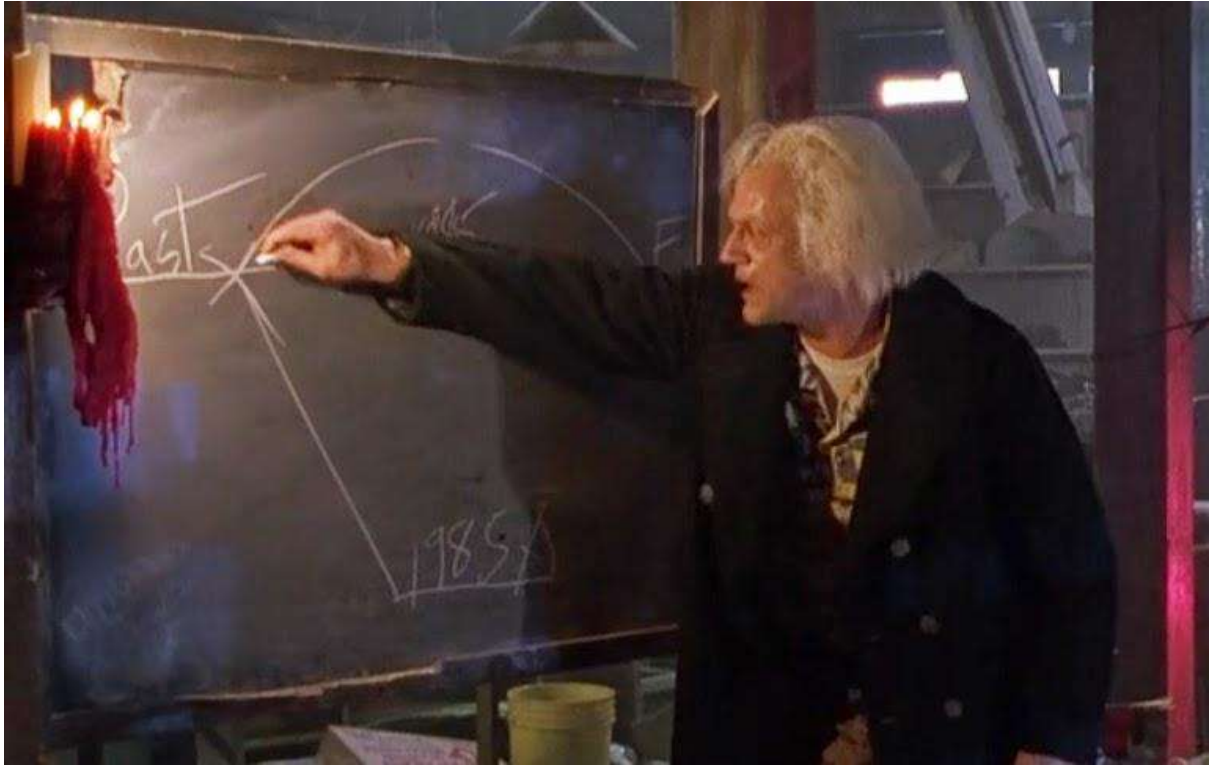


Programación Básica 2

Git Control de Versiones

¿QUÉ ES GIT?



Toda la información de esta unidad la puedes encontrar en más detalle en la página oficial de Git : <https://git-scm.com/book/es/v1/>

Git es un sistema de control de versiones, gratuito y de código abierto, diseñado para manejar desde pequeños a grandes proyectos de manera rápida y eficaz. Se entiende como control de versiones a todas las herramientas que nos permiten hacer modificaciones en nuestro proyecto. Un sistema que **registra los cambios realizados sobre un archivo o conjunto de archivos a lo largo del tiempo.**

Existen muchos sistemas de control de Versiones. Git es uno de ellos y el más aceptado por la comunidad de desarrolladores.

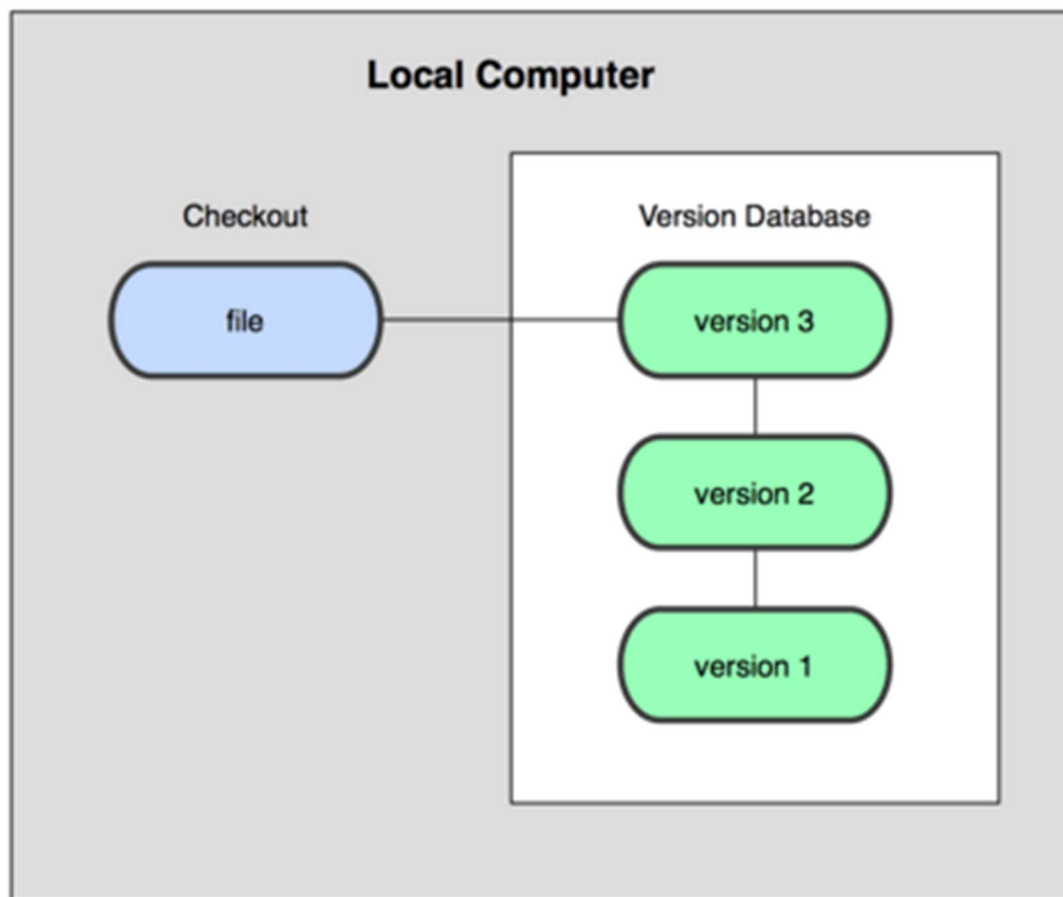
Gracias al sistema de control de Versiones, nuestro sistema se encuentra a salvo. Con Git podemos ir a cualquier estado de nuestro proyecto en cualquier momento.

