

Laboratorio



CL Oct20

Arrancando

- Ayuda comandos de GIT

`git help comando`

- Chuleta de comandos

[Disponible en el aula virtual](#)

- Simulador de git.

[Enlace](#) ([Recomendable revisar todas las simulaciones](#))

Configuración

```
git config --global user.name "Tu nombre aquí"  
git config --global user.email "tu_email_aquí@example.com"
```

```
git config --global -list
```

Lo que hace el sistema es crear un archivo de texto llamado **.gitconfig**, por lo que podemos mostrarlo también con cat:

```
cat ~/.gitconfig
```

Comenzar a trabajar con Git

A) Working Directory

Nos situamos en subdirectorio en el que vamos a trabajar que llamaremos **university**. Nos aseguramos con `pwd`, para saber dónde estamos. Lo creamos dentro de un directorio que llamaremos "**GitProjects**"

B) Repository (Repo)

Ahora con `git init` y el nombre del proyecto, creamos un nuevo repositorio:

```
git init university
```

El subdirectorio .git

- Vemos su contenido: `ls -C .git`

```
$ ls -C .git
COMMIT_EDITMSG  description  HEAD      index  logs/  refs/
config         gitk.cache  hooks/    info/  objects/
```

- The Object Store. Contiene varios subdirectorios nombrados con las dos primeras letras del hash asociado a cada objeto almacenado. Cada subdirectorio almacena un fichero comprimido con una parte de los objetos del repo.
- Branches and Tags:
 - `ls .git/refs`
 - `ls .git/refs/heads`
 - `ls .git/refs/tags`
- The HEAD File. El fichero HEAD contiene una referencia permanente a la rama actual (se suele denominar *master*)

Creamos nuestro primer prototipo

- Creamos nuestro primer prototipo a partir de una plantilla.
Descargamos un proyecto bootstrap en nuestro repo

[Descarga](#)

- Comprobamos la situación de partida

```
git status
```

- Hacemos el primer commit

```
git add .  
git commit -m "Primer commit"
```



Trabajando con Git en local

- Modificamos el fichero **index.html** para que incluya “The university of excellence” en lugar de “What We Offer”
- Modificamos el fichero **courses.html** para que el primer “Electric Engineering” se sustituya por “Software Engineering” y el segundo por “Computer Science Engineering”
- Comprobamos el estado del repo
- Se decide que solo pase el último fichero.
- Comprometemos los cambios.
- Sacamos index.html del staged
- Vemos retrospectiva del Proyecto (log)

IDE



git status

git add courses.html

git commit -m “Segundo Commit”

git reset HEAD index.html

git log --pretty=oneline

==> Prueba **gitk**

Git en acción

- Editamos el fichero *about.html*, eliminamos todo su contenido y lo sustituimos por el siguiente código, grabando tras los cambios...

```
<h2>About Us</h2>
```

```
<p>We are a friendly university for international students</p>
```

- Borramos desde las utilidades del sistema operativo el subdirectorio *images*. Vaciamos la papelera.
- Visualizamos el resultado desde el navegador. **Desastre!!!**
- La magia de git...
- Recuperación ante desastres....

git checkout -- .

Alias para hacer más fácil el trabajo

- Localizamos el fichero .gitconfig en el \$HOME de nuestro equipo.
- Editamos el fichero y añadimos la siguiente sección

[alias]

co = checkout

ci = commit

st = status

br = branch

hist = log --pretty=format:'%h %ad | %s%d [%an]' --graph --date=short

Importante: Para editar más fácilmente cualquier salida por consola que nos solicite git es conveniente utilizar editors más amigables que “Nano”. Para ello en Win: git config core.editor notepad

En Ub/Linux: git config core.editor gedit

La línea de tiempo: detached

- Obtener los hashes de commits previos.
- Moverse a un commit previo haciendo uso del hash asociado (los 7 primeros dígitos son suficientes)
- Observar el resultado mostrado: **'detached HEAD'**
- Visualizamos el ficheros `course.html` para comprobar que mantiene la información de ese momento.
- Volver al presente, es decir, al commit vigente.

`git hist` (*alias creado en .gitconfig*)

`git checkout <hash de tu repo>` →

`git checkout master`

```
$ git checkout 4445720
Note: checking out '4445720'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

    git checkout -b <new-branch-name>

HEAD is now at 4445720 First Commit
```

Etiquetando versiones: tagging

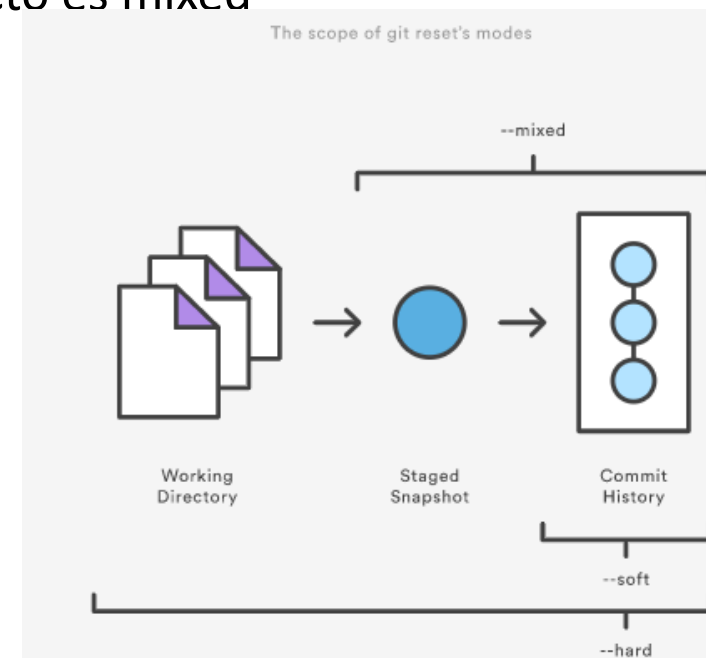
- Asignarle al prototipo actual la etiqueta V1.0
- Movernos en la línea de tiempos mediante etiquetas
- Etiquetar commits previos
- Ckecking con etiquetas
- Ver las etiquetas creadas
- Ver las etiquetas en la información de logs. Borramos etiqueta V1.0

```
git tag V1.0
git checkout V1.0^ (el padre del commit etiquetado con V1.0) Detached!!
    # También se podría haber utilizado: git checkout V1.0~1
    # Nota: esta notación (^,~) también se puede utilizar con hash y HEAD
git tag V1.0-Beta
git checkout V1.0-Beta; git checkout V1.0
git tag
git hist master --all
git tag -d V1.0
```

Eliminando commits

- Creamos un commit de prueba para borrar.
- Eliminar el commit actual. **Peligroso cuando se comparte código!!!**
- Otra forma de borrado “ligero” es `revert`. Se ignoran ciertos commits, aunque se mantienen en el log.
- Existen tres modalidades de `reset`: **hard, soft and mixed**. Por defecto es mixed

```
git commit -m "Oops, It's a dummy commit"  
git reset HEAD^ --hard  
git hist --all
```

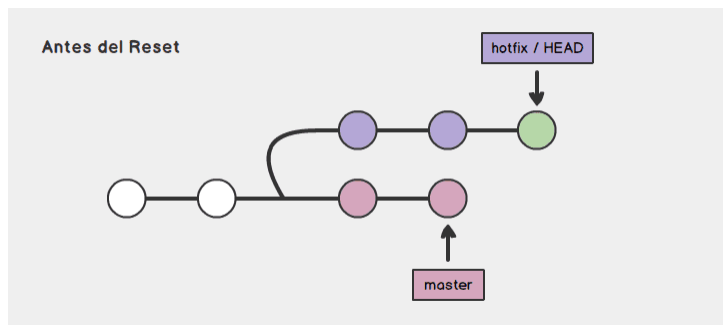


Eliminando commits: reset vs revert

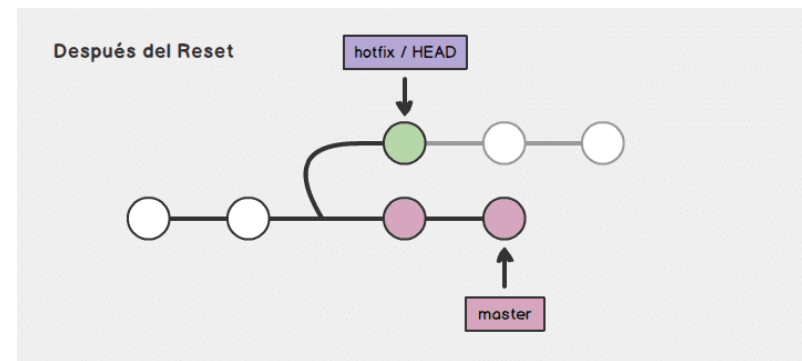
reset

Un *reset* es una operación que toma un *commit* específico y restablece el historial para que coincida con el estado del repositorio en ese *commit* específico.

Supongamos que, partiendo de la siguiente situación, ejecutamos el comando `git reset HEAD~2`

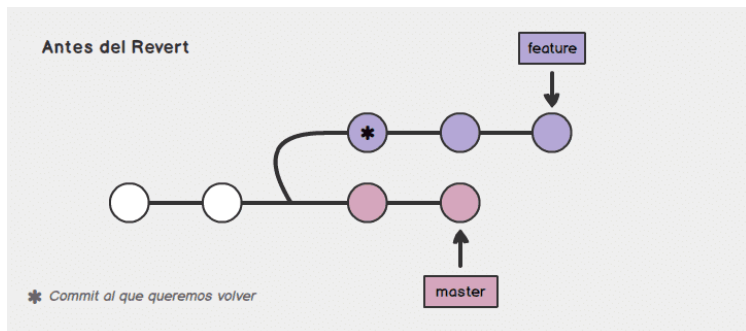


Deshacemos los dos últimos *commits* de la rama *Hotfix*. Estos dos *commits* quedarán huérfanos y se eliminarán la próxima vez que Git haga limpieza.

