

Git y más

Información

- Fuentes de información: YouTube (freeCodeCamp), git-scm.com/docs, [github.https://docs.github.com/es](https://docs.github.com/es).
- Autor: M.Sc. Lorenzo Antonio Pérez Carballo
- Fecha: 9/02/2023

Consejo de estudio

- (Novatos) Primeramente ver videos en YouTube, preferentemente videos largos (5h+). De uno a dos videos, así se tiene una noción de lo que es Git.
- (-Novato) Ver la documentación de git (digamos que casi toda).
- (Aprendiz) Ver la documentación de GitHub.

Para pegarlo en la nevera

<code>git log --graph --decorate --abbrev-commit --all -pretty=oneline</code>

<code>git log --oneline --decorate --graph -all</code>
--

git branch	
-d <rama>	Elimina la rama si esta ya fue fusionada.
-D <rama>	Elimina cualquier rama.
-v	Muestra las ultimas confirmaciones todas las ramas.
-vv	Lista las ramas locales con más información, incluida a que sigue cada rama.
--merged (--no-merged)	Ramas que (no) han sido fusionadas.
-m <new-nombre-rama> -m <rama-vieja> <new-nombre-rama>	Modifica el nombre de la rama.

git clone	
-o <nombre>	Establece el nombre del repo remoto.

Git

Tratando de convencerme

Motivos por los cuales Git es una buena opción:

- Git es **rápido** y es una implementación moderna de un controlador de versiones.
- Git proporciona un **historial de los cambios** de contenido.
- Git facilita los **cambios colectivos** de archivos.
- Git es **fácil de usar** para todo tipo de trabajo intelectual.
- **Git Local** es distribuido para que la conectividad no bloquee el trabajo.
- **Git Local** es fácil para que el aprendizaje de sus comandos pueda realizarse progresivamente.
- **Git Distribuido** es centrado en el equipo para que la colaboración sea natural.
- **Git Personalizable** es flexible para que se adapte a sus necesidades y no al revés.
- **Git Personalizable** es potente para satisfacer las necesidades de scripting de los desarrolladores avanzados.

Rompiendo el hielo

Ejemplo de un código local (una persona)

```
$ git init mi-proyecto
$ cd mi-proyecto
$ git add .
$ git commit -m"Importar todo el código"
```

Se ha creado un proyecto (a efectos reales se ha creado un directorio/carpeta que está siendo monitorizada por Git).

Nos movemos dentro del directorio.

Se dice que Git que monitorice todo lo que hay dentro de la carpeta (OJO, si se agregan nuevos archivos, se cambia el nombre a los existentes, nuevos directorios y demás, debemos volver a ejecutar dicha línea de código).

Se llevan los cambios al “History Tracking” – Seguimiento del Historial (HT), o como lo diríamos los mortales, es como si se le hiciera una radiografía (foto) a dicho directorio y guardáramos esto para ir confeccionando el HT.

Ejemplo de un código local (dos personas Eduardo & Miguel)

```
Ed $ git checkout master
Ed $ git commit -m"Mi nuevo logo"
Ed $ git push

Mg $ git checkout -b linkin
Mg $ git commit -m"Mi código"
Mg $ git push origin linkin
```

```
Ed $ git pull
```

(Eduardo) Se mueve a la rama (branch) *master*. Agrega un cambio al HT. Enviá los cambios al Repositorio Remoto (RR).

(Miguel) Crea (-b) una rama llamada *linkin* y se mueve directamente a dicha rama. Agrega un cambio. Sube los cambios al RR.

(Eduardo) Trae la rama de Miguel al RR. Mezcla la rama de Miguel con la rama principal.

Lo primero es tenerlo

Git se puede descargar desde su sitio oficial: [Git - Downloads \(git-scm.com\)](https://git-scm.com/downloads). El proceso de instalación se ve un poco complejo pero es aconsejable (si se está comenzando) dejar todos los parámetros por defecto y no agregar nuevas funcionalidades a no ser que se aconsejen.

Para instalarlo en Linux se utiliza la herramienta básica de administración de paquetes que trae la distribución, si es **Fedora**, se puede utilizar la siguiente línea:

```
yum install git
```

Si se está en una basada en **Debian**, pues

```
apt-get install git
```

Luego se deben introducir las credenciales del usuario ya que estas se registrarían en cada evento. Esto se logra con las dos líneas de código siguiente

```
$ git config --global user.name "Lorenzo"
$ git config --global user.email *****@ejemplo.com
$ git config --global core.editor <editor>

# para ver el valor de una clave
$ git config <key>

# Para ver todas los valores de todas las claves
$ git config --list
```

Si se ven valores repetidos la utilizar --list es debido a que Git lee todos los archivos de configuración, no solo el global.

Conceptos

Control de Versiones

Version Control – Sistema que **registra** los cambios realizados en un archivo o conjunto de archivos a lo largo del tiempo, de modo que se pueda recuperar versiones específicas más adelante. Esto se puede hacer con casi todo tipo de archivo.

VCS – Version Control System

Ver la figura 1

Git ~ Fundamentos ~

> *¿Cómo almacena los datos?* Como un conjunto de copias instantáneas de un sistema de archivos miniatura o como una secuencia de copias instantáneas.

> *Casi todas las operaciones son locales*, debido a que tiene toda la historia del proyecto ahí mismo.

> *Git tiene integridad* – Esto significa que es imposible cambiar los contenidos de cualquier archivo o directorio sin que Git lo sepa.

> *Git generalmente solo añade información* – Esto hace que usar Git sea un placer porque sabemos que podemos experimentar sin peligro de estropear las cosas ya que podemos recuperar datos aparentemente perdidos.

*> *Los tres estados* – Los archivos en Git se pueden encontrar en tres estados (lo cual implica tres asar de trabajo – *figura 2*):

>> confirmado (**committed**) :: Los datos están almacenados de manera segura en la base de datos local.

>> modificado (**modified**) :: Los datos han sido modificados y no se han confirmado a la base de datos local.

>> preparado (**staged**) :: se ha marcado un archivo modificado a su versión actual para que vaya en tu próxima confirmación.

Ver la figura 2

Línea de Comandos

La forma tradicional de utilizar Git es a traves de la línea de comandos ya que desde ella podemos utilizar **todos** los comandos Git.

Configuración

Para obtener o establecer las variables de configuración que controlan el aspecto y el funcionamiento de Git, tenemos la herramienta: **git config**. Dicha configuración se almacena en tres lugares y los niveles sobrescriben los niveles anteriores, o sea, que el nivel 3 tiene prioridad por encima del nivel 1:

1- */etc/gitconfig* :: contiene valores para todos los usuarios del sistema y todos sus repositorios, para leerlo o escribirlo se utiliza **git config -system**.

2- *~/.gitconfig* o *~/.config/git/config* :: es específico de tu usuario, para leerlo o escribirlo se utiliza **git config -global**.

3- *.git/config* :: es específico del repositorio actual.

Preguntas – Bloque I

¿Cómo obtengo ayuda?

```
$ git help <termino-a-buscar>
```

¿Cómo inicializo un nuevo repositorio – repo?

```
$ git init Mi-proyecto  
$ cd Mi-proyecto
```

Un repo es (dicho de manera bruta) una carpeta/directorio en la PC (o en un RR), por lo que con el código anterior se ha creado una carpeta (la cual será un futuro proyecto) y se entra en dicha carpeta.

Una forma alternativa de inicializar un repo es crear directamente la carpeta (en el sistema operativo) y dentro de dicha carpeta abrir Git Bash. Y con la ruta del Bash en dicho directorio solo se escribe: **git init**.

Luego si se desea comenzar a controlar versiones de archivos en dicho repo (esto no aplica para repos vacíos), se deben comenzar el seguimiento de archivos (add) y hacer una confirmación inicial (commit). En caso de hacer esto, ya tendríamos un repo de Git con archivos bajo seguimiento y una confirmación inicial.

¿Cómo clono un repositorio existente?

Para obtener una copia de un repo Git existente (por ejemplo, un proyecto en el que te gustaría contribuir) el comando que se necesita es **git clone [url]**. Por ejemplo, si se quiere clonar el repo en donde se encuentra este resumen

```
$ git clone https://github.com/lpcarballo/lanota
```

Esto creará un directorio llamado “lanota”, inicializa un directorio `.git` en su interior, descarga toda la información de ese repositorio y saca una copia de trabajo de la última versión. Si se quiere clonar el repositorio pero que el directorio no tenga el nombre “lanota” se puede especificar

```
$ git clone https://github.com/lpcarballo/lanota tu-lanota
```

Git permite utilizar diferentes protocolos de transferencia: https, git, SSH. Cuando clonas un repo todos los archivos estarán rastreados y sin modificar.

