

# Git, Github y Bitbucket

## Introducción

Antes de entrar a hablar en profundidad de los 3 proyectos, hay que saber diferenciarlos. Git es un software de control de versiones desarrollado por Linus Torvalds, que es el mismo que desarrolló el kernel de Linux, pensando, sobre en todo, en la fiabilidad y eficiencia del control de versiones de los proyectos subidos por los diferentes usuarios. En el desarrollo software, el control de la versiones es muy importante, pues permite trabajar a un grupo de personas, sobre un proyecto distribuido, para luego cada uno poder unir sus partes, revisando cuales pueden ser las partes que pueden entrar en conflicto. Lo bueno que tiene git, es que permite crear ramas separadas de la rama principal (la rama principal es la denominada master, que debería ser siempre una rama estable y que debería contener un código funcional del proyecto). A partir de la rama principal, o rama master, podría sacar otra rama (branch en inglés) y empezar a realizar allí los cambios que necesito. Después, una vez que vea que los cambios son estables, intentar intregarlo en la rama principal, teniendo en cuenta que los demás han podido hacer lo mismo y haber modificado la rama principal. Este estricto control de versiones, permite que proyectos tan grandes tengan cierto control sobre que se sube, quien y cuando lo subió, en caso de que algo falle quien provocó el fallo, y volver a un punto anterior en caso de inestabilidad. Por el contrario, GitHub y Bitbucket son software de desarrollo web, que usan el sistema de control de versiones de GIT. Además, no solo pueden utilizar este tipo de control de versiones, pueden utilizar SVN (subversión) o mercurial. Es decir, Github y Bitbucket usan el software proporcionado por git para que los usuarios suban en su plataforma los diferentes proyectos, añadiendo valor adicional con diferentes características.

## Concepto de repositorio

Tenemos que tener bien claro el concepto de repositorio. Cuando generamos un proyecto y lo almacenamos en uno de estos programas, estamos generando un repositorio, el repositorio permite **centralizar** y **compartir** esos archivos (por ejemplo, un proyecto que deseamos que varias personas pueden trabajar sobre él), teniendo lo anteriormente en cuenta, ¿podríamos considerar un repositorio como una nube o un hosting web, ya que también permite centralizar y compartir esos archivos? La respuesta es **NO**, aunque puedan usarse para el mismo fin (por ejemplo, que funcionen para almacenar y compartir un proyecto) el repositorio suele contar con sistemas de respaldo en caso de fallo de la máquina, además que se suele almacenar en un único elemento de hosting, sin encontrarse distribuido entre diferentes máquinas, o en caso de estarlo, lo que estaría distribuida sería la copia de respaldo.

Llegados a este punto, nos convendría comprender el alcance del repositorio, por ejemplo, ¿es conveniente usar un repositorio para un proyecto de **una** persona? La respuesta es: depende. Si vemos que ese proyecto puede ampliarse a más personas en

un futuro, o si el usuario de dicho proyecto prevé de alguna manera que va a necesitar volver a un punto anterior en su proyecto, entonces sí que es recomendable, en caso contrario, sería similar a subir el proyecto a la nube, en la que vas subiendo versiones estables de la misma sustituyendo lo que ya hay anteriormente. Y la segunda pregunta, ¿Qué ventajas ofrece, por tanto, subir un proyecto a un gestor de versiones, frente a una nube, por ejemplo? La ventaja es más que evidente. Los gestores de contenido no solo están optimizados para la compartición de proyectos y las actualizaciones de manera liviana, si no que permite recuperar el proyecto en cualquier punto de su desarrollo, considerándose un punto de guardado el hecho de que algún usuario haya realizado algún commit -> push contra alguna rama del proyecto en una fase estable (se pueden crear ramas para subir fases inestables del proyecto, por ejemplo, por investigación, estas ramas se van separando de la rama master principal, y no suelen precisar integración con la rama principal).

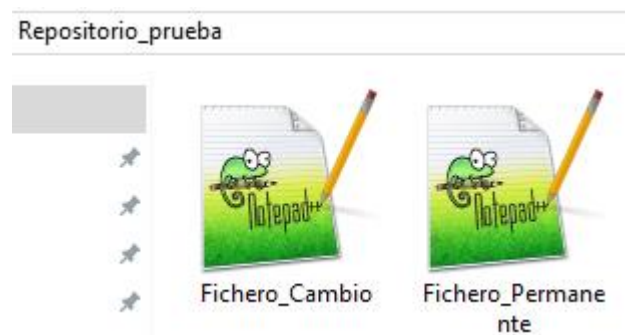
## 1º) Fase, crearse un repositorio y administrarlo a nivel de un único usuario

Ahora vamos a administrar un repositorio a nivel de monousuario, esto, evidentemente, es la forma más simple de administración de un repositorio, pero partimos de esta base para poder administrar un repositorio a nivel multiusuario.

Para realizar la práctica, hemos decidido usar bitbucket como gestor de versiones en lugar de github. La elección de una u otra se debe a que bitbucket la he usado con anterioridad y se manejarla mejor, pero a la hora de la verdad, se usan las mismas sentencias para administrar los proyectos al usar git.