VERSION CONTROL SYSTEM (VCS)

Los VCS están divididos en 3 tipos:

VCS Local

Es un VCS que **vive en tu computadora. Sólo allí**. Si nuestra computadora se rompe o destruye ya no podremos obtener nuestros archivos por razones obvias. Depende completamente de donde se esté manejando, en este caso, nuestra computadora.

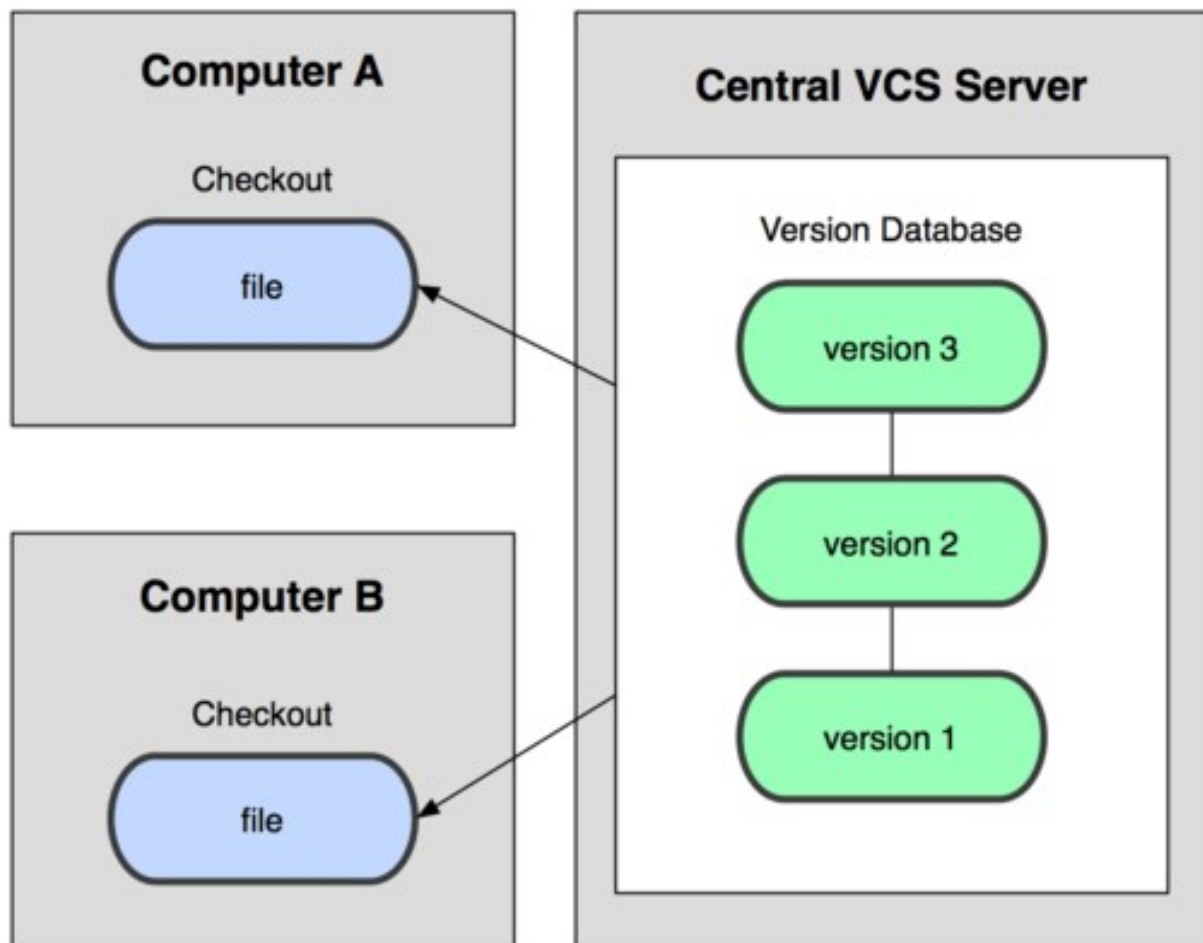


VCS centralizado

Estos sistemas, como CVS, Subversion, y Perforce, **tienen un único servidor que contiene todos los archivos versionados, y varios clientes que descargan los archivos desde ese lugar central**. Durante muchos años este ha sido el estándar para el control de versiones.

Esta configuración ofrece muchas ventajas, especialmente frente a VCSs locales. Por ejemplo, todo el mundo puede saber (hasta cierto punto) en qué están trabajando los otros colaboradores del proyecto. Los administradores tienen control detallado de qué puede hacer cada uno; y es mucho más fácil administrar un CVCS que tener que lidiar con bases de datos locales en cada cliente.

