Control de versiones: Git y Github.

Control de versiones

El control de versiones es el modelo de trabajo en el que se pueden guardar y gestionar únicamente los cambios que se realizan sobre un mismo elemento, permitiendo así viajar en la historia de las modificaciones de un archivo de manera precisa.

El sistema de control de versiones más utilizado y del que se desarrolla en este documento es Git, aunque existen otros sistemas como SVN.

Introducción a Git

Como se mencionó, Git es un sistema de control de versiones que guarda los cambios que se le hacen a los archivos en contenedores especiales llamados repositorios.

Una vez se instala y se inicializa en algún proyecto, en la carpeta del proyecto se añadirá una carpeta .git:

.git

La carpeta .git será la encargada de almacenar todos los cambios que sucedan en los archivos del proyecto.

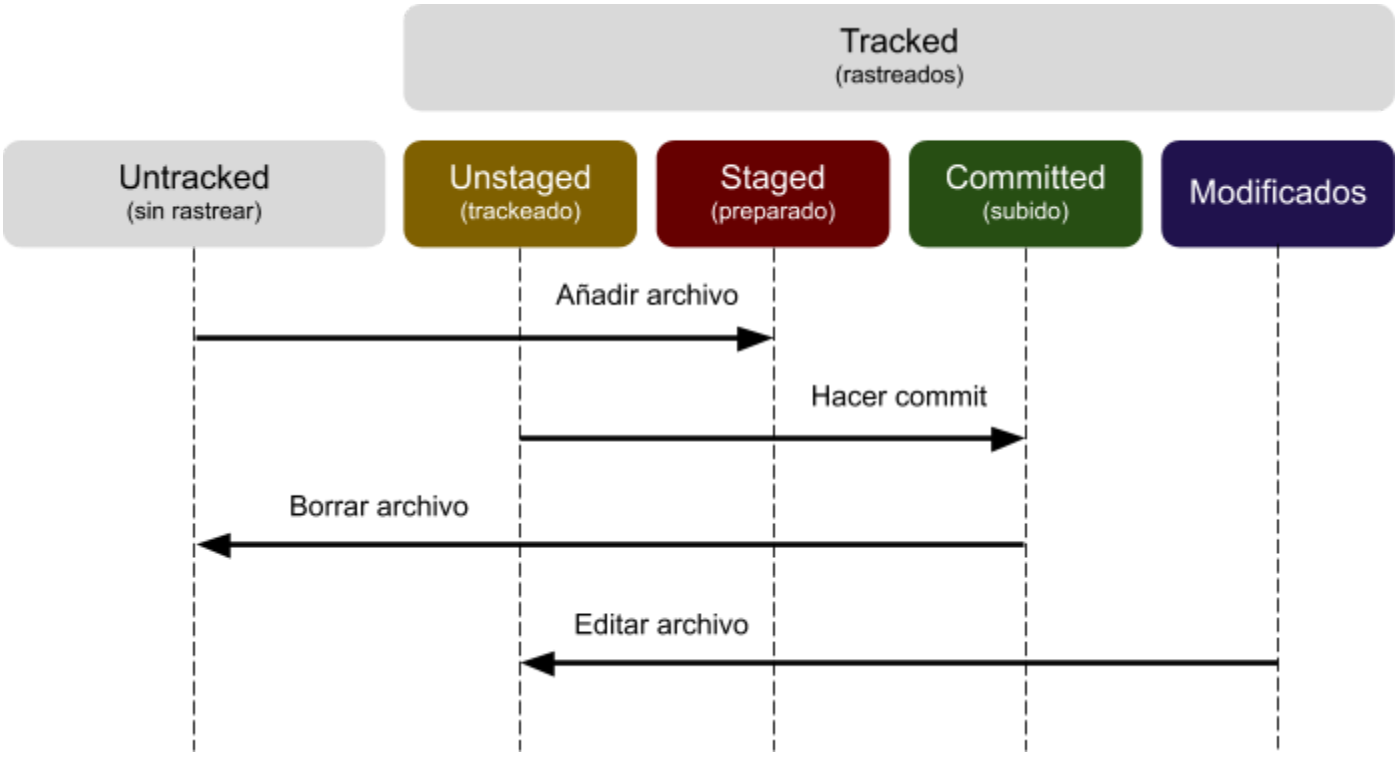
Estados de los archivos en Git.

Cuando se quiere empezar a trabajar en Git, deben tenerse claros los diferentes estados a los que se encuentran sujetos los archivos. Así, nos encontramos con archivos: Untracked (no rastreados), staged (del inglés: listos para la escenificación; es decir, preparados) y committed (ejecutados y guardados).

En síntesis, los archivos:

- **Untracked** (no rastreados) es el estado de aquellos archivos que aún no se encuentran en el repositorio. Es decir archivos que existen localmente, pero que no hacen parte del repositorio de su proyecto. Por ello los cambios en este archivo no se registrarán.
- **Tracked** (rastreados)**:**
    - **Unstaged:** Son archivos sobre los cuales Git ya está rastreando los cambios. Sin embargo, aún no se confirma que serán archivos que subirán al repositorio.
    - **Staged:** Son archivos que se encuentran rastreados por Git y que están preparados para ser subidos de forma definitiva al repositorio.
    - **Committed:** son archivos que se encuentran ya guardados en la historia del repositorio de git.

