



## 1.6.- Contenido 6: Gestionar el código fuente de un proyecto utilizando GIT para mantener un repositorio de versiones

---

### Objetivo de la jornada

---

1. Realiza operaciones de navegación de directorios utilizando los comandos básicos del terminal.
2. Aplica procedimiento de versionamiento de un proyecto utilizando GIT y el terminal para mantener un repositorio de versiones.
3. Aplica procedimiento de subida del código versionado utilizando una conexión SSH para la mantención de un repositorio remoto Github.

### 1.6.1.- Utilizando el terminal

La línea de comando de Linux es una interfaz de texto para interactuar con el sistema operativo, que también es llamada shell, terminal, consola, prompt u otros nombres, puede parecer complejo y confuso de usar en una primera aproximación. Sin embargo, los usuarios de los años 80s e incluso 90s debían utilizarla como algo rutinario para manejar sus computadores personales. Desde línea de comando se creaban directorios (conocidas actualmente como carpetas), archivos, se realizaban búsquedas y se iniciaban las distintas aplicaciones, tales como procesadores de texto, planillas de cálculo, juegos y hasta incluso se navegaba por internet desde ella. Los tiempos han cambiado, la línea de comando sigue existiendo, con mucha más popularidad en los sistemas basados en GNU/Linux, pero incluso Windows mantiene su línea de comandos heredada de su sistema operativo DOS. La línea de comandos, a pesar de ser más engorrosa de usar, tiene muchas facilidades que a veces, desde ambiente gráfico, es difícil encontrar equivalentes o hay que instalar programas adicionales para emular su funcionalidad. Una vez familiarizado con ella y memorizando algunos comandos esenciales, se puede disfrutar del poder de la línea de comandos.

En este caso, nos centraremos en los comandos relacionados con el sistema operativo GNU/Linux, que es el que recomendamos para uso en ambiente de desarrollo. Los ejemplos que se muestran han sido realizados en Ubuntu 16.04 LTS. Como referencia para los usuarios de sistema operativo Microsoft, también mostraremos una equivalencia para algunos comandos entre sistema GNU/Linux y Windows.



## ABRIR UN TERMINAL

GNU/Linux operando con ambiente gráfico, por ejemplo la distribución Ubuntu u otra, comúnmente permiten abrir un terminal con la combinación Ctrl + Alt + t. Se mostrará algo como lo que se muestra a continuación:

```
root@fourier94-Aspire-4830T: ~/Desktop
root@fourier94-Aspire-4830T:~/Desktop#
```

Donde se observan los siguientes componentes:

- **root**: Usuario actual utilizando el terminal.
- **fourier94-Aspire-4830T**: Nombre del computador o host.
- **~/Desktop**: Ruta actual.

Luego se observa el cursor, donde el usuario puede ingresar los comandos que quiera dar al sistema. La estructura de la sintaxis usada para ingresar comandos se indica a continuación, para el comando **ls** que despliega el listado de archivos y directorios de una ruta que se indique, con ciertas opciones:



La opción **-la** instruye que el resultado sea entregado como listado (**-l**), y que no se ignoren archivos o directorios ocultos (**-a**). **/home/user** indica el directorio que desea ser explorado en este caso.

Cada comando posee opciones posibles y argumentos.

Comando GNU/Linux	Descripción	Equivalente DOS
ls	muestra un listado del contenido de un directorio	dir
cd	cambia de directorio	cd



<code>cd ..</code>	cambia al directorio anterior	<code>cd..</code>
<code>mkdir</code>	crea un nuevo directorio	<code>md</code>
<code>rmdir</code>	elimina un directorio	<code>deltree</code>
<code>cp</code>	copia un archivo	<code>copy</code> , <code>xcopy</code>
<code>mv</code>	mueve un archivo	<code>move</code>
<code>rm</code>	elimina un archivo	<code>del</code>
<code>passwd</code>	cambia la contraseña del usuario	
<code>cat</code>	muestra el contenido del archivo	<code>Type</code>
<code>more</code>	muestra el contenido del archivo con pausas	<code>more</code>
<code>man</code>	ayuda para el comando requerido	<code>help</code>
<code>exit</code>	termina la sesión	<code>exit</code>

A su vez el comando **man** permite explorar la sintaxis soportada por cada comando, sus opciones posibles y argumentos. Por ejemplo, si hacemos **man ls** podremos ver las instrucciones para uso del comando **ls**, de la siguiente forma:

```
LS(1)                                User Commands                                LS(1)
NAME
  ls - list directory contents
SYNOPSIS
  ls [OPTION]... [FILE]...
DESCRIPTION
  List information about the FILES (the current directory by default). Sort entries alpha-
  betically if none of -cftuvSUX nor --sort is specified.

  Mandatory arguments to long options are mandatory for short options too.

  -a, --all
      do not ignore entries starting with .
  -A, --almost-all
      do not list implied . and ..
  --author
      with -l, print the author of each file
  -b, --escape
      print C-style escapes for nongraphic characters
  --block-size=SIZE
      scale sizes by SIZE before printing them; e.g., '--block-size=M' prints sizes in
      units of 1,048,576 bytes; see SIZE format below
  -B, --ignore-backups
      do not list implied entries ending with ~
```

## USO DE COMANDOS `cd`, `ls`, `mkdir`, `touch`, `cp`, `mv`

Ilustraremos la forma de desenvolverse en un terminal con comandos simples en una situación cotidiana donde queremos revisar algunos directorios, crear y mover archivos.

Abrimos un terminal y ejecutamos los siguientes comandos:

- `cd Documents/awakelab/` (Cambiar a directorio **Documents/awakelab/**).
- `ls -la` (Mostrar lista de contenidos del directorio actual).
- `mkdir proyecto/` (Crear directorio **proyecto/**).
- `cd proyecto/` (Cambiar a directorio **proyecto/**).



- `mkdir css/ js/ img/` (Crear directorios **css/**, **js/** e **img/**).
- `touch index.html` (Crear archivo de texto plano **index.html** vacío).
- `ls -la` (Mostrar lista de contenidos de directorio actual).

En el terminal veríamos los resultados de esta secuencia de comandos de la siguiente forma:

```
fourier94@fourier94-Aspire-4830T:~$ cd Documents/awakelab/
fourier94@fourier94-Aspire-4830T:~/Documents/awakelab$ ls -la
total 16
drwxrwxr-x  4 fourier94 fourier94 4096 jul 10 18:30 .
drwxr-xr-x 33 fourier94 fourier94 4096 jun 23 04:07 ..
drwxrwxr-x  4 fourier94 fourier94 4096 jun 21 06:02 full_stack_python
drwxrwxr-x  7 fourier94 fourier94 4096 jul  4 14:28 otros
fourier94@fourier94-Aspire-4830T:~/Documents/awakelab$ mkdir proyecto/
fourier94@fourier94-Aspire-4830T:~/Documents/awakelab$ cd proyecto/
fourier94@fourier94-Aspire-4830T:~/.../awakelab/proyecto$ mkdir css/ js/ img/
fourier94@fourier94-Aspire-4830T:~/.../awakelab/proyecto$ touch index.html
fourier94@fourier94-Aspire-4830T:~/.../awakelab/proyecto$ ls -la
total 20
drwxrwxr-x  5 fourier94 fourier94 4096 jul 10 18:31 .
drwxrwxr-x  5 fourier94 fourier94 4096 jul 10 18:31 ..
drwxrwxr-x  2 fourier94 fourier94 4096 jul 10 18:31 css
drwxrwxr-x  2 fourier94 fourier94 4096 jul 10 18:31 img
-rw-rw-r--  1 fourier94 fourier94   0 jul 10 18:31 index.html
drwxrwxr-x  2 fourier94 fourier94 4096 jul 10 18:31 js
fourier94@fourier94-Aspire-4830T:~/.../awakelab/proyecto$
```

Con esto hemos pretendido crear una estructura de directorios **css/**, **img/**, **js/** y archivo **index.html** de un proyecto web como los que hemos estudiado hasta el momento.

