el fichero1, le agregamos contenido, lo incluimos en staging y lo aprobamos.
- Creamos el fichero2, le agregamos contenido, lo incluimos en staging y lo aprobamos.
- Creamos el fichero3, le agregamos contenido, lo incluimos en staging y lo aprobamos.
- Hacemos un borrado de "fichero*" y nos aseguramos de que el borrado esté en* staging*.
- Lanzamos "ls -l"
- Luego reemplazamos todo el contenido del directorio con lo que existiese en el HEAD de la rama master.

Paso 2 - Git Reset

Si estamos en la mitad de un *commit* y tenemos agregados ficheros a la *staging area* pero cambiamos de idea, entonces necesitaremos usar el comando *git reset* . El comando *git reset*eliminará a los ficheros de la *staging area* y los dejará en el *working directory* en estado *untracked* (no registrados).

Si deseamos sacar todos los ficheros de la *staging area*, debemos usar el carácter "." para indicarlo. En otro caso, se admite una lista de ficheros separados por espacios.

Esto es muy útil cuando intentamos mantener nuestros *commits* pequeños y concentrados de tal forma que podamos sacarlos de *staging* si hemos añadido demasiados de una tacada.

Tarea

Creamos un escenario de nuestra elección. Hacemos un par de commits.

Por último creamos un fichero más. Lo agregamos a staging.

Lanzamos *git reset* para dejarlo en estado *untracked*.

Paso 3 - Git Reset Hard

El comando *git reset —hard* combinará tanto el efecto de un *git checkout* como el de un *git reset* en un único comando. El resultado es que se eliminarán los ficheros de la *staging area* y del *working directory* de tal forma que volvemos a los mismos contenidos que estaban presentes en el último commit. Es un comando al que hay que tenerle respeto, porque podemos perder información si no estamos muy seguros de lo que estamos haciendo.

Avanzado

Usar HEAD limpiará el estado hacia lo que había en el último *commit*; si usamos git reset —hardnos posibilita volver al estadod e cualquier *commit*. Recordemos que HEAD es un alias para el has del último *commit* de la rama.

Tarea

Pasos:

- Creamos un directorio de trabajo, entramos en él y lo inicializamos.
- Creamos el README.md, le agregamos contenido, lo incluimos en staging y lo aprobamos.
- Creamos el fichero1, le agregamos contenido, lo incluimos en staging y lo aprobamos.
- Creamos el fichero2, le agregamos contenido, lo incluimos en staging y lo aprobamos.
- Creamos el fichero3, le agregamos contenido, lo incluimos en staging y lo aprobamos.
- Creamos fichero4 y fichero5, los incluimos en staging pero NO APROBAMOS.
- Lanzamos “ls -l” y “git status”.
- Hacemos un reset hard hacia HEAD-3 y lanzamos “ls -l” y “git status”.
- Debemos estar en un estado sin staging y con la misma “foto” que en HEAD-3.

Paso 4 - Git Revert

Si ya has aprobado fichero pero te has dado cuenta de que has cometido un error, entonces el comando *git revert* te permite deshacer los cambios que se hayan producido en ese *commit*concreto. El comando creará un nuevo *commit* que tiene la afección invertida del *commit* al que hemos revertido (de modo que si volvemos a él, nos quedamos como estábamos).

Si aún no hemos subido (*pushed*) los cambios, entonces el comando git reset HEAD~1 tiene la misma afección y borrará el último *commit* que ha sido creado por el *revert*.

Tarea

Usar *git revert* para revertir los cambios del último *commit.*

- Para ello:
- Creamos un directorio de trabajo, entramos en él y lo inicializamos.
- Creamos el README.md, le agregamos contenido, lo incluimos en staging y lo aprobamos.
- Creamos el fichero1, le agregamos contenido, lo incluimos en staging y lo aprobamos.
- Creamos el fichero2, le agregamos contenido, lo incluimos en staging y lo aprobamos.
- Creamos el fichero3, le agregamos contenido, lo incluimos en staging y lo aprobamos.
- Deshacemos el initial commit y comprobamos si ha desaparecido algún fichero.

Avanzado

La motivación subyacente de crear nuevos commits es que reescribir la historia en GIT es algo desaconsejado. Si has subido (pushed) tus commits, entonces deberías crear nuevos commits para deshacer los cambios provocados, ya que otros usuarios podrían a ver hecho commits mientras tanto.

Paso 5 - Git Revert

Para hacer *revert* de múltiples *commits* de una tacada usamos el carácter ~ para significar "menos". Por ejemplo, HEAD~2, son dos *commits* desde HEAD. Esto puede combinarse con los caracteres "…" que indican el rango entre dos *commits*.

Tarea

Usamos el comando *git revert HEAD…HEAD~2* para revertir los *commits* hechos entre HEAD y HEAD~2.

Para ello:

- Creamos un directorio de trabajo, entramos en él y lo inicializamos.
- Creamos el README.md, le agregamos contenido, lo incluimos en staging y lo aprobamos.
- Creamos el fichero1, le agregamos contenido, lo incluimos en staging y lo aprobamos.
- Creamos el fichero2, le agregamos contenido, lo incluimos en staging y lo aprobamos.
- Creamos el fichero3, le agregamos contenido, lo incluimos en staging y lo aprobamos.
- Ahora, usamos el comando *git revert* tal y como se ha propuesto.

Avanzado

Si en un revert, no tenemos ganas de pasar uno a uno por los cambios del rango seleccionado, se puede hacer así:

```
git revert HEAD...HEAD~2 --no-edit
```

SIGUIENTE

En esta lección aprenderemos cómo podemos compartir los cambios en nuestro repositorio con otras personas, y combinarlos en el nuestro.

Simularemos un trabajo con otros desarrolladores y provocaremos un conflicto. Descubriremos qué opciones tenemos para resolverlo.

Paso 1 - Git Merge

El comando *git fetch* descarga los cambios en una rama separada que puede ser comprobada (*checked out*) y fusionada (*merge*). Durante un *merge* git intentará automáticamente combinar los *commits*.

Cuando no existen conflictos, el *merge* será tratado rápidamente y no tendremos que hacer nada. Si sucede un conflicto, entonces recibiremos un error y el repositorio entrará en estado *merge*.