Debe tenerse en cuenta que una vez un archivo está rastreado, todos los estados a los que está sujeto (unstaged, staged, committed) se convierten en un ciclo. Es decir, cada vez que un archivo que ya está committed se modifica, vuelve al estado de unstaged. Una forma gráfico de verlo es la siguiente:



Trabajo en Git

Git es un programa que esencialmente se manipula desde la línea de comandos. En ese sentido, su sintaxis básica es como la de cualquier otro comando:

$git diff --cached 648b70dd1 d5583e578af	
comando	Variables de entorno que ejecutan binarios.
flag	Modificador de los comandos.
argumento	Diferentes entradas que se les pueden dar al programa.

Configuración

Una vez instalamos Git existen una serie de configuraciones iniciales que debemos realizar. Estas son esencialmente: añadir nuestro correo electrónico y nombre. De esta manera Git podrá asignarle un autor a todos los cambios que se realicen. Para ello, disponemos del siguiente comando:

git config

$git config	
list	Nos devuelve todas las configuraciones de git.

```
core.excludesfile=~/.gitignore
core.legacyheaders=false
core.quotepath=false
mergetool.keepbackup=true
push.default=simple
color.ui=auto
color.interactive=auto
repack.usedeltabaseoffset=true
alias.s=status
alias.a=!git add . && git status
alias.au=!git add -u . && git status
alias.aa=!git add . && git add -u . && git status
alias.c=commit
alias.cm=commit -m
alias.ca=commit --amend
alias.ac=!git add . && git commit
alias.acm=!git add . && git commit -m
alias.l=log --graph --all --pretty=format:'%C(yellow)%h%C(cyan)%d%Creset %s %C(white)- %an,
 %ar%Creset'
alias.ll=log --stat --abbrev-commit
alias.lg=log --color --graph --pretty=format:'%C(bold white)%h%Creset -%C(bold green)%d%Cre
set %s %C(bold green)(%cr)%Creset %C(bold blue)<%an>%Creset' --abbrev-commit --date=relativ
e
alias.llg=log --color --graph --pretty=format:'%C(bold white)%H %d%Creset%n%s%n%+b%C(bold b
lue)%an <%ae>%Creset %C(bold green)%cr (%ci)' --abbrev-commit
alias.d=diff
alias.master=checkout master
alias.spull=svn rebase
alias.spush=svn dcommit
alias.alias=!git config --list | grep 'alias\.' | sed 's/alias\.\([^=]*\)=\(.*\)/\1\     =>
```

$git config	
Modificador	Por ejemplo, podemos usar --global para realizar modificaciones del entorno global de git.

configuracion	Indica la configuración que se quiere realizar (Puede ser cualquiera de las enseñadas en $git config --list)
argumento	El valor que se le quiere dar a la configuración (Haciendo uso de comillas)

Las configuraciones iniciales que necesitamos para empezar a usar Git son:

$git config --global user.email “correo@mail.com”

$git config --global user.name “Nombre”

Pero podemos cambiar todas las configuraciones que trae Git. Por ejemplo, core.editor para establecer el editor por defecto en PC.

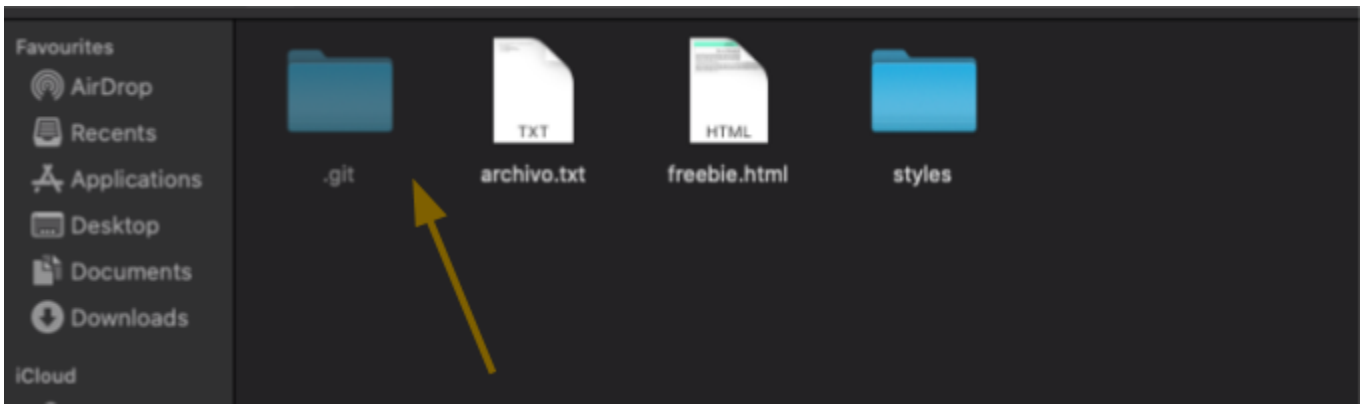
Comandos comunes

git init

Habiéndonos situado en la carpeta de nuestro proyecto, Git init nos permitirá iniciar el repositorio que incluirá todos los cambios históricos de nuestro proyecto.