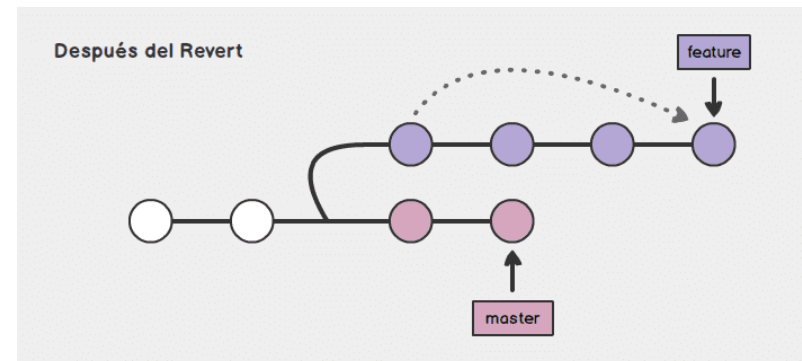


revert

Copia el *commit* que queremos mantener en uno nuevo y crea una nueva secuencia que salta los commits que se quieren eliminar. Supongamos que partimos de la siguiente situación y ejecutamos `git revert HEAD~2`



Deshacemos los cambios de los dos últimos *commits* de la rama *Hotfix* añadiendo un nuevo *commit* que ejecuta los cambios. No entra el garbage coll.



Creando ramificaciones del Proyecto: Branching

- Vamos a crear una nueva rama llamada “Develop”
- Modificamos el fichero teacher.html para cambiar el nombre de la profesora Bianca Wilson por Blanca Wattson.
- Preparamos y comprometemos los cambios.
- Modificamos el fichero blog.html para cambiar “Skills To Develop Your Child Memory” por “Build your future”.
- Preparamos y comprometemos estos cambios. Vemos el resultado

```
git branch develop; git checkout develop
```

```
# Se puede hacer de forma más directa: git checkout -b develop
```

```
#Modificación teacher.html
```

```
git add . ; git commit -m “Primer commit rama develop”
```

```
#Modificación blog.html
```

```
git add . ; git commit -m “Segundo commit rama develop”
```

```
git hist --all
```

Saltando entre las ramas

- Ver todas las ramas del repo
- Volver a la rama principal (máster)
- Saltar a la rama develop

```
git branch o git hist -all  
git checkout master  
git checkout develop
```

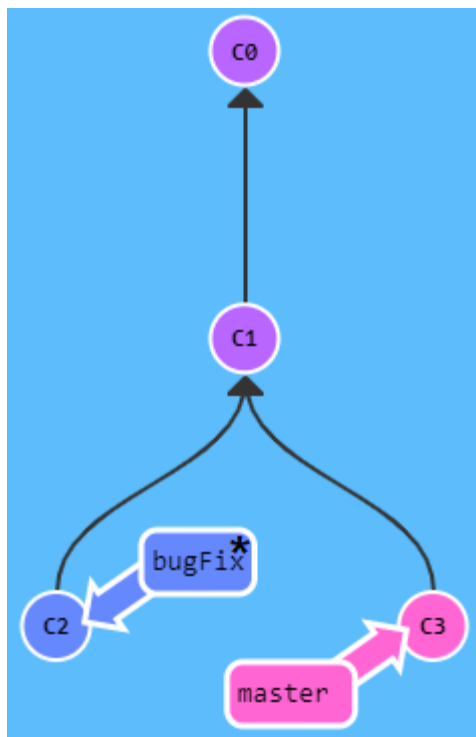
Trabajando con ramas

- Para ver el último commit en cada rama
`git branch -v`
- Ver qué ramas han sido “merged/unmerged” en la rama en la que estés
`git branch -merged`
`git branch -unmerged`
- Cambiar el nombre de una rama (Do not rename branches that are still in use by other collaborators)
`git branch --move <bad-branch-name>`
`<corrected-branch-name>`

Mezclando ramas: Merging

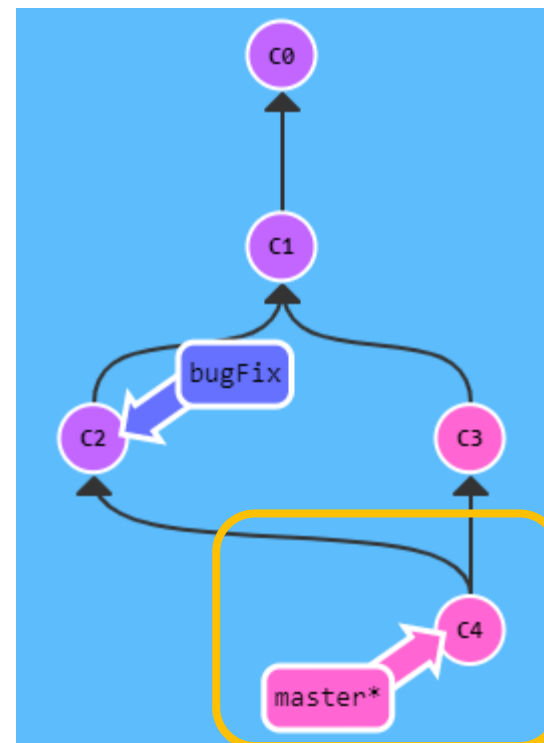
El *merging* nos permite unificar el trabajo de dos ramas diferentes. Esto nos va a posibilitar abrir una nueva rama de desarrollo, implementar alguna nueva funcionalidad, y después unirla de nuevo con el trabajo principal.

(1) Supongamos esta situación con dos ramas: bugFix y master

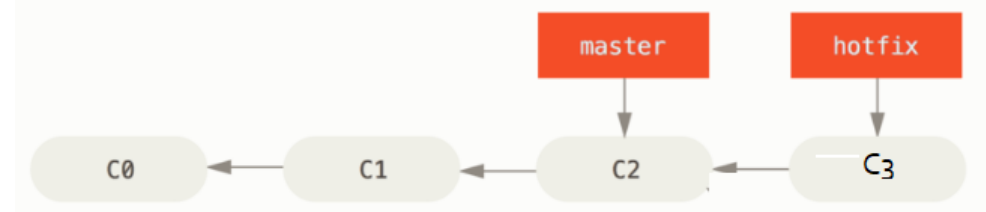


(2) `git checkout master`
`git merge bugFix`

(3) Se han fusionado ambas ramas y se ha creado un nuevo commit en master



Mezclando ramas: Merging en acción



- Borramos el repo creado usando las herramientas del sistema operativo e inicializamos uno nuevo en el working directory que ya tenemos.
- Preparamos y comprometemos los cambios
- Modificamos el fichero index.html para que el email del comienzo sea info@ufv.es
- Preparamos y comprometemos los cambios
- Modificamos el fichero courses.html para que el primer mensaje “10 SEATS” se sustituya por “NOT AVAILABLE”
- Preparamos y comprometemos los cambios.
- Creamos una nueva rama a partir de master denominada “hotFix”
- Modificamos el fichero index.html para que el primer teléfono sea +34 91 8880000
- Preparamos y comprometemos los cambios
- Pasamos a la rama master
- Integramos la rama hotFix en master
- Borramos la rama hotFix

Mezclando ramas: Merging solución

```
#Borramos subdirectorio .git
git init; git add .; git commit -m "C0"
```

```
#Modificamos index.html
git add .; git commit -m "C1"
```

```
#Modificamos courses.html
git add .; git commit -m "C2"
```

```
git checkout -b hotFix
```

```
#Modificamos index.html
git add .; git commit -m "C3"
```

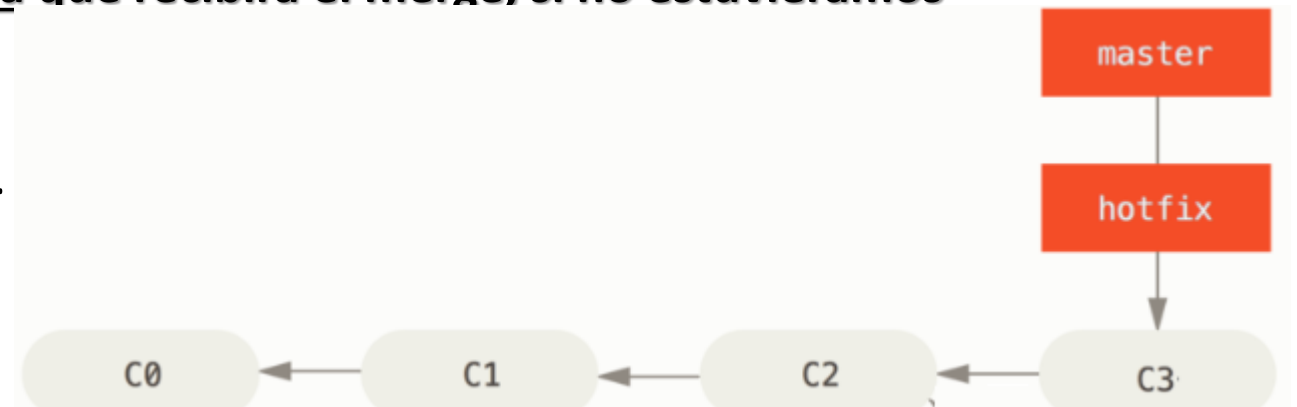
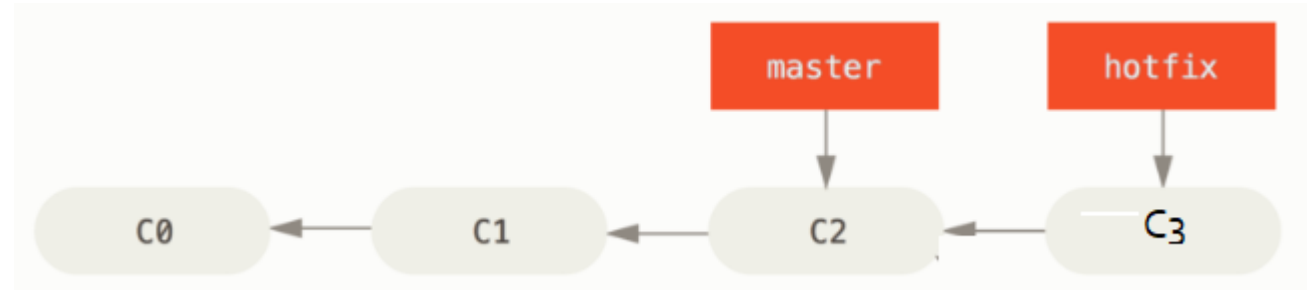
```
git checkout master #Nos colocamos en la rama que recibirá el merge, si no estuviéramos
```

```
git hist -all # Ver relación de ancestros ...
```

```
git merge hotFix # fast-forward
```

