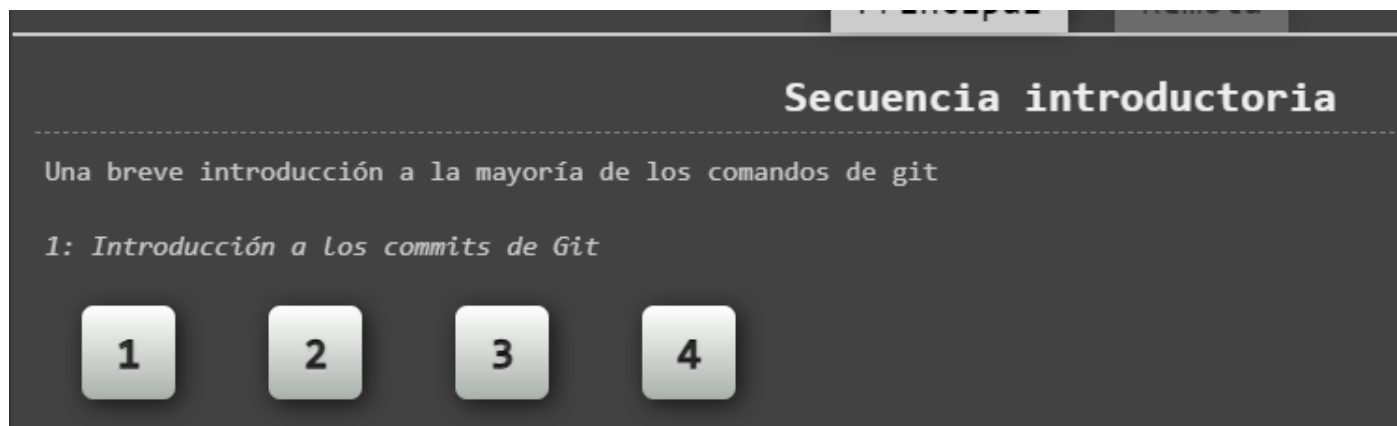


# GUIA DE GIT




## Commits de Git

Un commit en un repositorio git registra un snapshot de todos los archivos en tu directorio. Es como un *gran copy&paste*, ¡pero incluso mejor!

Git pretende mantener los commits tan livianos como sea posible, por lo que no copia ciegamente el directorio completo cada vez que haces un commit. Puede (cuando es posible) comprimir un commit como un conjunto de cambios (o un "*delta*") entre una versión de tu repositorio y la siguiente.

Git mantiene, también, un historial de qué commits se hicieron y cuándo. Es por eso que la mayoría de los commits tienen commits ancestros encima -- designamos esto con flechas en nuestra visualización. ¡Mantener el historial es genial para todos los que trabajan en el proyecto!


Hay un montón en lo que ahondar, pero por ahora puedes pensar en los commits como snapshots de tu proyecto. Los commits son muy livianos, y ¡cambiar de uno a otro es terriblemente rápido!




### Aprende Git Branching

Mostrar objetivo **Level Introducción a los commits de Git** Objetivo

```
$ level intro1
$ hint
¡Simplemente escribe 'git commit' dos veces para terminar!
$ delay 2000
$ show goal
$ git commit
$ git commit
```





**Ramas en Git**

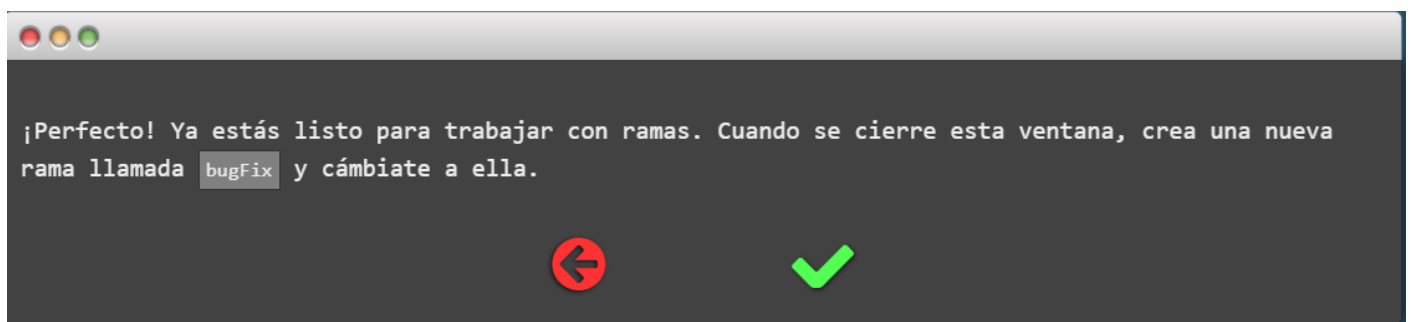
Las ramas (branches) en Git son increíblemente livianas. Son sólo referencias a un commit específico - nada más. Por esto es que tantos entusiastas de Git siguen el mantra:

crea ramas al principio y hazlo también de manera frecuente

Como no hay consumo extra de almacenamiento ni memoria al crear varias ramas, lógicamente es más fácil dividir tu trabajo que trabajar solamente con un par de ramas grandes.

Cuando empecemos a mezclar ramas y commits, vamos a ver cómo se combinan estas dos herramientas. Por ahora, en cambio, simplemente recuerda que una rama esencialmente dice "Quiero incluir el trabajo de este commit y todos su ancestros".

Navigation icons: a red left arrow and a green right arrow.



¡Perfecto! Ya estás listo para trabajar con ramas. Cuando se cierre esta ventana, crea una nueva rama llamada `bugFix` y cámbiate a ella.

Navigation icons: a red left arrow and a green checkmark.

```
In [2]: #git branch bugFix
        #git checkout bugFix; git commit
```

## Haciendo merge en ramas

¡Genial! Ya sabemos cómo crear un commit y cómo crear ramas. Ahora tenemos que aprender algún modo de unificar el trabajo de dos ramas diferentes. Esto nos va a permitir abrir una nueva rama de desarrollo, implementar alguna nueva funcionalidad, y después unirla de nuevo con el trabajo principal.

El primer método para combinarlas que vamos a explorar es `git merge`. Hacer merge en Git crea un commit especial que tiene dos padres diferentes. Un commit con dos padres esencialmente significa "Quiero incluir todo el trabajo de estos dos padres , y del conjunto de todos sus ancestros"

Es más simple visualizarlo, veámoslo a continuación



Para completar este nivel, sigue estos pasos:

- Crea una nueva rama, llamada `bugFix`
- Haz checkout de la rama `bugFix` usando `git checkout bugFix`
- Crea un commit
- Vuelve a `master` con `git checkout`
- Crea otro commit
- Haz merge de la rama `bugFix` a `master` usando `git merge`

*Recuerda: siempre puedes volver a ver este mensaje escribiendo "objective"!*





```
In [3]: #git branch bugFix
        #git checkout bugFix
        #git commit
        #git checkout master
        #git commit
        #git checkout master; git merge bugFix
```

## Git Rebase

El segundo modo de combinar el trabajo de distintas ramas es el *rebase*. Hacer rebase esencialmente selecciona un conjunto de commits, los "copia", y los aplica en algún otro lado.