$git init
weoka@MacBook-Air-de-Daniel Test % git init Initialized empty Git repository in /Users/weoka/Documents/Test/.git/ weoka@MacBook-Air-de-Daniel Test %

El comando generará la carpeta oculta .git:



git add

Refiriéndonos al estado de los archivos mencionados antes, git add nos permitirá empezar a rastrear archivos y añadirlos directamente al *staging area*.

$git add	
modificador (opcional)	Git add nos permite agregar modificadores. Por ejemplo, git add --all o git add --a para agregar todos los archivos que se encuentran en la carpeta padre del proyecto.
archivos	Nos permite indicarle archivos específicos. Por ejemplo: index.html. También podemos hacer uso del "." genérico para añadir todos los archivos en la carpeta padre del proyecto.

Por ejemplo:

$git add index.html

Rastreará los cambios producidos sobre el archivo "index.html".

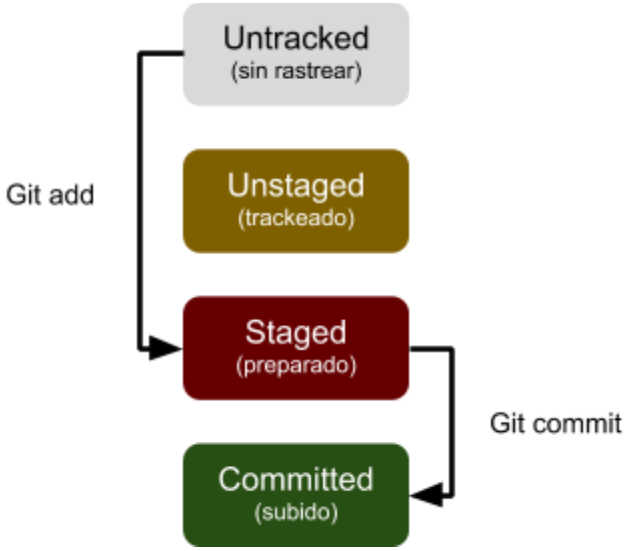
$git add --all

Rastreará todos los archivos que estén en la carpeta del proyecto. Lo mismo sucederá si usamos:

$git add .

git commit

Como mencionamos al referirnos a los diferentes estados de un archivo, los archivos que se encuentran en *staging area* están preparados para añadirse definitivamente al repositorio. Para hacerlo, utilizamos git commit. Como consideración debemos tener en cuenta que no puede existir ningún commit sin un mensaje de explicación (es decir, un mensaje que explique la razón del commit. Por ejemplo, comentando brevemente los cambios realizados).



$git commit	
-m	Es un modificador que le indica a git que añadiremos directamente un mensaje que describe el conjunto de cambios que incluyen los cambios que queremos guardar.
mensaje	El mensaje que queremos añadir a los archivos que estamos añadiendo al commit

Por ejemplo:
$git commit -m “Versión 1.0 del proyecto”
`[master 2511a90] Versión 1.0 del proyecto`
`1 file changed, 2 insertions(+), 1 deletion(-)`	

-am	Permite pasar archivos “unstaged” directamente al *staging area*, para posteriormente realizar un commit con ellos. **Vale la pena destacar que no sirve para archivos que no estén rastreados.**
mensaje	El mensaje que queremos añadir al commit que se realizará.

Por ejemplo:
$git commit -am “Versión 1.0 del proyecto”
Añadirá al staging area todos los archivos trackeados y posteriormente les hará un commit:
`[master 2511a90] Versión 1.0 del proyecto`
`1 file changed, 2 insertions(+), 1 deletion(-)`	

--amend	Realiza un commit que sobrescribirá el anterior.
-m	Permite modificar el mensaje del commit que reescribirá al anterior.
mensaje	El mensaje que queremos añadir al commit que se realizará.
Por ejemplo: $git commit --amend -m “Versión 1.1 del proyecto” En caso de que no tuviésemos cambios en el último commit, nos permitiría cambiar el mensaje del commit anterior por “versión 1.1 del proyecto”. Si tenemos cambios en el staging area, lo que sucederá es que esos cambios tomarán el lugar del anterior commit y este desaparecerá.	

git status

Git status nos permite ver el estado de los archivos de nuestro proyecto.

$git status

Si no hay ningún archivo listo para enviar a commit que no se haya enviado, aparecerá:
`nothing to commit, working tree clean`
 	
Si hay archivos sin rastrear nos lo hará saber:
`Untracked files:`
`(use "git add <file>..." to include in what will be committed)`
 	
`readme.html`
 	
También nos hará saber cuando un archivo esté en el staging area listo para ser enviado a un commit:

```
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   readme.html
```

git log

Git log nos permitirá ver la historia de commits de nuestro repositorio.

