**Tutorial de GIT** – Cuando se es programador, una de las herramientas que se convierten en nuestras mejores amigas es **Git**. Se trata de un sistema de control de versiones creado por nada más y nada menos que **Linus Torvalds**, el creador del **Kernel Linux** y es la mejor manera de evitar catástrofes en nuestro código además de ser un excelente gestor del trabajo cuando se trabaja en equipo.

fuente: https://www.marindelafuente.com.ar/tutorial-de-git-guia-de-uso-de-git-2018/

**Git** principalmente se usa para código, incluso hay lugares en la web (Como **BitBucket** y **GitHub**) que se dedican a crearnos un *frontend* para la gestión de nuestro código controlado por Git; Sin embargo, no veo razón alguna por la cual Git no pueda ser usado por otras personas para otros ámbitos (diseño, arquitectura, qué se yo) en proyectos que requieran trabajar con más de una versión de un archivo y varias modificaciones. Dicho esto, aprendamos Git pues:

***NOTA:*** *Para éste tutorial usaré un sistema Linux (Ubuntu) como referencia, pero (exceptuando las instrucciones de instalación) las mismas instrucciones de uso de Git aplican para cualquier otra distro Linux, además de para MacOS o bien, gente que use CygWin en Windows  por ejemplo.*

**1) Instalando Git**

Para instalar Git en **Ubuntu Linux** por ejemplo, haríamos lo siguiente en Terminal:

sudo apt-get install git git-core

**2) ¿Primera vez? Configura tus Opciones**

Ahora tendremos que correr éstos comandos (Globales entre sistemas) para configurar nuestras opciones de Git:

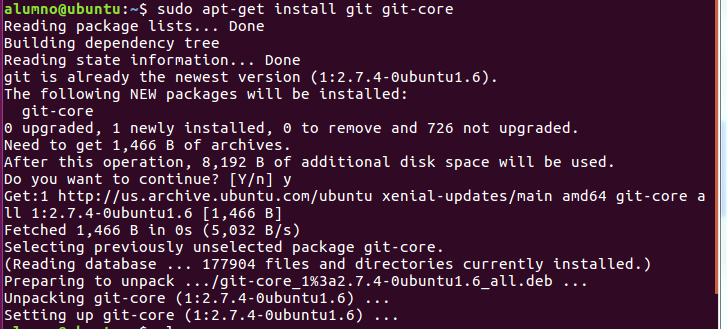
1. git config --global user.name "Tu Nombre"  
2. git config --global user.email tu-email@servicio.com  
3. git config --global push.default simple

*Como podrán notar, el nombre va entre comillas, el e-mail no.*

**3) Editor predeterminado**

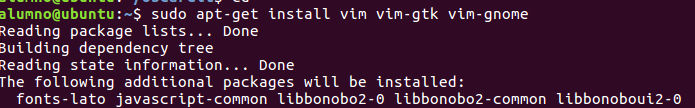
Cuando necesitemos añadir un mensaje a nuestros Commits y no lo hagamos directamente desde consola por ejemplo, necesitamos que un editor aparezca y nos recuerde/permita hacerlo, para configurar ésta opción hacemos:

git config --global core.editor "editor"



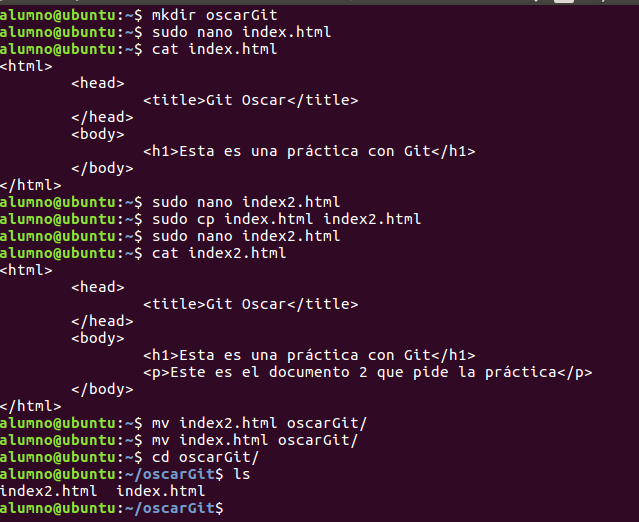
Algunas opciones a usar en lugar de **“editor”** son por ejemplo: **subl -w** (Para *SublimeText* en todas las plataformas) **mate -w** (Para *TextMate* en Mac) **gvim -f**  (Para *GVim* en Linux) o bien, **mvim -f** (Para *MacVim* en Mac); Yo personalmente uso **nano** para el editor nano (de consola) en Linux. Recuerda que debes seleccionar una opción según tu sistema operativo y asegurarte de que el editor está instalado… Para Linux por ejemplo usaremos *GVim* y lo instalaremos de la siguiente manera antes de establecer la opción (en Ubuntu):