Aunque esto pueda sonar confuso, la ventaja de hacer rebase es que puede usarse para conseguir una secuencia de commits lineal, más bonita. El historial / log de commits del repositorio va a estar mucho más claro si sólo usas rebase.


Veámoslo en acción...



Para completar este nivel, haz lo siguiente:

- Haz checkout de una nueva rama llamada `bugFix`
- Crea un commit
- Vuelve a la rama master y crea otro commit
- Haz checkout en bugFix otra vez y haz rebase sobre master

¡Misión cumplida!



```
In [4]: #git branch bugFix
        #git checkout bugFix
        #git commit
        #git checkout master
        #git commit
        #git checkout bugFix
        #git rebase master
```

## Secuencia introductoria

Una breve introducción a la mayoría de los comandos de git

1: Introducción a los commits de Git



## Moviéndote por ahí con Git

Antes de meternos en algunas de las funcionalidades más avanzadas de git, es importante entender las distintas maneras de moverse por el árbol de commits que representa tu proyecto.

Una vez que estés cómodo moviéndote por ahí, tus poderes con los otros comandos de git ¡van a amplificarse!



Para completar este nivel, detacheemos HEAD de `bugFix` y atacheemosla al commit, en cambio.

Especifica este commit por su hash. El hash de cada commit se muestra en el círculo que lo representa.



In [5]: `#git checkout C4`

## Referencias relativas

Moverse por git usando los hashes de los commits puede volverse un tanto tedioso. En el mundo real no vas a tener una visualización de commits tan linda en la terminal, así que vas a tener que usar `git log` para ver los hashes.

Peor aún, los hashes en general son mucho más largos en el git real, también. Por ejemplo, el hash del commit que introdujo en el nivel anterior es `fed2da64c0efc5293610bdd892f82a58e8cbc5d8`. No es algo particularmente fácil de nombrar...

Lo interesante es que git es bastante astuto con los hashes. Sólo requiere que especifiques una cantidad de caracteres suficientes para identificar unívocamente al commit. Entonces, yo podría simplemente tipear `fed2` en lugar de esa cadena larga de arriba.



Para completar este nivel, haz checkout sobre el padre del commit de `bugFix`. Esto va a detachear a `HEAD`.

Puedes especificar el hash si quieres, pero mejor ¡trata de usar la referencia relativa!



In [1]: `#git checkout C3`



## El operador "~"

Digamos que quieres moverte un montón de niveles atrás en tu árbol de commits. Podría ser tedioso escribir `^` muchas veces, por lo que git tiene el operador `~`.

El operador `~` (opcionalmente) toma una cantidad que especifica la cantidad de padres que quieres volver hacia atrás. Veámoslo en acción



## Forzando las ramas

Ahora que eres un experto en las referencias relativas, *usémoslas* para algo.

Una de las formas más comunes en que uso las referencias relativas es para mover las ramas. Puedes reasignar directamente una rama a un commit usando la opción `-f`. Algo así como:

```
git branch -f master HEAD~3
```

Mueve (forzadamente) la rama master tres padres por detrás de HEAD.



Ahora que viste las referencias relativas y el forzar ramas combinados, usémoslos para resolver el siguiente nivel.

Para completar este nivel, mueve `HEAD`, `master` y `bugFix` a sus destinos finales.




```
In [1]: #git checkout C1  
#git branch -f master C6  
#git branch -f bugFix C0
```

## Revirtiendo cambios en git


Hay varias maneras de revertir cambios en git. Y, tal como al commitear, revertir cambios en git tiene tanto un componente de bajo nivel (indexar archivos o fragmentos individualmente) como un componente de alto nivel (cómo son efectivamente revertidos los cambios). Nuestra aplicación se va a concentrar en esto último.

Hay dos formas principales de deshacer cambios en git -- uno es usando `git reset` y el otro es usando `git revert`. Vamos a ver cada uno de ellos a continuación



Para completar este nivel, deshaz los dos commits más recientes, tanto en `local` como en `pushed`.

Ten en cuenta que `pushed` es una rama remota y `local` es una rama local -- eso debería ayudarte a elegir qué métodos usar.




```
In [2]: #git reset C1  
#git checkout pushed  
#git revert pushed
```

## Acelerando

La próxima ración de git. Espero que estés hambriento

1: Desatachea tu HEAD




## Git Cherry-pick

El primer comando en esta serie se llama `git cherry-pick`. Tiene la siguiente forma:


```
git cherry-pick <Commit1> <Commit2> <...>
```

Es una manera bastante directa de decir que quieres copiar una serie de commits sobre tu ubicación actual (`HEAD`). Personalmente amo `cherry-pick` porque hay muy poca magia involucrada y es bastante simple de entender.

¡Veamos una demo!



Para completar este nivel, simplemente copia algo de trabajo desde otras tres ramas a master. Puedes ver qué commits queremos en la visualización del objetivo.




In [3]: `#git cherry-pick C3 C4 C7`

## git rebase interactivo



`git cherry-pick` es genial cuando sabes qué commits quieres (y sabes sus hashes) -- es difícil superar la simpleza que provee.

Pero ¿qué pasa cuando no sabes qué commits quieres? Por suerte ¡git te cubre en esta situación, también! Podemos usar el rebase interactivo para esto -- es la mejor manera de revisar una serie de commits que estás a punto de rebasear.

Entremos en los detalles...



Para completar el nivel, haz un rebase interactivo y alcanza el orden que se muestra en la visualización objetivo. Recuerda que siempre puedes hacer `undo` y `reset` para arreglar errores :D



 

```
In [4]: #git rebase -i master~4 --aboveAll
```

## Moviendo el trabajo por ahí

Ponte cómodo cuando modifiques el directorio fuente

1: *Introducción a cherry-pick*

## Commits localmente stackeados

Esta es una escena que suele pasar cuando uno desarrolla: estoy tratando de encontrar un bug bastante escurridizo. Para ayudar en mi tarea de detective, agrego un par de comandos de debug, y algunas sentencias para imprimir el estado de mi sistema.

Todas estas cosas de imprimir y debuggear estan en su propia rama. Finalmente encuentro el problema, lo soluciono, ¡y disfruto!

El único problema es que ahora necesito llevar mi `bugFix` a la rama `master`. Si simplemente fast-forwardeo `master`, entonces `master` va a tener todos mis agregados de debugging, lo cual no es deseable. Tiene que haber otro modo...

Este es un nivel más avanzado, así que debes decidir cuál de los dos comandos quieres usar, pero para completar el nivel asegurate de que `master` recibe el commit que `bugFix` referencia.



```
In [5]: #git checkout master  
#git cherry-pick C4
```

## Haciendo malabares con los commits

Esta es otra situación algo común. Tienes algunos cambios (`newImage`) y otro conjunto de cambios (`caption`) que están relacionados, entonces están apilados en tu repositorio uno encima del otro (es decir, uno después del otro).