Nos creamos un nuevo usuario en nuestra cuenta de bitbucket. Para ello es suficiente con registrarnos en el sistema. Después, para poder usar los comandos de git, basta con descargar git para nuestro sistema operativo. Para descargarnos git nos vamos a esta página: <https://git-scm.com/book/en/v2/Getting-Started-Installing-Git> para cada sistema operativo habrá una manera distinta de instalarse Git.

En nuestra zona de trabajo, vamos a crearnos una carpeta, esa carpeta va a ser la que contenga nuestro proyecto (por ahora, vamos a trabajar con dos tipo texto, le vamos a denominar fichero\_cambio.txt y fichero\_permanente.txt). Esa carpeta la vamos a nombrar como repositorio\_prueba.



Dentro de Bitbucket, me voy a crear un nuevo repositorio, para ello, en la parte de arriba, le damos a repositorios y nos creamos un nuevo repositorio. Voy a nombrarlo como Repositorio\_Text.

Ahora, una vez que tengo el repositorio, ¿Cómo soy yo capaz de subir lo que tengo al repositorio? Para ello, nos vamos a desplazar hasta la carpeta donde tengamos nuestro proyecto, pero usando la consola de comandos. En mi caso, se encuentra en el escritorio, por tanto esta será la ruta: `C:\Users\Potote\Desktop\Repositorio_prueba>`. Cuando estemos dentro del directorio donde queramos compartir los archivos, vamos a subir los archivos al repositorio:

## Git init

Con esta función, lo que le decimos a git, es que a partir de ahora, esa carpeta donde se encuentra nuestros archivos va a ser un repositorio local, y se va a crear una carpeta oculta denominada .git, a partir de ahora, podemos trabajar como si fuese un repositorio normal y corriente y usando los comandos de git

## git remote add origin “url del repositorio”

Con esta función ya estamos indicando que vamos a subir nuestro repositorio local a dicho repositorio remoto. A partir de ahora, los pull se subirán a dicho repositorio.

## Git add “Nombre\_del\_Fichero”

Esta función lo que hace es registrar los cambios que hayamos realizado en el archivo “Nombre\_del\_Fichero”, lo que decimos con esto es que, es que los cambios que vayamos almacenando en local antes de subirlo al repositorio, queremos que dicho archivo también se tenga en cuenta

## Git add .

Esta función lo que hace es añadir TODOS los ficheros dentro de la carpeta donde se encuentra nuestro proyecto, con esta función, nos ahorramos trabajo de ir metiendo uno por uno todos los ficheros. Vamos a ejecutar esta función para nuestro ejercicio, y así ahorrar tiempo.

## Git status

En cualquier momento podemos ver esta función para ver la diferencia que existe entre el repositorio local con respecto al repositorio remoto. Como nosotros hemos realizado cambios en nuestro repositorio local, pero no en el remoto, nos tiene que avisar que existen cambios que queramos modificar con respecto a lo que hay en remoto:

```
C:\Users\Potote\Desktop\Repositorio_prueba>git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   Fichero_Cambio.txt
        new file:   Fichero_Permanente.txt
```

Vemos que nos está diciendo que en la rama master, existe dos ficheros que necesitan ser puseados al repositorio remoto. Pero antes de subir al repositorio remoto, es necesario que lo comitemos en local:

## Git commit

Con git commit, lo que hacemos es, una vez añadidos los archivos que queremos que subamos, guardamos los cambios en LOCAL para luego subirlo a remoto. Existen varias opciones a la hora de hacer commit:

### -m "comentario"

Con esta opción lo que hacemos es añadirlo un comentario a la hora de ver el cambio en el repositorio remoto. Cuando hagamos el cambio y queramos ver ese cambio, aparecerá esta descripción. Vamos a poner lo siguiente para subir

```
C:\Users\Potote\Desktop\Repositorio_prueba>git commit -m "Subida del
fichero cambio y del fichero permanente"

*** Please tell me who you are.

Run

git config --global user.email "you@example.com"
git config --global user.name "Your Name"

to set your account's default identity.
Omit --global to set the identity only in this repository.

fatal: empty ident name (for <Potote@Potote-PC.(none)>) not allowed
```

Como podemos ver, sale un error, git no sabe quién somos, para ello simplemente corremos las sentencias que nos pone:

```
C:\Users\Potote\Desktop\Repositorio_prueba>git config --global user.email "Potote_18@hotmail.com"

C:\Users\Potote\Desktop\Repositorio_prueba>git config --global user.name "Potote"

C:\Users\Potote\Desktop\Repositorio_prueba>git commit -m "Subida del fichero cambio y del fichero permanente"
[master (root-commit) eb75ab5] Subida del fichero cambio y del fichero permanente
 2 files changed, 11 insertions(+)
 create mode 100644 Fichero_Cambio.txt
 create mode 100644 Fichero_Permanente.txt
```