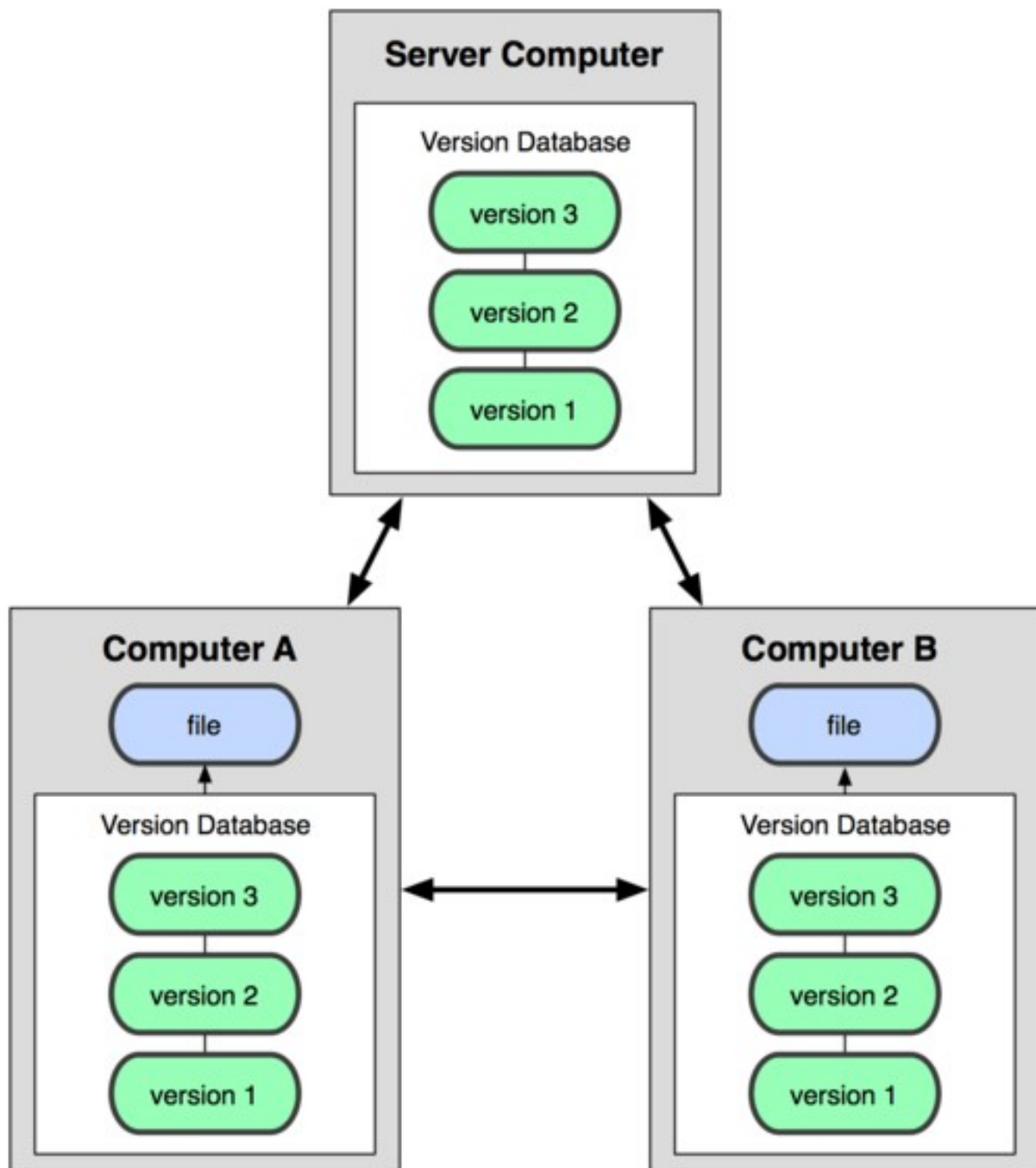
Sin embargo, esta configuración también tiene serias desventajas. La más obvia es el punto único de fallo que representa el servidor centralizado. **Si ese servidor se cae durante una hora, entonces durante esa hora nadie puede colaborar o guardar cambios versionados de aquello en que están trabajando**. Si el disco duro en el que se encuentra la base de datos central se corrompe, y no se han llevado copias de seguridad adecuadamente, pierdes absolutamente todo —toda la historia del proyecto salvo aquellas instantáneas que la gente pueda tener en sus máquinas locales. Los VCSs locales sufren de este mismo problema— **cuando tienes toda la historia del proyecto en un único lugar, te arriesgas a perderlo todo**.



VCS Distribuidos

Es aquí donde entran los sistemas de control de versiones distribuidos. En un DVCS (como Git, Mercurial, Bazaar o Darcs), los clientes **no sólo descargan la última instantánea de los archivos: replican completamente el repositorio**. Así, si un servidor muere, y estos sistemas estaban colaborando a través de él, **cualquiera de los repositorios de los clientes puede copiarse en el servidor para restaurarlo**. Cada vez que se descarga una instantánea, en realidad se hace una copia de seguridad completa de

todos los datos.



Es más, muchos de estos sistemas se las arreglan bastante bien teniendo varios repositorios con los que trabajar, por lo que puedes colaborar con distintos grupos de gente simultáneamente dentro del mismo proyecto. Esto **te permite establecer varios flujos de trabajo que no son posibles en sistemas centralizados**, como pueden ser los modelos jerárquicos.

HISTORIA DE GIT

Como muchas de las grandes cosas en esta vida, **Git comenzó con un poco de destrucción creativa y encendida polémica**. El núcleo de Linux es un proyecto de software de código abierto con un alcance bastante grande. Durante la mayor parte del mantenimiento del núcleo de Linux (1991-2002), los cambios en el software se pasaron en forma de parches y archivos. En 2002, **el proyecto del núcleo de Linux empezó a usar un DVCS propietario llamado BitKeeper**.

En 2005, la relación entre la comunidad que desarrollaba el núcleo de Linux y la compañía que desarrollaba BitKeeper se vino abajo, y la herramienta dejó de ser ofrecida gratuitamente. **Esto impulsó a la comunidad de desarrollo de Linux (y en particular a Linus Torvalds, el creador de Linux) a desarrollar su propia herramienta** basada en algunas de las lecciones que aprendieron durante el uso de BitKeeper. Algunos de los objetivos del nuevo sistema fueron los siguientes:

- Velocidad
- Diseño sencillo
- Fuerte apoyo al desarrollo no lineal (miles de ramas paralelas)
- Completamente distribuido
- Capaz de manejar grandes proyectos (como el núcleo de Linux) de manera eficiente (velocidad y tamaño de los datos)

Desde su nacimiento en 2005, **Git ha evolucionado y madurado para ser fácil de usar y aún conservar estas cualidades iniciales**. Es tremendamente rápido, muy eficiente con grandes proyectos, y tiene un increíble sistema de ramificación (branching) para desarrollo no lineal