El tema es que a veces tienes que hacer una pequeña modificación a un commit previo. En este caso, la gente de diseño requiere que cambiemos ligeramente las dimensiones de `newImage`, ¡incluso aunque ese commit ya se encuentre atrás en nuestra historia!



```
In [6]: #git rebase -i caption~2 --aboveAll  
#git commit --amend  
#git rebase -i caption~2 --aboveAll  
#git branch -f master caption
```

## Haciendo malabares con los commits #2

*Si no completaste Haciendo malabares con Los commits #1 (el nivel anterior), hazlo antes de continuar*

Como viste en el último nivel, usamos `rebase -i` para reordenar los commits. Una vez que el commit que queríamos cambiar se encontraba arriba de todo, pudimos `--amend`earlo fácilmente y reordenarlo a como queríamos.

El único problema con esto es que hay mucho reordenamiento, que puede generar conflictos al rebasear. Veamos otro método usando `git cherry-pick`.

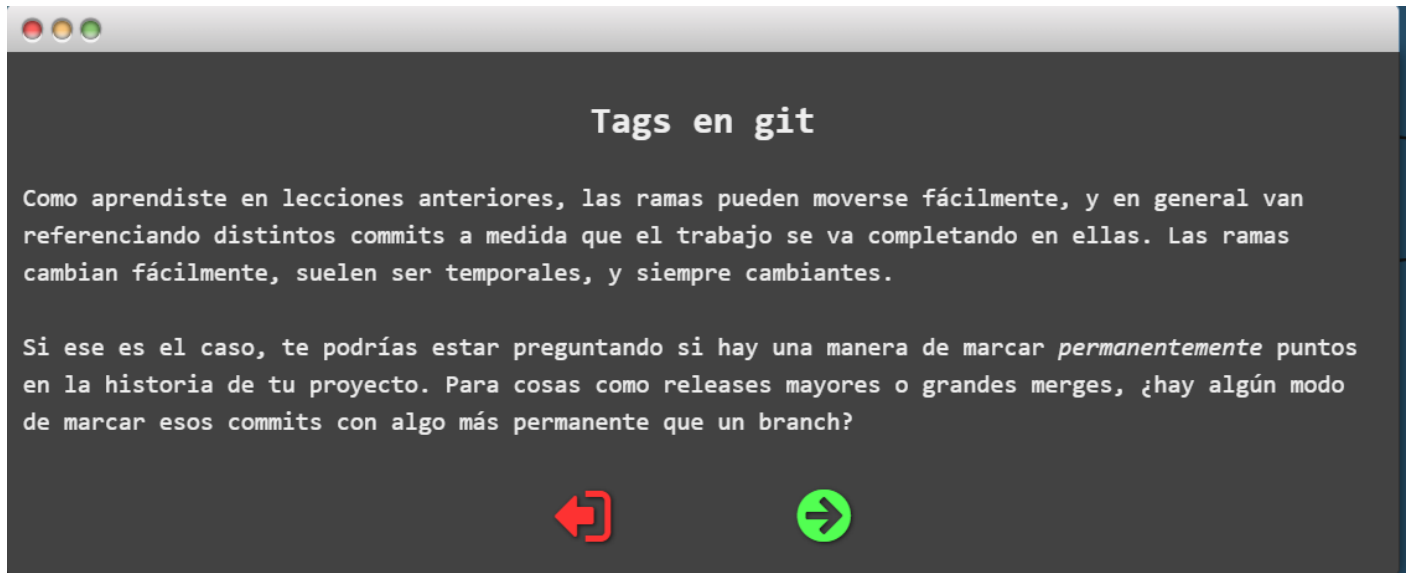


Entonces, en este nivel vamos a lograr el mismo objetivo de corregir `C2`, pero sin usar `rebase -i`. Te dejo a ti el darte cuenta cómo :D

Recuerda, la cantidad exacta de apóstrofes (') en el commit no es importante, sólo la diferencia relativa. Por ejemplo, le voy a dar una puntuación a un árbol que coincida con el objetivo pero cuyos commits tengan todos un apóstrofe extra.



```
In [7]: #git checkout master;
        #git cherry-pick C2;
        #git commit --amend;
        #git cherry-pick C3;
```

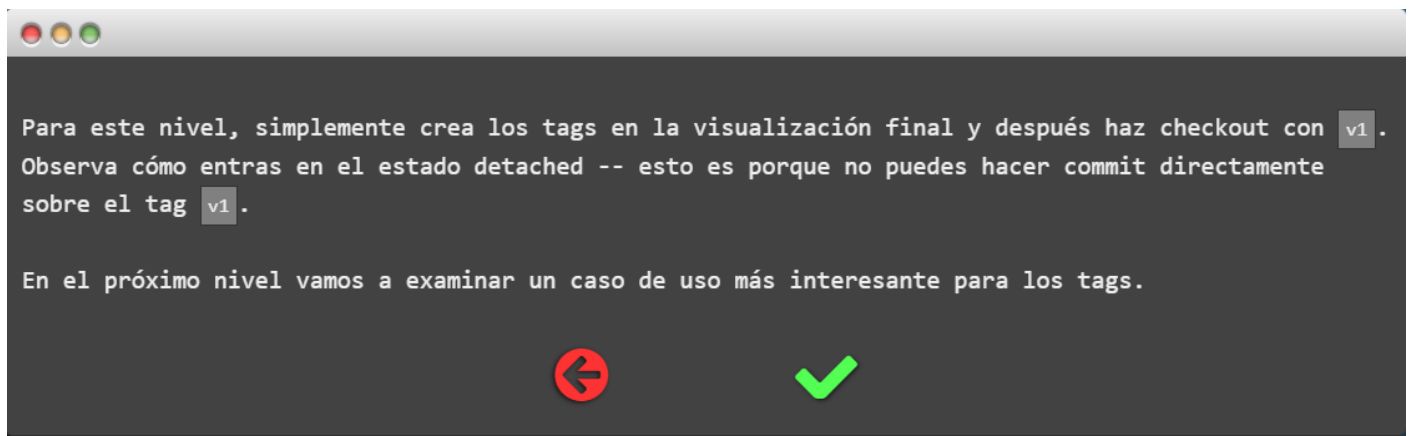


**Tags en git**

Como aprendiste en lecciones anteriores, las ramas pueden moverse fácilmente, y en general van referenciando distintos commits a medida que el trabajo se va completando en ellas. Las ramas cambian fácilmente, suelen ser temporales, y siempre cambiantes.

Si ese es el caso, te podrías estar preguntando si hay una manera de marcar *permanentemente* puntos en la historia de tu proyecto. Para cosas como releases mayores o grandes merges, ¿hay algún modo de marcar esos commits con algo más permanente que un branch?

Navigation icons: a red left arrow and a green right arrow.

