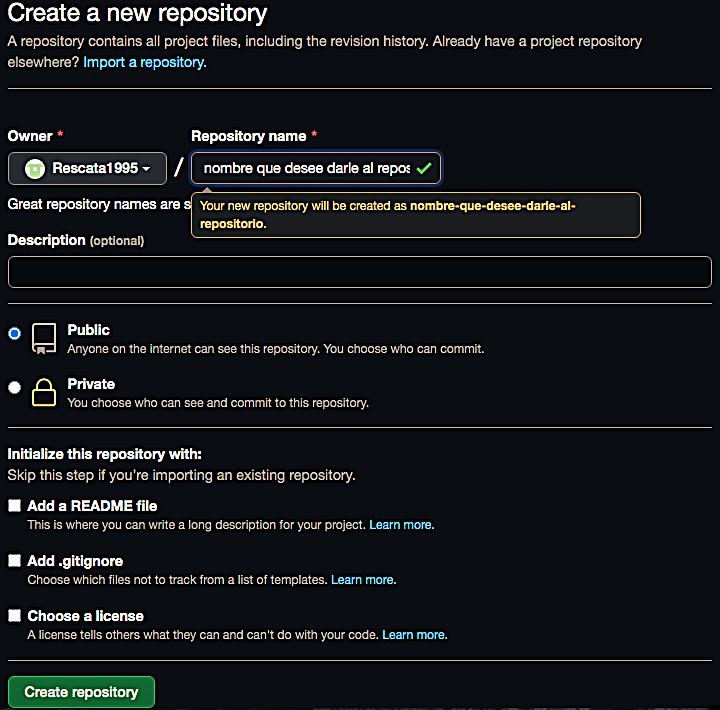
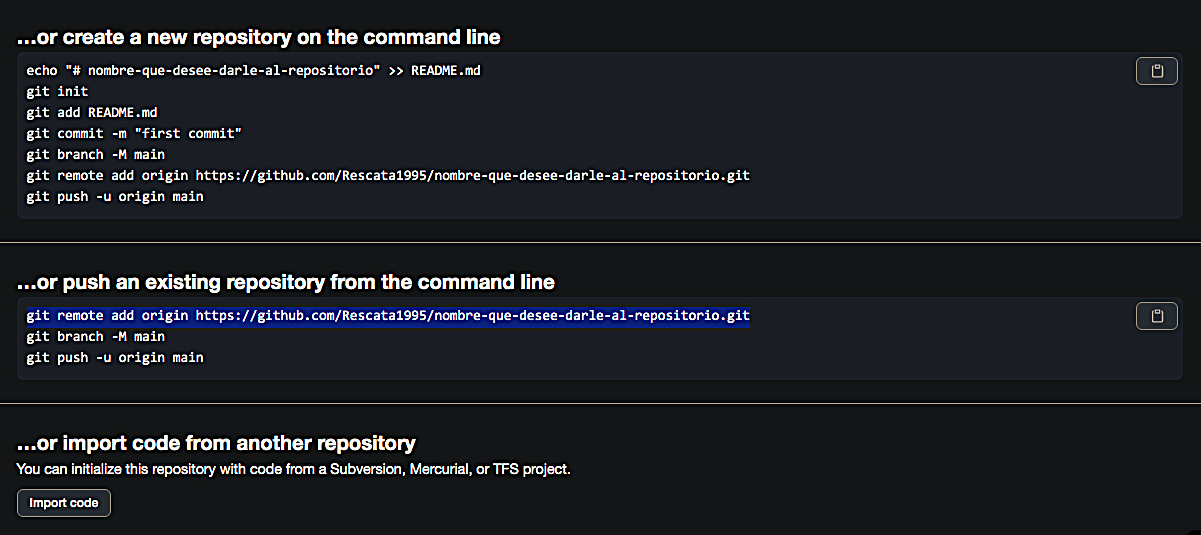
***Tutorial de GITHUB***

Todos los repositorios que hemos creado en GIT ahora nos interesa guardarlos en una nube o servidor externo (GITHUB ocupa GIT en la nube, eso hace). Si en caso dado se nos quema nuestro disco duro ya evitaremos completamente el riesgo de perder absolutamente todo o me encuentro trabajando en un proyecto grupal en el que necesito cooperar con más programadores; pues, todos nuestros repositorios estarán en la nube, el cual podemos retornar nuevamente a nuestra pc en cualquier momento. Esa interacción entre GIT & GITHUB es la que nos interesa. Pasos desde la creación de cuenta en GITHUB hasta la creación de un repositorio remoto vinculado a nuestro editor de texto y terminal:



Le damos un nombre al repositorio remoto que deseemos crear y decidimos si será de caracter público o privado y listo: create repository!

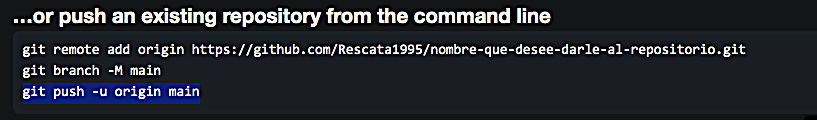


Luego, seleccionamos toda la ruta subrayada con fondo azul, la pegamos en nuestra terminal y presionamos “enter”, tal que así (el comando ***“git remote add origin”***... es el que tiene la función de vincular tu repositorio local con un repositorio remoto; de tal manera que, posteriormente, puedas exportar tus cambios guardados de un repositorio local en un repositorio remoto o, en su defecto, puedas importar los cambios guardados de un repositorio remoto en un repositorio local”):



Esto le dirá a GIT que ya estaremos trabajando de forma remota. Es decir, va a crear un identificador de todos nuestros archivos o de todo el trabajo que hemos hecho con GIT en la URL que hemos importado de GITHUB (nuestro repositorio local ahora estará en una nube).

Entonces, al terminar de registrar o guardar todos los commits correspondientes en nuestro repositorio local; es decir, cuando terminemos la tarea, importaremos la siguiente ruta (subrayada en fondo color azul) a nuestra terminal (y presionamos “enter”):



Esto cargará de forma automatica todos nuestros repositorios locales a GITHUB; de tal manera que, no sólo tengamos repositorios locales, sino que también en la nube (libre de riesgos de pérdida): sólo actualice su página en GITHUB. ***Importante:*** **la rama principal de GitHub también se llama *main;* sólo que, para distinguir entre la rama *main* del repositorio local y la rama *main* del repositorio remoto: a la rama *main de* GitHub se le identifica por su primer nombre como “*origin”, siendo entonces su nombre completo: “origin main”.***

Ahora bien, con esto ya tenemos vinculado al GIT local con el GIT remoto; es decir, no es necesario llamar de nuevo las anteriores rutas subrayadas si quisieramos guardar un nuevo commit en nuestro repositorio local y cargarlo en la nube de GITHUB. ¿Entonces qué se hace? pues se ejecuta el siguiente comando: “git push”. Listo, eso es todo. Así hemos cargado en la nube GITHUB nuestro último commit guardado en nuestro repositorio local. En el caso contrario, es decir, que quisieramos importar un “commit” del servidor externo de GITHUB (ya sea porque lo hayamos creado allá y no en nuestro repositorio local); pues, el comando a ejecutar sería: “git pull”. Por último, si deseamos compaginar por igual los “commits” que tenemos en ambos repositorios, local y remoto, debemos ejecutar el siguiente comando: “git fetch”. (la explicación de este último comando puede mejorar).