Tarea

Creamos un directorio que inicializamos.

Creamos un archivo README.md y le agregamos contenido. Este fichero provocará una colisión.

Lo agregamos a staging y hacemos *commit*.

Nos vincularemos al proyecto BAT accesible en **https://github.com/sharkdp/bat**

Trataremos de lanzar un *merge* y resolver el conflicto.

Avanzado

El comando *git pull* es una combinación de *fetch* y de *merge*.

En lugar de un fetch+merge, se puede usar:

```
git pull origin master –allow-unrelated-histories
```

En esta lección aprenderemos cómo podemos crear ramas en nuestro repositorio. Una rama (*branch*) permite trabajar, de forma efectiva, en un *working directory* completamente nuevo. El resultado es que un único repositorio Git puede tener múltiples versiones diferentes del código base, entre las cuales podemos intercambiarnos sin movernos de directorio.

La rama por defecto en Git se llama *master* . Las ramas adicionales permiten llevar a cabo las mismas operaciones y comandos que podrías efectuar en la rama *master* , como cambios de tipo *committing* , *merging* y *pushing*.

Como ramas adicionales, funcionan de la misma manera que *master* y son ideales para prototipado y experimentos o pruebas que permiten ser fusionadas a la rama *master* si se decide así.

Cuando conmutamos o nos cambiamos de rama, Git cambia los contenidos del *working directory*. Esto es muy llamativo y sorprendente las primeras veces que lo usamos. Usando esta funcionalidad, no necesitamos cambiar configuraciones o parámetros para reflejar que estamos en ramas o ubicaciones diferentes.

Paso 1 - Git Branch

Las ramas se crean basándonos en otra rama, generalmente *master*. El comando:

```
git branch <new_branch> <starting_branch>
```

toma una rama existente y crea una rama separada para trabajar en ella. Justo en el punto de lanzar el comando, las dos ramas son idénticas. Para cambiar de una rama a otra, usamos el comando:

```
git branch <new_branch>
```

Tarea

Creamos un nuevo repositorio y lo inicializamos.

Vamos a gitlab, creamos un nuevo proyecto de tipo privado, y lo inicializamos con un README.md automático.

Agregamos el origin de nuestro proyecto gitlab y lo descargamos a nuestro master.

Creamos una nueva rama réplica y la llamamos “new_branch”.

Nos cambiamos a ella.

Avanzado

El comando *git checkout -b* creará y hará *checkout* de la nueva rama creada; es decir, es lo mismo que un *git branch + git checkout*

Paso 2 – Listar las ramas

Para listar todas las ramas usamos el comando *git branch*.

El argumento adicional *-a* incluirá también las ramas remotas, y el parámetro -v incluirá el mensaje *commit* de HEAD de la rama. Recomendable usar ambos siempre.

Tarea

Usando la carpeta de proyecto del paso anterior, listar todas las ramas con su último mensaje de *commit* lanzando *git branch*

Paso 3 – Hacer fusión (merge) a master

Supongamos que se ha producido un *commit* a una nueva rama. Para fusionar (*merge*) dentro de la rama *master*, deberíamos primero hacer *chekout* a la rama objetivo (posicionarnos sobre ella), en

este caso *master*, y estando en ella, lanzar un *git merge* para fusionar los cambios de la nueva rama sobre la rama *master*.

Ejemplo de llamada:

```
git merge rama_origen rama_destino_del_merge
```

Tarea

Usando la carpeta de proyecto de nuestro ejercicio anterior, hacemos checkout a new_branch.

Modifico el archivo README.md, lo llevo a staging y lo apruebo.

Ahora, me cambio a master y fusiono.

Compruebo los cambios. Los apruebo.

Paso 4 - Push Branches

Tal y como hemos visto en pasos anteriores, si queremos subir el contenido de una rama a una ubicación remota, tenemos que usar el comando:

```
git push <remote_name> <branch_name>
```

Ejemplo:

```
$ git push origin master warning: redirecting to https://gitlab.com/juancarlos.rubio/ej06-branches.git/ Enumerating objects: 5, done. Counting objects: 100% (5/5), done. Delta compression using up to 2 threads. Compressing objects: 100% (2/2), done. Writing objects: 100% (3/3), 340 bytes | 340.00 KiB/s, done. Total 3 (delta 0), reused 0 (delta 0) To https://gitlab.com/juancarlos.rubio/ej06-branches    14a0d49..31980bc master -> master
```

Tarea

Trabajar en una carpeta de proyecto y subir cambios a gitlab. Comprobar que se suben correctamente.

Paso 5 – Limpiar ramas

Limpiar ramas es importante para borrar ruido y confusión en un proyecto. Para borrar una rama se necesita usar el argumento *-d*. Por ejemplo:

```
git branch -d <branch_name>
```

Tarea

Ahora que hemos fusionado la rama en *master* en el paso anterior, ya no nos hace falta. Borrémosla para mantener el repositorio limpio y comprensible.

```
git remote add origin
https://edzamo13@bitbucket.org/edzamo13/practica.git
```

conexión remota

git fetch se ve lo que contiene el repositorio

git branch lista los brach

crear una brach

git branch nombre_rama

conectarse con la rama

git checkout nombre_rama

mezclar las ramas

git merge nombre_rama master

Para eliminar una rama

Git branch –s nombre_rama

Para crear y seleccionar una ram utilizamos

Git branch –b new_nombreRam

Los errores de software o bugs han constituido un problema desde que existe el software. Puesto que Git guarda todos los cambios en el repositorio, se convierte en una gran fuente de información y en una herramienta de diagnóstico cuando tratamos de identificar cómo llegaron a introducirse los problemas de código.

En esta lección exploraremos diferentes formas para encontrar en cuál commit se introdujo un determinado problema.

Paso 1 - Git Diff entre dos Commits

El comando *git diff* es el modo más simple de comparar qué ha cambiado entre commits. Mostrará las diferencias entre dos commits.

