

Uso de GIT

Control de versiones

¿Qué es git?

Es un sistema de control de versiones que registra todos los cambios realizados en un proyecto y permite trabajar de forma colaborativa.

Fue creado por Linus Torvalds

Ventajas y desventajas

Ventajas

- Auditoría del código: para controlar quién ha tocado cada parte del código
- Control de cambios: para retroceder en el tiempo
- Uso de etiquetas para marcar cambios importantes importantes
- Trabajo colaborativo en equipo

Desventajas

- No es fácil de aprender, curva de aprendizaje elevada
- Está orientado a código fuente (formato texto) y no es adecuado para control de versiones para gráficos o videos

Ventajas y desventajas

Ventajas

- Auditoría del código: para controlar quién ha tocado cada parte del código
- Control de cambios: para retroceder en el tiempo
- Uso de etiquetas para marcar cambios importantes importantes
- Trabajo colaborativo en equipo

Desventajas

- No es fácil de aprender, curva de aprendizaje elevada
- Está orientado a código fuente (formato texto) y no es adecuado para control de versiones para gráficos o videos

Otros controles de versiones parecidos

- TortoiseSVN
- Subversion
- Concurrent Versions System (CVS)
- Aegis: escrito por Peter Miller, es un programa maduro, orientado a sistemas de ficheros, con soporte de red limitado.
- ArX: escrito por Walter Landry, empezó como una rama de GNU arch, pero ha sido totalmente reescrito.
- Bazaar: escrito en Python por Martin Pool y patrocinado por Canonical es un sistema descentralizado, que intenta ser rápido y fácil de usar.
- Codeville: escrito en Python por Ross Cohen; usa un algoritmo de injerto innovador.
- Darcs: escrito en Haskell y desarrollado originalmente por David Roundy, puede llevar el seguimiento de dependencias inter-parche y reagruparlas automáticamente y escogerlas usando "teoría de parches".
- DVCs: CVS descentralizado.
- Fossil: escrito por Richard Hipp para SQLite, presenta un control de versiones distribuido, wiki y seguimiento de fallos.
- Git: escrito en una combinación de Perl, C y varios scripts de shell, estuvo diseñado por Linus Torvalds según las necesidades del proyecto del kernel de Linux; con los requisitos de descentralización, rápido, flexible y robusto.
- GNU arch: discontinuado, sustituido por Bazaar.
- LibreSource: gestión de configuración.
- Mercurial: escrito en Python como un recambio en software libre de
- Bitkeeper; descentralizado, que pretende ser rápido, ligero, portable y fácil de usar.
- Monotone: descentralizado y funcionando en modo peer-to-peer (P2P).
- SVK: escrito en Perl por Kao Chia-liang sobre la base de subversión permitiendo hacer commit distribuidos.

Ángel González M.

Subir un proyecto existente a BitBucket o Github (Método manual)

```
cd <ruta de tu proyecto>
```

```
git init
```

```
git remote add origin https://AngelGonzalezM@bitbucket.org/AngelGonzalezM/prueba.git
```

```
git add --all
```

```
git commit -m "Commit inicial"
```

```
git push -u origin master
```

Clonar (descargarse) un proyecto de BitBucket o Github (Método manual)

```
cd <ruta de tu workspace>
```

```
git clone https://AngelGonzalezM@bitbucket.org/AngelGonzalezM/prueba.git <nombre_carpeta>
```

Añadir algo a un proyecto existente

...Añadimos ficheros nuevos o hacemos cambios en ficheros existentes de nuestro proyecto...

```
git add --all
```

```
git commit -m "Segundo commit"
```

```
git push -u origin master
```


Recuperar y unir la rama remota con tu rama actual

`git pull`

Esto es parecido a: `git fetch + git merge`

Instalar Git

Funciona en Windows, Linux y Mac.

<https://git-scm.com/>

En linux podemos instalarlo de forma más sencilla ej debian/ubuntu: con el comando:

sudo apt install git

Los 3 estados de Git

Working directory: Es donde editamos y trabajamos con nuestros archivos, es nuestro editor de código.

Staging area: Es donde escogemos los archivos que están listos para pasar al 3er estado. Al igual que decidimos que archivos no están listos aún para pasar, o cuales nunca van a pasar.

Repository: Aquí está el registro de todo lo que hicimos (desde siempre)

- Repositorio local
- Repositorio remoto



Estados de GIT

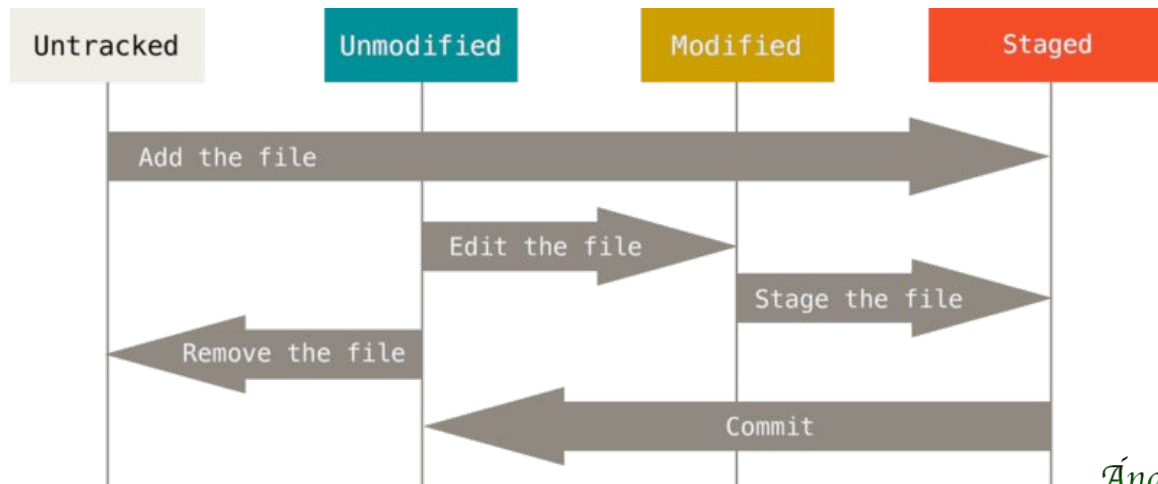
Git tiene tres estados principales en los que se pueden encontrar tus archivos:

- **Confirmado** (committed), modificado (modified), y preparado (staged).
Confirmado significa que los datos están almacenados de manera segura en tu base de datos local. Git (Git directory) **Repository**
- **Modificado** significa que has modificado el archivo pero todavía no lo has confirmado a tu base de datos. Directorio de trabajo (**working directory**)
- **Preparado** significa que has marcado un archivo modificado en su versión actual para que vaya en tu próxima confirmación. El área de preparación (**staging area**) (index)

Flujo de trabajo de GIT

El flujo de trabajo básico en Git es algo así:

- Modificas una serie de archivos en tu directorio de trabajo.
- Preparas los archivos, añadiéndolos a tu área de preparación
- Confirmas los cambios, lo que toma los archivos tal y como están en el área de preparación y almacena esa copia instantánea de manera permanente en tu directorio de Git.



Ángel González M.

Configurar el nombre de usuario y el password

Lo primero que deberás hacer cuando instales Git es establecer tu nombre de usuario y dirección de correo electrónico. Esto es importante porque los "commits" de Git usan esta información, y es introducida de manera inmutable en los commits que envías:

Para ver algún valor de la configuración

git config --global user.name

Para eliminar algún parámetro que ya no queramos en la configuración (--unset)

git config --global --unset color.ui true

Primeras configuraciones

Ver la versión de git

git --version

Ver la configuración de git

git config --list

Añadiendo nuestro nombre. Util para indicar quienes somos cuando guardamos o realizamos alguna acción

git config --global user.name "Angel Gonzalez"

Añadiendo nuestro email

git config --global user.email "gonzalezm@gmail.com"

Para que git nos ponga los mensajes con colorines

git config --global color.ui true

Ángel González M.

Ejercicio

Configura en git tu nombre y email

Solución:

```
git config --global user.name "PepitoPerez"  
git config --global user.email "tu@email.com"
```

Editor por defecto de GIT

Git usa el editor por defecto de tu sistema, que generalmente es Vim.

Si quieres usar otro editor de texto como Emacs, puedes hacer lo siguiente:

Para usar el editor de emacs;

git config --global core.editor emacs

Para usar el editor de Visual Studio Code

git config --global core.editor "code --wait"

Edición de la configuración global

Podemos editar

```
git config --global -e
```

Podemos simplemente verlo

```
git config --global -l
```

Ejercicio

Comprueba la versión instalada de tu git

Solución:

git --version

Comandos en GIT

comando: help

El comando git help nos da información de uso de git y sus comandos.

Uso: *git help* <nombre_comando>

Ejemplo:

git help status

git help commit

comando: Init

git init

Este comando sirve para iniciar un proyecto con git

Aquí le indicamos a git que comience a monitorear todos los cambios de nuestros archivos en el directorio en el que nos encontremos.

El comando solo se usa una vez, al iniciar un proyecto nuevo.

git init (avanzado)

Este comando crea una carpeta oculta llamada .git

Ver carpeta oculta:

- Windows: dir /ah
- Mac / linux: ls -al

En esta carpeta se guardan todos los datos del repositorio

comando: Status

git status

Este comando nos indica cual es el estado de nuestros archivos.

Nos dice cuáles archivos aún no han sido agregados al estado de **Staging area**. y cuales si.

Aparecerán de color **rojo** los archivos que estén en el working directory. Son archivos nuevos o archivos que han cambiado desde el último commit.

Aparecerán de color **verde** los archivos que están en el staging area

Ángel González M.

Avanzado: status de forma simplificada

Para ver el status en modo silence:

```
git status -s
```

Para ver el status en modo silence y además la rama en la que estamos

```
git status -s -b
```

Ejercicio

Crea un nuevo proyecto git en tu equipo

Solución:

mkdir proyecto-git

cd proyecto-git

git init

comando: Add

Este comando nos permite indicar qué archivos quiero añadir al estado de **Staging area**.

Por ejemplo puedo añadir al staging area un fichero llamado fichero.txt

git add fichero.txt

Por ejemplo puedo añadir todos los archivo usando el comando

git add --all

o también

git add .

Ejercicio

Crea un fichero en tu proyecto git y añade ese fichero al staging area

Solución:

code index.html

Editamos el fichero index.html insertando el código básico de una página web

git add index.html

```
!DOCTYPE html>

html lang="en">

  head>

    <title>Document</title>

  /head>

  body>

    <p>Mi pagina web </p>

  /body>

/html>
```

Para quitar un archivo del staging area

Podemos quitar un archivo que hemos metido en el staging area y volver a colocarlo en el working directory.

git reset HEAD <nombre_archivo>

Ejercicio: añade y quita un archivo del staging area

Creamos 2 archivos

code archivo1.txt

code archivo2.txt

Añadimos esos archivos al staging area

git add .

git status

Quitamos el archivo2.txt del staging area y lo ponemos en el working directory

git reset HEAD archivo2.txt

git status

Avanzado

Añade todos los archivos con extensión txt de TODO el proyecto

```
git add "*.txt"
```

Añade todos los archivos con extensión txt del directorio actual

```
git add *.txt
```

Añade todos los archivos

```
git add .
```

```
git add --all
```

Añade varios archivos

```
git add archivo1.txt archivo2.txt
```

Añade todos los archivos de una carpeta determinada

```
git add miCarpeta/
```

Ángel González M.

comando: Commit

Este comando guarda todos los cambios en el 3er estado, el estado de repositorio. Es decir pasa los archivos del staging area al repository area.

El Modificador **-m** permite poner un mensaje identificativo

git commit -m "primer commit"

NOTA: ejecuta git status para ver que información aparece

comando: Log

El comando git log nos va a dar un listado de todos los commits que hemos realizado.

git log

Si ejecutamos el comando, se mostrará por pantalla el nombre y email, la fecha y el mensaje del commit, así como un código muy largo llamado **sha** que identifica al commit.

```
git log
commit 587582f6444d501e5fef16999bb76736c5cc7a8f
Author: Angel Gonzalez <gonzalezm.angel@gmail.com>
Date: Thu Oct 20 13:48:27 2016 +0200
```

primer commit

Ángel González M.