Fuente(<https://git-scm.com/book/es/v1/Empezando-Una-breve-historia-de-Git>)

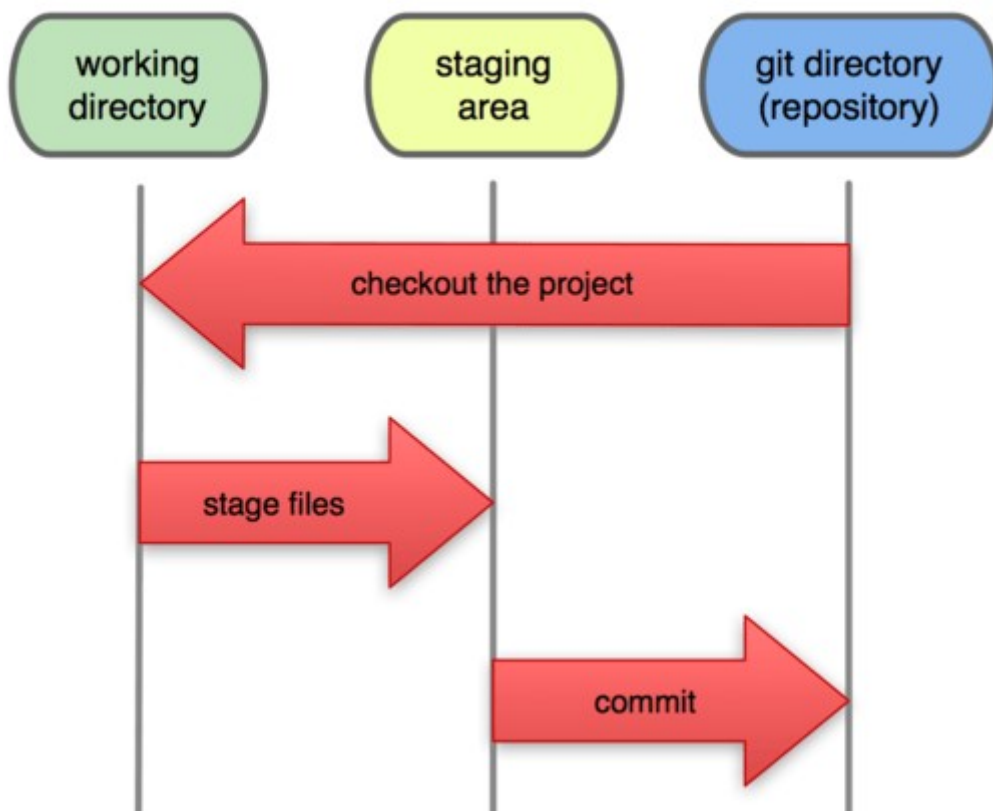
LOS TRES ESTADOS DE GIT

Git tiene tres estados principales en los que se pueden encontrar tus archivos: confirmado (committed), modificado (modified), y preparado (staged):

- -Confirmado significa que los datos están almacenados de manera segura en tu base de datos local.
- -Modificado significa que has modificado el archivo pero todavía no lo has confirmado a tu base de datos.
- -Preparado significa que has marcado un archivo modificado en su versión actual para que vaya en tu próxima confirmación.

Esto nos lleva a las tres secciones principales de un proyecto de Git: el directorio de Git (Git directory), el directorio de trabajo (working directory), y el área de preparación (staging area).

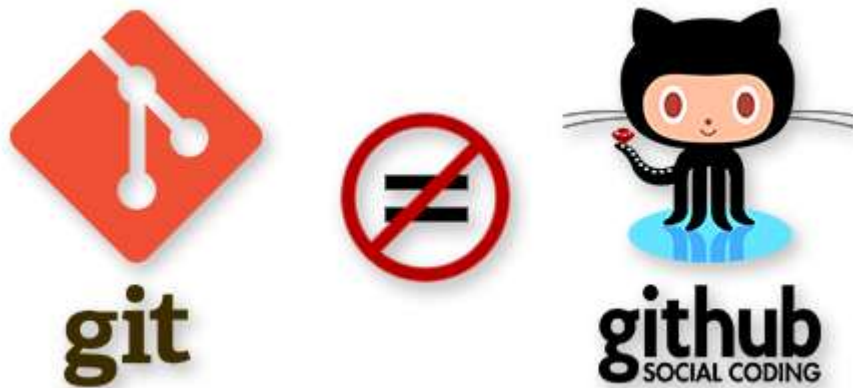
Local Operations



El flujo es: Creo los archivos en la carpeta del proyecto (*Working Directory*); luego indico

cuáles archivos voy a enviar al repositorio (*Staging Area*) y finalmente los envío al repositorio principal, llámese GitHub, Bitbucket u otro (*Git Directory*).

¿QUÉ ES GITHUB?



Git es uno de los sistemas de control de versiones más populares entre los desarrolladores. Y parte culpa de su popularidad la tiene GitHub, un excelente **servicio de alojamiento de repositorios de software con este sistema**, que lejos de quedarse en esta funcionalidad, ofrece hoy en día un conjunto de características muy útiles para el trabajo en equipo.

No en vano, es el servicio elegido por proyectos de software libre como jQuery, reddit, Sparkle, curl, Ruby on Rails, node.js, ClickToFlash, Erlang/OTP, CakePHP, Redis, y otros muchos. Además, algunas de las grandes empresas de Internet, como Facebook, alojan ahí sus desarrollos públicos, tales como el SDK, librerías, ejemplos, etc.

Github es un servicio para alojamiento de repositorios de software gestionados por el sistema de control de versiones Git. Github es un sitio web **pensado para hacer posible el compartir el código de una manera más fácil** y al mismo tiempo darle popularidad a la herramienta de control de versiones en sí, que es Git. **Cabe destacar que Github es un proyecto comercial, a diferencia de la herramienta Git que es un proyecto de código abierto.**

Para resumir:

GitHub: Plataforma de proyectos Git. Es popularmente conocida como la red social de desarrolladores, hay proyectos muy interesantes allí.

Otras plataformas pueden ser BitBucket o GitLab

INSTALACIÓN Y CONFIGURACIÓN DE GIT



Vamos a empezar a usar un poco de Git.