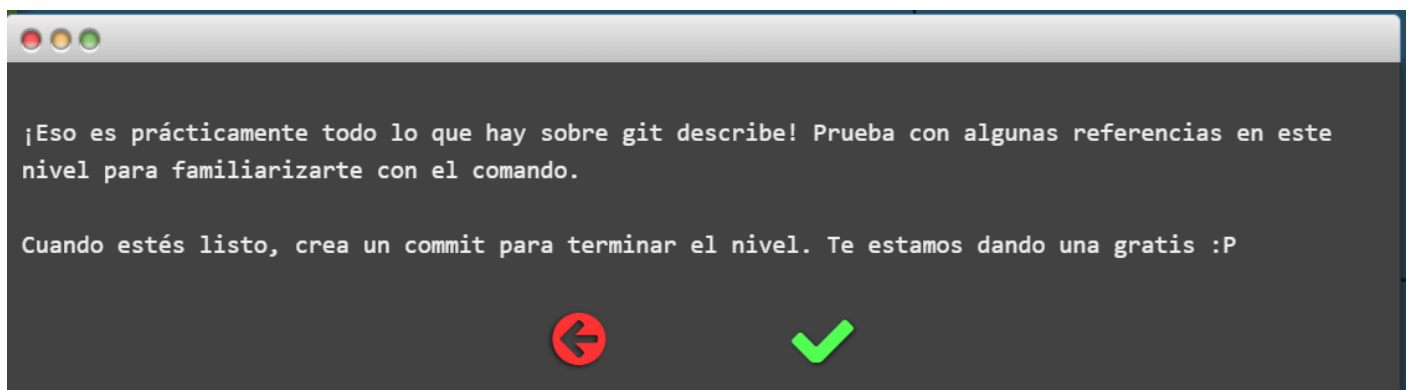
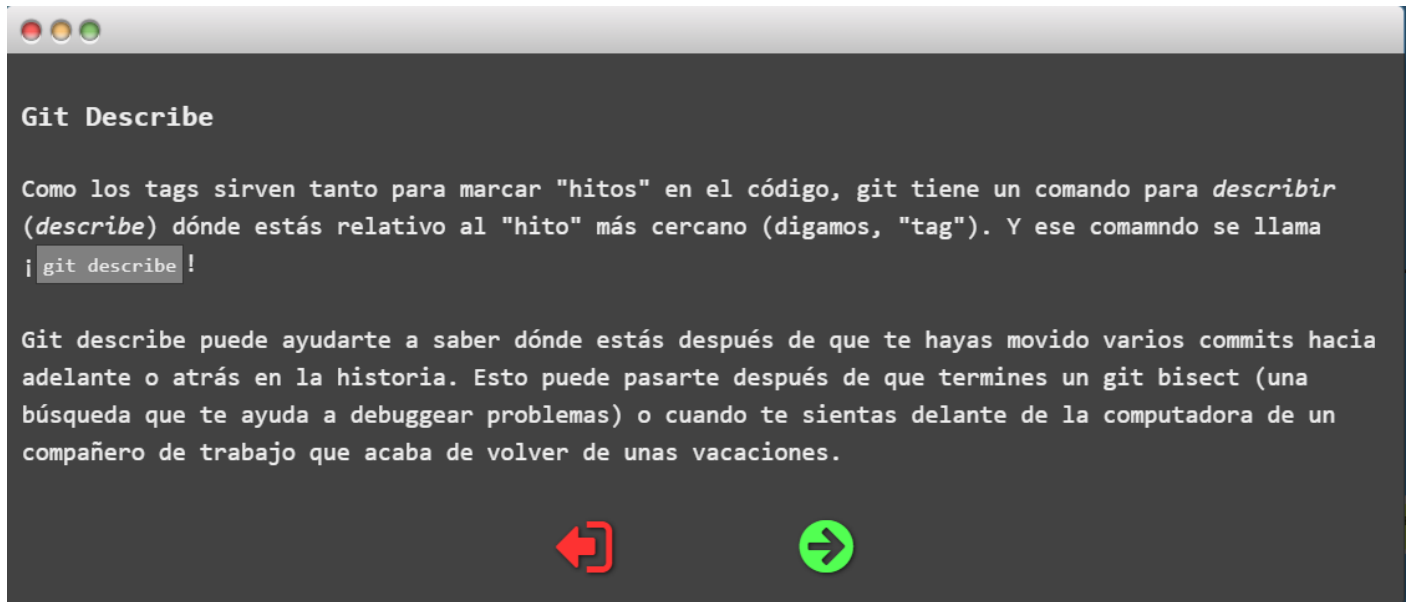


Para este nivel, simplemente crea los tags en la visualización final y después haz checkout con `v1`. Observa cómo entras en el estado detached -- esto es porque no puedes hacer commit directamente sobre el tag `v1`.

En el próximo nivel vamos a examinar un caso de uso más interesante para los tags.

Navigation icons: a red left arrow and a green checkmark.

```
In [8]: #git tag v0 C1  
        #git tag v1 C2  
        #git checkout C2
```



In [9]: `#git commit`



## Un poco de todo

Un batiburrillo de técnicas, trucos y sugerencias sobre Git

1: Tomando un único commit



### Rebaseando múltiples ramas

Fíjate, ¡hay un montón de ramas aquí! Rebaseemos todo el trabajo de esas ramas sobre master.

La gente de administración nos está haciendo las cosas un poco complicadas, igual -- quieren que nuestros commits estén todos en orden secuencial. Esto significa que nuestro árbol final tendría que tener `c7` al final, `c6` antes de ese, y así siguiendo, todos en orden.

Si te haces un lío durante el proceso, siéntete libre de usar `reset` para empezar de nuevo.

¡Asegúrate de verificar tu solución y ver si puedes realizarla en un número inferior de comandos!



```
In [11]: #git rebase master bugFix
          #git rebase bugFix side
          #git rebase side another
          #git rebase another master
```

## Especificando los padres

Como el modificador de `~`, `^` también acepta un número opcional después de él.

En lugar de especificar cuántas generaciones hacia atrás ir (como `~`), el modificador de `^` especifica por cuál de las referencias padres seguir en un commit de merge. Recuerda que un commit de merge tiene múltiples padres, por lo que el camino a seguir es ambiguo.

Git normalmente sigue el "primer" padre de un commit de merge, pero especificando un número junto con `^` cambia este comportamiento predefinido.

Demasiada charla, veámoslo en acción.



## Ponlo en práctica

Para completar este nivel, crea una nueva rama en la ubicación indicada.

Obviamente sería muy fácil especificar el commit directamente (algo como `c6`), pero te reto a usar los modificadores de los que estuvimos hablando, mejor.



In [12]: `#git branch bugWork master~^2~`


## Ensalada de ramas

¡Vaya, vaya! Tenemos un *pequeño* objetivo al que llegar en este nivel.

La rama `master` se encuentra algunos commits por delante de `one`, `two` y `three`. Por alguna razón, necesitamos actualizar esas tres ramas con versiones modificadas de los últimos commits de master.

La rama `one` necesita reordenarse, y eliminar `C5`. `two` necesita sólo reordenamiento, y `three` ¡sólo necesita un commit!

Vamos a dejar que intentes resolver este -- asegúrate de mirar la solución, después, usando `show solution`.




```
In [13]: #git checkout one
#git cherry-pick C4 C3 C2
#git checkout two
#git cherry-pick C5 C4 C3 C2
#git branch -f three C2
```

## Temas avanzados

¡Para los verdaderos valientes!

1: *Rebaseando más de 9000 veces*



Principal

Remota

## Secuencia introductoria

Una breve introducción a la mayoría de los comandos de git

1: *Introducción a los commits de Git*



## Acelerando

La próxima ración de git. Espero que estés hambriento

1: *Desatachea tu HEAD*



## Moviendo el trabajo por ahí

Ponte cómodo cuando modifiques el directorio fuente

1: *Introducción a cherry-pick*



## Un poco de todo

Un batiburrillo de técnicas, trucos y sugerencias sobre Git

1: *Tomando un único commit*



## Git Remotes

Los repositorios remotos no son *tan* complicados. En el mundo actual de la computación en la nube es bastante fácil pensar que hay un montón de magia detrás de los remotes de git, pero en realidad sólo son copias de tu repositorio en otra computadora. Típicamente vas a hablar con esta otra computadora a través de Internet, lo que permite transferir commits de un lado a otro.