Luego podríamos querer incorporar imágenes dentro del directorio **img/**. Para efectos de nuestro ejemplo, los archivos tendrán los siguientes nombres (Incluyendo la ruta absoluta de éstos):

[/home/fourier94/Pictures/awakelab/img1\\_awakelab.jpg](#)

[/home/fourier94/Pictures/awakelab/img2\\_awakelab.jpg](#)

[/home/fourier94/Pictures/awakelab/logo\\_awakelab.jpg](#)

Hemos introducido aquí el concepto de ruta absoluta. La ubicación de un archivo puede ser especificadas con su **ruta absoluta** o su **ruta relativa**.

### ***Rutas relativas absolutas y relativas***

Las **rutas absolutas** señalan la ubicación de un archivo o directorio indicando todos los directorios a los que deberíamos entrar sucesivamente para encontrarlo, tomando como origen el directorio raíz del sistema de archivo, que se identifica con el símbolo **"/**". Es usual utilizar para mayor comodidad y simplificación en rutas absolutas el alias **"~"** que reemplaza la necesidad de escribir en nuestro caso la primera parte (**/home/fourier94**) de las rutas mostradas más arriba. Por lo tanto,



continuando con nuestro ejemplo, utilizaremos indistintamente con los anteriores, los siguientes nombres de archivos:

[~/Pictures/awakelab/img1\\_awakelab.jpg](#)

[~/Pictures/awakelab/img2\\_awakelab.jpg](#)

[~/Pictures/awakelab/logo\\_awakelab.png](#)

Las **rutas relativas** señalan la ubicación de un archivo o directorio indicando todas las acciones de cambio de directorio que sería necesario realizar para llegar al archivo o directorio que se desea referenciar. En necesario conocer las siguientes formas de referenciar un cambio de directorio:

- `../` : Se utiliza para referenciar un directorio padre. Es posible usarlo de manera anidada para retroceder a directorios de jerarquía superior.
- `.` : Se utiliza para referenciar el directorio actual.

Volviendo a nuestro ejemplo, necesitamos incorporar los tres archivos mencionados al proyecto en el directorio **img/**. A modo ilustrativo, utilizaremos tres formas distintas para realizarlo:

- Haciendo una copia (Comando **cp**) del primer archivo usando rutas absolutas.
- Haciendo una copia (Comando **cp**) del segundo archivo usando rutas relativas.
- Moviendo (Comando **mv**) el archivo desde su ubicación original a la nueva usando rutas relativas.

Los comandos respectivos serán:

- ```
cp \
/home/fourier94/Pictures/awakelab/img1_awakelab.jpg \
/home/fourier94/Documents/awakelab/proyecto/img/
```

El símbolo `\` permite escribir comandos en el terminal con múltiples líneas, que permiten una mejor visualización de comandos de grandes logitudes)
- ```
cp \
../../Pictures/awakelab/img2_awakelab.jpg \
./img/
```
- ```
mv \
../../Pictures/awakelab/logo_awakelab.png \
./img/
```



Luego entramos al directorio **img/** para verificar que los archivos estén presentes.

- `cd img/` (Cambiar ubicación actual a **img/**).
- `ls -la` (Mostrar lista de contenidos del directorio actual).

Todo lo anterior en el terminal se verá de la siguiente forma:

```
fourier94@fourier94-Aspire-4830T:~/.../awakelab/proyecto$ cp \
> /home/fourier94/Pictures/awakelab/img1_awakelab.jpg \
> /home/fourier94/Documents/awakelab/proyecto/img/
fourier94@fourier94-Aspire-4830T:~/.../awakelab/proyecto$ cp \
> ../../Pictures/awakelab/img2_awakelab.jpg \
> ./img/
fourier94@fourier94-Aspire-4830T:~/.../awakelab/proyecto$ mv \
> ../../Pictures/awakelab/logo_awakelab.png \
> ./img/
fourier94@fourier94-Aspire-4830T:~/.../awakelab/proyecto$ cd img/
fourier94@fourier94-Aspire-4830T:~/.../proyecto/img$ ls -la
total 328
drwxrwxr-x 2 fourier94 fourier94 4096 jul 10 19:34 .
drwxrwxr-x 5 fourier94 fourier94 4096 jul 10 18:31 ..
-rw-rw-r-- 1 fourier94 fourier94 297719 jul 10 19:33 img1_awakelab.jpg
-rw-rw-r-- 1 fourier94 fourier94 19004 jul 10 19:34 img2_awakelab.jpg
-rw-rw-r-- 1 fourier94 fourier94 5740 jul 10 18:39 logo_awakelab.png
fourier94@fourier94-Aspire-4830T:~/.../proyecto/img$
```

Existe una infinidad de comandos para muchos objetivos distintos, unos más avanzados que otros. Es posible entrar a servidores remotos a través de comandos como **ssh**, o copiar archivos entre máquinas remotas con comandos como **scp**. Las necesidades enfrentadas en el camino de formarse como desarrollador, de manera autodidacta o formal, agregan incrementalmente más comandos a la batería de herramientas de las que se dispone para el trabajo diario.

Existen buenas referencias en la **www** para iniciarse en este tópico, tales como:

- <https://ubuntu.com/tutorials/command-line-for-beginners>
- <https://www.debian.org/doc/manuals/debian-reference/>
- [T William E. Shotts, Jr., The Linux Command Line, 2nd ed., 2013](#)

## 1.6.2.- Fundamentos de GIT

### 1.6.2.1.- ¿Por qué es importante un repositorio?

Git es un sistema de control de versiones distribuido, gratuito, de código abierto y eficiente<sup>[1]</sup>, y se complementa con la plataforma GitHub para alojar repositorios de código. Entre las ventajas<sup>[2]</sup> que posee es posible destacar:



- **Versionamiento de código:** cada cambio en un proyecto se puede almacenar en versiones diferentes, lo que permite poder ir “hacia atrás” en caso que se desee revisar casos previos de un módulo o plugin. A diferencia
- **Aprender y probar cambios:** existen muchos proyectos que son públicos en GitHub, pudiendo experimentar con cambios sobre éstos, sin afectar el código original. Este último proceso es conocido como **fork**.
- **Contribuir:** Una vez que se han realizado cambios y testeados exitosamente, se pueden enviar al propietario original a modo de aporte o mejora. Este proceso se llama **pull request**, y permite al encargado del repositorio validar la contribución, proponer mejoras, rechazar el cambio, o simplemente aceptarla para fusionarla con el proyecto original; este último escenario es conocido como “**merge**”.
- **Trabajo en equipo:** permite un excelente trabajo en equipo para el desarrollo de proyectos de diversa envergadura, incluso de alto impacto.
- **Visor de código:** GitHub tiene incorporada una herramienta que permite revisar el código almacenado en el repositorio, pudiendo comparar diversas versiones y volver a versiones anteriores de ser necesario.
- **Precio:** GitHub es completamente gratis e ilimitado para proyectos públicos, esto quiere decir que el código se podrá visualizar a través de la red. Se puede tener proyectos privados, pero de no más de tres colaboradores.