Lo primero es lo primero: tienes que instalarlo. Puedes obtenerlo de varias maneras; las

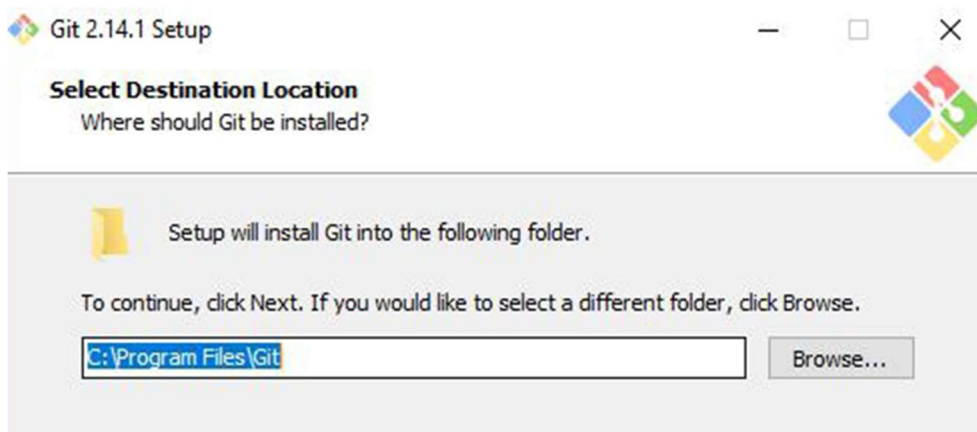
dos principales son instalarlo desde código fuente, o instalar un paquete existente para tu plataforma.

Instalando Git desde Windows

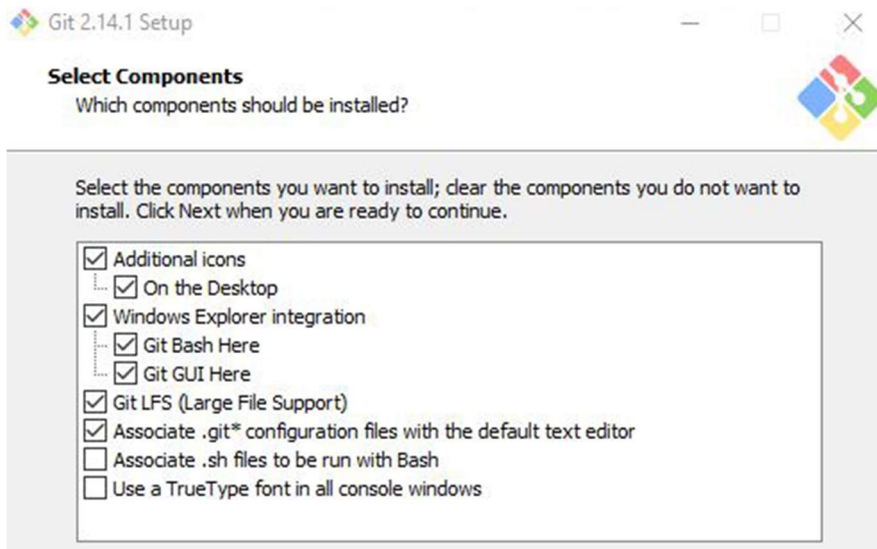
1 - Ve a <https://git-scm.com/> y descarga el paquete de instalación



2- Ahora debes **ejecutar el archivo descargado**, y elige la carpeta donde ubicar los archivos de Git



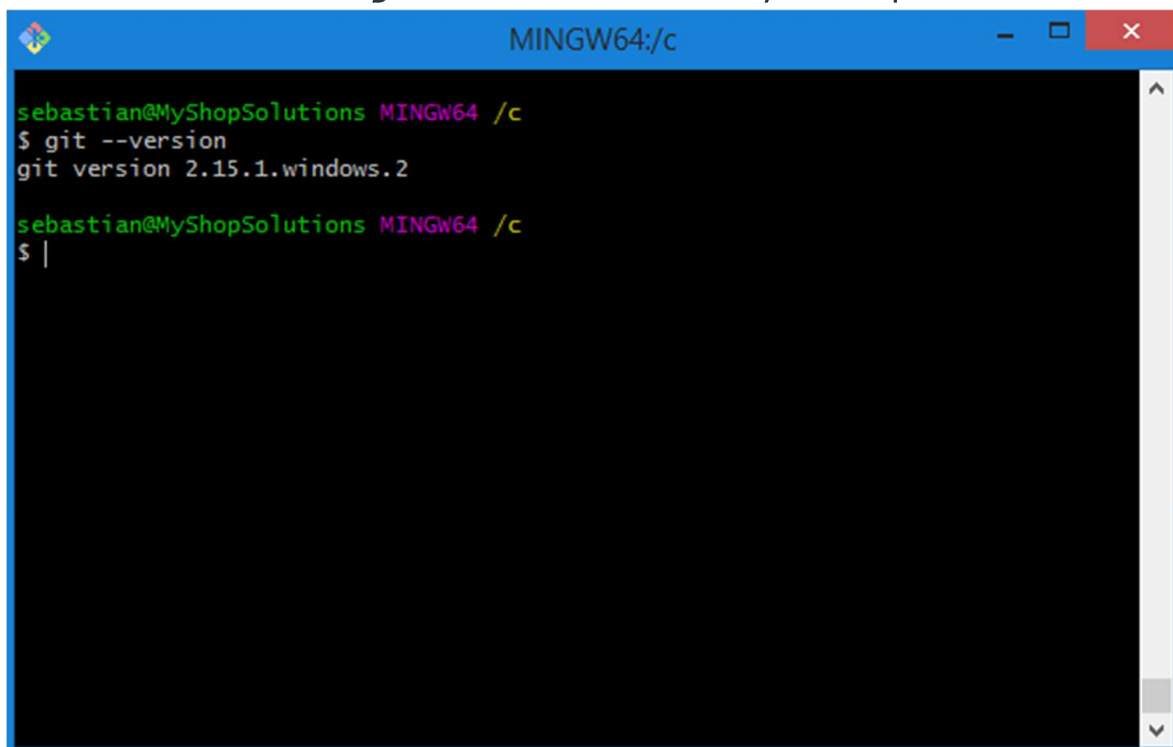
3- Asegúrate de tener seleccionado *git bash* que es la herramienta principal con la que trabajaremos y con esto se terminará la instalación



4- Ahora tendrás disponible *Git Bash* deste tu lista de programas, aquí es donde trabajaremos con Git.