Ejemplo:

Podemos visualizar las diferencias entre dos *commits*visualizando los *hash-id's*

```
$ git diff HEAD~2 HEAD diff --git a/README.md b/README.md index 77927c0..c8831b2 100644 --- a/README.md +++ b/README.md @@ -1,2 +1,3 @@  # ej06-branches  +## Y agrego este subtitulo al fichero README.md diff --git a/codigo.js b/codigo.js new file mode 100644 index 0000000..cac68b2 --- /dev/null +++ b/codigo.js @@ -0,0 +1 @@ +mas codigo
```

Posibilidades comunes:
git diff muestra las diferencias entre el *working directory*y el *commit*. En este caso, el commit es HEAD, así que mostraría los cambios que hemos hecho desde nuestro último *commit*. Incluyendo los ficheros *untracked* o los ficheros *unstaged*, es decir, nuestro working directory.

git diff —cached muestra las diferencias solo entre nuestro *staged changes* y el estado almacenado en él. Es decir, que si no he hecho "git add" a ciertos ficheros que haya modificado, no se analizan las diferencias entre ellos y el *commit*.

Paso 2 - Git Log

El comando *git log* permite ver los mensajes de *commit*pero por defecto no muestra una salida de lo que en realidad se modificó. Afortunadamente, el comando es extremadamente flexible y permite opciones adicionales que proporcionan visualizaciones útiles de lo sucedido en el histórico del repositorio.

Ejemplos:

Para ver una introducción o entradilla de información de los *commits* en una vista reducida, podemos usar el comando git log —oneline

La salida de información del *commit* con las diferencias que se hayan producido se puede lograr agregando el parámetro *-p* , lanzándolo así: git log -p

Esto mostraría todo el repositorio al completo… Demasiada información. Podemos delimitar un poco usando otro parámetro: *-n* . Con él, especificamos un límite de *commits* a mostrar desde HEAD.

Por ejemplo; el comando:

```
git log -p -n 2
```

mostraría HEAD y HEAD~1.

Si conocemos el periodo de tiempo o fechas que queremos consultar (útil para localizar versiones con garantías de estar libres de fallos, por ejemplo) podemos usar:

```
*--since="2 weeks ago"* y --until="1 day ago".
```

(admite "*1 hour ago*" y otras variantes en inglés).

Y como en otros comandos Git, podemos obtener un rango de salida de *commits* usando intervalos como por ejemplo HEAD…HEAD~1

Podemos localizar patrones en el asunto del *commit* con el parámetro *grep :*

Ejemplo:

```
git log --grep="Initial"
```

mostraría la salida de commits que incluyesen la cadena "Initial" en su mensaje. También podemos usar un pipeline y redirigirlo al comando unix *grep* normalmente. Esto puede ser muy útil cuando tratamos de localizar números de registro de bug (*bug-tracking numbers*).

Avanzado

Tal y como hemos visto, el log puede ser ruidoso. Probemos las posibilidades del comando *git log*agregándole el parámetro *-m* y juguemos con las consultas para obtener resultados deseados.

Paso 3 - Git Blame

Disponer de una cultura de "culpa" (*blame*) no es deseable, aunque puede ser útil para conocer quién trabajó en ciertas secciones del fichero para ayudar a realizar mejoras en el futuro.

En estas circunstancias es donde *git blame* puede ser de utilidad.

git blame muestra la revisión y el autor que modificó cada línea de un fichero.

Ejemplo:

Lanzar *git blame* sobre un fichero mostrará quién tocó cada línea.

```
git blame list.html
```

Y si conocemos las líneas que pueden estar afectadas, podemos filtrar el rango de líneas de salida:

```
git blame -L 6,8 list.html industriasi@alvarez MINGW32 ~/ownCloud/temporal/git/code/ej06-branches (maste $ git blame README.md ^14a0d49 (Juan Carlos Rubio 2018-09-06 16:38:29 +0000 1) # ej06-branches ^14a0d49 (Juan Carlos Rubio 2018-09-06 16:38:29 +0000 2) 31980bcd (Juan Carlos Rubio 2018-09-06 19:00:02 +0200 3) ## Y agrego este subtitulo al fichero README.md
```

Tarea

Probemos la ejecución y las posibilidades de estas herramientas.

Creamos un nuevo proyecto. Inicializamos. Nos vinculamos a **https://github.com/sharkdp/bat**

Hacemos pull. Buscamos logar los últimos 15 cambios.

Lanzamos *blame* sobre el fichero *src/output.rs*

Git maneja tres cahas working directory staging commit

Git diff –- cached
Loque es esta en staged

Una de las ventajas de pequeños *commits* es que podemos ser quisquillosos y detallistas acerca que cuáles de esos commits queremos unir / *merge*.

Este problema afecta particularmente a las ramas de larga vida, que se han convertido en obsoletas con una rama que tiene demasiados conflictos para hacer *merge*. Esto sucede de forma frecuente en muchos proyectos activos de *open source*.

Cuando esto sucede, queremos ser capaces de picotear *commits* individuales y simplemente hacer *merge* de ellos en la rama principal. Veremos cómo hacerlo.

Paso 1 – Resolviendo un conflicto de Cherry Picking

De igual forma que hacer un *merging* puede resultar en un conflicto, también puede producirse en un "entresacado" o "picoteo" (*cherry-pick*). Los conflictos se solucionan de la misma forma que fusionando una rama bien reparando manualmente los ficheros o bien seleccionando *theirs* o seleccionando *ours* a través del comando *git checkout*.

Si nos damos cuenta de que hemos cometido un error, podemos parar y revertir lo que hemos hecho lanzando:

```
git cherry-pick --abort
```

Imaginemos que lanzamos un *cherry-pick* ejecutando por ejemplo:

```
git cherry-pick new_branch~1
```

Si este comando resultase en un conflicto de *merge*, se puede resolver usando el comando git checkout y seleccionando el *commit* elegido cuidadosamente (*picked*).

Paso 2 - Continuando un Cherry Picking después de un conflicto

Una vez que los conflictos se han resuelto, se puede continuar con el *cherry-pick* usando el comando

```
git cherry-pick --continue
```

De forma similar a *merge*, resolver un *cherry-pick* puede resultar en un *commit*.

Tarea

En este caso, trabajaremos conjuntamente el ejercicio para comprobar las posibilidades del comando *cherry-pick* .