¿Cómo reviso el estado de los archivos?

```
$ git status
```

Cuando ejecutamos este código podemos obtener:

- **On branch main ... working directory clean** (lo cual nos indica que el directorio está vacío y no hay archivos para monitoriar).
- **On branch main, Untracked file** (y una lista de archivos que no se están monitorizando o rastreando, o sea, que Git ve archivos que no estaban en el commit anterior).

- **On branch main, Changes not staged for commit** (archivos que no se han agregado al área de preparados).
- **On branch main, Changes to be committed** (archivos que están listos para ser confirmados, o sea, archivos en el área de preparados).

La salida de `git status` es bastante explícita y extensa, pero esto puede cambiar si le agregamos la opción `-s` o `--short`. Con esta opción activa la salida es mucho más simplificada. A dicha salida se le conoce “Estado Abreviado”. En la salida los nuevos archivos que no están rastreados tiene un `??` a su lado, los preparados tiene una `A` y los modificados una `M`.

¿Cómo ignoro archivos?

Muchas veces queremos que Git ignore ciertos archivos. Un ejemplo suele ser los archivos generados automáticamente como trazas o archivos creados por el sistema de compilación. Para lograrlo tenemos que crear un archivo en la raíz del proyecto con el nombre `.gitignore` y dentro con los nombres de los archivos que queremos que ignore git.

También podemos crear patrones dentro del archivo así podemos ignorar conjuntos de archivos [<https://git-scm.com/docs/gitignore>].

¿Cómo rastreo archivos nuevos?

```
$ git add <archivo>
```

Una vez que el archivo se ha comenzado a rastrear está listo para ser confirmado (commit). Luego si se vuelve a revisar el estado de los archivos, el archivo saldrá en la lista: **Changes to be committed: ... new file: <archivo>**.

El comando `git add` puede recibir tanto un archivo como un directorio, lo cual hace que añada los archivos que se encuentran dentro.

¿Cómo preparo archivos modificados?

Tenemos un archivo que esta siendo rastreado al cual le realizamos alguna modificación y luego rastreamos el estado del archivo. El estado del archivo devolverá que pertenece a **Changes not staged for commit: ... modified: <archivo>**. Esto dice que el archivo rastreado ha sido modificado en el directorio de trabajo pero aún no está preparado. Para prepararlo se ejecuta el código siguiente

```
$ git add <archivo>
```

Se utiliza el mismo código para rastrear un nuevo archivo, preparar un nuevo archivo y para hacer otras cosas *como marcar archivos en conflicto por combinación como resultado*. Es útil ver dicho comando para “añadir este contenido a la próxima confirmación”.

Todavía puede pasar algo, imaginemos que se vuelve a modificar dicho archivo una vez que lo rastreamos. Si aplicamos `status`, el mismo archivo sale como listo para confirmar y como modificado ¿qué debemos hacer? Pues si confirmamos en este momento se enviará la versión del archivo antes de la última modificación y no la que se tiene el directorio. Por tanto si se desea enviar la versión actual se debe rastrear de nuevo dicho archivo.

Existe una herramienta para comparar lo que tienes en tu directorio de trabajo con lo que está en el área de preparación. La herramienta se llama **git diff**. También se pueden ver con otra herramienta llamada **git difftool**, si ejecutamos **git difftool --tool-help** se pueden ver que tenemos disponible en nuestro sistema.

¿Cómo confirmo los cambios?

Suponiendo que el área de preparación se encuentra como queremos, entonces ya podemos confirmar los cambios. Recordar los archivos que no han sido preparados no serán confirmados. Digamos que la última vez que se ejecutamos **git status** verificamos que todo estaba preparado y que estamos listos para confirmar los cambios. La forma más simple para hacerlo es

```
$ git commit
```

Luego, arranca el editor por defecto con un mensaje que muestra la última salida del **git status** comentada y una línea vacía encima. Este comentario nos ayuda a escribir el “*mensaje de la confirmación*”, en caso de querer más detalles se le puede agregar al commit la opción **-v**. Una vez creada la confirmación se podemos ver una salida descriptiva: rama que se ha confirmado, *checksum* SHA-1 del commit, cuántos archivos cambiamos y estadística sobre las líneas añadidas y eliminadas.

Recordar que cada vez que realizamos un commit, guardamos una instancia del proyecto a la cual podemos usar para comparar o volver a ella luego.

¿Cómo elimino archivos?

Para eliminar archivos de Git, debemos eliminarlos del área de preparación, o sea, dejar de rastrearlos y luego confirmar. Primero borramos el archivo simplemente y luego verificamos el estado de los archivos. En el estado saldría que se eliminó un archivo dentro del “*Changes not staged for commit*”. Solo falta preparar la eliminación del archivo

```
$ git add <archivo>
# o
$ git rm <archivo>
```

Con la próxima confirmación el archivo habrá desaparecido y no volverá a ser rastreado.

Digamos que queremos mantener el archivo en el directorio de trabajo pero eliminarlo del área de preparación, o sea, queremos mantener el archivo en local pero que Git no lo rastree más (esto es muy útil si preparamos cierto archivo y se nos olvidó pasarlo a **.gitignore**). El código es el siguiente

```
$ git rm --cached <archivo>
```

Una vez hecho esto dicho archivo sale de la zona de preparación, o sea, que Git ya no lo está rastreando. Algo muy útil es saber que el código **git rm** acepta directorios y patrones globales también.

¿Cómo cambio el nombre de los archivos?

Git no rastrea explícitamente los cambios de nombres en los archivos. Si renombramos un archivo no se guarda un metadato que indique que renombramos el archivo. Por esto git tiene un comando específico para renombrar un archivo

```
$ git mv <archivo-viejo> <nuevo-nombre>
```

Este código es equivalente a ejecutar este

```
$ mv <archivo-viejo> <nuevo-nombre>
$ git rm <archivo-viejo>
$ git add <nuevo-nombre>
```

Git se da cuenta de que es un renombramiento implícito, así que no importa si renombramos el archivo de esa manera (o con el propio visor de windows), la única diferencia es que `git mv` es un solo comando en vez de tres.

¿Cómo salto el área de preparación?

A pesar de que resulta muy útil para ajustar los commit a veces esta area solo alarga todo el proceso. Para casos mas sencillos o para cuando se tiene experiencia, se puede saltar dicha etapa. Git ofrece un atajo sencillo, añadiendo `-a` al comando `git commit` haremos que Git prepare automáticamente todos los archivos rastreados antes de confirmarlos, ahorrándonos el paso `git add`.

```
$ git commit -a
```

¿Cómo arreglo mi último commit?

Digamos que confirmamos un cambio antes de tiempo y olvidamos agregar algún archivo o nos equivocamos en el mensaje de confirmación. Para solucionar esto podemos reconfirmar con la opción `--amend`

```
$ git commit --amend
```

Si este comando lo ejecutamos después del commit, practicamente solo cambiaremos el mensaje ya que sería como olvidarse del commit anterior y pensar solo en este (esta vía es útil por si nos equivocamos al momento de realizar la descripción de un commit).

¿Pero qué sucede si hicimos más modificaciones al directorio después de ese commit y las queremos agregar al mismo commit y no a otro? Pues para solucionarlo se hace lo siguiente

```
$ git commit -m"Commit inicial"
$ git add cambio_olvidado
$ git commit --amend
```

Luego se ejecuta el editor por defecto con el mensaje *"Commit inicial"*, podemos dejarlo así o modificar también el mensaje.

¿Cómo deshago un archivo preparado?

Cada vez que se preparan los archivos en Git, este te recuerda como sacar algún archivo de dicha área. El código para lograr esto es el siguiente

```
$ git reset HEAD <archivo>
```

Esto hace que el archivo vuelva a estar “no preparado” pero modificado (puede ser que tengamos dos archivos preparados pero queramos confirmarlos por serado).