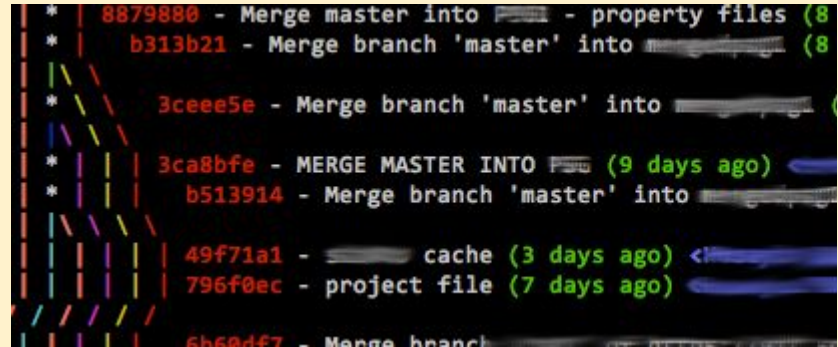
Avanzado: log con formato

Mostrar el log de forma sencilla

`git log --oneline`

Mostrar el log de forma gráfica

`git log --oneline --decorate --all --graph`



```
* | 8879880 - Merge master into branch - property files (8 days ago)
* | b313b21 - Merge branch 'master' into branch (8 days ago)
* | 3ceee5e - Merge branch 'master' into branch (8 days ago)
* | 3ca8bfe - MERGE MASTER INTO branch (9 days ago)
* | b513914 - Merge branch 'master' into branch (9 days ago)
* | 49f71a1 - branch cache (3 days ago) <1
* | 796f0ec - project file (7 days ago)
* | 6b60df7 - Merge branch 'master' into branch
```

comando: checkout

Este comando permite viajar en el tiempo y posicionarnos en un commit en particular (también para cambiar de ramas, lo veremos después).

git checkout <codigo_sha_commit>

Si queremos posicionarnos en el último commit de mi rama

git checkout master

Deshacer lo últimos cambios

Si necesitas deshacer los últimos cambios que has hecho usa el comando para que la información quede como estaba en el último commit usa el comando

En caso de que hagas algo mal, puedes reemplazar cambios locales usando el comando

git checkout -- <filename>

git checkout -- .

Este comando reemplaza los cambios en tu directorio de trabajo (working directory) con el último contenido de HEAD. Los cambios que ya han sido agregados al Index, así como también los nuevos archivos, se mantendrán sin cambio.

Ángel González M.

Ejercicio:

Crea un fichero en tu repositorio local llamado prueba.txt

code prueba.txt

Edita el fichero y rellénalo con algún contenido

Añade el fichero al repositorio

git add prueba.txt

git commit -m "fichero de prueba añadido"

Edita de nuevo el fichero y realiza algún cambio

code prueba.txt

Deshaz los últimos cambios para que los datos queden con la información guardada en el último commit.

git checkout -- .

Ángel González M.

Deshacer lo últimos cambios

Con este comando puedes pasar ficheros del staging area y devolvernos al estado de working directory.

git reset HEAD <fichero>

Ejercicio

Crea un fichero en tu repositorio local llamado pruebaR.txt

code pruebaR.txt

Edita el fichero y rellénalo con algún contenido

Añade el fichero al staging area

git add pruebaR.txt

git status

Quítalo del staging area y devuélvelo al estado de working

git reset HEAD pruebaR.txt

git status

Agregando todos los archivos y staging y hacer el commit

```
git commit -am "Subi un nuevo fichero"
```

Corregir el mensaje del último commit

```
git commit --amend -m "Nuevo mensaje"
```

Ejercicio

Crea un fichero en tu repositorio local llamado pruebaA.txt

code pruebaA.txt

Edita el fichero y rellénalo con algún contenido

Añade el fichero al repositorio (y comete a propósito un error en el mensaje)

git commit -am "Introduce un nuevo fichero"

Corrige el mensaje usando amend

git commit --amend -m "Introduje un nuevo fichero"

git log

concepto: HEAD

Head es el commit donde nos encontramos actualmente

Añadir más archivos al ultimo commit

Uno de las acciones más comunes, es cuando confirmas un cambio (commit) antes de tiempo y olvidas agregar algún archivo adicional.

Con este comando podrás añadir al último commit los archivos que has olvidado.

git add xxxxxx

git commit --amend

Deshacer los cambios realizados en el working directory

Imagina que tienes un fichero comiteado en el repositorio y haces cambios en él

code fichero.txt

Te das cuenta que te has equivocado y quieres deshacer todos los cambios. Es decir quieres volver al estado en que quedó el sistema en el último commit.

git checkout -- fichero.txt

git status

NOTA: este comando es peligroso, pues perderás todos los cambios realizado en el working directory

diff

Nos indica cual ha sido la última modificación entre el commit anterior y le momento actual (working directory)

```
# git diff
```

Pero si hemos añadido los archivos al staging area esto no mostrará esos cambios. Para mostrar cuál ha sido la última modificación entre le commit anterior y el staging area usaremos:

```
# git diff --staged
```

Ejercicio

Crea un fichero en tu repositorio local llamado pruebaD.txt

code pruebaD.txt

Edita el fichero y rellénalo con algún contenido

Añade el fichero al repositorio

git add .

git commit -m "cambios add"

Edita el fichero de nuevo incorporando algunos cambios, y vuelve a guardarlo

code pruebaD.txt

Comprueba que cambios has realizado

git diff

Añade el fichero pruebaD.txt al staging area

git add .

Comprueba los cambios que has realizado

git diff --staged

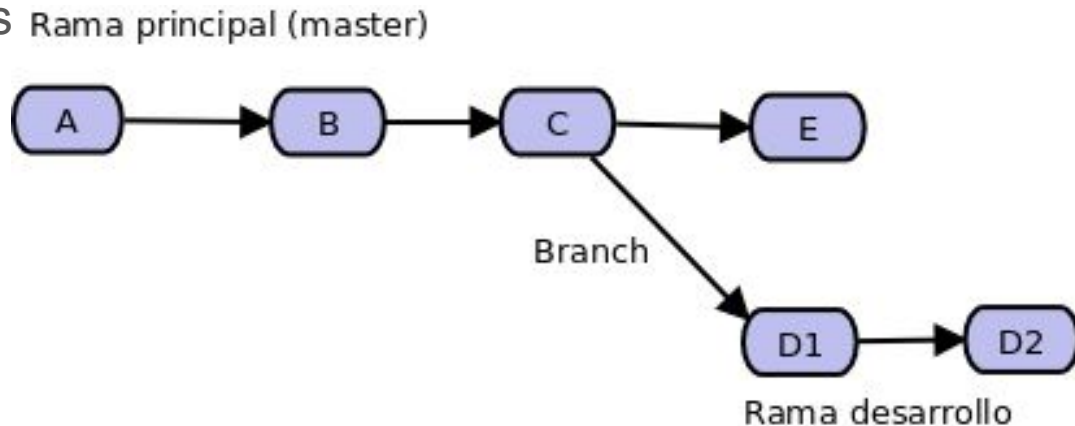
Ramas

concepto: Ramas

Es una línea de tiempo en nuestro proyecto

Sirve para:

- Hacer cosas que no afecten al proyecto en sí.
- Reparar bugs
- Hacer experimentos y pruebas
- Hacer grandes cambios
- ...



concepto: Rama Master

Es la rama que se crea al iniciar git por primera vez (**git init**)

Es la rama principal y estable de nuestro proyecto.

comando: Branch

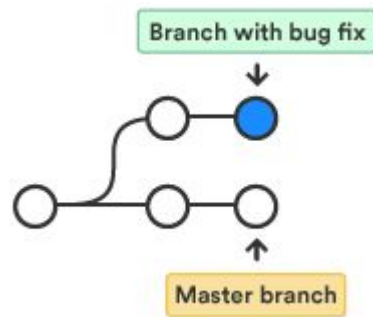
Este comando muestra la rama en la que me encuentro si simplemente escribo

git branch

También sirve para crear nuevas ramas

git branch test

(recuerda que `git checkout test` nos sirve para posicionarnos en nuestra nueva rama)



Concepto: eliminar una rama

si ya no me interesa una rama (y todos sus commits)

git branch -D prueba2016

concepto: crear una rama y movernos a ella

`git checkout -b experimento`

Merge

Comando merge

Antes hemos creado una rama llamada test, hemos hecho las pruebas y vemos que han ido bien.

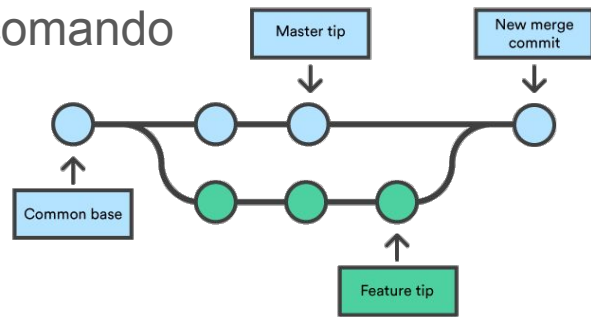
Ahora lo que hacemos es en la unión de las 2 ramas.

Primero nos situamos en la rama principal y usamos el comando

`git checkout master`

`git merge test`

Si ahora hacemos un **`git log`** ya veremos los cambios de test en master.

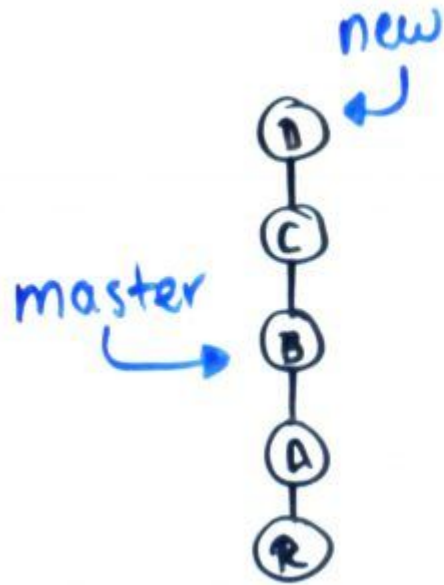


Tipos de fusiones en merge

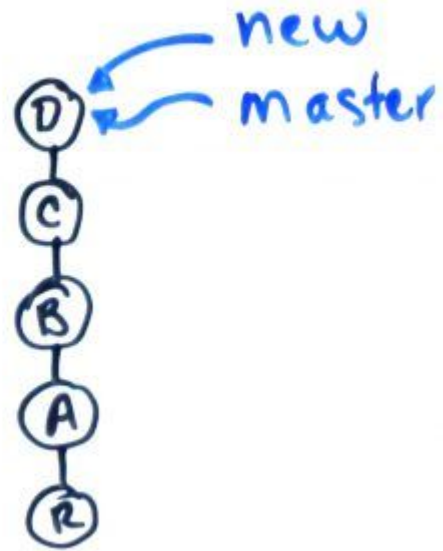
Al hacer el merge nos pueden salir 2 tipos de mensajes

- **Fast-Forward**: cuando en el proceso **no** tengo que intervenir yo
- **auto-merge por estrategia recursiva**: tampoco tengo que intervenir
- **Manual merge conflicto**: cuando en el proceso de mezclado tengo que intervenir para solucionar algo (tomando alguna decisión)

Ejercicio (fast-forward)



FF
MERGE
⇒



Ejercicio: Añade un fichero a otra rama

Vamos a crear un fast-forward sin conflictos

creamos un nuevo archivo

code prueba.txt

git branch nueva-rama

git branch

git checkout nueva-rama

git branch

git status -s

git add .