En esta lección, vamos a tratar de tomar parte del árbol del proyecto BAT y en lugar de hacer un pull completo (como un clone), nos vamos a traer dos porciones de modificaciones y las vamos a agregar con merge desmenuzado con cherry-pick.

Git blame ayuda para sacar los actores

Una de las ventajas de pequeños *commits* es que podemos ser quisquillosos y detallistas acerca que cuáles de esos commits queremos unir / *merge*.

Este problema afecta particularmente a las ramas de larga vida, que se han convertido en obsoletas con una rama que tiene demasiados conflictos para hacer *merge*. Esto sucede de forma frecuente en muchos proyectos activos de *open source*.

Cuando esto sucede, queremos ser capaces de picotear *commits* individuales y simplemente hacer *merge* de ellos en la rama principal. Veremos cómo hacerlo.

Paso 1 – Resolviendo un conflicto de Cherry Picking

De igual forma que hacer un *merging* puede resultar en un conflicto, también puede producirse en un "entresacado" o "picoteo" (*cherry-pick*). Los conflictos se solucionan de la misma forma que fusionando una rama bien reparando manualmente los ficheros o bien seleccionando *theirs* o seleccionando *ours* a través del comando *git checkout*.

Si nos damos cuenta de que hemos cometido un error, podemos parar y revertir lo que hemos hecho lanzando:

```
git cherry-pick --abort
```

Imaginemos que lanzamos un *cherry-pick* ejecutando por ejemplo:

```
git cherry-pick new_branch~1
```

Si este comando resultase en un conflicto de *merge*, se puede resolver usando el comando git checkout y seleccionando el *commit* elegido cuidadosamente (*picked*).

Paso 2 - Continuando un Cherry Picking después de un conflicto

Una vez que los conflictos se han resuelto, se puede continuar con el *cherry-pick* usando el comando

```
git cherry-pick --continue
```

De forma similar a *merge*, resolver un *cherry-pick* puede resultar en un *commit*.

Tarea

En este caso, trabajaremos conjuntamente el ejercicio para comprobar las posibilidades del comando *cherry-pick* .

En esta lección, vamos a tratar de tomar parte del árbol del proyecto BAT y en lugar de hacer un pull completo (como un clone), nos vamos a traer dos porciones de modificaciones y las vamos a agregar con merge desmenuzado con cherry-pick.

Un aspecto importante de Git es cómo mantener limpio el repositorio y su historia. Una historia limpia es más cómoda para trabajar y para comprender qué ha sucedido.

En esta lección cubriremos cómo reescribir la historia de Git usando el comando *rebase* para reestructurar los *commits* y asegurarnos de que son comprensibles antes de subir / *push* los cambios.

Recomendación: Solo deberíamos hacer *rebase* de *commits* que no hayamos compartido con otras personas vía *push*. El proceso de *rebasing* de *commits*provoca que los hash_id *commit-ids* cambien, lo cual puede resultar en pérdida de *commits* futuros.

Paso 1 - Enmendando (*amend) mensajes* de *commit*

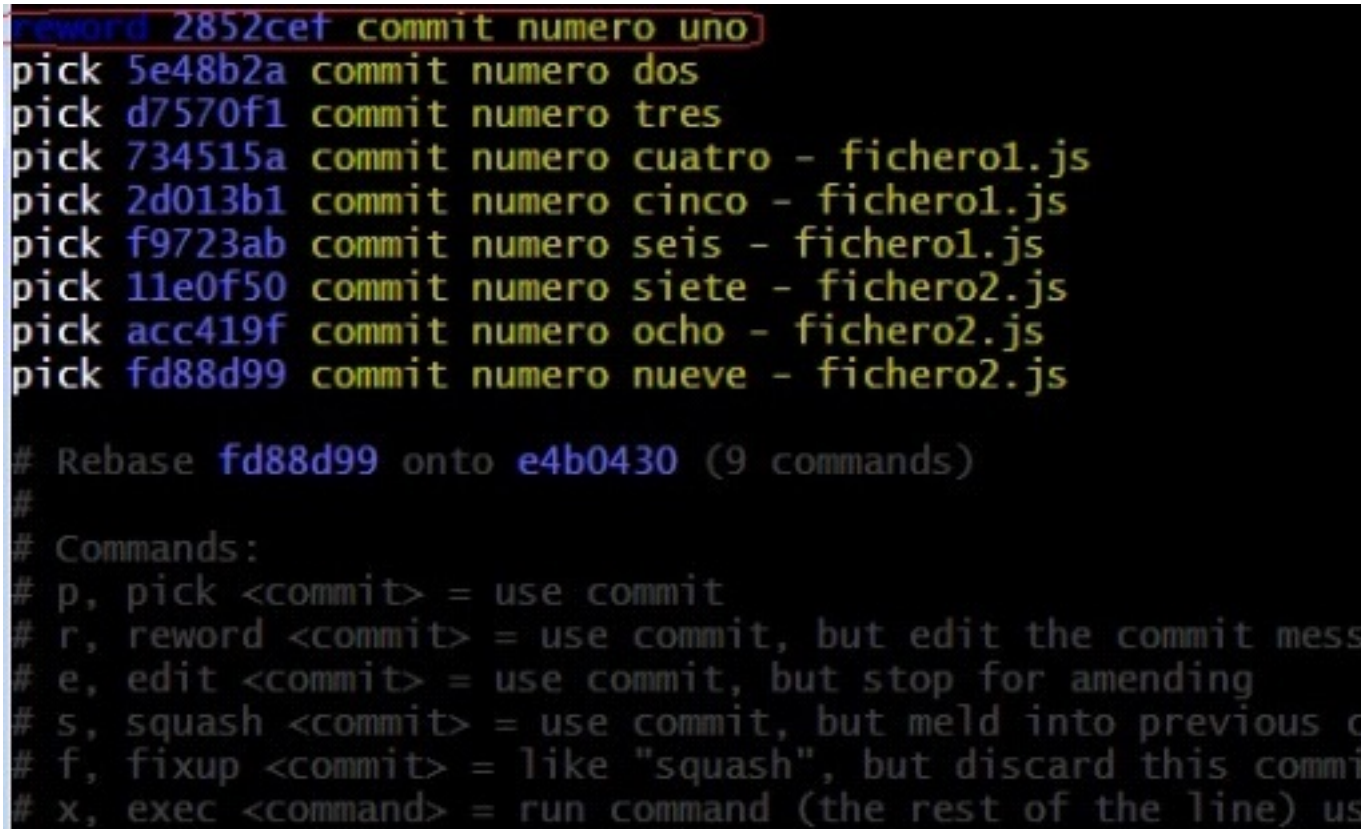
Para reescribir el histórico de los repositorios, usamos el comando:

```
git rebase -interactive
```

Al poner un *rebase* en modo interactivo, tenemos más control sobre los cambios que queremos hacer. Después de lanzar el modo interactivo, disponemos de 6 comandos para llevar a cabo en cada *commit* del repositorio. Usando el editor, podemos definir qué acciones queremos llevar a cabo en cada *commit* .

