Contenido

[**learn-git-branching-solutions/MainCategorySolutions** 1](#_Toc64668114)

[Rebasando múltiples ramas 7](#_Toc64668115)

[**learn-git-branching-solutions/RemoteCategorySolutions** 9](#_Toc64668116)

[**¿Qué es o/?** 9](#_Toc64668117)

[La solución 13](#_Toc64668118)

[**----------------------------------GIT REMOTE AVANZADO---------------------------** 13](#_Toc64668119)

[¿Por qué no hacer merge? 14](#_Toc64668120)

[**Ramas que trackean remotos** 15](#_Toc64668121)

[**¿Puedo especificarlo yo?** 16](#_Toc64668122)

[**Forma número 2** 16](#_Toc64668123)

[Detalles sobre el parámetro <lugar> 17](#_Toc64668124)

[arámetros de git fetch 18](#_Toc64668125)

[**El parámetro <lugar>** 18](#_Toc64668126)

[**Rarezas de <origen>** 19](#_Toc64668127)

[Parámetros de git pull 19](#_Toc64668128)

[**learn-git-branching-solutions**](https://github.com/saivittalb/learn-git-branching-solutions)**/MainCategorySolutions**

|  |
| --- |
| 1.1 Introduction to Git Commits |
|  |  |
|  | git commit |
|  | git commit |
|  |  |
|  | 1.2 Branching in Git  Son sólo referencias a un commit específico |
|  | 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". |
|  | git branch bugFix // or git branch bugfix; checkout bugfix; git commit |
|  | git checkout bugFix |
|  |  |
|  | **1.3 Merging in Git**  **Unificar el trabajo de 2 branch: 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 escencialmente significa "Quiero incluir todo el trabajo de estos dos padres, y del conjunto de todos sus ancestros"** |
|  |  |
|  | git checkout -b bugFix //crear rama bugfix e ir a rama bugFix |
|  | git commit |
|  | git checkout master |
|  | git commit |
|  | git merge bugFix //Vamos a unificar o merge a la rama bugFix en main |
|  |  |
|  | **1.4 Rebase Introduction** |
|  | **Hacer rebase escencialmente selecciona un conjunto de commits, los "copia", y los aplica en algún otro lado. la ventaja de hacer rebase es que puede usarse para conseguir una secuencia de commits lineal, más bonita.** |
|  | git checkout -b bugFix |
|  | git commit |
|  | git checkout master |
|  | git commit |
|  | git checkout bugFix |
|  | git rebase master //para que el Branch en paralelo parezca secuencial con Branch master (haga un rebase a master y lo vez como referencia a un commit mas actual que el commit de master) si luego hacemos git rebase bugFix estando en master veremos que ambos se juntaran en vez de hacer el rebase o la referencia de commit de master a bugFix porque en este caso ya main era ancestro de bugFix |
|  |  |
|  | **2.1 Detach yo’ HEAD** |
|  | **HEAD es el nombre simbólico del commit sobre el que hemos hecho checkout -- es, básicamente, el commit sobre el que estás trabajando.**  **Normalmente HEAD apunta al nombre de una rama (como bugFix). Cuando creas un commit, el estado de bugFix se altera y este cambio es visible a través de HEAD.** |
|  | git checkout C4 //git show head muestra el head en el que estas trabajando o mas bien donde esta apuntando el puntero de nombre simbólico head en ese momento |
|  |  |
|  | **2.2 Relative Refs (^)**  **git log te permite ver los hashes que referencian los commit. Buscar los commits por su hash no es la manera más conveniente, y por eso git tiene referencias relativas. Con las referencias relativas puedes arrancar de algún lugar recordable (como la rama bugFix, o HEAD) y trabajar desde ahí.**  **Moverse un commit hacia atrás con ^**  **Moverse una cantidad de commits hacia atrás con ~<num>** |
|  |  |
|  | git checkout C4^ //moverse un commit hacia atras en el commit referenciado con el head C4 |
|  | Git checkout main^  Git checkout C3  Git checkout head^ |
|  | 2.3 Relative Refs #2 (~) **//para evitar tener que hacer x veces git checkout master^**  **Solo pones git checkout master~x**  **git branch -f master HEAD~3** === **Mueve (forzadamente) la rama main tres padres por detrás de HEAD** |
|  |  |
|  | git branch -f master C6 |
|  | git branch -f bugFix C0 |
|  | git checkout C1 //just to move the head to point to C1 |
|  |  |
|  | 2.4 Reversing Changes in 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** |
|  | git reset local~1 //git reset va a mover la referencia de la rama hacia atrás, como si el commit nunca se hubiera hecho. ~x cantidad de veces atrás del commit donde estas. Funciona para ramas locales, no para remotas |
|  | git checkout pushed |
|  | git revert pushed //git revert funciona para ramas remotas |
|  |  |
|  | **3.1 Cherry-pick Intro**  **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). O sea si te ubicas en el master Branch y quieres los commit de c2 y c4, se cargaran sobre la misma secuencia lineal de master como c2’ y c4’ resultando en c4’ como tu head y teniendo todos los cambios de c2 y c4** |
|  |  |
|  | git cherry-pick C3 C4 C7 //copier los cambios de C3.C4,C7 a master Branch (nunca salimos de master Branch) |
|  |  |
