
ANALYZE
DEVELOPMENT
DESIGN
IMPLEMENT

Tema 3. INTRODUCCIÓN AL CONTROL DE VERSIONES

CASO PRÁCTICO

En *GoyoProg* toca empezar a diseñar la aplicación, realizar el análisis, programar, pero… de qué forma pueden acceder todos a la misma documentación, al mismo código desde cada uno de sus ordenadores. Esta parte es posible gracias a un repositorio remoto, de esta forma vamos a tener siempre una copia de seguridad dentro del servidor, además de un control de versiones. Utilizaremos la herramienta GIT, con GitHub y metodología GitFlow.

1. Concepto de control de Versiones

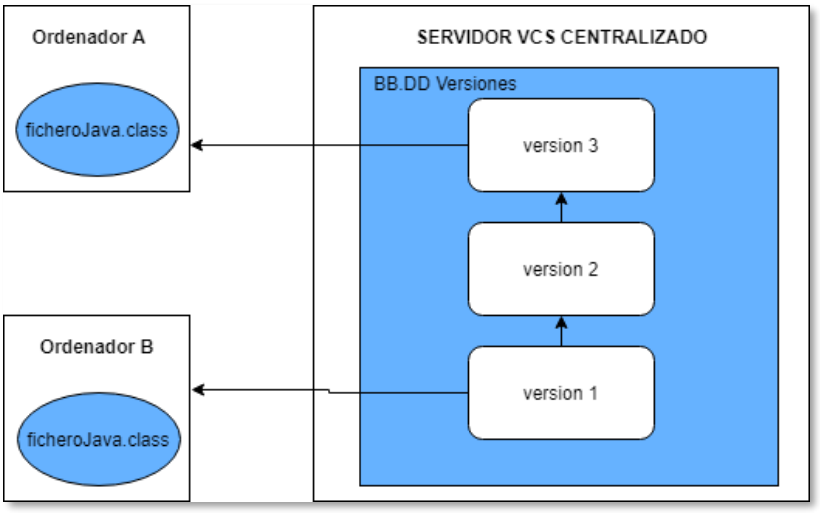
¿Qué es un control de versiones, y por qué es tan importante? Un **control de versiones** es un sistema que **registra los cambios realizados en un archivo o conjunto de archivos a lo largo del tiempo**, de modo que **es posible recuperar versiones específicas de ese fichero** más adelante.

Si eres diseñador gráfico o de web y quieres mantener cada versión de una imagen, usar un sistema de control de versiones (**VCS** por sus siglas en inglés) es una decisión muy acertada. Dicho sistema te permite volver a restaurar versiones anteriores de tus archivos, regresar a una versión anterior del proyecto completo, comparar cambios a lo largo del tiempo, ver quién modificó por última vez algo que pueda estar causando problemas, ver quién introdujo un problema y cuándo, y mucho más. Usar un VCS también significa generalmente que, si pierdes archivos, será posible recuperarlos fácilmente. Adicionalmente, obtendrás todos estos beneficios a un coste muy bajo.

1.1 Tipos de Software de Control de versiones: Centralizado y Descentralizado.

1.1.1 Sistemas de Control de Versiones Centralizados

Los **sistemas de control de versiones centralizados**, 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. Este ha sido el estándar para el control de versiones por muchos años.



Ventajas:

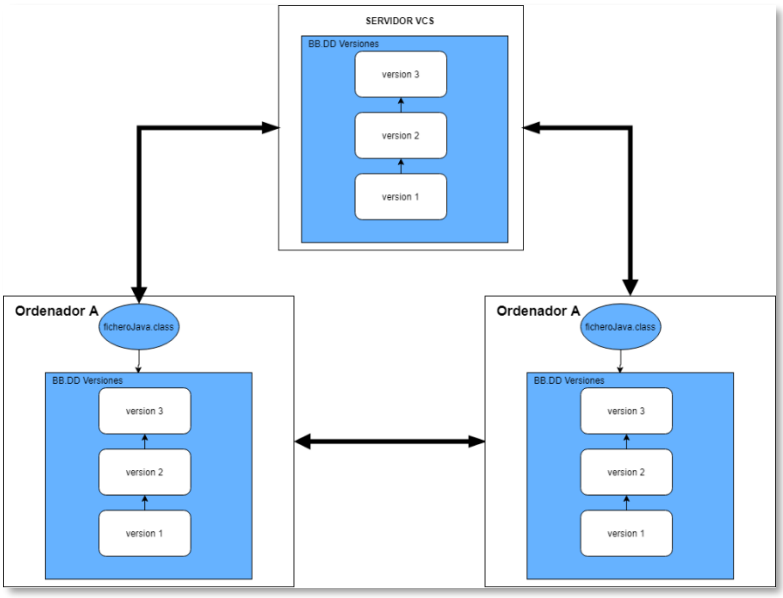
- ✓ Todos los desarrolladores saben hasta cierto punto en qué están trabajando los otros colaboradores del proyecto.
- ✓ Los administradores tienen control detallado sobre qué puede hacer cada usuario, y es mucho más fácil administrar un CVCS que tener que lidiar con bases de datos locales en cada cliente.

Desventajas:

- ✓ Punto único de fallo que representa el servidor centralizado. Si ese servidor se cae durante una hora, entonces durante esa hora nadie podrá colaborar o guardar cambios en archivos en los que hayan estado trabajando.
- ✓ Si el disco duro en el que se encuentra la base de datos central se corrompe, y no se han realizado copias de seguridad adecuadamente, se perderá toda la información del proyecto, con excepción de las copias instantáneas que las personas tengan en sus máquinas locales.