#### 1.6.2.2.- Instalación, configuración y comandos básicos.

La instalación de Git en un ambiente local es sencillo. Solo debe ir a la sección de descargas del sitio oficial<sup>[3]</sup>, y seleccionar el sistema operativo de su equipo. Esto último lo detecta el navegador, y por defecto se muestra la última versión disponible del instalador.

Debe considerar que lo que se instala en este punto es solo la funcionalidad que permite administrar repositorios a través de comandos en un terminal, esto independiente del sistema operativo en uso. Si se desea tener una mejor experiencia a través de un entorno más ameno, existen programas que ayudan bastante a los desarrolladores, además de la integración que se puede hacer entre Git y diversos IDEs.

Lo primero que se debe hacer posterior a la instalación es la creación de un repositorio local. Para ello, siga los siguientes pasos:

- Cree un directorio para un proyecto
- En una consola de comandos, acceda al directorio
- Escriba el comando **git init**; con esto se inicializará un repositorio vacío.





- Ahora ingresa el comando **git add [ARCHIVO]**, lo cual agregará al listado de documentos del proyecto el documento de nombre **[ARCHIVO]**. Esto debe hacerlo por cada documento del proyecto, o bien seleccionarlos todos.
- Finalmente, para aplicar los cambios debes ingresar el comando **git commit**; el programa solicitará un texto con el detalle de los cambios aplicados. De no agregarse, no se podrá hacer el proceso.

A modo de pruebas, se recomienda crear un pequeño proyecto de una página web con un archivo CSS en un directorio independiente. La idea es que se puedan ir haciendo cambios y agregando nuevos documentos al proyecto.

### 1.6.2.3.- Commit y restauración de archivos.

Tal como se indicó previamente, una de las características interesantes de Git es la posibilidad de volver a versiones anteriores de un archivo. Para hacerlo puede hacer uso de los comandos que se muestran a continuación:

- Realiza un cambio en el sitio del proyecto, idealmente el título y agregando un elemento nuevo.
- En un terminal o consola de comandos, abre el directorio del proyecto
- Ingresa el comando **git add [ARCHIVO]**, con lo cual se agregará el documento de **[ARCHIVO]** al listado de elementos para actualizar.
- Una vez realizado lo anterior, ingresa el comando **git commit**; al igual que en el caso anterior, se solicitará al usuario el detalle de las modificaciones realizadas.
- Finalmente, si el proceso se ejecutó correctamente informará la cantidad de elementos actualizados, incorporados y eliminados.
- Repite este proceso para otro documento del proyecto, y ten cuidado de hacer el “commit” final para que los cambios se vean reflejados.

Para restaurar el proyecto en primer lugar se debe revisar el registro de cambios del proyecto, lo cual se puede hacer a través del comando **git log**. Se desplegarán todas las versiones existentes desde la más reciente a la más antigua. Por cada versión almacenada se despliega un código alfanumérico llamado “hash” seguido de la palabra “**commit**”. Una vez conocido este identificador, se debe ingresar el comando **git checkout [IDENTIFICADOR]**, donde **[IDENTIFICADOR]** es el código obtenido del hash; ten en cuenta que no es necesario agregar el hash, con los cuatro primeros es suficiente, siempre y cuando no exista redundancia en los identificadores. Lo que hace esta acción en lo concreto es mover la cabecera o puntero HEAD a otra versión





del proyecto, y los cambios que se realicen a futuro serán en base a dicho caso. No debes olvidar que las otras versiones seguirán co-existiendo en el repositorio.

#### 1.6.2.4.- Cambiar nombre de archivo.

Renombrar un archivo en un proyecto de Git puede parecer trivial si creemos que se logra cambiando directamente el nombre del archivo; en realidad no es así. Si deseas lograr este objetivo, debes realizar los siguientes pasos:

- Por medio de una consola de comandos accede a la carpeta en la que está el proyecto.
- Usa un comando para cambiar el nombre el archivo: **git mv nombre\_original nombre\_final**.
- Con el comando **git status** puedes comprobar que se detectó el cambio de nombre.
- Para confirmar el cambio debes realizar el comando **git commit**. Antes de realizar la tarea pedirá ingresar un texto de forma obligatoria.
- Con el comando **git log** se pueden ver los cambios realizados. Deberá aparecer el último cambio incorporado.

#### 1.6.2.5.- Ignorar archivos

Cuando se hace una actualización sobre un repositorio Git, previo a esa acción se debe agregar cada archivo usando el comando “**git add [ARCHIVO]**”, y después “**git commit**”. Si se usa la opción “**git add .**” se subirían a la nueva versión todos los archivos que han sido creados o modificados sin distinción; por lo mismo, se hace necesario contar con un método que permita filtrar archivos que no se deseen agregar al repositorio.

Lo anterior se puede lograr creando y/o editando el archivo **.gitignore**, el cual en cada línea deberá contener nombres de archivos o patrones de nombres que deberá ignorar. Generalmente se ubica en la raíz del proyecto, sin embargo pueden existir múltiples archivos de este tipo, siendo los patrones o expresiones indicadas relativas a la ubicación del archivo. En este ejemplo se ignorarán todos los archivos con extensión **.log**, excepto el archivo **example.log**.

```
*.log
!example.log
```



Es posible asimismo bloquear un directorio completo; considerar que, si se bloquea un directorio, no se pueden desbloquear por separado sus archivos.

```
/logs
```

Además es posible agregar comentarios en un archivo de este tipo. En el ejemplo siguiente se ignorará el archivo `.DS_Store`, propio del sistema operativo Mac y que, para efecto de compartir proyectos, puede resultar molesto.

```
#Archivos macOS  
.DS_Store
```

#### 1.6.2.6.- Ramas, uniones, conflictos y tags.

##### RAMAS (BRANCHES)

Una ramificación o simplemente “rama”, es crear una copia de la rama principal con el fin de realizar pruebas antes de aplicar los cambios en un ambiente de producción. En el caso de Git, un aspecto a destacar es que la forma en que maneja las ramificaciones es muy rápida, casi instantáneo.

En cada confirmación de cambios o commit, Git almacena una instantánea del trabajo realizado, junto a los metadatos del autor y el mensaje explicativo del caso, además de unos apuntadores a las confirmaciones (commit) que sean padres directos (uno si es confirmación “normal”, o más de uno si hay una fusión o “merge”).

Cuando se crea una rama nueva se crea un nuevo apuntador que puede moverse libremente por las versiones anteriores a ella y las que provengan de la misma. Imagine que ha realizado una confirmación, y desea crear una versión alterna que le permita probar cambios, sin pasar a llevar la rama “master”, que corresponde al camino principal. Para crear una rama debe ingresar en consola el comando que se indica a continuación.

```
$ git branch testing
```



La estructura debiera ser similar a la que se indica en la figura a continuación. Como se puede ver cada **commit** se identifica con un hash o identificador.

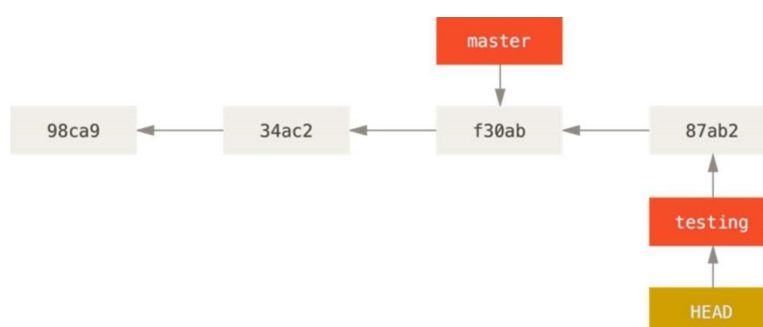


*Proyecto con dos ramas*

Es necesario recordar en este punto que usando el comando **git log** se puede revisar el listado de versiones existentes. Si se desea apuntar la cabecera ahora a otra versión, se puede usar el siguiente comando:

```
$ git checkout testing
```

Si se crea una nueva versión a través de una confirmación o **commit**, la rama seguirá desde el punto en que se quedó la última vez.