git commit -m "añadido fichero prueba.txt"

git log --oneline

Podemos resumir estos 2 pasos en:
git checkout -b nueva-rama

Ejercicio: Fusiona la rama anterior a la rama master

git log --oneline

```
e22746e (HEAD -> nueva-rama) add fichero  
7d00bc8 (master) nu
```

git checkout master

git merge nueva-rama

Esto no ha producido ningún tipo de conflicto, hemos hecho un fast-forward

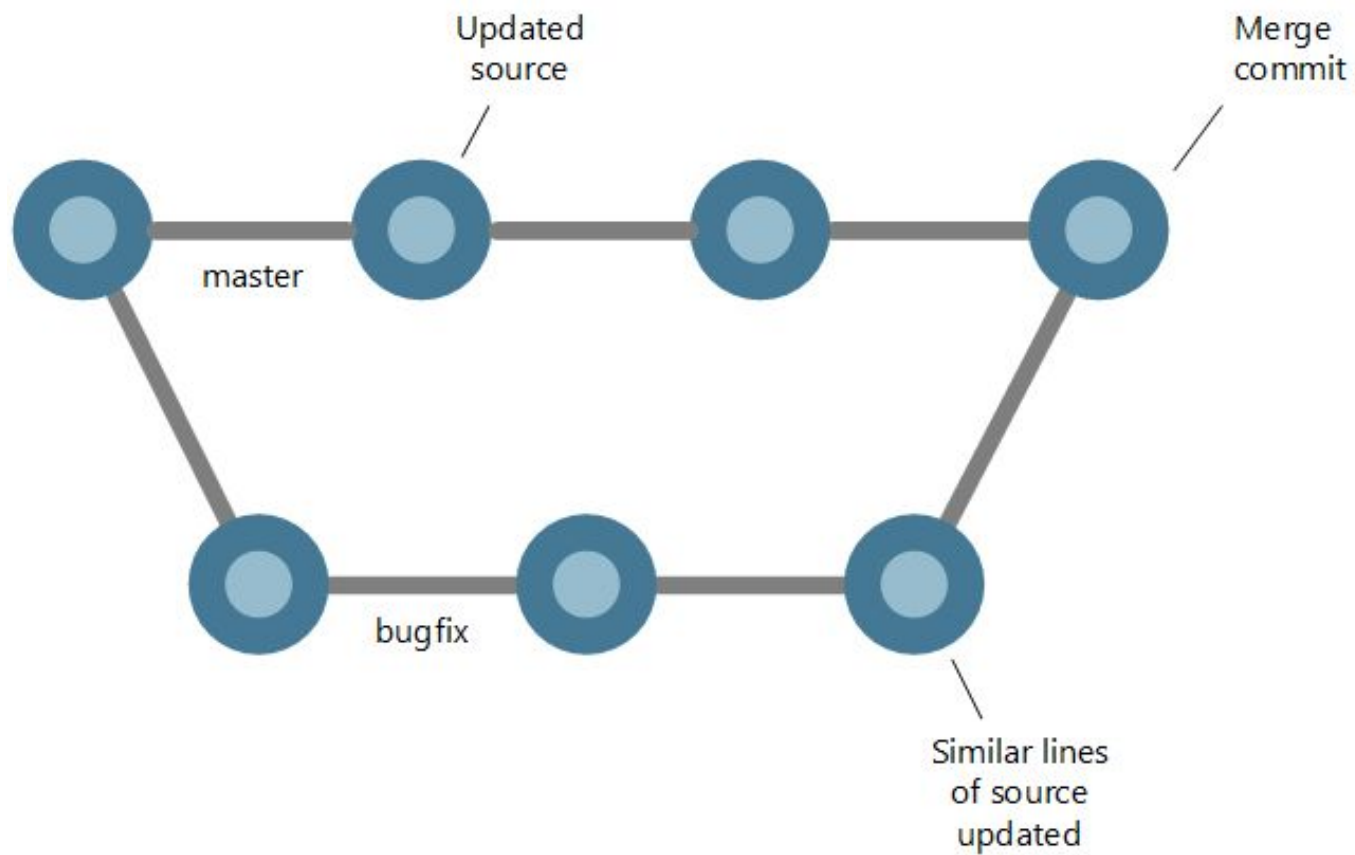
git log --oneline

```
e22746e (HEAD -> master, nueva-rama) add fichero  
7d00bc8 nu
```

Como no necesitamos la rama (nueva-rama) la podemos borrar

git branch -d nueva-rama

Ejercicio (auto-merge)



Ejercicio

crea un fichero llamado index.html

con el contenido básico de una página web

code index.html

añadelo al repositorio

git add .

git commit -m "creando pagina web"

Creamos una rama nueva llamada rama-prueba

git checkout -b rama-prueba

Editamos el fichero index.html y le ponemos un título al html

code index.html

git add .

git commit -m "he puesto un titulo a la pagina web"

git log --oneline --decorate --all --graph

Volvemos a la rama master y también realizamos un cambio en el index.html en esta ocasión añadiendo un párrafo al body

git checkout master

code index.html

añadimos los cambios al repositorio

git add .

git commit -m "he puesto un parrafo en el body"

git log --oneline --decorate --all --graph

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<meta http-equiv="X-UA-Compatible" content="ie=edge">

<title>Document</title>
</head>
<body>

</body>
</html>
```

```
* af7046f (HEAD -> master) he puesto un parrafo al body
| * 4c476ac (rama-prueba) puse un titulo a la web
|/
* 615bbdc creando pagina web
```

Ángel González M.

Ejercicio: Fusionamos los cambios

Vamos a fusionar los cambios. Recuerda:

- en el master hemos añadido un párrafo al body
- en la rama rama-prueba hemos cambiado el título

`git merge rama-prueba`

`git log --oneline --decorate --all --graph`

```
Auto-merging index.html
Merge made by the 'recursive' strategy.
index.html | 2 +-
1 file changed, 1 insertion(+), 1 deletion(-)
```

```
* 7fd4c6d (HEAD -> master) Merge branch 'rama-prueba'
| \
| * 2f392cc (rama-prueba2) add titulo
* | 2410b26 add parrafo al body
|/
* 79556c6 creo pagina web
```

Ejercicio (manual-merge) conflicto

Ejercicio

crea un fichero llamado index.html

con el contenido básico de una página web

code index.html

añadelo al repositorio

git add .

git commit -m "creando pagina web"

Creamos una rama nueva llamada rama-prueba

git checkout -b rama-conflicto

Editamos el fichero index.html y le ponemos un título al html **<title>TituloA</title>**

code index.html

git add .

git commit -m "pongo tituloA"

git log --online --decorate --all --graph

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<meta http-equiv="X-UA-Compatible" content="ie=edge">

<title>Document</title>
</head>
<body>

</body>
</html>
```

Volvemos a la rama master y también realizamos un cambio en el index.html le ponemos un título distinto **<title>TituloB</title>**

git checkout master

code index.html

añadimos los cambios al repositorio

git add .

git commit -m "pongo tituloB"

git log --online --decorate --all --graph

```
* cb1faf0 (HEAD -> master) pongo tituloB
| * 0d7ef6f (rama-conflicto) pongo tituloA
|/
* b4ba1f7 creo pagina web
```

Ángel González M.

Ejercicio

git merge rama-conflicto

CONFLICTO

```
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

Vamos a resolver el conflicto

code index.html

git status

git add .

git commit -m "resolviendo conflicto acepto tituloA"

git status

git log --oneline --decorate --all --graph

Fijate en el color rojo (cambio rechazado) y color


verde (cambio aceptado)

Ya podemos borrar la rama (si queremos)

git branch -d rama-conflicto

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="">
<meta http-equiv="X-UA-Compatible" content="ie=edge">
<<<<<<< HEAD
<title>TituloB</title>
=====
<title>TituloA</title>
>>>>>>> rama-conflicto
</head>
<body>

</body>
</html>
```



```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="">
<meta http-equiv="X-UA-Compatible" content="ie=edge">
<title>TituloA</title>
</head>
<body>

</body>
</html>
```

```
* ccc41db (HEAD -> master) resolviendo conflicto acepto tituloA
| \
| * 0d7ef6f (rama-conflicto) pongo tituloA
| * | cb1faf0 pongo tituloB
| /
* b4ba1f7 creo pagina web
```

Alias

Alias

Me permiten crear alias de comandos complejos

```
git config --global alias.superlog "log --oneline --decorate --all --graph"
```

Donde:

- **superlog**: será el nombre del alias
- **log --oneline --decorate --all --graph**: será el comando a ejecutar

Ahora ejecutamos el alias

```
git superlog
```

Podemos ver el cambio en:

```
git config --global -e  
git config --global -l
```

Ejercicio

Crea un alias del comando `git status -s -b`

```
git config --global alias.s "status -s -b"
```

```
git s
```

Ejercicio

Crea un alias del comando git add y commit juntos

```
git config --global alias.ac "!git add . && git commit"
```

```
git ac -m "mensaje"
```


Tags

Concepto de Tags

Sirven para etiquetar commits

Por ejemplo a un commit importante le puedo poner el tag de Version2.0

Tags (anotaciones)

Los tags son anotaciones asociadas a un commit. Por ejemplo para especificar una release (versión) específica.

Tag anotadas: contienen información extra

El siguiente comando añade un tag completo en el commit en el que me encuentro

```
git tag -a v1.0 -m "Mensaje"
```

Tag ligera: añade un simple tag

```
git tag -a v1.0
```

Para ver los tag aplicados

git tag

Para ver la información de un tag específico:

git show <nombre_tag>

Ejercicio: aplicando un tag

Crea un commit

git commit -m "añadi algo"

Añadele un tag a ese commit indicando que es la version 2.0

git tag -a v2.0

Muestra los tags aplicados

git tag

Muestra la información de ese tag en particular

git show v2.0

Borra el tag

git tag -d v2.0

Añadiendo tag a commits específicos

```
git tag -a v0.3 -m "version 0.3" 1234567890abcdef
```

donde *1234567890abcdef* sería el código sha del commit donde añadir el tag

Para borrar un tag

```
git tag -d <nombre_Tag>
```

Cambiar el nombre a un commit

```
git commit --amend -m "Nuevo nombre"
```


RESET

Viajes en el tiempo

git reset --soft HEAD^

Incorporar los últimos cambios al último commit sin crear un nuevo commit.

Imagina que creas un commit con el fichero Fichero1.txt

code Fichero1.txt

git add Fichero1.txt

git commit -m "add Fichero1.txt"

Ahora te das cuenta de que también tienes que añadir Fichero2.txt, pero no quieres crear un nuevo commit, sino que quieres que este Fichero2.txt se incorpore al commit que acabas de crear.

git reset --soft HEAD^

code Fichero2.txt

git add Fichero2.txt

git commit -m "add Fichero1 y 2.txt"

Diferencias entre reset soft, mixed y hard

Soft =>

- "elimina" los commits posteriores al commit al que estás haciendo el reset
- conserva los cambios en el stage area
- conserva los cambios que tengas en tus archivos (working directory)

Mixed =>

- "elimina" los commits posteriores al commit al que estas haciendo el reset
- Deshace los cambios en el stage area
- conserva los cambios que tengas en tus archivos (working directory)

Hard=>

- "elimina" los commits posteriores al commit al que estas haciendo el reset
- Deshace los cambios en el stage area
- Deshace los cambios que tengas en tus archivos (working directory)

Ángel González M.

comando: reset

Este comando es parecido a checkout, con la salvedad de que elimina los commits.

Hay varias variantes:

--Soft realiza el reset pero no toca la zona de working area

git reset --soft

--mixed borra el staging area sin tocar el working area

git reset --mixed

--hard borra todo el working area, el staging area y el repositorio

git reset --hard

Ejercicio

Crea un fichero llamado dato1.txt y añadelo al repositorio con un commit

Crea un fichero llamado dato2.txt y añadelo al repositorio con un commit

Crea un fichero llamado dato3.txt y añadelo al repositorio con un commit

Crea un fichero llamado dato4.txt y añadelo al repositorio con un commit

Crea un fichero llamado dato5.txt y añadelo al repositorio con un commit

Modifica el fichero dato1.txt y vuelve a añadirlo al repositorio con un commit

Modifica el fichero dato5.txt (pongo algún texto nuevo) y vuelve a añadirlo al repositorio con un commit

Ángel González M.

Vamos a resetear (soft) los cambios

```
git log --oneline
```

```
git reset --soft HEAD^
```

NOTA: También podemos usar el lugar de HEAD^ el código del commit `git log`

```
git log --oneline
```

editamos de nuevo el fichero dato5.txt y vemos que no se han borrado los cambios que teníamos. Además dato5.txt está en el staging area

Volvemos a incorporar con un commit los cambios

```
git add .
```

```
git commit -m "cambios en dato5.txt"
```

Vamos a resetear (hard) los cambios

```
git log --oneline
```

```
git reset --hard HEAD~1
```

NOTA: También podemos usar el lugar de HEAD^ el código del commit `git log`

```
git log --oneline
```

editamos de nuevo el fichero dato5.txt y vemos que si se han perdido todos los cambios.