Habiendo dicho eso, los repositorios remotos tienen un par de propiedades interesantes:

- Primero y principal, los remotos ¡son un backup genial! Los repositorios locales de git tienen la habilidad de restaurar archivos a un estado previo (como ya sabes), pero toda esa información se encuentra almacenada localmente. Al tener copias de tu repositorio git en otras computadoras, puedes perder todos tus datos locales y aún así retomar el trabajo en el punto donde lo habías dejado.
- Más importante, ¡los remotos sociabilizan la programación! Ahora que hay una copia de tu proyecto hosteada en otro lugar, tus amigos pueden contribuir a tu proyecto (o bajarse los últimos cambios) de un modo muy sencillo.

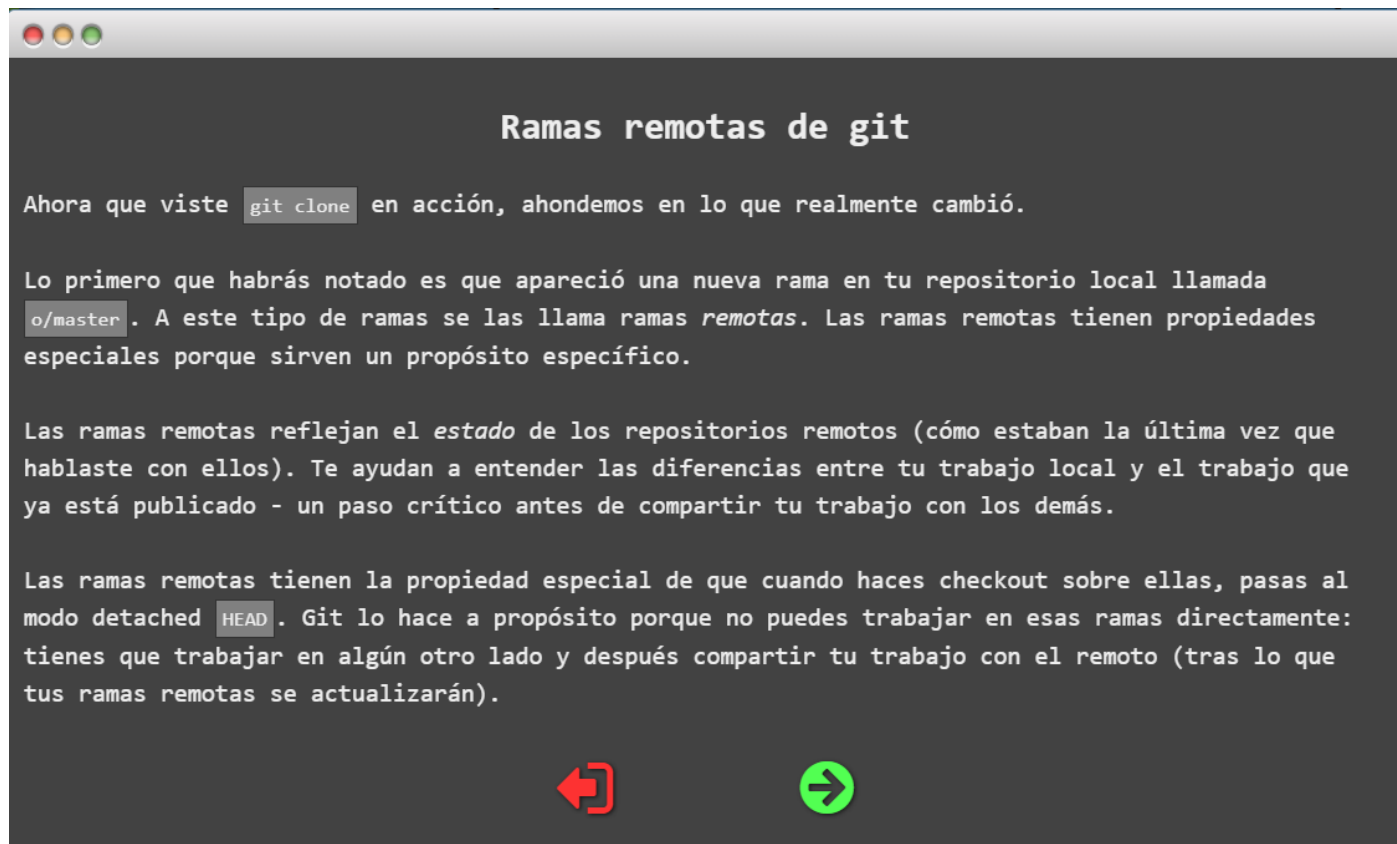
Se volvió bastante popular el uso de sitios web que muestran la actividad de los repositorios (como [GitHub](#) or [Phabricator](#)), pero esos repositorios remotos *siempre* sirven como la base subyacente de esas herramientas. Así que ¡es importante entenderlos!



Para completar este nivel, simplemente ejecuta `git clone` en tu repositorio existente. El verdadero aprendizaje viene en las próximas lecciones.



In [14]: `#git clone`



## Ramas remotas de git

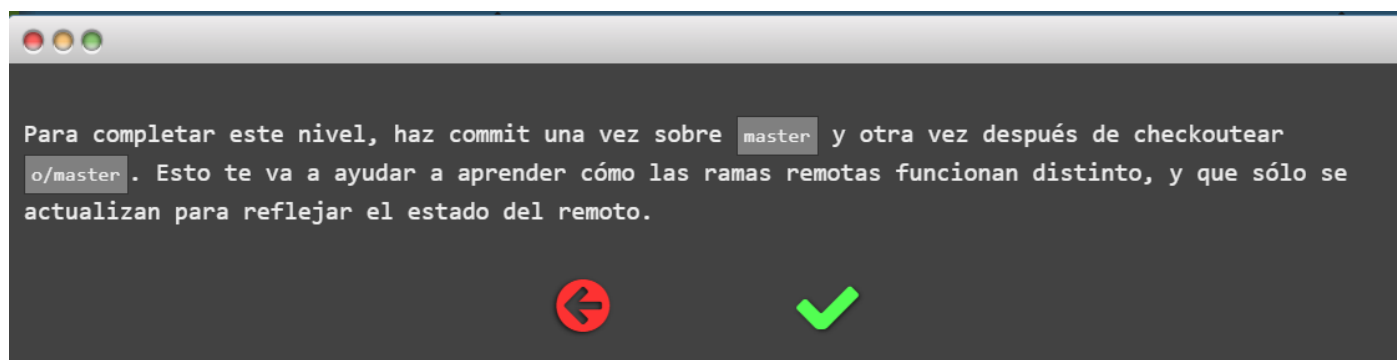
Ahora que viste `git clone` en acción, ahondemos en lo que realmente cambió.

Lo primero que habrás notado es que apareció una nueva rama en tu repositorio local llamada `o/master`. A este tipo de ramas se las llama ramas *remotas*. Las ramas remotas tienen propiedades especiales porque sirven un propósito específico.