¿Cómo veo los remotos?

```
$ git remote
```

Esto mostrara los nombres de los remotos (repositorios remotos) que se tienen especificados. Si hemos clonado el repositorio, deberías ver al menos origen (*origin*) este es el nombre que Git le da por defecto al servidor del que se ha clonado.

Si se le pasa la opción **-v** se muestran los URLs que Git ha asociado al nombre y que serán usados para leer y escribir en ese remoto.

¿Cómo añado repositorios remotos?

```
$ git remote add <nombre-repo> <url>
```

Ahora se puede traer toda la información que hay en el **<url>** a traves del **<nombre-repo>**, ejecutando

```
$ git fetch <nombre-repo>
```

¿Cómo traigo y combino remotos?

Como se vio en la pregunta anterior se pueden obtener los datos de un repositorio remoto a traves de **fetch**. El comando irá al repo remoto y se traerá todo los datos que aun no se tiene. Luego de hacer esto, tendremos referencias a todas las ramas del remoto, las cuales podemos combinar e inspeccionar cuando queramos. Por ejemplo, si ejecutamos

```
$ git fetch origin
```

Actualizamos toda la información que se trajo de un repo clonado. Es importante destacar que **fetch** solo trae los datos a nuestro repo local, no los combina automáticamente ni modifica. La combinación con el repo local debemos hacerla manualmente cuando estemos listo.

¿Cómo envió a los remotos?

```
$ git push <nombre-remoto> <nombre-rama>
```

Así se enviarán todos los *commit* realizados en **<nombre-rama>** al repositorio **<nombre-remoto>**. Para caso de servidores clonados el código anterior sería: **git push origin main**. Si alguien más clona el mismo repositorio que nosotros y envía información antes que nosotros, nuestro envío será

rechazado. Tenemos que traer su trabajo y combinarlo con el de nosotros antes de enviarlo al servidor.

¿Cómo inspecciono un remoto?

```
$ git remote show <nombre-del-remoto>
```

El comando lista el URL del repo remoto y las ramas de este. El comando indica claramente que si nos encontramos en la rama main y ejecutamos el comando **git pull**, automáticamente combinamos la rama main remota con tu rama local, luego de haber traído toda la información de ella. En proyectos más avanzados la información que muestra el código es mucho más detallada.

¿Cómo elimino y renombro un remoto?

Si queremos cambiar el nombre de la referencia a un remoto se puede ejecutar el código

```
$ git remote rename <nombre-viejo> <nombre-nuevo>
```

Es importante destacar que al hacer esto también cambia el nombre de la rama remota.

Por otro para eliminar un repositorio remoto, se puede utilizar el siguiente código

```
$ git remote rm <nombre-repo>
```

¿Qué son las etiquetas y qué puede hacer con ellas?

Las etiquetas o *tag* (en casi todos VCS) se utilizan para etiquetar (nombrar) ciertos puntos del historial como importantes. Esta función se utiliza típicamente para marcar versiones de lanzamiento (v1.0).

Git utilizar dos tipos fundamentales de etiquetas: **ligeras** y **anotadas**. Una etiqueta ligera es muy parecida a una rama que no cambia, simplemente es un punto a un commit específico. Por otro lado, las etiquetas anotadas se guardan en la base de datos de Git como objetos enteros. Tienen un *checksum*, contienen el nombre del etiquetador, correo electrónico y fecha, tiene un mensaje asociado y pueden ser firmadas y verificadas con GNU Privacy Guard (GPG). Normalmente se recomienda que se creen estas etiquetas cuando se desea dicha información sino es mejor crear etiquetas ligeras.

A continuación mostraremos algunas de las características y funciones que se pueden realizar con las etiquetas.

<code>git tag</code>	Lista las etiquetas disponibles en git en orden alfabético. <i>* También se pueden buscar etiquetas con un patrón particular.</i>
<code>git tag -a <etiqueta> -m "mensaje asociado"</code>	La opción -a indica que es una etiqueta asociada y -m el mensaje de la misma. Si no se especifica la opción -m Git abre el editor de texto.
<code>git show <etiqueta></code>	Muestra la información de la etiqueta junto con el <i>commit</i> que está etiquetado. <i>* El mensaje muestra la información del etiquetador, la fecha en la</i>

	que el commit fue etiquetado y el mensaje de la etiqueta.
<code>git tag <etiqueta></code>	Crea una etiqueta ligera .
<code>git tag -a <etiqueta> <checksum-del-commit></code>	Etiqueta un commit que ya fue realizado. Basta con especificar el <i>checksum</i> (identificador) o parte del identificador.
<code>git push <remoto> <etiqueta-local></code>	Envía etiquetas a un remoto. El comando <code>git push</code> no transfiere las etiquetas a los remotos, debemos enviarlas de manera explícita al remoto después de haberlas creado.
<code>git push <remoto> --tags</code>	Envía al remoto todas las etiquetas que todavía no existen en él.
<code>git checkout -b <nombre-rama> <etiqueta-a-extraer></code>	Extraer una etiqueta. Git no puede sacar una etiqueta, pues no es algo que pueda mover. Si queremos colocar en el directorio de trabajo una versión de tu repositorio que coincida con alguna etiqueta, debemos crear ramas nuevas en esa etiqueta.

En caso de querer mas información ir a la docs oficial: <https://git-scm.com/docs/git-tag>.

¿Qué son alias y cómo los configuro?

Git no auto-completa el comando si se teclea parcialmente. Si no queremos teclear el nombre completo de cada comando de Git, podemos establecer fácilmente un alias para cada comando mediante `git config`. Ejemplos de alias

```
$ git config -global alias.co checkout
$ git config -global alias.br branch
$ git config -global alias.ci commit
$ git config -global alias.st status

$ git config -global alias.last "log -1 HEAD" # ver última confirmación
```

¿Qué pautas debo seguir al realizar una confirmación?

- Los espacios en blanco, los cuales se pueden verificar antes de realizar una confirmación con el código siguiente

```
$ git diff --check
```

- Tratar de que las confirmaciones vayan por temáticas a resolver.
- Tener el hábito de crear mensajes de compromiso de calidad: deben comenzar con una sola línea que no supere los 50 caracteres y que describa el conjunto de cambios de forma concisa, seguido de una línea en blanco, seguida de una explicación más detallada - explicación más detallada incluye su motivación para el cambio y contraste la implementación con el comportamiento anterior-.

Preguntas – Bloque II

¿Qué es una rama?

Para entender como Git realiza las ramificaciones debemos entender primero como maneja sus datos. En cada confirmación (commit) Git guarda una instantánea¹ del trabajo preparado apuntando a la confirmación precedente (*figura 4*). Una rama Git es simplemente un apuntador (indicador) móvil que nos dice en cual confirmación se encuentra. La rama por defecto main (antes era master). Con la primera confirmación de cambios que realicemos, se creará la rama main apuntando a dicha confirmación².

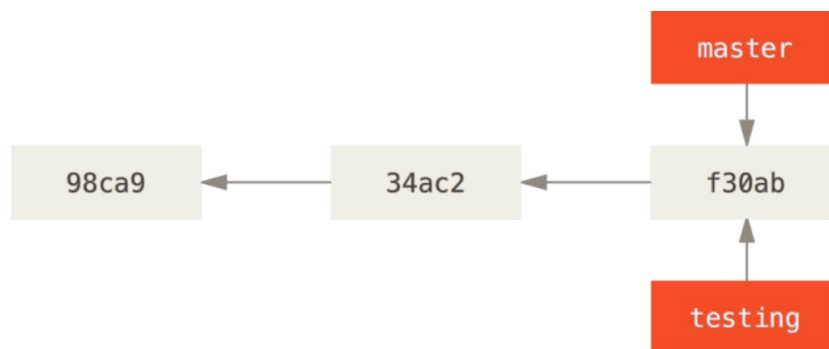
En la *figura 5* se muestra un ejemplo gráfico de una rama en Git y su historial de confirmaciones.

¿Cómo creo una rama nueva?

Cuando creamos una nueva rama lo que estamos creando es un nuevo apuntador que podemos mover libremente. La mejor manera de explicar la idea de apuntador es con un ejemplo, supongamos que creamos una rama nueva llamada “testing”

```
$ git branch testing
```

Esto crea un apuntador que indica a la misma confirmación en donde se esté actualmente.