Lo modificamos otra vez, añadiéndole algo de contenido

Volvemos a incorporar el cambio y con un commit guardamos en el repositorio

```
git add .
```

```
git commit -m "cambios en dato5.txt"
```

Vamos a resetear (mixed) los cambios

```
git log --oneline
```

```
git reset --mixed HEAD^
```

NOTA: También podemos usar el lugar de HEAD^ el código del commit `git log`

```
git log --oneline
```

editamos de nuevo el fichero dato5.txt y vemos que no se han perdido los cambios. Además el fichero dato5.txt se ha quitado del staging area.

Volvemos a incorporar el cambio y con un commit guardamos en el repositorio

```
git add .
```

```
git commit -m "cambios en dato5.txt"
```


Ejercicio

Usando un reset (hard) ponde en el commit en el que añadiste el dato2.txt

```
git log --oneline
```

```
git reset --hard ed94690
```

```
git log --oneline
```

```
dir
```

Hemos perdido todos los cambios que hemos realizado desde el añadido el dato2.txt

Aun así podemos recuperar la información borrada ya que git siempre mantiene un registro de todos los cambios realizado.

```
git reflog
```

```
git reset --hard 706ac5c
```

```
dir
```

```
ed94690 (HEAD -> master) HEAD@{0}: reset: moving to ed94690
e7b14be HEAD@{1}: reset: moving to HEAD^
706ac5c HEAD@{2}: commit: cambio dato5.txt por cuarta vez
e7b14be HEAD@{3}: reset: moving to HEAD^
363423d HEAD@{4}: commit: cambio dato5.txt por tercera vez
e7b14be HEAD@{5}: reset: moving to HEAD^
4cabd38 HEAD@{6}: commit: cambio dato5.txt por segunda vez
e7b14be HEAD@{7}: reset: moving to HEAD^
4d4c991 HEAD@{8}: commit: cambio dato5.txt
e7b14be HEAD@{9}: commit: cambio dato1.txt
6732800 HEAD@{10}: commit: add dato5.txt
852d637 HEAD@{11}: commit: add dato4.txt
39510a2 HEAD@{12}: commit: add dato3.txt
ed94690 (HEAD -> master) HEAD@{13}: commit: add dato2.txt
c1d2d4e HEAD@{14}: commit (initial): add dato1.txt
```

Ángel González M.

Cambiar el nombre de un archivo

Podemos hacerlo de 2 formas

Cambiamos el nombre del fichero con el sistema operativo

code cambiarNombre2.txt

git add .

git commit -m "nuevo archivo"

git status -s

[linux/mac] `mv` cambiarNombre2.txt cambiado2.txt

[windows] `rename` cambiarNombre2.txt cambiado2.txt

git status -s

```
D  cambiarNombre2.txt
?? cambiado2.txt
```

Cambiamos el nombre del fichero con git, esto le notifica directamente a git que el fichero ha cambiado.

code cambiarNombre.txt

git add .

git commit -m "nuevo archivo"

git status -s

`git mv` cambiarNombre.txt cambiado.txt

git status -s

```
R  cambiarNombre.txt -> cambiado.txt
```

Ángel González M.

Eliminar un archivo

Podemos hacerlo de 2 formas:

Eliminamos el fichero con el sistema operativo

code eliminate.txt

git add .

git commit -m "nuevo archivo"

git status -s

[linux/mac] **rm** eliminate.txt

[windows] **del** eliminate.txt

git status -s

```
D eliminate.txt
```

Eliminamos el fichero con git, esto le notifica directamente a git que el fichero ha cambiado.

code eliminate2.txt

git add .

git commit -m "nuevo archivo"

git status -s

git rm eliminate2.txt

git status -s

```
D eliminate2.txt
```

Ángel González M.

Stash

Stash

Según se está trabajando en un apartado de un proyecto, normalmente el espacio de trabajo suele estar en un estado inconsistente. Pero puede que se necesite cambiar de rama durante un breve tiempo para ponerse a trabajar en algún otro tema urgente. Esto plantea el problema de confirmar cambios en un trabajo medio hecho, simplemente para poder volver a ese punto más tarde. Y su solución es el comando 'git stash'.

Para almacenar el trabajo actual en el stash

git stash tambien sirve: **git stash save**

Para ver los stash creados actualmente

git stash list

Para recuperar el trabajo y volver a la rama principal

git stash pop

NOTA: El uso de stash puede o no dar conflictos que hay que solucionar

Ejercicio: trabajando con el stash (sin conflictos)

Crea una página web en un directorio con el html básico y comitealo

Añade un h1 y un párrafo a la web y vuelve a comitear

Ahora comienza a realizar más cambios en la web, por ejemplo añade una foto de un gatito en algún sitio de la página web, pero no comitees los cambios pues aun no has terminado de trabajar

Llega nuestro jefe de proyecto y nos pide de repente: ¿cómo quedaría la web con el fondo azul?

Pero... no podemos mostrárselo puesto que estoy con el trabajo a medio hacer poniendo fotos de gatitos...

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport"
content="width=device-width,
initial-scale=1.0">
  <meta
http-equiv="X-UA-Compatible"
content="ie=edge">
  <title>Document</title>
</head>
<body>
  <h1>Lista de frutas</h1>
  <ul>
    <li>manzana</li>
    <li>pera</li>
  </ul>
</body>
</html>
```

Solución: usando el stash

Ejercicio continuación

Vamos a guardar los cambios en el stash

git stash

Si vemos el fichero index.html no tenemos el código de la imagen de los gatitos

Veamos los stash que hemos creado:

git stash list

Veamos el log:

git log

git log --all

```
* b9df1af (refs/stash) WIP on master: 794f8da add cabecera y parrafo
| \
| * 551bf47 index on master: 794f8da add cabecera y parrafo
| /
* 794f8da (HEAD -> master) add cabecera y parrafo
* 87017e2 pagina web vacia
```

git log --oneline --all --graph

Ángel González M.

Ejercicio continuación

Vamos a hacer los cambios que nos pide nuestro jefe. Editamos el index.html y ponemos su fondo azul.

`<body style="background-color:lightblue">`. Y comiteamos los cambios

git add .

git commit -m "pagina con fondo azul"

Listo el jefe ya está contento... pero... ¿y mis gatitos?

Ahora volvemos a incorporar los cambios con los que estaba trabajando poniendo fotos de gatitos (antes de ser interrumpido por mi jefe)

git stash pop

git log --oneline --all --graph

```
* aff73a3 (HEAD -> master) pagina con fondo azul
* 794f8da add cabecera y parrafo
* 87017e2 pagina web vacia
```

git status

Ya podemos seguir trabajando, mi pagina tiene fondo azul, y vuelven a estar las fotos de los gatitos

Ángel González M.

Ejercicio stash con conflictos

Ejercicio: trabajando con el stash (sin conflictos)

Crea un página web en un directorio con el siguiente html y comítéalo

Ahora comienza a realizar más cambios en la web:

- Cambia el título de la página web por FRUTAS
- cambia los elementos de la lista coche y moto, por pera y manzana

pero no comitees los cambios pues aún no has terminado de trabajar

Llega nuestro jefe de proyecto y nos pide de repente: cambia la página web para que tenga en la lista los elementos mango, coco y piña.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" >
  <meta http-equiv="X-UA-Compatible">
  <title>Comidas</title>
</head>
<body>
  <h1>Lista de frutas</h1>
  <ul>
    <li>coche</li>
    <li>moto</li>
  </ul>
</body>
</html>
```

Ejercicio continuación

Vamos a guardar los cambios en el stash

git stash

Si vemos el fichero index.html no tenemos el incorporado por nosotros (titulo y la lista con pera y manzana)

Veamos los stash que hemos creado:

git stash list

Veamos el log:

git log --oneline --all --graph

Realizamos los cambios que pide el jefe, ponemos en la lista solo los elementos mango, coco y piña

code index.html

git add .

git commit -m "añadido mango coco y pinna pedidos por el jefe"

Ángel González M.

Ejercicio continuación

`git log --oneline --all --graph`

Genial el jefe ya ha quedado contento, ahora volvemos a incorporar los cambios con los que estaba trabajando poniendo fotos de gatitos (antes de ser interrumpido por mi jefe)

Pero... vamos a tener problemas... hay conflictos

`git stash pop`

```
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
```

No pasa nada, tenemos que solucionar el conflicto como ya sabemos, abrimos el fichero con un editor y decidimos con qué cambios nos vamos a quedar.

Date cuenta que el título de la web no ha dado conflicto, sólo ha dado conflicto la lista.

```
<body>
  <h1>Lista de frutas</h1>
  <ul>
    <li>mango</li>
    <li>coco</li>
    <li>piña</li>
  </ul>
</body>
```

Accept Current Change | Accept Incoming Change | Accept Both

<<<<<< Updated upstream (Current Change)

```
<li>mango</li>
<li>coco</li>
<li>piña</li>
```

=====

```
<li>pera</li>
<li>manzana</li>
```

>>>>>> Stashed changes (Incoming Change)

```
</ul>
</body>
```

Ángel González M.

Ejercicio continuación

Decidimos elegir los cambios propuestos por el jefe (mango, coco y piña)

Por último comiteo los cambios

git add .

git commit -m "incorporo mis cambios juntos con los del jefe"

git log --oneline --all --graph

Podemos deshacernos de la rama del stash que ha quedado colgando

git stash list

git stash drop

```
* e928e67 (HEAD -> master) incorporo mis cambios juntos con los del jefe
* 6399b7d añadido mango coco y pinna pedidos por el jefe
| * 2074cf1 (refs/stash) WIP on master: 2a3ce17 cambios del jefe
| | \
| | /
| / /
| * 2d7f8bd index on master: 2a3ce17 cambios del jefe
| /
* 2a3ce17 cambios del jefe
* 0f63bd6 add uvas y limones
* 5b9ce62 subi pagina web con lista frutos ul
```

Stash: Apply

Podemos tener múltiples stash creados

```
git stash list
```

```
stash@{0}: WIP on master: 123456A
```

```
stash@{1}: WIP on master: 123456B
```

```
stash@{2}: WIP on master: 123456C
```

Para restaurar una entrada determinada del stash usaremos

```
git stash apply stash@{1}
```

Resumen stage

<code>git stash == git stash save</code>	<code># guarda el estado en el stage</code>
<code>git stash apply == git stash apply stash@{0}</code>	<code># recupera el estado del stage</code>
<code>git stash drop == git stash drop stash@{0}</code>	<code># borra un stage</code>
<code>git stash pop == git stash apply + git stash drop</code>	<code># recupera y borra el estado</code>

Stash: keep-index

Podemos guardar en el stash todo menos que esté en el staging area (index) con el comando:

```
git stash save --keep-index
```

Stash: include-untracked

Podemos guardar en el stash todo incluido a los que git no le da seguimiento (working area) con el comando:

git stash save --include-untracked

Stash: show

Si queremos información más completa de los cambios aplicados al stash usamos:

git show stash

Stash: save con mensaje

Cuando guardamos un stash podemos ponerle un mensaje aclarativo:

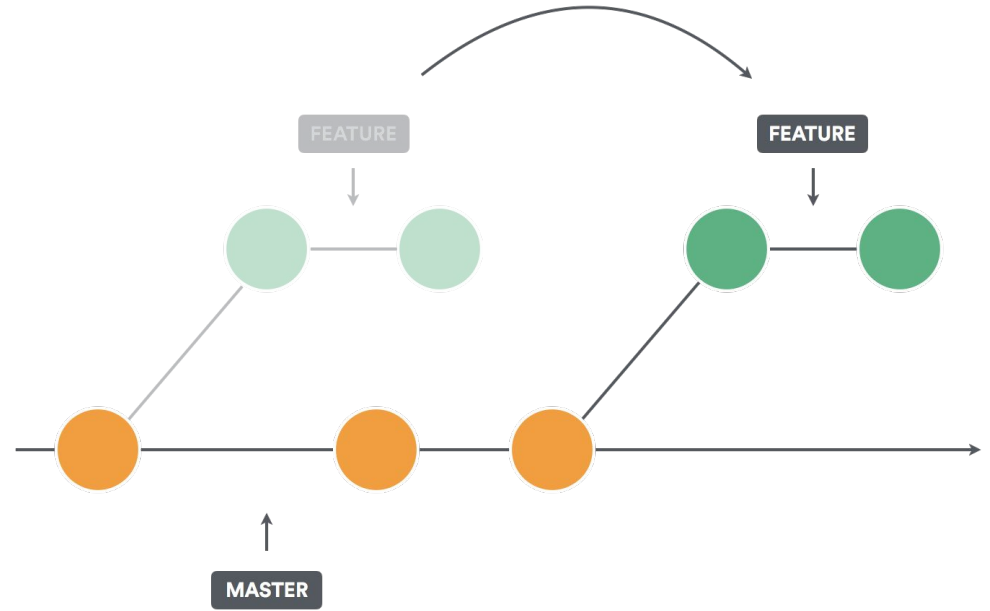
git stash save "este es mi mensaje"