$git log	

Haciendo uso de únicamente el comando git log, se nos mostrarán los últimos commits del proyecto:
`commit 2511a90c29df4a33120e7cf38678bd00b06c6c49 (HEAD -> master)`
`Author: Weoka <weokap@gmail.com>`
`Date:   Wed Oct 21 10:06:03 2020 +0200`
 	
`    Versión 1.0 del proyecto`	

--all	Muestra absolutamente todos los commits de la historia del proyecto en caso de que no quepan únicamente en git log.
--graph	Estructura los commits de una forma más gráfica
--decorate	Decora el historial de una forma más clara
--oneline	Muestra el historial en una sola línea.

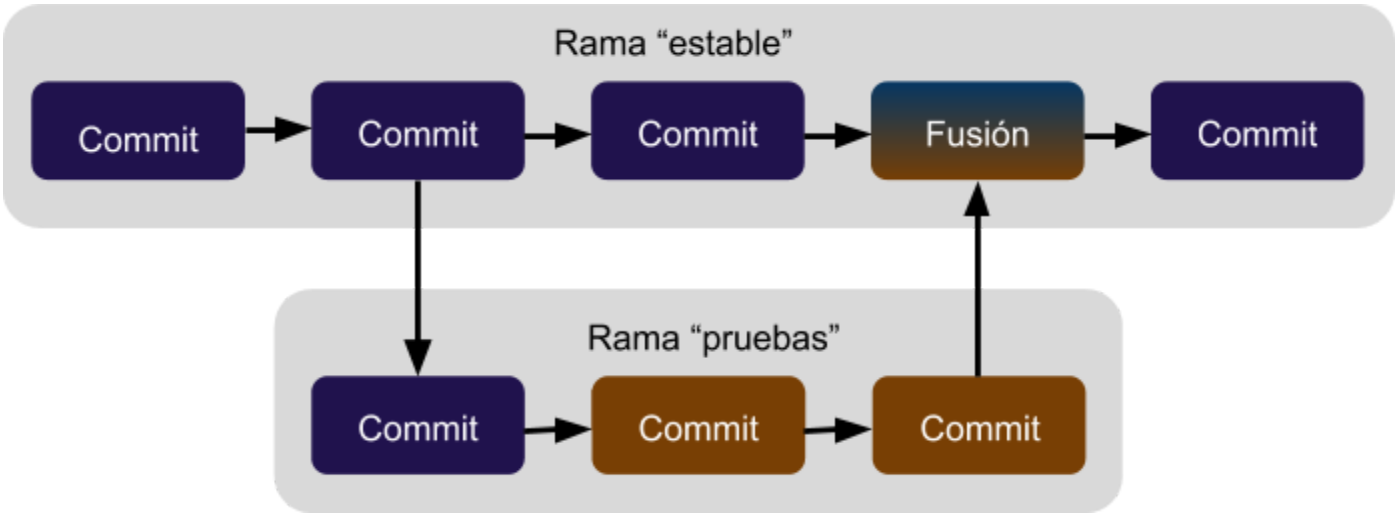
Podemos juntar todos los modificadores (git log --all --graph --decorate --oneline) y obtener algo como esto:
 	
`*   13b7e05 (HEAD -> master) Solución de conflicto`
`|\`
`| * d5583e5 (header) Slogan changes`
`* | 648b70d Confused slogan`
`* |   1f780c0 Merge branch 'header'`
`|\ \`
`| |/`
`| * 58bb094 Header styles`
`| * 61dd2a8 Primeros pasos del header`
`* | 2271fe3 Cambio de font`
`|/`
`* b4c343d Estandarización de entry`
`* 0ec99c8 new course`
`* d88dfa7 new course`
`* 4190a70 Site web start`
`* 7455140 Ultima versión`
`* 4caa297 Primer commit`	

-S	Permite buscar dentro de los commits.
query	Parámetro que queremos encontrar.
Por ejemplo: git log -S “ERROR_BAD_EMAIL” Nos retornará el commit en el que exista la palabra “ERROR_BAD_EMAIL”.	

git branch

Las ramas (branches) son una de las utilidades más potentes de Git. Nos facilita crear características, probar nuevo código y trabajar con más personas sin tocar la versión estable de nuestro proyecto (Es decir, aquella que ya está funcionando).

Una vez terminemos nuestras pruebas o nuestros compañeros terminen sus características y estén perfectamente probadas, podrán fusionarlas con nuestra versión estable de código de manera relativamente sencilla.



Lo que nosotros llamamos “rama estable”, git lo llama por defecto rama master, mientras tanto todas las demás ramas pueden ser creadas con cualquier nombre. Para manipular, gestionar y crear ramas, disponemos del comando git branch.

$git branch	
nombre_rama	Crea una nueva rama.
$git branch header Creará la rama “header”	
-a	Muestra todas las ramas de nuestro repositorio, tanto locales como remotas. Marcará con un asterisco en la que nos encontremos situados en ese momento.