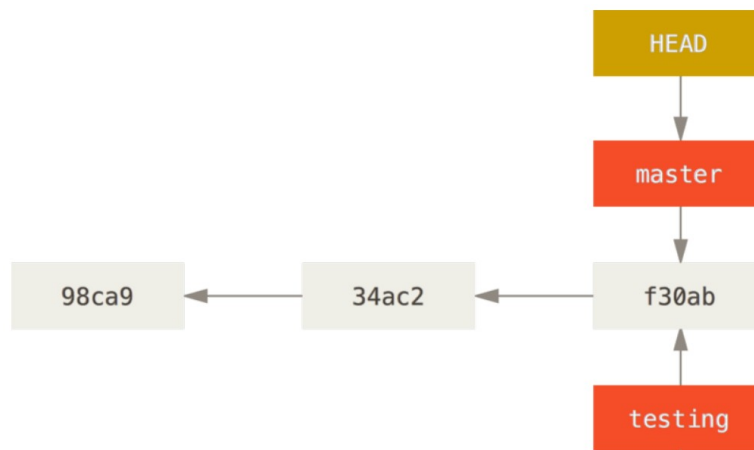
“Dos ramas apuntando al mismo grupo de codificaciones”

Git tiene un apuntador especial denominado HEAD, el cual indica en que rama se encuentra en este momento³. Head en este caso se encuentra en master.

1 Contiene los metadatos del autor, un mensaje explicativo y uno o varios apuntadores a las confirmaciones que sean padres directos de esta.

2 La rama main en Git, no es una rama especial. Es como cualquier otra rama. La única razón por la cual aparece en casi todos los repositorios es porque es la que se crea por defecto el comando `git init` y la gente no se molesta en cambiarla.

3 HEAD es totalmente distinto a los HEAD que se ven en otros CVS.

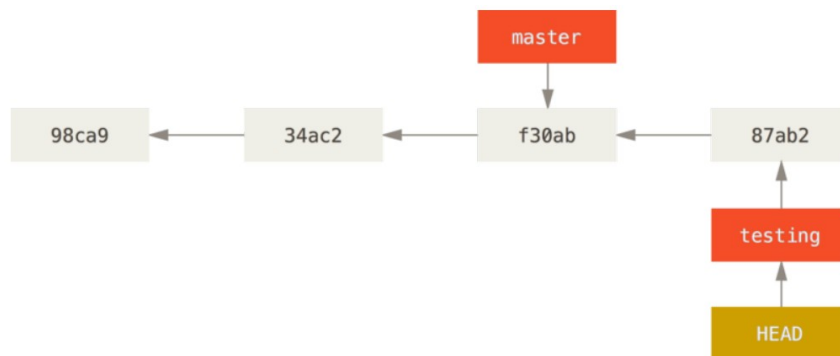


Apuntador HEAD.

¿Cómo cambio de rama?

```
$ git checkout <rama>
```

Esto mueve el apuntador HEAD a la rama <rama>. ¿Cuál sería el objetivo de esto? Pues se ve cuando se realizan otros commit. Supongamos que realizamos otro commit con la rama *testing* del ejemplo anterior, el resultado sería algo así

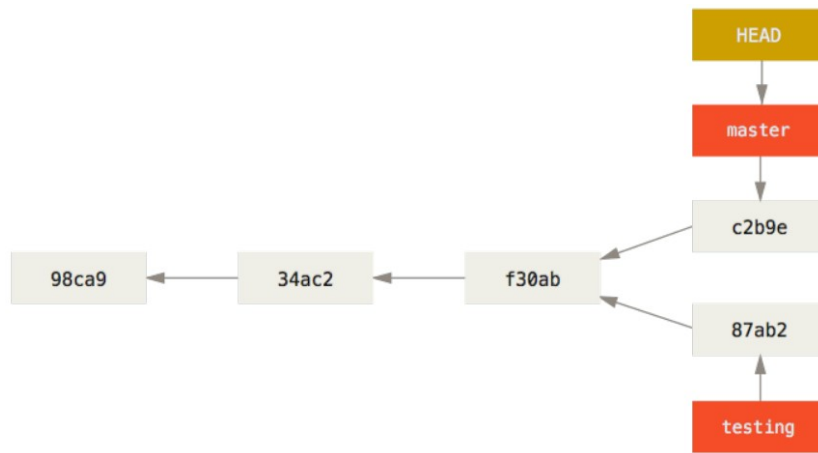


La rama apuntada con HEAD avanza con cada commit.

Lo interesante es que la rama *testing* avanza, mientras que la rama *master* permanece en la confirmación. Si volvemos ahora a la rama *master* (**checkout**) suceden dos cosas: se mueve el apuntador HEAD a *master* y revierte tus archivos del directorio de trabajo, dejándolos tal y como estaban en la última confirmación realizada en dicha rama *master*. Esto nos permite avanzar en otra dirección⁴.

Para concluir con esta pregunta, supongamos que agregamos otro cambio más pero esta vez con la rama *master*. Esto trae consigo que los cambios realizados en ambas sesiones de trabajo están aislados en ramas independientes: podemos saltar libremente de una a otra según estimemos oportuno.

⁴ Hay que resaltar que cuando saltamos a una rama en Git, los archivos de tu directorio de trabajo cambian. Si saltamos a una rama antigua, los archivos del directorio retrocederán para verse como lo hacían la última vez que confirmamos un cambio en dicha rama. Si Git no puede hacer el cambio limpiamente, no te deja saltar.



Los registros de las ramas divergen

¿Cómo funciona la ramificación y la fusión?

La mejor manera de explicar como funciona la ramificación y la fusión es con un ejemplo de flujo de trabajo común:

1. Trabajamos en un sitio web.
2. Creamos una rama para un nuevo tema sobre el que queremos trabajar.
3. Realizamos algo de trabajo en esa rama.
4. Surge un problema crítico por lo que nos movemos a la rama original (producción).
5. Creamos una nueva rama para resolver el problema.
6. Después de resolver el problema, fusionamos (merge) esa rama y la enviamos (push) a la rama original.
7. Por último, volvemos a la rama del tema y continuamos trabajando.

¿Cuáles son los procesos básicos de ramificación?

Supongamos que estamos trabajando en un problema de nuestro proyecto y creamos una nueva rama

```
$ git checkout -b <rama>
```

Este comando crea la rama (branch) y se mueve directamente a ella. Ahora se nos presenta otro problema en la rama principal. Para resolverlo nos movemos a dicha rama, pero para por hacerlo no podemos tener cambios sin confirmar en la rama del primer problema. Una vez en la rama principal Git añade, quita y modifica archivos automáticamente para asegurar que la copia de trabajo luce exactamente como lucía antes. Una vez que el segundo problema es solucionado se fusiona dicha rama con la rama main

```
$ git checkout main  
$ git merge <rama>
```

Nos posicionamos en la rama que recibirá los cambios para luego realizar la fusión. Luego de realizar la fusión, se recomienda eliminar la rama <rama> ya que estaría apuntando a la misma dirección que main. Para eliminar una rama se ejecuta el siguiente código

```
$ git branch -d <rama>
```

Cabe destacar que los cambios realizados en la rama del segundo problema, no están en la rama del primer problema.

¿Cuáles son los procesos básicos de fusión?

Utilicemos el ejemplo anterior y supongamos que queremos fusionar también la rama asociada al primer problema. El código es exactamente igual, nos posicionamos en la rama main (checkout) y luego fusionamos la rama (merge). Ahora, este caso es diferente porque la confirmación en la rama actual no es ancestro directo de la rama que se pretende fusionar. Para solucionar esto Git realiza una fusión a tres bandas utiliza las dos confirmaciones extremas de cada rama y el ancestro común de dichas ramas. Git crea automáticamente una confirmación resultante de la fusión de las tres bandas. Nos referimos a este proceso como *fusión confirmada* y su particularidad es que tiene más de un padre.

¿Cuáles son los conflictos básicos que pueden surgir en la ramificación?

Si hay modificaciones dispares en una misma porción de un mismo archivo en las dos ramas distintas que pretendemos fusionar, Git no será capaz de fusionarlas directamente. Git no crea una nueva fusión confirmada, sino que hace una pausa en el proceso, esperando a que uno resuelva el conflicto. Para ver los archivos que están generando el conflicto se ejecuta el código `git status` y todos aquellos que sean Unmerged tienen un conflicto. También Git añade unos marcadores a los archivos conflictivos, para hacer más fácil la solución. Una vez que se ha resuelto el problema se añaden dichos archivos (add). Luego el estado de la rama quedará como **(main | MERGING)**, lo cual indica que debe continuar el **merge** con el siguiente comando

```
$ git merge --continue
```

Resumiendo todo un poco: lo primero es hacer el **merge**, luego la línea de comandos señala que tenemos un conflicto y nos envía al editor por defecto; aquí se arregla el conflicto y se regresa a la línea de comandos en donde continuamos con el **merge**; y por último, se guarda toda la fusión en un **commit**.