Stash: clear

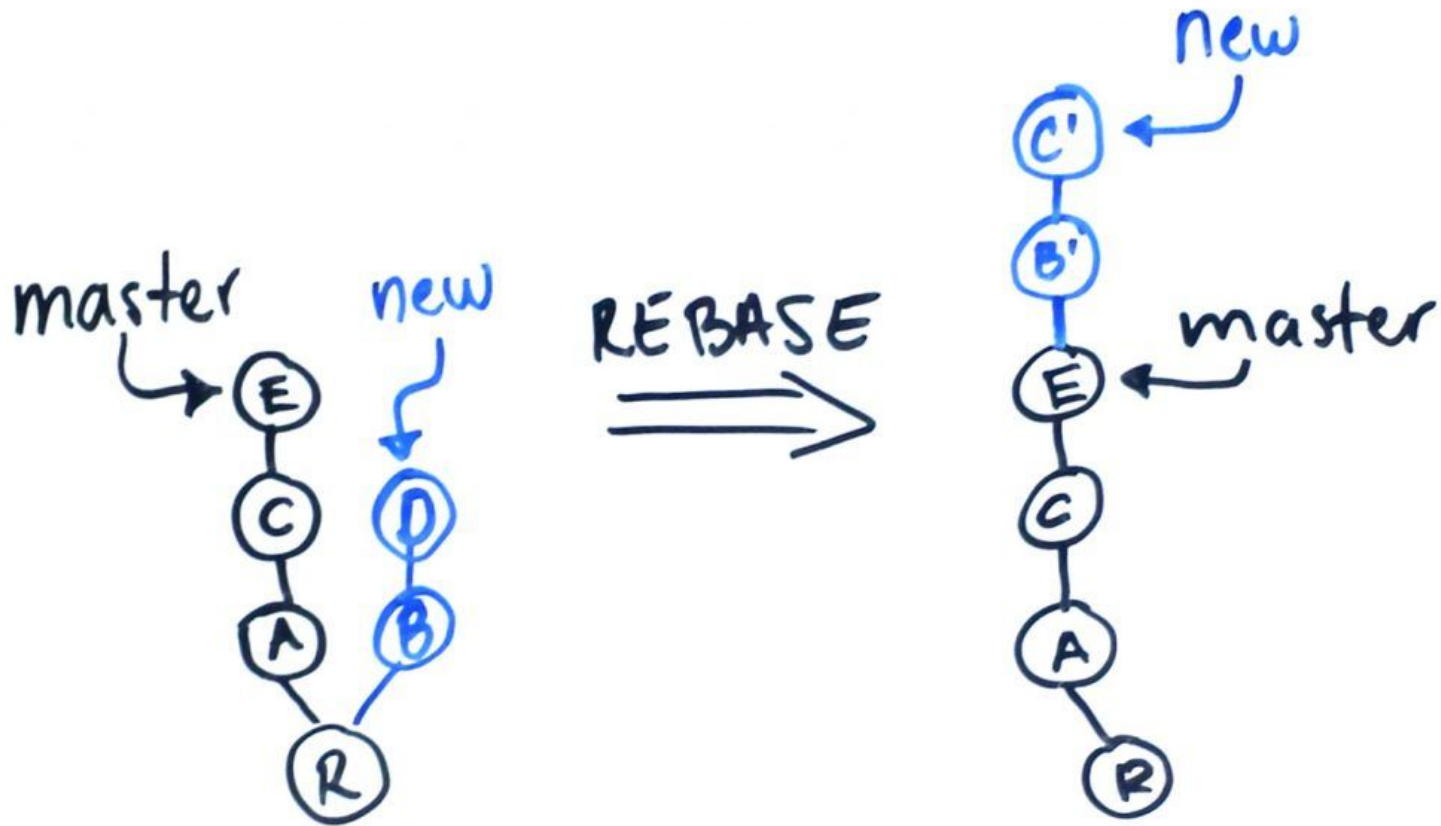
Borra todas las entradas de stash.

git stash clear

Rebase



Ángel González M.



Ejercicio

Hagamos el siguiente ejercicio:

- Crea una carpeta con un proyecto git: **git init**
- Crea un esqueleto básico de una página web (index.html) y comitea los cambios
- Crea una rama llamada **secundaria** y colócate en ella: **git checkout -b secundaria**
- Añade una foto de un unicornio y comitea los cambios
- Añade un tittle (texto footer) a la web y comitea los cambios
- Colócate en la rama master: **git checkout master**
- Añade un h1 y comitea los cambios
- Añade un párrafo (debajo del h1) y comitea los cambios

Al final la estructura de tus comits/ramas debería quedar como:

```
* ff58441 (HEAD -> master) pongo un parrafo
* 89c636e pongo un h1
| * c0637a4 (secundaria) pongo footer
| * d569744 foto unicornio
|/
* af8aeafa esqueleto web
```

Ahora vamos a realizar un rebase:

```
git log --oneline --all --graph
```

git checkout secundaria

```
* af9fccf (HEAD -> secundaria) pongo footer
* c37aeff foto unicornio
* ff58441 (master) pongo un parrafo
* 89c636e pongo un h1
* af8aeafa esqueleto web
```

git rebase master

git log --oneline --all --graph

Si nos da un conflicto podemos solucionarlo **code index.html** aceptando ambos cambios **git add index.html** y realizar un **git rebase --continue**

Ángel González M.

Ejercicio continuación

Ahora podemos fusionar las 2 ramas (master y secundaria)

git checkout master

git merge secundaria

git log --oneline --all --graph

```
* af9fccf (HEAD -> master, secundaria) pongo footer
* c37aeff foto unicornio
* ff58441 pongo un parrafo
* 89c636e pongo un h1
* af8aefa esqueleto web
```

Ya no necesitamos la rama secundaria y la podemos borrar

git branch -d secundaria

Rebase: interactivo (unir commits)

Rebase para unir commits

Cuando trabajas en un proyecto con muchas ramas y mucha gente puede que generes muchos commits, puedes compactar varios commits en uno usando:

```
git rebase -i HEAD~8
```

// unir todos los commits no en master

```
git rebase -i master
```

// unir los ultimos 2 commits

```
git rebase -i HEAD~2
```

// une todos los commits desde el seleccionado (3hru3hu3h9r33oio)

```
git rebase -i 3hru3hu3h9r33oio
```

Ángel González M.

Ejercicio

Hagamos el siguiente ejercicio:

- Crea una carpeta con un proyecto git: **git init**
- Crea un esqueleto básico de una página web y comitea los cambios
- Añade una foto de un perro y comitea los cambios
- Añade una foto de un gato y comitea los cambios
- **git log --online**

```
* 8bd734f (HEAD -> master) add foto gato
* 4acdc37 foto perro
* 0599207 esqueleto web
```

Nos damos cuenta que los 2 commits donde añadimos fotos podrían haberse realizado en un solo commit. Queremos fusionar esos 2 commits en uno solo.

git rebase -i HEAD~2

Nos aparecerá una ventana con el listado de commits elegidos. Hay que cambiar la palabra pick(p) por squash(s) en el commit del gato. Y guardamos el documento.

```
pick 4acdc37 foto perro
s 8bd734f add foto gato

# Rebase 0599207..8bd734f onto 0599207 (2 commands)
#
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
# x, exec = run command (the rest of the line) using shell
# d, drop = remove commit
#
```

A continuación ponemos el texto del commit unificado. `inserte foto de perro y de gato`

ya tenemos los 2 commit unificados **git log --online**

```
* 5ce5d3f (HEAD -> master) inserte foto de perro y de gato
* 0599207 esqueleto web
```


Ejercicio

Crea la siguiente estructura de commits en un fichero index.html

Nº	Hash	Mensaje del commit
1	111111	Add fichero7
2	22222	Add fichero6
3	33333	Add fichero5
4	44444	Add fichero4
5	55555	Add fichero3
6	66666	Add fichero2
7	77777	Add fichero1

```
* 05c0ed5 (HEAD -> master) add fichero7
* 35793c9 add fichero6
* 282d6b9 add fichero5
* 0f76508 add fichero4
* 531e7d2 add fichero3
* d191ac1 add fichero2
* e61e592 add fichero1
```



Falla,
corregir

Con los siguientes comandos borraremos los commits fichero6(22222) y fichero4(44444) de nuestro repositorio:

Ejercicio

Checkout al commit justo anterior al que queremos borrar (en este ejemplo, al 5(55555) fichero3).

```
git checkout 55555
```

Creamos una nueva rama (llámala fix) y nos posicionamos en ella.

```
git checkout -b fix
```

Añadimos todos los comits excepto los que queremos borrar fichero6 (22222). Añadimos con cherry-pick el fichero5(33333) y el fichero7(11111).

```
git cherry-pick fichero5 (33333)
```

```
git cherry-pick fichero7 (11111)
```

Checkout al master.

```
git checkout master
```

Reset en la branch master al commit justo anterior al que queríamos borrar (en este ejemplo, al fichero3 (55555)).

```
git reset --hard 55555
```

Hacemos merge de nuestra rama con la master.

```
git merge fix
```

```
* 21f8570 (HEAD -> master) Merge branch 'fix'
| \
|  * 1505f04 (fix) add fichero7
|  * 6acea48 add fichero5
|  * 282d6b9 add fichero5
|  * 0f76508 add fichero4
| /
* 531e7d2 add fichero3
* d191ac1 add fichero2
* e61e592 add fichero1
```

```
* 05c0ed5 (master) add fichero7
* 35793c9 add fichero6
* 282d6b9 add fichero5
* 0f76508 add fichero4
* 531e7d2 (HEAD -> master) add fichero3
* d191ac1 add fichero2
* e61e592 add fichero1
```

Falla,
corregir

```
* 05c0ed5 (master) add fichero7
* 35793c9 add fichero6
* 282d6b9 add fichero5
* 0f76508 add fichero4
* 531e7d2 (HEAD -> fix) add fichero3
* d191ac1 add fichero2
* e61e592 add fichero1
```

```
* 1505f04 (HEAD -> fix) add fichero7
* 6acea48 add fichero5
| * 05c0ed5 (master) add fichero7
| * 35793c9 add fichero6
| * 282d6b9 add fichero5
| * 0f76508 add fichero4
| /
* 531e7d2 add fichero3
* d191ac1 add fichero2
* e61e592 add fichero1
```

```
* 1505f04 (fix) add fichero7
* 6acea48 add fichero5
| * 282d6b9 (HEAD -> master) add fichero5
| * 0f76508 add fichero4
| /
* 531e7d2 add fichero3
* d191ac1 add fichero2
* e61e592 add fichero1
```

Ángel González M.

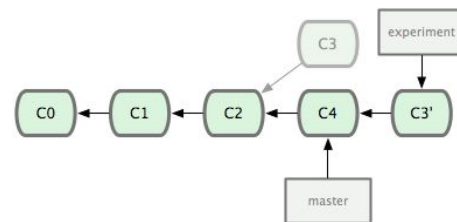
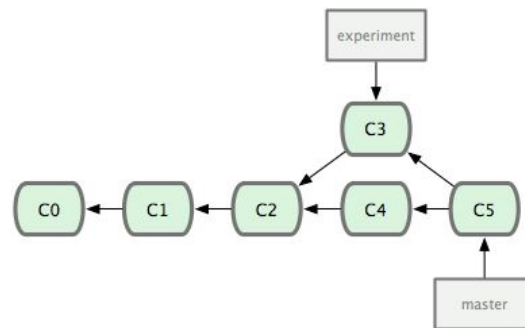
Rebase VS Merge

Rebase VS Merge

git merge realiza una fusión a tres bandas entre las dos últimas instantáneas de cada rama (C3 y C4) y el ancestro común a ambas (C2); creando una nueva instantánea (snapshot) y la correspondiente confirmación (commit).

