Curso de Introducción a GIT

Sistema de control de Versiones

Un **sistema de control de versiones** VCS (del inglés *Version Control System*) nos permite gestionar los diversos cambios que se realizan sobre un software o sobre la configuración del mismo. Su principal función es registrar todos los cambios que se realizan a lo largo del tiempo sobre el conjunto de ficheros que conforman dicho software, de manera que siempre se pueda recuperar versiones específicas del mismo. Además los VCS más modernos nos posibilitan que varias personas apliquen cambios sobre un mismo software, proporcionando información de cuándo y quién ha hecho cada cambio.

Por lo que dando una definición lo más resumida posible sería:

Un **Sistema de control de versiones** nos permite gestionar los cambios en un grupo de ficheros a lo largo del tiempo, de manera que podemos recuperar versiones específicas de los mismos cuando queramos.

Tipos de Sistemas de Control de Versiones

Los podemos englobar en 3 grupos:

VCS Locales

Son los más sencillos, consiste en tener una copia del grupo de ficheros en otro directorio. Generalmente constan de una simple base de datos que registra todos los cambios de los ficheros.

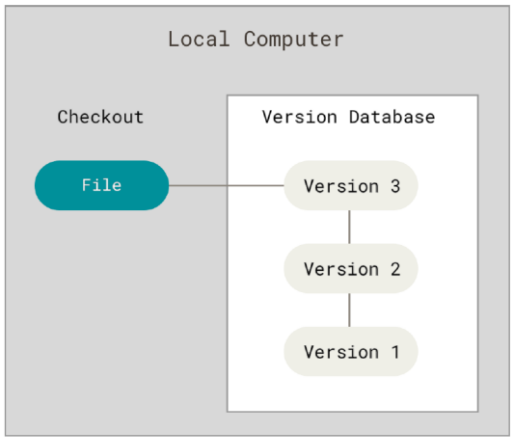


Figure 1. Local version control

VCS Centralizados

Uno de los principales problemas que surgieron rápidamente fue el hecho de que varios desarrolladores debían colaborar entre ellos en un mismo proyecto. Para solventar esto, surgieron los sistemas Centralizados que básicamente consisten en que toda la información de los ficheros versionados se almacenan en un único servidor central. Los clientes se descargan y suben los ficheros y sus cambios a dicho servidor central. Este ha sido durante muchos años el estándar para los sistemas de control de versiones.

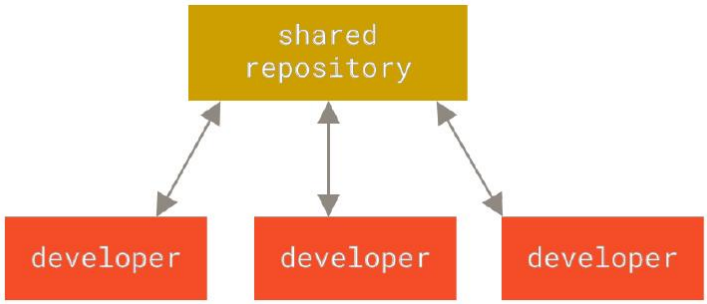


Figure 2. Centralized version control

Este modelo ofrece una serie de ventajas obvias respecto a los modelos locales (permite el desarrollo colaborativo), pero también tiene una serie de desventajas. Para empezar hay un único punto de fallo (el servidor central), por lo que si el servidor se encuentra fuera de servicio, nadie puede colaborar en el proyecto mientras dure el fallo. De la misma manera, si se corrompe el disco duro del servidor y no se han hecho copias de seguridad, todo el historial de cambios que no se haya guardado en las máquinas de los colaboradores se perderá para siempre. Piense incluso en un escenario más catastrófico, si por ejemplo se incendia una oficina dónde se encuentra el servidor central y todos los puestos fijos de los desarrolladores, ¡todo el código de los proyectos se perderían para siempre!

VCS Distribuidos

Esta es la siguiente evolución de los Sistemas de control. En este modelo ya no hay un único servidor central, si no que existen una serie de servidores remotos de los cuales cada cliente clona el proyecto de cualquiera de ellos, incluyendo todo el historial completo de cambios (cada clonado es en realidad una copia completa de todos los cambios).

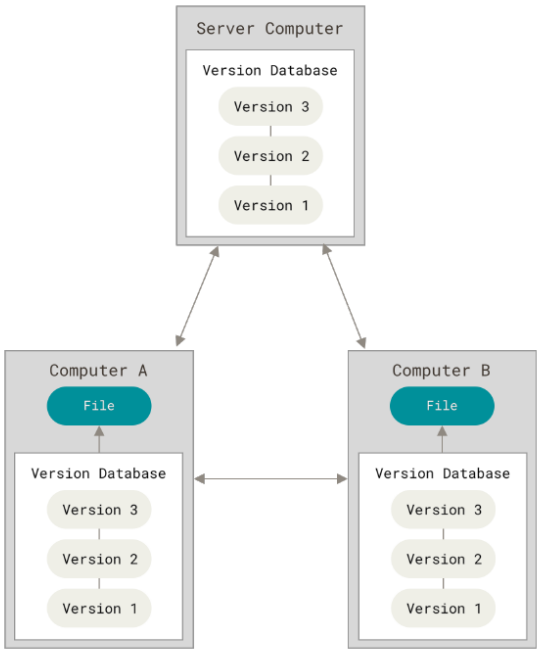


Figure 3. Distributed version control

Este nuevo modelo de múltiples servidores abre la puerta a establecer diferentes maneras de trabajar entre grupos de personas para un mismo proyecto. Por ejemplo se podrían crear diferentes modelos jerárquicos entre desarrolladores, algo que sería imposible en un sistema centralizado.

¿Cómo nace GIT?

Una breve historia de GIT

La historia comienza con el famoso proyecto de código abierto, el *Kernel* de Linux. El desarrollo del *Kernel* de Linux comienza a mediados de 1991 liderado por Linus Torvadls. Dicho software en su primera versión 0.01 tenía unas 10 mil líneas de código, pero para finales de 2010 el código había ascendido a más de 13 millones de líneas de código:

Evolución del código Fuente del *Kernel* de Linux			
Versión	Fecha	Nº líneas de código	Nº Ficheros
0.01	17 de septiembre de 1991	8.400	88
0.11	8 de diciembre de 1991	11.907	100
0.95	7 de marzo de 1992	19.200	122
1.0.0	13 de marzo de 1994	170.581	561
1.1.0	6 de abril de 1994	170.320	561
1.2.0	6 de marzo de 1995	294.623	909
1.3.0	12 de junio de 1995	323.581	992
2.0.0	9 de junio de 1996	716.119	2.015
2.1.0	30 de septiembre de 1996	735.736	1.727
2.2.0	26 de enero de 1999	1.676.182	4.599
2.3.0	11 de mayo de 1999	1.763.358	4.721
2.4.0	4 de enero de 2001	3.158.560	8.187
2.5.0	23 de noviembre de 2001	3.833.603	9.893
2.6.0	18 de diciembre de 2003	5.475.685	15.007
2.6.25	16 de abril de 2008	8.396.250	23.810
2.6.30	10 de junio de 2009	10.419.567	27.878
2.6.35	1 de agosto de 2010	12.250.679	33.315
3.0.0	22 de julio de 2011	13.688.408	36.782

Para dar a otra gente la capacidad de cooperar en el sistema o sugerir mejoras, los archivos fueron colocados en el servidor ftp (ftp.funet.fi) de la Universidad de Tecnología de Helsinki (Helsinki University of Technology) (HUT), en septiembre de 1991. Algunos desarrolladores que estaban interesados en el proyecto, contribuyen con mejoras y extensiones. Para 1993 más de 100 desarrolladores trabajan sobre el núcleo Linux.

Actualmente, miles de personas repartidas a lo largo del mundo han contribuido al núcleo Linux. Se estima que aproximadamente el 2 % del código del Linux actual está escrito por propio Torvalds.

Para coordinar dicho desarrollo, El proyecto Linux comenzó a utilizar en 2002 un Controlador de Versiones distribuido llamado BitKeeper.

Pero en 2005 la relación entre la comunidad de desarrolladores y la empresa encargada de desarrollar BitKeeper se rompió. Desgraciadamente no existía un software similar que cumpliese con los requisitos tan exigentes que el proyecto de Linux requería. Esto forzó a la comunidad de desarrolladores (y en particular al propio Linus Torvalds) a desarrollar su propia herramienta basándose en las lecciones aprendidas mientras usaban BitKeeper.

Dicha nueva herramienta debía seguir los siguientes objetivos:

- Rapidez.
- Diseño simple.
- Desarrollo no lineal (uno de los principales requisitos era el desarrollo de múltiples funcionalidades en paralelo).
- Completamente distribuido. Los miles de desarrolladores que participaban en el desarrollo del *kernel* de Linux estaban repartidos por mundo.
- Gestión eficiente de proyectos grandes, tanto por número de ficheros como en líneas de código.

Por lo que en 2005 nació GIT, un sistema de control de versiones distribuido, que se caracterizaba por cumplir dichos requisitos.

Qué es GIT

El primer concepto que hay que tener claro sobre GIT y que le diferencian de otros VCS es la manera en la que GIT gestiona sus datos. Muchos de estos sistemas alternativos (como podrían ser Subversion, Perforce, Bazaar y demás) almacenan información en forma de una lista de ficheros y sus modificaciones a lo largo del tiempo. Este enfoque es a menudo conocido como *delta-based version control*.

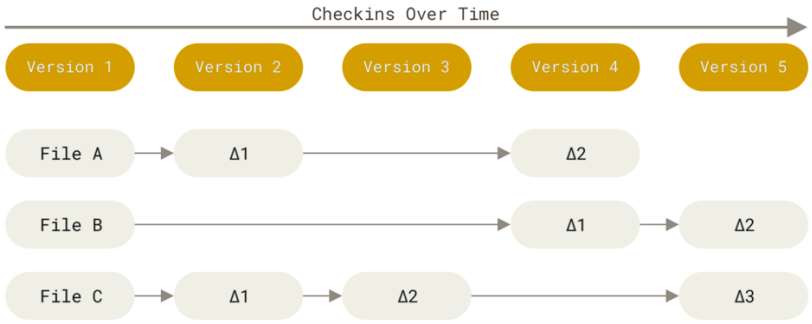


Figure 4. Storing data as changes to a base version of each file

Fotografías (Snapshots), no diferencias

Pero Git no piensa de la misma manera. Su enfoque básicamente consiste en que cada vez que se introduce un cambio nuevo, se saca una especie de "fotografía" (llamada *snapshot*) del contenido de los ficheros, guardándose una referencia a dicho *snapshot*. Por lo que Git trata el histórico de cambios como una especie de colección de *snapshots* de un conjunto de ficheros. Para ser eficiente, si un fichero no ha cambiado, Git no vuelve a guardar otra vez el mismo fichero, lo que hace es enlazarlo al de la versión anterior (ya que al no cambiar son idénticos).

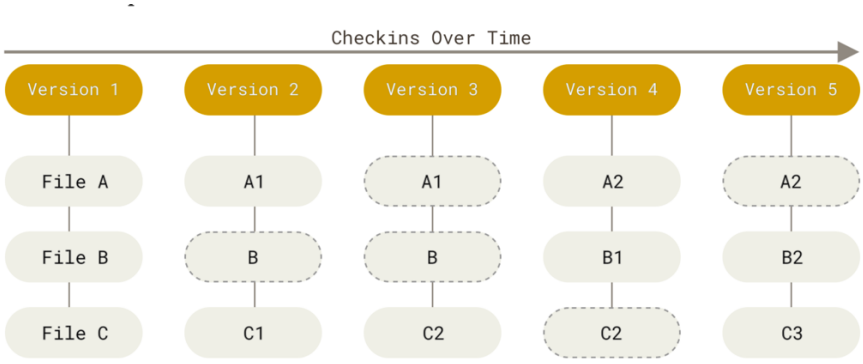


Figure 5. Storing data as snapshots of the project over time

Esto es una diferencia clave entre Git y el resto de los sistemas de control de versiones.

Las Operaciones sobre Git son locales

Otra diferencia importante es que cuando realizamos una operación en GIT no es necesario realizar ninguna llamada a otro ordenador de la red. Esto es debido a que en nuestra máquina tenemos una copia completa del historial del proyecto guardado en el disco. Por lo que cualquier operación es casi instantánea (en otros sistemas diferentes dichas operaciones necesitan de llamadas de red a un servidor central, lo que puede llevar a latencias y fallos). Por ejemplo, si queremos visualizar el histórico de cambios del proyecto a Git no le hace falta ir a buscar dichos cambios a un servidor, ya que simplemente los lee directamente en el disco. Si queremos ver el cambio que se produjo hace dos años, Git lo calcula localmente, mostrando dicha información instantáneamente en lugar de solicitar dicho calculo a un servidor que podría tener problemas de encontrarse no operativo, la red congestionada etc.

Esto además significa que puedes trabajar con Git incluso aunque estés desconectado de la red.

Integridad en Git

Hemos hablado de "cambios" sobre nuestros ficheros, pero ¿Cómo se da cuenta Git de que un cambio ha ocurrido en un fichero? Esto es posible gracias a que Git guarda un *checksum* de cada fichero y lo usa como referencia. Esto significa que es imposible hacer una modificación de un fichero sin que Git se entere. Esto también sirve para que, en caso de que accidentalmente el contenido de un fichero se corrompa, Git automáticamente lo detectará.

El mecanismo que tiene Git para calcular dicho checksum es mediante el hash SHA-1. Dicho hash es una cadena de 40 caracteres compuesto por caracteres hexadecimales (0-9 y a-f) y es calculado basándose en el contenido de dicho fichero.

```
AMAC02XJ55WJGH6:git d.a.somoza.paletta$ echo "esto es el contenido del fichero" > file1.txt
AMAC02XJ55WJGH6:git d.a.somoza.paletta$ shasum -a 1 file1.txt
ea152dfa38840d1ed092227f8244b9ffb6e99f01  file1.txt
AMAC02XJ55WJGH6:git d.a.somoza.paletta$
```

Un hash sha-1 tiene una apariencia como esta: **ea152dfa38840d1ed092227f8244b9ffb6e99f01**

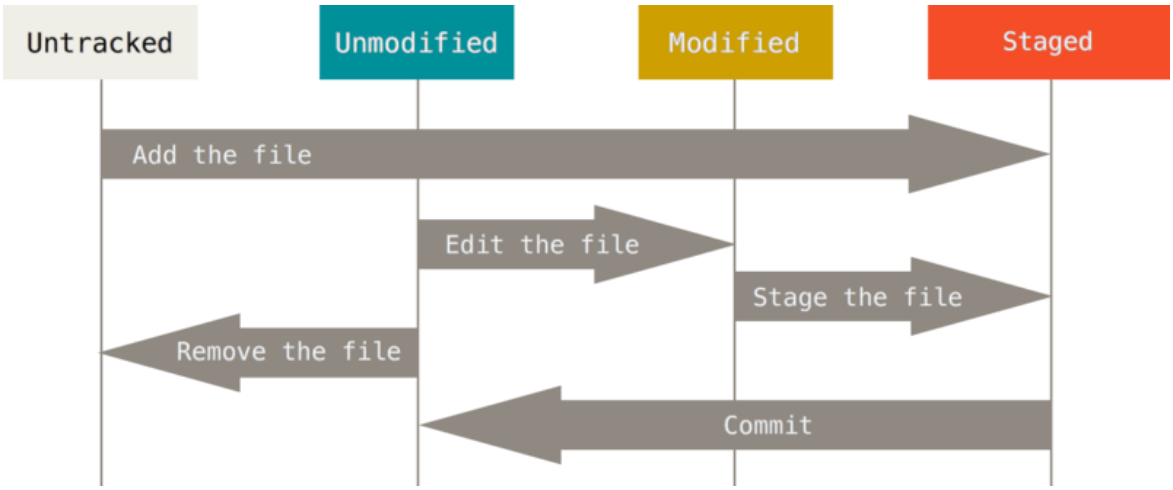
Git guarda todo en su base de datos no como nombres de archivo, si no que usa el hash sha-1 de su contenido.

Git simplemente añade nuevos datos

Cuando realizas operaciones con comandos de Git, siempre se añaden datos al histórico. Esto significa que es difícil perder información una vez que algo se registra en Git. Además, nos da la posibilidad de experimentar con el código añadiendo los cambios que queramos con la seguridad de que podemos reestablecerlo a una versión anterior en cualquier momento.

Flujo de Trabajo en Git

Para incluir un cambio o una serie de cambios en el siguiente *snapshot*, hay que seguir un flujo de trabajo específico. En pocas palabras, el flujo consiste en editar los ficheros añadiéndoles las modificaciones pertinentes (estado de *modified*), comunicarle a Git el conjunto de ficheros modificados que van a ser incluidos en la siguiente versión (estado de *staged*), por último, se revisan los cambios introducidos y se confirma la creación de una nueva *snapshot* (estado de *commited*).