1.1.2 Sistemas de Control de Versiones Distribuidos

Los **sistemas de Control de Versiones Distribuidos** (***DVCS*** por sus siglas en inglés) ofrecen soluciones para los problemas que han sido anteriormente nombrados. En un *DVCS* (como Git, Mercurial, …), los clientes no solo **descargan la última copia instantánea de los archivos**, sino que **se replica completamente el repositorio**. De esta manera, si un servidor deja de funcionar y estos sistemas estaban colaborando a través de él, cualquiera de los repositorios disponibles en los clientes puede ser copiado al servidor con el fin de restaurarlo. **Cada clon es realmente una copia completa de todos los datos.**



Además, muchos de estos sistemas se encargan de manejar numerosos repositorios remotos con los cuales pueden trabajar, de tal forma que puedes colaborar simultáneamente con diferentes grupos de personas en distintas maneras dentro del mismo proyecto.

2. Sobre GIT

El *kernel* de Linux es un proyecto de software de código abierto con un alcance bastante amplio. Durante la mayor parte del mantenimiento del *kernel* de *Linux* (1991-2002), los cambios en el software se realizaban a través de parches y archivos. En el 2002, el proyecto del *kernel* de Linux empezó a usar un **DVCS** propietario llamado ***BitKeeper***.

En el 2005, la relación entre la comunidad que desarrollaba el *kernel* de *Linux* y la compañía que desarrollaba *BitKeeper* desapareció y la herramienta dejó de ser ofrecida de manera gratuita. 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 mientras usaban *BitKeeper*. Algunos de los objetivos del nuevo sistema fueron los siguientes:

- ✓ Velocidad
- ✓ Diseño sencillo
- ✓ Gran soporte para desarrollo no lineal (miles de ramas paralelas)
- ✓ Completamente distribuido
- ✓ Capaz de manejar grandes proyectos (como el *kernel* de Linux) eficientemente (velocidad y tamaño de los datos)

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

2.1. Cómo funciona GIT

La mayoría de las operaciones en **Git** sólo **necesitan archivos y recursos locales para funcionar**. Por lo general **no se necesita información de ningún otro ordenador de tu red.** A diferencia de otros CVCS donde la mayoría de las operaciones tienen el coste adicional del retardo de la red, este aspecto de Git no tiene ese retardo, debido a que **tienes toda la historia del proyecto en tu disco local**, la mayoría de las operaciones parecen prácticamente inmediatas.

Por ejemplo, para navegar por la historia del proyecto, Git no necesita conectarse al servidor para obtener la historia - simplemente la **lee directamente de la base de datos local**. Esto significa que **ves la historia del proyecto casi instantáneamente**. Si quieres ver los cambios introducidos en un archivo entre la versión actual y la de hace un mes, Git puede buscar el archivo de hace un mes y hacer un cálculo de diferencias localmente, en lugar de tener que pedirle a un servidor remoto que lo haga, u obtener una versión antigua desde la red y hacerlo de manera local.

Esto también significa que hay muy poco que no puedes hacer si estás desconectado o sin VPN. Si te subes a un avión o a un tren y quieres trabajar un poco, puedes confirmar tus cambios felizmente hasta que consigas una conexión de red para subirlos.

La principal diferencia entre Git y cualquier otro VCS (incluyendo Subversion y sus amigos) es la forma en la que manejan sus datos. Conceptualmente, **la mayoría de los otros sistemas** almacenan la información como una lista de cambios en los archivos. Estos sistemas (CVS, Subversion, Perforce, Bazaar, etc.) **manejan la información que almacenan como un conjunto de archivos y las modificaciones hechas a cada uno de ellos a través del tiempo.**

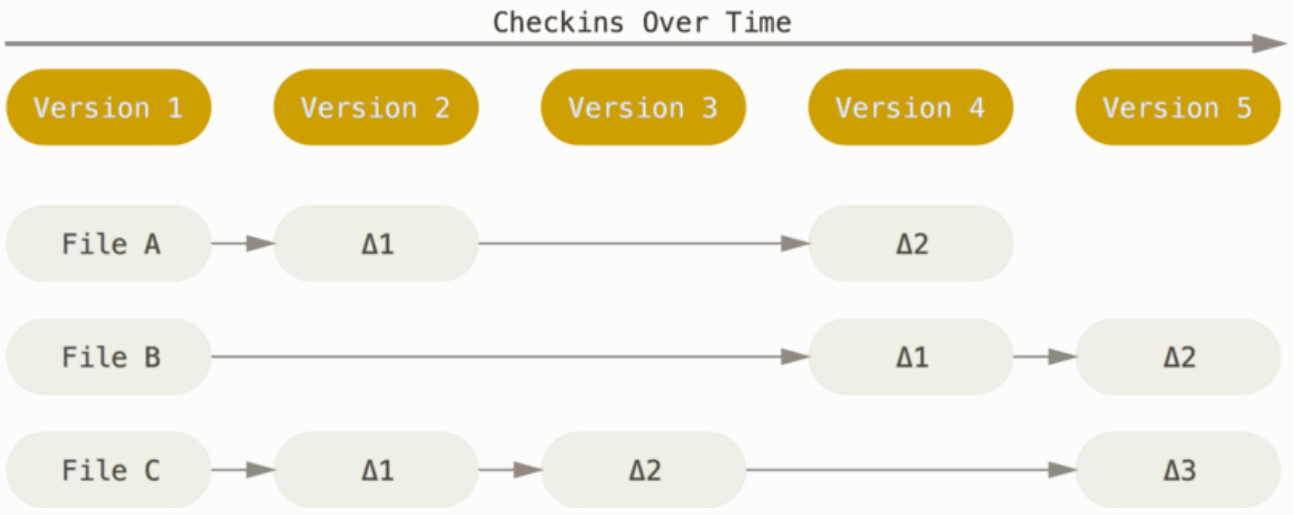


Figura 4. Almacenamiento de datos como cambios en una versión de la base de cada archivo.