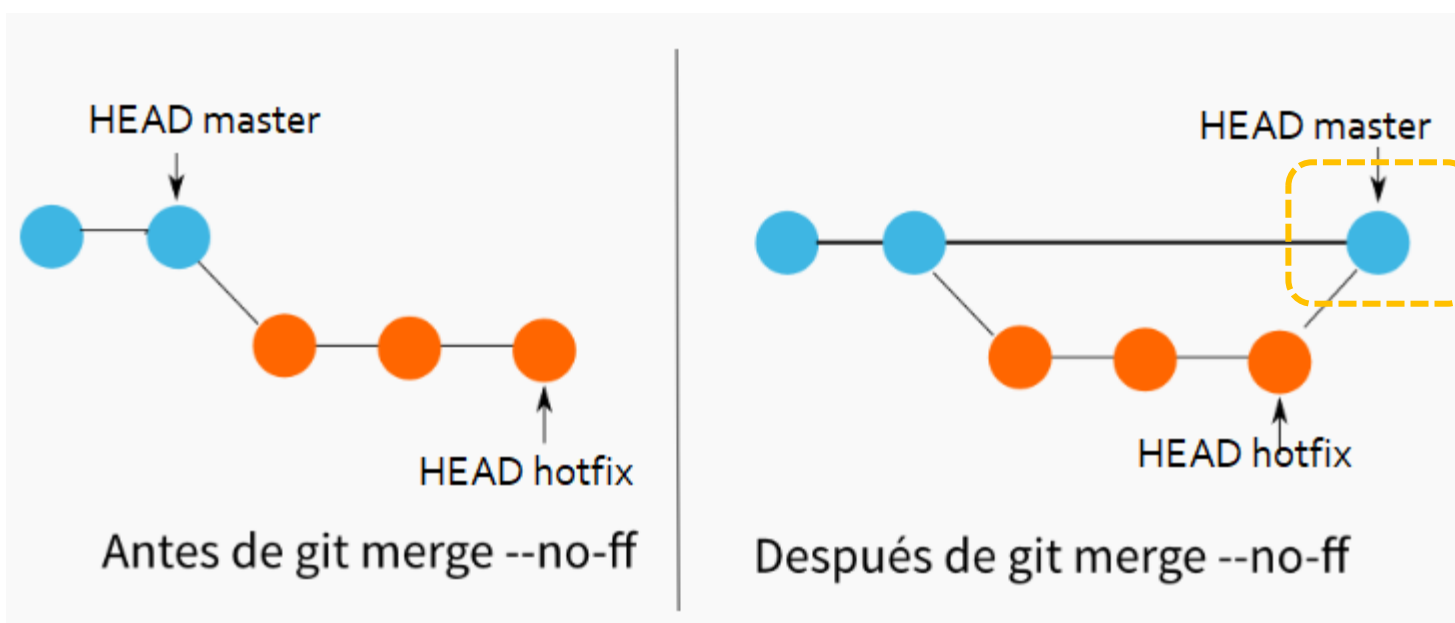
```
git hist -all # Ver nueva relación de ancestros ...
```

```
git branch -D hotFix
```



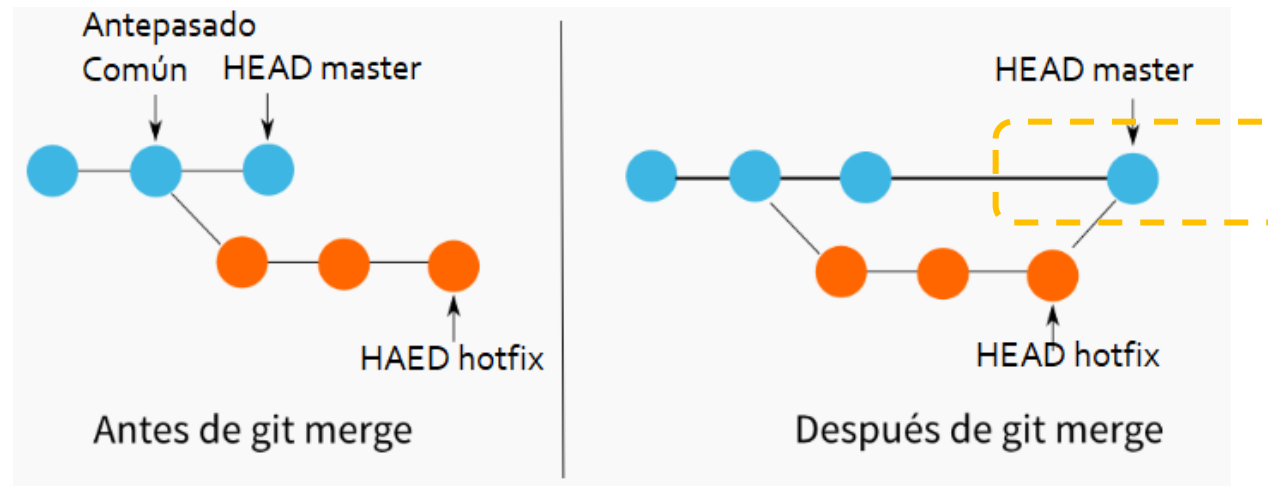
Mezclando ramas: Alternativas I

Fast-Forward: En el ejemplo anterior hemos visto que en el momento de hacer el merge la rama master no se había cambiado tras haber creado la rama hotfix. En este caso el problema se resuelve avanzando el HEAD de master para que coincida con hotfix. No se ha creado un nuevo commit como consecuencia de la fusión. Incluso en estos casos puede ser interesante dejar constancia mediante un **commit “forzado”**: `git merge --no-ff hotfix`



Mezclando ramas: Alternativas II

Recursive: si la rama master tuviera un nuevo commit después de haber creado la rama hotfix ya no es posible un fast-forward merge, debido a que el commit de la rama donde actualmente se está (master) no es un antepasado directo de la rama a fusionar (hotfix) por tanto, git realiza un merge a tres bandas (**three-way algorithm**), es decir, se genera siempre un commit adicional para fusionar las dos ramas, tomando en cuenta el HEAD de cada una de ellas y el antepasado común de las dos



Puede haber otras estrategias del merging que te animo a investigar:

- Recursive: `git merge -s recursive branch1 branch2` (forzar recursive)
- Resolución: `git merge -s resolve branch1 branch2`
- Octopus: `git merge -s octopus branch1 branch2 branch3 branchN`
- Ours: `git merge -s ours branch1 branch2 branchN`
- Subtree: `git merge -s subtree branchA branchB`

Mezclando ramas: Conflictos

Algunas veces la unión de dos ramas origina un conflicto, ya que los commits de la rama a fusionar y la rama actual modifican la misma parte de código de un mismo archivo, git no podría decidir con qué versión quedarse. Veámoslo con un ejemplo, previamente elimina el repositorio e inicializa uno nuevo:

Supongamos que en el archivo index.html modificamos lo siguiente:

```
<title>UFV University</title>
```

Preparamos y comprometemos: `git add . ; git commit -m "Cambio 1 title en index"`

Creamos una nueva rama hotFix: `git checkout -b hotFix`

Volvemos a cambiar el título de index: `<title>Fco. De Vitoria University</title>`

Preparamos y comprometemos: `git add . ; git commit -m "Cambio 2 title en index"`

Nos situamos en la rama master: `git checkout master`

Nuevo cambio en index: `<title>Francisco de Vitoria University</title>`

Preparamos y comprometemos: `git add . ; git commit -m "Cambio 3 title en index"`

Hacemos merge con la rama creada anteriormente: `git merge hotFix`

```
Auto-merging index.html
```

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

```
Automatic merge failed; fix conflicts and then commit the result.
```



Mezclando ramas: Conflictos - Solución

Afortunadamente, git nos proporciona una ayuda para indicarnos qué archivo tiene el conflicto. En nuestro ejemplo debemos volver al IDE donde estamos editando index.html y veremos los cambios tanto de una rama como de la otra:



```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4 <<<<<< HEAD
5 <title>Francisco de Vitoria University</title>
6 =====
7 <title>Fco. De Vitoria University</title>
8 >>>>>> hotFix
```

Commit “Cambio 3” → master

Commit “Cambio 2” → hotFix

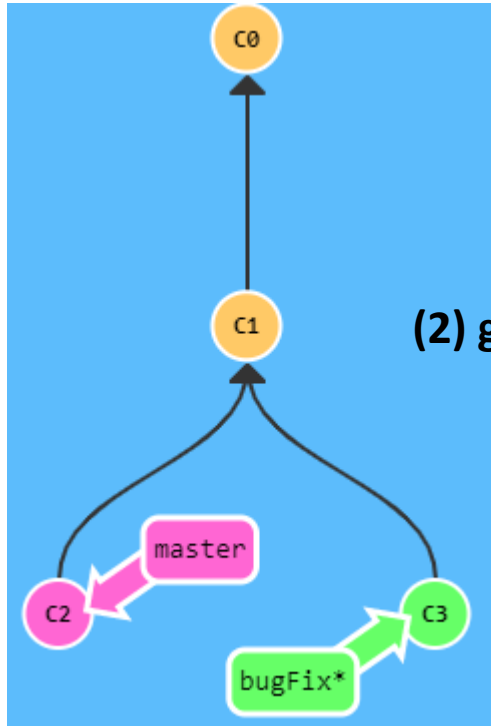
Ahora tenemos que elegir lo que está entre <<<<<< HEAD y ===== que es contenido que tenemos en la rama donde estamos haciendo el merge (master) o ===== y >>>>>> contenido donde están los cambios hechos en la rama que queremos unir (hotFix). Supongamos que modificamos el fichero para quedarnos sólo con “Cambio 3”.