sudo apt-get install vim vim-gtk vim-gnome



**4) Nuestro primer Repo!**

Ahora sí, es hora de empezar a trabajar con Git para conocer sus beneficios: Crea una **nueva carpeta** en algún lugar y añade unos **2 o 3 archivos de texto** (o HTML o de cualquier tipo de script o código que quieras) además de una imagen **.jpg o .png** por ejemplo, una **carpeta vacía** (con el nombre **“empty”**) y un **archivo vacío** con extensión **“.dumm”** (y de nombre “archivo”) jaja.En tu terminal haz **cd** hacia la ruta de la carpeta donde pusiste todo, luego inicializa un repositorio Git… Éstos dos pasos se hacen así:



cd ruta/a/carpeta  
git init



**5) Git Ignore**

Por defecto, Git siempre toma todo el contenido de la carpeta del proyecto donde se inicializó el repo y lo utiliza cuando trabajamos con el repositorio… El problema con ésto es que hay ciertos archivos que ciertamente no queremos en nuestro repositorio o es más, no queremos que se les tome en cuenta porque son constantemente cambiados. Éstos son aquellos como por ejemplo los “Logs” o también, el famoso “Thumbs.db” en Windows por solo mencionar algunos. Para evitar que éstos archivos se nos interpongan, tenemos que crear un archivo llamado **.gitignore** (obviamente oculto) en la raíz de la carpeta del proyecto y debe de tener el siguiente formato según nuestras necesidades:



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | # Ignorando grupos de archivos individuales por extensión    rutas/de/locación/\*.extensión    # Ignorando una Carpeta    ruta/a/carpeta/ignorada    # Ignorando un Archivo Individual    ruta/a/archivo.ext |

Si ésto de arriba no tuvo mucho sentido para ti, veamos un ejemplo: Por decir, supongamos que quiero evitar que Git cargue las imágenes que puse en mi carpeta de ejemplo que hice arriba, el archivo “.dumm” que me inventé y la carpeta vacía. Mi archivo ***.gitignore*** quedaría algo así:



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | # Ignorando JPG's y PNG's    /\*.jpg  /\*.jpeg  /\*.png    # Ignorando la Carpeta Vacía    /empty    # Ignorando un Archivo Individual    /archivo.dumm |

Si estás acostumbrado a trabajar con rutas de carpetas en Linux o MacOS por ejemplo, entonces éste archivo no te parecerá raro: es lo mismo. La única diferencia es que toma como Raíz a la carpeta del proyecto y de ahí uno selecciona los archivos que quiere ignorar. En el caso de querer ignorar en base a  las extensiones usamos el comodín **(\*)** para que ignore todos los archivos que sean de ésa extensión. Si te das cuenta, en ningún momento pasamos parámetros (Como -R) puesto que Git está muy bien configurado y hará su trabajo a la perfección de manera *recursiva*, ésto quiere decir que aplicará las reglas adecuadas en directorios y subdirectorios (además de en todos sus archivos) cuando se amerite según nuestras instrucciones del archivo **.gitignore**.

**6) Añadiendo archivos**

Lo siguiente que se hace es añadir los archivos a nuestro *árbol git*. Dentro de la terminal (estando en la ruta de la raíz de nuestro proyecto) hacemos lo siguiente:

git add .

Éso es simplemente para hacer que todos los archivos (nuevos y modificados) sean añadidos a una especie de “lista de espera” de Git. El mismo comando lo tenemos que correr al borrar y/o añadir archivos/carpetas para que git tome en cuenta las modificaciones en el proyecto.

**7) Haciendo válidos los cambios**

Ahora tenemos que realizar un *commit*, para hacer válidos los cambios en nuestro proyecto. Ésto se hace con:

git commit -m "Mensaje"

Lo que va después de **-m** es el mensaje que describe a éste commit, siempre hagan sus commits con un mensaje de preferencia. Si no quieren añadir mensaje, omitan el **-m**. *git commit* hace válidos los cambios de manera **Local** en nuestro árbol del proyecto.