Git no maneja ni almacena sus datos de esta forma. **Git maneja sus datos como un conjunto de copias instantáneas de un sistema de archivos miniatura.** Cada vez que confirmas un cambio, o guardas el estado

de tu proyecto en Git, él básicamente toma una foto del aspecto de todos tus archivos en ese momento y guarda una referencia a esa copia instantánea. Para ser eficiente, si los archivos no se han modificado Git no almacena el archivo de nuevo, sino un enlace al archivo anterior idéntico que ya tiene almacenado. Git maneja sus datos como una secuencia de copias instantáneas.

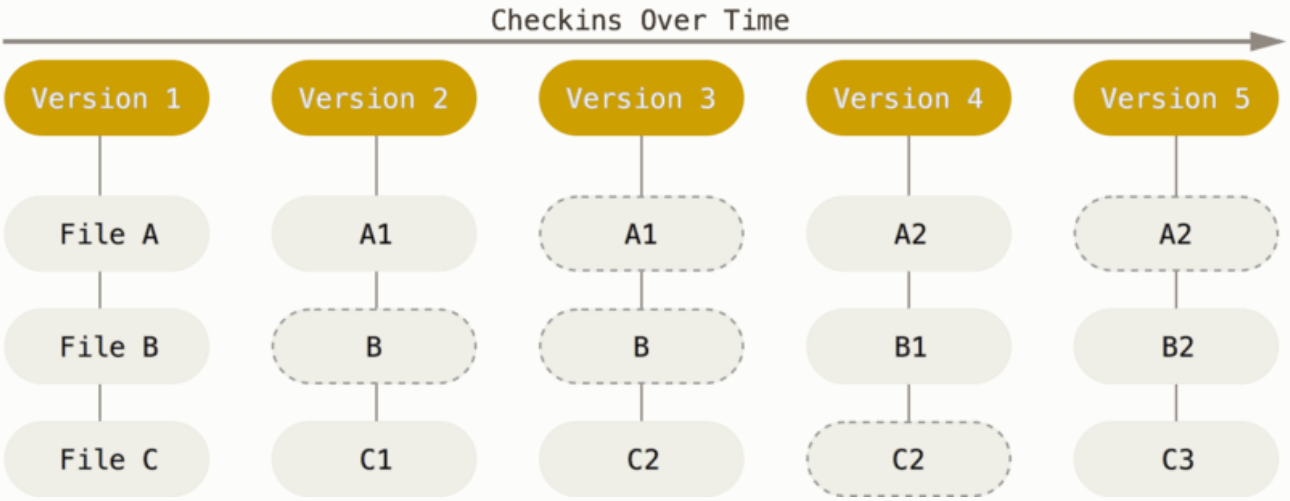


Figura 5. Almacenamiento de datos como instantáneas del proyecto a través del tiempo.

Esta es una diferencia importante entre Git y prácticamente todos los demás VCS. Hace que Git reconsidere casi todos los aspectos del control de versiones que muchos de los demás sistemas copiaron de la generación anterior. Esto hace que Git se parezca más a un sistema de archivos miniatura con algunas herramientas tremendamente poderosas desarrolladas sobre él, que a un VCS. Exploraremos algunos de los beneficios que obtienes al modelar tus datos de esta manera cuando veamos ramificación (branching) en Git en el (véase [ch03-git-branching]).

2.2. Integridad

Dentro de Git, **todo es verificado mediante una suma de comprobación (*checksum* en inglés) antes de ser almacenado, y es identificado a partir de ese momento mediante dicha suma. Esto significa que es imposible cambiar los contenidos de cualquier archivo o directorio sin que Git lo sepa. Esta funcionalidad está integrada en Git al más bajo nivel y es parte integral de su filosofía. **No puedes perder información durante su transmisión o sufrir corrupción de archivos sin que Git sea capaz de detectarlo.**

El mecanismo que usa Git para generar esta suma de comprobación se conoce como **hash SHA-1.** Se trata de una cadena de 40 caracteres hexadecimales (0-9 y a-f), y se calcula con base en los contenidos del archivo o estructura del directorio en Git. Un hash *SHA-1* se ve de la siguiente forma:

```
24b9da6552252987aa493b52f8696cd6d3b00373
```

Estos valores hash se utilizan por todos lados en Git, porque son usados con mucha frecuencia. De hecho, **Git guarda todo no por nombre de archivo, sino por el valor hash de sus contenidos.**

2.3. 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**).

