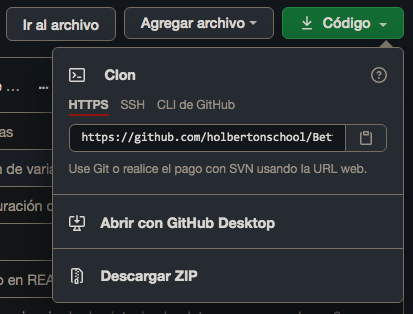
***Introducción a repositorios remotos (GITHUB)***

Resulta que si, por ejemplo, usted no se encuentra solo en un proyecto y necesita que varias personas, no solo usted, estén manipulando constantemente el proyecto en cuestión (haciendo avances y modificaciones sobre él, creando nuevas versiones del mismo, etc) y desea, al igual, contar con un sistema de versiones que los permite controlar cada copia de seguridad que se haga sobre el proyecto: es aquí donde se ve necesario el uso de ***“repositorios remotos”.*** Es así porque es el repositorio remoto el que nos da la facultad, a mí y a mi equipo de trabajo, de poder contar con un servidor unificado; es decir, un servidor donde estarían las versiones unificadas del trabajo de multiples desarrolladores (incluyendome).

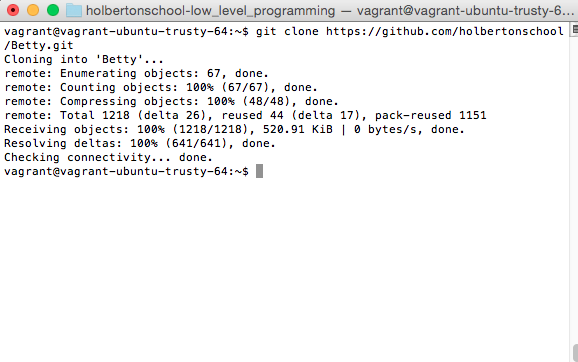
Con un repositorio local, solamente, no se podría trabajar en equipo sobre el mismo proyecto estando cada desarrollador en su propio entorno de trabajo o *setup.* He aquí donde surge la importancia de un servidor remoto (o siendo más precisos, un repositorio remoto): es, potencialmente, el punto de encuentro de todos los repositorios locales (de cada desarrollador) que quieran participar en un repositorio que los unifique sobre un mismo proyecto. Entonces, cada quién trabaja en local y, cuando terminan, lo mandan todo a esa nube en común que se llama repositorio remoto, donde todos pueden revisar el trabajo de cada uno. El repositorio de mayor aceptación laboral y el más popular a la vez es: **github.**

Ahora, si por ejemplo, hemos decidido primero crear un repositorio remoto (antes que un local). Si quisieramos importar ese repositorio remoto a nuestro potencial repositorio local, se hace através de un comando: **“git clone”.**

**git clone:** tiene la función de importar un repositorio remoto de *github* (u otro servidor externo), que aún no está vinculado a repositorio local (por lo general esto se hace cuando queremos vincular el repositorio remoto -o en nube- de un tercero... *para poder trabajar conjuntamente en un proyecto, tal cual*). Si el repositorio no nos pertenece, para importarlo debe hacerlo de esta forma (tomaremos un ejemplo):



Nos dirigimos a esta parte del repositorio remoto que queramos importar (este ejemplo se hace con GitHub) y copiamos link. Luego, para vincularlo a nuestro repositorio local, sería así:



Al llamar este *link* lo que hace “git clone”, en su maquina, es llamar completamente a la dirección del repositorio remoto en nuestro repositorio local; lo que quiere decir que, se trae sus archivos y lo hace de tal manera que: se trae una copia del “master” (del repositorio remoto) a nuestro directorio de trabajo (la carpeta que se le inicializó con git y que está ejecutando “git clone”) y, adicionalmente, crea una base de datos de todos los cambios historicos (copias de seguridad o “commits”) del repositorio remoto en nuestro repositorio local.