Se resolvería al ejecutar: `git commit -m "Merge con hotFix resuelto"`

Mezclando ramas: "Rebasing"

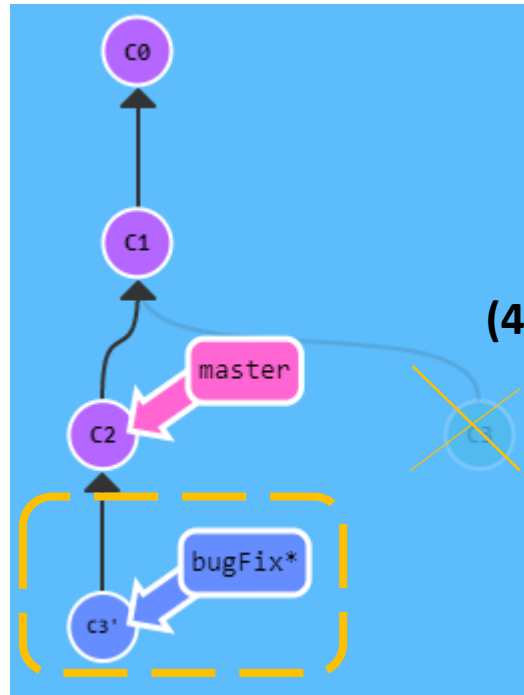
- ✓ Otro modo de combinar el trabajo de distintas ramas es **rebase**. Selecciona un conjunto de commits de una rama, los "copia", y los lleva a otra.
- ✓ La ventaja de hacer rebase es que puede usarse para conseguir una secuencia de commits lineal. El historial / log de commits del repositorio va a estar mucho más claro si sólo usas rebase.

(1) Dos ramas: master y bugFix (activa)



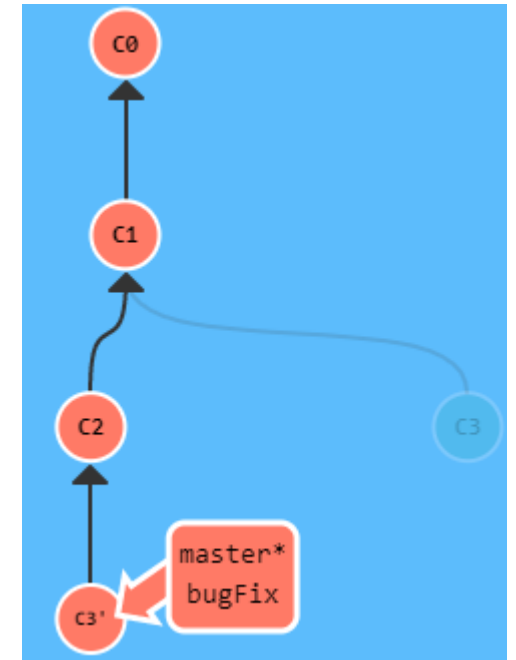
(2) git rebase master

(3) Dos ramas pero con commits lineales



(4) git checkout master
git rebase bugFix

(5) Se podría eliminar bugFix



Mezclando ramas: “Rebasing en acción”

Creamos e inicializamos un nuevo repositorio en nuestro directorio de trabajo

Creamos una nueva rama a partir de master denominada “hotfix”

Modificamos el fichero index.html para que el email del comienzo sea help@ufv.es

(1) Preparamos y comprometemos los cambios.

Volvemos a la rama develop

Modificamos el fichero index.html para que el teléfono del comienzo sea 900 000 000

Preparamos y comprometemos los cambios.

Movemos el commit (1) de hotfix a develop (rebasing)

Movemos develop a Feature2 (rebasing)

Borramos la rama hotfix

```
git checkout -b hotfix
#Modificamos index.html
git add . ; git commit -m "Commit hotfix"
git checkout master
#Modificamos index.html
git add .; git commit -m "Commit master antes de rebasing"
git checkout hotfix # Nos colocamos en la rama desde la que se hará el rebasing
git rebase develop
git checkout master; # Nos colocamos en la rama desde la que se hará el segundo rebasing
git rebase hotfix;
git branch -D hotfix
```

Otra forma de ramificar: Stashing

`git stash` nos provee un almacén temporal para cambios puntuales que no deben afectar a la rama en la que estamos trabajando. Evidentemente, el archivo tiene que estar bajo control de versiones. Es decir, tienes que haber hecho un `git add` sobre él o, lo que es lo mismo, debe estar pendiente de hacer `commit`.

Por otro lado, `git stash` dispone de una serie de herramientas que posibilitan hacer todo tipo de operaciones, operaciones relacionadas con guardar y sacar archivos de este *almacén temporal*:

- `git stash push -m "Mensaje"`, también se puede hacer directamente `git stash`. Pasamos a la rama temporal de stash los cambios que están pendientes de commit en la rama de trabajo, donde tras su ejecución podremos comprobar con `git hist` que no hay nada pendiente. Dejará activa la rama temporal.
- `git stash list`. Nos permitirá ver los cambios que tenemos en el almacén temporal. Los *stashes* se van apilando con la notación `stash@{n}` donde "n" representa un nº incremental de stash. Con "On" se identifica el último. `show stash@{<n>}` Otra opción es `git stash show stash@{<n>}`

```
stash@{0}: WIP on master: 85967ff Mensaje 1
stash@{1}: WIP on master: 976da25 Mensaje 2
stash@{2}: On master: Mensaje 3
```
- `git stash pop stash@{<n>}`. Extrae el elemento n-ésimo de la pila y lo devuelve a la rama de trabajo.
- `git stash apply stash@{<n>}`. Devuelve el elemento n-ésimo de la pila a la rama de trabajo y lo mantiene en stash.
- `git stash drop stash@{<n>}`. Elimina el elemento n-ésimo de la pila
- `git stash clear`. Limpia la pila de cualquier cambio temporal

Moviendo commits entre ramas directamente: cherry-pick

- Borramos nuestro repo desde las utilidades del Sistema operativo e inicializamos uno nuevo con el mismo working directory
- Preparamos y commit.

```
git init; git add .; git commit -m "Primer commit master"
```
- Creamos una nueva rama "hotFix1" a partir de master.

```
git checkout -d hotFix1
```
- Hacemos un cambio en index.html para poner el teléfono +34 91 8880000
- Preparamos y commit

```
git init; git add .; git commit -m "Primer commit hotFix1"
```
- Hacemos otro cambio para sustituir "yoursite.com" por "ufv.es"
- Preparamos y commit

```
git init; git add .; git commit -m "Segundo commit hotFix1"
```
- Creamos una nueva rama "hotFix2" a partir de master.

```
git checkout master; git checkout -d hotFix2
```
- Cambiamos en el fichero teacher.html el primer nombre "Stella Smith" por "Daniela Yates"
- Preparamos y commit

```
git init; git add .; git commit -m "Primer commit hotFix2" #hash1
```
- En el fichero teacher.html cambiamos el segundo nombre "Stella Smith" por "Ashley Gates"
- Preparamos y commit

```
git init; git add .; git commit -m "Segundo commit hotFix2" #hash2
```
- Consultamos todos los commits actuales **y movemos los dos de hotFix2 a partir del último de master**

```
git hist -all; git checkout master; git cherry-pick <hash1> <hash2>
```

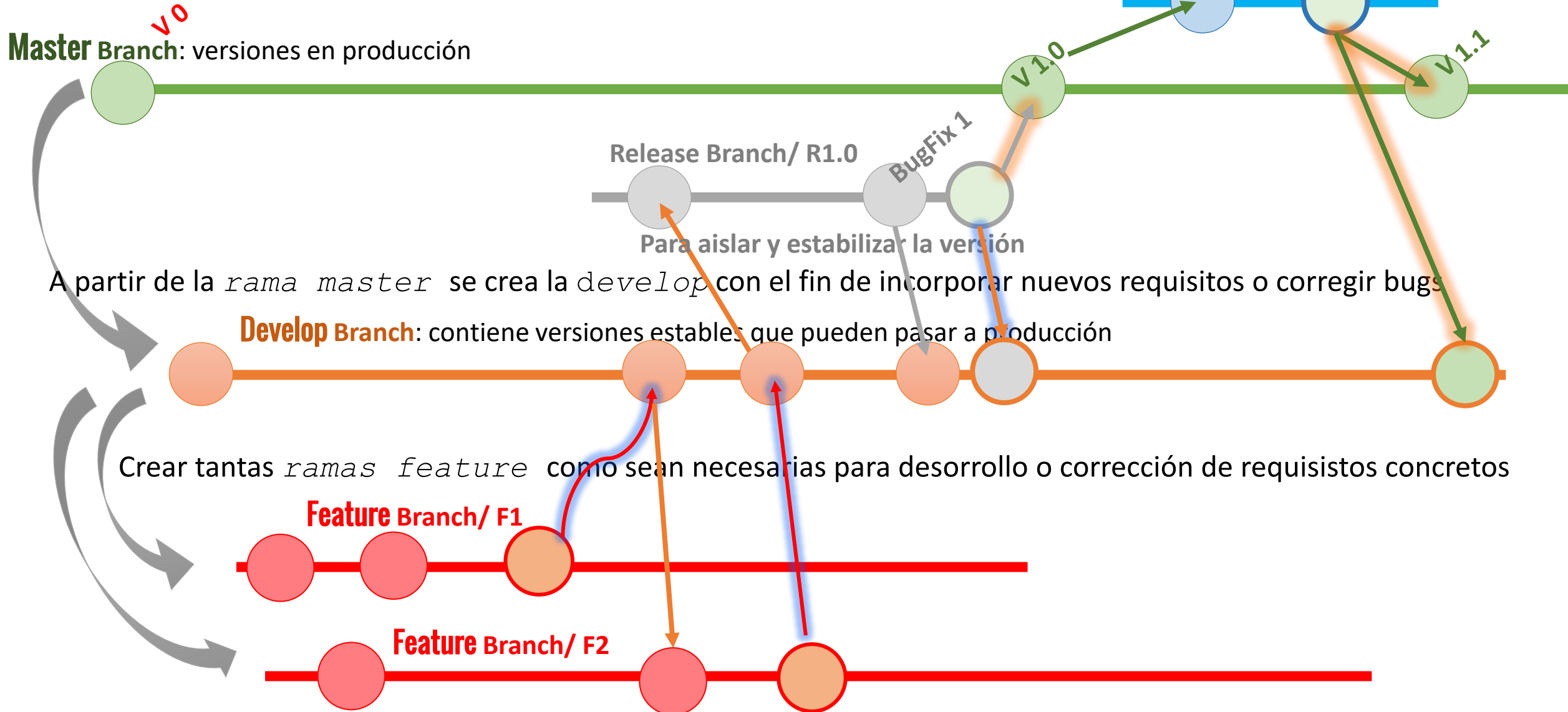
Git Flow. Ideas

- Podemos definirlo como una serie de “reglas” para facilitar el trabajo en equipo.
- El trabajo se organiza en dos ramas principales:
 - **Master**: todos los commits a Master deben estar preparados para subir a producción.
 - **Develop**: contendrá la siguiente versión planificada del proyecto.
- Cada vez que se incorpora código a la rama Master, se genera una nueva versión.