5- Escribir *git* --version y presionar enter



```
sebastian@MyShopSolutions MINGW64 /c
$ git --version
git version 2.15.1.windows.2

sebastian@MyShopSolutions MINGW64 /c
$ |
```

¡Ya tienes Git instalado!

Configurando Git por primera vez

Ahora que tienes Git en tu sistema, querrás hacer algunas cosas para personalizar tu entorno de Git. Sólo es necesario hacer estas cosas una vez; se mantendrán entre actualizaciones. También puedes cambiarlas en cualquier momento volviendo a ejecutar los comandos correspondientes.

Tu identidad

Lo primero que deberías hacer cuando instalas Git es establecer tu nombre de usuario y dirección de correo electrónico. Esto es importante porque las confirmaciones de cambios (commits) en Git usan esta información, y es introducida de manera inmutable en los commits que envías:

```
john@MyShopSolutions MINGW64 /c
```

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

Comprobando tu configuración

Si quieres comprobar tu configuración, puedes usar el comando `git config --list` para listar todas las propiedades que Git ha configurado:

```
john@MyShopSolutions MINGW64 /c
$ git config --list
user.name=John Doe
user.email=johndoe@example.com
color.status=auto
color.branch=auto
color.interactive=auto
color.diff=auto
...
```

También puedes comprobar qué valor cree Git que tiene una clave específica ejecutando

```
john@MyShopSolutions MINGW64 /c
$ git config user.name
John Doe
```

Obteniendo ayuda

Si alguna vez necesitas ayuda usando Git, hay tres formas de ver la página del manual (manpage) para cualquier comando de Git:

```
john@MyShopSolutions MINGW64 /c
$ git help config
$ git config --help
$ man git-config
```

Resumen

Deberías tener un conocimiento básico de qué es Git y en qué se diferencia del CVCS que puedes haber estado utilizando. También deberías tener funcionando en tu sistema una versión de Git configurada con tu identidad. Es el momento de aprender algunos

fundamentos de Git.

COMANDOS BÁSICOS DE LA TERMINAL



Al principio era DOS. Cualquier acción, tarea o programa se ejecutaban mediante órdenes escritas. MKDIR, COPY, FIND, DELTREE o CD eran algunos de los comandos más populares.

Luego vino Windows y trajo consigo las ventanas, los iconos y las carpetas. Al principio dependía de DOS, pero más tarde se independizó y relegó la línea de comandos a lo que ahora conocemos como símbolo del sistema (command prompt o CMD) o consola de Windows.

Aunque la consola de comandos no es imprescindible, y muchos viven felices sin ella, todavía sigue siendo útil y podemos sacarle provecho si conocemos algunos de sus comandos más relevantes.

Aquí van algunos de ellos:

Para abrir la línea de comandos de Windows o símbolo del sistema tan sólo tenemos que ir a **Inicio > Ejecutar o Buscar > CMD.exe** y se abrirá una pequeña ventana que nos recordará al antiguo MS-DOS.

Para abrir **la Terminal de Mac OS** haz clic en el icono "Finder" situado en el Dock, luego seleccione "**Aplicaciones > Utilidades**" Finalmente dale doble clic al icono "**Terminal**"

Hay que tener varias cosas en cuenta:

- No diferencia entre mayúsculas y minúsculas
- Al escribir un nombre de archivo o carpeta con espacios conviene escribirlo entrecomillado
- Los nombres pueden ser de hasta 255 caracteres y con extensiones de hasta 3 caracteres. Si eliminas un archivo desde CMD no va a la Papelera.

/? Si quieres saber más de un comando, añade **/?** para ver la ayuda relacionada. Te será muy útil para ver las muchas opciones de cada comando.

HELP Te mostrará una lista de comandos disponibles.

DIR Es el comando más conocido de DOS y sirve para ver el contenido de una carpeta.
(en **MAC-OS** usar **LS**)

CD Sirve para entrar en una carpeta o salir de ella con **CD...**

CLEAR limpiar consola

MKDIR Con este comando crearás una carpeta nueva. Con **RMDIR** podrás eliminarla.

MOVE y **COPY** Son los comandos para mover y copiar respectivamente archivos. Deberás indicar el nombre del archivo con su ruta (si está en otra carpeta en la que te encuentras) y la ruta de destino.

RENAME Sirve para renombrar un archivo o carpeta. Hay que indicar el nombre original y el definitivo.

DEL Es el comando para eliminar un archivo. Recuerda que no irá a la **Papelera**, así que piénsatelo antes de borrar nada. Y para eliminar carpeta usa el comando **RD** (en **MAC-OS** usar **RM** para archivos / para eliminar carpetas **RM -RF**)

EXIT Cierra la ventana de la línea de comandos o símbolo del sistema.

COPY CON Crear archivos. (en **MAC-OS** usar **TOUCH**)

CREANDO REPOSITARIOS



Git init

Este comando se usa para crear un nuevo repositorio en Git. Nos creará un repositorio de manera local y lo hará en la carpeta donde estamos posicionados o se le puede pasar *[nombre_de_la_carpeta]* y creará la carpeta con ese nombre.

```
john@MyShopSolutions MINGW64 /c/git  
$ git init nuevo_repo
```

```
john@MyShopSolutions MINGW64 /c/git
$ mkdir nuevo_repo
$ git init
```

Git status

Ya hemos visto cómo inicializar un repositorio localmente utilizando *git init*. Ahora nos toca crear los archivos que vamos a usar en este repositorio.