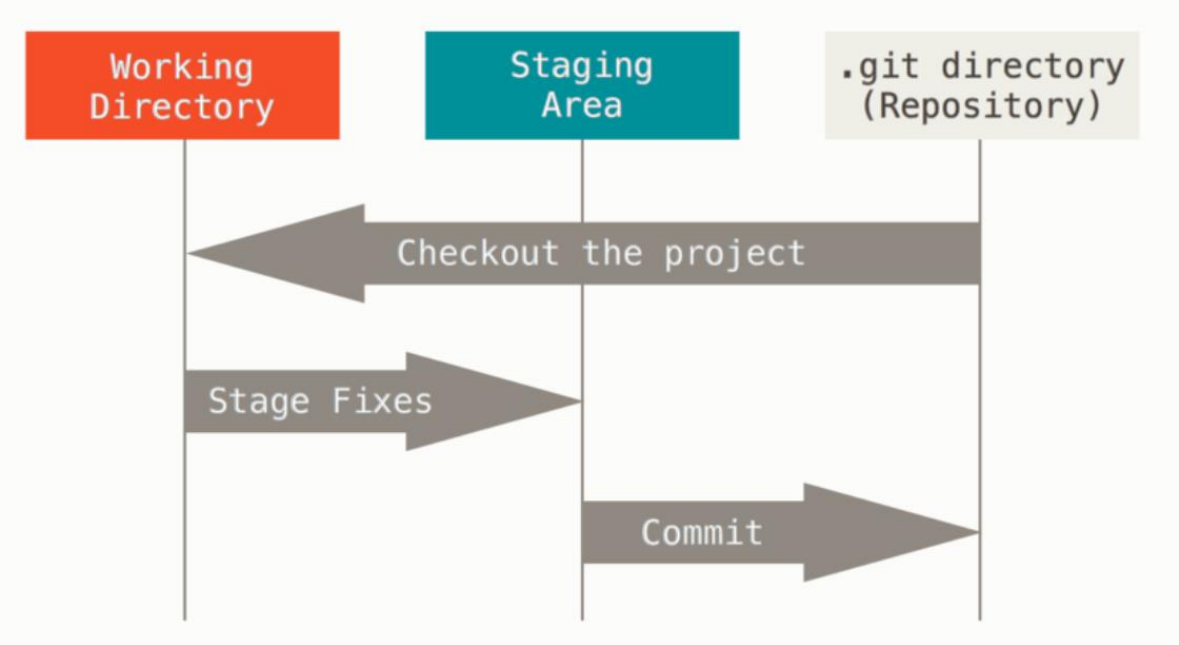
- ✓ **Modificado**: significa que **has modificado el archivo**, pero todavía **no lo has confirmado a tu base de datos.**
- ✓ **Preparado**: significa que **has marcado un archivo modificado en su versión actual para que vaya en tu próxima confirmación.**
- ✓ **Confirmado**: significa **que los datos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

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***).

- ✓ El directorio de Git es donde **se almacenan los metadatos y la base de datos de objetos para tu proyecto.** Es la parte más importante de Git, y es lo que **se copia cuando clonas un repositorio desde otro ordenador o servidor.**
- ✓ El directorio de trabajo es una **copia de una versión del proyecto**. Estos archivos se sacan de la base de datos comprimida en el directorio de Git, y se colocan en disco para que los puedas usar o modificar.
- ✓ El área de preparación es **un archivo**, generalmente contenido en tu directorio de Git, que **almacena información acerca de lo que va a ir en tu próxima confirmación**.

El flujo de trabajo básico en Git funciona de la siguiente forma:

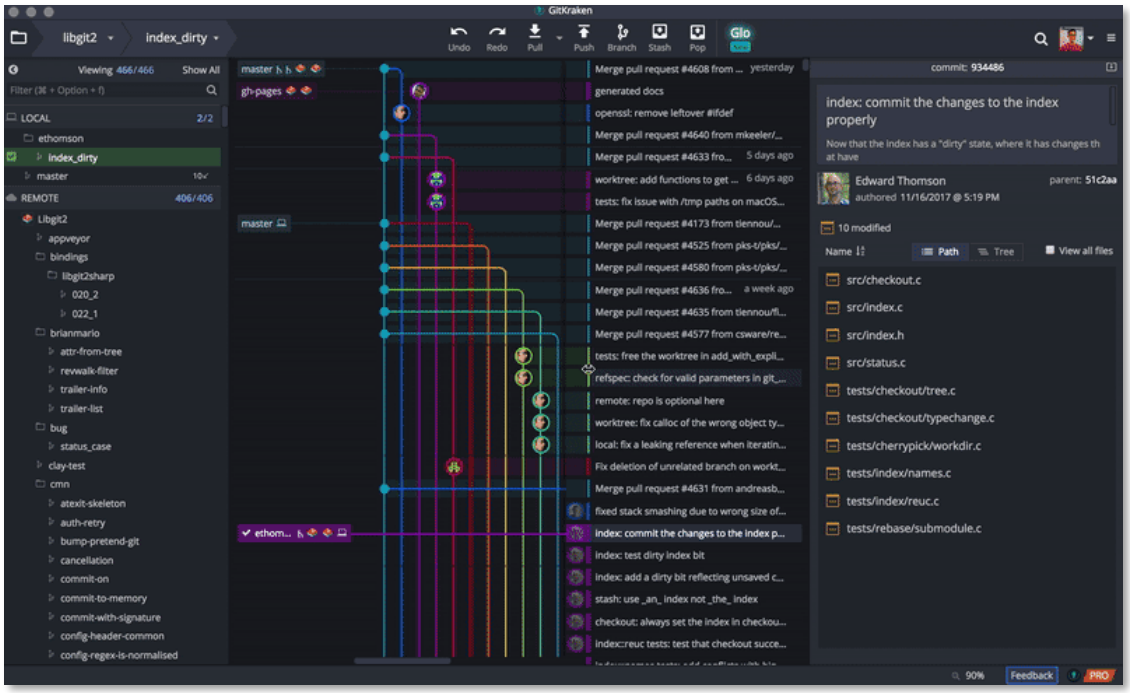
1. Modificas una serie de archivos en tu directorio de trabajo.
2. Preparas los archivos, añadiéndolos a tu área de preparación.
3. Confirmas los cambios, lo que toma los archivos tal y como están en el área de preparación y almacena esa copia instantánea de manera permanente en tu directorio de Git.
4. Si una versión concreta de un archivo está en el directorio de Git, se considera confirmada (*committed*). Si ha sufrido cambios desde que se obtuvo del repositorio, pero ha sido añadida al área de preparación, está preparada (*staged*). Y si ha sufrido cambios desde que se obtuvo del repositorio, pero no se ha preparado, está modificada (*modified*).