***NOTAS:*** *Cabe destacar que puedes usar los comandos* ***git status*** *y* ***git log*** *para monitorizar tu proyecto con Git de cuando en cuando. Por otro lado, si haces cambios menores en tu proyecto (como renombrar archivos, editarlos etc) Puedes usar el siguiente comando para añadir y hacer el commit:*

git commit -a -m "Mensaje"

*Pero recuerda que si haces cambios importantes (Como añadir nuevos archivos o eliminar otros por ejemplo) debes de usar el comando de* ***add .*** *y luego el de* ***commit*** *como lo harías regularmente.*

**8) Git y su Utilidad**

Supongamos que por alguna razón eliminaste de manera forzada e irrecuperable (**rm -rf** jajajaja) algún archivo o carpeta de tu proyecto desde tu último commit. Bueno, pues siempre puedes volver al último *snapshot* de tu código (es decir, al código como estaba al momento de realizar el último commit registrado) con el comando:

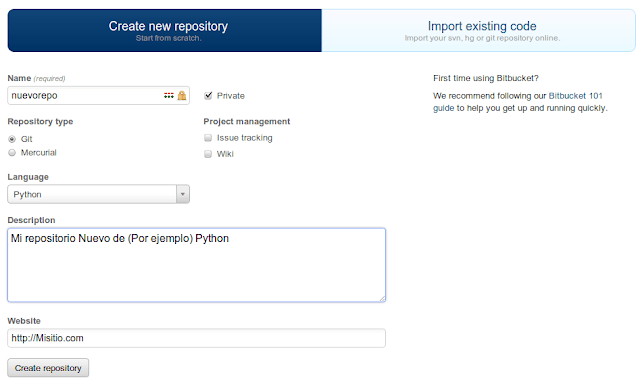
git checkout -f

El parámetro **-f** hace que git force a sobreescribir los cambios hechos por accidente en el proyecto y se restauren los archivos y carpetas de nuestro último *commit*.

**9) A la Nube!**

Todo ésto que hemos estado viendo es en el ámbito Local. Pero, ¿Qué pasa si estoy colaborando con varios programadores en un mismo proyecto? Qué tal si quiero tener un backup de mi código en alguna parte que no sea mi PC? Qué tal si quiero que más y más gente me conozca y tenga acceso a lo que hago? Aquí es donde entran en juego los servicios de almacenamiento de repositorios Git como podrían ser [**GitHub**](https://github.com/) y [**BitBucket**](https://bitbucket.org/) o bien, [**Gitorious**](https://gitorious.org/). Personalmente, prefiero **BitBucket** debido a que ofrece prácticamente las mismas funciones que GitHub y similares además de algunas ventajas que se agradecen, como el hecho de poder tener todos los repositorios públicos y privados que quieras completamente gratis. Es por ésto que en éste tutorial usaremos ése servicio, pero estás en libertad de irte con el que más te convenza.

Lo primero que haremos será sacar una cuenta de *BitBucket* en [**bitbucket.org**](http://bitbucket.org/), configuraremos nuestras preferencias y validaremos la cuenta. Más adelante, crearemos un nuevo repositorio PRIVADO que funcione con Git como control de versiones:

[](https://www.marindelafuente.com.ar/wp-content/uploads/2018/05/git-bit-bucket.png)

 Le ponemos un nombre, seleccionamos un lenguaje (la selección de lenguaje ahorita es indiferente, pero trata de seleccionarlo según el tipo de lenguaje más utilizado en tu proyecto) y de momento no marcamos ni Wiki ni Issue Tracking, pues no los vamos a usar. Añadimos una descripción y nuestra web si así lo queremos también.

**10) Git Push**

Una vez teniendo un lugar en la web que nos sirva de alojamiento para nuestro repo, tenemos que pasar nuestros cambios a la web, así como hacemos *commits* locales. Justo después de hacer un commit local, si uno quiere que ésos cambios pasen a la nube (usando bitbucket) hace:

git remote add origin https://usuario@bitbucket.org/usuario/repo.git

git push -u origin master

*Reemplazando “usuario” por tu nombre de usuario en bitbucket y “repo” por el nombre del repo hacia donde quieres enviar tu código.*

Y eso hará que Git sepa que tiene que subir la rama principal del proyecto (es decir todo el proyecto pues) a **BitBucket**. Para fines prácticos (no tener que introducir credenciales) uno usa integración con **SSH** pero eso está para explicarse en otra ocasión. También debo recalcar que el comando 1 se tiende a usar una sola vez cada que añadimos un nuevo origen remoto (por cada repo inicializado) según las instrucciones que se nos hayan dado en el mismo, y el 2 pasa a ser simplemente *git push* en la mayoría de sistemas tras haberlo usado como aquí se muestra la primera vez en cada repo inicializado.