|  | **3.2 Interactive Rebase Intro**  **git cherry-pick es genial cuando sabes qué commits quieres. Cuando no sabes que commit quieres podemos usar el rebase interactivo para esto -- es la mejor manera de revisar una serie de commits que estás a punto de rebasear. Todo rebase interactivo significa usar el comando rebase con la opción -i**  **Si incluyes esta opción, git abrirá una UI (vim) para mostrarte qué commits están a punto de ser copiados sobre el objetivo del rebase. También muestra sus hashes y mensajes, que ayuda mucho para saber qué es cada commit.**  **Básicamente rebase copia los commit lo pone linear con el ancestro mas antiguo que escogiste y copia los commit que elegiste debajo de ese ancestro como C2’, C1’, C4’, etc en el orden q eligas en el VIM o editor que sale cuando pones el comando rebase.**  **Cuando el diálogo de rebase interactivo se abre, tienes la capacidad de hacer 3 cosas:**   * **Puedes reordenar los commits con solamente cambiar su orden en la UI (en nuestra ventana, eso significa hacer drag & drop con el mouse).** * **Puedes elegir ignorar completamente algunos commits. Esto se designa con pick -- no hacerle pick a algún commit significa que quieres ignorarlo.** * **Finalmente, puedes *squashear* commits. Desafortunadamente, nuestros niveles no soportan esto por cuestiones logísticas, por lo que voy a ahorrarte los detalles. Resumiendo, te permite combinar varios commits en uno solo.** * **Git rebase -i HEAD~4 copiara todos commit 4 ancestros atrás y lo pondrá por debajo del 4to ancestro** |
|  |  |
|  | * git rebase -i master~4 –aboveAll // Git rebase -i master~4 copiara todos commit 4 ancestros atrás y lo pondrá por debajo del 4to ancestro, selecciona orden C3,C5,C4 como dice en el objetivo en la ventana de selección. AboveAll aun no se para que es |
|  |  |
|  | **4.1 Grabbing Just 1 Commit**  **necesito llevar mi bugFix a la rama main. Si simplemente fast-forwardeo main, entonces main va a tener todos mis agregados de debugging, lo cual no es deseable. Tiene que haber otro modo.**  **Necesitamos decirle a git que sólo copie uno de los commits. Esto es tal como los niveles anteriores de mover commits por ahí -- podemos usar los mismos comandos:**   * **git rebase -i** * **git cherry-pick**   **Para conseguir este resultado.** |
|  |  |
|  | git checkout master |
|  | git cherry-pick C4 //simplemente fui a master branch y desde ahi jale los cambios de C4 con Cherry-pick, siendo C4 solo el commit con los cambios de bugFix Branch, sin los debug ni otros commit de cosas agregadas mientras hacia de detective |
|  |  |
|  | **4.2 Juggling 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!** |
|  | **hacer una pequeña modificación a un commit previo.**   * **Vamos a reordenar los commits para que el que queremos cambiar quede arriba de los demás con git rebase -i** * **Vamos a hacer git commit --amend para aplicar la ligera modificación** * **Después vamos a reordenar los commits a como estaban con git rebase -i** * **Finalmente, vamos a mover main a esta parte actualizada de nuestro árbol de commits para terminar el nivel (usando el método que prefieras)** |
|  | git rebase -i caption~2 –aboveAll //mover de manera iterativa el commit caption 2 commit atrás (te mueves con ~x veces) (y reordenar C3, C2 en vez de C2,C3) (aboveAll no es necesario en el juego) |
|  | git commit –amend //hacer el cambio en ese commit antiguo |
|  | git rebase -i caption~2 –aboveAll //mover de manera iterativa el commit caption 2 commit adelante (y reordenar C2, C3 en vez de C3,C2) |
|  | git branch -f master caption //mover main forzadamente (-f) a donde esta caption ahora y luego de haber modificado el commit antiguo y adquirir los cambios |
|  |  |
|  | **4.3 Juggling Commits #2**  **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 para hacer un cambio en ese commit antiguo fácilmente y luego reordenarlo denuevo a como queríamos y hacer que caption fuera el nuevo main con todos los cambios sobre el commit antiguo reordenado sobre el commit caption. El único problema con esto es que hay mucho reordenamiento, que puede generar conflictos al rebasear. Veamos otro método usando git cherry-pick.**  **Recuerda que git cherry-pick va a traer un commit de cualquier parte del árbol sobre HEAD (en el commit donde estas actualmente y siempre que ese otro commit no sea un ancestro de HEAD).**  **Entonces, en este nivel vamos a lograr el mismo objetivo de corregir C2, pero sin usar rebase -i.**  **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.** |
|  |  |
|  | git checkout master //poner tu head en la rama main con el commit C1 |
|  | git cherry-pick C2 //copiar el commit de C2 en tu head (aquí aparecera tu head sobre el commit C2 con un apostrofe ‘ ya que es la copia de newImage al cual le queremos hacer el cambio o commit) |
|  | git commit –amend //hacemos el cambio a esa copia de newImage |
|  | git cherry-pick C3 //jalar los cambios del commit C3 a tu head donde está la copia con los cambios sobre el commit C2 del Branch newImage. De esta manera estábamos en el commit C3 de nombre caption, nos fuimos atrás en el tiempo al commit newImage, le hicimos un cambio a ese commit antiguo, y volvimos adelante en el tiempo y fundimos el cambio anterior con lo que hay en el commit de nombre caption, y como el primer paso fue poner el head en main, significa que seguimos en main y todos los cambios ahora están en la rama main |