2.4 Aplicaciones Gráficas en GIT

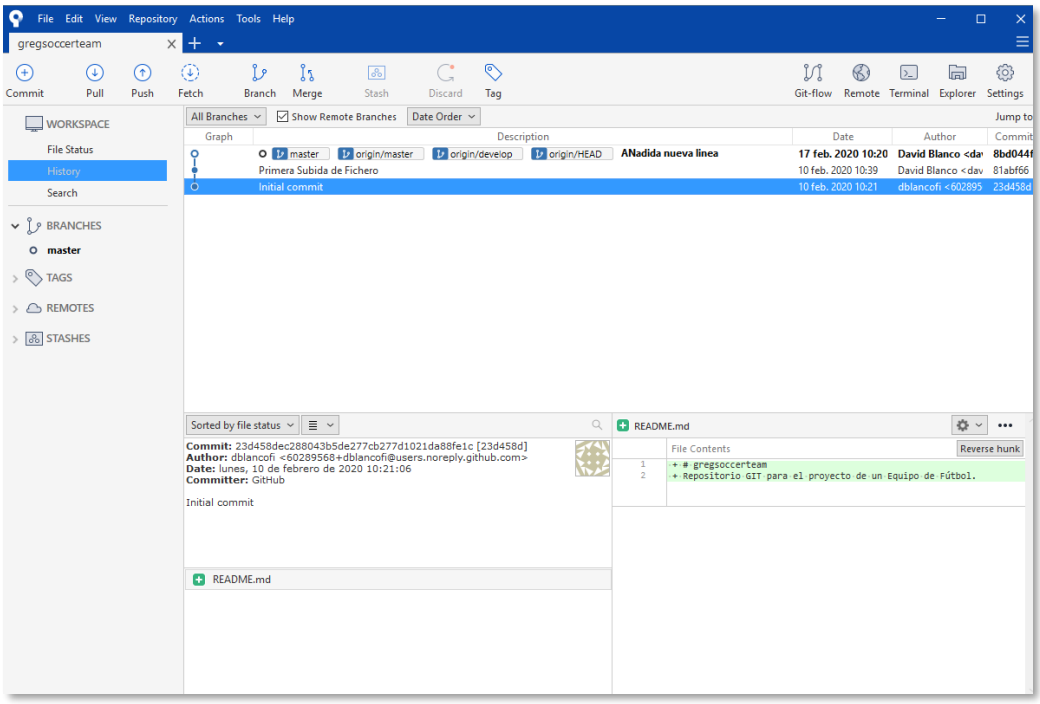
Dentro de las explicaciones sobre el uso de GIT, vamos a aprender cómo utilizar la herramienta de control de versiones mediante comandos pero también es posible el uso de herramienta de entorno gráfico como son las siguientes.

Git Kraken.
https://www.gitkraken.com/

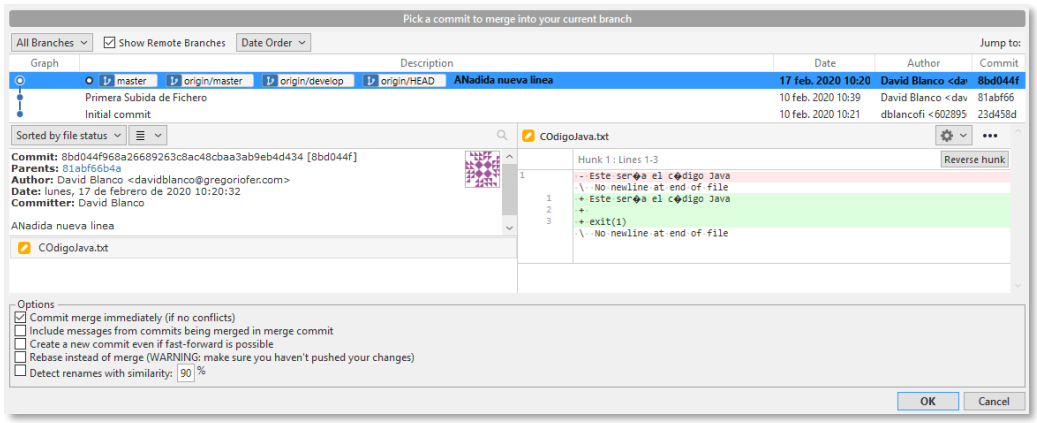


Vista del Dashboard del cliente Git Kraken

SourceTree
https://www.sourcetreeapp.com/



Vista Principal de Repositorio dentro de la herramienta *SourceTree*.



Vista de detalle en la herramienta *SourceTree*.