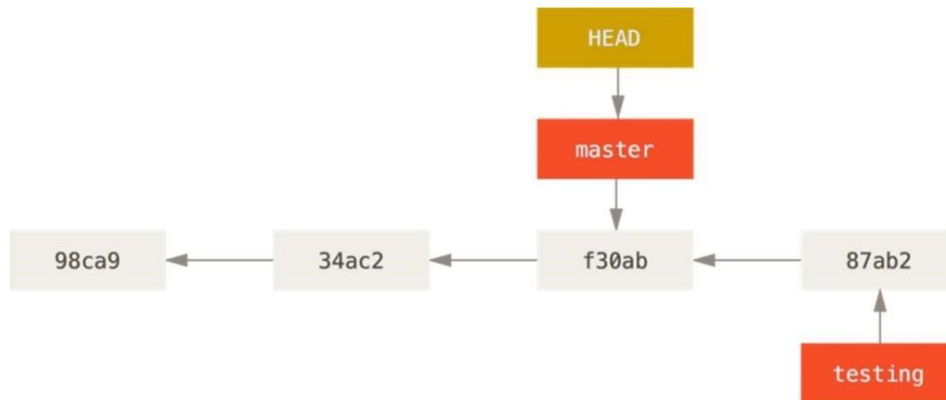


*Avance en confirmación de cambios desde rama alternativa*

Es posible que en el proceso se desee retomar la rama original. A partir de esta versión se creará una nueva división. Para ello, lo primero es realizar apuntar la cabecera a la rama inicial.

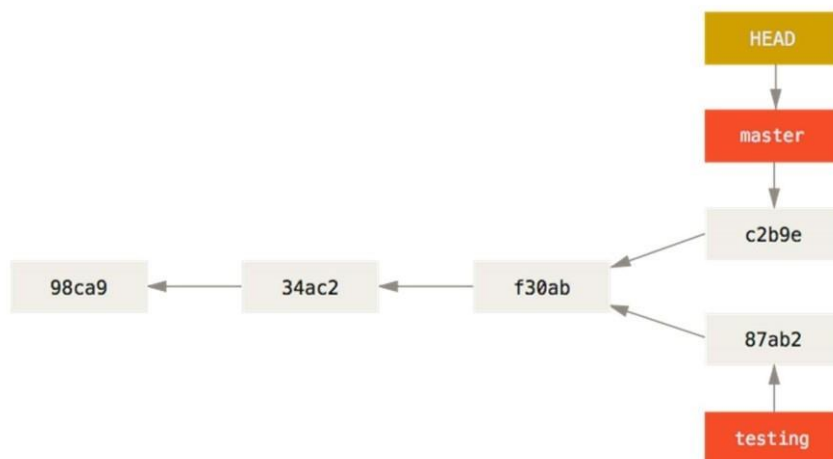


```
$ git checkout master
```



*Cabecera apuntando a rama principal del proyecto*

Una vez que se hayan realizado los cambios pertinentes, al invocar al comando `commit` se creará un nuevo punto de ramificación, pudiendo retornar a la rama “testing” en cualquier momento.



*Creación de una nueva versión desde la rama principal*

### UNIONES (MERGE)

La herramienta **git merge** se utiliza para fusionar una o más ramas dentro de aquella que se encuentre activa; posterior a ello avanzará la rama actual al resultado de la



fusión. Si se desea fusionar una rama específica, debe indicar la opción **git merge [branch]**, donde **[branch]** es el nombre de la rama individual que se desea combinar.

Una fusión aplastada o “**squashed merge**”, implicará que Git fusiona el trabajo, pero finge como si fuera simplemente un nuevo **commit** sin registrar la historia de la rama que se está fusionando.

En resumen, los pasos que se deben seguir en la creación de una rama son los siguientes:

1. Crear una rama basada en otra
2. Crear cambios en la nueva rama
3. Guardar los cambios
4. Volver a la rama principal
5. Fusionar los cambios

Lo anterior es similar a ejecutar los siguientes comandos (la nueva rama se llamará feature):

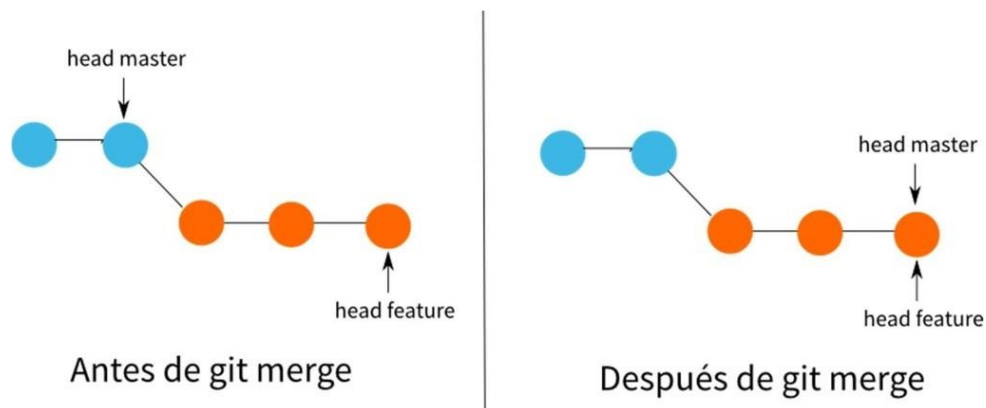
1. **git checkout -b feature**
2. Realizar los cambios
3. **git commit -m “cambios realizados sobre feature”**
4. **git checkout master**
5. **git merge feature**

Una vez que se fusiona una rama, ésta no desaparece, por lo cual como acción adicional opcional se podría eliminar. Lo anterior se logra usando el comando:

```
$ git branch -d feature
```

Después que una fusión se hace correctamente, se pueden dar dos escenarios posibles: una fusión de avance rápido (fast-forward merge) o una fusión a tres bandas (3-way merge).

El primero de ellos se aplica cuando al momento de realizar una fusión no se ha aplicado ningún commit luego de crear la rama nueva (en este caso **feature**), es decir, que la cabecera de la rama **master** es el antepasado de **feature**. Por tanto no es necesario hacer un nuevo commit para agregar los cambios de la nueva rama, sino que la cabecera apuntará a esta última. En la imagen siguiente se muestra la idea que esto conlleva.