Las ramas remotas reflejan el *estado* de los repositorios remotos (cómo estaban la última vez que hablaste con ellos). Te ayudan a entender las diferencias entre tu trabajo local y el trabajo que ya está publicado - un paso crítico antes de compartir tu trabajo con los demás.

Las ramas remotas tienen la propiedad especial de que cuando haces checkout sobre ellas, pasas al modo detached `HEAD`. Git lo hace a propósito porque no puedes trabajar en esas ramas directamente: tienes que trabajar en algún otro lado y después compartir tu trabajo con el remoto (tras lo que tus ramas remotas se actualizarán).

Navigation icons: a red left arrow and a green right arrow.



Para completar este nivel, haz commit una vez sobre `master` y otra vez después de checkoutear `o/master`. Esto te va a ayudar a aprender cómo las ramas remotas funcionan distinto, y que sólo se actualizan para reflejar el estado del remoto.

Navigation icons: a red left arrow and a green checkmark.

```
In [15]: #git commit
         #git checkout o/master
         #git commit
```

## Git Fetch

Trabajar con remotos en git en realidad se reduce a transferir datos *de* y *hacia* otros repositorios. Mientras podamos mandar commits de un lado al otro, podemos compartir cualquier tipo de actualización registrada por git (y, por ende, compartir trabajo, archivos nuevos, ideas nuevas, cartas de amor, etc).

En esta lección aprenderemos cómo traer (hacer `fetch`) datos *desde* un repositorio remoto - el comando para esto se llama, convenientemente, `git fetch`).

Vas a notar que a medida que actualicemos nuestra representación de nuestro repositorio remoto, nuestras ramas *remotas* van a actualizarse para reflejar esa nueva representación. Esto está ligado a la lección anterior sobre ramas remotas.



Para completar este nivel, simplemente ejecuta `git fetch` y bájate todos los commits.



In [16]: `#git fetch`

## Git Pull

Ahora que vimos cómo traer datos de un repositorio remoto con `git fetch`, ¡actualicemos nuestro trabajo local para reflejar esos cambios!

Realmente hay varias formas de hacer esto: una vez que tienes los commits disponibles localmente, puedes integrarlos como si fueran commits comunes de otras ramas. Esto significa que podrías ejecutar comandos como:

- `git cherry-pick o/master`
- `git rebase o/master`
- `git merge o/master`
- etc., etc.

De hecho, el flujo de trabajo de *fetchear* cambios remotos y después *mergearlos* es tan común que git incluye un comando que hace ambas cosas de una: ¡`git pull`!





Exploraremos los detalles de `git pull` después (incluyendo sus opciones y parámetros), pero por ahora probémoslo en este nivel.

Recuerda: puedes ejecutar este comando simplemente con `fetch` y `merge`, pero eso te costaría un comando extra :P



In [17]: `#git pull`

## Simulando la colaboración

Entonces, hay algo un poco tramposo -- para algunas de las lecciones siguientes, necesitamos explicarte cómo descargar cambios introducidos en el repositorio remoto.



Eso significa que esencialmente tenemos que "tener en cuenta" que el repositorio remoto fue actualizado por algún colega, amigo o colaborador tuyo, incluso a veces en alguna rama específica o una cantidad determinada de commits.

Para lograr esto, introdujimos el bien llamado comando `git fakeTeamwork`! Es bastante autoexplicativo: simula trabajo de nuestros colegas. Veamos una demo...



Los niveles siguientes van a ser algo difíciles, así que vamos a exigirte un poco más en este nivel.

Anímate y crea un remoto (con `git clone`), simula algunos cambios en ese remoto, haz commit en tu repo local, y luego haz pull de esos cambios. ¡Es como si fueran varias lecciones en una!

```
In [18]: #git clone
         #git fakeTeamwork master 2
         #git commit
         #git pull
```


## git push

Entendido, entonces ya descargué los cambios de un repositorio remoto y los integré en mi trabajo localmente. Esto suena muy bien... pero ¿cómo comparto *mis* cambios con el resto?

Bueno, la forma de subir el trabajo compartido es la opuesta a cómo descargar trabajo. Y ¿qué es lo opuesto a `git pull`? ¡`git push`!

`git push` es el responsable de subir *tus* cambios a un remoto específico y de actualizar ese remoto para incluir tus nuevos commits. Cuando `git push` termina, todos tus amigos pueden descargar tu trabajo del remoto.

Puedes imaginarte `git push` como un comando para "publicar" tu trabajo. Tiene un par de sutilezas con las que vamos a meternos pronto, pero empecemos poco a poco.

Para completar este nivel, simplemente comparte dos nuevos commits con el remoto. Igualmente, no te confíes, ¡las lecciones van a empezar a complicarse!



```
In [19]: #git clone  
#git commit  
#git commit  
#git push
```

## Trabajo divergente

Hasta ahora hemos visto cómo hacer pull a commits de otros y cómo hacer push a los nuestros. Parece bastante simple, así que ¿cómo puede confundirse tanto la gente?

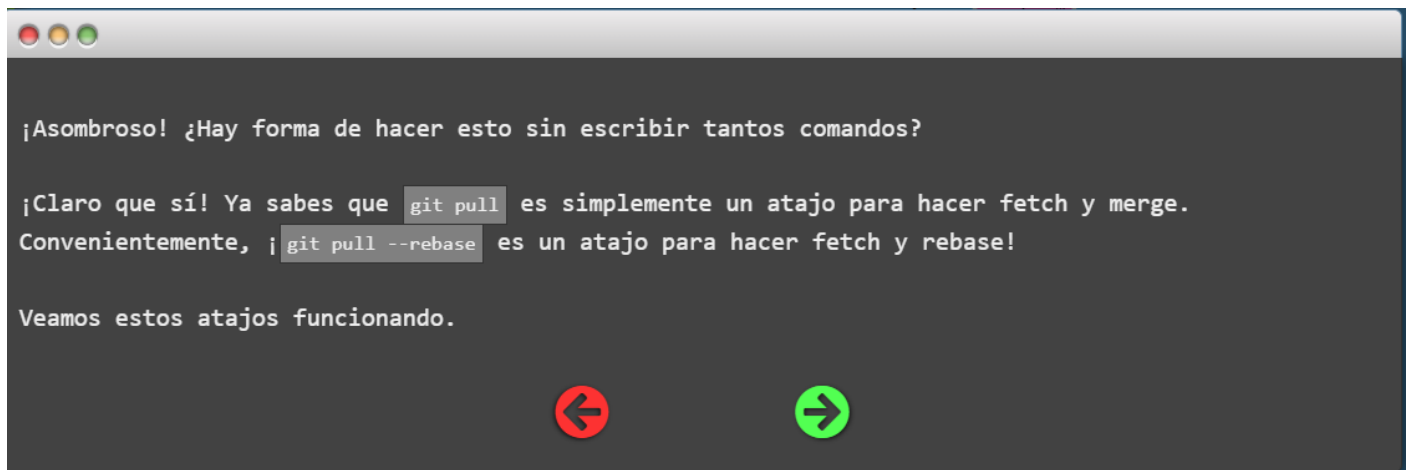
La dificultad viene cuando la historia de los repositorios *diverge*. Antes de entrar en detalles, veamos un ejemplo...