2.5 Configuración inicial de Git

La primera vez que abramos Git en nuestro dispositivo, nos encontraremos con que no se ha configurado la mayoría de las funciones en relación a nuestra propia identidad, siendo **necesaria para un correcto orden de nuestro proyecto, sobre todo en el trabajo con otras personas**. Para ello, tendremos que configurar nuestro nombre de usuario con el comando **git config -- global user.name "nombre"** y el correo que nos identificará con **git config --global user.mail "mail".** Las comillas en el nombre nos permiten tener un espacio en el Nombre y así poder el Apellido si lo deseamos.

```
$ git config --global user.name "nombre"
$ git config --global user.email "correo"
```

```
rhtuf@LAPTOP-HGE11LLJ MINGW64 ~/proyecto (main)
$ git config --global user.name "Ruth"

rhtuf@LAPTOP-HGE11LLJ MINGW64 ~/proyecto (main)
$ git config --global user.mail "ruthfernandez@gregoriofer.com"
```

Si queremos cambiar el editor, en caso de no haberlo hecho en la configuración inicial, podremos utilizar el comando git config --global core.editor (Editor). Cuando queramos, o para comprobar que se ha configurado correctamente, podremos hacer git config --list, donde tendremos un vistazo rápido.

```
rhtuf@LAPTOP-HGE11LLJ MINGW64 ~/proyecto (main)
$ git config --list
diff.astextplain.textconv=astextplain
filter.lfs.clean=git-lfs clean -- %f
filter.lfs.smudge=git-lfs smudge -- %f
filter.lfs.process=git-lfs filter-process
filter.lfs.required=true
http.sslbackend=openssl
http.sslcainfo=C:/Program Files/Git/mingw64/ssl/certs/ca-bundle.crt
core.autocrlf=true
core.fscache=true
core.symlinks=true
pull.rebase=false
credential.helper=manager-core
credential.https://dev.azure.com.usehttppath=true
init.defaultbranch=main
user.name=Ruth
user.mail=ruthfernandez@gregoriofer.com
core.repositoryformatversion=0
core.filemode=false
core.bare=false
core.logallrefupdates=true
core.ignorecase=true
```

2.6 Creación de repositorios

2.6.1 Inicializando un repositorio en un directorio existente

Mediante la ejecución del siguiente comando, dentro del directorio de un proyecto existente, logramos comenzar el proceso de enlace de nuestro proyecto con git.

```
$ git init
```

Esto **crea un subdirectorio** nuevo llamado .**git,** el cual contiene todos los archivos necesarios del repositorio , un esqueleto de un repositorio de Git. Todavía no hay nada en tu proyecto que esté bajo seguimiento.

```
rhtuf@LAPTOP-HGE11LLJ MINGW64 ~
$ mkdir proyecto

rhtuf@LAPTOP-HGE11LLJ MINGW64 ~
$ cd proyecto

rhtuf@LAPTOP-HGE11LLJ MINGW64 ~/proyecto
$ git init
Initialized empty Git repository in C:/Users/rhtuf/proyecto/.git/
```

Si queremos empezar a controlar versiones de archivos existentes (a diferencia de un directorio vacío), debemos comenzar el seguimiento de esos archivos y hacer una confirmación inicial. Empezaremos con los siguientes comandos:

- ✓ '**git add**' para especificar qué archivos queremos añadir.
- ✓ '**git commit'** para confirmar los cambios añadidos.

```
$ git add *.java (añadimos todos los archivos con extension .java)

$ git commit -m 'Primera subida'
```

```
rhtuf@LAPTOP-HGE11LLJ MINGW64 ~/proyecto (main)
$ git add *.java
```

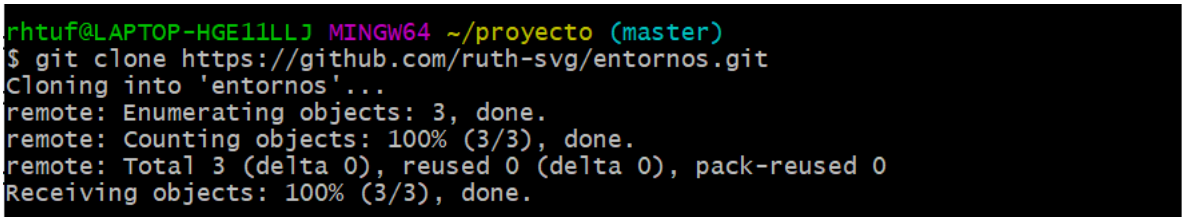
```
rhtuf@LAPTOP-HGE11LLJ MINGW64 ~/proyecto (main)
$ git commit -m "Primera subida"
```

2.6.2 Clonando un repositorio existente

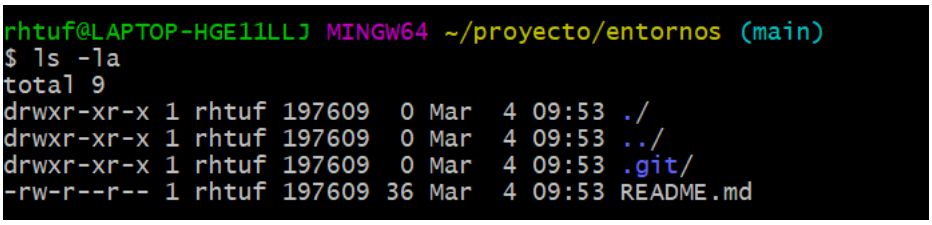
Si queremos obtener una copia de un repositorio Git existente — por ejemplo, un proyecto en el que te gustaría contribuir — el comando que necesitamos es **'git clone'**. Este comando se corresponde con los de otros sistemas de control de versiones como *Subversion*, "*checkout*". Es una distinción importante, ya que **Git recibe una copia de casi todos los datos que tiene el servidor**. Cada versión de cada archivo de la historia del proyecto es descargada por defecto cuando ejecutas 'git clone'. De hecho, si el disco de tu servidor se corrompe, puedes usar cualquiera de los clones en cualquiera de los clientes para devolver el servidor al estado en el que estaba cuando fue clonado.