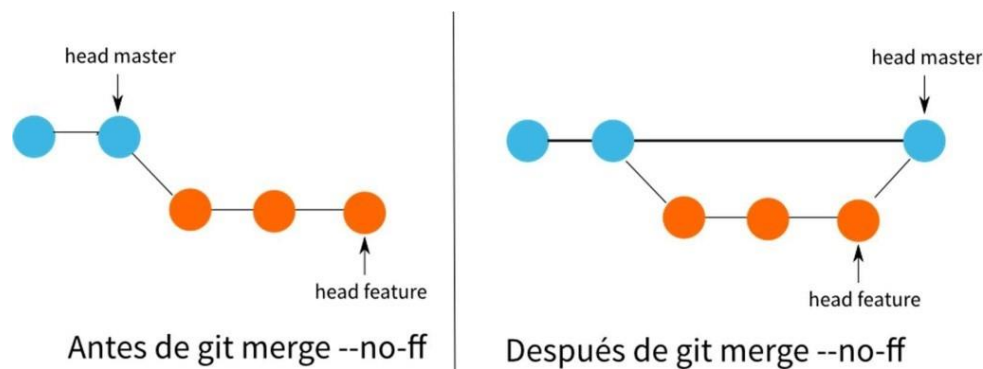


### *Fusión de avance rápido o fast-forward merge*

Sin embargo, si se desea hacer un commit para documentar o simplemente dejar constancia que se hizo una fusión de otra rama en “master”, es recomendable usar la opción `--no-ff`. En comando esto sería:

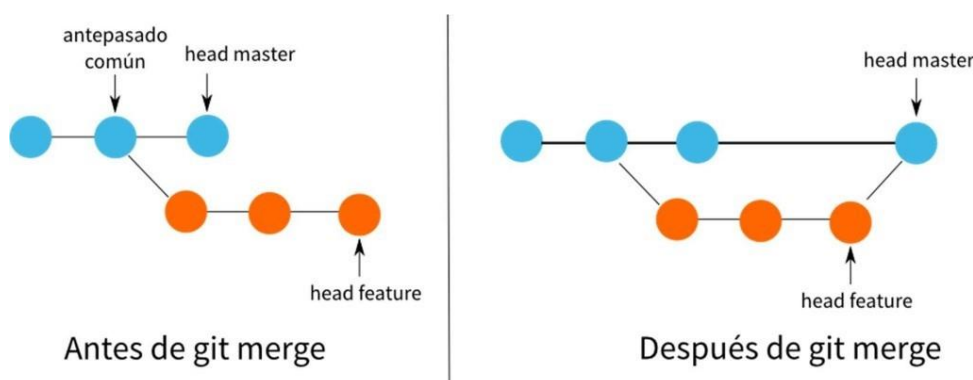
```
$ git merge --no-ff feature
```

La rama quedaría entonces de la siguiente manera.



### *Fusión sin avance rápido*

En otro caso, si la rama “master” ha cambiado después de haber creado la rama **feature** ya no es posible un **fast-forward merge**, debido a que el commit de la rama donde actualmente se está (master) no es un antepasado directo de la rama a fusionar (**feature**), por tanto git hace una fusión a tres bandas, es decir, que genera un commit para fusionar las dos ramas, tomando en cuenta el HEAD de cada una de ellas y el antepasado común de las dos.



*Fusión a tres bandas (3-way merge)*

### Resolver conflictos de fusión

Algunas veces la unión de dos ramas no resulta tan bien, sino que ocurre un conflicto, esto cuando el commit de la rama a fusionar y el de la rama actual modifican la misma parte en un archivo en particular, y git no puede decidir cuál versión elegir, y avisa que debe ser resuelto. Lo anterior se explicará a través de un ejemplo.

Supongamos que un directorio se tiene un archivo index.html con el siguiente contenido:

```
1 <html>
2   <head>
3     <title>Titulo</title>
4   </head>
5   <body>
6     <p>Contenido de la web</p>
7   </body>
8 </html>
9
```

Se inicializará en el mismo directorio del sitio un repositorio, llamando al comando **git init**, y luego se hará el primer commit a través del comando **git add index.html** y luego **git commit -m "commit inicial"**.

Se creará una nueva rama, con objeto de agregar nuevo contenido. Para ello use el comando **git checkout -b contenido**, lo que permitirá al mismo tiempo posicionarse sobre la nueva rama y hacer los cambios pertinentes. Modifique entonces el contenido de la etiqueta **<title>** del sitio ejemplo; una vez realizado esto, se realiza un commit a través del comando **git commit -a -m "cambios en el titulo"**, y se mueve la cabecera a la rama original por medio del comando **git checkout master**.





Recordemos que en este punto se ha vuelto a la bifurcación original, y que la versión creada en la rama **contenido** sigue intacta. El siguiente paso es crear otra modificación en la etiqueta **<title>** del archivo **index.html**, y aplicar los cambios a través del comando **git commit -a -m "tercera versión modificada"**.

Cuando se intente hacer una fusión o "merge", no se podrá hacer e indicará que no continuará el proceso hasta que se solucione dicha situación. Es importante considerar que Git no entrega ayuda indicando que archivo tiene el conflicto, solo abriéndolo se sabrá que cambios son de una rama y cuales de otra.

```
$ git reset --merge ORIG_HEAD
```

```
1  <html>
2    <head>
3      <title>Titulo</title>
4    </head>
5    <body>
6      <<<<<< HEAD (Current Change)
7      <p>Contenido de la web segunda version cambiada</p>
8      =====
9      <p>Contenido de la web modificado</p>
10     >>>>>> contenido (Incoming Change)
11     </body>
12 </html>
13
```

### *Resolución de conflictos en Git usando Visual Studio Code*

En el anterior caso se debe elegir entre lo que está entre **<<<<<< HEAD y =====**, que es contenido que se posee en la rama donde se está haciendo la fusión (master) o entre **===== y >>>>>>** contenido donde están los cambios hechos en la rama que se desea unir (contenido). Una vez que se arregle el archivo, se guarda, se agrega y se hace **commit**. En la imagen ejemplo que se muestra antes, se puede apreciar en distintos colores ambas versiones que generan el conflicto; la herramienta que se usa en el ejemplo es Visual Studio Code, y una de sus ventajas en este proceso es que permite seleccionar la sección del código que se aplicará definitivamente, haciendo más sencillo el proceso de resolución.

Para deshacer una fusión, se puede usar el comando **git reset**; se indica un ejemplo más adelante.



```
$ git reset --merge ORIG_HEAD
```

## ETIQUETAS (TAGS)

Dada la naturaleza de versionamiento, Git tiene la posibilidad de etiquetar puntos específicos del historial como importantes. Esta funcionalidad se usa típicamente para marcar versiones de lanzamiento (v1.0, por ejemplo). Es posible, entonces, listar las etiquetas disponibles, crear nuevas etiquetas y aplicar distintos tipos de ellas.

Para listar las etiquetas se usa el comando **git tag**.

```
$ git tag
```

Este comando lista las etiquetas en orden alfabético; el orden en el que aparecen no tiene mayor importancia.

Es posible también buscar etiquetas con un patrón particular. Solo por citar un caso, el repositorio del código fuente de Git, contiene más de 500 etiquetas. Si sólo interesa ver la serie 1.8.5, se puede ejecutar el comando que se indica en la siguiente ilustración.

```
$ git tag -l 'v1.8.5*'
```

Git utiliza dos tipos principales de etiquetas: ligeras y anotadas. Una etiqueta ligera es muy parecida a una rama que no cambia; dicho de otro modo, es un puntero a un **commit** específico.

Sin embargo, las etiquetas anotadas se guardan en la base de datos de Git como objetos enteros. Poseen las siguientes características:

- Tienen un **checksum**: llamado también “suma de comprobación”, es el resultado de la ejecución de un algoritmo dentro de un archivo único, función denominada **Cryptographic hash function**.
- Contienen el nombre del etiquetador, correo electrónico y fecha
- Tienen un mensaje asociado



- Pueden ser firmadas y verificadas con GNU Privacy Guard (GPG)

Normalmente se recomienda crear etiquetas anotadas, de manera que se tenga toda la información antes indicada; pero si se desea una etiqueta temporal o por alguna razón no existe interés sobre esa información, entonces es posible usar las etiquetas ligeras.

