Guía rápida de GIT

Para trabajo por línea de comandos

Esta guía asume que están trabajando con GIT classroom y con un workflow de trabajo muy básico (sólo lo necesario para los TP de la materia). La mayor parte de los comandos mencionados tienen múltiples agregados opcionales que pueden modificar el comportamiento que se menciona en esta guía.

Glosario

repositorio

O simplemente **repo**, es un lugar de almacenamiento de archivos (y directorios) gestionado. El repo puede ser local o remoto, según se encuentre en nuestra PC o en algún otro lugar (normalmente es “en la nube”).

commit

Es un “paquete” de cambios realizados (y agregados al **stage** mediante comandos add) que se van a gestionar como una unidad. Al generar un commit, se obtiene un ID del mismo (commitID), que luego puede usarse en otros comandos para referenciar este paquete de cambios. Al referenciar mediante un ID no hace falta usar todo el ID (40 caracteres). Con los primeros 8 caracteres en general alcanza para identificar unívocamente el commit, y git permite hacerlo de esa forma.

HEAD

¡Así, en mayúsculas! Es una referencia al último commit realizado[1], el que representa el estado actual de los cambios realizados y confirmados en el repositorio local. Puede usarse en cualquier lugar donde se espera el uso de un “commitID”. También se pueden referenciar <n> commits hacia atrás en el historial mediante HEAD~<n> (ejemplo: `HEAD~1` es el anteúltimo commit).

stage

También referido como **index** o **staging area**, refiere a los cambios realizados y preparados para ser agregados al próximo commit que se genere. Algo así como un “área de empaquetado”.

origin

Es el nombre por defecto que se le da al repo remoto que fue clonado para generar al repo local en el que trabajamos.

workspace

Es el “área de trabajo”, como su nombre lo indica, en la que se van registrando todos los cambios que se van haciendo sobre el código. Una vez conformes con los cambios, aquellos que querramos manejar como una unidad se deberán agregar al **stage**, para luego generar el commit.

[1] En realidad es más complejo que eso, pero de acuerdo a como lo vamos a usar nosotros nos vamos a quedar con esta definición.

Comandos

git clone <url>

Clona un repo remoto (que luego será referenciado como **origin**) indicado por <url> a un repo local, creando un nuevo subdirectorio para el mismo dentro del directorio actual. Requiere identificación para el repo remoto (por ejemplo, usuario y contraseña).
Se hace una vez por cada TP, al principio, con la url obtenida del repo remoto correspondiente al grupo y paradigma. Ejemplo:
`git clone https://github.com/pdep-sm/grupo4-tp-funcional`

git add <filename>

Agrega el archivo <filename> al **stage**. Si en lugar de <filename> usamos simplemente un punto (`git add .`), se agregan los cambios de todos los archivos del directorio.
*Cada vez que hagamos cambios al código, tenemos que agregar el archivo modificado al **stage** para poder luego armar el paquete de cambios. Ejemplos:*
`git add Grupo4TPFuncional.hs` (agrega el archivo `Grupo4TPFuncional.hs` al **stage**)
`git add .` (agrega todos los archivos modificados al **stage**)

git commit -m <message>

Genera un nuevo commit con el mensaje indicado en base a los archivos del **stage**. El mensaje sirve para identificar qué se está modificando. Además, si el mensaje menciona un determinado "issue" (Por ejemplo, "fix #5"), cuando este commit es enviado a **origin**, el issue es cerrado automáticamente.
Cuando terminamos de implementar una funcionalidad o arreglar un issue, generamos un commit. Ejemplos:
`git commit -m "Punto 3.2. Implementación de la función pepito"`
`git commit -m "Mejora de expresividad en la función pepito. Fix #5"`

git push

Es una sincronización saliente, desde el repo local hacia **origin**. Envía los commits generados localmente que no se hayan enviado anteriormente.
Si localmente sólo generamos tags y no hay nuevas modificaciones al código (no hay nuevos commits), entonces podemos usar el agregado "`--tags`" para enviar los mismos (`git push --tags`) a **origin**.
Si existen cambios previos en **origin** que no existan localmente y afectan a los mismos archivos que afectan los cambios que se están enviando, se genera un conflicto. Ver Resolución de conflictos más adelante.
Requiere identificación para el repo remoto.