|  |  |
|  | **4.4 Git Tags**  **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.** |
|  | **¿hay algún modo de marcar esos commits con algo más permanente que un branch? Si, Los tags de git soportan exactamente este caso de uso – marcan permanentemente determinados commits como "hitos" que puedes referenciar como a un branch.** |
|  | git tag v0 C1 //Nombramos al tag v0 y referenciamos explícitamente al commit C1. Si no especificas el commit, git va a usar al apuntado por HEAD. |
|  | git tag v1 C2 (el commit sobre el que estes se le asigna el tag si no especificas).  git tag v1 C2 |
|  | git checkout V1 // Puedes hacer checkout sobre V1 pero no puedes hacer commit directamente sobre el tag v1 |
|  |  |
|  | **4.5 Git Describe**  **git tiene un comando para describir (describe) dónde estás relativo al Tag más cercano** |
|  | **Git describe puede ayudarte a saber dónde estás después de que te hayas movido varios commits hacia adelante o atrás en la historia. Esto puede pasarte después de que termines un git bisect (una búsqueda que te ayuda a debuggear problemas)**  **Git describe tiene la siguiente forma:**  **git describe <ref>**  **Donde <ref> es cualquier cosa que git puede resolver a un commit. Si no especificas ninguna referencia, git simplemente usa el commit en el que estás parado ahora (HEAD).**  **La salida de ese comando se ve así:**  **<tag>\_<numCommits>\_g<hash>**  **Donde tag es el tag más cercano en la historia, numCommits dice a cuántos commits de ese tag estás, y <hash> es el hash del commit que estás describiendo** |
|  | git commit //solo un commit… nos regalaron ese punto lol… pero probando git describe main, nos mostrara el tag cercano V0, nos dira que esta a 2 commit de distancia en el pasado y nos dará el hash de ese commit que tiene como tag V0. |
|  |  |
|  | 5.1 Rebasing over 9000 times |
|  | Rebasando múltiples ramas Rebasemos todo el trabajo de todas las ramas sobre main.  nuestros commits deben estar 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 commandos |
|  | git rebase master bugFix //como aparecimos con el head en main ya no tenemos que movernos a esa rama, con este comando rabasamos a nuestro head que esta en main con el commit C3 de la rama bugFix, quedando C2, C3 y ahora debemos hacer nuevamente lo mismo para que quede C2,C3,C4 y así sucesivamente |
|  | git rebase bugFix side //para que la rama side con el commit C6 rebase y se ponga de manera secuencial con la rama bugFix con el commit C3. Dado que C6 tiene C5,C4 como ancestros, quedara C1,C2,C3,C4,C5,C6 |
|  | git rebase side another //para que la another side con el commit C7 rebase y se ponga de manera secuencial con la rama side con el commit C6. Dado que C7 tiene los mismos ancestros de C6: C5,C4, estos se omiten y quedara C1,C2,C3,C4,C5,C6,C7 |
|  | git rebase another master // para que la rama main que sigue siendo C2, rebase a todos por delante de la rama another el cual es el commit C7. Así obtendremos todos los commit de manera secuencial y la rama main va a adquirir todos los cambios y parecera que siempre se trabajo solo con rama main. |
|  |  |
|  | **5.2 Multiple Parents**  **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.**  Aquí tenemos un commit de merge. Si hacemos checkout de main^, sin modificadores, vamos a seguir al primer padre después del commit de merge.  (*En nuestras visualizaciones, el primer padre se ubica directamente arriba del commit de merge.*)  **Git checkout main^ te mueve el head un commit en el pasado**  **Git checkout main^2 te mueve el head 2 commit en el pasado, peero si main es producto de un merge de 2 padres, este 2do commit en el pasado sera el otro padre con el cual se hizo el merge.**  **Git checkout HEAD~; git checkout HEAD~2; git checkout HEAD~2 esto significa mueve el HEAD un commit antes, luego mueve el HEAD 2 commit antes, y finalmente mueve el HEAD 2 commit antes**  **Git checkout HEAD~^2~2 es un encadenado de modificadores, esto significa, vuelve tu head en 1, luego devuélvete 2 commit, luego devuélvete 2 commit nuevamente** |
|  |  |
|  | git branch bugWork master~^2~ //Crear nuevo branch bugWork en el 4 commit mas antiguo y llevarlo o copiar los cambios al Branch main. ~^2 te lleva un commit atrás y te pasa de una rama del Merge a la otra rama del merge.  también puedes hacer git checkout main~^2~; git Branch bugWork; git rebase bugWork main |
|  |  |
|  | **5.3 Branch Spaghetti** |
|  | **¡Vaya, vaya! Tenemos un *pequeño* objetivo al que llegar en este nivel.**  **La rama main 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 main.**  **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.** |
|  | git checkout one |
|  | git cherry-pick C4 C3 C2 //tomar los commit C4,C3,C2 y ponerlos en secuencia con la rama one |
|  | git checkout two |
|  | git cherry-pick C5 C4 C3 C2 //tomar los commit C5,C4,C3,C2 de rama main y ponerlos en secuencia con la rama two |
|  | git branch -f three C2 //tomar a la rama three y forzarla a cargar sus cambios en el commit C2 |