En este ejercicio, vamos a cambiar el comentario de un *commit.* Tras ponerlo en estado *rebase* , tendremos que cambiar la palabra *pick* por la palabra *reword para* el identificador de commit para el que queramos cambiar la descripción.

En esta lección, solo vamos a cambiar la descripción. Ejemplo:



Tarea

Para este ejercicio, vamos a crear un nuevo repositorio.

Crearemos 9 ficheros, y tras la creación de cada uno, haremos un commit y agregaremos un texto inequívoco que nos sirva como descriptor.

Para empezar el *rebase* lanzamos:

```
git rebase --interactive --root
```

Entrando en “Interactive Mode”

Una vez que estemos en la pantalla de la ilustración anterior, cambiaremos la palabra “*pick*“ del primer *commit* de la lista por la palabra “*reword*“.

Grabamos

Cambiando el mensaje

Ahora cambiamos el mensaje por lo que queramos. Grabamos y salimos.

Lanzamos el comando:

```
git log --oneline
```

para comprobar que se ha actualizado el mensaje.

Avanzado

El argumento —*root* permite hacer un *rebase* de TODOS los commits del repositorio, incluyendo *el primer commit*.

Cuando lo que queremos es únicamente cambiar el mensaje del último commit realizado, la alternativa más rápida es lanzar el comando:

```
git commit --amend
```

y efectuar los cambios.

Paso 2 - Aprobaciones condensadas (Squash Commits)

Hemos realizado una serie de commits en nuestro entorno. En el momento en el que se hicieron, quizás esos cambios tenían sentido como operaciones independientes, pero ahora necesitamos condensarlos en un único commit.

Usando *rebase* podemos condensar (*squash*) los *commits* todos juntos.

Lanzando git rebase —interactive HEAD~4, tendremos 4 commits disponibles. Para condensarlo, necesitamos una base sobre la que todo será condensado. Por ello, en el ejercicio siguiente, dejamos el primer *commit* como *pick* y marcamos los demás como *squash* . Después de guardar, podremos cambiar el mensaje de commit por algo más clarificador.

Tarea

Cuando entramos en modo *interactive rebase* podemos especificar lo que queremos modificar en los 4 previos commits. Lanzamos:

```
git rebase --interactive HEAD~4
```

En el ejercicio anterior usamos *reword* . Aquí vamos a usar *squash*. Queremos condensar 4 commits en uno; si etiquetáramos todos los commits como squash, obtendríamos un error: "*Cannot 'squash' without a previous commit*" ya que se supone que debe existir un commit base sobre el que condensar los demás.

Para condensar los commits tenemos que dejar el primer commit (el más antiguo temporalmente) como nuestra base, y etiquetar el resto con *squash*.

Mensaje de commit

Al salvar y salir se mostrará una ventana con una combinación de los cuatro mensajes de commit en el *rebase*.

Después de guardar el mensaje de commit, el histórico se modificará. Podemos comprobar que el hasid no coincide lanzando:

```
git log --oneline
```

Reordenar commits

Reordenar commits puede ayudar a construir una foto más clara del orden lógico cómo se ha completado lo que hemos trabajado.