Ej. Clonar un repositorio con *git clone [url].*

```
$ git clone https://github.com/ruth-svg/entornos.git
```



Esto crea un directorio llamado *entornos*, inicializa un directorio .git en su interior, descarga toda la información de ese repositorio y saca una copia de trabajo de la última versión. Si te metes en el directorio *entornos*, verás que están los archivos del proyecto listos para ser utilizados.



Si quieres clonar el repositorio a un directorio con otro nombre que no sea '*entornos*', puedes especificarlo con la siguiente opción de línea de comandos:

```
$ git clone https://github.com/ruth-svg/entornos.git miProyecto
```

Ese comando hace lo mismo que el anterior, pero el directorio de destino se llamará *miProyecto*.

Git te permite usar distintos protocolos de transferencia. El ejemplo anterior usa el protocolo https://, pero también puedes utilizar git:// o usuario@servidor:ruta/del/repositorio.git que utiliza el protocolo de transferencia SSH.

2.7 Otros comandos Importantes.

Iniciando el repositorio

```
$ git init
o
$ git clone <url del repositorio>
```

Commits

Añadir cambios

```
$ git add <archivo.*>
```

Comprobar cambios

```
$ git status
```

Commit

```
$ git commit -m "mensaje "
```

Ver los últimos Commit

```
$ git log
```

Volver a una versión anterior de un fichero

```
$ git checkout <número en cuestión>
```

Subir cambios al repositorio remoto

```
$ git push origin main
```

Descargar última versión del código del repositorio remoto

```
$ git pull origin
```

Ver diferencias entre un fichero local y su fichero remoto homólogo

```
$ git diff <nombrefichero>
```

Guardar progreso del repositorio pero no commitear (Cuando hay dudas de que algo esté correcto)

```
$ git stash
```

Revertir un commit produciendo uno nuevo con los cambios inversos.

```
$ git revert <commit>
```

Descartar los cambios locales en tu directorio

```
$ git reset -hard HEAD
```

Ayuda de git en cualquier comando con -h

2.7.1 Ejemplo manejo comandos Git

2.7.1.1 Local

1.Crea un repositorio en local

```
rhtuf@LAPTOP-HGE11LLJ MINGW64 ~/entornos/eshopgregiorio (main)
$ git init
Initialized empty Git repository in C:/Users/rhtuf/entornos/eshopgregiorio/.git/
```

2.Configuración inicial de Git

```
rhtuf@LAPTOP-HGE11LLJ MINGW64 ~/entornos/eshopgregiorio (main)
$ git config --global user.name "Ruth"

rhtuf@LAPTOP-HGE11LLJ MINGW64 ~/entornos/eshopgregiorio (main)
$ git config --global user.mail "ruthfernandez@gregoriofer.com"
```

3.Creamos el fichero README.md

```
rhtuf@LAPTOP-HGE11LLJ MINGW64 ~/entornos/eshopgregiorio (main)
$ echo "# eshopgregorio" >> README.md
```

4.Confirma la primera subida a la base de datos local

```
rhtuf@LAPTOP-HGE11LLJ MINGW64 ~/entornos/eshopgregiorio (main)
$ git commit -m "first commit"
```

5.Indicar la rama main como principal

```
rhtuf@LAPTOP-HGE11LLJ MINGW64 ~/entornos/eshopgregiorio (main)
$ git branch -M main
```

6.Añadir a remoto en nuevo repositorio

```
rhtuf@LAPTOP-HGE11LLJ MINGW64 ~/entornos/eshopgregiorio (main)
$ git remote add origin https://github.com/ruth-svg/eshopgregorio.git
```

7. Subir a la rama main

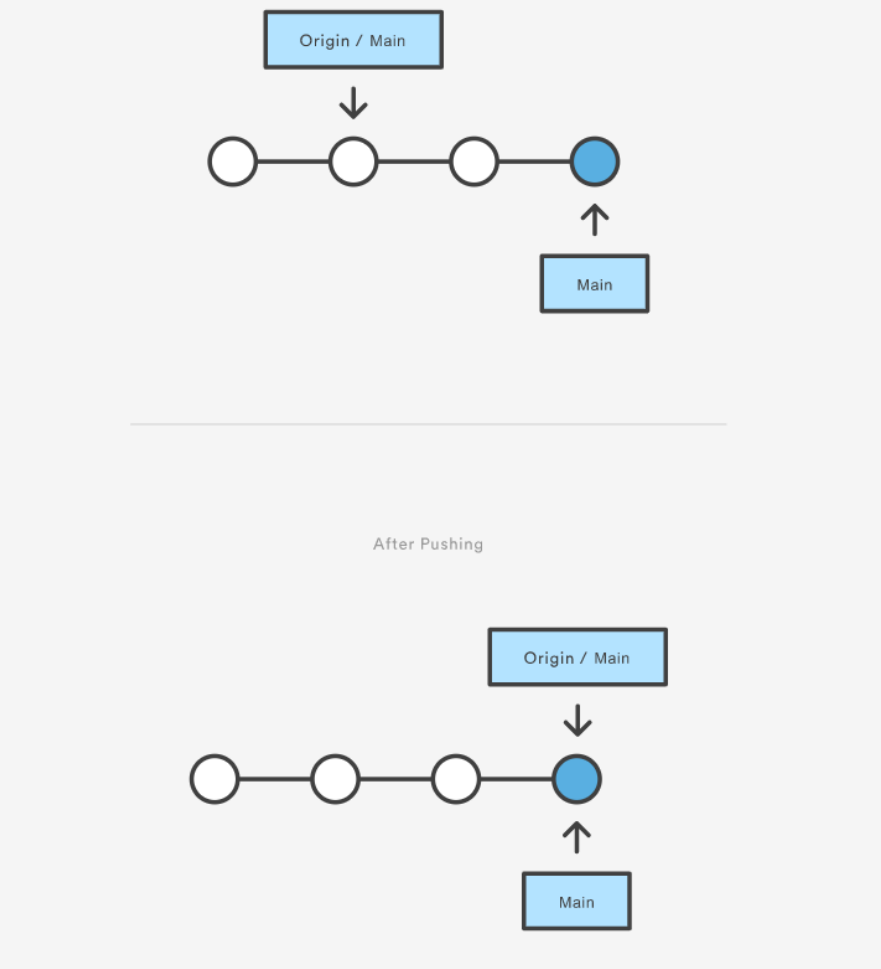
```
rhtuf@LAPTOP-HGE11LLJ MINGW64 ~/entornos/eshopgregiorio (main)
$ git push -u origin main
```