git pull

Es una sincronización entrante, desde **origin** hacia el repo local. Obtiene los commits existentes en **origin** que no se encuentran en el repo local. Si existen cambios locales en los mismos archivos que incluyen los commits recibidos, se genera un conflicto. Ver Resolución de conflictos más adelante.
`git pull` es equivalente a hacer `git fetch` (traer desde **origin** hacia el repo local) y luego `git merge` (traer del repo local al **workspace**), pero nosotros podemos manejarnos con `git pull` directamente, que trae desde **origin** hacia el repo local y el **workspace**. Requiere identificación para el repo remoto.

git status

Muestra los archivos del **stage** que tienen diferencias en comparación al **HEAD**, los que tienen diferencias con el **workspace** (es decir, están en el **stage** pero se les hicieron cambios sin add), y los archivos que no están en el stage.

git diff

Muestra las diferencias dentro de los archivos entre **stage** y el **workspace**. Si se agrega "`-- <filename>`" (`git diff -- <filename>`) se acota la operación a ese archivo. Ejemplo:
`git diff -- musico.wlk`

git log

Muestra el historial de commits en orden cronológico inverso con ID, autor, fecha y mensaje del commit. Si se agrega `-p` (quedando `git log -p`) muestra además los cambios de cada commit.

git reset -- <filename>

Es la operación opuesta a `git add <filename>`, eliminando el archivo del **stage** para que NO forme parte del próximo commit que se genere.
*Lo vamos a usar cuando detectemos que un archivo que ya agregamos al **stage** en realidad no debería incluirse en el paquete de cambios, ya sea que es incorrecto o porque queremos trabajarlo por separado. Ejemplo:*
`git reset -- Grupo4TPFuncional.hs`

git checkout <commitID> -- <filename>

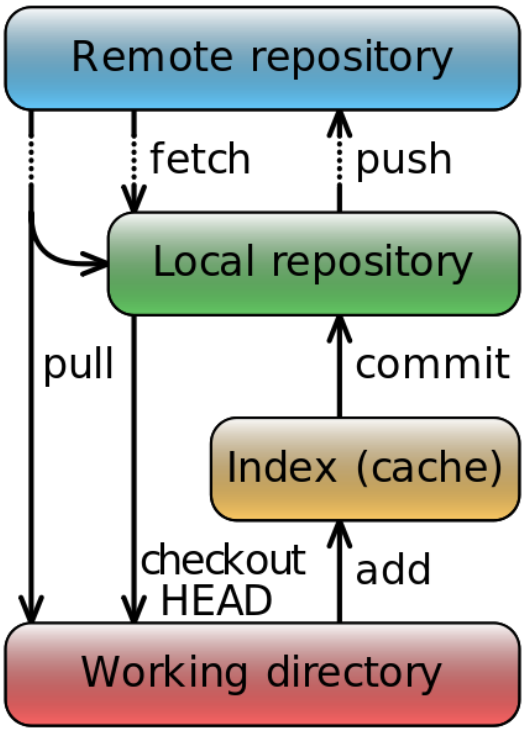
Sirve para sobreescribir la versión del **workspace** de <filename> con la versión correspondiente al <commitID>. Como <commitID> se puede usar, por ejemplo, **HEAD**, o un id específico obtenido en un commit.
Ejemplo, para descartar los cambios hechos a un archivo desde el último commit:
`git checkout HEAD -- Grupo4TPFuncional.hs`

git tag <etiqueta> <commitID>

Permite marcar con una etiqueta a un determinado commit.
Cuando una entrega de TP está lista, le vamos a poner una etiqueta indicando la versión (v1.0), y cada re-entrega que se haga va aumentar el subnúmero de versión (v1.1, v1.2, etc.). Ejemplos:
`git tag v1.0 HEAD`
`git tag v1.1 g8d9bc`