Podemos resolver los conflictos con la herramienta gráfica: **git mergetool**.

¿Qué es una rama de largo recorrido?

Se mantiene en la rama main el código estable y en las otras ramas denominadas “desarrollo” o “siguiente” se trabaja y se realizan pruebas. Una vez que la rama se considera estable se fusiona inmediatamente con main. Las ramas estables apuntan hacia posiciones más antiguas en el historial de confirmaciones, mientras que las ramas avanzadas, las que van abriendo camino, apuntan hacia posiciones más recientes.

Para lograr este flujo de trabajo es necesario respetar:

- La rama **main** es la rama principal, por lo que no se puede fusionar con ramas inestables.
- Cada rama solo puede tener un padre y un hijo.
- Siempre se trabaja en la última rama creada, una vez estable se fusiona con su padre.

Todas estas pautas hacen que el proyecto se comporte como una única línea de cambios y las ramas (una vez sean estables) se fusionan con **main**.

¿Qué es una rama puntual?

Una rama puntual es aquella rama de corta duración que abres para un tema o para una funcionalidad determinada. En estas se hacen varias confirmaciones y luego se borra tras fusionarla. Esta técnica posibilita realizar cambios de contexto rápidos y complejos. Podemos mantener los cambios durante minutos, días o meses y fusionarlos cuando realmente estén listos, sin importar el orden en el que fueron creados o en el que comenzamos a trabajar en ellos.

¿Qué es una rama remota?

Las ramas remotas son referencias al estado de las ramas en los repo remotos. Las ramas remotas funcionan como marcadores, para recordar en que estado se encontraba el repo remoto la última vez que conectamos con él. Suelen referenciarse como (remoto)/(rama).

Cuando clonamos un repositorio remoto, Git denomina automáticamente dicho repo **origin**, trae sus datos y crea un apuntador hacia donde esté en ese momento su rama **main** y denomina la copia local **origin/main**. Git proporciona también una rama **main**, apuntando al mismo lugar que la rama **main** de **origin**.

Git asigna por defecto **origin** al repo remoto, pero esto puede ser modificado con el siguiente código

```
$ git clone -o <nombre>
```

¿Qué sucede si hacemos cambios en la rama local y al mismo tiempo alguien más lleva su trabajo al servidor? Debemos sincronizar con el comando **git fetch origin**. Este comando localiza el servidor **origin**, recupera cualquier dato presente que uno no tenga y actualiza tu base de datos local, moviendo la rama **origin/main** para que apunte a la posición más reciente. Los cambios realizados en el repo local se mantiene siempre y cuando no se encuentren conflictos.

¿Cómo publico ramas?

Nuestras ramas locales no se sincronizan automáticamente con los repos remotos en los que escribimos. De esta manera, podemos usar ramas privadas para el trabajo que no deseamos compartir, llevando a un remoto tan solo aquellas partes que deseamos aportar. Supongamos que queremos compartir una rama a un repo remoto, esto lo hacemos mediante el comando

```
$ git push <remoto> <rama-local>
```

Digamos que no queremos que los nombres de las ramas coincidan, pues esto se logra con

```
$ git push <remoto> <rama-local>:<rama-remota>
```

La próxima vez que los colaboradores recuperen desde el servidor, obtendrán bajo la rama remota **<remoto>/<rama-remota>** una referencia a donde está la versión de **<rama-local>** en el servidor. Es importante destacar que cuando recuperamos (fetch) nuevas ramas remotas, no obtenemos automáticamente una copia local editable de las mismas, sino un puntero no editable a **<remoto>/<rama-remota>**. Para tener dicha rama remota de forma local se utiliza el código

```
$ git checkout -b <rama-local> <remoto>/<rama-remota>
```

¿Cómo hago seguimiento a una rama?

Al activar (checkout) una rama local a partir de una rama remota, se crea automáticamente una “rama de seguimiento” (tracking branch). Las ramas de seguimiento son ramas locales que tiene una relación directa con alguna rama remota. Si nos encontramos en una rama de seguimiento y tecleamos **git pull**, Git sabe de cuál servidor recuperar (fetch) y fusionar (merge) datos.

Cuando clonamos un repo remoto, este suele crear automáticamente una rama **main** que hace seguimiento a **origin/<rama>**. Sin embargo podemos crear otras ramas de seguimiento si deseamos tener unas que sigan ramas de otros remotos o no seguir la rama **main**. La forma más simple es con el código

```
$ git checkout -b <rama-local> <remoto>/<rama-remota>
```

Esta operación es tan común que Git brinda el parámetro **--track**

```
$ git checkout --track <remoto>/<rama-remota>
```

La rama local se llamará igual que en el repo remoto.

También pudiéramos querer asignarle una rama remota (recientemente traída) a una rama local o querer cambiar la rama de seguimiento de una rama local. Pues para esto utilizamos el siguiente código (debemos estar sobre la rama local)

```
$ git branch -u <remoto>/<rama-remota>
```

¿Qué traigo y fusiono un repo remoto?

A pesar de que el comando **git fetch** trae todos los cambios que tenemos del remoto, este no modifica los directorios de trabajo. Sin embargo existe un comando **git pull**, el cual básicamente realiza **git fetch** seguido de un **git merge** en la mayoría de los casos. Normalmente es mejor usar los comandos **fetch** y **merge** de manera explícita pues la magia de **git pull** puede resultar confusa.

¿Cómo elimino ramas remotas?

Imaginemos que ya hemos terminado con una rama remota, es decir, tanto uno como los colaboradores habéis terminado una funcionalidad y la habéis incorporado a **main** (merge) del repo remoto. Dicha rama se puede eliminar con la opción **--delete**

```
$ git push <remoto> --delete <rama-remota>
```

Básicamente lo que hacemos es eliminar el apuntador del servidor. El servidor Git suele mantener los datos por un tiempo hasta que el recolector de basura se ejecute, de manera que si la hemos borrado accidentalmente, suele ser fácil recuperarla.

¿Qué estilo de ramificación usar?

Lo importante es saber como contribuir a un proyecto tanto para uno como para otros. Para hacer esto existen diferentes diseños, de los cuales podemos seleccionar uno o combinarlos.

¿Cómo funciona el flujo de trabajo centralizado? Existe un repositorio o punto central que acepta código y todos sincronizan su trabajo con él (figura 1). esto significa que si dos desarrolladores clonan desde el punto central, y ambos hacen cambios, solo el primer desarrollador en subir sus cambios lo podrá hacer sin problemas. El segundo debe fusionar el trabajo del primero antes de subir sus cambios, para no sobrescribir los cambios del primero. Para lograr este tipo de trabajo basta con crear un solo repositorio y darle a cada uno del equipo acceso de push (Git no permite que los usuarios se sobrescriban entre sí).

¿Cómo es el flujo de trabajo Administrador-Integración? Debido a que Git permite tener varios repos remotos, es posible que cada desarrollador tenga acceso de escritura en su propio repo público y acceso de lectura a todos los demás. El proceso funciona de la siguiente manera:

1. El admin hace un **push** al repositorio público.
2. El contribuidor **clona** ese repositorio y realiza los cambios.
3. El contribuidor realiza un **push** con su copia pública.
4. El contribuidor envía un correo pidiendo al admin que haga **pull** de los cambios.
5. El admin agrega el repo del contribuidor como remoto y fusiona ambos localmente.
6. El admin realiza un **push** con la fusión al repo principal.

Este es un flujo de trabajo muy común con herramientas basadas en hubs como **Github** o **Gitlab**, donde es fácil hacer un **fork** de un proyecto e introducir los cambios en este **fork** para que todos puedan verlos.