Git Flow. Ideas

- A las dos ramas principales se pueden añadir las siguientes ramas auxiliares:
 - **Feature:**
 - Se originan a partir de la rama develop.
 - Se incorporan siempre a la rama develop.
 - Nombre: cualquiera que no sea master, develop, hotfix-* o release-.*.
 - **Release:**
 - Se originan a partir de la rama develop.
 - Se incorporan a master y develop.
 - Nombre: release-.*.
 - **Hotfix:**
 - Se origina a partir de la rama master.
 - Se incorporan a Master y develop.
 - Nombre: hotfix-.*.

Git Flow: Estructura



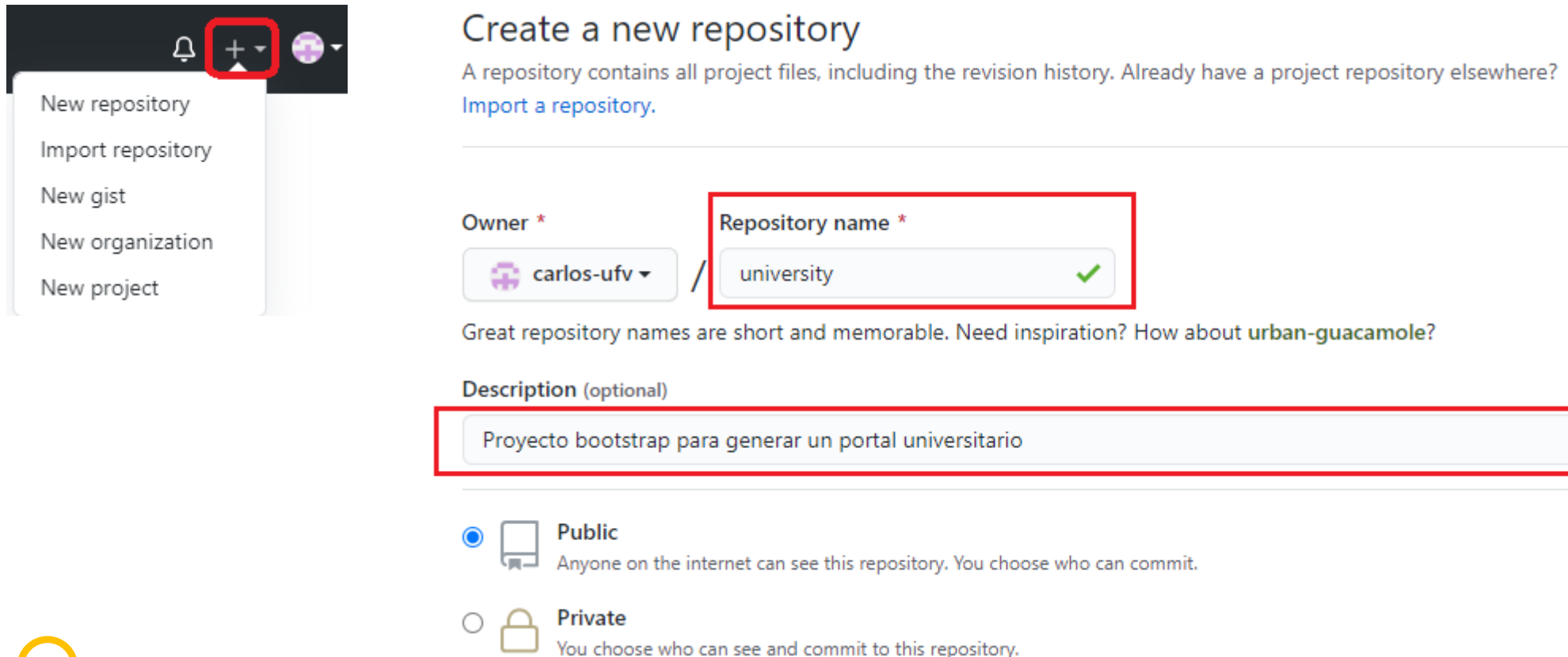
Git: trabajo en remoto. Hacia una solución centralizada y distribuida

git nos permiten mover el trabajo realizado entre dos repositorios cualesquiera sin necesidad de otras herramientas. Sin embargo, en la práctica es más sencillo establecer uno de ellos como repositorio central y tenerlo en la red en lugar de en local. La mayoría de desarrolladores usan servicios de alojamiento en la red, tales como [GitHub](#), [BitBucket](#) o [GitLab](#), para alojar ese repositorio central, que posteriormente se podrá distribuir entre diferentes repositorios. En este caso haremos uso de GitHub



GitHub: empezando

Empecemos por compartir con todos los demás los cambios que hemos realizado en nuestro proyecto actual. Para ello, accedemos a la cuenta de GitHub y hacemos click en el icono que hay en la esquina superior derecha para crear un nuevo repositorio llamado **university**



Create a new repository


A repository contains all project files, including the revision history. Already have a project repository elsewhere?
[Import a repository.](#)


Owner * carlos-ufv / Repository name * university ✓

Great repository names are short and memorable. Need inspiration? How about **urban-guacamole**?

Description (optional)

Proyecto bootstrap para generar un portal universitario

☒  **Public**
Anyone on the internet can see this repository. You choose who can commit.

☐  **Private**
You choose who can see and commit to this repository.



Investiga qué hace la opción **“Add .gitignore”**



GitHub: empezando

Disponemos de una URL e información para configurar nuestro repo local

Quick setup — if you've done this kind of thing before



Set up in Desktop

or

HTTPS

SSH

`https://github.com/carlos-ufv/university.git`

Get started by [creating a new file](#) or [uploading an existing file](#). We recommend every repository include a [README](#), [LICENSE](#), and [.gitignore](#).

...or create a new repository on the command line

```
echo "# university" >> README.md
git init
git add README.md
git commit -m "first commit"
git branch -M main
git remote add origin https://github.com/carlos-ufv/university.git
git push -u origin main
```

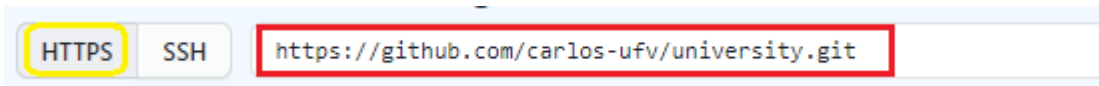
...or push an existing repository from the command line

```
git remote add origin https://github.com/carlos-ufv/university.git
git branch -M main
git push -u origin main
```



GitHub: conectando

Conectar los dos repositorios implica convertir el **repositorio de GitHub en un repositorio remoto del repositorio local**. La página de inicio del repositorio en GitHub incluye la URL para direccionarlo.



Para ello, utilizaremos los siguientes comandos:

```
git remote add origin https://github.com/carlos-ufv/university.git # en la URL debe aparece tu usuario
```

```
git remote -v # Verificamos conexión correcta. Origin es la forma de identificar el repo remoto
```

```
origin https://github.com/carlos-ufv/university.git (push)
```

```
origin https://github.com/carlso-ufv/university.git (fetch)
```

El nombre **origin** es un alias local para tu repositorio remoto. Se puede usar cualquier otro nombre si se desea, pero **origin** es la elección más habitual.

Un mismo repositorio local puede estar conectado a varios remotos, para cambiar entre los remotos:

```
git remote set-url origin https://github.org/repo.git
```



GitHub: push & pull

El siguiente paso es enviar los cambios realizados en nuestro repositorio local al repositorio en GitHub:

`git push origin master` # También se podría haber usado `git push -u origin main` (permitirá usar pull sin argumentos)

```
$ git push origin master
git: 'credential-cache' is not a git command. See 'git --help'.
Enumerating objects: 226, done.
Counting objects: 100% (226/226), done.
Delta compression using up to 4 threads
Compressing objects: 100% (221/221), done.
Writing objects: 100% (226/226), 6.38 MiB | 2.69 MiB/s, done.
Total 226 (delta 22), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (22/22), done.
To https://github.com/carlos-ufv/university.git
```

Observa que se solicita autenticación ante el servidor de GitHub, para evitarlo puedes configurar tu entorno local de la siguiente forma:
`git config remote.origin.url https://<USERNAME>:<PASSWORD>@github.com/<USERNAME>/<REPO_NAME>.git`

O bien `unset SSH_ASKPASS`

HTTPS

SSH

`https://github.com/carlos-ufv/university.git`

También podemos hacer la operación inversa, traer los cambios de GitHub a nuestro repo local:

`git pull origin master`

```
From https://github.com/carlos-ufv/university
* branch      master      -> FETCH_HEAD
Already up to date.
```

Observa que permanentemente aparece el mensaje `"git: 'credential-cache' is not a git command. See 'git --help'."`. Para evitarlo comenta las siguientes líneas de `.gitconfig`

[credential]

helper = cache



GitHub: en acción


Modifica el título de courses.html para que muestre “<title>Francisco de Vitoria University </title>”