8. Subir a remoto el repositorio local

```
rhtuf@LAPTOP-HGE11LLJ MINGW64 ~ (main)
$ git remote add origin https://github.com/ruth-svg/eshopgregorio.git
```

10. Subir a remoto main

```
rhtuf@LAPTOP-HGE11LLJ MINGW64 ~ (main)
$ git push -u origin main
```



2.7.1.2 Remoto

1. Clonar repositorio remoto

```
rhtuf@LAPTOP-HGE11LLJ MINGW64 ~/entornos (main)
$ git clone https://github.com/ruth-svg/eshopgf.git
Cloning into 'eshopgf'...
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (3/3), done.
```

```
rhtuf@LAPTOP-HGE11LLJ MINGW64 ~/entornos (main)
$ cd eshopgf

rhtuf@LAPTOP-HGE11LLJ MINGW64 ~/entornos/eshopgf (main)
$ ls
README.md
```

2. Añadir archivo prueba.txt y guardar en repositorio local

```
rhtuf@LAPTOP-HGE11LLJ MINGW64 ~/entornos (main)
$ echo "Prueba desde local">prueba.txt

rhtuf@LAPTOP-HGE11LLJ MINGW64 ~/entornos (main)
$ git add prueba.txt
```

```
rhtuf@LAPTOP-HGE11LLJ MINGW64 ~/entornos (main)
$ git commit -m "Primera subida a remoto"
```

3. Subir cambios a repositorio remoto a la rama principal

```
rhtuf@LAPTOP-HGE11LLJ MINGW64 ~/entornos (main)
$ git push origin main
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 8 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 326 bytes | 326.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/ruth-svg/entornos.git
   7ea0926..9359a63  main -> main
```

4. Realizo cambios en el archivo prueba.txt desde el repositorio remoto y actualizo los cambios en el repositorio local

```
rhtuf@LAPTOP-HGE11LLJ MINGW64 ~/entornos (main)
$ git pull origin
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 706 bytes | 50.00 KiB/s, done.
From https://github.com/ruth-svg/entornos
   9359a63..794e409  main       -> origin/main
Updating 9359a63..794e409
Fast-forward
 prueba.txt | 1 +
 1 file changed, 1 insertion(+)
```

5. Si no quiero realizar los cambios automáticamente y actualizarlos con el repositorio local. Se puede bajar los cambios y fusionarlos de manera manual

```
rhtuf@LAPTOP-HGE11LLJ MINGW64 ~/entornos (main)
$ git fetch
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 720 bytes | 55.00 KiB/s, done.
From https://github.com/ruth-svg/entornos
   794e409..b4b2131  main       -> origin/main
```

```
rhtuf@LAPTOP-HGE11LLJ MINGW64 ~/entornos (main)
$ git merge origin/main
Updating 794e409..b4b2131
Fast-forward
 prueba.txt | 1 +
 1 file changed, 1 insertion(+)
```

6. Compruebo que no hay cambios que realizar

```
rhtuf@LAPTOP-HGE11LLJ MINGW64 ~/entornos (main)
$ git status
On branch main
Your branch is up to date with 'origin/main'.

nothing to commit, working tree clean
```

7. Comprueba los cambios que se han efectuados, junto con los mensajes de commit

```
rhtuf@LAPTOP-HGE11LLJ MINGW64 ~/entornos (main)
$ git log
commit 794e409bd4c0e9fa47153c2b850e880938fe0866 (HEAD -> main, origin/main, ori
in/HEAD)
Author: ruth-svg <79632499+ruth-svg@users.noreply.github.com>
Date:   Tue Mar 9 09:28:45 2021 +0100

    Update prueba.txt

commit 9359a63f455331a51a2179fd5be70f95855d2b96
Author: RuthGF <ruthfernandez@gregoriofer.com>
Date:   Tue Mar 9 09:06:00 2021 +0100

    Prueba subida remoto

commit 7ea092685c31e133c67b00a8c9973134c35e67b0
Author: ruth-svg <79632499+ruth-svg@users.noreply.github.com>
Date:   Thu Mar 4 09:45:42 2021 +0100

    Initial commit
```