Flujo habitual de trabajo

1. Tener actualizado el **workspace**:
   a. *git clone <url>* si aún no cloné el repo (**sólo una vez**, al principio).
   b. *git pull* si ya lo tenía clonado, para traerme los cambios que hayan subido otros compañeros de grupo (**siempre**, antes de empezar a programar los cambios para un nuevo commit y antes de hacer un push).
2. Escribir código (por ejemplo, resolver algún/os punto/s de algún TP).
3. *git add .* para agregar los cambios al **stage**.
4. (Cuando terminé de programar y probar los cambios) *git status* para verificar si me quedó algo fuera del **stage**.
5. *git commit -m <mensaje>* para empaquetar los cambios.
6. *git push* para llevar los cambios al repo remoto.



Resolución de conflictos

Un **conflicto** se da cuando una misma porción de código dentro de un archivo es modificada por dos commits "no secuenciados" o "paralelos" (en general, porque uno es local y el otro es remoto), y git no sabe qué versión es la correcta. Cuando hago el pull para obtener los commits remotos, en el archivo modificado voy a ver marcado el conflicto de la siguiente forma:

```
<<<<<<< HEAD
suma a b = a + b
=======
suma a b c = a + b + c
>>>>>>> Agrego un tercer sumando
```

Del ejemplo, se puede deducir:

1. La función `suma a b` (entre `<<<<<<< HEAD` y `=======`) es la que actualmente está en el repo remoto.
2. La función `suma a b c` (entre `=======` y `>>>>>>> Agrego ...`) es la que yo quiero empaquetar en un commit.

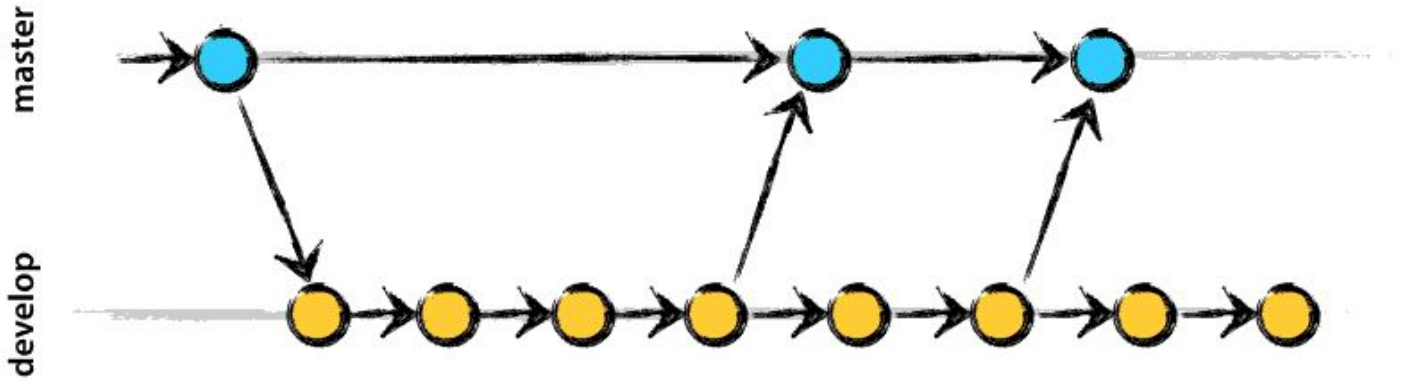
Para resolver el conflicto, la corrección se realiza en forma local y luego se envía la versión arreglada a **origin**. Se debe:

1. Elegir una versión (la de **HEAD** o la mía), o bien armar una nueva.
2. Borrar los caracteres marcadores que agregó git (es decir, `<<<<<<< HEAD`, `=======` y `>>>>>>>`)
3. Agregar los nuevos cambios al **stage** con *git add <filename>*
4. Hacer el *git commit -m "Arreglo la función suma"*
5. Hacer el *git push*

Guía rápida extendida de GIT

Agregados para trabajo con “branches”

El sistema de “branches” o ramas nos permite desarrollar en paralelo diferentes funcionalidades (cada una en un branch distinto) sin mezclar el código hasta tener una versión final y estable. Luego, cuando se pretenden integrar los cambios de cada branch para obtener un producto más completo, se fusionan mediante un merge.