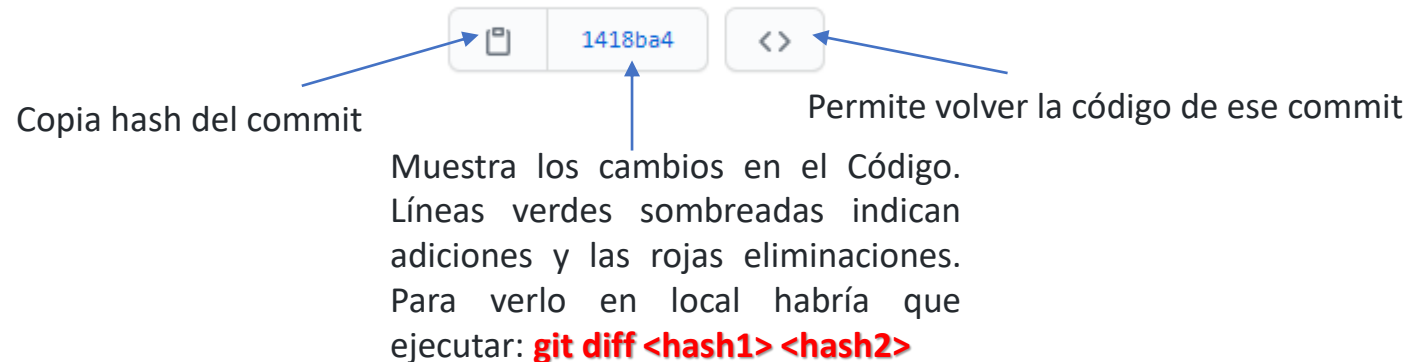
Prepara y confirma cambios en local

```
git add .; git commit -m "Commit local/github"
```

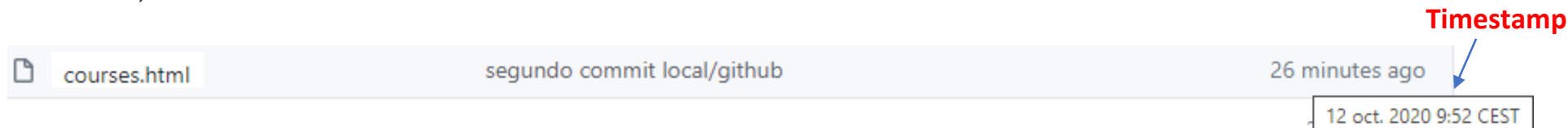
Sube los cambios a GitHub

```
git push origin master
```

Navega hasta tu repositorio [university](#) en GitHub. En la pestaña **Code**, localiza el texto  **XXX commits** (donde “XX” es algún número) y haz click en él. Mueve el cursor sobre los tres botones que hay a la derecha de cada **commit** para determinar qué hace cada uno



En la pestaña **Code**, observa cómo se muestra la información tras las modificaciones introducidas del último commit.





GitHub: trabajo colaborativo

Para trabajar de forma colaborativa debemos diferenciar dos perfiles: dueño del repositorio y colaboradores. El dueño puede añadir diversos colaboradores accediendo a “[Settings](#)” / “[Manage Access](#)” y presiona el botón “[Invite a collaborator](#)”.

The screenshot shows the GitHub repository settings page. The top navigation bar includes links for Code, Issues, Pull requests, Actions, Projects, Wiki, Security, Insights, and Settings (highlighted with a red circle). On the left sidebar, the 'Manage access' option is highlighted with a red arrow. The main content area is titled 'Who has access' and shows two sections: 'PUBLIC REPOSITORY' and 'DIRECT ACCESS'. The 'DIRECT ACCESS' section indicates that 1 person has access to the repository and 1 invitation is pending. A red arrow points to the 'Invite a collaborator' button. Below this, the 'Manage access' section shows a list of users with a 'Select all' checkbox. At the bottom, a notification from 'carlos-ufv' states 'invited you to collaborate' with 'Accept invitation' and 'Decline' buttons. A list of permissions is also shown, including public profile information, repository activity, country of request origin, access level, and IP address.

El colaborador deberá acceder a su GitHub y aceptar la invitación



GitHub: clonando repositorios

Para que el colaborador pueda trabajar de forma colaborativa es habitual que clone el repositorio en su máquina local, para ello accederá a su interfaz de comandos git para ejecutar el siguiente comando en el subdirectorio de destino local:

```
git clone https://github.com/repo.git <subdirectorio_local>
```

Por ejem: `git clone https://github.com/carlos-ufv/university.git /GitProjects/UniversityClone`

El colaborador pueda realizar los cambios localmente tal y como se viene explicando en los puntos anteriores. Regularmente se enviarán los cambios hacia el *repositorio del dueño* en GitHub haciendo **push**:

```
git push origin master
```

Nota: **no** es necesario crear un directorio remoto llamado *origin*. Git utiliza este nombre de manera automática cuando clonamos un repositorio. El repositorio local quedará configurado e inicializado conforme al contenido de .git

```
$ git push origin master
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 4 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 339 bytes | 339.00 KiB/s, done.
Total 3 (delta 2), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To https://github.com/carlos-ufv/university.git
1418ba4..a4c0de7 master -> master
```



GitHub: actualizando repo con colaboradores

El dueño del repositorio podrá ver en GitHub los cambios incorporados por el colaborador

carlos-ufv Primer commit master f4001ba 4 hours ago 1 commits		
css	Primer commit master	4 hours ago
fonts	Primer commit master	4 hours ago
images	Primer commit master	4 hours ago
js	Primer commit master	4 hours ago
scss	Primer commit master	4 hours ago
about.html	Commit colaborador local/github carlos-ufv	15 minutes ago

Para que el dueño integre el código en su repo debe descargar los cambios hechos por el colaborador en GitHub:

```
git pull origin master
```

```
$ git pull origin master
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (1/1), done.
remote: Total 3 (delta 2), reused 3 (delta 2), pack-reused 0
Unpacking objects: 100% (3/3), 319 bytes | 2.00 KiB/s, done.
From https://github.com/carlos-ufv/university
* branch      master      -> FETCH_HEAD
1418ba4..a4c0de7 master    -> origin/master
Updating 1418ba4..a4c0de7
Fast-forward
 about.html | 2 +-
1 file changed, 1 insertion(+), 1 deletion(-)
```

Ahora hay tres repositorios sincronizados (el local del dueño, el local del colaborador y el GitHub del dueño)



GitHub: flujo colaborativo

Un flujo de trabajo colaborativo básico

Una buena práctica es aquella que consiste en verificar habitualmente que el repo de trabajo tiene una versión actualizada del repositorio con el que se colabora. Es conveniente hacer un `git pull` antes de hacer cambios. El enfoque sería:

1. Actualizar el repositorio local `git pull origin master`
2. Realizar localmente los cambios de código que sean necesarios.
3. Añadir cambios localmente: `git add`,
4. Confirmar cambios localmente: `git commit -m`.
5. Cargar las actualizaciones a GitHub con `git push origin master`

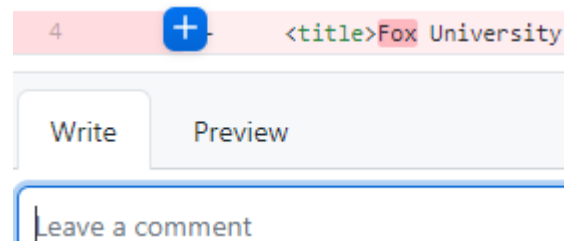


Es mejor hacer varias actualizaciones pequeñas que un commit grande con cambios importantes de código.

Commits pequeños son más fáciles de leer y revisar. (Trazabilidad)



Investiga cómo el colaborador puede enviar comentarios al dueño sobre las líneas de código que haya modificado el propio dueño.





GitHub: conocer cambios (fetch)

Es bastante habitual que el dueño del repositorio vaya realizando *push* de los *commits* sin informar a los colaboradores.

En este caso, el colaborador puede realizar un **fetch** desde la línea de comandos para acceder a los cambios remotos en el repositorio local, sin hacer un **merge**, y posteriormente comparar ambos repos mediante **diff**.

```
git fetch origin master  
git diff master origin/master
```

Diferencia entre fetch y pull

De manera simplificada podemos decir que **git pull** hace un **git fetch** seguido de una **git merge**.

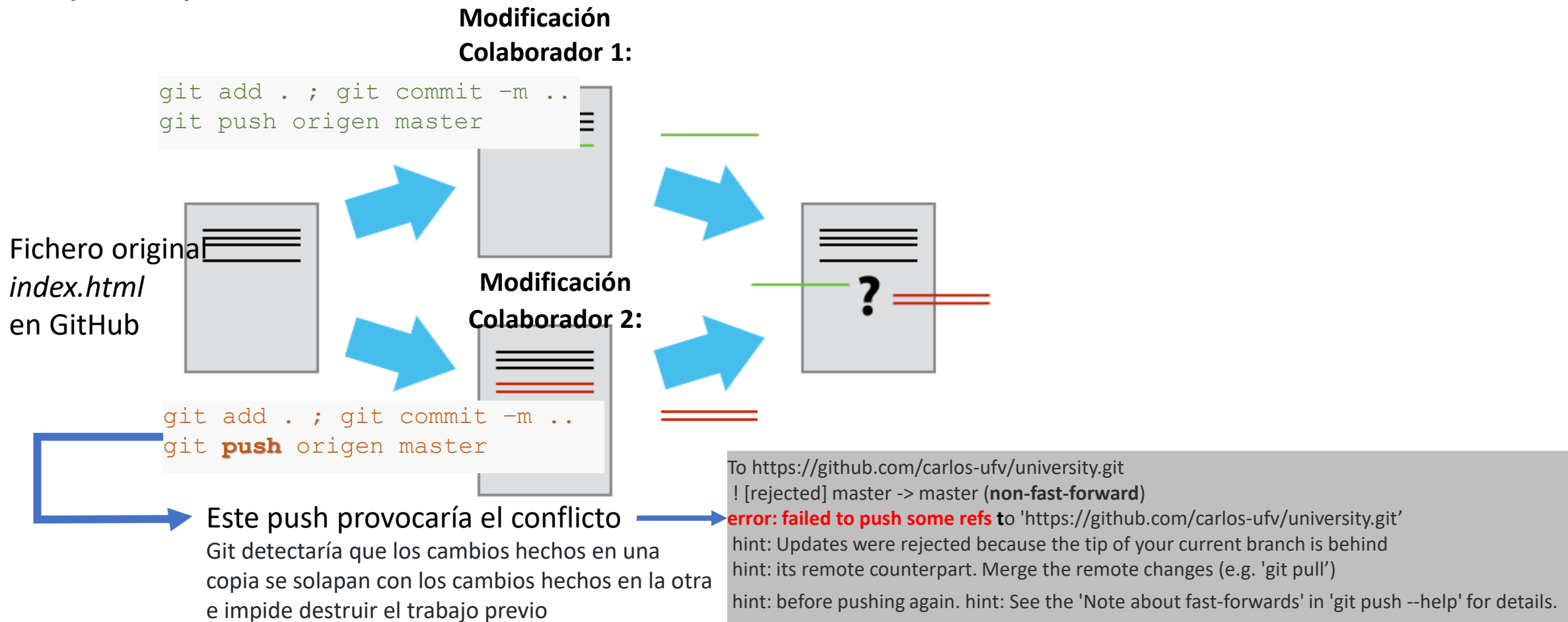
git fetch identifica los metadatos del repo remoto, a continuación comprueba si existen cambios del repositorio local frente a los metadatos del remoto (sin descargar nada), es decir, comprueba cambios de una forma meramente informativa.