***Introducción a llaves SSH***

Aunque, en principio, github sea una plataforma o sitio web seguro (*HTTPS: Protocolo seguro de transferencia de hipertexto*) existe cierta vulnerabilidad a la que estamos expuestos: al introducir tu nombre de usuario y contraseña en la página web, éstas se guardan también en nuestro entorno local (digamos, en nuestro computador). Entonces, en el eventual caso de que sufras de un robo de tu dispositivo, estarás completamente expuesto a sufrir lo que se conoce como: ***“Password Cracking”***, estás expuesto a que “crackeen” (descifren) la contraseña de tu repositorio. “Password Cracking” es la ciencia que consiste en descifrar contraseñas o también se refiere al acto o proceso que se lleva a cabo para recuperar contraseñas que se han sido almacenadas en un equipo (cualquiera, desde dispositivos moviles a dispositivos de escritorio). En nuestro caso entonces estamos planteando la posible idea aterradora de que nuestro nombre de usuario y contraseñas de Github se vean expuestas a ser descifradas; seríamos vulnerables a ataques y quedaría en riesgo nuestra información sensible: servidor, proyectos (ya sean propios o de clientes), etc. Esta es la forma en la que sitios web, por ejemplo, son hackeados.

Para evitar todo esto, tenemos que agregarle una capa de seguridad mayor a nuestros sitios web de interés; donde, a veces, colocamos información confidencial o muy sensible y que deseamos proteger: esto se hace por medio de una combinación entre llaves públicas y privadas ***(llaves SSH)***, a este proceso (de cifrar por medio de llaves, especialmente una pública y una privada) más tarde se dará cuenta que se le conoce también como: ***cifrado o criptografía asimétrica.***

***El mecanismo funciona más o menos así:*** en nuestro entorno local (computadora) se debe crear una llave pública y una llave privada; y, una vez creadas, la llave pública debe ser enviada a github (en nuestro caso, nuestro sitio web de interés); pero puntualmente la llave pública es asignada a un repositorio remoto propio de github (al repositorio de nuestro interés *-se supone, el que se quiere cifrar*): todo este proceso se lleva a cabo conectandonos al repositorio por medio de un protocolo nuevo, ya no *HTTPS*; sino, ***SSH (Secure Shell)***. La llave privada sigue intacta en nuestra computadora, no es enviada nunca a ninguna parte.

dato: SSH es el mismo protocolo que se debe usar para conectarnos a servidores remotos, a otras computadoras que queramos controlar remotamente por consola.

Github, a partir de la llave pública que está correlacionada con tu llave privada y que le acabas de enviar; te regresa el favor y te envía, cifrado con tu propia llave pública (la que le mandaste), su propia llave pública de github (es un cifrado hecho, recuerde, a partir de la llave pública que te recibe github; y github, a partir de ahí, con ese cifrado, te genera una llave pública propia de él para ti). Entonces, github le manda su propia llave pública a su entorno local y la conecta: y así ambos quedan, mutuamente, conectados por medio del protocolo SSH, en lo que se conoce como una ***conexión de doble camino***, la cuál es 100% cifrada por SSH.

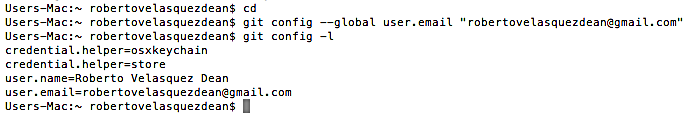
Por otro lado, la llave privada de su entorno local y que nunca es compartida, si desea, le agrega una contraseña encima (sobre ella): esto para tener una llave privada mucho más robusta y poderosa.

***Ahora sí, manos a la obra, veamos paso a paso todos estos procesos en un repositorio real de github.***

Aclaremos algo de entrada, *las llaves SSH no son por repositorio o por proyecto, sino por persona*; es decir, puedes crearlas en cualquier parte donde te situes parcialmente dentro de tu consola. Idealmente se crean en la carpeta home *(cd, cd ~)*.

Posteriormente, verificamos que el correo registrado en nuestra cuenta *github* coincida con el configurado (o potencialmente a configurar) en nuestro entorno local, puntualmente en *git.* El comando de git para registrarle un correo es el siguiente: ***git config --global user.email “inserte correo aquí -que debe coincidir con el registrado en su cuenta de github, recuerde”.***

Para verificar que efectivamente hemos registrado con exito el correo insertado, podemos corroborar esta información con el siguiente comando: ***“git config -l”***, donde podemos revisar también cuál es el nombre propiamente de usuario registrado en git. Hasta ahora el registro de comandos ejecutados dentro de su terminal se debe ver algo más o menos así:



Hecho esto, estando en sintonia nuestro correo electronico registrado en git y, posteriormente, nuestro correo electronico registrado en github; **Ya podemos crear nuestras llaves SSH y configurarlas en nuestro entorno local primeramente antes que conectarnos a github con SSH**

**(no olvide estar situado en *home*).**

**1.** Corra el siguiente comando en su terminal: ***ssh-keygen -t rsa -b 4096 -C “robertovelasquezdean@gmail.com”***

*Vamos por partes:*

***rsa*** es un algoritmo más, entre otros, para crear la llave ssh (siendo el más popular de hecho)

***-b + número*** es un elemento que nos indica, a partir de un número, qué tan compleja queremos que sea la llave.

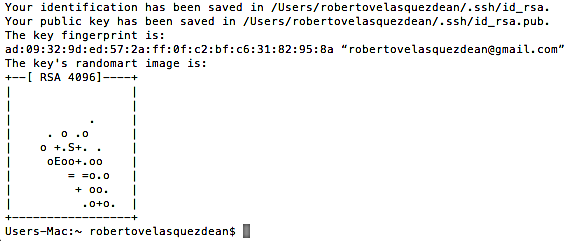
***-C “email...”*** se refiere a definir *qué correo electronico queremos que esté conectado a la llave en cuestión.* Debe intuir que es el mismo correo que hemos estado configurando, y en sintonia, en git y github.

Inmediatamente luego de correr la línea de comando, con el comando para crear llave SSH, se nos mostrará esto:



*Este proceso ya me está generando la llave pública y privada a partir del comando ejecutado.* Posteriormente nos pregunta dónde deberiamos guardar las llaves (y he aquí la importancia de situarnos inicialmente desde el *home*), si usted desea sólo presiona *enter,* ya la consola sobreentiende que usted desea guardar sus llaves en su dirección actual. Es decir, en nuestro caso, las llaves serán guardadas en *home* y, más puntualmente, en una carpeta oculta llamada: ***“.ssh”***

Luego de presionar *enter* la consola pide un ***“passphrase”,*** *passphrase* en realidad es una manera *fancy* de escribir *“passwords con espacios”.* Es básicamente la contraseña, adicional y de texto, que le vas a poner *(me imagino que por temas de respaldo)* a tu llave pública y privada (a la carpeta *.ssh* para poder acceder a ella); sin embargo, es preciso decir que, esto es opcional, pero lo más recomendable es que sí se le coloque *(ahora no lo haremos así, presionamos enter dos veces para darle entender a la consola que no queremos contraseñas de respaldo sobre la carpeta .ssh).* Verá algo más o menos así:

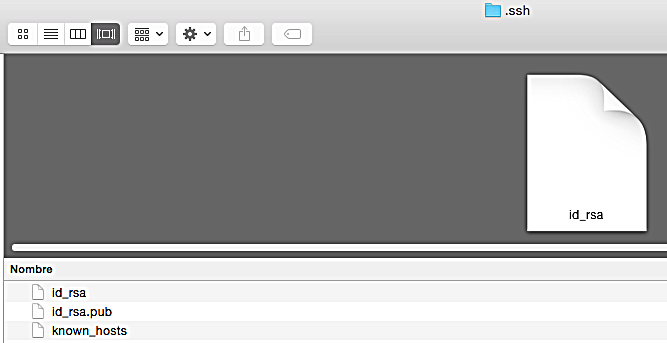


*Expliquemos por parte...*

Your identification... Quiere decir, tu identificación (llave privada) ha sido guardada en el archivo text: ***“id\_rsa”*** de la carpeta oculta: ***.ssh*** (se sobreentiende que en *home* como carpeta padre). La llave pública (your public key... ) también ha sido guardada en la carpeta oculta “.ssh”; sin embargo, ésta se guardó puntualmente en el archivo: ***“id\_rsa.pub”.***

Uno confirma que las llaves resultantes son de verdad por el ***key fingerprint*** (the key fingerprint is... )***: la huella.*** The ***key’s randomart image*** *es también otra forma de saber que tus llaves son de verdad y es una forma adicional, también, de compartir la llave pública.*

Ahora, vamos a situarnos dentro de la carpeta oculta *.ssh,* tal que así:



Como ya lo hemos dicho, dentro de esa carpeta se encuentra tanto nuestra llave pública como la llave privada. **id\_rsa** = archivo de texto donde está la llave privada; **id\_rsa\_pub** = archivo de texto donde está la llave pública.

Ahora, como nuestro interés es proteger a un sitio web, a nuestro repositorio en github más especificamente... Copiamos nuestra llave pública y la llevamos a github. **PERO STOP. PRIMERO VAMOS A CONFIGURAR LAS LLAVES SSH EN LOCAL.**

***Así es, primeramente, debemos agregar a la propia Mac las llaves (hacer que las identifique).*** Cómo se hace esto? En el mundo Mac es distinto que en Linus y en Windows, es de la siguiente forma:

*Primero revise antes que nada que el “ssh agent” esté corriendo*, **ssh Agent** es un programa auxiliar que realiza un seguimiento de las claves de identidad del usuario y sus frases de contraseña. El agente puede usar las claves para iniciar sesión en otros servidores sin que el usuario escriba una contraseña o frase de contraseña nuevamente. Esto implementa una forma de inicio de sesión único (SSO). Entonces, para revisar que todo esté bien aquí, ejecute el siguiente comando: **eval “$(ssh-agent -s)”***(presione enter)*

Luego, para Macs modernos (Mac OS Sierra o superior a 10.12) toca hacer un proceso particular con el ***archivo*** ***config*:** **hay que modificarlo; y, si no existe, toca crearlo.** Yo no lo tengo (verifiqué incluso mis archivos ocultos de la propia carpeta oculta ***.ssh,*** donde tengo que buscarlo,y no se encuentra), tengo entonces que crearlo con un editor de texto de mi preferencia, tal que así: ejecutando el comando ***“vim config”. Ojo, este archivo debe crearse dentro de la carpeta oculta “.ssh”; es decir, tiene que situarse antes en esta carpeta para crear el archivo “config”.***

Dentro del archivo “config” **debe escribir lo siguiente, respetando completamente la identación y las mayusculas y minusculas**, tal que así:

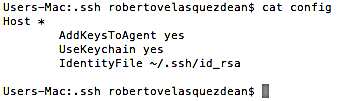
**Host \***

**AddKeysToAgent yes**

**UseKeychain yes**

**IdentityFile ~/.ssh/id\_rsa**

Luego, guarda. *(Si se fija, hemos hecho también un salto de línea al final y se ha respetado completamente).* En la terminal, se debe ver algo más o menos así:

***Algo importante:***Mi Mac no es *“moderna”,* por lo que el archivo *“config”* me da problemas. Para que Git me dejara importar el repositorio remoto de GitHub, ya con protocolos SSH, fue necesario eliminar, al menos, las líneas de código 2 y 3 (AddKeys... & UseKey...).

Verificado lo anterior y ya teniendo nuestro archivo *config,* con su texto especial, dentro de la carpeta *.ssh.* **Regresamos a *home*** *(con comando cd o cd ~).*

Ahora para añadir la identidad de la llave privada a nuestro computador (para que su entorno local la identifique y pueda trabajar con ella), finalmente debe ejecutar este comando: **ssh-add -K ~/.ssh/id\_rsa.**

**Y listo, ya con esto hemos configurado nuestras llaves SSH en local, entorno local (nuestro PC). Debe verse algo así:**



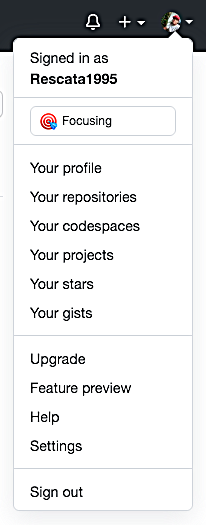
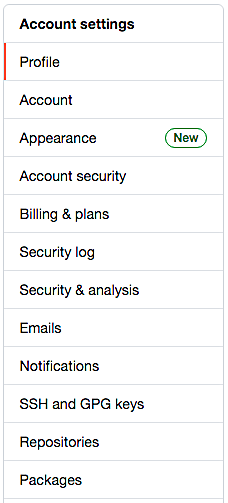
***Ahora sí, conexión a GitHub con SSH***

*Algo importante para aclarar antes de empezar:* si usted hace uso de varios computadores, digamos, tres. Los tres deben generar sus propias llaves SSH (su llave pública y su llave privada para cada tal) y hacer el mismo proceso que, anteriormente, ya fue explicado. Luego, con cada par de llaves SSH generadas en cada computadora, debe conectarse a su mismo repositorio en GitHub... lo que ahora explicaremos.

*pd. Es mejor tener llaves propias SSH para cada dispositivo; esto por un mismo tema de seguridad, no es bueno compartir llaves privadas de computador a otro (incluso aunque los computadores sean de un mismo dueño).*

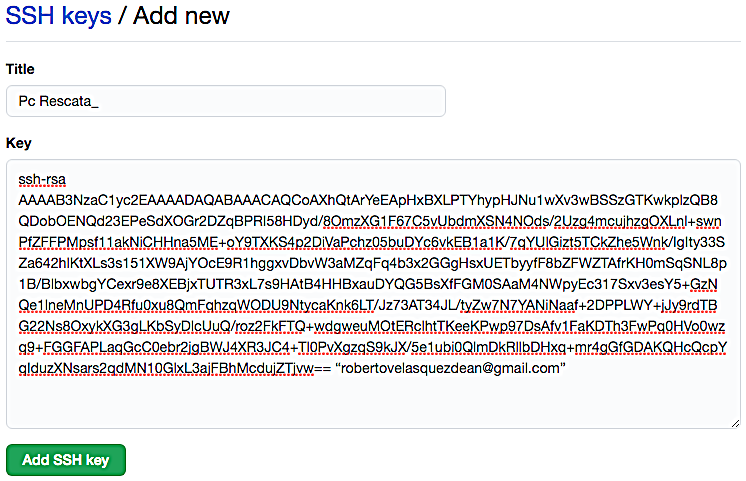
**Ahora sí! Vamos con GitHub.**

Tome su llave pública SSH, copiela y vayase a GitHub. Estando en *GitHub* se va a la sesión de ***Settings***y, posteriormente, se va al apartado que dice: ***“SSH and GPG keys”****, tal que así:*