$git branch -a
`header`
`* master`	

-d	Modificador de git branch que permite eliminar ramas.
nombre_rama	El nombre de la rama que queremos eliminar
$git branch -d header `Deleted branch header (was d5583e5).`	
-m	Modificador de git branch que permite renombrar ramas.

nombre_rama_antiguo	El nombre de la rama a la que le queremos modificar el nombre
nombre_rama_nuevo	Nuevo nombre
$git branch -m master main Cambiará el nombre de nuestra rama “master” por “main”.	
-r	Modificador de git branch que muestra todas las ramas remotas

git checkout

No tendría sentido que pudiésemos guardar nuestros cambios en commits y crear ramas si luego no podemos viajar a través de ellas. Precisamente es eso lo que nos permite git checkout: viajar a través de la historia de git. Es decir, movernos a commits pasados y ramas alternas.

Para ello, necesitaremos conocer el id de los commits (el identificador que utiliza Git para guardar en su base de datos cada serie de cambios). Git nos lo enseña cuando hacemos uso de git log.

```
commit 648b70dd1b0211ba3177bda968ab381ffe635388
Author: Weoka <weokap@gmail.com>
Date:   Mon Oct 12 18:06:09 2020 +0200

    Confused slogan
```

Por ejemplo, en ese caso mostrado anteriormente, el id del commit sería: 648b70dd1b0211ba3177bda968ab381ffe635388

$git checkout	
posicion	Se posiciona en donde se le indique. El argumento puede ser el nombre de una rama o el identificador de un commit.
$git checkout 648b70dd1b0211ba3177bda968ab381ffe635388 Nos traería a nuestro directorio local todos los archivos que se encontraban en el commit “confused slogan”. De igual manera funciona para ramas. Por ejemplo, podríamos volver a nuestro contenido más actual: $git checkout master Y así volveríamos a la rama master de nuestro proyecto. Pero en realidad podríamos situarnos en cualquier otra: $git checkout ramax	
posicion	Se posiciona en donde se le indique. El argumento puede ser el nombre de una rama o el identificador de un commit.
archivo	Traerá de vuelta un solo archivo de la posición que se le indique.
$git checkout 648b70dd1b0211ba3177bda968ab381ffe635388 index.php Git checkout nos traería de vuelta al master el archivo index.php que existía en el commit “648b70dd1b0211ba3177bda968ab381ffe635388”.	
-b	Modificador que permite crear una nueva rama y posicionarse directamente en ella.
nombre_rama	Nombre de la rama que se creará
$git -b cambios	

Creará la rama “cambios” y se situará directamente en ella.

git merge

Una vez terminamos con la vida de una rama (por ejemplo, porque la característica que queríamos desarrollar ya está completada), podemos fusionarla (merge en inglés) con nuestra rama principal (comúnmente llamada master) o cualquier otra.

$git merge	
nombre_rama	Fusionará la rama en la que nos encontremos con la que se indique en el nombre.
Es importante destacar que debemos estar situados en la rama definitiva. (Es decir, si queremos fusionar una rama llamada “header” a master, deberemos estar en master). Entonces habiéndonos situado en master con $git checkout master podemos proceder a fusionar la rama con $git merge header. Vale la pena destacar que este proceso no borrará la rama fusionada, simplemente la fusionará con la rama indicada.	
--allow-unrelated-histories	Habrá ocasiones en las que la historia de commits de una rama sea significativamente diferente a otra. En estos casos git no nos permitirá realizar la fusión a menos que añadamos el modificador --allow-unrelated-histories a modo de confirmación.
$git merge --allow-unrelated-histories header Nos fusionará la rama header con la rama en la que nos encontremos sin importar que la historia de ambas ramas sea muy diferente. Este modificador suele ser especialmente útil para el trabajo con repositorios remotos.	

git diff

Para poder saber con precisión las diferencias entre un commit y otro, podemos hacer uso del comando git diff.

$git diff	
entrada1	Elemento 1 que se quiere comparar (hash de commit o nombre de rama).
entrada2	Elemento 2 que se quiere comparar (hash de commit o nombre de rama).
$git diff 648b70dd1b0211ba3177bda968ab381ffe635388 d5583e578afcf9c6649db11fb1381c40916f16f2 Mostraría los cambios que existen entre el commit “648b70dd1b0211ba3177bda968ab381ffe635388” y “d5583e578afcf9c6649db11fb1381c40916f16f2”. Vale la pena recalcar que el orden importa, en esta ocasión se hace una diferencia entre los cambios de entrada1 en relación a entrada2, pero si se ingresaran los commits al revés la diferencia sería al revés.	
entrada	Si se hace una sola entrada se compara con la posición en la que nos encontremos.
Por ejemplo, si estando posicionados en el master añadimos: $git diff 648b70dd1b0211ba3177bda968ab381ffe635388 Se nos retornarán las diferencia entre el commit “648b70dd1b0211ba3177bda968ab381ffe635388” con respecto a la rama master.	

git show

Muestra de manera detallada las modificaciones o adiciones a la que fue sometido un commit.