[**learn-git-branching-solutions**](https://github.com/saivittalb/learn-git-branching-solutions)**/RemoteCategorySolutions**

|  |
| --- |
| **1.1 Clone Intro** |
|  | **git clone en el mundo real es el comando que usarías para crear copias locales de un repositorio remoto (uno de GitHub, por ejemplo). Aquí usamos este comando de un modo un tanto distinto, en cambio -- git clone va a crear un repositorio remoto a partir del tuyo local. Estamos de acuerdo en que es el significado técnicamente opuesto al del comando real, pero ayuda bastante a entender la conexión entre el clonado y el trabajo en un repositorio remoto, así que vamos a vivir con ello por ahora.** |
|  | git clone //para hacer totalmente lo puesto a lo que debes hacer lol. En vez de clonar un repositorio remoto, en el juego clonamos repositorios locales… pero igual sirve |
|  |  |
|  | **1.2 Remote Branches** |
|  | **Lo primero que habrás notado es que apareció una nueva rama en tu repositorio local llamada o/main. 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).** **¿Qué es o/?** **las ramas remotas también tienen una convención de nombres obligatoria -- se muestran con el formato:**   * **<nombre del remoto>/<nombre de la rama>**   **La mayoría de los desarrolladores llaman origin a su remoto en lugar de o. Esto es tan común que git efectivamente crea tu remoto llamándolo origin cuando haces git clone de un repositorio.** |
|  | git commit |
|  | git checkout o/master |
|  | git commit |
|  | ////git nos puso en el modo detached HEAD y no actualizó o/main cuando creamos un nuevo commit, en vez de esto solo creo el commit C4 con el HEAD apuntando a el pero fuera del Branch remoto origin/main U o/main. Esto ocurre porque o/main sólo va a actualizarse cuando el remoto se actualice. |
|  | **1.3 Git Fetchin’**  **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.** |
|  | **git fetch hace dos simples pasos, y sólo dos simples pasos:**   * **baja los commits que el remoto tiene pero no están en nuestro repositorio local, y...** * **actualiza a dónde apuntan nuestras ramas remotas (por ejemplo, o/main)**   **git fetch escencialmente sincroniza nuestra representación local del repositorio remoto con el verdadero estado del repositorio remoto (en este momento).**  **Usualmente, git fetch habla con el repositorio a través de internet (usando un protocolo como http:// O git://).**  **Sin embargo, git fetch no modifica en absoluto tu estado local. No va a actualizar tu rama main ni va a cambiar nada sobre cómo se ve tu sistema de archivos en este momento. Entonces, después de todo, puedes pensar sobre git fetch como un proceso de descarga.** |
|  | git fetch //descargar los commit sin afectar ni cambiar el repositorio local |
|  |  |