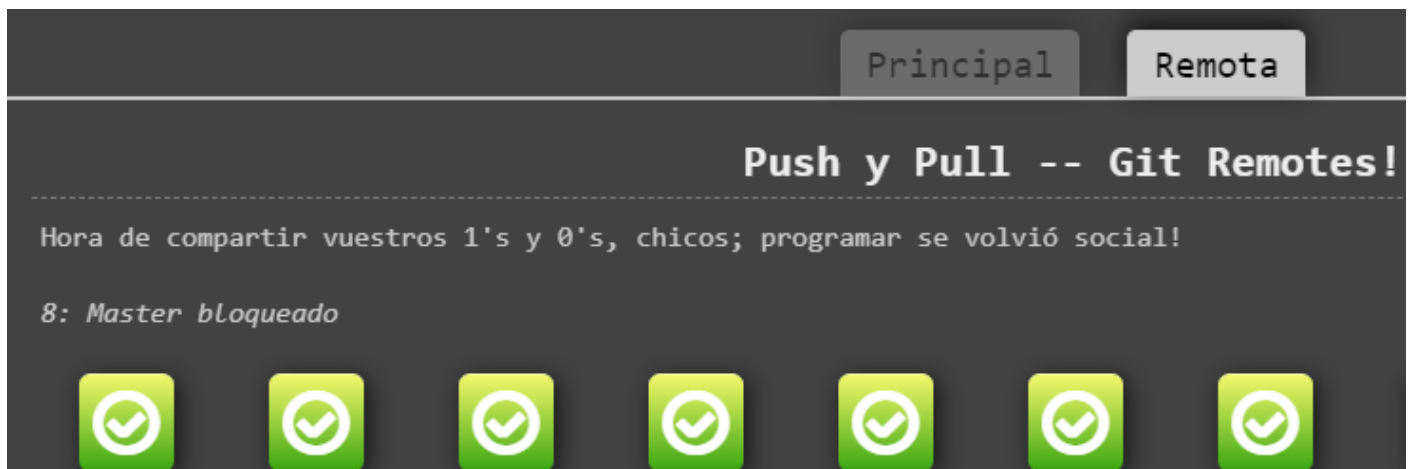
¿Cómo puedes resolver esta situación? Es fácil, todo lo que tienes que hacer es basar tu trabajo en la versión más reciente de la rama remota.

Hay un par de maneras de hacer esto, pero la más simple es mover tu trabajo haciendo un rebase. Probémoslo a ver cómo se ve.





```
In [20]: #git clone
         #git fakeTeamwork
         #git commit
         #git pull --rebase
         #git push
```



## Haciendo merge con ramas de trabajo

Ahora que estás cómodo fetcheando, pulleando y pusheando, pongamos a prueba estas habilidades con un nuevo flujo de trabajo.

Es bastante común que los desarrolladores en los grandes proyectos trabajen sobre ramas específicas para cada tarea (feature branches o ramas de trabajo) basadas en `master`, y que las integren sólo cuando estén listas. Esto es similar a la lección anterior, en la que hicimos push de las ramas periféricas al remoto, pero aquí tenemos un paso más.

Algunos desarrolladores sólo pushean y pullean cuando están en `master`: de ese modo, `master` siempre se mantiene actualizado con el remoto (`o/master`).

Entonces, en este flujo de trabajo combinamos dos cosas:

- integramos el trabajo de las ramas específicas a `master`, y
- pusheamos y pulleamos del remoto



Este nivel es bastante pesado. Aquí tienes algunas ideas para resolverlo:

- Tenemos tres ramas específicas -- `side1` `side2` and `side3`
- Queremos hacer push de cada una de esas ramas, en orden, al remoto
- El remoto fue actualizado, así que vamos a tener que integrar esos cambios también

:O ¡Genial! ¡Éxito! Completar este nivel representa un gran avance.



```
In [21]: #git fetch
#git rebase o/master side1
#git rebase side1 side2
#git rebase side2 side3
#git rebase side3 master
#git push
```

## ¿Por qué no hacer merge?

Para hacer push con tus novedades al remoto, todo lo que tienes que hacer es *integrar* los últimos cambios del remoto con los tuyos. Eso significa que puedes hacer tanto rebase como merge con la rama remota (por ejemplo, `o/master`).

Así que si puedes hacer cualquiera de las dos, ¿por qué las lecciones sólo se han centrado en rebasear hasta ahora? ¿Por qué no dedicarle algo de amor al `merge` cuando trabajamos con remotos?



Para este nivel, tratemos de resolver el nivel anterior, pero *mergeando*. Puede volverse un poco cuesta arriba, pero ilustra la idea bastante bien.



```
In [22]: #git checkout master  
         #git pull  
         #git merge side1  
         #git merge side2  
         #git merge side3  
         #git push
```

## Ramas que trackean remotos

Una de las cosas que te pueden haber parecido "mágicas" de las últimas lecciones es que git sabía que la rama `master` estaba relacionada con `o/master`. Obviamente, estas ramas tienen nombres parecidos, y podría parecer lógico conectar la rama `master` del remoto con la rama `master` local, pero esta conexión es bien evidente en dos escenarios:

- Durante una operación de pull, los commits se descargan a `o/master` y después se *mergean* a la rama `master`. El objetivo implícito del merge se determina con esta conexión.
- Durante un push, el trabajo de la rama `master` se sube a la rama `master` del remoto (que estaba representada localmente por `o/master`). El *destino* del push se determina con esta conexión entre `master` y `o/master`.



¡Perfecto! Para este nivel, haz push de tu trabajo a la rama `master` del remoto *sin* estar parado sobre `master` localmente. Te dejo que te des cuenta del resto solo, que para algo este es el curso avanzado :P



```
In [23]: #git checkout -b side o/master
         #git commit
         #git pull --rebase
         #git push
```





## Parámetros de push



¡Genial! Ahora que has aprendido sobre las ramas que trackean remotos podemos empezar a desvelar algo del misterio detrás de `git push`, `fetch` y `pull`. Vamos a atacar un comando cada vez, pero los conceptos entre ellos son muy similares.

Veamos primero `git push`. Ya aprendiste en la lección sobre ramas remotas que `git` determinó el remoto y la rama a la que pushear mirando las propiedades de la rama actual (el remoto al que "trackea"). Este es el comportamiento por defecto para cuando no se especifican parámetros, pero `git push` toma, opcionalmente, parámetros de la forma:

```
git push <remoto> <lugar>
```



Perfecto. Para este nivel, actualicemos tanto `foo` como `master` en el remoto. El tema está en que ¡tenemos deshabilitado `git checkout` en este nivel!



```
In [24]: #git push origin master  
         #git push origin foo
```



## Detalles sobre el parámetro `<lugar>`

Recuerda de la lección anterior que cuando especificamos `master` como el parámetro lugar de git push, especificamos tanto el *origen* del que sacar los commits como el *destino* al que enviarlos.

Podrías estar preguntándote ¿Y si quisiéramos que el origen y el destino fuesen distintos? ¿Si quisieras hacer push de los commits de la rama local `foo` a la rama `bar` del remote?