$git show	
entrada	Elemento del que se quiere ver las modificaciones.

$git show 13b7e05803f22147cc8cbfa7fd28d7d3e2f4127d

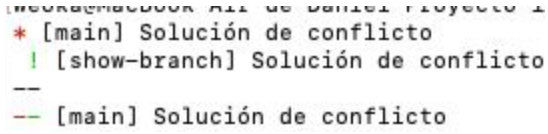
Nos devolverá los cambios que representan al commit “13b7e05803f22147cc8cbfa7fd28d7d3e2f4127d”:

git show-branch

Nos enseña las ramas existentes y su historia.

$git show-branch	
--all	Muestra todas las ramas que existen.

$git show-branch --all



git grep

Nos permite buscar palabras dentro del repositorio y nos indica el archivo en el que se utilizó.

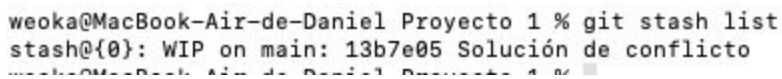
$git grep	
término	El término de búsqueda que queremos encontrar.
-c	Muestra el archivo que lo contiene y cuántas veces se ha usado.
-n	Muestra el archivo que lo contiene y el número de línea exacto donde se encuentra.

$git grep -n “echo” Nos mostrará los archivos y líneas exactos que contienen la palabra “echo”.

git stash

Git stash nos permite guardar aquellos cambios que no queremos perder, pero que no están listos para ser guardados en forma de commit. Por ejemplo, supongamos que estamos trabajando en una característica que aún no está terminada y repentinamente aparece un error que requiere ser resuelto lo antes posible.

Si nos cambiáramos de rama para solucionar el error sin antes hacer commit perderíamos todo el progreso que tengamos. Para solucionar esta situación disponemos del comando git stash, que guarda todo aquello que no hayamos guardado como commit en un stash.

<table>
<tr><td colspan="2">$git stash</td></tr>
<tr><td colspan="2">$git stash crea un stash con el contenido al que no se le haya hecho commit. Es decir, ya podríamos hacer $git checkout a cualquier otro estado del proyecto sin perder nuestro trabajo.</td></tr>
<tr><td>list</td><td>Retorna un listado de todos los stash que tengamos.</td></tr>
<tr><td colspan="2">$git stash list
  </td></tr>
<tr><td>apply</td><td>Recupera a nuestro directorio de trabajo el último stash creado.</td></tr>
<tr><td>stash</td><td>Podemos especificarle un parámetro stash de la forma stash@{numero stash} para recuperar un stash específico en vez del último.</td></tr>
<tr><td colspan="2">$git stash apply stash{0}
Recuperará en nuestro working directory el stash #0.</td></tr>
<tr><td>branch</td><td>Crea una rama, guarda el contenido sin commitear y se posiciona en ella.</td></tr>
<tr><td>rama</td><td>Nombre de la rama a crear.</td></tr>
<tr><td colspan="2">$git stash branch ecommerce
Creará el branch “ecommerce” y guardará en ella el último stash.</td></tr>
<tr><td>drop</td><td>Elimina el último stash guardado.</td></tr>
<tr><td>stash</td><td>Si en vez de eliminar el último queremos eliminar un stash específico, podemos especificarlo.</td></tr>
<tr><td colspan="2">$git stash drop stash@{0}
Borrará el stash #0.</td></tr>
<tr><td>pop</td><td>Recupera el contenido del último stash en nuestro working directory y borra el stash.</td></tr>
<tr><td colspan="2">$git stash pop
Recuperaría stash@{0} al working directory y borraría el stash.</td></tr>
</table>

git reset

Vimos el uso de git checkout como una forma de ir a cualquier otro commit o rama del repositorio. Obviamente, así como este comando nos permitía ir al pasado también nos permitía volver al presente. Pues bien, con git reset también podemos volver a una versión antigua del proyecto, pero con la diferencia de que esta sobrescribirá el estado actual del proyecto y no podremos volver al futuro.

<table>
<tr><td colspan="2">$git reset</td></tr>
<tr><td>--soft</td><td>Nos permite retornar a un estado pasado de un proyecto sobrescribiendo la historia y registros de git, pero manteniendo lo que tenemos en nuestro staging area.</td></tr>
<tr><td>commit</td><td>El commit al que queremos volver.</td></tr>
<tr><td colspan="2">Si tenemos un proyecto con este log:
 