¿Cómo es el flujo de trabajo Dictador-Tenientes? Es una variante de un flujo de trabajo de múltiples repositorios. Generalmente es utilizado por grandes proyectos. El proceso funciona así:

1. Los desarrolladores trabajan en su propia rama específica y fusionan sus códigos en la rama **main**, la cual, es una copia de la rama del *dictador*.
2. Los tenientes fusionan el código de las ramas **main** de los desarrolladores en sus ramas **main** de *tenientes*.
3. El dictador fusiona la rama **main** de los tenientes a su rama **main** de *dictador*.
4. El dictador hace **push** del contenido de su rama **main** al repo para que otros fusionen los cambios a sus ramas.

¿Qué es la Reorganización?

La manera más sencilla de integrar ramas es con **merge**. Realiza una fusión a tres bandas entre las 3 últimas instantáneas de cada rama y el ancestro común a ambas; creando una nueva instantánea (snapshot) y la correspondiente confirmación (figura 6). Sin embargo también hay otra

forma de hacerlo: puede capturar los cambios introducidos en C4 y replicarlos encima de C3. Esto se llama *reorganizar* (rebasing). Con el código git rebase se pueden capturar todos los cambios confirmados en una rama y reaplicarlos sobre otra.

```
$ git checkout experiment
$ git rebase master
```

Haciendo que Git vaya al ancestro común de ambas ramas, saque las diferencias introducidas por cada confirmación en la rama donde estamos y guarde esas diferencias en archivos temporales, reinicie la rama actual (reset) hasta llevarla a la misma confirmación que la rama de donde queremos reorganizar y finalmente vuelva a aplicar ordenadamente los cambios (figura 7). Ya en este punto solo queda volver a la rama master y fusionar ambas ramas (merge).

Si la instantánea apuntada en C4' es exactamente la misma apuntada por C5 en el ejemplo de la fusión. No hay ninguna diferencia en el resultado final de la integración, pero el haberlas hecho reorganizando nos deja un historial más claro. Si se examina el historial de una rama reorganizada, este aparece siempre como un historial lineal: como si todo el trabajo se hubiera realizado en series, aunque se haya hecho en paralelo.

¿Cuándo utilizarlo? Cuando queramos estar seguros que nuestras confirmaciones de cambios se pueden aplicar limpiamente sobre una rama remota. Podemos trabajar sobre una rama y luego reorganizar lo realizado en la rama origin/main cuando lo tengamos todo listo para enviarlo al proyecto principal. Cabe destacar que la reorganización vuelve a aplicar cambios de una rama de trabajo sobre otra rama, en el mismo orden en que fueron aplicados en la primera, mientras que la fusión combina entre sí los dos finales de ambas ramas.

¿Qué puede salir mal con la reorganización? Nunca reorganicemos confirmaciones de cambios que hayamos enviado a un repositorio público. Cuando reorganizamos algo, estamos abandonando las confirmaciones de cambio ya creadas y creando nuevas confirmaciones; que son similares pero diferentes.

¿Qué es mejor fusionar o reorganizar?

Par algunos el historial de confirmaciones de un repo es **un registro de todo lo que ha pasado**. Un documento histórico, valioso por si mismo y que no debería ser alterado. Desde este punto de vista, cambiar el historial sería un error ya que estaríamos mintiendo sobre lo que en verdad ocurrió.

La otra forma de verlo puede ser que, el historial es **la historia de cómo se hizo el proyecto**. Uno no publica el primer borrador de tu novela, y el manual de cómo mantener los programas también debe estar editado con mucho cuidado. Esta es el área que utiliza herramientas como **rebase** y **filter-branch** para contar la historia de una mejor manera.

Normalmente, la manera de sacar la mejor de ambas es reorganizar el trabajo local, que aún no hemos compartido, antes de enviarlo a algún lugar, pero **nunca** reorganizar nada que haya sido publicado.

Figuras

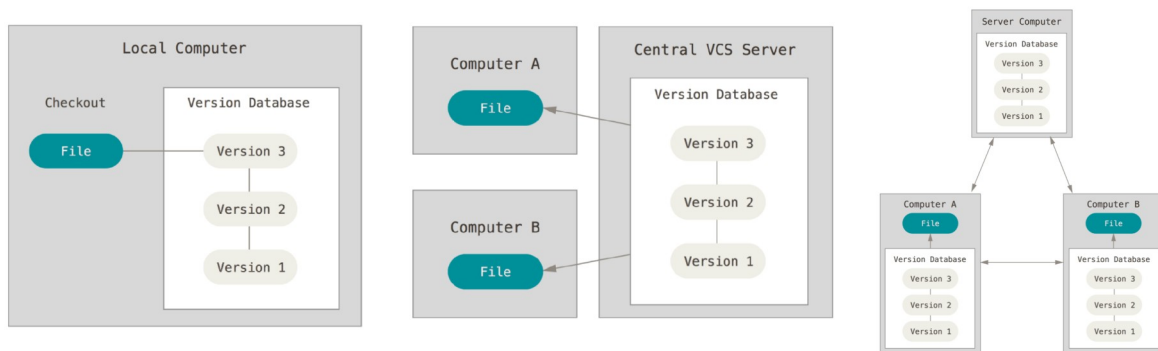


figura 1 “CV Local – CV Centralizado – CV Distribuido”

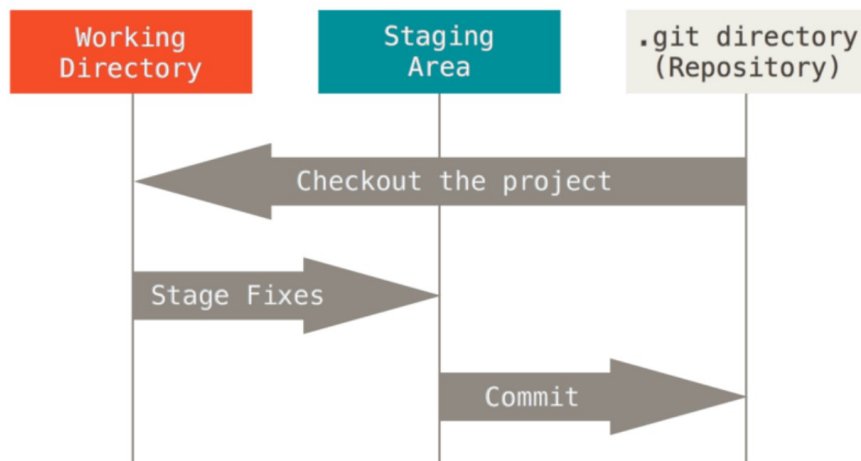


figura 2 “Directorio de trabajo, área de almacenamiento y directorio Git”

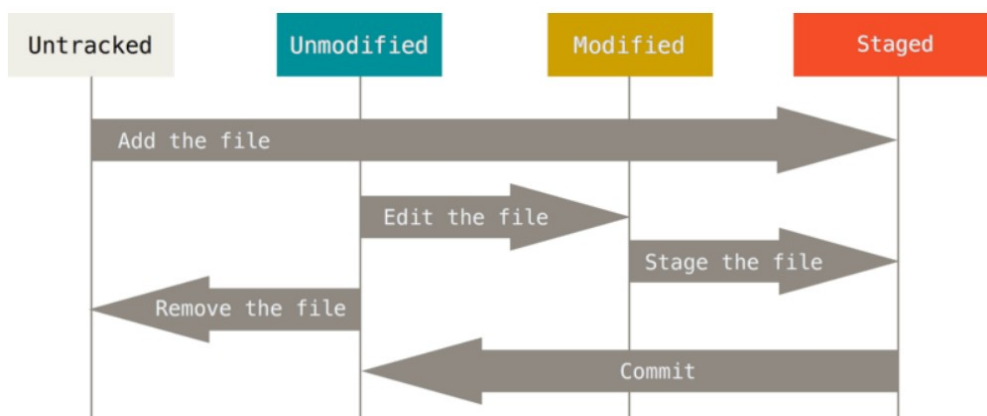


figura 3 “El ciclo de vida del estado de los archivos en un repo”

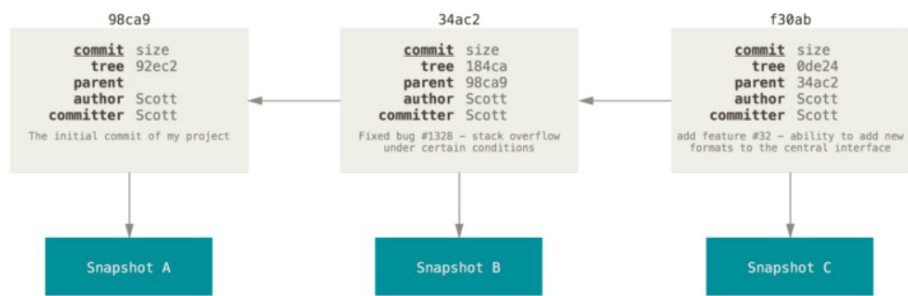


figura 4 “Confirmaciones y sus predecesoras”