Del resto, seguiría trabajando igual cada vez que quiera exportar copias de seguridad, de su repositorio local, al repositorio remoto clonado. Cómo se exportan nuestros “commits” del repositorio local a un repositorio remoto vinculado? Con el comando: **“git push”.** *Git push* se encargaría de exportar nuestro “head” o última versión del proyecto guardada en copia de seguridad.

Ahora, si deseamos *importar* copias de seguridad del repositorio remoto a nuestro repositorio local (luego de ya haber clonado, ojo, al repositorio en sí. Importante esto), lo podemos hacer con una variedad de comandos de git, entre esos: **“git fetch”**, **“git merge”** y, el más poderoso, **“git pull”**. Vamos uno a uno.

***Introducción a Branches (o ramas)***

La rama, por predeterminación, al inicializar git en una carpera se llama: “Master”. En la rama *Master* se guardan inicialmente todas nuestras copias de seguridad. Sin embargo, tenemos la opción de crear otros *espacios de memoria* que se encarguen de guardar también otras copias de seguridad del mismo directorio de trabajo (el inicializado en git). Entonces, en resumidas cuentas, las ramas son formas en la que nosotros podemos hacer cambios (o versiones o ediciones en copias de seguridad) sin afectar el espacio de memoria, en absoluto, de la rama principal *master.* Qué utilidad podría tener esto?

Anteriormente habíamos expuesto un par de ejemplos, si recuerda? Habiamos dicho que, *“con* ***Git Branch*** *es posible crear diferentes ramas de desarrollo que pueden converger en el mismo repositorio. Podríamos hablar de: una rama estable, una rama experimental* ***“development”*** *(por ejemplo, para probar nuevo código, fuera de lo convencional que te pide o te exige el proyecto, que podría ayudarte a encontrar “por accidente” algo mejor que lo normal o para probar una posible solución en sí a un problema planteado del que no encuentras todavía una salida concreta)* y una rama *inestable*, en las industrias a un tipo de ellas se les conoce como:***“bugfixing”*** *o* ***“hotfix”,*** *que sirven para arreglar un bug de la versión actual de nuestra rama principal, sin estropear en absoluto todo el trabajo que ya se lleva hecho en ella (para cuidar el trabajo que ya tenemos hecho en código precisamente); entonces, todos los errores a los que nos expongamos, posiblemente, en el intento de resolución de nuestro bug... que se queden y se den, con toda tranquilidad, en la rama experimental. Cuando ya por fin hayamos podido encontrar, en dicha rama experimental, la resolución de problema a nuestro bug, luego de desordenarlo todo sin mosquearnos por eso (no hay nada de qué preocuparse, para eso se creo esta rama); bueno, ahí sí podrá unir la rama experimental, su última versión, a la versión última de la rama master. Ya entre poco dejaremos claro cuál es el comando qué se encarga de unir las ramas.*

También habíamos dicho que,... *“si está trabajando en equipo, cada integrante del mismo puede trabajar su parte en ramas diferentes; y, al final, cuando cada integrante ya tenga la edición definitiva de su parte (de cada sub-rama derivada de la rama master); entonces, ahí puede unir todas las sub-ramas a la rama principal master, logrando así que todos los avances definitivos de cada compañero de trabajo estén unidos a un proyecto final. En general, esta unión entre los avances de las sub-ramas y la rama master se hace por medio de un comando que se llama:* ***“git******merge”.”***