|  |  |
| --- | --- |
|  | **1.4 Git Pullin’**  **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 (luego de que descargas los commit con fetch), puedes integrarlos como si fueran commits comunes de otras ramas. Esto significa que podrías ejecutar comandos como:**   * **git cherry-pick o/main** * **git rebase o/main** * **git merge o/main** * **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!** |
|  | git pull //hacer el fetch o descarga de commit del repositorio remoto y también hacer el merge de los commit descargados en el repositorio local |
|  | //también puedes hacer por separado con git fetch; git merch o/main y también podrás descargar y hacer merge a todos los cambios del repositorio remoto en el repo local |
|  | **1.5 Fakeing Teamwork** |
|  | **hay algo un poco tramposo -- para algunas de las lecciones siguientes, necesitamos explicarte cómo descargar cambios introducidos en el repositorio remoto.**  **Eso significa que escencialmente 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. (Básicamente es un commit al repositorio remoto )** |
|  | git clone |
|  | git fakeTeamwork master 2 //hacer 2 commit en branch main en repositorio remote |
|  | git commit //hacer commit en main local |
|  | git pull //hacer fetch y merge del main remote en el main local |
|  |  |
|  | **1.6 Git Pushin’** |
|  | **¿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.** |
|  | git commit |
|  | git commit |
|  | git push //luego de hacer los 2 commit se puede ver como se actualiza el o/main en el repositorio local indicando que ya el main del repositorio remoto tiene los cambios |
|  |  |
|  | **1.7 Diverged History** |
|  | **Cuando trabajas sobre un respositorio remoto que luego tiene cambios y tu nuevos commit están basados sobre un repositorio remoto antiguo y desactualizado**  **¿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.**  **Podemos hacerlo con git fetch; git rebase o/main; git push. De esta manera descargamos los commit del repositorio remoto, con el rebase ponemos los commit nuevos del repositorio remoto por detrás de nuestro main local y luego ya si podemos hacer git push porque nuestro repo local ya esta actualizado con el repo remoto.**  **-otro way de hacerlo:**  **Por más que git merge no mueva tu trabajo (sólo crea un commit de merge), es un modo de decirle a git que integraste todos los cambios del remoto. Esto es porque ahora una rama remota pasó a ser un ancestro de tu propia rama, lo que significa que tu commit refleja los cambios de todos los commits de la rama remota.**  **Podemos hacerlo con git fetch; git merge o/main; git push. De esta manera descargamos los cambios del repo remoto y lo fusionamos con nuestro repo local con merge de o/main sobre nuestro main local y ya si nos deja hacer push.** |
|  | **-otro way mas simple de hacerlo: git pull –rebase. De este modo ya hacemos el fetch y el merge juntos con git full y al aplicar el rebase incluimos los cambios del repo remoto por detrás de nuestro main local**  **-otro way todavía mas simple de hacerlo: git pull; git push. El –rebase es necesario solo si lo quieres todos los cambios de manera secuencial o lineal. Git pull por si solo ya te incluye todos los cambios del repo remoto sin alterar tu ultimo commit local**  **Toda este tinglado de hacer fetch, rebase/merge y push es bastante común. En lecciones futuras vamos a ver formas más complejas de estos flujos de trabajo, pero por ahora vamos a probar esto que acabamos de ver.**  **Para resolver este nivel, haz lo siguiente:**   * **Clona tu repositorio** * **Simula algo de trabajo de un colega (1 commit)** * **Haz commit de algún cambio tuyo (1 commit)** * **Publica tu trabajo en secuencial con rebase o *rebaseando*** |
|  | git clone |
|  | git fakeTeamwork master 1 |
|  | git commit |
|  | git pull —rebase |
|  | git push |
|  |  |
|  | **1.8 Locked branch Master** |
|  | **Si trabajas en un equipo colaborativo, es probable que la rama main esté bloqueada y requiera algún proceso de Pull Request para poder mergear los cambios. Si haces commit directamente a main e intentas realizar push, recibirás un mensaje similar a este:**  **! [remote rejected] main -> main (TF402455: Pushes to**  **this branch are not permitted; you must use a pull request to update**  **this branch.)**  **Trataste de crear una rama y luego hacer pushs creando un Pull Request, pero te olvidaste e hiciste commit directamente a main. Ahora estás atascado y no puedes hacer push de tus cambios.** La solución **Crea otra rama llamada feature y haz push a remoto. También restablece la rama main nuevamente para que esté sincronizado con el repositorio remoto; de lo contrario, puedes tener problemas la próxima vez que realices un pull y el commit de otra persona entre en conflicto con el tuyo.** |