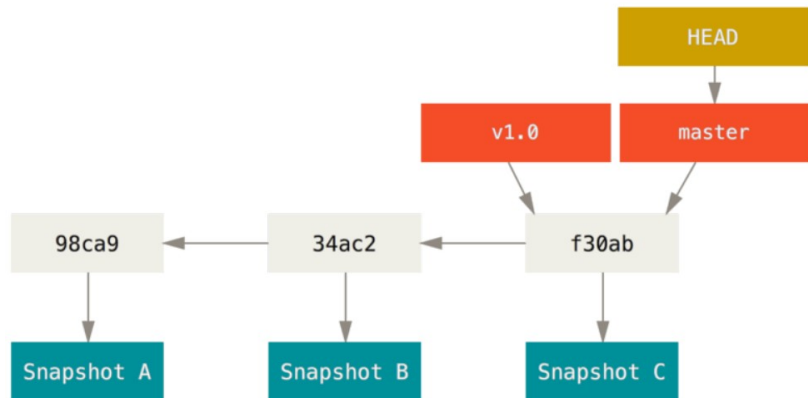


figura 5 “Una rama y su historial de confirmaciones”

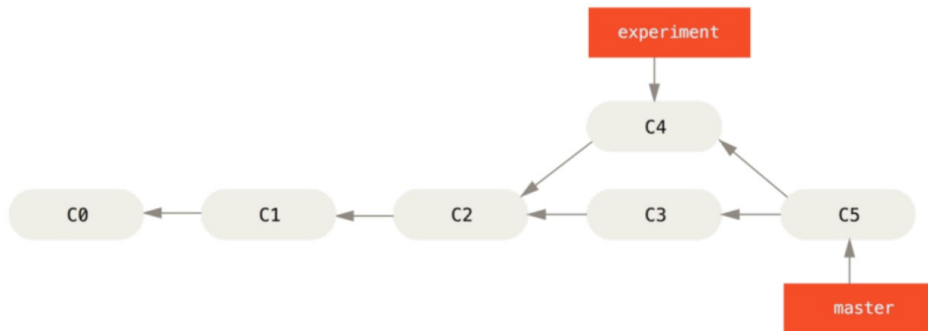


figura 6 “Fusión de ramas con registros divergentes”

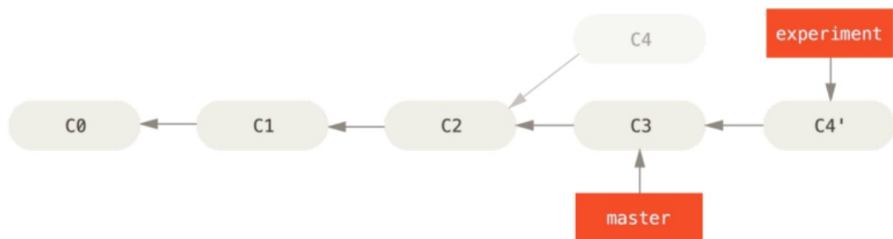


figura 7 “Reorganizar sobre C3 los cambios introducidos en C4”

!!! todos los capítulos con un asterisco serán eliminados o fusionados con la nueva versión.

* GitHub

GitHub

Es una plataforma pensada para colaboración entre personas en un proyecto o para almacenar el repositorio de un proyecto en la nube. Formalmente: es un servicio de hosting que nos permite almacenar proyectos de desarrollo de software y control de versiones usando Git.

*** Crear una cuenta en GitHub**

- Para esto hace falta tener un correo para la cuenta.
- Luego debes poner un usuario, lo cual debe ser algo a pensar bien ya que con este nombre se conocerá tu repositorio.
- También se debe configurar cuantas personas trabajarán con esa cuenta y si eres estudiante o profesor.
- GitHub nos suministra muchas aplicaciones internas con las cuales se puede hacer todo el proceso de administración del proyecto aun más fácil.

*** Crear un repositorio**

- Primeramente se le coloca un nombre (de preferencia en minúscula y separadas las palabras con “-”).
- Se coloca una pequeña descripción del repositorio.
- Se selecciona si se quiere que el repositorio sea público o privado.
- Se le puede agregar un README en el cual iría una pequeña descripción del proyecto.
- Se puede agregar el archivo .gitignore.
- Se puede seleccionar una licencia.
- Se configura el nombre de la rama principal.

*** Ajustes del repositorio**

Estudiar mas a fondo después.

*** Agregar archivos desde GitHub**

* Clonar un servidor remoto

Clonar

Crear una copia local de un repositorio remoto, incluyendo sus versiones e historial de commits.

Origin

Nombre que comúnmente le asignamos al repositorio remoto que clonamos.

Para clonar un repositorio existen tres vías según GitHub: HTTPS, SSH, GitHub CLI [Estudiar]. En el curso se utiliza HTTPS, con la siguiente línea de comandos

```
git clone <url suministrado por GitHub>
```

Una vez hecho esto se puede trabajar de manera local en dicho repositorio sin problemas. Se aconseja primero de realizar algún commit, configurar el correo y el usuario, para que salgan tus credenciales.

Luego, una vez que se realiza un commit en el repositorio local y se comprueba el estado de git (**status**) de seguro saldría un mensaje así: *Your branch is ahead of "origin/main" by 1 commit.* Esto nos quiere decir que existen cambios en el repositorio (commit) local que no han sido llevados al repositorio en la nube (GitHub) o repositorio remoto. La palabra "*origin*" en el mensaje indica el nombre que se le asignó al repositorio remoto. En caso de querer ver el nombre de dicho repositorio basta con poner en la consola

```
git remote -v
```

NOTA: Si le agregamos a la línea anterior la propiedad **-v** este nos brinda más información sobre el repositorio remoto, por ejemplo, las direcciones de *fetch* y *push*. Fetch nos sirve para ver los cambios del repositorio remoto (ejemplo, si otros desarrolladores han realizado cambios) y push para agregar nuestros cambios al repositorio remoto.

* Enviar cambios a GitHub

push

Comando usado para enviar los cambios realizados en un repositorio local a un repositorio remoto para que ambos tengan la misma información.

Luego para enviar los cambios (pero antes leer el capítulo siguiente) se hace así

```
git push origin main
```

*** Ocultar el correo personal**

Para esto se va a Opciones/Emails. Luego aquí en el apartado de “Primary email address” viene la opción por defecto configurada. Pero cuando se hace un clon local y se realizan cambios, se debe estar atento para no realizar los commit con la información personal. Por lo que se debe copiar el correo configurado por defecto en git y utilizarlo de igual manera de forma local.

* Configurar HTTPS con un Token

Cualquier persona puede clonar un repositorio que es publico pero para poder hacerle modificaciones necesita una especie de autenticación. En el curso se clona un repositorio local (vía HTTPS) y se intenta agregar algunos cambios locales al repositorio.

Lo primero sería, comprobar si existe un repositorio remoto (lo cual será cierto debido a que es un clon) asociado al repositorio local. Para esto, se puede utilizar la sentencia (ya utilizada anteriormente): **git remote -v**. Puede devolver tres cosas:

- Que te devuelva dos direcciones del directorio remoto (*origin*) para realizar el *fetch* (traer los cambios) y *push* (enviar los datos), donde las direcciones son iguales para ambos.
- No devuelva nada.

En nuestro caso saldría la primera opción.

Luego se realizan unos cuantos cambios en el repo (commit) y se trata de hacer un push con la siguiente linea de código

```
git push origin main
```

De forma automática GitHub abre una ventana emergente en la cual nos pide que iniciemos GitHub y nos brinda tres opciones:

- Navegador.
- Código.
- Token.