Vemos que se han actualizado 2 ficheros, y que hay un total de 11 líneas de cambio. Si se nos ha olvidado cambiar algo, por ejemplo, añadir una línea al fichero permanente, podemos cambiarlo, y volver a añadir dicho fichero al commit, solo que ahora solo va a cambiar 1 fichero y va a haber solo una única inserción.

```
C:\Users\Potote\Desktop\Repositorio_prueba>git add .

C:\Users\Potote\Desktop\Repositorio_prueba>git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   Fichero_Permanente.txt

C:\Users\Potote\Desktop\Repositorio_prueba>git commit -m "Subida del fichero cambio y del fichero permanente"
[master 291726e] Subida del fichero cambio y del fichero permanente
 1 file changed, 1 insertion(+)
```



Tenemos que subir dichos cambios al repositorio remoto:

## Git push -u origin master

Con esta función, vamos a subir los cambios del local al remoto, a la rama master, cuando hagamos esto, nos va a pedir la contraseña de nuestra cuenta de bitbucket, para ello debemos haber metido de manera correcta quienes somos a git:

```
C:\Users\Potote\Desktop\Repositorio_prueba>git push -u origin master
Password for 'https://Potote@bitbucket.org':
Counting objects: 7, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (7/7), 719 bytes | 0 bytes/s, done.
Total 7 (delta 1), reused 0 (delta 0)
To https://Potote@bitbucket.org/Potote/repositorio_text.git
 * [new branch]      master -> master
Branch master set up to track remote branch master from origin.
```

Con esto ya tendremos los cambios realizados en nuestro repositorio remoto, ahora podemos ver como nuestro repositorio ha sido modificado añadiéndole ese commit que habíamos hecho, y con ese mismo comentario:

Autor	Commit	Mensaje	Fecha
 Alvaro Nevado	<a href="#">291726e</a>	Subida del fichero cambio y del fichero permanente	12 minutos ago
 Alvaro Nevado	<a href="#">eb75ab5</a>	Subida del fichero cambio y del fichero permanente	13 minutos ago

Vale, ya podemos administrar los cambios que tenemos en nuestro repositorio pero, ahora, ¿Cómo continuamos?

## 2º) Fase, añadir a una persona a tu repositorio y que pueda descargárselo, añadir .ignore al archivo, conflictos con archivos.

Para este ejemplo, vamos a tener otra cuenta, y vamos a ir paso a paso para que se descargue nuestro repositorio.

Una vez se haya registrado y descargado GIT en su ordenador, le vamos a dar acceso a nuestro repositorio. Una vez tenga acceso al repositorio, va a realizar un cambio sobre el repositorio. Vamos a intentar pushear dicho cambio, ¿Qué error sale?

```
C:\Users\Potote\Desktop\Repositorio_Prueba>git push -u origin master
Username for 'https://github.com': cabala18
Password for 'https://cabala18@github.com':
To https://github.com/Potote/Repositorio_Prueba.git
 ! [rejected]      master -> master (fetch first)
error: failed to push some refs to 'https://github.com/Potote/Repositorio_Prueba.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
C:\Users\Potote\Desktop\Repositorio_Prueba>
```

Lo que está diciendo es que alguien ha realizado un push a la rama master mientras nosotros estábamos trabajando, lo que nos pide es primero, bajarnos mediante un pull lo que ya hay, para integrarlo con nuestro proyecto:

```
C:\Users\Potote\Desktop\Repositorio_Prueba>git pull origin master
From https://github.com/Potote/Repositorio_Prueba
 * branch      master      -> FETCH_HEAD
Auto-merging Fichero_Cambio.txt
CONFLICT (add/add): Merge conflict in Fichero_Cambio.txt
Automatic merge failed; fix conflicts and then commit the result.
```

El auto-merging lo que intenta es unir los cambios de ambos usuarios, solo funciona SI AMBOS USUARIOS REALIZAN CAMBIOS QUE NO SUPONGAN UN CAMBIO SOBRE EL MISMO FICHERO, si por ejemplo, Juan realiza un cambio sobre Juan.txt y Pedro sobre Pedro.txt, no hay problema, pero si los dos realizan cambios sobre Juan.txt, entonces el auto-merging no

funciona. Además de este tipo de conflicto, existe el conflicto de cuando un usuario borra un archivo que otro usuario, posteriormente, intenta modificar, en este caso, el auto-merging también falla, el usuario que está subiendo el archivo debe decidir si sus cambios persisten. Para comprobar dichos cambios, lo que tiene que hacer a continuación el usuario es un git pull. Haciendo esto, en el archivo con cambios se le marcarán que partes del código son las que resultan afectadas. Las partes del código afectadas empezaran con la sentencia >>>>>>HEAD y acabarán con la sentencia <<<<<<<< "Código de cambio". Dentro de esas partes afectadas, las partes que requieren atención estarán entre los símbolos =====. De tal manera que si realizamos un cambio en una línea de código, pero otro usuario ha realizado un cambio sobre la misma línea, ambos cambios estarían separados entre =====, y el usuario que está haciendo el push debe decidir cómo van a quedar los cambios almacenados. Una vez que realice los cambios oportunos, el usuario hará push para que los cambios persistan, esta vez sin que haya un mergering de por medio.