***1) 2)***

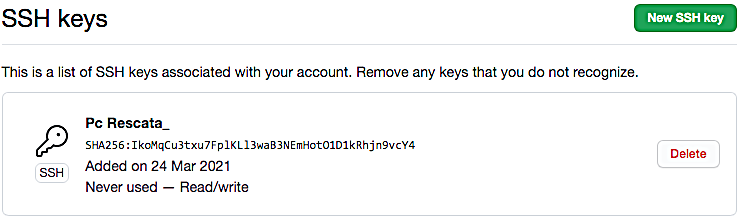
Mientras no tenga una SSH pública asociada, verá algo más o menos así: *“There are no SSH keys associated with your account”.* Presione entonces en: *.* ***Tome su llave pública*** *y...*

*Proceda, entonces, así (intuya que se debe colocar en cada espacio según el ejemplo a continuación):*



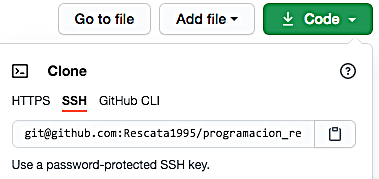
Luego de agregada la llave pública, luego de haber presionado en “Add SSH key”, proceda e introduzca su contraseña de *GitHub* (se la pedirán)*.*

Posteriormente, debe ver algo así:



La anterior imagen significa que ha registrado con exito su llave pública SSH en GitHub. *“Never used”* significa que nunca ha sido usada y *“Delete”* que puede ser borrada.

Luego, debe situarse en el repositorio de GitHub de su interés; y, si desea asegurarlo con una llave *SSH*, ya no más con protocolo *HTTPS,* debe hacer lo siguiente: presione *Code* y luego vayase a *SSH,* tal que así:



**Copiamos la URL contenida** y nos vamos de nuevo a nuestra terminal, puntualmente a donde tenemos situado, **ahora sí**, **nuestro repositorio local**: el que está vinculado con el repositorio remoto de GitHub de interés (que acaba de copiarle la URL de SSH)

Estando ahí, en su repositorio local de interés, debe ejecutar el siguiente comando: ***git remote -v***

*git remote -v* lo que nos muestra es la URL actual de nuestro repositorio local de interés; lo que queremos es, justamente, cambiarlo por la URL SSH del repositorio remoto, de interés, de GitHub. Veamos el proceso paso a paso:



Tenemos que nos salen dos repositorios con nombre de *origin,* este nombre nos interesa; ya que, necesitamos reemplazar las URL de todos los repositorios locales *(los Origin’s)* por la URL SSH copiada del repositorio remoto de GitHub... ejecutando el siguiente comando: ***git remote set-url origin +*** *pega***la URL SSH del repositorio remoto de nuestro interés de GitHub (el que habíamos copiado anteriormente).** Tal que así:

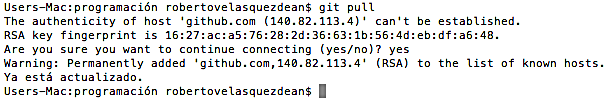


*Con este cambio, ya no tenemos una URL en protocolo HTTPS dentro de nuestro repositorio local (tampoco en nuestro repositorio remoto de interés); sino que, hemos pasado nuestras URL a protocolo SSH.* Revisemos:

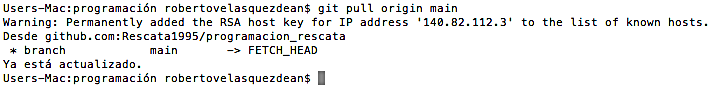


*Si se da cuenta, ya no sale al inicio del repositorio “https://...”*; es decir, ya no estamos bajo ese protocolo efectivamente. Listo, lo hemos logrado.

Ahora, nos traemos esta última versión de nuestro repositorio remoto de GitHub (ahora con protocolos SSH) a nuestro repositorio local, esto lo logra con el comando: ***git pull****.* Ejecute por favor...



Luego de teclear *“yes”,* debe llamar ahora, de forma completa, a la rama donde está situado su repositorio remoto de interés en GitHub, tal que así: ***git pull origin main*** (siendo que *main* es la rama del repositorio remoto en cuestión... pudo ser otra, pero fue *main*):



*Y Listo, esto es todo. Configuradas llaves SSH tanto en entorno local como en GitHub (y, puntualmente, en un repositorio remoto seleccionado del mismo).*

*un dato: ejecute comando en consola:* **defaults write com.apple.finder AppleShowAllFiles TRUE** *(presione Enter),*

*luego ejecute comando:* **KillAll Finder** *(presione Enter).* Todo este proceso para ver archivos ocultos. Sólo copie y pegue (y respete las mayusculas y minusculas).

***Tags y Versiones en Git y Github***

Antes de saber cómo ponerle una etiqueta a un “commit” o copia de seguridad para señalar con claridad hasta qué punto hemos trabajado en un proyecto o qué versión del proyecto hemos realizado, por ejemplo. Nos interesa saber primeramente la historia de nuestros “commits”; o, en últimas, de nuestro proyecto.

Recuerde que con el comando *git log* podemos tener un listado recién de nuestros últimos commits guardados por proyecto; sin embargo, este comando se derivan varias extensiones del mismo. Por ejemplo, ***git log --all*** nos muestra, en definitiva, todos los commits (desde el primero hasta el último) que han sido guardados historicamente para un proyecto en cuestión (no es un resumen de ellos, de los *commits* más recientes, no. Muestra la totalidad historica de ellos).

Ahora, hay una extensión (todavía más compleja) de este comando. Si ejecutamos ***git log --all --graph*** nos muestra los commits guardados; pero, según la interacción de las ramas que han *metido mano* dentro del proyecto. Es decir, se nos muestra todos los *commits* gurdadosa partir de cómo han interactuado las ramas y han hecho sus fusiones (dejando entre dicho también, claro, el origen de los *commits* según la rama en que se registraron inicialmente): esto herramienta resulta muy visual ya que muestra, explicitamente, las fusiones que han hecho las ramas entre sí (cada una con sus *commits)* por medio de trazos de lineas coloridas.

Ahora, hay otra extensión todavía más larga que hace exactamente lo mismo que el anterior comando *(mostraranos la historia de las ramas)*: sólo que muestra la información, de relación entre *commits y branches,* mucho más comprimida para su fácil e inmediata comprensión: ***git log --all --graph --decorate --oneline***

Ahora, un truco que aprendimos de los comandos de la Shell. Es probable que no nos acordemos cómo ejecutar, por ejemplo, la extensión del comando git log: ***git log --all --graph --decorate --oneline,*** resulta muy larga. Qué podemos hacer entonces? Asignarle un ***alias*** y, reemplazar toda esa línea de código que ejecuta especificamente dicho comando, por un *nombre* más corto que se encargue de hacer lo mismo (para facilitarnos más las cosas), tal que así, por ejemplo: ***alias arbolito=“git log --all --graph --decorate --oneline”.*** Entonces, lo que hemos hecho es pasarle a la palabra ***arbolito*** la misma función del comando ***git log --all --graph --decorate --oneline.*** Así de sencillo. Si escribe la palabra ***arbolito*** en su terminal y presiona enter se ejecutará el comando ***git log --all --graph --decorate --oneline.***