A continuación se indican diversos comandos que se pueden ejecutar gracias a etiquetas.

```
$ git tag -a v1.4 -m 'acá va la versión 1.4'
```

*Crear una etiqueta anotada*

```
$ git show v1.4
```

*Ver detalles de una versión*

```
$ git tag v1.4-v1
```

*Crear una etiqueta ligera*

```
$ git checkout -b version1p5 v1.5  
Switched to a new branch 'version1p5'
```

*Generar una rama a partir de una etiqueta*

```
$ git tag -a v1.5 [IDENTIFICADOR]
```

*Etiquetado tardío*

#### 1.6.2.7.- Stash y Rebase.

##### STASH

Es un comando que “congela” el estado en el que se encuentra el proyecto en un momento determinado, con todos los cambios que se tienen en estado "sin comitear", y lo guarda en una pila provisional, brindando la posibilidad de poder recuperarlo más adelante. Siguiendo con el ejemplo anterior, lo que se debería hacer es guardar los cambios que se han hecho a través del comando **git stash**, esto deja el “**working tree**” limpio y permite cambiar de rama sin problema. Una vez



solucionado el bug, se debe volver a la rama original, y se recuperarían los cambios con **git stash apply**.

## REBASE

En Git existen dos formas que permiten unir ramas: **git merge** y **git rebase**. La forma más conocida es **git merge**, la cual realiza una fusión a tres bandas entre las dos últimas instantáneas de cada rama y el ancestro común a ambas, creando un nuevo **commit** con los cambios mezclados.

**Git rebase** básicamente lo que hace es recopilar uno a uno los cambios confirmados en una rama, y re aplicarlos sobre otra. Utilizar **rebase** puede ayudar a evitar conflictos, siempre que se aplique sobre **commits** que están en local y no han sido subidos a ningún repositorio remoto. Si no tienen cuidado con esto último y algún compañero utiliza cambios afectados, seguro que tendrá problemas ya que este tipo de conflictos normalmente son difíciles de reparar.

Utilizando **git rebase** se consigue un log de **commit** más sencillo y como se puede apreciar es muy fácil utilizarlo; pero como se ha comentado antes, cuidado con usarlo con **commits** que están en servidores remotos.

## 1.6.3.- Fundamentos de GitHub

### 1.6.3.1.- Repositorios remotos, Push, Pull y Fetch.

Tal como se mencionó al inicio de este documento, GitHub es un sistema de gestión de proyectos y control de versiones de código, incluyendo además una plataforma de red social diseñada para desarrolladores. Permite trabajar en colaboración con otras personas de todo el mundo, planificar proyectos y realizar un seguimiento del trabajo. Es también uno de los repositorios online más grandes de trabajo colaborativo en todo el mundo.

Si Git es el corazón de GitHub, entonces Hub es su alma. El hub de GitHub es lo que convierte una línea de comandos, como Git, en la red social más grande para desarrolladores. Además de contribuir a un determinado proyecto, GitHub le permite a los usuarios socializar con personas de ideas afines. Puedes seguir a las personas y ver qué hacen o con quién se conectan.

GitHub es una excelente plataforma que cambia la forma en que trabajan los desarrolladores. Sin embargo, toda persona que quiera administrar su proyecto de



manera eficiente y trabajar en colaboración también puede usar GitHub. Si un equipo trabaja en un proyecto que necesita actualizaciones constantes y quiere hacer un seguimiento a los cambios realizados, GitHub es adecuado para ello. Hay otras alternativas como GitLab o BitBucket, pero GitHub debería estar entre las primeras opciones.

Existen conceptos importantes que validar respecto de esta herramienta, tales como **push** y **pull**. A continuación se explicará cada uno de ellos.

### Subir confirmaciones (push)

Utiliza **git push** para subir confirmaciones de cambios realizadas en una rama local a un repositorio remoto. El comando **git push** toma dos argumentos:

- Un nombre remoto, por ejemplo, **origin**
- Una rama remota, por ejemplo, **master** (principal)

```
$ git push [REMOTENAME] [BRANCHNAME]
```

Generalmente se ejecuta **git push origin master** para subir los cambios locales a tu repositorio en línea.

Para renombrar una rama se debe utilizar el mismo comando git push, pero agregando un argumento más: el nombre de la nueva rama. El comando estándar sería el que se indica a continuación.

```
$ git push <REMOTENAME> <LOCALBRANCHNAME>:<REMOTEBRANCHNAME>
```

Esto sube **LOCALBRANCHNAME** a tu **REMOTENAME**, pero es renombrado a **REMOTEBRANCHNAME**.

Si una copia local de un repositorio está desincronizada, o "atrasada", con respecto al repositorio ascendente al que se está subiendo, se obtendrá un mensaje que indica **"non-fast-forward updates were rejected"** (las actualizaciones sin avance rápido se rechazaron). Esto significa que se debe recuperar, o "extraer", los cambios ascendentes, antes de poder subir los cambios locales.



Por defecto, y sin parámetros adicionales, **git push** envía todas las ramas que coinciden para que tengan el mismo nombre que las ramas remotas. Para subir una etiqueta única, se debe ejecutar el mismo comando que al subir una rama:

```
$ git push <REMOTENAME> <TAGNAME>
```

En caso que se desee subir todas las etiquetas, se puede escribir el comando:

```
$ git push <REMOTENAME> --tags
```

Para eliminar una etiqueta o una rama específica, se puede usar igualmente el comando **push**. La sintaxis respectiva se indica a continuación; tener en cuenta que en el ejemplo hay un espacio antes de los dos puntos. El comando anterior es similar a los mismos pasos que se aplican para renombrar una rama; sin embargo, aquí se está indicando a Git que no suba nada dentro de **[BRANCHNAME]** en **[REMOTENAME]**, y es gracias a eso que **git push** elimina la rama en el repositorio remoto.

```
$ git push <REMOTENAME> :<BRANCHNAME>
```

### Integrar un repositorio (pull y fetch)

El comando **pull** incorpora cambios desde un repositorio remoto en la rama actual. En su modo predeterminado, la instrucción **git pull** es la abreviatura de **git fetch** seguido de **git merge FETCH\_HEAD**. La sintaxis del comando **pull** sigue la estructura que se muestra en el siguiente ejemplo.

```
$ git pull [<OPCIONES>] [<REPOSITORIO> [<OPCIONES>...]]
```

Dicho de otra manera, **git pull** ejecuta el comando **git fetch** con los parámetros dados y llama a **git merge** para fusionar las cabezas de las ramas recuperadas en la rama actual. A través del comando **--rebase**, se ejecuta dicha acción en lugar de **git merge**.

En la ilustración anterior, el texto **<REPOSITORIO>** debería ser el nombre de un repositorio remoto. **<OPCIONES>** puede nombrar una referencia remota arbitraria (por ejemplo, el nombre de una etiqueta) o incluso una colección de referencias con



las correspondientes ramas de seguimiento remoto (por ejemplo, **refs / heads / \***: **refs / remotes / origin / \***), pero generalmente es el nombre de una rama en el repositorio remoto.