Y usamos git reset soft para retornar al commit “Confused slogan”, utilizaríamos el siguiente comando.
$git reset --soft 648b70dd1b0211ba3177bda968ab381ffe635388
Tras hacerlo el commit “Solución de conflicto” dejará de existir y el último commit de la historia del proyecto será “Confused slogan”. Aún así, todo el contenido que tengamos en staging se mantendrá.</td></tr>
<tr><td>--hard</td><td>Nos permite retornar a un estado pasado de un proyecto sobrescribiendo la historia y registros de git, al igual que el contenido de nuestro working directory.</td></tr>
<tr><td>commit</td><td>El commit al que queremos volver.</td></tr>
<tr><td colspan="2">Si tenemos el mismo caso que en el ejemplo anterior, pero usamos $git reset --hard 648b70dd1b0211ba3177bda968ab381ffe635388 en vez de soft, “Confused slogan” pasaría a ser igualmente el último commit del historia. Además, reemplazaría el contenido de nuestro working directory.</td></tr>
<tr><td>HEAD</td><td>Nos permite quitar elementos del staging area. Así, podremos eliminar archivos a los que no queramos hacerles commit. Para añadirlos de nuevo, haríamos uso de $git add</td></tr>
</table>

git rm

Vimos la existencia del comando git add para añadir archivos al *staging area*. En ese sentido podemos ver a git rm como su opuesto: Nos permite eliminar archivos del *staging area* y si lo queremos, también del *working directory.*

$git rm	
--cached	Borra los archivos del *staging area*, pero los mantiene en el *working directory.*
archivo	El archivo a eliminar
$git rm --cached index.php Eliminará el archivo “index.php” del *staging area* pero lo dejará en el *working directory.*	
--forced	Borra los archivos tanto del *staging area* como del *working directory.*
archivo	El archivo a eliminar
$git rm --forced index.php Eliminará el archivo “index.php” del *staging area* y el *working directory.*	
--dry-run	Podemos hacer uso de --dry-run para hacer una prueba experimental. Es decir, que git nos indique qué elementos eliminaría sin necesidad de eliminarlos realmente..

git rebase

Vimos el uso de git merge como una forma de fusionar los cambios de una rama con otra (por ejemplo, con la rama master). En este proceso, únicamente mantenemos el último commit de ambas ramas. Si quisiéramos adicionar a la rama final todo el historial de commits de la rama que fusionaremos, podemos hacer uso de git rebase.

$git rebase	
-i	Permite añadir de manera interactiva todos los commits de una rama a otra.
rama	Rama a superponer.
Si estando situados en una rama llamada “hotfix” hacemos uso del comando $git rebase -i master Una vez se realice, procedemos a hacer lo mismo desde la rama master: $git rebase -i hotfix Tras hacerlo, ambas ramas quedarían fusionadas conservando el historial de ambas en una sola de manera interactiva.	

git cherry-pick

git cherry-pick es una utilidad que nos permite importar commits de otras ramas.

$git cherry-pick	
hash	Estando situados en la rama en la que queremos importar el commit, ingresamos el hash del commit que queremos traer.
--no-commit	Podemos hacer uso del modificador --no-commit para que git traiga el commit a nuestro working directory en vez de generar un commit en la rama en la que nos encontramos.
$git cherry-pick --no-commit 648b70dd1b0211ba3177bda968ab381ffe635388 Nos traería el contenido del commit "648b70dd1b0211ba3177bda968ab381ffe635388" al working directory.	

git clean

Si quisiéramos eliminar archivos que no están rastreados en git.

$git clean	
--dry-run	Hace una simulación sobre los archivos que se borrarán.
-f	Elimina todos los archivos que no están siendo rastreados por git.
-d	Elimina todas las carpetas que no están siendo rastreadas por git.
$git clean -f -d Eliminaría todas las carpetas y archivos que no están siendo rastreados por git. Podemos hacer uso de --dry-run para tener una previsualización de qué se borrará antes de que git proceda a eliminar.	

git reflog

Vimos la existencia del comando git log para ver el historial de un repositorio. Ahora veremos el comando git reflog, un comando que también nos permite ver la historia de un repositorio pero con ciertos matices. Por ejemplo, git log tiene información que vive en el repositorio (local o remoto). Mientras que git reflog únicamente está disponible de forma local. De este modo, git log representa la historia del repositorio (historia que puede ser modificada como hemos visto a través de comandos como reset o rebase), mientras que git reflog, muestra la historia de un repositorio en el ámbito local, que no puede ser modificada con el uso de comandos que modifiquen la historia del repositorio.

$git reflog

git --help

Al igual que cualquier otro comando de consola, git también tiene un asistente que provee información sobre cualquier comando. Para exprimir al máximo la capacidad de cada utilidad de git, nunca es mala idea añadir el modificador --help para ver qué más tiene para ofrecer un comando.

$git commit --help

Trabajo colaborativo en Git

Una de las mayores bondades de Git es su aplicación al trabajo colaborativo: Ya no hará falta preocuparnos a la hora de trabajar con un equipo en un mismo proyecto (o al menos no de la misma forma que antes).

Comandos comunes para el desarrollo en equipo

git blame

Git blame (culpa en inglés) nos permite saber quién hizo qué sobre un archivo de manera detallada.

$git clean	
nombre_archivo	El nombre del archivo sobre el que queremos obtener información.
-c	El modificador -c permite ver quién hizo cada línea especificando la línea y la fecha y hora con tabulaciones entre cada atributo.
-L	Permite establecer el rango de líneas que se quiere estudiar en un archivo. Por ejemplo: -L35,53 mostrará las personas que han intervenido entre las líneas 35 y 53 de un archivo.

git shortlog

Git shortlog permite ver información sobre cada colaborador en relación a su interacción con el proyecto.