En cambio **git pull** sincroniza el repositorio original con tu repositorio local transfiriendo los meta datos y los archivos si es que existiese algún cambio.



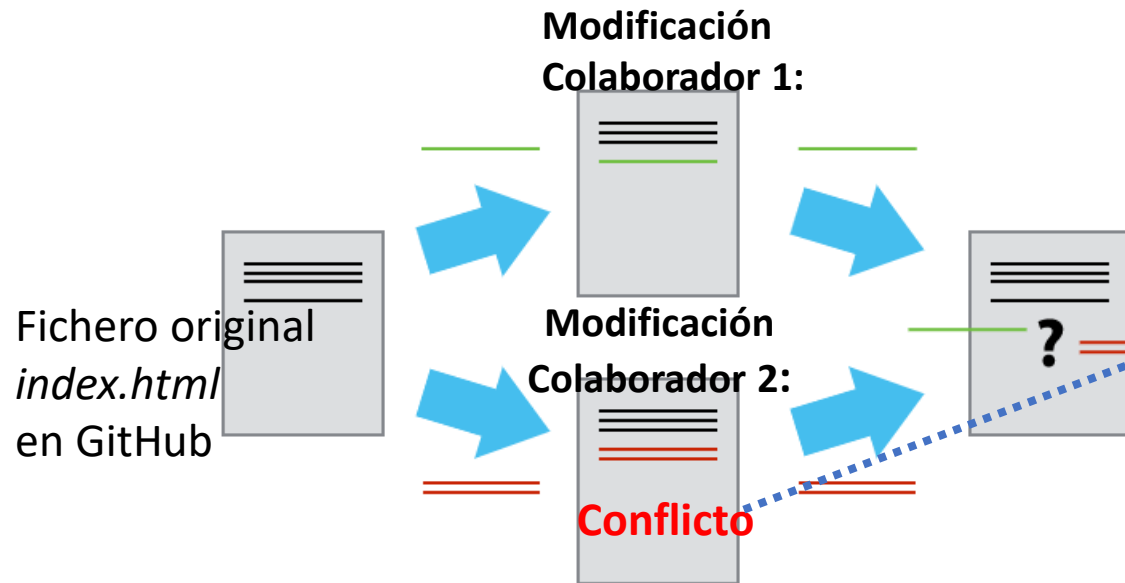
GitHub: conflictos

Tal y como vimos en el uso de git en local, el control de versiones nos ayuda a manejar los conflictos también cuando estamos en un entorno distribuido. El procedimiento esencialmente es semejante al ya [visto](#). Veamos el caso habitual de **situación conflictiva**:





GitHub: conflictos - solución



1. El *colaborador 2* deberá traer los cambios desde GitHub y unirlos a la rama local:

```
git pull origin master
```

- Y nos informa del conflicto

```
From https://github.com/carlos-ufv/university.git
* branch master -> FETCH_HEAD
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

2. El fichero en el que se observa el conflicto (*index.html*) tiene marcados los cambios que colisionan:

```
<<<<<< HEAD
We added a different line in the other copy
=====
This line added to colaborator2's copy
>>>>>> dabb4c8c450e8475aee9b14b4383acc99f42af1d
```

Contenido
descargado de
GitHub

3. Editar *index.html* y quedarse con el código que se considere válido o añadir otro nuevo. Grabar los cambios y actualizar:

```
git add . ; git commit -m "Resuelto conflicto index.html colab. 1/2"
git push origin master
```



GitHub: conflictos - buenas prácticas

La habilidad de Git de identificar conflictos es muy útil, pero su resolución **cuesta tiempo y esfuerzo**, y puede introducir errores si los conflictos no son resueltos correctamente.

Buenas prácticas :

- A. Hacer **pull** a pequeños intervalos, especialmente antes de empezar una nueva tarea
- B. Usar ramas temáticas para separar cada trabajo, uniéndolas a la rama principal - **master**- cuando finalicen (modelo [GitFlow](#))
- C. Hacer commits más cortos y concisos
- D. Siempre que sea posible, dividir archivos grandes en varios pequeños para reducir la probabilidad de modificaciones concurrentes.
- E. Los conflictos también pueden ser minimizados con **estrategias de gestión de proyectos**, como:
 - i. Definir con claridad qué colaboradores son responsables de cada tarea.
 - ii. Acordar con el equipo de colaboradores en qué orden han de realizarse las tareas en las que alguna/s puede depender de otra/s.
 - iii. Establecer convenciones a la hora de programar y utilizar nombres del programador, si fuera necesario, utilizar herramientas de estilo de código como [htmltidy](#), [perltidy](#), [rubocop](#), etc.



GitHub: Proyectos bifurcados (forked)

- Se trata de hacer copias de proyectos en un repositorio propio y plenamente independiente del repositorio original.
- Los cambios que se hacen en el repositorio original no se transmiten automáticamente a la copia (fork). Esto tampoco ocurre a la inversa.
- Es otra forma de colaboración en el proyecto pero sin haber sido explícitamente nombrado como colaborador del mismo (sin usar la opción “[Invite a collaborator](#)”).
- Los dueños del repositorio son los que aceptan o rechazan las modificaciones propuestas a través de “**Pull Requests**”.
- Cuando se trabaja con proyectos *forked* la estructura de repositorios se hace más compleja e incluye los siguientes:
 1. Repositorio **Remoto** (original)
 2. Repositorio **Bifurcado** (copia del anterior)
 3. Repositorio **Local** (clonado del anterior donde se incorporan las modificaciones o aportaciones que posteriormente se subirán al bifurcado)

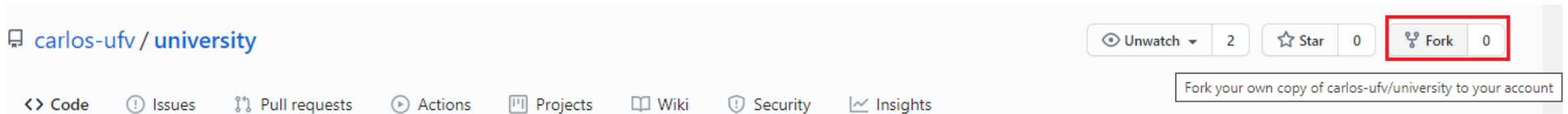
Diferencia entre fork y clone:

Al hacer un **clone** de un repositorio, se obtiene una copia del mismo en la máquina local. El *push* de las modificaciones locales se hace directamente sobre el repositorio clonado.

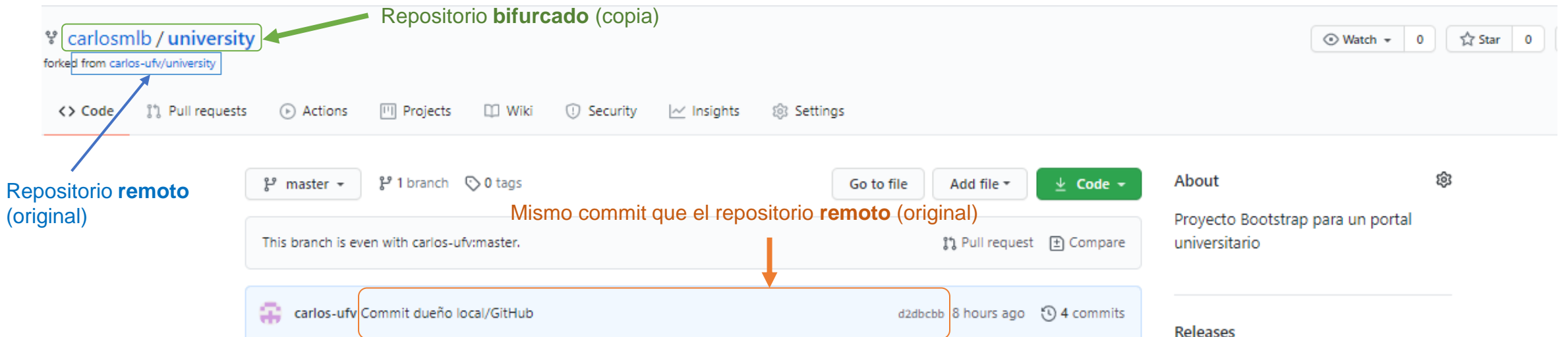
El **fork** crea un nuevo repositorio en la cuenta de Github con una URL diferente que se ha de clonar en la máquina local. El *push* modifica el repositorio *forked* manteniendo intacto el original.

GitHub: Creando el repo bifurcado

- Localizar en GitHub el proyecto que se quiere bifurcar. (Utiliza como ejemplo carlos-ufv/university)
- Crear la bifurcación:



- El usuario con el que hemos hecho el fork tendrá en su cuenta este nuevo repositorio





GitHub: Clonando nuestro repositorio forked

A - Clonamos el repositorio **bifurcado** a un nuevo repositorio **local** que ubicamos en `GitProjects/university_forked`