Bueno, lamentablemente eso no se puede hacer en git... ¡zasca! Claro que se puede :)... git es extremadamente flexible (casi casi que demasiado).

Veamos cómo hacerlo a continuación...



Para este nivel, trata de llegar al objetivo final, y recuerda el formato:

```
<origen>:<destino>
```



```
In [25]: #git push origin master~1:foo  
         #git push origin foo:master
```

## Parámetros de git fetch

Acabamos de aprender todo sobre los parámetros de push, como el parámetro `<lugar>`, e incluso las referencias separadas por dos puntos (`<origen>:<destino>`). ¿Podremos usar todo ese conocimiento para `git fetch`, también?

¡Dalo por hecho! Los parámetros para `git fetch` son realmente *muy, muy* similares a los de `git push`. Es el mismo tipo de conceptos, pero aplicados en la dirección opuesta (dado que ahora estás bajando commits en lugar de subirlos).

Veamos los conceptos de a uno en uno...

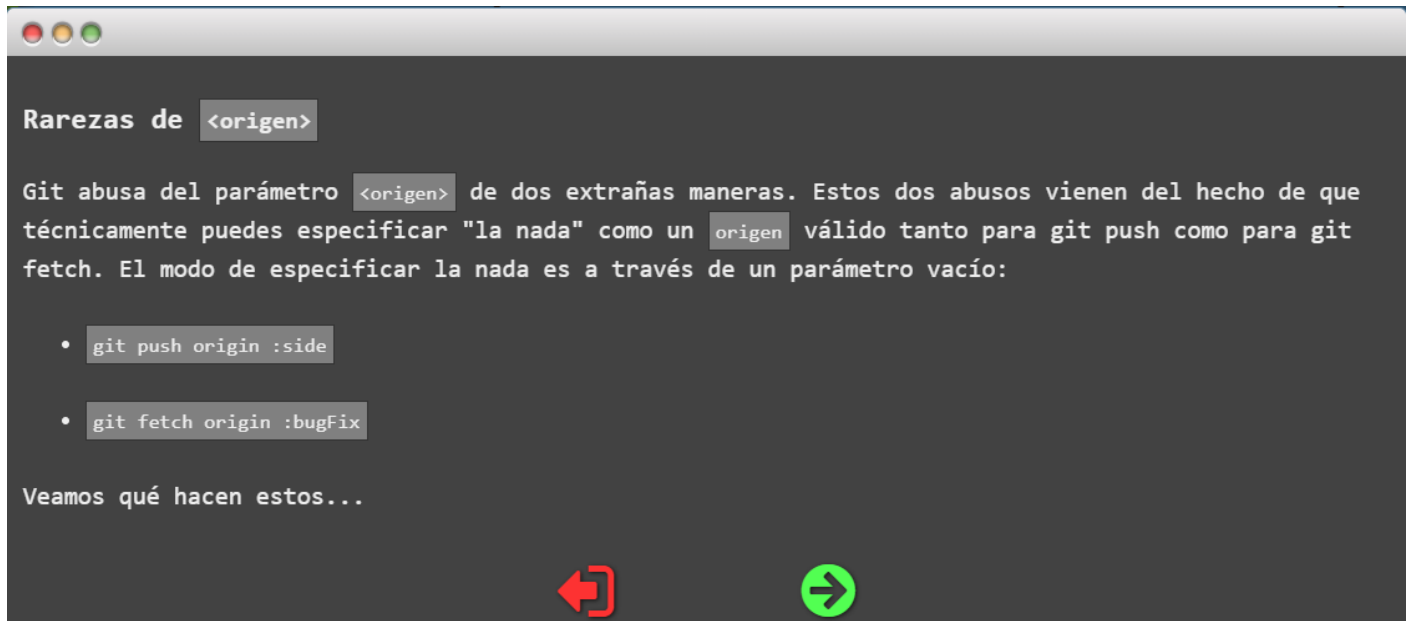


Bueno, demasiada charla. Para completar este nivel, haz fetch sólo de los commits especificados en la visualización del objetivo. ¡Familiarízate con esos comandos!

Vas a tener que especificar el origen y el destino para ambos comandos fetch. Presta atención al objetivo dado que ¡los IDs pueden estar invertidos!



```
In [26]: #git fetch origin master~1:foo
          #git fetch origin foo:master
          #git checkout foo
          #git merge master
```



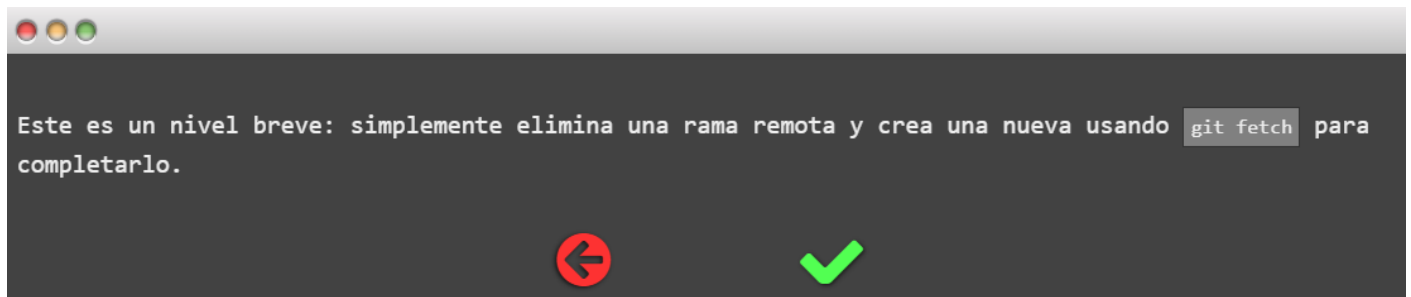
Rarezas de `<origen>`

Git abusa del parámetro `<origen>` de dos extrañas maneras. Estos dos abusos vienen del hecho de que técnicamente puedes especificar "la nada" como un `origen` válido tanto para `git push` como para `git fetch`. El modo de especificar la nada es a través de un parámetro vacío:

- `git push origin :side`
- `git fetch origin :bugFix`

Veamos qué hacen estos...

Navigation icons: a red left arrow and a green right arrow.



Este es un nivel breve: simplemente elimina una rama remota y crea una nueva usando `git fetch` para completarlo.

Navigation icons: a red left arrow and a green checkmark.

In [27]: `#git push origin :foo`  
`#git fetch origin :bar`

## Parámetros de git pull

Ahora que sabes prácticamente *todo* lo que hay que saber sobre los parámetros de `git fetch` y `git push`, casi no queda nada por cubrir del comando `git pull` :)

Eso es porque `git pull` es simplemente un atajo para hacer un `fetch` seguido de un `merge`. Puedes imaginártelo como ejecutar `git fetch` con los *mismos* parámetros, y después hacer `merge` de ello hacia donde esos `commits` hayan ido a parar.

Esto aplica incluso cuando utilizas parámetros rebuscados en exceso. Veamos algunos ejemplos:



OK: para terminar, intenta alcanzar el estado del objetivo. Vas a necesitar descargar algunos `commits`, crear algunas ramas nuevas, y `mergear` esas ramas junto con otras, pero no debería llevar demasiados comandos :P



```
In [28]: #git pull origin bar:foo
         #git pull origin master:side
```



In [ ]: