

GIT

CLASE 1 : QUÉ ES GIT ?

INTRODUCCIÓN A GIT

QUÉ ES Y PARA QUÉ NOS SIRVE GIT?

GIT es un software de control de versiones diseñado por Linus Torvalds, pensando en la eficiencia y la confiabilidad del mantenimiento de versiones de aplicaciones cuando éstas tienen un gran número de archivos de código fuente.

Su propósito es llevar registro de los cambios en archivos de computadora y coordinar el trabajo que varias personas realizan sobre archivos compartidos.

Pero qué es todo esto de **Control de Versiones**?

Imaginemos que nos despertamos un día y tenemos que comenzar con un proyecto nuevo. Este nuevo proyecto consiste en sólo un archivo fuente. Todos los cambios que le podamos hacer a ese archivo durante el día, lo más probable es que puedan ser deshechos. Al día siguiente continuamos con el desarrollo de nuestro proyecto y mejoramos nuestro archivo pero en el intento nuestro programa empieza a presentar errores. Necesitamos volver nuestro proyecto a una versión anterior, donde no había errores pero ahora estamos restringidos a solo poder deshacer los cambios hechos durante el día.

GIT entonces nos ayuda con este aspecto del desarrollo. Nos ofrece herramientas para poder gestionar cada una de las etapas y versiones por las que va transitando un proyecto de desarrollo. Por otro lado puede realizar, pero sin estar restringido a, **Desarrollo Colaborativo**.

QUÉ ES EL DESARROLLO COLABORATIVO?

El **Desarrollo colaborativo de software** es un modelo de desarrollo de software cuyas bases son la disponibilidad pública del código y la comunicación vía Internet. Este modelo se hizo popular a raíz de su uso para el desarrollo de Linux en 1991.

Teniendo como contexto a **GIT**, podríamos decir que el **Desarrollo Colaborativo** nos proporciona herramientas para poder desarrollar entre un gran número de individuos de una manera más fácil , menos propensa a errores y rápida de implementar. De esta manera siempre tenemos la opción de , por medio de algún cliente , publicar nuestro código junto con todas las etapas y versiones que nos llevó el proyecto para que otras personas puedan sumar y aportar nuevas ideas a nuestro repositorio.

PRIMEROS PASOS : DESCARGA E INSTALACIÓN DE GIT

Si puedes, en general es útil instalar **GIT** desde código fuente, porque obtendrás la versión más reciente. Cada versión de **GIT** tiende a incluir útiles mejoras en la interfaz de usuario, por lo que utilizar la última versión es a menudo el camino más adecuado si te sientes cómodo compilando software desde código fuente. También ocurre que muchas distribuciones de Linux contienen paquetes muy antiguos; así que a menos que estés en una distribución muy actualizada o estés usando backports, instalar desde código fuente puede ser la mejor opción.

Para instalar **GIT**, necesitas tener las siguientes librerías de las que **GIT** depende: curl, zlib, openssl, expat y libiconv.

Para comenzar con la descarga en un sistema operativo Windows, de cualquier manera, simplemente basta con descargar el programa instalador ejecutable desde el siguiente link :

```
https://gitforwindows.org/
```

Una guía de instalación detallada con pasos a seguir y que opciones elegir durante la instalación puede ser encontrada en la sección de descargas del Alumni. Instalaciones para otros Sistemas Operativos pueden ser encontradas desde el siguiente link :

```
https://git-scm.com/book/es/v1/Empezando-Instalando-Git
```

Tengamos en cuenta también que podemos hacer uso de alguna herramienta gráfica que nos agilice el proceso de uso y/o aprendizaje como la que podemos descargar desde la página de GitHub :

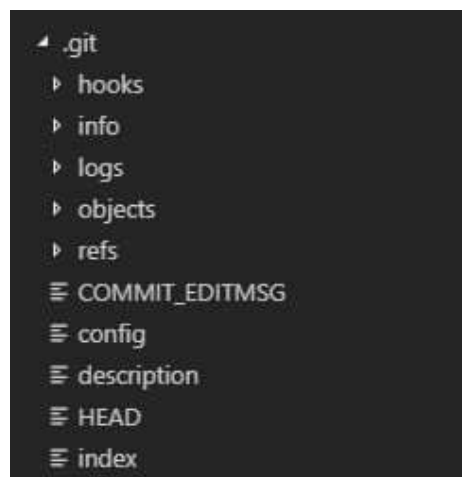
```
https://desktop.github.com/
```

Una guía detallada de descarga e instalación puede ser encontrada también en la sección de descargas de esta clase en el mismo Alumni.

QUÉ ES UN REPOSITORIO?

Un repositorio **GIT** es la carpeta `.git` / dentro de un proyecto. Este repositorio rastrea todos los cambios realizados en los archivos de su proyecto, creando un historial a lo largo del tiempo. Es decir, si elimina la carpeta `.git` /, entonces elimina el historial de su proyecto.

Un directorio `.git` tiene una estructura similar a la siguiente :



- **objects/**

En este directorio se almacenan los datos de sus objetos **GIT**: todo el contenido de los archivos que haya registrado, sus commits, branches y tags.

- **objects/[0-9a-f][0-9a-f]**

Un objeto recién creado se almacena en su propio archivo. Los objetos se colocan en subdirectorios 256 utilizando los dos primeros caracteres del nombre de objeto SHA1 para mantener el número de entradas de directorio en los objetos en un número manejable. Los objetos que se encuentran aquí a menudo se denominan objetos desempaquetados o sueltos.

- **objects/pack**

Los archivos que almacenan muchos objetos en forma comprimida, junto con los archivos de índice para permitir el acceso aleatorio, se encuentran en este directorio.

- **objects/info**

La información adicional sobre el objeto almacenado se coloca en este directorio.

- **refs**

Las referencias se almacenan en los subdirectorios de este directorio. El comando git prune sabe que debe preservar los objetos accesibles a partir de las referencias que se encuentran en este directorio y sus subdirectorios.

- **refs/heads/**

Contiene objetos de commit.

- **refs/tags**

Contiene cualquier nombre de objeto.

- **refs/remotes**

Contiene los objetos de confirmación de las ramas copiadas desde un repositorio remoto.

- **Archivo HEAD**

Este archivo contiene una referencia al branch en la que se encuentra actualmente. Esto le dice a **GIT** qué usar como padre de su próximo commit.

- **archivo de configuración config**

Este es el archivo de configuración principal de **GIT**. Mantiene opciones específicas de **GIT** para su proyecto, como sus controles remotos, configuraciones de inserción, branches de seguimiento y más. Su configuración se cargará primero desde este archivo, luego desde un archivo `~ / .gitconfig` y luego desde un archivo `/ etc / gitconfig`, si existe.

- **hooks/**

Este directorio contiene scripts de shell que se invocan después de los comandos **GIT** correspondientes. Por ejemplo, después de ejecutar un commit, **GIT** intentará ejecutar el script posterior a la confirmación.

- **archivo de índice index**

El index **GIT** se utiliza como Stage Area entre su Working Directory y su repositorio. Puede utilizar el index para crear un conjunto de cambios que desea confirmar juntos. Cuando crea un commit, lo que se confirma es lo que está actualmente en el index, no lo que está en su directorio de trabajo. Es un archivo binario que contiene una lista ordenada de nombres de ruta, cada uno con permisos y el SHA-1 de un objeto blob.

- **info/**

La información adicional sobre el repositorio se registra en este directorio.

- **logs/**

Almacena los cambios realizados en las referencias en el repositorio.

- **logs/refs/heads/**

Registra todos los cambios realizados en las diferentes puntas de rama.

- **logs/refs/tags/**

Registra todos los cambios realizados en las diferentes etiquetas.

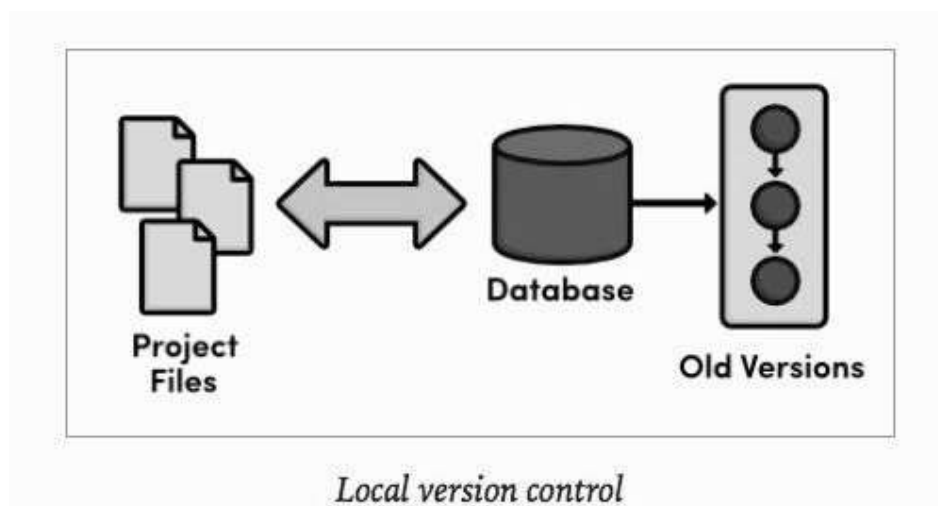
- **modules/**

Contiene los repositorios **GIT** de los submódulos.

Cuando realiza acciones en **GIT**, casi todas solo agregan datos a la base de datos de **GIT**. Es muy difícil lograr que el sistema haga algo que no pueda deshacerse o hacer que borre los datos de alguna manera. Como en cualquier SCV, puede perder o alterar los cambios que aún no ha confirmado pero después de confirmar un snapshot en **GIT**, es muy difícil perderlo, especialmente si realiza push regularmente de su base de datos a otro repositorio.

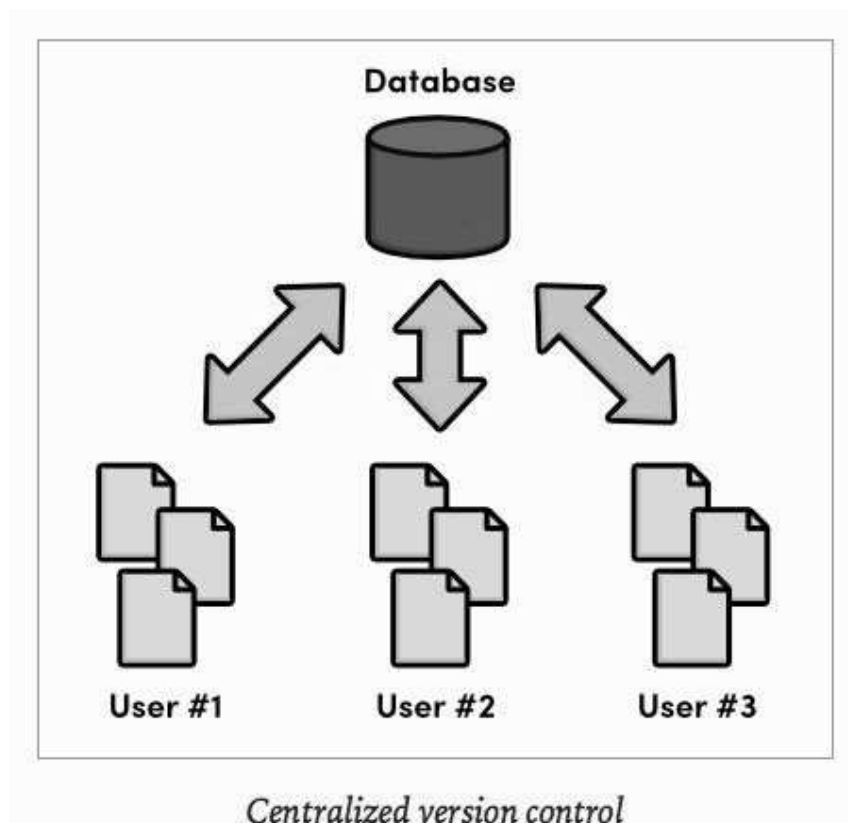
REPOSITORIO LOCALES VS DISTRIBUIDO VS CENTRALIZADO

- **LOCALES** : Estrictamente hablando, este tipo de control de versiones si bien puede tener una API flexible , es el menos conveniente ya que solo se presenta en la computadora de cada desarrollador con lo cual no tenemos la posibilidad de compartir nuestro código en caso que quisiéramos hacer **Desarrollo Colaborativo**.



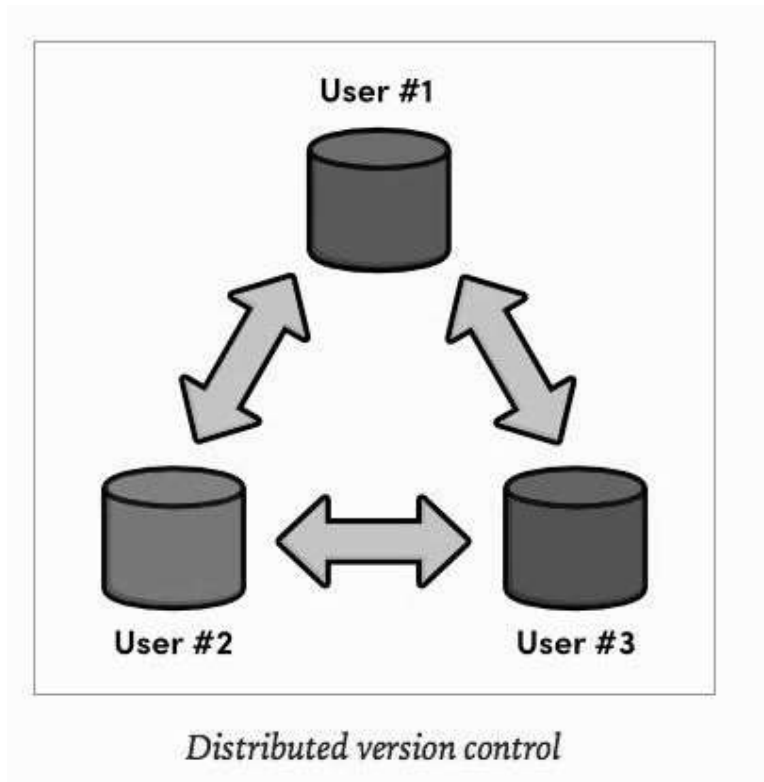
- **CENTRALIZADOS** : Para facilitar la colaboración de múltiples desarrolladores en un solo proyecto los sistemas de control de versiones evolucionaron: en vez de almacenar los cambios y versiones en el disco duro de los desarrolladores, estos se almacenaban en un servidor. Sin embargo, aunque el avance frente a los sistemas de control de versiones locales fue enorme, los sistemas centralizados trajeron consigo nuevos retos: ¿Cómo trabajaban múltiples usuarios en un mismo archivo al mismo tiempo?

Por otro lado , los sistemas centralizados solían usar como fuente de verdad un repositorio almacenado en algún servidor, con lo cual si perdíamos conexión a internet o al servidor no íbamos a poder seguir trabajando.



- **DISTRIBUIDOS** : La siguiente generación de sistemas de control de versiones se alejó de la idea de un solo repositorio centralizado y optó por darle a cada desarrollador una copia local de todo el proyecto, de esta manera se construyó una red distribuida de repositorios, en la que cada desarrollador podía trabajar de manera

aislada pero teniendo un mecanismo de resolución de conflictos mucho más elegante que un su versión anterior.



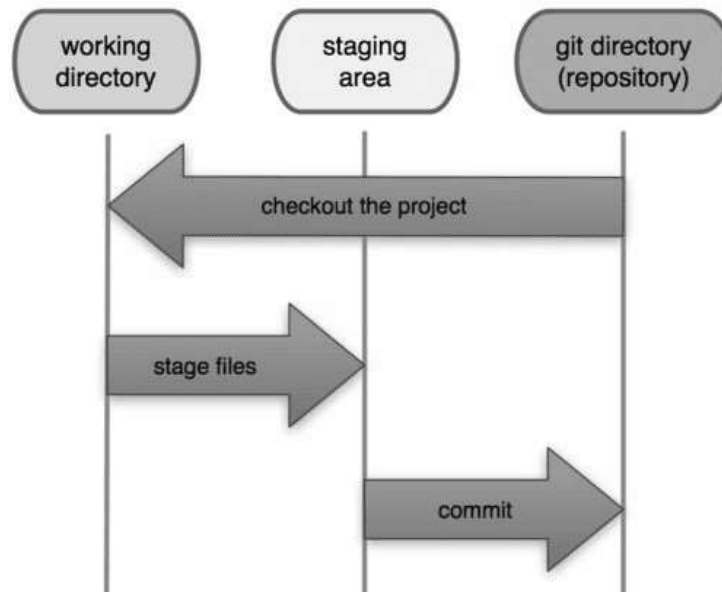
ESTADOS DE UN REPOSITORIO GIT

Esto es lo más importante que debe recordar acerca de **GIT** si desea que el resto de su proceso de aprendizaje se realice sin problemas.

GIT tiene tres estados principales en los que sus archivos pueden residir: **committed**, **modified** y **staged**. **Committed** significa que los datos se almacenan de forma segura en su base de datos local. **Modified** significa que ha cambiado el archivo pero aún no lo ha confirmado en su base de datos. **Staged** significa que ha marcado un archivo modificado en su versión actual para pasar a su próximo snapshot de confirmación.

Esto nos lleva a las tres secciones principales de un proyecto **GIT**: el directorio `.git`, el `working directory` y el `stage area` :

Local Operations



INFORMACIÓN TÉCNICA

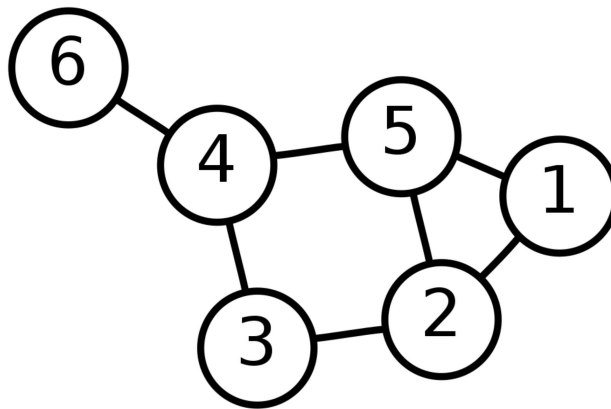
QUÉ ES UN GRAFO?

En matemáticas y ciencias de la computación, un grafo (del griego grafos: dibujo, imagen) es un conjunto de objetos llamados vértices o nodos unidos por enlaces llamados aristas o arcos, que permiten representar relaciones binarias entre elementos de un conjunto.¹ Son objeto de estudio de la teoría de grafos.

Típicamente, un grafo se representa gráficamente como un conjunto de puntos (vértices o nodos) unidos por líneas (aristas).

Desde un punto de vista práctico, los grafos permiten estudiar las interrelaciones entre unidades que interactúan unas con otras. Por ejemplo, una red de computadoras puede representarse y estudiarse mediante un grafo, en el cual los vértices representan terminales y las aristas representan conexiones (las cuales, a su vez, pueden ser cables o conexiones inalámbricas).

Prácticamente cualquier problema puede representarse mediante un grafo, y su estudio trasciende a las diversas áreas de las ciencias exactas y las ciencias sociales.



Grafo etiquetado con 6 vértices y 7 aristas.

Un grafo G es un par ordenado $G(V,N)$, donde:

- V es un conjunto de vértices o nodos, y
- N es un conjunto de aristas o arcos, que relacionan estos nodos.

TEORÍA DE GRAFOS

Formalmente, un grafo $G=(V,E)$ es una pareja ordenada en la que V es un conjunto no vacío de vértices y E es un conjunto de aristas. Donde E consta de pares no ordenados de vértices, tales como $\{x,y\}$ in E entonces se dice que x e y son adyacentes; y en el grafo se representa mediante una línea no orientada que una dichos vértices. Si el grafo es dirigido se le llama dígrafo, se denota D , y entonces el par (x,y) es un par ordenado, esto se representa con una flecha que va de x a y y se dice que (x,y) in E .

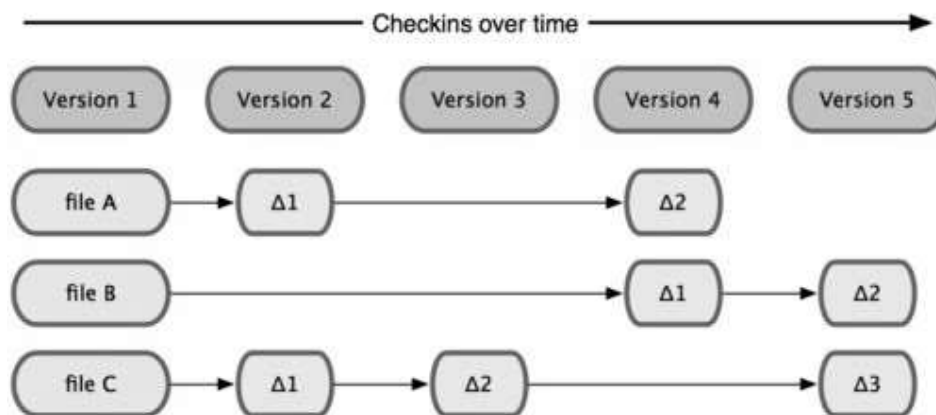
Un grafo está compuesto por :

- **Aristas:** Son las líneas con las que se unen los vértices de un grafo.
 - **Aristas adyacentes:** 2 aristas son adyacentes si convergen en el mismo vértice.
 - **Aristas paralelas:** Son dos aristas conjuntas si el vértice inicial y final son el mismo.
 - **Arista cíclicas:** Es la arista que parte de un vértice para entrar en sí mismo.
 - **Cruce:** Son 2 aristas que cruzan en un mismo punto.

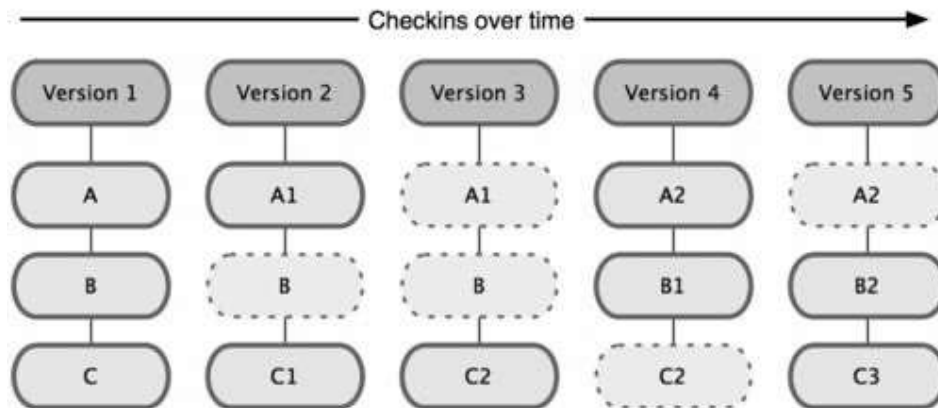
- **Vértices:** Los vértices son los elementos que forman un grafo. Cada uno lleva asociada una valencia característica según la situación, que se corresponde con la cantidad de aristas que confluyen en dicho vértice.
- **Camino:** Se denomina camino de un grafo a un conjunto de vértices interconectados por aristas. Dos vértices están conectados si hay un camino entre ellos.

RELACIÓN DE GRAFOS CON GIT

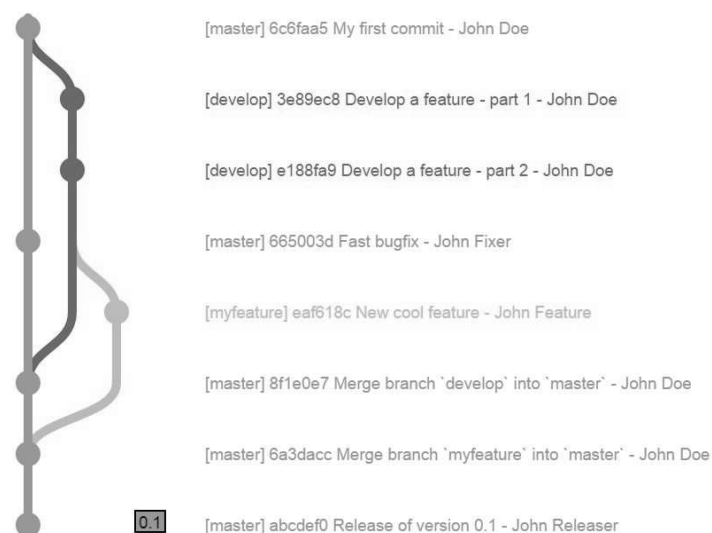
La principal diferencia entre **GIT** y cualquier otro SCV (Ej.: Subversion) es la forma en que **GIT** piensa acerca de sus datos. Conceptualmente, la mayoría de los otros sistemas almacenan información como una lista de cambios basados en archivos. Estos sistemas (CVS, Subversion, Perforce, Bazaar, etc.) piensan en la información que mantienen como un conjunto de archivos y los cambios realizados en cada archivo a lo largo del tiempo.



GIT no piensa ni almacena sus datos de esta manera. En cambio, **GIT** piensa en sus datos más como un conjunto de snapshots (instantáneas) de un mini sistema de archivos. Cada vez que confirma o guarda el estado de su proyecto en **GIT**, básicamente toma una fotografía de cómo se ven todos sus archivos en ese momento y almacena una referencia a esa instantánea. Para ser eficiente, si los archivos no han cambiado, **GIT** no almacena el archivo nuevamente, solo un enlace al archivo idéntico anterior que ya ha almacenado.



Esta es una distinción importante entre **GIT** y casi todos los demás SCV. Hace que **GIT** reconsidere casi todos los aspectos del control de versiones que la mayoría de los otros sistemas copian de la generación anterior y que además se parezca más a un mini sistema de archivos con algunas herramientas increíblemente poderosas construidas sobre él, en lugar de simplemente un SCV.



Consideramos entonces que la manera de representación de cada versión creada en nuestro repositorio corresponde a la de un sistema de grafos.

CONFIGURACIÓN INICIAL

Ahora que tenemos **GIT** instalado en nuestro Sistema Operativo vamos a querer hacer unas cuantas cosas para personalizar su entorno. Estas operaciones deberán ser realizadas una sola vez por computadora; van a quedar configuradas a través de actualizaciones. También podemos editarlas en cualquier momento usando los mismos comandos nuevamente.

GIT viene con una herramienta llamada `git config` la cual nos permite obtener y configurar variables que controlan aspectos de cómo **GIT** se va a ver y cómo va a operar. Estas variables pueden ser guardadas en tres lugares distintos:

1. `/etc/gitconfig` : Contiene los valores aplicados a todos los usuarios del Sistema Operativo y todos los repositorios. Si utilizamos la opción `--system` con `git config`, entonces vamos a estar leyendo y escribiendo desde este archivo específicamente. Dado que éste es un archivo de configuración del sistema, vamos a necesitar privilegios de superusuario o administrador para realizar cambios en él.
2. `~/.gitconfig` ó `~/.config/git/config` : Contiene valores específicos tuyos, del usuario. Podés hacer que **GIT** lea y escriba sobre este archivo específico usando la opción `--global` y estaremos afectando a todos los repositorios en los trabajemos en el sistema.
3. `./git/config` : Este archivo, el cual puede ser encontrado dentro del mismo directorio del repositorio donde estamos actualmente trabajando nos sirve para configurar específicamente un solo repositorio. Podemos forzar **GIT** para que lea y escriba sobre este archivo usando la opción `--local`, que de hecho es la opción por defecto.

Cada nivel anula los valores del nivel anterior, es decir que los valores en `.git/config` pesan más que aquellos en `/etc/gitconfig`.

En sistemas Windows, **GIT** busca por el archivo `.gitconfig` en el directorio `$HOME` (`C:\Users\USER` para la mayoría de las personas). También busca por `/etc/gitconfig`, que cual es relativo a la raíz de MSys, la cual está donde sea que hayan decidido instalar **GIT** cuando corremos la instalación. Si estás usando la versión 2.x o superior de **GIT** para Windows, hay también un archivo de configuración de nivel sistema en `C:\Documents and Settings\All Users\Application Data\Git\config` en Windows XP, y `C:\ProgramData\Git\config` en Windows Vista y superior. Este archivo de configuración puede ser cambiado únicamente por `git config -f <archivo>` como administrador.

También podemos ver todas las configuraciones y dónde se encuentra cada una usando :

```
> git config --list --show-origin
```

Lo primero que tenemos que hacer cuando instalamos **GIT** es configurar nuestro nombre de usuario y dirección de correo electrónico. Esto es importante porque cada commit de **GIT** usa esta información y es inmutablemente adherida a los commits que creemos.

NOMBRE DE USUARIO

```
> git config --global user.name "Mi Nombre de Usuario"
```

DIRECCIÓN DE CORREO ELECTRÓNICO

```
> git config --global user.email "mi.direccion.de.email@ejemplo.com"
```

CONFIGURACIÓN ADICIONAL

Ahora que ya tenemos nuestra identidad configurada, podemos configurar el editor que va a ser usado cuando **GIT** necesite que escribas un mensaje. Si no está configurado, **GIT** usa el editor que venga configurado por defecto en el sistema.

En un sistema Windows, si queremos usar un editor de texto diferente, tenemos que especificar la ruta completa del archivo ejecutable.

EDITOR DE TEXTO

```
> git config --global core.editor "'C:/Program  
Files/Notepad++/notepad++.exe'"
```

Para más información sobre cómo configurar distintos editores de texto, pueden acercarse al siguiente link :

```
https://git-scm.com/book/en/v2/Appendix-C%3A-Git-Commands-Setup-and-Config
```

VERIFICANDO CONFIGURACIONES

Si queremos verificar las configuraciones que hayamos hecho en nuestro sistema podemos usar el siguiente comando :

```
> git config --list
```

El mismo va a mostrarnos una lista de todas las variables que estén configuradas y que **GIT** pueda encontrar hasta este punto.

Es probable que se vean variables más de una vez, porque **GIT** lee la misma variable de diferentes archivos. En este caso, **GIT** usa el último valor por cada variable única que vea.

También podemos preguntarle a **GIT** cuál piensa que es el valor de una variable en particular con :

```
> git config user.name
```

PRIMEROS PASOS : NUESTRO PRIMER REPOSITORIO

Usualmente podemos obtener un repositorio de **GIT** de dos maneras :

- Podemos convertir un directorio local que actualmente no esté bajo ningún control de versión a un repositorio de **GIT**
- Podemos clonar un repositorio de **GIT** existente de algún otro lugar

De cualquier forma, vamos a terminar con un repositorio de **GIT** en nuestra máquina local listo para trabajar.

Si tenemos un directorio de proyecto que actualmente no esté bajo ningún control de versión y queremos empezar a controlarlo con **GIT**, primero tenemos que dirigirnos a ese directorio. Una vez ubicados en el directorio tenemos que correr el comando :

```
> git init
```

Esto va a crear un nuevo subdirectorío llamado `.git` que contiene todos los archivos necesarios para tu repositorio - el esqueleto de un repositorio de **GIT**.

CONFIRMANDO CAMBIOS

En este punto, nada en tu proyecto está siendo controlado aún. Si nuestro directorio está actualmente vacío porque acabamos de crearlo, vamos a entonces crear un nuevo archivo dentro de él llamado `info.txt`. Ahora podemos ejecutar el siguiente comando :

```
> git status
```

Y vamos a poder ver el archivo sin seguimiento de la siguiente manera :

```
> git status
On branch master
Your branch is up-to-date with 'origin/master'.
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    info.txt

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

Podemos ver que nuestro archivo `info.txt` no tiene seguimiento ya que se encuentra debajo del titular "Untracked files". Untracked básicamente significa que **GIT** ve un archivo que no tenía en la versión anterior del proyecto (snapshot). **GIT** no lo va a incluir hasta que no se lo digamos explícitamente.

EL COMANDO ADD : CREANDO SNAPSHOTS

Para incluir y darle seguimiento a un archivo podemos usar el siguiente comando :

```
> git add info.txt
```

Si volvemos a ejecutar el comando de status nuevamente vamos a ver que nuestro archivo está bajo seguimiento y listo para ser confirmado, es decir, formar parte de la nueva versión del repositorio(commit) :

```
> git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   info.txt
```

Podemos darnos cuenta que está listo porque está debajo del titular "Changes to be committed". Si generamos un commit en este punto, la versión del archivo en el momento en que hayamos ejecutado el comando git add va a permanecer en el snapshot histórico anterior.

El comando git add puede aceptar el nombre de ruta de un archivo o directorio; si es un directorio, el comando agrega todos los archivos en ese directorio recursivamente.

Si modificamos nuestro archivo info.txt que ya tenía seguimiento anteriormente y volvemos a ejecutar el comando git status vamos a ver algo similar a lo siguiente :

```
> git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   info.txt

Changes not staged for commit:
```



```
(use "git add <file>..." to update what will be committed)
(use "git checkout -- <file>..." to discard changes in working
directory)
```

```
modified:   info.txt
```

El archivo ahora aparece también bajo la sección llamada "Changes not staged for commit", lo cual significa que un archivo con seguimiento ha sido modificado en el directorio de trabajo (working directory) pero no ha sido preparado aún. Podemos preparar un archivo para el próximo commit utilizando también el comando git add.

Podemos entonces percibir que este comando es multipropósito ya que nos permite darle seguimiento a archivos nuevos y además preparar archivos que ya tenían seguimiento para el próximo commit.

Tengamos en cuenta que si realizamos un commit ahora mismo, la versión del archivo que formará parte del mismo va a ser la que estaba cuando ejecutamos el comando git add no cuando realicemos el commit, es decir que si modificamos un archivo luego de haber ejecutado el comando git add vamos a tener que ejecutarlo nuevamente para preparar la última versión.

Alternativamente podemos usar una versión interactiva del comando add

```
> git add -i
```

Esta otra forma nos va a dar mucho más detalle y control sobre los cambios que podemos realizar en un momento dado en nuestro repositorio. Para más información sobre cómo poder utilizar este comando, podemos dirigirnos al siguiente link :

```
https://git-scm.com/book/en/v2/Git-Tools-Interactive-Staging
```

EL COMANDO COMMIT : CONFIRMANDO SNAPSHOTS

Ahora que nuestro archivo se encuentra en el área de preparación(stage area), estamos listos para confirmar los cambios, es decir, realizar un commit. La manera mas corta es ejecutar el siguiente comando :

```
> git commit
```

Hacer esto va a lanzar nuestro editor elegido. El editor va a mostrar lo siguiente :

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Your branch is up-to-date with 'origin/master'.
#
# Changes to be committed:
#   new file:   info.txt
#   modified:   info.txt
#
~
~
~
".git/COMMIT_EDITMSG" 9L, 283C
```

Podemos ver que el mensaje por defecto del commit contiene la última salida del comando `git status` comentada y una línea vacía arriba de todo. Podemos remover estos comentarios y escribir nuestro mensaje o podemos dejarlos para ayudarnos a recordar lo que estamos confirmando en el commit. Cuando salimos del editor, **GIT** crea el commit con nuestro mensaje.

Alternativamente podemos escribir el mensaje del commit en la misma línea en la que estemos ejecutando el comando especificando la opción `-m` de la siguiente forma :

```
> git commit -m "Agrego cambios en el archivo"
[master 463dc4f] Agrego cambios en el archivo
1 file changed, 2 insertions(+)
create mode 100644 info.txt
```

Podemos observar que el comando nos ha dado información sobre sí mismo: la rama(branch) donde confirmamos los cambios (master), el identificador SHA-1 del commit (463dc4f), cuantos archivos fueron modificados, y estadísticas acerca de las líneas que fueron insertadas o removidas en el commit.

Cada vez que hagamos un commit vamos a estar generando una nueva versión del proyecto, ó snapshot, la cual podemos revertir o comparar luego.

Aunque el área de preparación puede ser útil para elaborar commits exactamente de la manera en que los queremos, muchas veces la misma puede ser muy compleja dado nuestro flujo de trabajo. Si queremos omitir el área de preparación, **GIT** viene con un atajo el cual nos permite automáticamente agregar todos los cambios realizados al área de preparación antes de realizar el commit, de esta manera podemos omitir el comando git add de la siguiente forma :

```
> git commit -a -m 'agrego nuevo cambio'
[master 83e38c7] agrego nuevo cambio
1 file changed, 5 insertions(+), 0 deletions(-)
```

REESCRITURA DEL COMMIT

Si simplemente lo que necesitamos es editar el mensaje del último commit que hayamos hecho porque hemos tenido un error, podemos hacerlo con el comando :

```
> git commit -amend
```

Este comando va a usar el mensaje del último commit que hayamos hecho y lo va a cargar en una sesión de edición en el editor que tengamos configurado en GIT en donde podemos hacer los cambios que queramos, guardarlos y salir. Cuando salgamos del editor, se va a escribir un nuevo commit el cual contiene nuestros cambios realizados.

Tengamos en cuenta que si ya estamos trabajando con repositorios remotos (lo cual vamos a ver más adelante en esta clase) y hemos subido el último commit a nuestro servidor distribuido, no debemos cambiar nada del último commit ya que al hacerlo estamos reemplazándolo con uno nuevo el cual va a diferir del historial de commits que aparece en nuestro repositorio distribuido en el servidor, por lo cual los demás integrantes del proyecto que estén participando van a tener problemas en actualizarse.

Para cambios más drásticos en nuestros commits, como por ejemplo cambiar el mensaje de commits anteriores al último, reordenamiento, borrado o integración de los mismos, necesitamos herramientas de depuración de historial las cuales vamos a ver más adelante en el curso.

REGISTRO DE CAMBIOS

Luego de haber creado varios commits, o mismo si hemos clonado un repositorio con un historial de commits existente, probablemente queramos mirar hacia atrás para ver qué ha pasado en el repositorio. La herramienta más básica y poderosa para hacer esto es el comando git log :

```
> git log
commit ca82a6dff817ec66f44342007202690a93763949
Author: Horacio Gutierrez<mi.direccion.email@ejemplo.com>
Date:   Mon Mar 17 21:52:11 2008 -0700
```

Cambio el número de versión

```
commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Horacio Gutierrez<mi.direccion.email@ejemplo.com>
Date:   Sat Mar 15 16:40:33 2008 -0700
```

Remuevo test innecesario

```
commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Horacio Gutierrez<mi.direccion.email@ejemplo.com>
Date:   Sat Mar 15 10:31:28 2008 -0700
```

Primer Commit

Por defecto, sin parámetros, git log lista los commits hechos en un repositorio en orden cronológico inverso; esto es, el commit más reciente se verá primero. Como podemos ver, este comando lista cada commit con su identificador SHA-1, el nombre de autor e email, la fecha de escritura y el mensaje del commit.

Este comando viene con muchos atajos y parámetros que podemos agregar para que la salida en la línea de comandos no sea tan abundante. Particularmente las opciones de `--oneline` y `--graph` son sumamente útiles para mostrar información abreviada sobre cada commit y para poder ver un gráfico ASCII mostrándonos nuestro historial de commits :

```
> git log --oneline --graph
* 2d3acf9 ignore errors from SIGCHLD on trap
* 5e3ee11 Merge branch 'master' of git://github.com/dustin/grit
```

```
| \
| * 420eac9 Added a method for getting the current branch.
* | 30e367c timeout code and tests
* | 5a09431 add timeout protection to grit
* | e1193f8 support for heads with slashes in them
| /
* d6016bc require time for xmlschema
* 11d191e Merge branch 'defunkt' into local
```

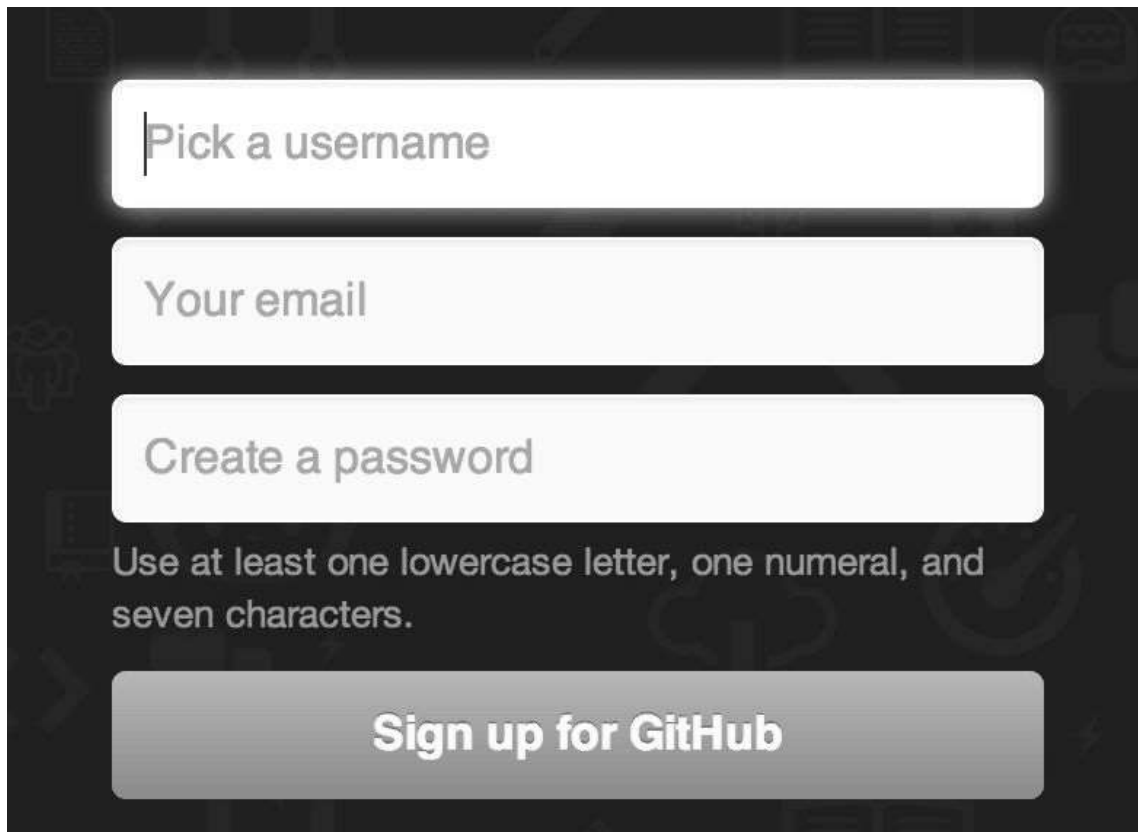
Este tipo de salida va a ser de mucha ayuda en las próximas clases cuando veamos branching y merging.

CLIENTES DE GIT

QUÉ ES GITHUB ?

GitHub es el host de almacenamiento de repositorios **GIT** mas grande y es el punto central de colaboración de millones de proyectos y desarrolladores. Un gran porcentaje de todos los repositorios **GIT** están alojados en GitHub, y muchos proyectos open-source lo usan como almacenamiento, registro de problemas(issue tracking), revisión de código, etc. GitHub no es parte central de **GIT** pero hay grandes probabilidades de que queramos interactuar con él en algún punto de nuestra vida profesional usando **GIT**.

Lo primero que vamos a necesitar es crearnos una cuenta gratuita. Simplemente visitando <https://github.com>, podemos elegir un nombre de usuario que no esté en uso actualmente, una dirección de correo electrónico y una contraseña y hacer click en el botón verde que dice "Sign up for GitHub" :

A screenshot of the GitHub sign-up form. It features three white input fields on a dark background. The first field is labeled 'Pick a username', the second 'Your email', and the third 'Create a password'. Below the password field, there is a text requirement: 'Use at least one lowercase letter, one numeral, and seven characters.' At the bottom, there is a large, rounded rectangular button with a gradient, labeled 'Sign up for GitHub'.

Pick a username

Your email

Create a password

Use at least one lowercase letter, one numeral, and seven characters.

Sign up for GitHub

QUÉ ES UN REMOTE ?

Un repositorio remoto son versiones de nuestros proyectos que están siendo almacenadas en la Internet o en algún otro lado. Podemos tener varios remotos, los cuales generalmente son de solo-lectura o lectura/escritura para nosotros.

Vamos a comenzar a trabajar con remotos hechos en GitHub, para lo cual necesitamos crear nuestro primer repositorio haciendo click donde veamos la opción de “+ New repository” :



Esto nos va a llevar al formulario de nuevos repositorios :

Owner **Repository name**

PUBLIC ben / iOSApp ✓

Great repository names are short and memorable. Need inspiration? How about **drunken-dubstep**.

Description (optional)

iOS project for our mobile group

☒ **Public**
Anyone can see this repository. You choose who can commit.

☐ **Private**
You choose who can see and commit to this repository.

☒ **Initialize this repository with a README**
This will allow you to `git clone` the repository immediately. Skip this step if you have already run `git init` locally.

Add .gitignore: **None** | Add a license: **None** ⓘ

Create repository

Todo lo que necesitamos es completar el nombre del proyecto; el resto de los campos son completamente opcionales. Por ahora vamos a hacer click en "Create Repository" para obtener así un repositorio en GitHub llamado <usuario>/<nombre_del_proyecto>.

Dado que no tenemos ningún código aún, GitHub nos va a mostrar instrucciones de cómo crear un nuevo repositorio de **GIT** o conectar con un repositorio existente.

Ahora que nuestro proyecto está almacenado en GitHub, podemos darle la URL a cualquier persona con la que queramos compartir nuestro trabajo. Cada proyecto en GitHub es accesible a través de HTTPS como

```
https://github.com/<usuario>/<nombre_del_proyecto>
```

O a través de SSH

```
git@github.com:<user>/<nombre_del_proyecto>
```

CONFIGURANDO REMOTES

Para agregar un nuevo remoto a nuestro repositorio **GIT** podemos ejecutar el siguiente comando :

```
> git remote add <alias>  
https://github.com/<usuario>/<nombre_del_proyecto>
```

Podemos notar que el comando `git remote add` nos permite configurar nuestro nuevo remoto con un nombre al cual podemos referenciar fácilmente y usarlo posteriormente para futuras operaciones sobre él.

Alternativamente podemos revisar el listado de remotos que hayamos configurado en nuestro repositorio **GIT** con el siguiente comando :

```
> git remote -v
```

EL COMANDO PUSH : SUBIENDO CAMBIOS

Cuando tenemos un proyecto que se encuentra en un punto donde ya queremos empezar a compartir nuestro trabajo con otros, podemos utilizar nuestros remotos para subir dichos cambios. Para hacer esto podemos ejecutar el comando :

```
> git push <remote> <branch>
```

Este comando va a funcionar solo si tenemos permiso de escritura en dicho cliente remoto y si nadie subió cambios mientras que realizábamos e intentábamos subir los nuestros. Si alguien más realizó un push antes que nosotros, nuestra operación será rechazada. Tenemos entonces que primero incorporar los cambios nuevos subidos por las demás personas antes de poder realizar un push.