```
git clone https://github.com/carlosmlb/university.git university_forked
```

B - Conectamos el repositorio **local** con el repositorio **remoto** (original)

```
git remote add upstream https://github.com/carlos-ufv/university.git
```

De esta forma estamos indicando a git que agregue la **ubicación remota** a la que estamos asociando el alias **upstream**. Es conveniente mantener este nombre, aunque se puede indicar cualquier otro.

C - Actualizar periódicamente el repositorio local desde el remoto (original)

```
git fetch upstream
```

D - Incorporar los cambios en nuestra rama **master local**

```
git checkout master  
git merge upstream/master
```

También se podría haber utilizado *rabase*: `git rebase upstream/master`



GitHub: Aportando cambios al repo original

Pull request

A – Hacemos los cambios de código sobre los ficheros almacenados en nuestro working directory del repositorio local

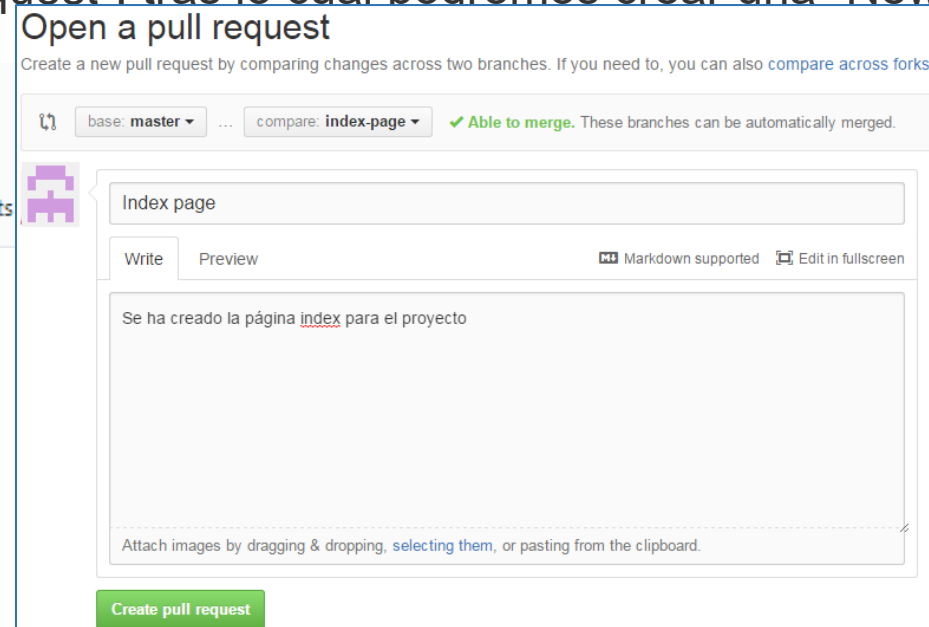
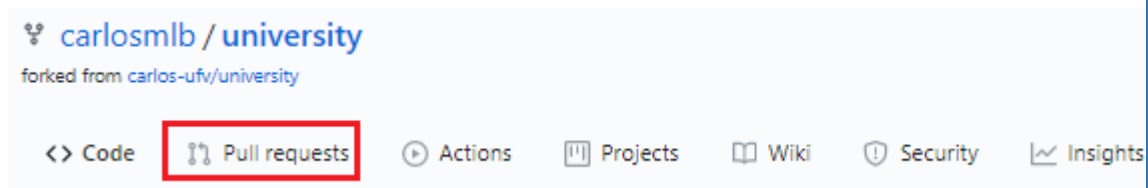
B – Añadimos y confirmamos los cambios en el repositorio **local**

```
git add . ; git commit -m "Cambios para pull request"
```

C – Subimos los cambios desde el repositorio local al repositorio **bifurcado**

```
git push origin master
```

D – Creamos el **pull request** para informar al dueño del repositorio remoto de la propuesta. Accedemos a la URL de nuestro proyecto forked y a la opción “Pull request”, tras lo cual podremos crear una “New Pull Request”





GitHub: Aportando cambios al repo original

Pull request II

Información para lanzar la pull request

Comparing changes

Choose two branches to see what's changed or to start a new pull request. If you need to, you can also [compare across forks](#).

base repository: angelmones/AJMBubbleVi... base: master + head repository: AMonalesK/AJMBubbleView compare: master

✓ Able to merge. These branches can be automatically merged.

Create pull request Discuss and review the changes in this comparison with others.

1 commit 1 file changed 0 commit comments 1 contributor

Commits on Aug 17, 2019

Declarando funciones y propiedades privadas en SampleViewController b4ec0f1

Showing 1 changed file with 7 additions and 7 deletions.

Unified Split

14 AJMBubbleView/SampleViewController.swift

```
@@ -8,18 +8,18 @@
8
9
10
11 - class SampleViewController: UIViewController {
11 + final class SampleViewController: UIViewController {
```

Repositorios y ramas, tanto del remoto (base rep.), como del bifurcado (head rep.)

Estadísticas de nuestra propuesta de cambio como:

- Nº de commits.
- Ficheros modificados
- Nº de comentarios...

Y además, el listado de commits realizados

Cambios realizados en el código

Una vez verificada toda la información se activa **Create pull request** y aparecerá un formulario para indicar el título y la descripción del cambio. Toda esta información la podrá ver el dueño del repositorio original en su ventana de *pulls requests*. De manera iterativa otros programadores podrán añadir otros comentarios a la pull request

GitHub: Aprobando los cambios *Pull request III*

El dueño del repositorio remoto (original) podrá visualizar todas las pull requests enviadas

carlos-ufv / university

<> Code

Issues 5

Pull requests 24

Actions

Projects 8

Wiki

Settings

El proceso de revisión puede requerir aclaraciones adicionales. Pulsando sobre el código modificado por la *pull request* el dueño del repositorio remoto puede introducir *reviews*, comentarios o preguntas antes de la aprobación definitiva.

Una vez seleccionada y revisada una *pull request* podrá ser aprobada pulsando en el botón «Merge pull request», tras lo cual:




This branch is up-to-date with the base branch
Merging can be performed automatically.

 **Merge pull request** You can also open this in GitHub for Windows or view [command line instructions](#).


Index page



Merged merged 2 commits into master from index-page 15 seconds ago


Conversation 0 Commits 2 Files changed 1

 commented 5 minutes ago Owner

Se ha creado la página index para el proyecto

 added some commits 14 minutes ago

-  página de inicio de la aplicación 66f26c5
-  Merge branch 'master' of https://github.com/jeffer8a/StydeNet-pull-re... b33637d

 merged commit 96e2423 into master 14 seconds ago Revert

Pull request successfully merged and closed Delete branch

You're all set—the index-page branch can be safely deleted.

Write Preview Markdown supported Edit in fullscreen

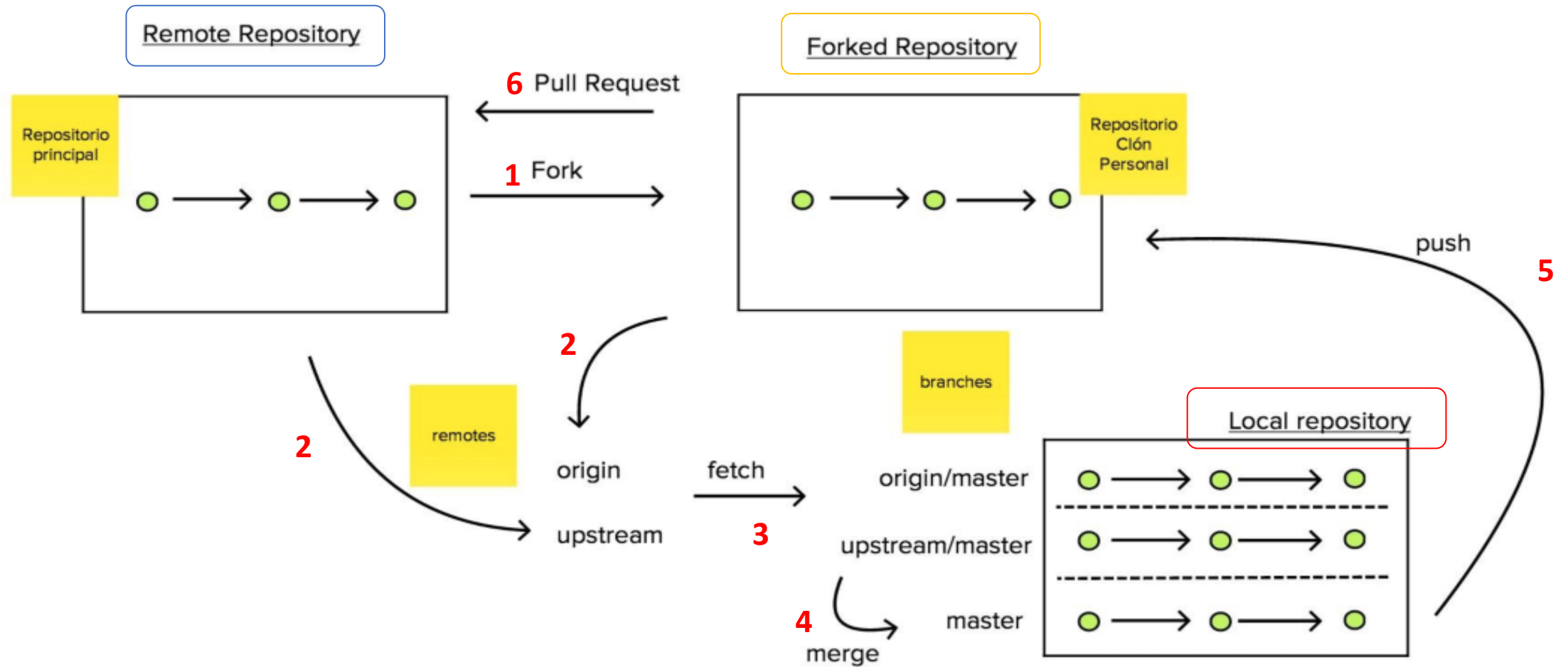
Leave a comment

Attach images by dragging & dropping, selecting them, or pasting from the clipboard.

Comment



GitHub: Forks y *Pull requests*. Visión global



Fin git & GitHub