Cuando se trabaja de esta forma, se tienen múltiples versiones del desarrollo tanto en forma local como en forma remota. Es decir, pueden existir distintos branches tanto localmente como en **origin**.

Glosario

branch

Cada branch es una referencia a un commit y, por lo tanto, puede usarse en cualquier lugar donde se requiera un **commitID**, como pasa con **HEAD**. Cada vez que generamos un nuevo commit cuando estamos trabajando sobre un determinado branch, este se mueve automáticamente para referenciarlo, como también pasa con **HEAD**. La diferencia es que, al cambiar el branch en el que estoy trabajando, cambia la referencia de **HEAD** para apuntar al nuevo branch, pero la del branch anterior no cambia, con lo cual pueden mantenerse historiales de cambios paralelos mediante el uso de múltiples branches.

master

Es el nombre por defecto que se le da al branch principal, aquel en el que se tiene una versión estable y funcional del desarrollo. Al trabajar con múltiples branches, los cambios se trabajan sobre algún otro branch y luego se incorporan a master, una vez testeados y aprobados, mediante un merge.

merge

Es la acción de unificar los cambios referenciados por distintos commitID. En el gráfico de referencia de branches puede verse cómo se incorporan 2 veces los cambios del branch develop al branch master, primero desde el cuarto commit (y todos los cambios de sus 3 antecesores) y luego el sexto commit (y los cambios de su único antecesor que no había sido incorporado antes).

pull request[2]

Es un paso “online”, ya que no es propio de la herramienta Git sino de la implementación de los repositorios online con los que se trabaja. Consiste en un pedido (request) para que alguien realice un pull de los cambios realizados en los commits de un determinado branch (online) hacia otro branch (también online). Luego de solicitado, pueden abrirse discusiones sobre los cambios que se introducen al branch destino e incorporar nuevos cambios al branch de origen. Cada commits agregados sobre el branch (siempre online) de origen se incorpora al pull request automáticamente hasta que el mismo sea aprobado, momento en el cual todos los commits del branch de origen se incorporan al de destino, y se cierra el pull request.
Es el método habitual con el que se incorporan cambios a un desarrollo, y es el que vamos a usar para hacer las entregas (y correcciones). Ver explicación del flujo de trabajo más adelante.

Comandos

git checkout <nombreBranch>

Permite movernos entre los diferentes branches de nuestro proyecto para poder trabajar en cambios que tienen como base el commit al que hace referencia dicho branch.
Con el agregado `-b` (quedando `git checkout -b <nombreBranch>`) se crea un nuevo branch con el nombre indicado antes de moverse al mismo.
Ejemplo para crear el branch dev y comenzar a trabajar en el mismo:

```
master> git checkout -b dev
dev>
```

git branch --list

Lista los branches existentes e indica sobre cuál estamos trabajando actualmente.

git merge <commitID>

Incorpora los cambios realizados en el commitID (generalmente, una referencia a un branch, y así se lo considera a continuación) y sus antecesores sobre el branch en el que estamos trabajando actualmente.
`git merge` puede generar conflictos de versiones, como pasa cuando hacemos `git pull`. De hecho, hacer `git pull` es hacer `git fetch` + `git merge`, y es por el paso de `merge` que el comando `pull` nos muestra los conflictos. La forma de resolverlos es la misma que ya se mostró en la correspondiente sección. Hay que considerar que ya no hay un repo remoto u **origin**, y por eso no requerimos de la parte de `fetch` (que es ir a buscar los nuevos commits a **origin**) que hace el pull, sino que es otro branch del mismo repo local.
Ejemplo, para incorporar los commits del branch `correccionesEntrega1` al branch de trabajo actual, en este caso `dev` (mostrado como prompt de la consola, en azul):

```
dev> git merge correccionesEntrega1
```

Tener en cuenta que, como se mencionó anteriormente, este comando puede generar conflictos de versiones.