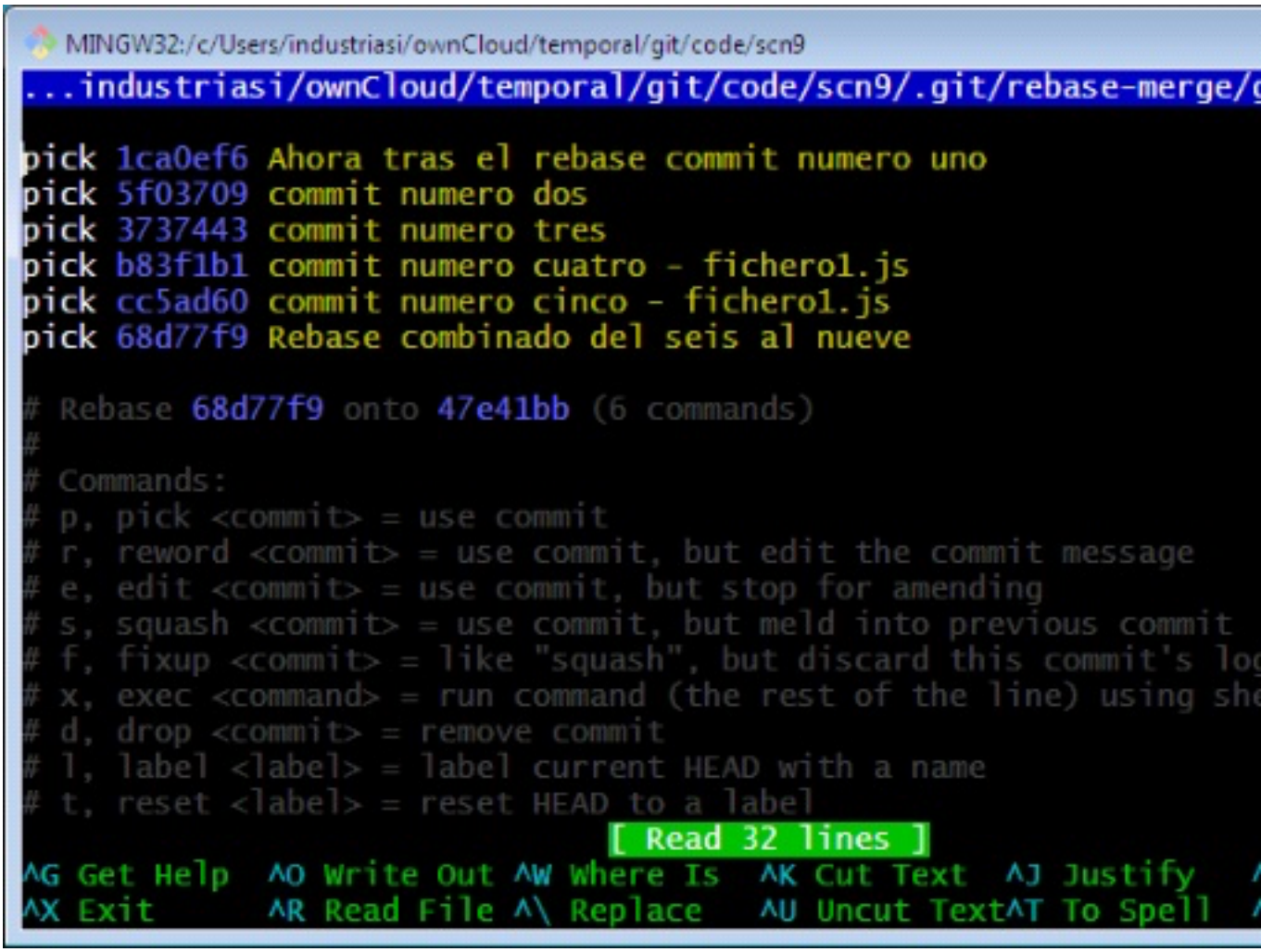
Tarea

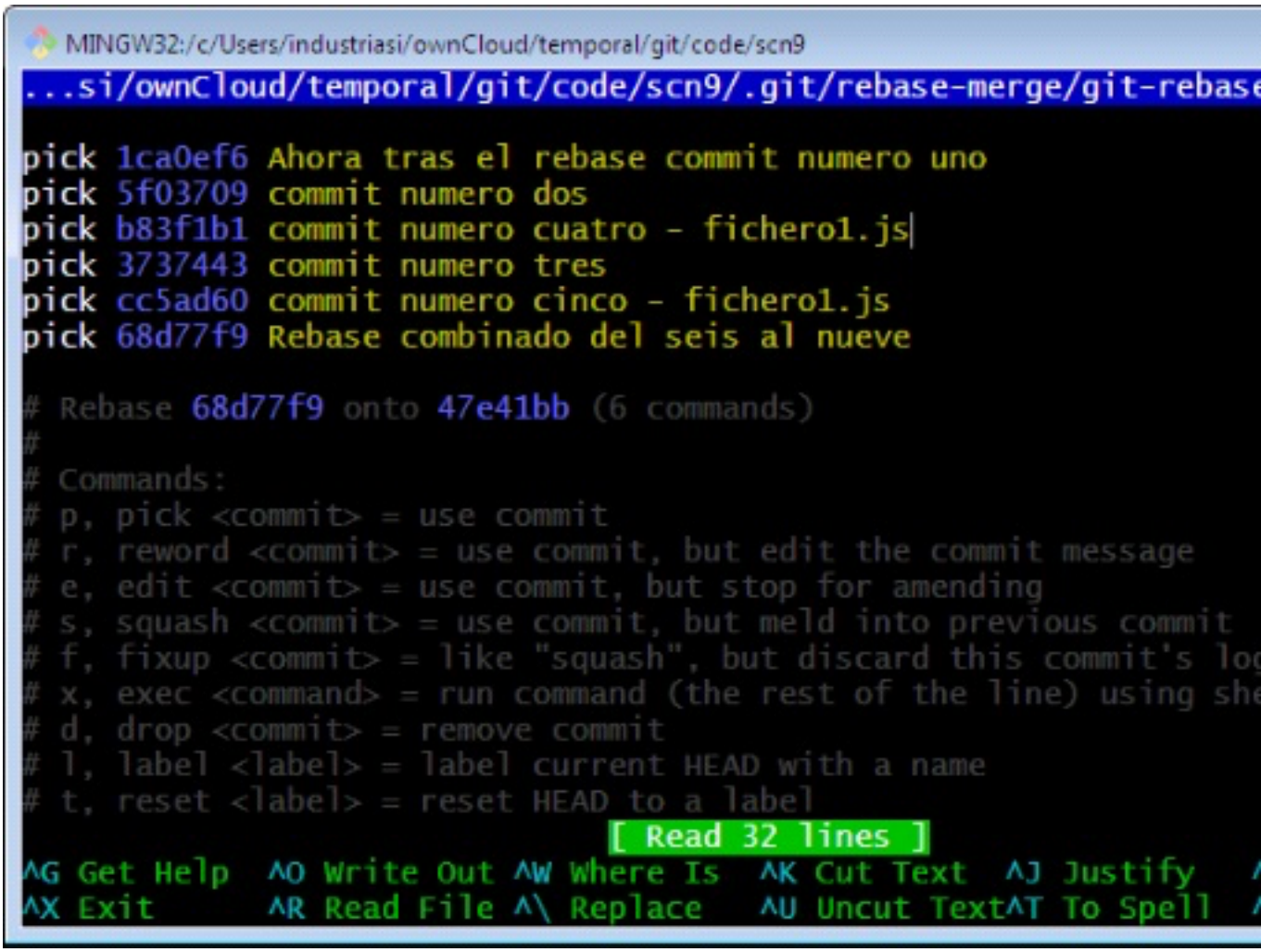
Queremos reordenar nuestros últimos dos commits. Usando HEAD~2 podremos hacerlo:

```
git rebase --interactive HEAD~2
```

Usando el editor, simplemente reordenaremos las líneas, guardaremos y saldremos y los commits reflejarán el nuevo orden.