Creando un archivo index.html

```
john@MyShopSolutions MINGW64 /c/git/nuevo_repo (master)
$ touch index.html
```

Luego utilizando el comando *git status* vamos a poder ver el estatus de los archivos que tenemos localmente, o sea en nuestro **Working Directory**

Lo que aparezca en **rojo** es lo que se ha modificado y hay que pasarlo a nuestro **Staging Area** (el limbo)

```
john@MyShopSolutions MINGW64 /c/git/nuevo_repo (master)
$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)

  index.html

nothing added to commit but untracked files present (use "git add" to track)
```

Git add

Para **agregar el o los archivos al Staging Area** vamos a usar el comando *add* lo cual podemos verificar si funciona nuevamente con el *git status*

```
john@MyShopSolutions MINGW64 /c/git/nuevo_repo (master)
```

```
$ git add index.html
$ git status
On branch master

No commits yet

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

    new file:   index.html
```

Ahora en **verde** nos indica que tenemos un archivo listo para hacer un commit en nuestro repositorio, hasta que no hagamos un commit nuestro archivo permanecerá en “el limbo” en el estado Ready justo antes de enviar nuestro archivo a nuestros repositorios.

Git commit

Una vez que nuestros archivos están en el **Staging Area** debemos pasarlos a nuestro repositorio local y para eso debemos usar el *git commit* que es el comando que **nos va a permitir comprometer** nuestros archivos.

```
john@MyShopSolutions MINGW64 /c/git/nuevo_repo (master)
$ git commit -m "nuevo archivo"
[master (root-commit) 1734915] nuevo archivo
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 index.html
```

Agregamos nuestros index.html

Entonces cumplimos las 3 etapas de Git

1. Creamos nuestro *index.html* dentro de **Working Directory**
2. Con el comando *add* agregamos al **Staging Area**
3. Y como último enviamos a nuestro repositorio **Git Directory** el archivo *index.html* con el comando *commit -m* (mensaje)

Git log

Por medio de este comando **veremos la historia de nuestro proyecto**.

Utilizando *git log* podrás ver los códigos hexadecimales que crea Sha-1, el autor y la fecha en la que se envió ese commit.

La documentación de *git log* es super extensa y puedes revisar completamente desde aquí

<https://git-scm.com/book/es/v1/Fundamentos-de-Git-Viendo-el-hist%C3%B3rico-de-confirmaciones>

```
john@MyShopSolutions MINGW64 /c/git/nuevo_repo (master)
$ git log
commit 1734915470ce9983f703b77807a68e42166b47dd (HEAD -> master)
Author: John Doe <johndoe@example.com>
Date: Sat Dec 15 18:53:24 2017 -0300

nuevo archivo
```

Git diff

Para hacer un control de la revisión de los cambios podemos usar el *git diff*

Si queremos **saber cuales son los cambios entre un commit y otro diferente**, solo debemos hacer una comparación entre un commit y otro usando el código proveniente desde el Sha-1

Entonces para comparar diferentes commit, lo primero es obtener los números de commit a comparar, podemos usar el comando *git log --oneline* (para ver los commits en su versión corta)

```
john@MyShopSolutions MINGW64 /c/git/nuevo_repo (master)
$ git log --oneline
95bd15b (HEAD -> master) nuevos cambios
1734915 nuevo archivo
```

Podemos ver las diferencias entre las versiones del documento utilizando *git diff [version 1] [version 2]*

```
john@MyShopSolutions MINGW64 /c/git/nuevo_repo (master)
$ git diff 1734915 95bd15b
diff --git a/index.html b/index.html
index e69de29..3c00a7b 100644
--- a/index.html
+++ b/index.html
@@ -0,0 +1,4 @@
- This is an old line
```

```
+This is a new version
+New text.
+
+And I add this new line
\ No newline at end of file
```

En **verde** nos marcan las líneas nuevas y en **rojo** las líneas eliminadas de nuestro documento

Extras

Por supuesto existen mucho más comandos, puedes encontrar la información completa desde la página oficial de Git y la documentación que nos ofrece.

<https://git-scm.com/book/es/v1/>

O ver el listado de comandos basicos para Git:

<https://www.hostinger.es/tutoriales/comandos-de-git>

También puedes practicar y ejercitar tus comandos Git desde aquí:

<https://try.github.io/levels/1/challenges/1>

O puedes también seguir esta guia sencilla de como instalar Git:

<http://rogerdudler.github.io/git-guide/index.es.html>

Si te gustaria probar consolas diferentes en Windows

<https://alternativeto.net/software/iterm2/?platform=windows>

O si quieres usar una de las mejores consolas para MAC-OS

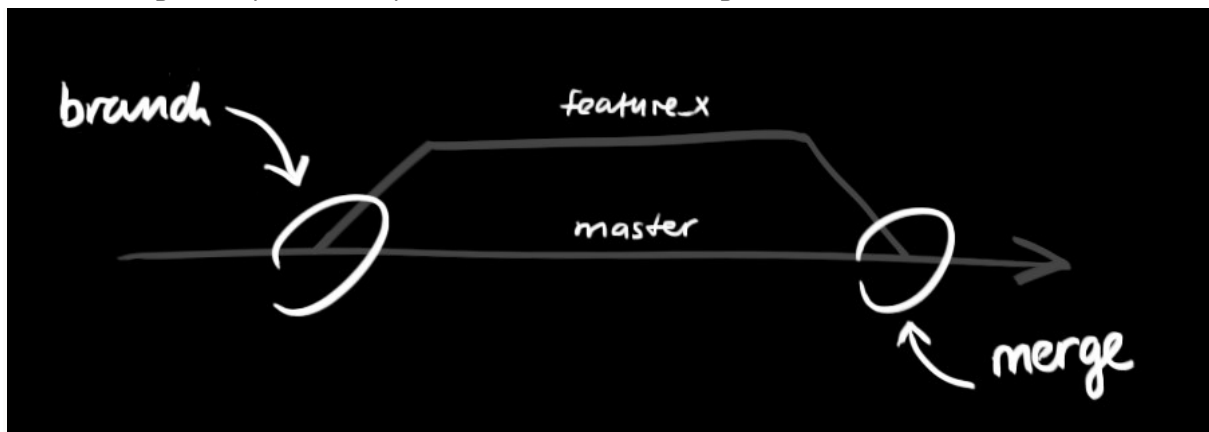
<https://www.iterm2.com/>

Ramas

En el trabajo con Git una de las características más útiles es el trabajo con ramas. Las ramas son caminos que puede tomar cualquier desarrollador dentro del proyecto, algo que ocurre naturalmente para resolver problemas o crear nuevas funcionalidades. Hemos estado sobre "Master" (la rama principal de nuestro proyecto), podríamos crear ramas para que otros desarrolladores del equipo trabajen sobre el proyecto en nuevas funcionalidades.