**11) Branching**

Imaginemos a nuestro proyecto en Git como un árbol donde la raíz es el proyecto principal y de ahí crecen varias ramas. El branching en Git es justo lo que te imaginarás: Se trata de crear ramificaciones del proyecto en base a copias del código del proyecto original en las cuales podemos realizar modificaciones sin preocuparnos por comprometer el proyecto en sí. Ésto quiere decir que si yo tengo un proyecto en el cual se me ocurre añadir una nueva característica que quizá no sea lo suficientemente confiable a nivel estabilidad puedo hacerlo en una rama de mi proyecto y dejar el original intacto. Para crear ramas, usamos el siguiente comando:

git checkout -b NombredeRama

Donde **“NombredeRama”** es el nombre que le pondremos a nuestra rama que se creará entonces. Podemos usar el comando **git branch** para saber en qué rama estamos trabajando actualmente en el proyecto. Si queremos saltar de rama en rama, usamos el comando:

git checkout rama

Donde reemplazaremos **“rama”** (obviamente) por el nombre de la rama a la que queramos saltar.

**12) Merging**

Supongamos que creaste una rama nueva donde hiciste algunas modificaciones que resultaron ser una buena idea y quieres que el proyecto original (rama master) las integre. Para ésto sirve el *merging*… Si queremos combinar 2 ramas, nos moveremos a la rama donde queramos integrar el código nuevo (en el ejemplo la master) y haremos el merge con:

git merge Rama-a-Mezclar

Donde **“Rama-a-Mezclar”** es el nombre de la rama que mezclaremos con la rama a la que nos movimos. Si vamos a mezclar **“ramapatito”** con **“master”** por ejemplo entonces haríamos:

git checkout master  
git merge ramapatito

Y eso añadiría los cambios realizados en **“ramapatito”** a la rama principal del proyecto.

**13) Borrando Ramas**

Cuando queramos borrar una rama que ya haya sido mezclada, lo haremos con:

git branch -d Rama

*Donde “Rama” es obviamente el nombre de la rama…*

Y si queremos borrar una rama que no ha sido mezclada tendríamos que usar **-D** en lugar de **-d** como en el ejemplo de arriba. Ojo que ésto no elimina los archivos mezclados, sino simplemente la rama como contenedor, para eliminar los archivos mezclados (si al final no te gustó) harías un *rollback* como mostramos en el paso 8.

**PRÁCTICA 1 de GIT A REALIZAR en W7.**

**Crea en GITHUB y en local un repositorio llamado “calendario”.**

1. En cada línea, realiza un commit indicando una etiqueta “C1”, “C2” y una descripción del cambio que has realizado, y debes subir los cambios al remoto.
2. Añade los siguientes directorios, estaciones del año: PRIMAVERA, VERANO, OTOÑO, INVIERNO
3. Añade un nueva RAMA llamada “Con Fotos”
4. Añade en el directorio raíz, UN JPG O PNG (puedes sacarlo de google images) estaciones del año: PRIMAVERA.jpg, VERANO.jpg, OTOÑO.jpg, INVIERNO.jpg.
5. Vuelve a la rama “default”
6. Añade los siguientes meses: ENERO.txt, FEBRERO.txt, MARZO.txt, ABRIL.txt, MAYO.txt, JUNIO.txt.
7. Agrega una nueva rama “En inglés”
8. Añade los siguientes meses(ficheros) en raíz : JANNUARY.txt, FEBRUARY.txt, MARCH.txt, APRIL.txt, MAY.txt, JUNE.txt, JULY.txt, AUGUST.txt, SEPTEMBER.txt, OCTOBER.txt, NOVEMBER.txt, Y DECEMBER.txt
9. Vuelve a la primera rama “default”
10. Haz un merge con la rama “En inglés”
11. Añade el resto de los siguientes meses: JULIO.txt, AGOSTO.txt, SEPTIEMBRE.txt, OCTUBRE.txt, NOVIEMBRE.txt, Y DICIEMBRE.txt.
12. Añade la rama “OrdenadoEnCarpetas”: Cada txt y jpg debe ir en su directorio correspondiente.

Comenta el resultado final brevemente y si por el camino se han perdido algún archivo, ¿se pueden recuperar?

**PRÁCTICA 2 de GITHUB ONLINE.**

Realizad la práctica que está en : (10min aprox)

<https://guides.github.com/activities/hello-world/>