Considere el siguiente caso: suponga que existe el siguiente historial y que la rama actual es "**master**". El árbol en Git se vería de la manera en que se muestra en la ilustración siguiente.

```
      A---B---C master on origin
      /
D---E---F---G master
      ^
      origin/master in your repository
```

#### *Estructura de ejemplo previo a comando pull*

Después de ello, **git pull** buscará y reproducirá los cambios desde la rama maestra remota, ya que divergió del maestro local (es decir, E) hasta su confirmación actual (C) en la parte superior de la maestra y registrará el resultado en una nueva confirmación junto con el nombres de las dos confirmaciones principales y un mensaje de registro del usuario que describe los cambios.

```
      A---B---C origin/master
      /           \
D---E---F---G---H master
```

#### *Estructura de ejemplo posterior a comando pull*

En Git desde las versión 1.7.0 y posteriores, para cancelar una fusión conflictiva, tal como se había visto se debe usar el comando **git reset --merge**. Se debe tener en consideración que en versiones anteriores de Git, se desaconseja ejecutar **git pull** con cambios no confirmados: si bien es posible, lo deja en un estado que puede ser difícil de abandonar en caso de conflicto. Si alguno de los cambios remotos se superpone con cambios locales no confirmados, la fusión se cancelará automáticamente y el árbol de trabajo no se tocará. En general, es mejor obtener los cambios locales en funcionamiento antes de extraerlos o guardarlos con **git-stash**.

#### **1.6.3.2.- Clonar un repositorio.**

Cuando se crea un repositorio en GitHub, se puede decir que existe un repositorio remoto, ubicado en los servidores del proveedor. De ahí en más es posible clonar







dicho repositorio, a fin de crear una copia local en una computadora personal y sincronizarla entre ambas ubicaciones.

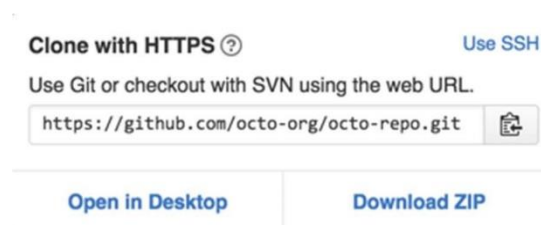
Los pasos que se indicarán a continuación parten de la base que se ha creado un repositorio en GitHub, o bien que se posee un repositorio existente que es propiedad de alguien más con quien se desee colaborar.

1. En GitHub, visita la página principal del repositorio. Si el repositorio está vacío, es posible copiar manualmente la URL de las páginas del repositorio desde el navegador y omitir el paso cuatro.
2. Bajo el nombre del repositorio, haz clic en “Clonar o Descargar”.



*Sitio de Github - Botón para clonar un repositorio*

3. Para clonar el repositorio utilizando HTTPS, en la opción "Clonar con HTTPS" se debe presionar el botón . Para clonar el repositorio utilizando una clave SSH, incluido un certificado emitido por la autoridad de certificación de la organización a la que aplique la tarea, se debe hacer clic en la opción “Usar SSH” y luego en el botón .



*Formulario con dirección de acceso a repositorio*

4. Las siguientes sentencias serán comandos implementados en una consola. Por tanto, en Windows se debe abrir la consola de comandos de Git, mientras que en Mac o Linux se debe abrir una ventana del terminal.
5. Se debe cambiar el directorio de trabajo actual a la ubicación en la que se desea que se clone el directorio.



6. Se debe agregar la sentencia **git clone**, y luego añadir la URL que se obtuvo como resultado en el Paso 2; en los valores **[USUARIO]** y **[REPOSITORIO]** se debe indicar el nombre del usuario y nombre del repositorio a los que se desea acceder, respectivamente. Presionando la tecla Enter (Intro), se creará un clon local del repositorio en la nube.

```
$ git clone https://github.com/[USUARIO]/[REPOSITORIO]
```

*Sentencia para clonar un repositorio*

### 1.6.3.3.- Documentando un proyecto con Markdown.

#### ¿Qué es Markdown?

Markdown es una base de texto sin formato utilizada para escribir documentos estructurados, basado en convenciones que indican el formato en correos electrónicos y publicaciones de Usenet (sistema de comunicación entre ordenadores para el intercambio de archivos). Fue desarrollado por John Gruber (con la ayuda de Aaron Swartz) y lanzado en 2004 en forma de una descripción de sintaxis y un script de Perl (Markdown.pl) para convertir Markdown a HTML. En la década siguiente se desarrollaron varias implementaciones en muchos idiomas, algunas de ellas ampliando la sintaxis original de Markdown con convenciones para notas al pie, tablas y otros elementos del documento, otras permitieron que los documentos de Markdown se representaran en formatos distintos al HTML.

Sitios web como Reddit, StackOverflow y GitHub tenían millones de personas que usaban Markdown. Y Markdown comenzó a usarse más allá de la web, para crear libros, artículos, presentaciones de diapositivas, cartas y apuntes. Lo que distingue a esta herramienta de otras sintaxis de marcado livianas, que a menudo son más fáciles de escribir, es su legibilidad.

#### ¿Cómo se relaciona GitHub con Markdown?

GitHub combina la sintaxis para el texto con formato Markdown con algunas características de escritura únicas. Tal como se indicó antes, Markdown es una sintaxis fácil de leer y fácil de escribir para el texto simple con formato. Es importante destacar que este formato de escritura aplica tanto para los comentarios al subir una nueva versión, como a nivel de sitio web.

Por lo mismo, GitHub le ha agregado funcionalidades personalizadas para crear el formato Markdown de GitHub, usado para dar formato a la prosa y al código en



todo el sitio. También es posible interactuar con otros usuarios en las solicitudes de extracción y las propuestas, usando funciones como @menciones, propuestas y referencias PR, incluso emojis.

En lo que respecta a los comentarios en el sitio web, para crear uno existe una barra de herramientas de formato de texto, lo que permite dar formato al texto sin tener que aprender la sintaxis de Markdown de memoria. Además del formato de Markdown como la negrita y la cursiva y crear encabezados, enlaces y listados, la barra de herramientas incluye características específicas de GitHub, como las @menciones, los listados de tareas y los enlaces a propuestas y solicitudes de extracción.

A continuación se indican algunos tipos de formatos aplicables.

| Estilo                       | Sintaxis                  | Atajo del teclado   | Ejemplo                                                     | Resultado                               |
|------------------------------|---------------------------|---------------------|-------------------------------------------------------------|-----------------------------------------|
| Negrita                      | <b>** ** o _ _</b>        | command/control + b | <b>**Este texto está en negrita**</b>                       | Este texto está en negrita              |
| Cursiva                      | <i>* * o _ _</i>          | command/control + i | <i>*Este texto está en cursiva*</i>                         | Este texto está en cursiva              |
| Tachado                      | <del>~~ ~~</del>          |                     | <del>~~Este texto está equivocado~~</del>                   | <del>Este texto está equivocado</del>   |
| Cursiva en negrita y anidada | <b><i>** ** y _ _</i></b> |                     | <b><i>**Este texto es _extremadamente_ importante**</i></b> | Este texto es extremadamente importante |
| Todo en negrita y cursiva    | <b><i>*** **</i></b>      |                     | <b><i>***Todo este texto es importante***</i></b>           | Todo este texto es importante           |

Para acceder a más estilos y formas de codificación y resaltado de texto, se recomienda visitar el sitio de ayuda del proyecto GitHub<sup>[4]</sup>.

#### 1.6.3.4.- Administrando Pull Request.

##### ¿Qué es un fork?

La palabra fork, en el contexto de GitHub, se puede traducir como “bifurcación”. Cuando se hace un fork de un repositorio, se hace una copia exacta en crudo (en inglés «bare») del repositorio original que se puede utilizar como un repositorio git cualquiera. Después de hacer fork se contará con dos repositorios git idénticos pero, con distinta URL. Justo después de hacer el fork, estos dos repositorios tienen exactamente la misma historia, son una copia idéntica. Finalizado el proceso, se tendrán dos repositorios independientes que pueden cada uno evolucionar de forma totalmente autónoma. De hecho, los cambios que se hacen el repositorio original NO se transmiten automáticamente a la copia (fork). Esto tampoco ocurre a la inversa: las modificaciones que se hagan en la copia (fork) NO se transmiten automáticamente al repositorio original.