|  | git reset --hard o/master //devolverte forzadamente en este caso del commit en Branch main a el ultimo cambio en donde estaba cuando copiaste del repositorio remoto, es decir a donde esta apuntando o/main ahora, que es la imagen del repositorio remoto en tu repositorio local en el momento. |
|  | git checkout -b feature C2 //crear y entrar en rama nueva “feature” en el commit referenciado como C2 |
|  | git push origin feature //hacer push de feature, o sea tomar los últimos cambios en la rama feature y subirlos al repositorio remoto, creando así una referencia de ese Branch remoto feature en el repositorio local como o/feature y también superponiendolo como ultimo cambio en ambos repositorios  **----------------------------------GIT REMOTE AVANZADO---------------------------** |
|  |  |
|  | **2.1 Push Master!** |
|  | **Algunos desarrolladores sólo pushean y pullean cuando están en main: de ese modo, main siempre se mantiene actualizado con el remoto (o/main).**  **Entonces, en este flujo de trabajo combinamos dos cosas:**   * **integramos el trabajo de las ramas específicas a main, 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** |
|  | git fetch //solo descarga los últimos cambios o commit del repositorio remoto en tu repo local |
|  | git rebase o/master side1 //poner de manera secuencial por debajo de o/main todo el trabajo del Branch side1. Es decir tomar los cambios del Branch side1 y agregarlo de manera lineal como cambio mas actual de la rama que referencia al repo remoto o/main |
|  | git rebase side1 side2 //denuevo lo anterior, tomar todo Branch side 2 (todos los commit) y ponerlo como los cambios mas actuales de manera lineal sobre el Branch side1 |
|  | git rebase side2 side3 //denuevo lo anterior, tomor todo el Branch side3 y poner todos sus cambios sobre el Branch side2 |
|  | git rebase side3 master //mover el HEAD de Branch main a donde esta el ultimo commit del Branch side3 el cual ya tiene todos los commit en manera secuencial de los Branch side2, side1, o/main |
|  | git push //copiar todos los cambios del repo local al repositorio remoto |
|  |  |
|  | **2.2 Mering with remotes / haciendo merge con los remotos** |
|  | ¿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/main).**  **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?**  **Hay mucho debate entre los desarrolladores sobre los pros y contras de mergear vs rebasear. Aquí te mostraré los pros y contras de rebasear:**  **Pros:**   * **Rebasear hace que tu árbol de commits se vea bastante limpio, porque todos los commits siguen una única línea**   **Contras:**   * **Rebasear modifica la historia (aparente) de tu árbol de commits.**   **Por ejemplo, el commit C1 puede rebasearse para que aparezca *después* de C3. Entonces, parece que el trabajo de C1' se hizo después de C3, cuando en realizad se había hecho antes.**  **Algunos desarrolladores prefieren preservar la historia, por lo que deciden mergear. Otros prefieren tener un árbol de commits limpios, y preferimos rebasear. Todo es una cuestión de preferencias.** |
|  | git checkout master |
|  | git pull //adquirir el ultimo cambio del repo remoto (git fetch + git merge) descargo y fusiono |
|  | git merge side1 //hacer merge en main con el Branch side1 |
|  | git merge side2 //hacer merge en main con el Branch side2 |
|  | git merge side3 //hacer merge en main con el Branch side3 |
|  | git push //tendrás todo un tollo en comparación con hacerlo con rebase en vez de merge porque no estará de manera lineal... pero igual funciona |
|  |  |
|  | **2.3 Remoting Tracking** |