Ahora sí, vamos con los ***Tags.***

Ya teniendo el listado de comandos *git log* claros (y habiendo ejecutado uno); para crear (añadir) un **tag,** para darle una etiqueta a un *commit* en especifico (y recalcar alguna información relevante del *commit* en función al proyecto), se procede de la siguiente manera:

1. Se copia el identificador del *commit* (el de su interés) y se ejecuta el siguiente comando: ***git tag -a “aquí pone el nombre de su etiqueta” -m “aquí va un mensaje referencial, así como hace con los commits” +identificador (lo copia aquí).***

*Veamos la ejecusión de este comando con un ejemplo:****git tag -a V0.1 -m “esta es la versión uno del proyecto” 1b56158***

Eso último, ***1b56158,***sería el identificador copiado del commit de nuestro interés; es entonces, de la anterior forma, que se le asigna un ***tag*** a un *commit.* El ***tag*** en este ejemplo sería ***“V0.1”.***

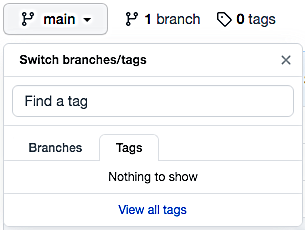
Ahora, para verificar que hemos registrado con exito el *tag “V0.1”*, podemos ejecutar el siguiente comando: ***git tag***, y se nos mostrarán todos los *tags* que hemos creado (y asignado a *commits*) en el historial de nuestro repositorio o proyecto actual. Sin embargo, si queremos ir más allá, y saber a qué ***tag*** pertenece a un *commit* en especifico*,* podemos ejecutar el siguiente comando: ***git show-ref --tags,*** verá algo más o menos así:



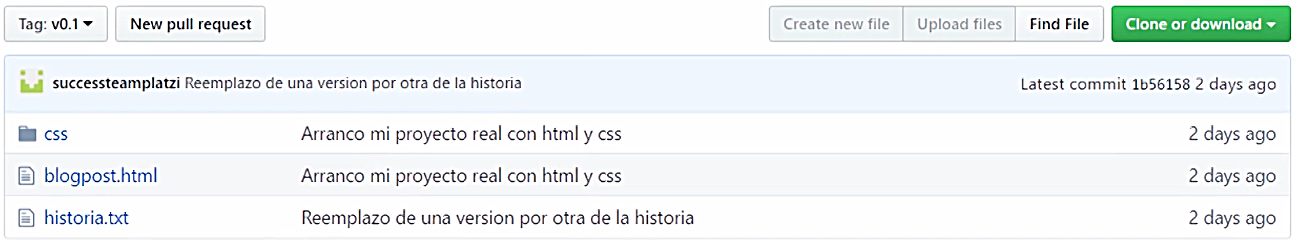
Ahora, para enviar los *tags* creados en Git a GitHub (que realmente no son considerados como *cambios* dentro de Git ni resultan muy utiles -su uso- dentro de Git propiamente) se ejecuta el siguiente comando: ***git push origin --tags***

Los *tags* tienen la fama de resultar útiles solamente en GitHub, es así porque en GitHub es donde se trabaja en proyectos conjuntamente con otros desarrolladores, entonces *esos otros desarrolladores* podrían ver qué *versiones* ocurrieron o dejaste tú resueltas (digamos, es un punto referencial que se deja del proyecto).

En GitHub, para revisar los *tags* que hemos exportado recién desde Git (desde nuestro repositorio local ya vinculado a un repositorio remoto de interés), tiene que situarse sobre:



Ahí en *Tags* podrá visualizar todas las etiquetas asignadas (tags) a los *commits* de su interés; es decir, si desea ver un *tag* en especifico; se clickea sobre el *tag* en cuestión y, posteriormente, se nos muestra toda la información relacionada a los registros y archivos cargados (y guardados) hasta el avance final del *commit,* al que se le ha asignado la etiqueta (tag) *clickeada:* todo esto bajo el nombre de la etiqueta propiamente (a la que pertenece el *commit* en discusión). En nuestro ejemplo, dentro de Git, la etiqueta creada sobre un commit se llama: ***V0.1,*** y el commit está con el identificador: ***1b56158***. Se vería algo más o menos así:



Ahora, si por ejemplo, queremos borrar un *tag* en especificoque hemos creado en Git (por alguna razón, no importa), se hace ejecutando el siguiente comando: ***git tag -d “y aquí va el nombre del tag de interés”***; por ejemplo, si quisieramos borrar el *tag* “V0.1”, *sería así:* ***git tag -d V0.1.*** Y, de nuevo, si deseo exportar los *tags* actuales de git, luego de eliminar uno o varios, debo ejecutar el mismo comando: ***git push origin --tags***

Ahora, si ya no deseo ver los *tags* en GitHub que eliminé de Git, debo ejecutar otro comando; digamos, si puntualmente ya no quiero que se vea el *tag “*V0.1” en GitHub (porque ya lo eliminé de Git), debo ejecutar así: ***git push origin :refs/tags/v0.1*** *(lo último que se coloca es el nombre del tag que desea eliminar de vista en GitHub).*

*Y listo, así se crean (y se eliminan también) las etiquetas de los commits (o Tags).*

***Manejo de ramas (branches) en Git***

Una de las funciones más importantes de Git es el control de las ramas de desarrollo, conocidas como Branch, que ayudan a mejorar la creación de un proyecto. Las Git Branch ayudan a organizar múltiples versiones de una aplicación. Por eso es muy importante trabajar con ellas.

El uso de las ramas de desarrollo de Git es una excelente manera de trabajar con una aplicación mientras rastreas sus versiones.

Usar las ramas de desarrollo de Git tiene múltiples ventajas. Sin embargo, queremos enfatizar las siguientes dos:

- Hace posible desarrollar nuevas funciones para una aplicación sin obstaculizar el desarrollo de la rama principal.

- Con Git Branch es posible crear diferentes ramas de desarrollo que pueden converger en el mismo repositorio. Por ejemplo, una rama estable, una rama de prueba y una rama inestable.

***Cómo usar el comando Git Branch***

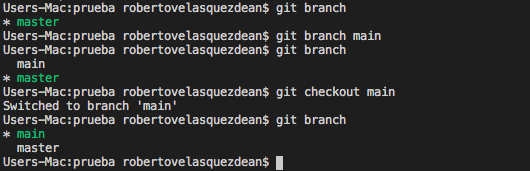
Los usos de las ramas de Git son inicialmente simples, como notarás en el uso de los comandos de Git Branch. Pero como con la mayoría de las cosas, cuantas más ramas tengas, más difícil será administrarlas.

En cualquier proyecto Git puedes ver todas las ramas ingresando el siguiente comando en la línea de comando: “git branch”.

Si no has creado una rama antes, no verás ningún resultado en el terminal. Crear una rama es realmente simple: “git branch [nueva\_rama]”. Luego, tienes que pasar a la rama de desarrollo recién creada. Para hacer esto, ejecuta el siguiente comando: “git checkout [nueva\_rama]”.

La salida te informará que cambiaste a una nueva rama. Para el ejemplo, digamos que llamamos a esta rama “prueba”, entonces será: “ Switched to branch ‘prueba’ ”.

