OpenBootcamp

Git / Github

* 1. **- Introducción al Control de versiones**

El **control de versiones** consiste en guardar los distintos cambios que voy realizando en mis proyectos en distintas carpetas, a razón de poder volver en el tiempo en caso de querer revertir un nuevo cambio o recuperar alguna forma de trabajo realizada.

Esto podríamos lograrlo creando distintas carpetas con su nombre, día y hora de guardado, pero esto daría paso a una infinidad de carpetas y carpetas que sería casi imposible de manejar, incluso de entender.

Un control de versiones es distinto de *copias de seguridad*.

Además, a la hora de trabajar en equipo no es posible. Deberías compartir todas las carpetas a través de mail o USB. Luego, configuraría uno de cada lado y al volver a verse, sería un problema para ver qué cambios deben quedarse, cuales no, etc.

Esto no se hace así y no funciona.

Existen 3 tipos de controles de versiones en la actualidad: ***locales, centralizados*** y ***distribuidos***.

**Locales**

Debido a esto es que se inventaron los sistemas de control de versiones. El primero que se creó fue “**rcs**”, que era para el control de versiones de manera *local*, lo malo es compartirlo. Hoy ya no está utilizándose.

Se iniciaba el repositorio con el comando en el *cmd*:

* **rcs**

Para meter un fichero en **rcs** se utiliza el comando:

* **ci -u *nombreFichero.extension***

Esto genera un fichero “*nombreFichero.extension****,v”*** y te pediría que ingreses una descripción terminando con un punto (**.**)

**>> Descripcion inicial del readme // Enter**

**>> . // Enter**

Con ***co*** llamarías a la version que quisieras ver.

Si bien ya no existe esta forma de trabajar, si fue el punta pie inicial.

**Centralizados**

A partir de aqui, surgieron los sistemas de control de versiones **centralizados**, algunos se siguen utilizando hoy en dia.

Existe la evolución del rcs llamado **cvs** y uno mas avanzado, el **subversion** (**svn**).

Ser *centralizados* significa que los cambios y las versiones se van a guardar en un **servidor**, por lo que aunque mis ficheros se dañen, mi carpeta se borre o mi computadora se rompa, puedo recuperar esos archivos.

A través del comando *cvs* podemos recuperar información de un servidor:

* **cvs -d** [**anoncvs@anoncvs.ca.openbsd.org:/cvs**](mailto:anoncvs@anoncvs.ca.openbsd.org:/cvs) **checkout -P src**

Aquello que aparece en celeste es una ruta de servidor:

* **anoncvs@...**: es el usuario
* **…**@**anoncvs.ca.openbsd.org…**: es la url
* **…@...:/cvs**: es la carpeta que vas a extraer
* **checkout**: significa que traiga el código del servidor
* **-P src**: es la rama dentro de la carpeta

**cvs** hasta el 2010 era la norma, y se utilizaba junto a la herramienta **Source Forge** ( *sourceforge.net* ).

**Subversion** es una herramienta parecida a *cvs*, pero con algunas otras facilidades y mas amigable para el desarrollador, además aparecieron los conceptos como las *branches*.

* **svn co** [**https://svn@.us-west.FreeBSD.org/ports/head**](https://svn@.us-west.FreeBSD.org/ports/head) **/PortsFBSD**

La principal diferencia es que al final especificábamos las carpetas de donde queríamos bajarlo.

Una de las principales desventajas que continuaban eran la imposibilidad de commitear cambios al servidor central si no había conexión o si estaba caido.

**Distribuidos**

Llego finalmente la tercera generación con muchas ventajas respecto de las anteriores: la principal diferencia es que estos funcionan con server y con local.

Es decir, podemos enviar cambios dentro del propio ordenador, tenemos una copia completa en local y, además, podemos subir a un servidor para compartir con mis compañeros.

Ademas, existe el historial de confimracion (commits), por lo que podemos ir y volver a cualquier versión que querramos o necesitemos.