Llegará un momento en el que, quizás, aquellos cambios experimentales los quieras subir

a producción. Entonces harás un proceso de fusión entre la rama experimental (branch) y la rama original, operación que se conoce como merge en Git.



(Fuente: <http://rogerdudler.github.io/git-guide/>)

Veamos a continuación algunos comandos para crear y manejar variantes del repositorio por medio de ramas.

Git branch

Este comando lo usaremos para crear una nueva rama o un nuevo branch. En el ejemplo vamos a crear una nueva rama para los archivos de la nueva versión responsive de nuestro proyecto.

```
john@MyShopSolutions    MINGW64    /c/git/nuevo_repo    (master)
$ git branch responsive
```

Vamos ahora a listar nuestras ramas con el comando *git branch -l*

```
john@MyShopSolutions    MINGW64    /c/git/nuevo_repo    (master)
$ git branch -l
*
responsive
```

Aparece un listado con nuestras ramas (la rama principal -master-)

Para borrar una rama usamos el comando *git branch -D [nombre de la rama]*

```
john@MyShopSolutions    MINGW64    /c/git/nuevo_repo    (master)
$ git branch -D responsive
Deleted branch responsive (was 6d6c28c)
```

Volvemos a crear una nueva rama y esta vez vamos a renombrarla con el comando *git branch -m [nombre de la rama] [nombre nuevo de la rama]*

```
john@MyShopSolutions MINGW64 /c/git/nuevo_repo (master)
$ git branch features
$ git branch -m features f-responsive
```

Ahora la rama *features* pasó a llamarse *f-responsive*

Git checkout

Ahora vamos a ver con que comando podemos movernos entre nuestras diferentes ramas y posicionarnos en una como en otra para empezar a trabajar sobre ellas.

```
john@MyShopSolutions MINGW64 /c/git/nuevo_repo (master)
$ git checkout f-responsive
```

Luego de ejecutar el comando estaríamos trabajando sobre la rama indicada (en este caso “f-responsive”)

```
john@MyShopSolutions MINGW64 /c/git/nuevo_repo (f-responsive)
$
```

Y vamos a ver la diferencia en el sufijo de nuestra consola de Git Bash, “master” ha cambiado por el nombre de la rama en donde estamos parados.

Para volver a la rama “master” usamos el mismo comando pero esta vez apuntado a nuestro “master” *git checkout master*

También podríamos movernos y posicionarnos sobre algún commit específico y desde allí sacar una nueva rama

Primero buscamos y listamos los commits (usamos el comando *git log --online*, para que nos haga una lista de los commits con sus identificadores en su versión corta)

```
john@MyShopSolutions MINGW64 /c/git/nuevo_repo (master)
$ git log --online
6d6c28c (HEAD -> master, f-responsive) Agregue una línea en la cuarta posición
95bd15b nuevos cambios
1734915 nuevo archivo
```

Luego con el comando checkout nos posicionamos sobre el commit que seleccionamos

```
john@MyShopSolutions MINGW64 /c/git/nuevo_repo (master)
```



```
$ git checkout 95bd15b
Note: checking out '95bd15b'.
```

You are in 'detached HEAD' state. You can look around, make experimental changes and commit them, and you can discard any commits you make in this state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may do so (now or later) by using -b with the checkout command again. Example:

```
git checkout -b <new-branch-name>
```

```
HEAD is now at 95bd15b... nuevos cambios
john@MyShopSolutions MINGW64 /c/git/nuevo_repo ((95bd15b...))
```

Como podemos ver estamos ahora sobre el commit seleccionado (el HEAD está ahora) y podríamos sacar una nueva rama desde ahí, lo que hicimos fue **movernos en el tiempo para posicionarnos sobre ese commit pasado**.

Podríamos también crear una rama y posicionarnos con una sola línea de comando

```
john@MyShopSolutions MINGW64 /c/git/nuevo_repo (master)
$ git checkout -b nueva-imagen
john@MyShopSolutions MINGW64 /c/git/nuevo_repo (nueva-imagen)
```

Git merge

Habiendo trabajado en diferentes ramas deberíamos de mezclarlas. Lo primero que debemos hacer para mezclar ramas es pararnos sobre la rama master y desde allí usar el comando *git merge [rama a fundir con el master]*

```
john@MyShopSolutions MINGW64 /c/git/nuevo_repo (master)
$ git merge nueva-imagen
Updating 6d6c28c..fa52a3c
Fast-forward
 nuevo.txt | 0
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 nuevo.txt
```

Ahora los cambios de nuestra rama “nueva-imagen” fueron fundidos a nuestra rama “master”

Con el comando `git log --oneline --graph` podremos ver la bifurcaciones de nuestro proyecto con sus respectivas y diferentes ramas y como se fueron fusionando con nuestra rama master.

```
john@MyShopSolutions      MINGW64      /c/git/nuevo_repo      (master)
$ git log --oneline --graph
* 3bb3eac (HEAD -> master) Nuevo cambio
* db7e9e8 Mix merge de cambios
|\
| * 054f907 (nueva-imagen) Nueva línea de nueva imagen
* | 5c6788d Nuevo cambio por conflicto
|\ \
| * | 8bdde3d (f-responsive) Nueva línea de responsive
* || 54617a6 Nueva línea
* || 0e45eb0 Nuevo archivo en el master
|/ /
* | fa52a3c Nuevo en nueva rama
|/
* 6d6c28c Agregue una línea en la cuarta posición
* 95bd15b Nuevos cambios
* 1734915 Nuevo archivo
```

Clientes Gráfico para Git:

- Git-gui
- GitHub Desktop
- GitKraken
- SmartGit
- SourceTree

El primero de ellos es un juego llamado Oh My Git! Muy interactivo y entretenido.
<https://ohmygit.org/>

El segundo de ellos es Git Explorer en el cual podemos ver los comandos que podemos usar por terminal/console seleccionando la acción que queremos ejecutar.
<https://gitexplorer.com/>