Ahora, en esa nueva rama de desarrollo, puedes crear tantas modificaciones de código como quieras, sin tener que cambiar nada en la principal. Como puedes ver, el programa se mantendrá organizado para nuevas inclusiones de código. Si ejecutas el comando para volver a listar las ramas, verás que se agrega una nueva rama y que estás en ella: *git branch.* Tal que así:



Hay algo que debes tener en cuenta si quieres crear una nueva rama de desarrollo. Primero, debes comprometerla con la rama principal para que Git entienda cuál es la rama maestra. Si no haces esto, obtendrás un error. Primero, crea el enlace y luego crea las ramas de desarrollo.

***Si quieres eliminar una rama de Git, puedes hacerlo con el siguiente comando: “git branch -d [nombre\_de\_ rama]” (sigamos suponiendo que la rama se llama: “prueba”).***

Sin embargo, para hacer esto, no debes estar en la rama que quieres eliminar. Entonces, en este caso, te mueves a la rama maestra y desde allí eliminas la rama que acabas de crear, tal que así:

1. “git checkout master”.

***2. “git branch -d prueba”... si se pone complicado, “git branch -D prueba”***

Finalmente, llega un punto en el que has realizado muchas modificaciones en una rama de desarrollo adicional y de “tasting” (creada a partir de la principal) y se vuelve estable, por lo que quieres vincular eso nuevos cambios a tu rama principal. ***Para eso, existe el comando “merge” (fusión).*** Primero, ubica la rama de desarrollo (la principal) a la que se le adjuntarán los cambios de la segunda rama (la rama de “tasting” que creamos a partir de la principal para probar modificaciones y ver con qué quedarnos de ella, de forma definitiva, al llamarla luego en la rama principal).

Por ejemplo, si queremos adjuntar la rama de **“prueba”** a la rama **master** (porque ya nos gustó en su totalidad los cambios que hicimos en la rama de “prueba” y queremos vincular esas modificaciones definitivas en nuestra rama master: *la rama con la que presentaremos nuestro proyecto final, por decir algo*); lo que hay que hacer es, primero, ubicarnos de nuevo en la rama master (“***git checkout master***”) y fusionarla con la rama de “prueba” usando el siguiente comando: ***git merge prueba***

***Ahora, un detalle,*** *si nuestro repositorio local se encuentra vinculado a un servidor externo, tipo repositorio remoto GIT HUB; antes que nada,* ***e incluso antes de crear nuestra rama “prueba” en repositorio local*** *(por continuar con nuestro ejemplo), para manipular distintas ramas secundarias que desee, debe crearlas primero en GITHUB (antes que en su repositorio local, recuerde).*

*Luego, en repositorio local, sí procede a crearlas (con el mismo nombre con el que las registró en GITHUB); y, es desde ahí, desde su repositorio local, donde hará todas las modificaciones necesarias en las ramas secundarias que crea o creó a partir de la rama principal (en este caso, hemos creado la rama “prueba” a partir de la rama principal “master”, esa rama “prueba” ya debio estar asi tal cual creada en GITHUB desde un principio). Entonces, luego de que haga los respectivos cambios definitivos en repositorio local sobre la rama “prueba”; luego de eso, es que usted va a exportar dichos cambios a repositorio externo GITHUB con un comando:* ***“git push”. Y si dichos cambios tornan ser definitivos y utiles para incorporalos ya en la rama principal (master), es ahí que se hace una fusión entre rama principal “master” y rama “prueba”, tal cual como la explicamos.***

***En conclusión, una rama es como una copia de su proyecto. Se utiliza principalmente para:***

1. agregar una característica en desarrollo.

2. colaborando en el mismo proyecto con otros desarrolladores.

3. no romper todo tu repositorio.

4. no molestar a tus compañeros de trabajo.

5. El propósito de una rama es aislar su trabajo del código base principal de su proyecto y / o del trabajo de sus compañeros de trabajo.

***Otros comandos Git interesantes***

***git status:*** *nos indica en que rama (branch) estamos ubicados actualmente.*

***git checkout -b “nombre de rama”:*** *este comando lo que hace es crear una nueva rama secundaria; pero, adicionalmente y luego de creada, la terminal nos lleva a la ubicación de dicha rama. Es decir, no sólo la crea, sino que además nos situa en ella. (antes de ejecutar este comando, idealmente asegurese de estar ya situado en la rama principal, la rama de origen).*

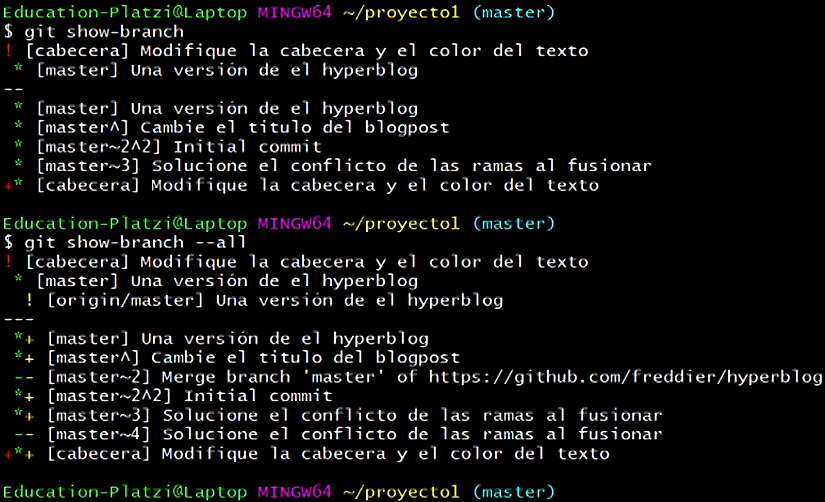
***git branch -m “nombre de la rama” “nuevo nombre de la rama”:*** *este comando lo que hace es cambiar el nombre de una rama existente, especificando también el nuevo nombre que va a recibir. (antes de ejecutar este comando, idealmente asegurese de estar ya situado en la rama principal, la rama de origen).*

***git branch -h:*** *comando que ejecuta el asistente de ayuda para conocer todo lo se puede hacer con las ramas (Branch).*

***Manejo de ramas (branches) en GitHub***

Ahora nos interesa saber cómo exportar las sub-ramas (o ramas de desarrollo) que creemos, a partir de una rama principal, en Git a GitHub. En GitHub siempre visualizamos la rama *origin* *master* u *origin* *main,* que vendría siendo la rama principal (la que usted también tiene inicialmente en su repositorio local, *main*). Sin embargo, usted también podría tener sus ramas de desarrollo, las que crea en repositorio local, en GitHub, ¿cómo sería?