Pero bien, profundicemos estas ideas sueltas ahora. Cuando uno crea una rama, con el comando “**git branch + nombre de nueva rama”**; digamos, la primera rama (una rama hecha a partir de la rama *master*) lo que sucede es que creamos una copia exacta del último commit (*head)* de la rama *master,* pero en otro lado (en otro espacio de memoria que no interfiere en absoluto sobre el espacio de memoria de la rama *master* ni la altera); es decir, todos los cambios que hagamos en la nueva rama no se verán evidenciados en la rama *master* hasta que no las *fucionemos* con la rama *master,* esa fusión se hace con el comando **“git merge + nombre de rama a importar”** (suponiendo que nos hemos desplazado de nuevo a la rama *master* luego de terminar con la nueva rama), lo que se importa es su última copia de seguridad, cambio guardado o “commit” (o en últimas su commit actual de trabajo, donde lo dejó): ojo con esto último; si hace un cambio y luego desea exportarlo, no se le olvide hacerle un “commit” a ese cambio que desea exportar (o que importará la rama *master* posteriormente).

Sin embargo, uno desea importar algo de la nueva rama cuando tenemos en esa nueva rama un contenido ya trabajado en ella que no tenemos en nuestra rama principal *master,* por ejemplo*.* Para que tengamos “contenido” en esa nueva rama creada, primero tenemos que desplazarnos hacia ella. Para desplazarnos hacia una rama se hace por medio del comando **“git checkout + nombre de rama destino”**.

Ahora, si queremos saber la totalidad de las ramas existentes que tenemos en nuestro directorio inicializado en git, sólo ejecutamos el comando **“git branch”.**

***Profundizaciones sobre “git merge”.***

Idealmente, el comando **“git merge”** se debe ejecutar en la rama principal; de lo contrario, se expone a perder todo el avance inicial que había hecho, obviamente en su rama principal, de su proyecto (pero igual también puede intentarlo, no está escrito en ninguna parte que no deba hacerlo de la forma contraria). Dicho lo anterior, si por ejemplo deseo importar los avances, ya guardados y registrados en su último “commit” o copia de seguridad, de la rama derivada o creada a partir de la rama *master;* debo primero situarme sobre la rama *master;* tal que así: **“git checkout master”**; y, posteriormente, llamar los últimos cambios en los que se avanzó (y fueron almacenados) en la sub-rama y que no tengo en rama *master* aún*.* Para llamar a ese último “commit”, de la nueva rama creada, en la rama *master* y fusionar el contenido de ambas; de tal manera que, ahora sean “dos proyectos” en uno (mezclarlos entre sí mientras no existan conflictos entre sus líneas de códigos),debe ejecutar el siguiente comando: **“git merge + nombre de rama a importar”.**

Ahora, ya fusionados, si ejecutamos el comando “git log” en la rama *master* nos daremos cuenta que el comando “merge” nos incorporó un “commit adicional” en el listado de “commits” originales de la rama *master:* ese “commit adicional” es realmente el último “commit” almacenado (head), de la nueva rama o rama secundaria, que ha sido importado en rama *master*. Lo que mezcla “merge” es el último “commit” de la rama secundaria con el contenido, del “commit” actual de trabajo, de la rama *master*.

Ahora bien, cualquier conflicto que haya entre ambas ramas al ser fucionadas, la rama *master* eventualmente y, de momento, se interpondrá sobre las líneas de código de la rama secundaria (sin generar problema alguno... parcialmente). Es, digamos, una manera “recursiva” de trabajar de Git.

***Profundizaciones sobre “conflictos con git merge”.***

A veces resulta que se está trabajando en un proyecto conjuntamente con otros desarrolladores y, ya sea por accidente o por cualquier otro motivo, dos programadores hacen sus propias modificaciones sobre un archivo y, da la casualidad, que esas modificaciones personalizadas las hacen sobre las mismas líneas de código del mismo archivo en cuestión. Potencialmente lo que podría generar esto, al intentar hacer un *merge* entre la rama del programador 1 y la rama del programador 2 (los dos programadores “conflictivos”), es un conflicto. Cómo podríamos solucionarlo?