Aquí tenemos una ilustración:


MINGW32:/c/Users/industriasi/ownCloud/temporal/git/code/scn9
...industriasi/ownCloud/temporal/git/code/scn9/.git/rebase-merge/
pick 1ca0ef6 Ahora tras el rebase commit numero uno
pick 5f03709 commit numero dos
pick 3737443 commit numero tres
pick b83f1b1 commit numero cuatro - fichero1.js
pick cc5ad60 commit numero cinco - fichero1.js
pick 68d77f9 Rebase combinado del seis al nueve
Rebase 68d77f9 onto 47e41bb (6 commands)
#
Commands:
p, pick <commit> = use commit
r, reword <commit> = use commit, but edit the commit message
e, edit <commit> = use commit, but stop for amending
s, squash <commit> = use commit, but meld into previous commit
f, fixup <commit> = like "squash", but discard this commit's log
x, exec <command> = run command (the rest of the line) using she
d, drop <commit> = remove commit
l, label <label> = label current HEAD with a name
t, reset <label> = reset HEAD to a label
[Read 32 lines]
^G Get Help ^O Write Out ^W Where Is ^K Cut Text ^J Justify
^X Exit ^R Read File ^\ Replace ^U Uncut Text^T To Spell




```
$ git log --oneline 3254464 (HEAD -> master) Rebase combinado del seis al nueve c58f563 commit numero cinco - fichero1.js 25cd399 commit numero tres b7f8abc commit numero cuatro - fichero1.js 5f03709 commit numero dos 1ca0ef6 Ahora tras el rebase commit numero uno
```

Paso 3 - Separar los commits (split)

Del mismo modo que cuando se condensan los commits, a veces es útil separar o desmenuzar los commits con el objeto de mantener la atención y posibilitar un *cherry-pick* o un *revert* más sencillo.

Separar los commits es un proceso de dos fases. Primero necesitamos definir qué commit queremos dividir o separar, y segundo, necesitamos definir cómo queremos que se muestren los nuevos *commits*.

Definiendo el *commit* que separar (*split*)

Aquí queremos separar el commit anterior. Lanzamos:

```
git rebase --interactive HEAD~1
```

Como en *rebase's* previos, necesitamos cambiar la tarea al término *edit*

Ahora estamos en un estado de edición interactiva del histórico. Git grabará todos los cambios y el resultado final será aplicado al repositorio.

Separando commits

Después de definir que queremos *edit*ar el commit, nos encontramos en un estado que nos permite cambiar el histórico (estamos en modo interactivo y finalizaremos la "grabación" de acciones con un *—continue)*

1. Como queremos separar un commmit existente, primero necesitamos borrarlo lanzando el comando: git reset HEAD~1.
2. El commit ha sido borrado pero aún existe. Ahora podemos llevar a cabo los commits como deseamos, es decir, como dos acciones separadas.

Ejecutamos los siguientes comandos:

```
git add file3.txt git commit -m "File 3" git add file4.txt git commit -m "File 4"
```

Guardando "la grabación" del *rebase*

Una vez que estamos felices con el estado del repositorio, y conformes, podemos decirle a git que continúe el *rebase* y que actualice el repositorio lanzando el comando:

```
git rebase --continue
```

Podemos visualizar la salida y los dos nuevos commits con el comando:

```
git log --oneline
```

La capacidad de reescribir el histórico es útil para mantener el histórico del repositorio limpio y preciso. Esto ayudará en el futuro para indicar las razones para un cambio o para depurar problemas de código.

Recomendación

Solo deberíamos hacer un *rebase* en commits que no hayamos compartido con otras personas vía push. El *rebasing* causa que los identificadores de commit cambien, lo que puede desembocar en pérdida de commits futuros.

Como muchos VCS, Git tiene la posibilidad de etiquetar puntos específicos del historial como importantes. Esta funcionalidad se usa típicamente para marcar versiones de lanzamiento (v1.0, por ejemplo). En esta sección, aprenderás como listar las etiquetas disponibles, como crear nuevas etiquetas cuales son los distintos tipos de etiquetas. En esta lección aprenderemos cómo trabajar con ellas.

Paso 4 - Listar Tus Etiquetas

Listar las etiquetas disponibles en Git es sencillo. Simplemente escribe git tag:

```
$ git tag v0.1 v1.3
```

Este comando lista las etiquetas en orden alfabético; el orden en el que aparecen no tiene mayor importancia.

También puedes buscar etiquetas con un patrón particular. El repositorio del código fuente de Git, por ejemplo, contiene más de 500 etiquetas. Si solo te interesa ver la serie 1.8.5, puedes ejecutar:

```
$ git tag -l 'v1.8.5*' v1.8.5 v1.8.5-rc0 v1.8.5-rc1 v1.8.5-rc2 v1.8.5-rc3 v1.8.5.1 v1.8.5.2 v1.8.5.3 v1.8.5.4 v1.8.5.5
```

Paso 5 - Crear Etiquetas

Git utiliza dos tipos principales de etiquetas: ligeras y anotadas. Una etiqueta ligera es muy parecido a una rama que no cambia - simplemente es un puntero a un *commit* especifico.

Sin embargo, las etiquetas anotadas se guardan en la base de datos de Git como objetos enteros.

Tienen un *checksum*; contienen el nombre del etiquetador, correo electrónico y fecha; tienen un mensaje asociado; y pueden ser firmadas y verificadas con *GNU Privacy Guard* (GPG). Normalmente se recomienda que crees etiquetas anotadas, de manera que tengas toda esta información; pero si quieres una etiqueta temporal o por alguna razón no estas interesado en esa información, entonces puedes usar las etiquetas ligeras.

Paso 6 - Etiquetas Anotadas

Crear una etiqueta anotada en Git es sencillo. La forma más fácil de hacer es especificar la opción -a cuando ejecutas el comando tag:

```
$ git tag -a v1.4 -m 'my version 1.4' $ git tag v0.1 v1.3 v1.4
```

La opción -m especifica el mensaje de la etiqueta, el cual es guardado junto con ella. Si no especificas el mensaje de una etiqueta anotada, Git abrirá el editor de texto para que lo escribas.