Empecemos primero conociendo un par de comandos asistenciales para ese fin, por ejemplo: ***git show-branch,*** este comando nos muestra cuáles son las ramas que existen (en primer lugar); y, posteriormente, cuál ha sido el historial de ellas. Una extensión del anterior comando podría ser el siguiente: ***git show-branch --all,*** hace lo mismo pero nos muestra más detalles de ese *historial de ramas*; y, entre otras cosas, nos permite visualizar también cuáles ramas remotas están vinculadas a nuestro repositorio local (las ramas de tipo *Origin).* Tal que así, por ejemplo:



*Se percata de las diferencias?* En el último comando ejecutado ***git show-branch --all***, por ejemplo, podemos ver que existen *dos ramas locales y una rama remota de una de las dos ramas* locales, tenemos que: **[cabecera]** es una rama de desarrollo y está en local; **[master]** es, inicialmente, nuestra rama principal (que también está en local); y, por último, **[origin/master]** es la única rama remota que, de momento, tenemos en GitHub, la cual parte de la rama principal en local *master.*

Ahora bien, existe un comando muy especial de *git* que nos facilita toda esa tarea de entender el historial entre ramas y commits (e incluso *etiquetas);* tal que nos muestra, toda esa información acumulada, de manera exageradamente visual. El comando es: ***gitk.***

***Ahora bien, a lo que vinimos.*** **Si quisieramos, por ejemplo, exportar alguna rama de desarrollo que todavía no tenemos en nuestro repositorio remoto;** pongamos el caso de la rama *cabecera,* **sería así:**

Primeramente, nos tendríamos que situar en la rama de desarrollo de interes que deseamos exportar; tal que así (en este caso): ***git checkout cabecera*** *(suponiendo que ya habíamos creado anteriormente la rama “cabecera”).*

Luego, simplemente *empujarla* (empujar a la rama), tal que así: ***git push origin cabecera*** (es preciso señalar la palabra *“origin”;* ya que con eso se especifica que quieres crear, a partir de tu rama de desarrollo *cabecera* - y que está situada en repositorio local-, una **rama remota** en base a esa rama *cabecera).* La rama remota entonces pasaría a recibir el mismo nombre (cabecera) y conservaría el mismo contenido en términos de *commits*.

***Algo puntual para recordar:*** *todas las ramas de desarrollo, las que crea el mismo usuario (las que yo llamo “sub-ramas”) deben ser creadas a partir de la rama principal; es decir, desde rama* ***master*** *o rama* ***main*** *(son los dos nombres que reciben, convencionalmente, las ramas principales: donde se trabaja al ser inicializado Git).*

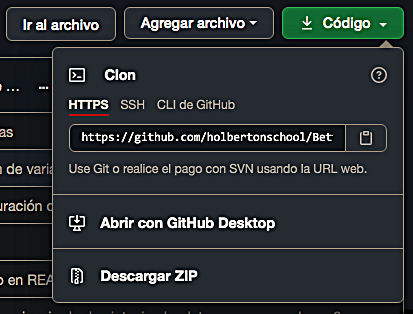
***Ahora veremos cómo configurar multiples colaboradores en un repositorio de GitHub***

Primero, lo de siempre, el nuevo colaborador (o desarrollador) debe tener una cuenta de GitHub ya registrada (a este paso usted ya sabe cómo registrar su cuenta en GitHub); luego, posteriormente, debe irse al repositorio remoto de interés en el cuál desea incorporarse. Entonces, para incorporar un repositorio de GitHub ya existente (de un tercero, por ejemplo) en su entorno local de trabajo, en Git, debe hacer una ***clonación***de dicho repositorio remoto en cuestión*;* sin embargo, antes de llevar a cabo ese proceso, debe cerciorarse primero de que el correo electronico con el que se registró en GitHub, y su nombre de usuario, estén registrados en su entorno local mediante el comando de Git: ***git config -l***; de lo contrario, debe registrarlos primeramente.

*Y, ojo a lo siguiente: No debe inicializar Git cuando se trata solamente de importar con* ***clone*** *un repositorio remoto en nuestro entorno de trabajo dentro de la terminal.* Sin embargo, sí es preciso que deje claro hacia qué dirección (y qué directorio) desea que sea el lugar donde se importe el repositorio remoto de GitHub de interés; es decir, crear (si no la tiene creada) la carpeta en su entorno local donde desea trabajar el repositorio remoto que será importado desde GitHub.

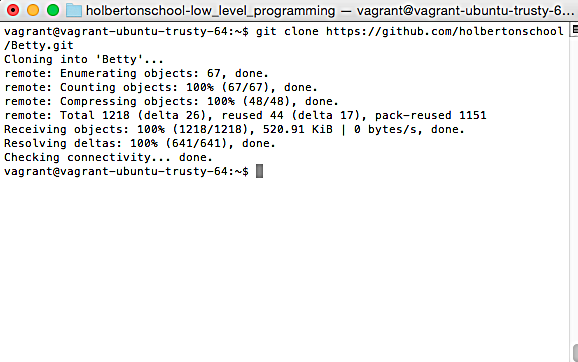
Confirmado lo anterior procedemos a importar el repositorio remoto de un tercero, que está en Github, en nuestro entorno local con el comando ***git clone:*** este comando tiene la función de importar un repositorio remoto de github (u otro servidor externo), que aún no está vinculado en el entorno local (por lo general esto se hace cuando queremos vincular el repositorio remoto o en nube de un tercero). Si el repositorio no nos pertenece, para importarlo debe hacerlo de la siguiente forma (partamos de un ejemplo):

Ya situados en el repositorio remoto, debe irse a la parte que dice “Código” (de color verde):



Ya situados sobre “código”, observemos la URL que queda en evidencia; y, en base al protocolo de seguridad con el que se desee trabjar (HTTPS o SSH), se copia la URL de nuestro interés. Luego, para vincular este repositorio remoto a nuestro entorno de trabajo local, sería así:

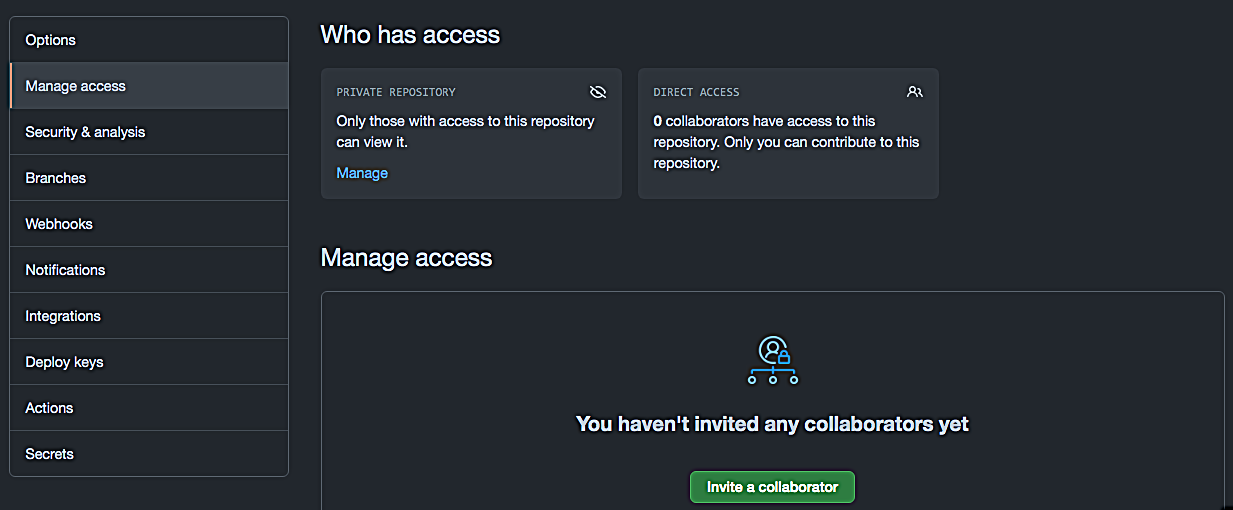
Situados ya en nuestro directorio de trabajo seleccionado, ejecutamos el siguiente comando: ***git clone + URL copiada (según protocolo HTTPS O SSH)*** y listo. Por ejemplo, debe verse algo más o menos así:



Este proceso ya nos inicializa por sí solo a GIT dentro de nuestro directorio de trabajo seleccionado; es decir, nos monta como nuevo repositorio local el repositorio remoto que ha sido importado desde GitHub (y viene con todos sus archivos, commits y accesorios).