La vía aconsejada es trabajar con el *Token* ya que no tendríamos que escribir nuestras credenciales cada vez que queremos realizar un push. Para generar este Token debemos:

1. ir a las opciones generales del perfil de GitHub,
2. Developer setting – Opciones de desarrollador
3. Personal access tokens / Tokens (classic)
4. *Generate a personal access token*
 1. Nota asociada a ese token, lo normal es para señalar a que está dando acceso.
 2. Fecha de expiración (GitHub recomienda que debe tener una fecha de aspiración)
 3. Scope – Permisos
 1. Repositorios (SI -Todos)
 2. Paquetes o opciones
 3. organizaciones o equipos
 4. gist (SI)
 5. Los datos del usuario

6. eliminar repositorios
7. control sobre los proyectos
8. empresas
9. las claves SSH (otro tipo de autenticación)

El Token generado debe guardarse en algún lugar porque GitHub no los volverá a mostrar. Una vez creado el Token, si regresamos a las opciones de generar Token, muestra toda la información del mismo y de todos los demás Token creados (los permisos de un Token específico pueden ser editados). Para casos de perder Token, se puede crear uno nuevo (regenerar el mismo) pero se deben actualizar los proyectos locales.

Una vez colocado el token en la ventana emergente de GitHub, se regresa al Bash y debe andar todo bien (los cambios se realizaron correctamente) lo cual se muestra en un mensaje.

* git pull

pull

Comando usado para descargar el contenido de un repositorio remoto e inmediatamente actualizar un repositorio local para que ambos tengan la misma información.

El objetivo del pull es agregar cambios realizados en el repositorio remoto a nuestro repositorio local (ejemplo, si otro desarrollador realizo actualizaciones). Para explicar esto se crea un nuevo repositorio en GitHub: *New repository* (ejemplo-git-pull).

- Luego se clona el repositorio (HTTPS) de manera local.
- Se realiza un cambio en el repositorio remoto (por ejemplo, se modifica el README).
- Luego en el repositorio local para recibir los cambios realizados en el repo remoto se hace con: **git pull origin main**.

* pull vs fetch

pull

Comando usado para descargar el contenido de un repositorio remoto e **inmediatamente actualizar** un repositorio local para que ambos tengan la misma información.

fetch

Comando usado para **verificar** los cambios realizados en el repositorio remoto sin combinar esos cambios con el repositorio local. Te permite saber **si se han realizado cambios** en el repositorio remoto desde la última vez que actualizaste tu repositorio local con git pull.

*** Repo local a repo remoto**

Se crea un repositorio local. Luego se crea un repositorio en GitHub (el nombre no tiene que ser el mismo) pero totalmente vacío. Luego GitHub en la página del repositorio te guía de todo el proceso, pero lo más importante es saber que la línea que conecta los repositorios, es escribir en la línea de comandos:

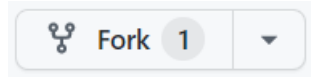
```
git remote add origin <dirección generada por GitHub>
```


* Bifurcar un repositorio

Bifurcar

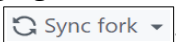
Fork – Crear una copia del repositorio remoto en tu cuenta de GitHub. Es prácticamente clonar otro repositorio de una cuenta de GitHub en tu cuenta de GitHub.

Ir al botón Fork en GitHub ::



Los repositorios que son bifurcaciones se conocen porque se marcan con  a la izquierda del nombre y debajo dicen al creador del mismo.

¿Qué se puede hacer con los mismos? Todo lo que queramos ya que esa bifurcación es un repositorio personal. Pero además de esto, podemos trabajar en conjunto con el creador del mismo (si fuera el caso claro) ya que podemos agregar nuestros cambios al repositorio original (*push*)

 o actualizar (*fetch*) nuestro *fork* .

Ya una vez que se tiene la bifurcación, se puede realizar una copia local del mismo (ir a 27). Para trabajar con dicha copia desde casa y realizare las modificaciones deseadas, pero ¿qué pasa si queremos agregar dichas modificaciones al proyecto original? Pues esto se pude hacer, gracias al botón anterior “Contribute” o mejor conocido como “*pull request*”. Ya una vez ahí se escribe en detalle el mensaje ya que el desarrollador del repo original debe entender bien cual a sido el cambio y el motivo por el cual se debería cambiar el repo. Cuando dicho mensaje está listo se puede enviar, pero si no está listo se puede crear un borrador (en el mismo botón de enviar – *Create draft pull request*).

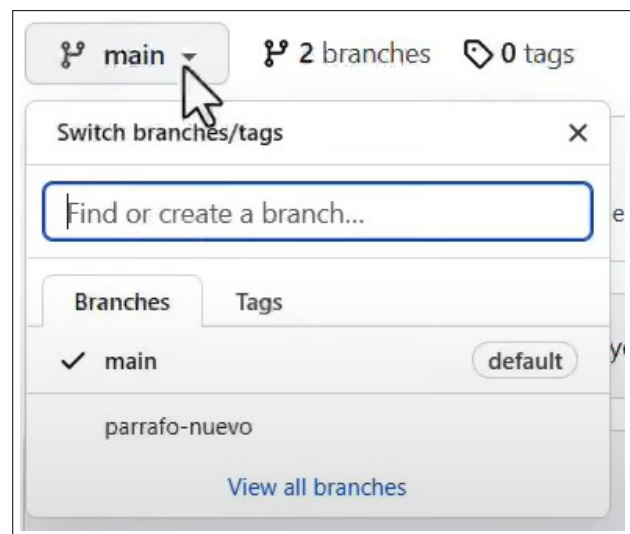
* pull request a partir de una rama

code .

Abre la carpeta que se encuentra en la línea de comandos en VSC.

Lo primero es crear una rama nueva (**git checkout -b <nombre-de-la-rama>**) en el repo local del repo bifurcado. Realizamos las modificaciones y agregamos el archivo al monitoreo (**add**). Luego se agregan los cambios (**commit**). Luego se envían los cambios del repo local al repo en GitHub, pero con una particularidad debido a que estamos trabajando en una rama de el repo y no podemos hacerlo de manera normal (**git push origin main**) ya que estaríamos enviando los cambios de la rama main, lo cual no tiene sentido (ya que no hay cambios). Por tanto, solo tenemos que cambiar en el código la rama en la que trabajamos ya que en esta si están los cambios (**git push origin <nombre-de-la-rama-nueva>**).

Ya una vez con la rama en el repo remoto cambiamos a la rama creada (en la imagen la rama sería “parrafo-nuevo”) y después se realiza el *pull request* en “Contribute”.



* Actualizar una Bifurcación

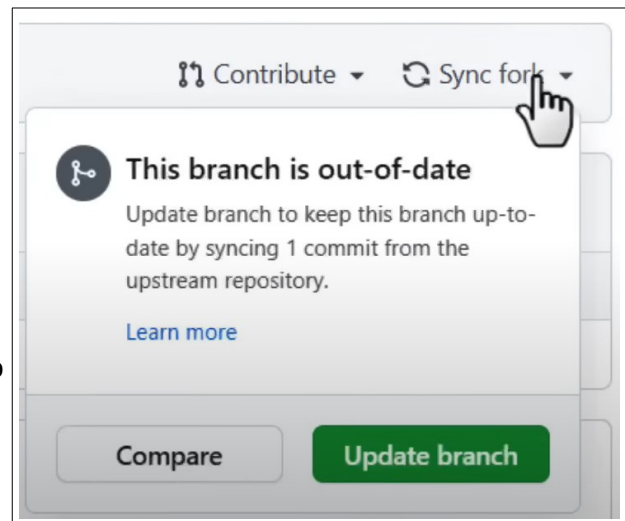
Muchas veces sucede que la bifurcación queda desactualizada con respecto al repo original. Normalmente GitHub te avisa que la bifurcación está desactualizado y te dice lo que tiene de diferente.



GitHub de deja mesclar la bifurcación con el repo original (*contribute*) o actualizar la bifurcación (*Sync fork*). Seleccionamos la segunda opción y tenemos dos caminos a seguir

“Esta rama esta obsoleta. Actualizar la rama con 1 commit del repositorio superior”

Podemos comparar los cambios o actualizar la bifurcación.



CONSEJO: Cuando se trabaja con una bifurcación es mejor dejarlo intacto, así cuando el desarrollador original del repo realice algún cambio podemos actualizar la bifurcación sin conflicto con cambios que podamos haber agregado nosotros. Por esto, la buena es hacerle una rama (*branch*) a la bifurcación y trabajar en esta rama.

*** Issues**

¿Qué es un ISSUES?

¿Cómo crear un issues?

Algunas plantillas

+++ Regresar a esta parte del video para tomar mejores notas +++