Puedes ver la información de la etiqueta junto con el *commit* que está etiquetado al usar el comando git show:

```
$ git show v1.4 tag v1.4 Tagger: Ben Straub <ben@straub.cc> Date: Sat May 3 20:19:12 2014 -0700 my version 1.4 commit ca82a6dff817ec66f44342007202690a93763949 Author: Scott Chacon <schacon@gee-mail.com> Date: Mon Mar 17 21:52:11 2008 -0700 changed the version number
```

El comando muestra la información del etiquetador, la fecha en la que el *commit* fue etiquetado y el mensaje de la etiquetar, antes de mostrar la información del *commit*.

Paso 7 - Etiquetas Ligeras

La otra forma de etiquetar un *commit* es mediante una etiqueta ligera. Una etiqueta ligera no es más que el *checksum* de un *commit* guardado en un archivo - no incluye más información. Para crear una etiqueta ligera, no pases las opciones -a, -s ni -m:

```
$ git tag v1.4-lw $ git tag v0.1 v1.3 v1.4 v1.4-lw v1.5
```

Esta vez, si ejecutas git show sobre la etiqueta, no veras la información adicional. El comando solo mostrará el commit:

```
$ git show v1.4-lw commit ca82a6dff817ec66f44342007202690a93763949 Author: Scott Chacon <schacon@gee-mail.com> Date: Mon Mar 17 21:52:11 2008 -0700 changed the version number
```

Con el comando:

```
git checkout <hash-id>
```

Entramos en modo desconectado, pero tenemos una foto idéntica a la que teníamos en ese instante del tiempo.

En **subversion** hubiera sido con el comando:

```
svn up -r800
```

(suponiendo la versión 800)

Para volver al HEAD, escribimos:

```
git checkout master
```

o bien

```
git checkout -
```

(un único guion al final del comando).

Tarea:

Movernos al commit inicial y al HEAD y comprobar cómo los ficheros aparecen y desaparecen

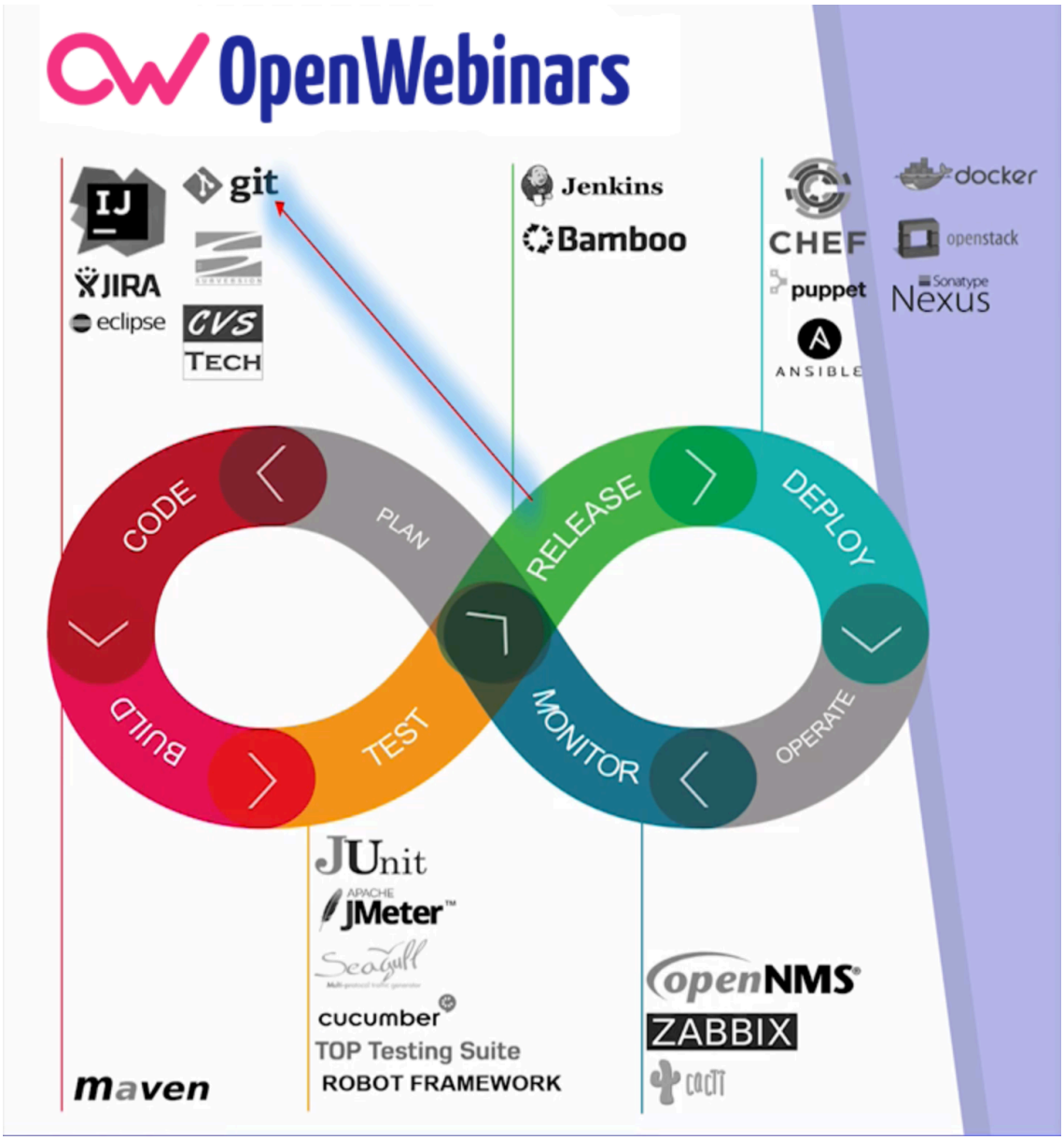
Nota:

git stash guarda el estado que tengamos en ese momento sin aprobar o en staged.

Se recupera con:

```
git stash apply
```

+


OpenWebinars
IJ
git
JIRA
eclipse
CVS
TECH
Jenkins
Bamboo
CHEF
docker
openstack
puppet
Sonatype
Nexus
ANSIBLE
CODE
PLAN
RELEASE
DEPLOY
BUILD
TEST
MONITOR
OPERATE
JUnit
APACHE
JMeter™
Seagull
cucumber
TOP Testing Suite
ROBOT FRAMEWORK
maven
openNMS
ZABBIX
cacti