Vale, hasta aquí todo bien. Pero, se nos olvida verificar algo super importante para antes que se nos ocurra, como terceros, empujar (hacer *push*) de nuestros cambios personales, desde nuestro repositorio local recién clonado, al repositorio remoto original de GitHub que recién cloné: ***Y es contar con los permisos de autorización, por parte del dueño real del repositorio remoto de GitHub clonado, para hacerle modificaciones al mismo.***

Entonces, el propietario del repositorio remoto en GitHub debe hacer lo siguiente: situarse sobre las configuraciones *settings* del repositorio en cuestión *(no sobre la configuración del usuario)*; y, lo siguiente, hacer click en *Manage access.* Ahí debe agregar al nuevo *collaborator* por medio de su *username*, tal que así:



Y listo, el nuevo *colaborador* podrá posteriormente aceptar la invitación de *colaborar* con el repositorio remoto al que un tercero, en GitHub, le acaba de dar u otorgar permisos de edición o modificación. De aceptar la invitación, ese nuevo colaborador (nosotros) ya podra *empujar* commits desde su entorno de trabajo local a ese repositorio remoto de GitHub que, inicialmente, no nos pertenece; pero, al que hemos sido posteriormente invitados. *De interés:* en esta misma sesión se pueden eliminar también a los colaboradores invitados.

*Ahora bien, a modo de conclusión, es una buena práctica que siempre haga un* ***git pull (y mejor si especifica qué rama, del repositorio remoto de GitHub, desea importar)*** *justo antes que se disponga a trabajar en su entorno de trabajo local con git* ***(y justo antes de, incluso, empujar un nuevo commit almacenado)****; por lo menos, hagalo cuando ya es sabido que trabaja en un proyecto compartido con varios desarrolladores que influyen sobre un mismo repositorio remoto. Esto para qué? Para que se mantenga actualizado de todos los cambios (o commits) cargados por otro, en el repositorio remoto de GitHub, mientras usted estuvo por fuera de la PC.*

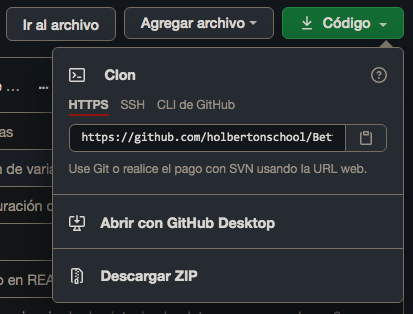
*Resulta ideal que se especifique la rama a importar, también, debido a que hay veces en que uno, como nuevos colaboradores de un proyecto en un repositorio remoto en GitHub al que fuimos invitados (por ejemplo), sólo contamos en entorno local con nuestra rama base (la rama* ***master*** *o* ***main);*** *entonces, se hace necesario también en ese sentido importar al primer instante todas las ramas que están en el servidor y que no tengamos en nuestro entorno local todavía; tal que así:* ***git pull origin “nombre de la nueva rama de desarrollo”.***

*Puntualmente si lo que se requiere es que trabaje, usted como desarrollador invitado, en una sola rama a la que fue asignado; entonces,* ***jala*** *esa rama en cuestión y; posteriormente, se situa en ella (ya que no le corresponde tocar otra rama), eso se hace con:* ***git checkout “nombre de la rama que acaba de importar y que le corresponde”.***

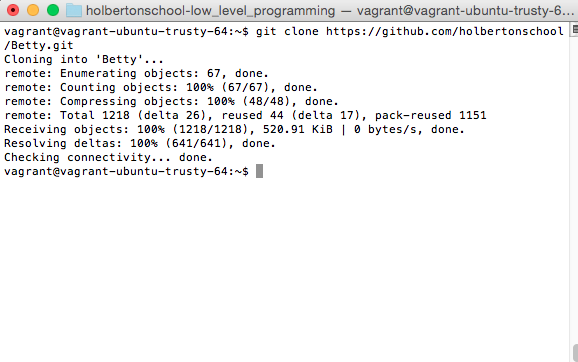
*Ahora, si ya es sabido que, en un repositorio remoto se está trabajando con varias ramas (no sólo con una); entonces, cada que haga un cambio (o registre un nuevo commit) en su entorno local, debe percatarse primero sobre qué rama lo está haciendo; de tal manera que, cuando desee* ***empujar*** *ese último commit al repositorio remoto de Github, debe también especificar a qué rama desea enviarle esa última copia de seguridad almacenada: estando en sintonia la rama de mi entorno local con la rama destino del repositorio remoto:* ***git push origin “nombre de la rama congruente a mi rama actual de trabajo en entorno local”.***

***Últimos detalles y repaso sobre 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). Si el repositorio no nos pertenece, para importarlo debe hacerlo de esta forma.*

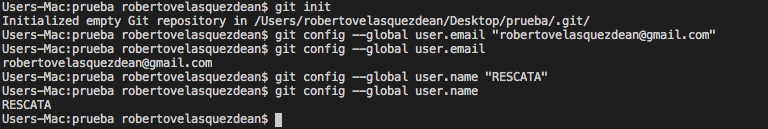


*Nos dirigimos a esta parte del repositorio remoto que queramos importar (este ejemplo se hace con GitHub) y copiamos link. Luego, para vincularlo a nuestro repositorio local, sería así:*



***Dato de importancia: trate de que se encuentren sincronizadas las ramas (branch) de su repositorio local (GIT) con el repositorio remoto (GITHUB); si no lo están, pues sincronicelas, no podrá cargar la información de su repositorio local a remoto (y viceversa) si no están sincronizadas .***

***Opcional: No se olvide registrar su nombre de “user” y “email” por cada GIT inicializado, GIT REMOTO necesita saber quién está registrando copias de seguridad por cada repositorio local en su nube o servidor externo: la idea es conocer quién nos está colaborando con el material cargado en GIT HUB por ejemplo. Se ve más o menos así:***



***Obligación: tampoco se olvide de su user en GITHUB: GITHUBuser; ni de su contraseña: GITHUBpassword. La terminal se lo exige luego de que vincule y trabaje de la mano con dicho repositorio remoto.***

***Dato: Si instalo GIT por medio de la terminal, pues ejecute GIT directamente desde la terminal. Evitese futuros dolores de cabeza.***

***Dato: No se puede inicializar git luego de ya inicializado antes una carpeta, esto va a desconfigurar su repositorio local actual y todo lo que ya haya hecho en él; pues, justamente le estás pidiendo a GIT que se restaure, que se registre de nuevo un nuevo repositorio; entonces, si por ejemplo, desea cargar algún cambio a su repositorio remoto que anteriormente ya estaba vinculado al local... no pasará nada, porque prácticamente ahora está trabajando en un repositorio local nuevo que no está vinculado a ningún servidor externo (le tocaría crear uno nuevo, un nuevo repositorio remoto y vincularlo al nuevo repositorio local inicializado).***

***Ahora, si por ejemplo desea, eliminar un repositorio de Git creado con “git init” en un directorio; lo hace de la siguiente forma:***

***cd “directorio\_de\_interés\_donde\_inicializó\_git\_que\_desea\_eliminar”  
rm -rf .git***

***Es de la anterior manera porque “Git” guarda, de forma automatica, todos los cambios realizados en un directorio llamado “.git” (dentro del directorio donde se inicializó “git” inicialmente)***