Un fork no es lo mismo que clonar el repositorio. Cuando se hace un clon de un repositorio, se baja una copia del mismo a una máquina personal. Se empieza a trabajar, se hacen modificaciones y se hace un **push**. Cuando se hace el **push** se está modificando el repositorio que se ha clonado. Cuando se hace un fork de un repositorio, se crea un nuevo repositorio en una cuenta de Github o Bitbucket, con una URL diferente (fork); acto seguido se tiene que hacer un clon de esa copia sobre la que empiezas a trabajar de forma que cuando se hace **push**, se está modificando **solo la copia** (fork), o sea, el repositorio original sigue intacto.

Los pasos para hacer un fork son los siguientes:

1. Ingrese al sitio de <http://www.github.com>
2. Haga clic en el botón “Sign In”, e ingrese su usuario y clave GitHub
3. En otra pestaña del navegador, pegue la dirección del repositorio que desee copiar.
4. En la parte superior, aparecerá un botón “Fork”; presiónelo y se creará una copia del mismo en su cuenta.
5. Recuerde que el espacio creado será totalmente independiente del original, por tanto los cambios que se realicen aplicarán solo en la cuenta actual.

### ¿Qué es un pull request y cómo se realiza?

Un **pull request** es una petición que el propietario de un fork de un repositorio hace al propietario del repositorio original para que este último incorpore los cambios o **commits** que están en el fork.

El procedimiento para crear una nueva rama del repositorio es la siguiente:

```
$ git checkout master
Switched to branch 'master'
$ git checkout -b nuevarama
Switched to a new branch 'nuevarama'
```

*Crear una rama desde un fork*

Una vez hecho esto, se debe cambiar a la rama creada, hacer los cambios y aplicarlos.



```
$ git checkout nuevarama  
$ git push -u origin nuevarama
```

***Aplicar cambios sobre nueva rama***

Cuando se realizan los cambios, la tarea siguiente es solicitar un pull request al creador del repositorio original. Esta acción se hace desde el sitio, proceso que se puede realizar siguiendo los pasos que se indican a continuación.

- Ingresar al sitio <http://www.github.com>.
- Haga clic en el botón **"Sign In"**, e ingrese su usuario y clave GitHub.
- Haga clic en el repositorio que desea proponer para actualización.
- En donde dice **"branch: master"**, cambie la opción por la rama **"nuevarama"**, la que creó anteriormente.
- Presione el botón **"Pull Request"** que se despliega en la sección superior.
- En la pestaña **"New Pull Request"**, dentro de la información que se debe enviar para la revisión, debe indicar:
  - Rama base: en qué rama destino se desean incorporar los cambios o Head branch: rama en la que está la versión que se desea aplicar o Título del pull request
  - Descripción del pull request (admite formato Markdown y archivos adjuntos)
- En la pestaña **"Commits"** se verán las actualizaciones que el propietario original agregará si acepta la solicitud.
- En la pestaña **"Files Changed"** se indican los archivos específicos que serán parte de la actualización.
- Presionar el botón **"Send pull request"**.

El último paso es que el propietario original del repositorio acepte la solicitud, para lo cual debe ingresar al sitio <http://www.github.com>, ingresar al repositorio aludido y analizar las solicitudes recibidas. Una vez revisado el código y confirmado que todo está bien, se hace clic sobre el botón **"Merge Pull Request"**; en ese momento Github pedirá introducir un comentario para el **merge commit**. Finalmente se hace clic sobre el botón **"Commit merge"**, con lo cual Github automáticamente cierra el pull request.

Al mismo tiempo, la plataforma envía un email a los interesados avisándoles que se han incorporado los cambios en el repositorio.



### 1.6.3.5.- Flujos de trabajo con GitHub.

Los flujos de trabajo son procesos automatizados personalizados que se pueden configurar en un repositorio para crear, probar, empaquetar, lanzar o implementar cualquier proyecto en GitHub. Con los flujos de trabajo, es posible automatizar el ciclo de vida del desarrollo de cualquier software, gracias a una amplia gama de herramientas y servicios. Dentro de las opciones que existen, se puede crear más de un flujo de trabajo en un repositorio; para lograr esto se debe almacenar los flujos de trabajo en el directorio **.github/workflows** en la raíz de su repositorio.

Los flujos de trabajo deben tener al menos un trabajo, y los trabajos deben contener un conjunto de pasos que ejecuten tareas individuales. Los pasos pueden ejecutar comandos o utilizar una acción. Un usuario puede crear acciones, o bien usar acciones compartidas por la comunidad GitHub y personalizarlas según sea necesario. Otra acción que se puede realizar, es configurar un flujo de trabajo para que comience cuando se produce un evento GitHub, en un horario o desde un evento externo.

Los flujos de trabajo se deben configurar a través de la sintaxis **YAML**<sup>[7]</sup> y guardarlos como archivos de flujo de trabajo en algún repositorio. Una vez creado con éxito un archivo de flujo de trabajo YAML y activado este último, se desplegarán los registros de construcción, los resultados de las pruebas, los artefactos y los estados de cada paso de un flujo de trabajo.

Para crear un flujo de trabajo, se recomienda seguir los siguientes pasos:

1. En la raíz del repositorio, cree un directorio denominado **.github/workflows** para almacenar los archivos de flujo de trabajo.
2. En **.github/workflows**, agregue un archivo **.yml** o **.yaml** al flujo de trabajo. Por ejemplo, **.github/workflows/continuous-integration-workflow.yml**.
3. Utilice la documentación de referencia "**Sintaxis de flujo de trabajo para GitHub Actions**"<sup>[8]</sup> para elegir eventos a fin de desencadenar una acción, agregar acciones y personalizar el flujo de trabajo.
4. Confirme los cambios en el archivo de flujo de trabajo en la rama donde se desea que se ejecute el flujo de trabajo.

Para revisar ejemplos de flujos de trabajo, se recomienda visitar el sitio [6]. Se recomienda partir por acciones simples como las que ahí se indican, hasta personalizar las acciones de acuerdo a las necesidades del proyecto.



#### 1.6.4.- Referencias

[1] Sitio oficial de Git

Referencia: <https://git-scm.com/>

[2] Diez razones para usar Github

Referencia: <https://gist.github.com/erlinis/57a55dfb0337f5cd15cd>

[3] Ramificaciones en Git

Referencia: <https://git-scm.com/book/es/v2/Ramificaciones-en-Git-%C2%BFQu%C3%A9-es-una-rama%3F>

[4] Sintaxis de escritura y formatos básicos

Referencia: <https://help.github.com/es/github/writing-on-github/basic-writing-and-formatting-syntax>

[5] Proyecto GitHub

Referencia: <https://www.github.com/>

[6] Configurar un flujo de trabajo

Referencia: <https://docs.github.com/es/free-pro-team@latest/actions/reference/workflow-syntax-for-github-actions>

[7] El formato YAML

Referencia: [http://gitnacho.github.io/symfony-docs-es/components/yaml/yaml\\_format.html](http://gitnacho.github.io/symfony-docs-es/components/yaml/yaml_format.html)

[8] Sintaxis de flujo de trabajo para acciones de GitHub

Referencia: <https://docs.github.com/es/free-pro-team@latest/actions/reference/workflow-syntax-for-github-actions>