[2] **pull request** es el nombre que se usa en el sitio GitHub, que en general es el más conocido. Otros sitios pueden darle otro nombre a esto mismo, por ejemplo el caso de GitLab que lo llama **merge request**.

Flujos de trabajo[3]

Al estar trabajando con múltiples branches, debemos considerar el contexto en el que estamos para describir los pasos. Lo más importante que tenemos que considerar es no hacer merge en master, ya que esto se va a usar para la entrega del TP. Debemos trabajar en branches secundarios, y hacer cada entrega por medio de un pull request.

Caso 1

Para la primera entrega, podemos trabajar desarrollando sobre un branch `entrega1` de la misma forma que lo hacíamos anteriormente. Al momento de realizar la misma, haremos un pull request sobre master. El docente va a revisar la implementación y, posiblemente, pedir correcciones o mejoras. Vamos a incorporar los commits con las correcciones/mejoras sobre el mismo branch `entrega1`.

1. *entrega1> git push* para enviar nuestra versión local a **origin**.
2. Ir al sitio GitHub y realizar el pull request a `master`.
3. El docente va a revisar lo entregado y marcar issues a corregir. Esto genera una "revisión" en GitHub con los puntos observados.
4. Implementamos las correcciones solicitadas en la revisión en nuestro código.
5. *entrega1> git add .* para agregar nuestros cambios al **stage**.
6. *entrega1> git commit -m "Delego el cálculo de energía de Pepita"* para empaquetar nuestros cambios.
7. *entrega1> git push* para enviar las correcciones realizadas a **origin** nuevamente e incorporarlas al pull request.

Caso 2

Supongamos que estamos trabajando en el branch "`correccionesEntrega1`" y vemos que alguien realizó cambios sobre el branch "`dev`" (remoto), posiblemente agregando otras correcciones. Siempre nos conviene estar actualizados con el branch principal de desarrollo ("`dev`" en nuestro caso) ya que, de lo contrario, podríamos estar modificando código que ya no existe, o cambió su implementación. Los pasos a seguir serían:

1. Subimos nuestros cambios a la versión remota de nuestro branch (con los comandos `add`, `commit` y `push`).
2. Hacemos un `pull` del branch `dev` para obtener los cambios que se hicieron en forma remota en el mismo e incorporarlos a nuestra versión local de `dev`.
3. Parados en nuestro branch `correccionesEntrega1`, que es el que va a integrar los cambios propios y ajenos, hacer el correspondiente `merge` (mediante `git merge dev`).

Nota

En muchos casos, en los workflows algo más avanzados se mantiene una versión productiva en el branch `master` (que en nuestro caso, va a ser la versión ***aprobada***), una versión en desarrollo en un branch `dev`, y cada corrección o incorporación de una nueva funcionalidad se hace en un tercer branch. Por ejemplo, si mi cambio consiste en cambiar mi modelo para que pase de utilizar un auto a múltiples autos distintos, mi branch puede llamarse `multiplesAutos`. Cuando puedo confirmar que mis cambios funcionan (en general, mediante tests automáticos), entonces puedo incorporar mis cambios a `dev` mediante un `merge`. Una vez incorporados todos los cambios necesarios en `dev`, podemos hacer el pull request a `master` para que el código sea revisado por alguien más antes de ser incorporado a la versión productiva. En caso de ser requeridos, se

[3] Existen muchos workflows posibles. Acá se presenta una opción, pero se puede variar.

pueden incorporar más cambios al mismo pull request mediante el agregado de commits al branch desde el que se realizó el mismo, que en nuestro ejemplo es `dev`.