Con el comando **git log** podemos ver el historial de confirmacinoes:

* **git init**
* **echo hola > mensaje**
* **git commit -am ‘Mensaje inicial’**
* **git log**

Al iniciar **git**, se creará una carpeta inivisible llamada “.git” y es donde se va a guardar toda la información de los commits que vayamos realizando.

Para poder compartir con tus compañeros,deberíamos hacer como la vez anterior, pasarlo por zip o correo, pero hay dos opciones: crear un servidor o usar Github, Gitlab, Bitbucket o Source Forge.

Una de las ventajas de Source Forge en comparación a las demás (adem,as de poder trabajr con Mercurial y Subversion) es que te ofrece hosting grtuito y base de datos, mientras que las demás ofrecen solo hosting para páginas estáticas.

Una práctica común es que se utilice los beneficios de Source Forge y se uilicen las demás páginas como copias de seguridad.

Utilizaremos Github, ya que es el mas utilizado.

Para comenzar, debemos crear un repositorio en Github.

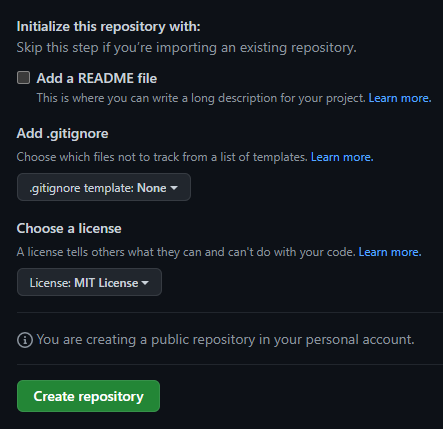
Daremos click en donde dice “New”. Luego nos pedirá nombre, descripción y demas.



La descripción es opcional, luego podemos cambiar la privacidad del repositorio, es decir, que sea **Public** (que cualquier persona fuera y dentro de Github pueda ver su contenido) o **Private** (necesita nuestra invitación para que un usuario de Github pueda verlo).

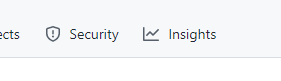


Y por último, nos preguntará si queremos agregar unos ficheros predeterminados: un **README.md** (donde irá la descripción del repo), un **.gitignore** (donde especificaremos qué archivos no queremos guardar en el repo) y una **licencia** (para proyectos opensource se recomienda elegir la licencia *MIT Licence*).



Finalmente, crearemos el repositorio. Ahora, cualquier persona que entre al enlace tendrá acceso al repositorio.

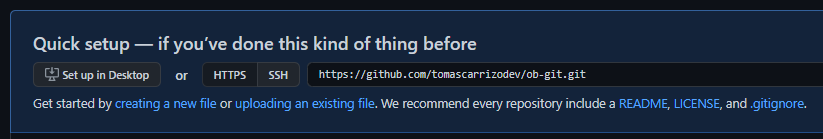
La diferencia principal radica en las opciones de configuración que tendrá el dueño del repositorio.



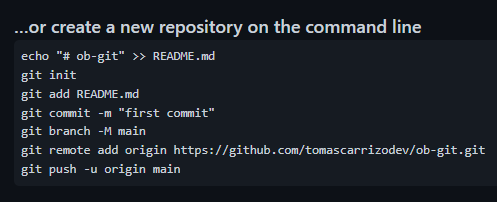


Donde tendremos muchas mas opciones para contralar el repo.

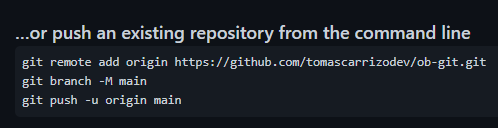
En caso de haber iniciado el repositorio sin ningún archivo dentro, nos mostrará esto:



Para que conmencemos a llevar nuestro ficheros al repositorio, si aun no lo hemos creado, tendramos estas opciones:

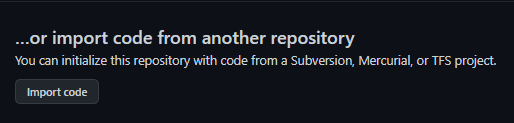


Utilizando esos comandos. Crearemos un repositorio desde cero y lo conectaremos con el remoto. En el caso de que ya tengamos creado el repositorio, tenemos este opción:



Que con tan solo copiar y pegar estos comandos en la consola, ya estará enlazado al remoto.