No hay ninguna diferencia en el resultado final de la integración, pero el haberla hecho reorganizando nos deja un registro más claro. Si examinas el registro de una rama reorganizada, este aparece siempre como un registro lineal: como si todo el trabajo se hubiera realizado en series, aunque realmente se haya hecho en paralelo.

Habitualmente, optarás por esta vía cuando quieras estar seguro de que tus confirmaciones de cambio (commits) se pueden aplicar limpiamente sobre una rama remota; posiblemente, en un proyecto donde estés intentando colaborar, pero lleves tu el mantenimiento. En casos como esos, puedes trabajar sobre una rama y luego reorganizar lo realizado en la rama origin/master cuando lo tengas todo listo para enviarlo al proyecto principal. De esta forma, la persona que mantiene el proyecto no necesitará hacer ninguna integración con tu trabajo; le bastará con un avance rápido o una incorporación limpia.



Diferencias entre rebase y merge

REBASE

El rebase unifica las ramas dejando un árbol lineal o más bonito.

El rebase unifica sin necesidad de crear un nuevo commit .

El rebase puedes repetir este proceso otra vez cuando hay nuevos cambios en la otra rama: siempre terminarás con una serie de cambios limpios en la cabeza de la rama

El rebase unifica las ramas perdiendo el historial de los commit y el merge no . Esto puede resultar importante cuando se necesite llevar o saber el historial de commit

MERGE

El merge mantiene el gráfico de las ramas.

El merge a la hora de querer unificar nos toca realizar un commit de más (muchos dicen commit basura ó innecesario).

Si haces varios merge's empezarás a crear una serie de historiales intercalados... lo que puede ser lioso

Repositorios remotos

concepto: repositorio

Un repositorio es un Almacén o lugar donde se guardan ciertas cosas.

Tenemos 2 tipos de repositorios

- **Local:** por ejemplo el que tenemos en nuestro ordenador
- **Remoto:** por ejemplo los que están en github, gitlab, gogs o bitbucket (entre otros)

Importante GIT y GITHUB son cosas diferentes

comando: clone

Nos bajamos un proyecto remoto en nuestro ordenador, esto creará una carpeta con el nombre del repositorio remoto y clonará su contenido en su interior.

```
git clone https://AngelGonzalezM@bitbucket.org/AngelGonzalezM/prueba.git
```

También podemos clonar nuestro proyecto en una carpeta en particular dentro de nuestro ordenador

```
git clone https://AngelGonzalezM@bitbucket.org/AngelGonzalezM/prueba.git <nombre_carpeta>
```

Ejercicio: clona un proyecto de github

Clona el proyecto que guarda el código fuente del juego DOOM escrito por John Carmack

```
git clone https://github.com/id-Software/DOOM.git
```

Comando remote

Este comando vincula nuestro proyecto local con el repositorio remoto.

Básicamente, le decimos que nuestro repositorio local es el mismo que el remoto.

```
git remote add origin https://AngelGonzalezM@bitbucket.org/AngelGonzalezM/prueba.git
```

Con este comando nos permite comprobar los remotos que tengo configurados

```
git remote -v
```

Con este comando podemos eliminar conexiones de tipo remote

```
git remote remove origin
```

Podemos tener varios remotos en un mismo repositorio

Comando push

Este comando manda los datos del repositorio local a repositorio remoto.

Es decir si mi remoto está asociado a la web de GitHub, este comando subiría todos los datos almacenados en mi repositorio local a GitHub.

git push origin master

(donde master es la rama que vamos a subir)

Que es el repositorio remoto GitHub

Es una plataforma de desarrollo colaborativo de software para alojar proyectos.

Ofrece:

- Repositorios públicos ilimitados
- Creación de páginas web (html, css, javascript)
- Permite lanzar push, pull, clonados ilimitados
- Wikis, Issues, estadísticas, gestion proyectos
- Creación de organizaciones
- Colaboración con colaboradores

Ejercicio: subamos un repositorio a github

Crea una carpeta e inicia un proyecto git en ella

Haz 2 commits de 2 archivos (archivo1.txt y archivo2.txt)

```
echo " " > archivo1.txt
```

```
git add archivo1.txt
```

```
git commit -m "subo archivo1"
```

```
echo " " > archivo2.txt
```

```
git add archivo2.txt
```

```
git commit -m "subo archivo2"
```

```
git log --oneline --all --decorate --graph
```

Ahora crea un repositorio remoto por ejemplo en GitHub llamado prueba

Conecta tu repositorio local con el remoto

```
git remote add origin https://github.com/kant003/prueba2.git
```

```
git remote -v
```

```
git branch -a
```

Ahora subimos los cambios guardados en local al el repositorio remoto

```
git push origin master
```

```
git branch -a
```

```
* bc6f7c8 (HEAD -> master, origin/master) subo archivo2
* 812e898 subo archivo1
```

Ángel González M.

Issues

Sirven para comunicar un problema, una necesidad de cambio, mejora, funcionalidad o una idea.

En github se pueden añadir issues con formato

Milestones

Son grupos de issues que se aplican al proyecto

Pueden tener una fecha de finalización

Por tanto los issues se pueden colocar dentro de un milestone.

Labels

Los labels son etiquetas que podemos usar para etiquetar los issues.

subir a remoto un tag o varios

Con el siguiente comando subimos un tag específico a remoto

git push origin v1.0

También podemos subir de un golpe todos los tags con el comando

git push origin --tags

Flujos de trabajo

Tipos de flujos de trabajo

- Flujo de trabajo manager (basado en forks)
- Flujo de trabajo centralizado
 - Flujo de trabajo centralizado en la rama master
 - Flujo de trabajo centralizado con gestor de integraciones
- Flujo de trabajo con Dictador y Tenientes

Flujo de trabajo manager (basado en forks)

Existe un repositorio A en el cual solo una persona (llamémosle mánager) tiene permisos de escritura.

Los demás integrantes:

- realizan un fork del repositorio remoto A obteniendo un repositorio remoto nuevo que ya es de su propiedad. Llamémosle A1
- clonan ese repositorio remoto A1 en su máquina local
- Trabajan en los cambios, haciendo uno o varios commits
- Hacen un push a su repositorio remoto A1
- Lanzan un pull-request al mánager.

El Manager aceptará los cambios o los rechazará

Es bastante usado en proyecto open-source con muchos desarrolladores.

Flujo de trabajo centralizado

Existe un repositorio global en el que todos los desarrolladores tienen permisos de escritura.

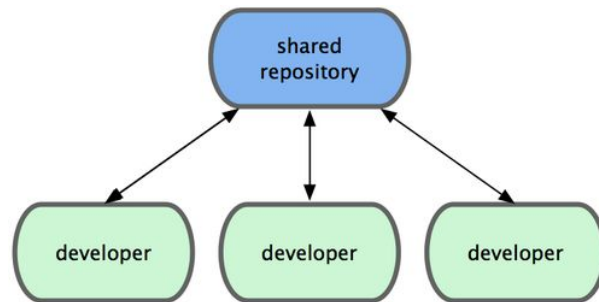
Tenemos 2 variantes:

- Flujo de trabajo centralizado en la rama master (recomendado para grupos muy pequeños)
- Flujo de trabajo centralizado con gestor de integraciones (Usado normalmente en empresas)

Flujo de trabajo centralizado en única rama master

En este caso, cualquier desarrollador publica todos su cambios directamente en la rama master del repositorio remoto (global)

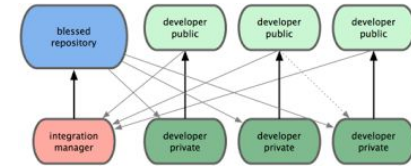
Solo recomendado para equipos con 1, 2 o 3 integrantes, donde el trabajo de cada desarrollador esté perfectamente delimitado.



Flujo de trabajo centralizado con gestor de integraciones

En este caso cuando un desarrollador va a trabajar en un cambio creando una rama:

- Un desarrollador crea una rama nueva
- Realiza cambios haciendo uno o más commits sobre esa rama
- Publica su rama en el repositorio remoto
- Abre un pull-request pidiendo que alguna persona lo revise
 - Estos cambios los puede revisar cualquiera
 - Lo normal es que en la empresa exista un **rol** llamado el **Gestor de integraciones** o **Integrador** que se va a encargar de verificar que los nuevos cambios cumplan con los estándares de calidad de la empresa.
- Integrar los cambios haciendo un merge



Ejercicio: flujo de trabajo centralizado con gestor de integraciones (sin conflictos)

Supongamos que existe un repositorio remoto que va a funcionar como global para todos los integrantes del equipo.

Lo clonamos en nuestro repositorio local

git clone https://github.com/xxxx/yyy.git

Nos posicionamos en la rama master

git checkout master

Actualizamos nuestro repositorio local con los cambios que pudieran existir en el repositorio remoto

git pull

Creamos una nueva rama

git checkout -b UH-45-List-users

Trabajamos en los cambios realizando commits. Añade un <h1> a la pagina index.html

code index.html git add . git commit -m "add un h1"

Verificamos contra master si nuestros cambios están al día con respecto a la rama master del repositorio remoto, ya que esta rama pudo haber avanzado, es decir, los otros desarrolladores pudieron integrar sus cambios a esa rama y es posible que tengamos conflictos.

Actualizamos las ramas de seguimiento remoto

git fetch origin

Cogemos nuestros nuevos commits, separarlos de la base actualizamos la rama local y aplicamos los commits encima de esos cambios.

git rebase origin/master

Subimos la rama a remoto

git push -u origin UH-45-List-users

Accedemos a github y abrimos un pull request pulsando compare & pull request o accediendo a la pestaña branches y pulsando en new pull request de tus branches.

Comprobamos que la rama base es master y la rama a comprar es **UH-45-List-users** y creamos el pull request

Ahora el **Integrador** entra en juego y aceptará o rechazará los cambios. En este caso vamos a suponer que acepta los cambios. En github pulsaría Merge

Nosotros volvemos a la rama master

git checkout master

Actualizamos el repositorio

git pull

Podemos borrar la rama, tanto en local como en remoto pulsando el botón en Github (Delete branch)

git branch -d UH-45-List-users

Ángel González M.

Ejercicio: flujo de trabajo centralizado con gestor de integraciones (con conflictos)

Actualizamos nuestro repositorio local con los cambios que pudieran existir en el repositorio remoto

git pull

Creamos una nueva rama

git checkout -b UH-46-filter-user

Trabajamos en los cambios realizando commits. Añade un <h2> a la pagina index.html

code index.html **git add .** **git commit -m "add <h2>"**

Ahora vamos a suponer que otro compañero de trabajo también añadió un <h2> a la página web (tendremos un conflicto). Los simulamos haciendolo desde github, añadimos un cambio desde github en una nueva rama, creamos un pullrequest y aceptamos (merqueamos los cambios)

Verificamos contra master si nuestros cambios están al día con respecto a la rama master del repositorio remoto, ya que esta rama pudo haber avanzado, es decir, los otros desarrolladores pudieron integrar sus cambios a esa rama y es posible que tengamos conflictos.

git log --oneline --graph **git branch -a**

Actualizamos las ramas de seguimiento remoto

git fetch origin

Cogemos nuestros nuevos commits, separarlos de la base actualizamos la rama local y aplicamos los commits encima de esos cambios.

git rebase origin/master

Esto nos va a dar un conflicto

Abrimos el fichero index.html para solucionar el conflicto

code index.html

Resolvemos el conflicto (normalmente hay que hablar con la persona correspondiente)

git add index.html

git rebase --continue

Subimos la rama a remoto

git push -u origin UH-46-filter-user

Accedemos a github y abrimos un pull request pulsando compare & pull request o accediendo a la pestaña branches y pulsando en new pull request de tus branches.