|  | **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 main estaba relacionada con o/main. Obviamente, estas ramas tienen nombres parecidos, y podría parecer lógico conectar la rama main del remoto con la rama main local, pero esta conexión es bien evidente en dos escenarios:**   * **Durante una operación de pull, los commits se descargan a o/main y después se mergean a la rama main. El objetivo implícito del merge se determina con esta conexión.** * **Durante un push, el trabajo de la rama main se sube a la rama main del remoto (que estaba representada localmente por o/main). El destino del push se determina con esta conexión entre main y o/main.**   **Resumiendo, esta conexión entre main y o/main se explica simplemente por la propiedad de "trackear (seguir) remotos" de las ramas. La rama main está configurada para trackear o/main -- osea, que hay un objetivo implícito para el merge y un destino implícito para de la rama main.**  **Podrías estar pensando cómo esa propiedad apareció en tu rama main si no ejecutaste ningún comando para especificarlo. Bueno, cuando clonas un repositorio con git, esta propiedad es asignada por ti automáticamente.**  **Durante un clone, git crea una rama remota por cada rama en el remoto (por ejemplo, ramas como o/main). Pero después crea una rama local que trackea la rama activa del remote, que suele ser main.**  **Una vez completado el git clone, sólo tienes una única rama local (para que no te asustes) pero puedes ver todas las ramas del remoto (si fueses curioso). ¡Es lo mejor de ambos mundos!**  **Esto también explica por qué podrías ver un mensaje como este durante la clonación:**  **local branch "main" set to track remote branch "o/main"**  **rama local "main" establecida para trackear la rama remota "o/main"** **¿Puedo especificarlo yo?** **¡Claro que sí! Puedes hacer que cualquier rama que quieras trackee o/main, y si lo hicieras, esa rama va a tener el mismo destino implícito de push y objetivo implícito de merge que main. Eso significa que puedes ejecutar git push en una rama llamada absolutamenteNoEsMaster y ¡que tu trabajo se pushee a la rama main del remoto!**  **Hay dos formas de establecer esta propiedad. La primera es hacer checkout sobre una nueva rama usando una rama remota como la referencia especificada. Ejecutar**  **git checkout -b absolutamenteNoEsMaster o/main**  **Crea una nueva rama llamada absolutamenteNoEsMaster y la hace trackear a o/main.**  **Ej: Git checkout -b foo o/main; git pull //crear e ir a rama foo y hacer que trackee rama o/main,**  **Ej con git push: git checkout -b foo o/main; git commit; git push //** **Forma número 2** **Otra forma de especificar la rama a trackear es usar la opción git branch -u. Ejecutando**  **git branch -u o/main foo**  **establecemos que la rama foo trackee a o/main. Si encima estás parado en foo, incluso puedes obviarlo:**  **git branch -u o/main**  **ej: git Branch -u o/main foo; git commit; git push;** |
|  | git checkout -b side o/master //crear e ir a rama side y hacer que rama side trackee o/master |
|  | git commit |
|  | git pull –rebase //al hacer este pull, rama side se suponer pone de manera secuencial por el rebase con rama o/main por q la esta trackeando |
|  | git push //se sube todo y queda rama o/main con rama side en el mismo commit, rama main nunca cambia en el local, mas en el repo remoto si. |
|  |  |
|  | **2.4 Git push arguments** |
|  | **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>**  **git push origin main**  **se traduce así al español:**  ***Vete a la rama llamada "main" en mi repositorio, coge todos los commits, y después vete a la rama "main" del remoto llamado "origin". Copia ahí todos los commits que falten, y avísame cuando termines.***  **Especificando main como el parámetro "lugar", le dijimos a git de dónde traer los commits, y a dónde mandarlos. Es, básicamente, el "lugar" o "ubicación" que sincronizar entre ambos repositorios.**  **Ten en cuenta que, como le dijimos a git todo lo que necesitaba saber (especificando ambos parámetros), ¡ignora totalmente dónde estamos parados en este momento¡**  **Ej: git checkout C0; git push origin main**  **Ej 2: git checkout C0; git push //aqui el commando falla porque no especificamos ninguna rama que apunte a algún remoto y por ende el HEAD esta en el commit C0 pero no se sabe que cambios van a subir** |
|  | git push origin master //copiar los cambios de Branch main en repo remoto |
|  | git push origin foo //copiar todos los cambios de Branch foo en repo remoto. Ojo como no se hizo Merge solo tendrás 2 branch separados en repo remoto |
|  |  |
|  | **2.5 Git push arguments — Expanded!** |