$git shortlog	
Muestra todos los commits hechos por cada uno de los integrantes del proyecto.	
nombre_archivo	El nombre del archivo sobre el que queremos obtener información.
-sn	Muestra el número de commits hechos por cada uno de los integrantes.

git tag

Los tags son referencias que apuntan a estado específicos en la historia de un repositorio, se utilizan generalmente para capturar un punto en la historia que se utilizará para marcar algún goal o release del código (por ejemplo, un tag podría ser la versión 1.0 de un proyecto).

$git tag	
-l	Lista todos los tags existentes.
-a	Indica que se añadirá un tag
nombre_tag	Indica el nombre del tag.
-m	Mensaje que describa el contenido del tag.
hash	Hash al que queremos asignarle un tag.

Github

Github es una plataforma que permite alojar un repositorio en internet, compartirlo con más personas y trabajar en él de forma colaborativa. Los comandos más comunes en relación a los repositorios remotos son los siguientes.

git clone

Git clone es un comando de git que permite copiar localmente el contenido de repositorios remotos. Puede hacerse bien a través del protocolo HTTP o SSH.

$git clone	
URL	URL HTTP o SSH que apunta al repositorio.
Estando posicionado en la carpeta en la que queremos copiar nuestro repositorio, podríamos ejecutar: git clone git@github.com:DogeSo/core_backend.git para copiar el contenido e historia del repositorio “core_backend”.	

git remote

Git remote es un comando que permite vincular nuestro repositorio local con un repositorio remoto. De esta manera podremos subir los cambios realizados localmente al repositorio online, al igual que podremos descargar localmente los cambios hechos por otros.

$git remote	
add	Nos permite especificar el repositorio remoto al que estará vinculado nuestro repositorio local

rama	Permite especificar la rama remota que queremos vincular.
URL	Y la URL del repositorio remoto.
$git remote add origin git@github.com:DogeSo/core_backend.git Vincularía nuestro repositorio remoto con la rama “origin” del repositorio remoto.	
remove	Nos permite eliminar un vínculo remoto.
rama	Especifica la rama que queremos eliminar.
$git remote remove origin Eliminaría la rama recién añadida.	
set-url	Permite modificar la URL de un repositorio remoto vinculado.
rama	Especifica la rama de un repositorio remoto al que le queremos modificar la URL.
URL	Especifica la nueva URL.
$git remote set-url origin git@gitlab.com:DogeSo/core_backend.git Cambiaría la URL a la que se encuentra vinculada origin por “git@github.com:DogeSo/core_backend.git”	
-v	Retorna todas las conexiones existentes.

$git remote -v retornaría algo similar a esto si hay vinculado algún repositorio remoto:
`weoka@MacBook-Air-de-Daniel platzi-notes % git remote -v`
`origin  git@github.com:weoka/platzi-notes.git (fetch)`
`origin  git@github.com:weoka/platzi-notes.git (push)`	

git fetch

Con el comando git fetch podemos cargar todo el contenido de nuestro repositorio remoto en nuestro repositorio local. Debe tenerse en cuenta que el repositorio remoto se encuentra en una rama diferente a la de nuestro master.

Es decir, cuando importamos un repositorio remoto podríamos tener la existencia de dos ramas: La rama master (que es la que crea git por defecto) y la rama origin (que es la que importamos desde el repositorio remoto). Por tanto, el comando git fetch únicamente actualiza el contenido de la rama origin y si quisiéramos tener dichos cambios en nuestra rama master deberíamos realizar un merge entre ambas ramas.

$git fetch
Actualiza el contenido de los repositorios remotos.

git pull

Por el contrario que git fetch que únicamente obtiene los cambios del repositorio remoto pero que no los fusiona con el repositorio master, está git pull, que obtiene los cambios desde el repositorio remoto y realiza un merge con la rama en la que nos encontremos posicionados.

$git pull	
Actualiza el contenido de los repositorios remotos y realiza un merge con la rama en la que estemos situados. Es decir, si nos encontramos en la rama master, el merge se haría con la rama master.	
rama	Permite especificar la rama remota que queremos importar y fusionar.
$git pull cabecera obtendría la rama cabecera y la fusionaría con la rama en la que nos encontremos situados.	
--allow-unrelated-histories	Al igual que sucedía con este mismo parámetro en el caso de git merge, permite fusionar ramas que no tienen una historia en común.

git push

git push nos permitirá enviar nuestros cambios locales al repositorio remoto.

$git push	
repositorio_remoto	Especifica la rama remota a la que queremos enviar cambios.
repositorio_local	Especifica la rama local a la que queremos enviar cambios.
--tags	Permite enviar los tags creados localmente al repositorio remoto.
$git push origin master Enviaría el contenido desde la rama “master” a la rama remota “origin”.	
:refs/tags/nombre_tag	Permite que un tag remoto sea actualizado por uno local.