Comprobamos que la rama base es master y la rama a comprar es **UH-45-List-users** y creamos el pull request

Ahora el **Integrador** entra en juego y aceptará o rechazará los cambios. En este caso vamos a suponer que acepta los cambios. En github pulsaría Merge

Nosotros volvemos a la rama master **git checkout master**

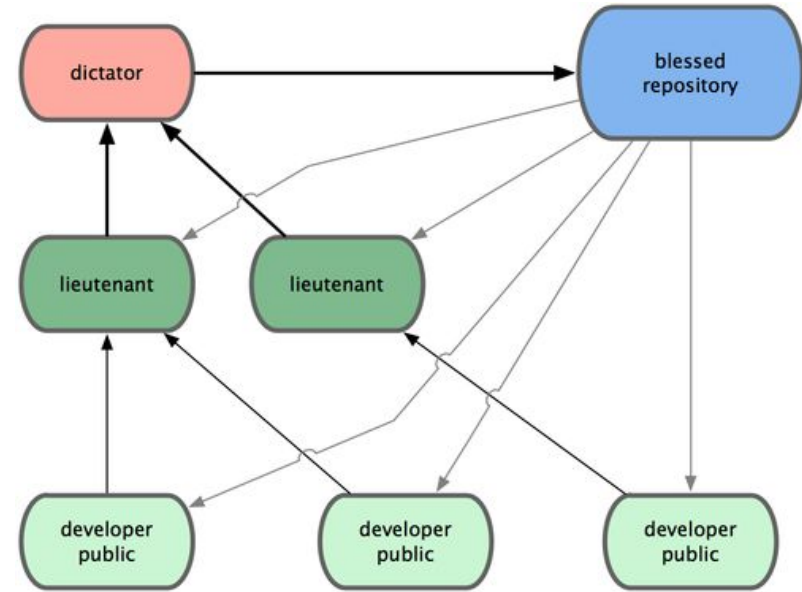
Actualizamos el repositorio **git pull**

Podemos borrar la rama, tanto en local como en remoto pulsando el botón en Github (Delete branch)

Ángel González M.
git branch -d UH-46-filter-user

Flujo de trabajo con Dictador y Tenientes

Es una variante del flujo de trabajo con multiples repositorios. Se utiliza generalmente en proyectos muy grandes, con cientos de colaboradores. Un ejemplo muy conocido es el del kernel de Linux. Unos gestores de integración se encargan de partes concretas del repositorio; y se denominan tenientes. Todos los tenientes rinden cuentas a un gestor de integración; conocido como el dictador benevolente. El repositorio del dictador benevolente es el repositorio de referencia, del que recuperan (pull) todos los colaboradores.



Feature Branch Workflow

Feature Branch Workflow

Vamos a suponer que hay 2 ramas: master y develop

Trabajamos en una nueva características creando una rama en develop

git checkout develop

git checkout -b nuevaFeatureDelUsuarioA

Estos 2 comandos se pueden resumir en **git checkout -b nuevaFeatureDelUsuarioA develop**

Realizamos commits en esta rama (los que hagan falta) **git add git commit**

En algún momento (incluso antes de comenzar a trabajar) podemos subir esta rama a remoto **git push -u origin nuevaFeatureDelUsuarioA**

- así hacemos ver a nuestros compañeros que el UsuarioA comenzó a trabajar en la característica X
- el modificador -u sube todo el upstream a remoto

Cuando el usuarioA lo decida subirá el código a remoto (vamos a suponer que el usuarioA no ha terminado ninguna release) **git push origin nuevaFeatureDelUsuarioA**

Lo siguiente que hace el usuarioA es acceder a GitHub y hacer un **pull-request** para que sus compañero o el Integrator revisen el código.

Una vez que los cambios han sido aceptados mergueamos la nueva rama (mediante un fast forward) en develop

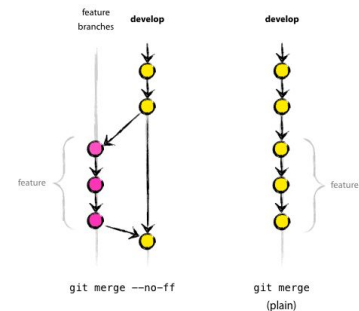
git checkout develop

git merge --no-ff nuevaFeatureDelUsuarioA El modificador --no-ff queremos que en el histórico quede reflejado quien y cuando se hizo el merge. Esto es tremendamente útil para investigar errores. Es decir se va a hacer siempre un commit, aunque se pudiera hacer por fast-forward

git push origin develop Subimos los cambios a la rama remota develop

git branch -D nuevaFeatureDelUsuarioA Ya no necesitamos la rama, así que la borramos

git push origin :nuevaFeatureDelUsuarioA Con esto no enviamos y borramos la rama en remoto



Ahora que el UsuarioA ha terminado, los demás usuarios tienen sus repositorios desactualizados. Vamos a actualizarlos

`git fetch origin`

`git pull origin`

Subir una release

Cuando se terminan de realizar todas las historias de usuario del sprint, hay que subir una release

```
git checkout master
```

```
git fetch origin
```

```
git pull origin    # aseguramos que tenemos todo actualizado
```

```
git checkout develop
```

Creamos una rama release. La idea es que desde ese momento no se permite incorporar nada nuevo a la release. Aunque se pueden seguir trabajando en nuevas características. Es lo bueno de este flujo de trabajo, el flujo nunca termina.

```
git checkout -b release/1.2
```

Esto lo podríamos resumir en `git checkout -b release-1.2 develop`

```
git checkout master
```

```
git merge --no-ff release/1.2
```

 Integramos todo nuestro histórico. El código en production va a tener todos los cambios

```
git push origin master
```

```
git tag -a 1.2 -m "Nueva release 1.2. Creación de clientes"
```

Ponemos un tag para identificar correctamente la release

```
git push origin 1.2
```

 subimos el tag

```
git checkout develop
```

```
git merge --no-ff release/1.2
```

 Si la release/1.2 no ha incluido ningún cambio, este merge dejará el histórico como lo teníamos anteriormente

```
git push origin develop
```

```
git branch -D release/1.2.
```

 Podemos borrar la rama de la release

Uso de git flow

git flow init

git flow feature start nuevafeature

code nuevoFichero.txt

git add .

git commit -m "add nuevo fichero"

git flow feature publish nuevafeature

git flow feature finish nuevafeature

PROYECTOS EN EQUIPO

Método de trabajar cuando trabajamos en un equipo, donde nosotros somos los dueños o somos colaboradores en un proyecto.

Rama oculta espejo

Cuando nos conectamos con un repositorio remoto con el comando

```
git remote add origin https://AngelGonzalezM@bitbucket.org/AngelGonzalezM/pruebaangel.git
```

se crea una rama oculta que se sitúa entre medias del repositorio local y el repositorio remoto y funciona como una rama espejo.

para poder ver esa rama oculta usamos el comando

```
git branch -a
```

```
* master  
test  
remotes/origin/master
```

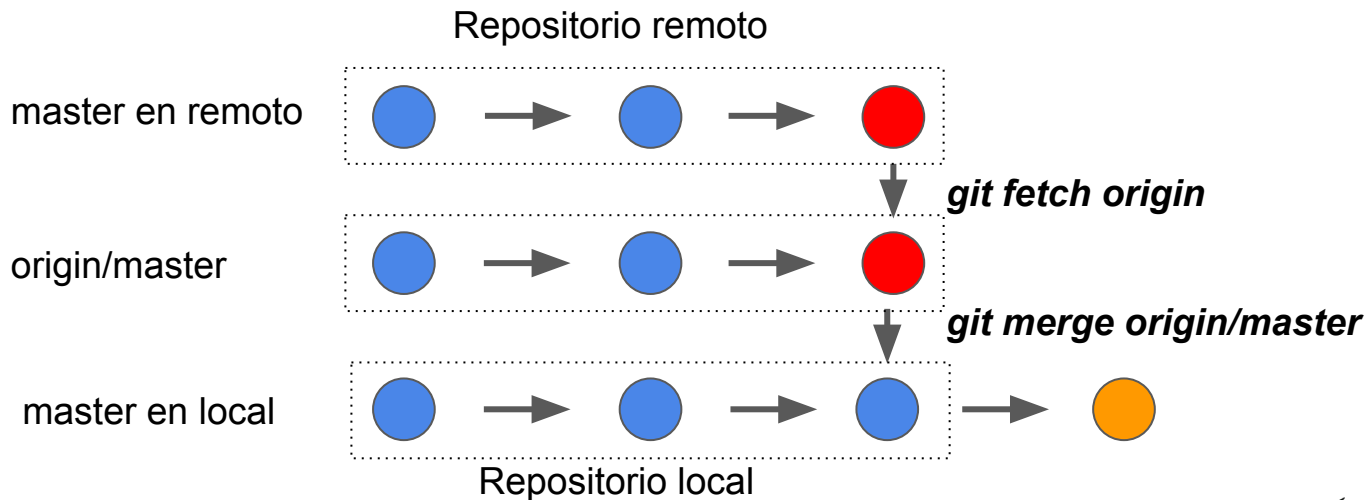
Recuerda `git pull` es la combinación de `git fetch` y `git merge` vamos a ver el funcionamiento de estos 2 comandos

Comando: fetch

Imaginemos que tengo un comint en remoto que aun no tengo en local.

Con el comando **git fech origin** paso el comit de remoto a la rama intermedia origin/master

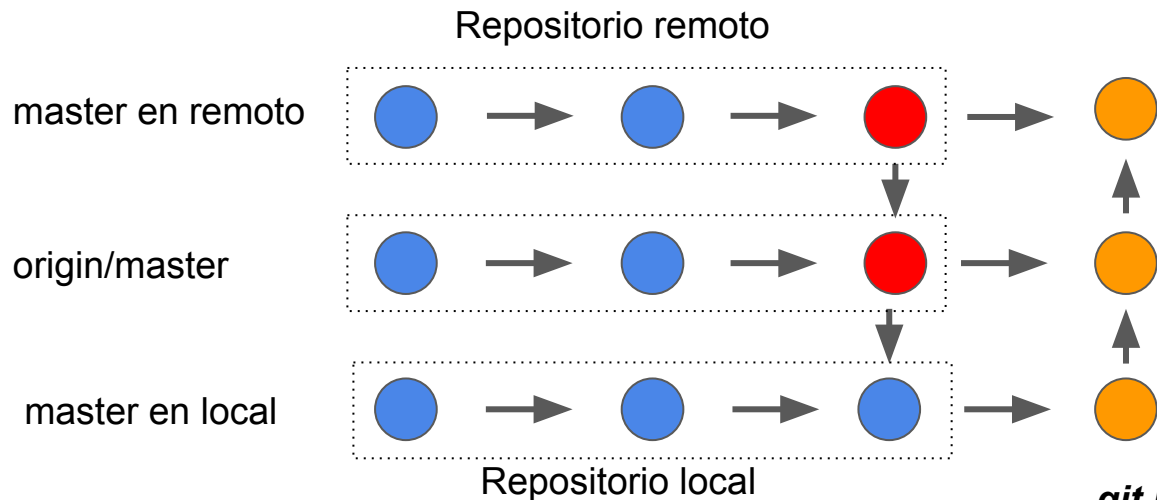
Con el comando **git merge origin/master** paso el commit de la rama intermedia a la rama local



continuando

Ahora ya podemos hacer un push

git merge origin/master



git push origin master

Ángel González M.

Ejemplo sin conflictos

Imaginemos que 2 desarrolladores modifican partes distintas del código

Tenemos 2 desarrolladores (llamémosles A y B) modifican cosas distintas en el código y suben sus cambios.

El usuario que suba primero sus cambios no va a tener problemas, imaginemos que es A quien hace el primer push

```
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

El segundo usuario B en subir los cambios con **git push -u origin master** se va a encontrar con un mensaje de git indicando que tiene que actualizar los cambios en el repositorio

El segundo usuario ejecuta los comandos

git fetch origin

git merge origin/master

y ya podrá hacer un **git push -u origin master**

Ángel González M.

Ejemplo con conflictos

Lo visto anteriormente funciona muy bien si varios desarrolladores tocan partes del código diferente...

Pero que pasa si por ejemplo 2 desarrolladores (llamémosle A y B) modifican la misma línea a la vez y suben sus cambios.

El usuario que suba primero sus cambios no va a tener problemas, imaginemos que es A quien hace el primer push

El segundo usuario B en subir los cambios se va a encontrar con un mensaje de git indicándole que quiere que actualizarse los cambios en el repositorio

```
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

El segundo usuario ejecuta el comando `git fetch origin` (sin problemas)

Pero cuando ejecuta el comando `git merge origin/master` git lanzará un error

```
Auto-merging nuevo.txt
CONFLICT (content): Merge conflict in nuevo.txt
Automatic merge failed; fix conflicts and then commit the result.
```

Este error habrá que resolverlo manualmente ya que git no es capaz de hacerlo solo.