Tambien tenemos la opción de importar otro repositorio de otra parte.



Antes de continuar, es mejor aclarar algunas cuestiones.

Solia hablarse de listas blancas y negras, de maestros y esclavos por la claro connotación negativa que conllevan, por lo que la rama principal en varios repositorios de git la rama principal pasó a llamar **main**, aun asi es posible encontrar repositorios con la rama *master* como principal.

Para enviar el historial de confimaciones al remoto se utiliza el comadno **push**:

* **git push -u origin main**

**origin** significa que se habla de la rama del repositorio remoto. Esto debe hacerse la primera vez, luego no hace falta especificar, a menos que querramos enviar los cambios a otra rama del repo remoto.

* 1. **– Repositorio Local I**

Supongamos que has comenzado un nuevo proyecto y no sabes por dónde empezar con el control de versiones, hay una solución.

Comenzaremos con el comando:

* **git init .**

Lo que va a hacer este comando es crear un repositorio local en la carpeta actual. Con el comando **ls** (en Git Bash, no cmd) seremos capaces de ver el contenido de la carpeta, y con **ls -altr** veremos el contenido con las carpetas y archivos ocultos includios.

* **ls -altr**

Para empezar con el control de versiones, puesto que aun no ha comenzado, debemos agregar o mandar los archivos que querramos al repo mediante el comando **add**. Después de este colocaremos el nombre del archivo o carpeta que vamos a subir o si queremos subir todo, utilizamos un **.** (punto).

* **git add <nombre>**
* **git add .**

Ahora, luego de esto, no significa que el archivo ya fue enviado, está en un lugar **intermedio** esperando por una confirmación que elevaremos a través del comando **commit**.

* **git commit -m “mensaje”**

Antes de hacer la confirmación, vamos a ver la configuración de git, en donde cada commit está relacionado con un nombre de usuario y un correo.

Existen 3 configuraciones posibles, primero está la **System Wide** (casi no se utiliza), luego está la zona **Global** (se aplica a todos los repo de la carpeta usuario), y finalmente la **Local** (repo de la carpeta actual).

* **git config –-system**
* **git config –-global**
* **git config --local**

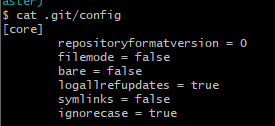
En este sentido, una configuración va pisando a la otra, la Local pisa a la Global que a su vez pisa a la System.

Con el comando **list** podemos ver las configuracines de cada tipo.

* **git config --list –-system**
* **git config --list –-global**
* **git config --list --local**

A través del comando **cat** podemos ver lo que contiene el fichero config del repo en local. Y es aquello que vamos a configurar.

* **cat .git/config –-local**



Para configurar algún error dentro del fichero o porque debamos utilizar otro usuario y mail, por ejemplo del trabajo, lo haremos de la siguiente manera.

* **git config –-local user.name “Tomas Carrizo”**

De esta manera, accedemos a las variables de dentro del fichero config y le asignamos un nuevo valor, como si de entrar a un objeto de JavaScript y sobrescribiendo una de sus keys.

Esto funciona tanto para *system* como para *global*.

Con el comando **pwd**, podemos ver la ruta en las que se encuentra abierta la consola.

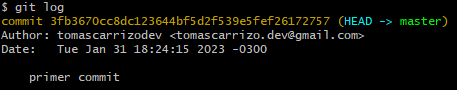
Tambien podemos crear nuevas variables a pesar de que no se encuentren en el config. Como en el *local* no hay un mail definido podríamos ponerlo a través del mismo comando de antes.