|  | Detalles sobre el parámetro <lugar> **Recuerda de la lección anterior que cuando especificamos main 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?**  **Para especificar tanto el origen como el destino de <lugar>, simplemente únelos usando el signo dos puntos:**  **git push origin <origen>:<destino>**  **A esto se le suele llamar refspec con dos puntos. Refspec es simplemente un nombre genial para una ubicación que git puede entender (como la rama foo, o incluso HEAD~1)**  **Ej: git push origin foo^:main //mover el head un paso atrás en el tiempo y subir ese commit de la rama foo al Branch main en el repo remoto**  **Ej 2 si no existe la rama en el repo remoto puedes crearla: git push origin main:newBranch** |
|  | git push origin master~1:foo //tomar los cambios de Branch main un paso antes en el pasado, el commit C5 y moverlo al repositorio remoto debajo del Branch foo |
|  | git push origin foo:master //copiar todos los cambios del Branch foo al Branch remoto master |
|  |  |
|  | **2.6 Fetch arguments** |
|  | arámetros 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).** **El parámetro <lugar>** **Si especificas un lugar con git fetch como en el comando siguiente:**  **git fetch origin foo //descargar los commit del branch foo en el repo remoto, sobre el Branch o/foo en el repo local**  **Git va a ir a la rama foo en el remoto, va a traer todos los commits que no estén presentes localmente, y luego los aplicará sobre la rama o/foo localmente. n acción (refresquemos el concepto).**  **Por qué git aplicó esos commits sobre la rama origin/foo en lugar de aplicarlos sobre la rama foo local? Pensé que el parámetro <lugar> era un lugar que existía tanto local como remotamente"**  **Bueno, git hace una excepción especial en este caso, porque podrías tener trabajo en la rama foo que no quieras mezclar. Esto hace referencia a la lección anterior sobre git fetch - no actualiza tus ramas locales no-remotas, sólo descarga los commits (para que pueadas verlos o mergearlos después).**  **fetch y push son bastante similares, sólo que en direcciones opuestas.**  **Ej: git fetch origin foo~1:bar //descargar los cambios de foo un paso atrás en el pasado de los commit de foo del repo remoto hacia el Branch bar en el repo local**  **Ej 2 si el Branch no existe local: git fetch origin foo~1:bar //funciona igualito, solo que ahora git va a crear automáticamente el Branch bar en el repo local. Igual que como git push crea el branch en el repo remoto si no existe**  **Ej 3: git fetch // si git fetch no recibe argumentos simplemente descargara todos los commit de todas las ramas del repo remoto y creara las ramas si no existen**  **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!** |
|  | **git fetch origin master~1:foo //ir y descargar un paso atrás en commit de Branch main remoto, descargarlos en Branch foo local** |
|  | **git fetch origin foo:master //ir y descargar los commit de Branch foo remoto en Branch main local** |
|  | **git checkout foo //ir a branch foo** |
|  | **git merge master //luego de ir a Branch foo, hacer merge y copiar todo lo de Branch master en Branch foo** |
|  |  |
|  | **2.7 Source of nothing //cuando no pones nada en origen noorigen:destino** |
|  | **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** |
|  | **git push origin :foo //esto elimina el Branch foo de ambos repo local y remoto** |
|  | **git fetch origin :bar //esto crea la rama si no existe en el repo local** |
|  |  |
|  | **2.8 Pull arguments** |
|  | Parámetros de git pull **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.**  **Estos son algunos comandos equivalentes de git:**  **git pull origin foo equivale a:**  **git fetch origin foo; git merge o/foo**  **Y...**  **git pull origin bar~1:bugFix equivale a:**  **git fetch origin bar~1:bugFix; git merge bugFix**  **git pull es simplemente un atajo para un fetch + merge, y todo lo que le importa a git pull es dónde terminaron esos commits (el parámetro destino que determina durante el fetch).**  **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 comando** |
|  | git pull origin bar:foo //en un unico comando crear rama foo, descargar los commit de bar del repo remoto a esta nueva Branch foo en el repo local y después mezclamos esa rama remota a nuestra rama local main |
|  | git pull origin master:side //en un unico comando crear rama side en repo local, descargar los commit de la rama main del repo remoto a esta nueva Branch side en el repo local y después mezclamos esa rama remota a nuestra rama local main |