Ángel González M.

Colaborar en proyectos de terceros

Proyectos de terceros en los que yo pretendo colaborar. Pero no soy colaborador, ni integrante de un grupo, ni por supuesto propietario del proyecto.

Comando fork

El comando fork crea una copia exacta de un repositorio remoto y lo guarda en tu repositorio local.

Podríamos pensar que es igual que lo que hace el comando clone, pero hay una diferencia.

Clone crea una copia exacta en tu local y cualquier cambio que hagas, al hacer un push modificará el repositorio remoto.

Fork también crea una copia exacta en tu local pero cuando haces un cambio y lanzas un push, el repositorio remoto no cambia.

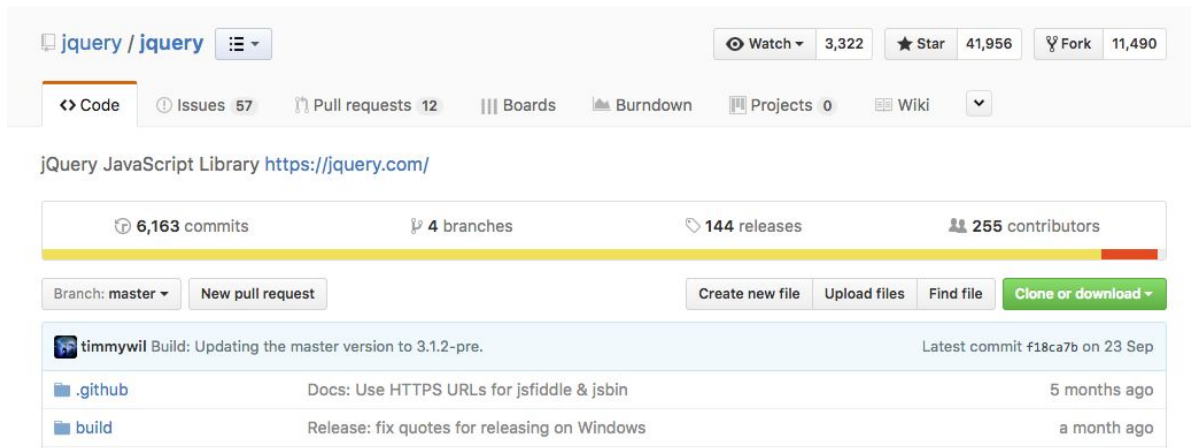
Interesante para contribuir en proyecto de terceros cuando tu no eres el dueño, ni eres colaborador de un repositorio remoto.

Ángel González M.

Como se realiza un fork

Accedemos al repositorio buscandolo en la web de github.

Y buscamos un enlace que diga Fork



The screenshot shows the GitHub repository page for jQuery. At the top, the repository name 'jquery / jquery' is displayed. To the right, there are buttons for 'Watch' (3,322), 'Star' (41,956), and 'Fork' (11,490). Below these, there are tabs for 'Code', 'Issues' (57), 'Pull requests' (12), 'Boards', 'Burndown', 'Projects' (0), and 'Wiki'. The repository description is 'jQuery JavaScript Library' with a link to 'https://jquery.com/'. Below this, statistics are shown: '6,163 commits', '4 branches', '144 releases', and '255 contributors'. A yellow progress bar is visible. At the bottom, there are buttons for 'Branch: master', 'New pull request', 'Create new file', 'Upload files', 'Find file', and 'Clone or download'. A commit history table is also shown.

Commit	Message	Time
timmywil	Build: Updating the master version to 3.1.2-pre.	Latest commit f18ca7b on 23 Sep
.github	Docs: Use HTTPS URLs for jsfiddle & jsbin	5 months ago
build	Release: fix quotes for releasing on Windows	a month ago

Ahora en nuestra cuenta, tendremos el mismo proyecto (de tipo fork)

Ángel González M.

Si antes mediante el comando **git fetch origin** teníamos una rama oculta llamada **origin/master**

Ahora vamos a tener una rama oculta adicional llamada **upstream/master** mediante el comando **git fetch upstream**

Ejemplo de uso de fork

Accedemos a un repositorio de la (página de github) del que nos interese contribuir: Ej Ruby

Pulsamos en el botón **fork**. Esto creará un clon de jquery en nuestro repositorio remoto.

Nos descargamos el proyecto a nuestro local con el comando

git clone <https://github.com/kant003/ruby.git>

Hacemos los cambios permanentes (añadimos, corregimos bug's, etc) y lo subimos a remoto.

git add .

git commit -m "Soluciono bug de funcionamiento ... que encontré"

git push -u origin master

Ahora accedemos a la web de github, entramos en el proyecto de nuestro repositorio remoto y pulsamos el botón de **new pull request**

Cuando el dueño del repositorio original vea el mensaje, podrá aceptar o rechazar el cambio.

Ángel González M.

Git Ignore

Podemos crear un fichero de texto llamado `.gitignore`, indicando en su interior los ficheros que queremos que nunca se suban al repositorio.

```
#Test files
```

```
test.xml
```

```
#ficheros de password
```

```
password.lock
```

```
password.txt
```

```
#Others
```

```
build
```

```
vendor
```

```
coverage.clover
```

```
/cache/*
```

```
temp/*.tmp
```

```
#Mac files
```

```
.DS_Store
```

Más información acerca de gitignore <http://aprendegit.com/tag/gitignore/>

Ángel González M.

Ejercicio

Haz que no se guarden en el repositorio todos los archivos de log (extensión .log)

Edito el fichero .gitignore

code .gitignore

Añado en su interior

***.log**

(* = significa cualquier cosa)

SSH

Claves SSH (**Secure SHell**)

Es el nombre de un protocolo y del programa que lo implementa, y sirve para acceder a máquinas remotas a través de una red. Permite manejar por completo la computadora mediante un intérprete de comandos, y también puede redirigir el tráfico de X (Sistema de Ventanas X) para poder ejecutar programas gráficos si tenemos ejecutando un Servidor X (en sistemas Unix y Windows).

Además de la conexión a otros dispositivos, SSH nos permite copiar datos de forma segura (tanto archivos sueltos como simular sesiones FTP cifradas), gestionar claves RSA para no escribir claves al conectar a los dispositivos y pasar los datos de cualquier otra aplicación por un canal seguro tunelizado mediante SSH.

Generación de claves

El siguiente comando genera las claves privada y pública (certificado de usuario) necesarias para el proceso de comunicación

```
ssh-keygen
```

Nos pedirá un directorio (si no especificamos nada nos crea una carpeta oculta llamada .ssh) y un password de seguridad para cifrar la clave privada.

Los ficheros que obtenemos son los siguientes:

id_rsa: este fichero es la clave **privada** y nadie debería poder acceder a ella

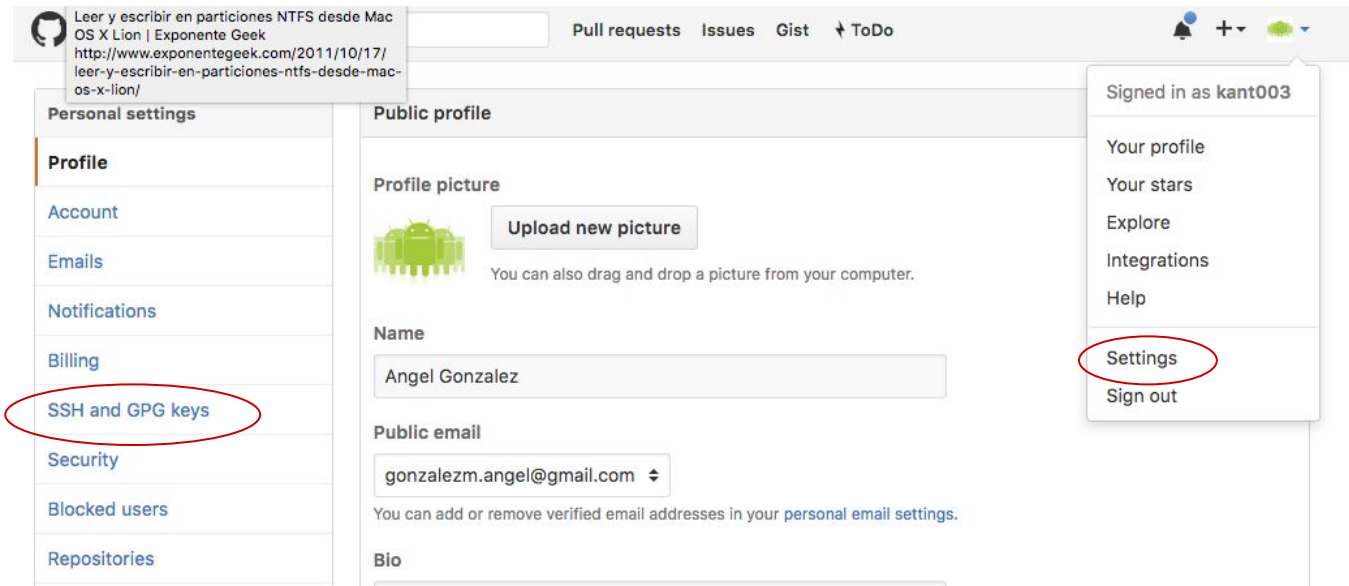
id_rsa.pub: esta es la clave **pública** y cualquiera puede recibir este fichero.

Ángel González M.

Configurando github con la clave ssh pública

Accedemos a la configuración de github pulsando en Settings

Y luego al la zona de **SSH and GPG key's**



Pegamos el contenido del fichero **id_rsa.pub**

Ángel González M.

Comprobamos si lo hemos hecho bien

```
ssh -T git@github.com
```

También puedes probar a clonar un proyecto vía ssh

```
git clone git@github.com:kant003/JavaPracticeHacktoberfest.git
```


Usando SSH

Ahora cuando nos conectemos a un remoto como hacíamos antes por HTTPS

```
git remote add origin https://AngelGonzalezM@bitbucket.org/AngelGonzalezM/prueba.git
```

Lo haremos por SSH

```
git remote add origin git@github.com:kant003/pruebas1.git
```

concepto: Hooks

Los hooks son acciones que se ejecutan automáticamente después de que ciertos comandos ocurran

Creando una página web personal asociada a github

Crea un nuevo repositorio y llámale `<tu_nombre_usuario>.github.io`

Crea un fichero `index.html` o `index.md`.

También puedes usar un tema predefinido (nos creará un fichero `_config.yml`).

En settings, activa Github Pages

Accede a `https://<tu_nombre>.github.io` (esto puede tardar un rato)

Creando una página web asociada a un repositorio de github

Podemos crear un fichero index.html o index.md en:

- En la rama master de nuestro repositorio
- En la carpeta /docs de la rama master de nuestro repositorio
- En un rama especial llamada gh-pages (antes tendremos que crear esa rama)

Visitamos https://<tu_nombre>.github.io/<tu_repositorio>

Gist

Son como mini-repositorios compuestos de unos pocos archivos

Sirven para compartir notas, trozos de código, snippets,...

Pueden ser públicos o secretos (solo el que recibe la url puede verlo)

Ejercicio

Crea un archivo Gist publico con el esqueleto de página web.

Llamale index.html

Crea un personal access token en github en: Settings>Developer settings>Personal Access Token

Instala el plugin(extension) de gits en visual studio code (logueate con token)

Usa el plugin shift+cmd+p

Configurar git con proxy

```
git config --global http.proxy http://10.100.13.3:3128
```

```
git config --global https.proxy http://10.100.13.3:3128
```

```
git config --list
```

QUITAR EL PROXY

```
git config --global --unset http.proxy
```

```
git config --global --unset https.proxy
```