Y por fin podemos usar el commit para enviar los cambios. Con la configuración **-a** vamos a hacer referencia a todos los cambios.

* **git commit -a -m “primer commit”**

A través del comando **log**, podemos ver el historial de confirmaciones.

* **git log**



Aquí podemos observar el id del commit, el mensaje que enviamos, la rama a donde se envio, el usuario y el mail y la fecha y hora en que se hizo tal commit.

Existe otra forma de crear repositorios locales a través del comando **–bare** seguido del nombre de la carpeta donde se alojará el repo.

* **git init --bare aplicacion**

Es una función bastante parece a un repositorio remoto pero desde la misma computadora, almacenas los cambios en esta carpeta mientras trabajas desde otra.

Volvamos a la primera carpeta que creamos. Con el comando **rm -rf**  eliminamos todo lo que no queramos.

* **rm -rf .git**

La principal diferencia con un repo normal, es que el repo *bare* contiene todo el historial de cambios pero no vamos a poder trabajar con él, hay que clonarlo a la carpeta en la que queremos trabajar. Esto es útil para carpetas compartidas de Windows via LAN.

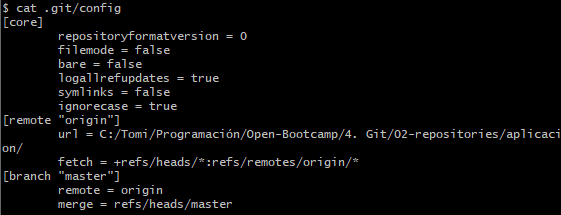
Para poder utilizarlo se debe clonar el repo bare en una carpeta vacia. Existe un truco para no tener que eliminar los archivos o moverlos de la carpeta en la que estamos trabajando.

Primero la renombramos, por ejejmplo como “nombre.original”, luego clonamos el repo utilizando la dirección dentro del disco (entre comillas si en los nombres de las carpetas hay espacios) y el nombre de la carpeta a donde vamos a clonarlo

* **mv <carpeta actual> <carpeta actual>.original**
* **git clone <dirección> <carpeta actual>**

Con el comando **ls -al** podemos ver lo que hemos traído del repo bare, que es distinto de lo que veíamos dentro del repo.

Esto lo podemos ver mejor usando el comando **cat**. Donde veremos que la url del supuesto repositorio remoto es la dirección de la carpeta bare.



Ahora solo quedaría traernos todo lo de la anterior carpeta que teníamos utilizando el comando **cp -prf**.

* **cp -prf ../<carpeta actual>.original/\* .**

Así solo quedaría añadir los cambios y commitearlos.

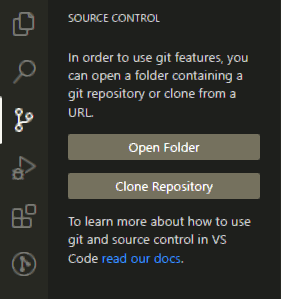
* **git add .**
* **git commit -a -m “primer commit”**

Ahora podríamos enviar los cambios locales al repo remoto bare con el comando **push**.

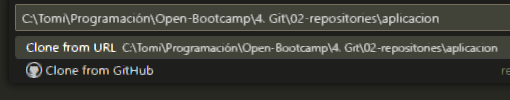
* **git push**

Y ya estarían subidos los cambios al repo, de esta manera si eliminamos la carpeta actual por cualquier razón, podríamos volver a tenerla tan solo clonandola desde el repo bare.

Ahora, todo esto puede controlarse desde el mismo VS Code, si vamos al menú de la izquierda queremos la opción de Source Control, en la que tendremos la opción de abrir una carpeta o clonar un repositorio.



Si damos en clonar un repositorio, nos abrirá un input donde debemos colocar una url a Github o una dirección de carpeta, en este caso pondremos la dirección de la carpeta del repo bare.



Luego, nos preguntará la carpeta en la cual queremos clonar el repositorio. Luego de esto, ya estaríamos listos para trabjar en el repo desde Visual Studio Code.