Antes de tocar terreno, tengamos en cuenta algo: los conflictos no se generán si estás haciendo una adicción de una rama hacía otra (al ser exportada y pasarle un contenido, en términos de líneas de código, que aún no tiene la rama que está haciendo la importación); se generan es cuando, en vez de evidenciarse una adicción, ambas tienen ya contenidos diferentes en exactamente las mismas líneas de código de sus últimas copias de seguridad o “commits”. Dicho lo anterior, prosigamos...

Supongamos que hacemos el *merge* situados en *master;* es decir, para traernos los últimos cambios guardados de la “sub-rama” y que pretenden sobre-escribir sobre el contenido que ya tiene, en esas mismas líneas del mismo archivo, la rama *master* y su último “commit”. Lo que pasará es que Git nos notificará sobre el conflicto y nos exigirá solucionarlo (además de que, parcialmente, no se podría importar realmente el “commit” de la otra rama con la que habíamos decidido fusionarnos).

Si usted trabaja con un estorno de desarrollo avanzado (tipo Visual Studio Code), la IDE le preguntará automaticamente con qué cambios quedarnos ... a un click para decidirlo!

**Sin embargo, si no cuenta con este tipo de ayudas sofisticadas, usted manualmente debe decidir con qué contenido, de las líneas de código en conflicto, quedarse: si el contenido de la rama *master* o con el contenido de la sub-rama.** Debe manualmente borrar el código que no le interesa, ya sea el de la rama *master* o el de la sub-rama; e intentar de nuevo, luego de eso, ejecutar el comando “git merge” desde su rama *master* idealmente. (pero antes no se le olvide hacer los respectivos pasos de seguimiento y generación de “commit”).

Ahora bien, hay que interpretar correctamente el código de notificación de conflicto que hace GIT. Si la rama que ejecuta el comando “git merge”; es decir, la rama que está tratando de importar y fusionar, su último “commit”, con la última copia de seguridad de la otra rama, está presentando efectivamente conflictos con esa otra rama...

Git lo notificará tal que así: **<<<<<<< HEAD (Current change)**; es decir, Git se está refiriendo a la rama que está ejecutando el *merge* como la rama, *menor que \* 7*, HEAD (tendría sentido que sea *head;* pues, estamos situados actualmente en esa rama); y, por ejemplo, en un entorno de desarrollo como VSC, se le preguntará explicitamente al usuario, a un sólo click, si desea quedarse con los cambios registrados en la rama que está tratando de hacer la importación (en nuestro caso la rama *master*) y lo hace de la siguiente manera: ***“Accept current change”,*** que traduce literalmente como: *“aceptas los cambios actuales, ~~de la rama actual~~”*...y, si así lo desea, debe presionar sobre ese texto.

Ahora, en el caso contrario, Git para referirse a la rama que está siendo importada por otra, la pone en evidencia de la siguiente forma: **>>>>>>> “nombre de la rama importada” (Incoming change)**; luego, en un entorno de desarrollo como VSC, se le preguntará también a un sólo click si si desea quedarse con los cambios registrados en la rama que está tratando de ser importada (en este caso, la sub-rama) y lo hace de la siguiente manera: ***“Accept incoming change”***, que traduce literalmente como: *“aceptas los cambios entrantes, ~~de la rama que está siendo importada~~”*... y, si así lo desea, debe presionar sobre ese texto.

*VSC lo que intenta es ahorrarte el proceso de eliminar manualmente las líneas de código con las que no quieres quedarte, ya sea de una rama o de otra, para borrarlas de forma automatica y a un sólo click* ***(no se te olvide guardar el archivo luego de haber decidido con qué líneas de código quedarte)****.*

Ahora bien, como esto no deja de ser un proceso donde estás registrando “nuevos cambios” al decantarte por las líneas de códigos de una rama y suprimiendo las otras (de la otra rama), tienes que repetir entonces el mismo proceso de: ***hacer seguimiento (con git add) y generarles una copia de seguridad (con git commit -m “solucionado el conflicto de las ramas, etc...”)*** a ese archivo en cuestión.