Resumidamente, los diferentes estados de un fichero en Git son:

- *Modified*: En este estado un fichero ha sido modificado, pero dichos cambios no han sido marcados para ser incluirlos en el siguiente *snapshot*.
- *Staged:* Este es un paso intermedio antes de aplicar un cambio. Al marcar un fichero ya modificado al área de *staged*, será incluido en el siguiente *snapshot* del proyecto. Dicho de otra manera, sólo se incluirán en el siguiente *snapshot* únicamente el conjunto de ficheros modificados que estén marcados como *staged*, el resto de los ficheros modificados no formarán parte de dicha nueva versión.
- Commited: En este estado, significa que los cambios del fichero ya han sido incluidos en Git y que actualmente no se han añadido nuevas modificaciones.

Resumidamente el flujo típico de trabajo en Git sería:

1. Modificas un conjunto de ficheros en tu proyecto.
2. Seleccionas un conjunto de los ficheros modificados (no tienen por qué ser todos) y los marcas para ser incluidos en el siguiente *snapshot*. (los marcas como *staged*)
3. Realizas el *Commit*, por lo que se genera una nueva versión del proyecto (*snapshot*) incluyendo únicamente aquellos archivos modificados marcados como *staged* en el paso anterior.

Línea de Comandos en Git

Existen múltiples maneras de interactuar con Git, desde la propia línea de comandos hasta poderosas interfaces visuales que nos permiten realizar este tipo de acciones de manera sencilla.

En este documento nos centraremos en la línea de comandos ya que básicamente es la mejor manera de interiorizar cómo funciona Git realmente. Las interfaces visuales están pensadas para abstraer al usuario de la complejidad de los comandos de Git que se ejecutan a bajo nivel y nos ofrecen una visión simplificada de las acciones a realizar.

Instalando Git

Para comprobar si tenemos instalado Git correctamente simplemente ejecutamos en la línea de comandos:

```
$ git --version
```

Esto nos devolverá la versión actual instalada en nuestro sistema.

Si no disponemos de Git instalado, simplemente accedemos la web, https://git-scm.com/ seleccionamos nuestro sistema operativo y le damos a descargar.

Primeros pasos Configurando Git

Nada más instalar Git, deberemos configurarlo. Para ello existe un comando, el git config que nos permite dar valor a una serie de variables que manejarán el comportamiento de ciertos aspectos de git. Puedes ver la configuración actual mediante el comando:

$ git config --list

Lo primero que se configura es tu identidad. Este es un paso crítico ya que Git usa esta información para incluirla en el historial de cambios del proyecto:

$ git config --global user.name "John Doe"

$ git config --global user.email johndoe@example.com

Obteniendo ayuda

Para obtener ayuda sobre cómo funciona un comando de Git:

$ git help config

$ git help merge

La mejor manera podría ser simplemente consultar en Internet cómo funciona un comando, pero puede darse el caso de que no dispongas de acceso a internet, por esa razón el comando *git help <comando>* puede ser de gran utilidad incluso hoy en día.

Puedes también obtener una información reducida de un comando en específico usando dicho comando pasándole la opción –h:

$ git merge -h

$ git add -h

Inicializar un Repositorio en Git

Llegados a este punto conocemos de manera conceptual y básica cómo funciona Git y qué lo diferencia de otros sistemas.

Pero ahora veremos, cómo configurar e inicializar un repositorio. Comenzar a seguir o ignorar ficheros, meterlos en la zona de *stage* y confirmar dichos cambios haciendo un *commit*. También aprenderemos a deshacer errores que hayamos cometido, ver el historial de cambios del proyecto y cómo propagar y obtener los cambios de los repositorios remotos.

Hay dos maneras de obtener un repositorio de Git:

1. Puedes configurar un directorio cualquiera para convertirlo en un repositorio de Git.
2. Puedes clonar un repositorio que ya existe desde Internet (por ejemplo Github).

En cualquiera de estos dos escenarios, tendrás tu propio repositorio de Git en tu máquina local listo para trabajar.

Inicializar un repositorio local de Git

Si tienes un proyecto en tu ordenador y quieres empezar a controlarlo usando Git, simplemente hay que ejecutar el comando “git init”:

$ mkdir mi-fabuloso-proyeto

$ cd mi-fabuloso-proyeto

$ git init

El comando “git init” inicializará el proyecto. Creará un subdirectorio oculto, llamado “.git” en el que Git guardará todos los datos necesarios para gestionar nuestro proyecto.

Clonando un repositorio remoto de Git

Si lo que quieres es obtener una copia de un proyecto existente, por ejemplo, en el caso de que quieras contribuir en él. El comando a ejecutar es el de “git clone <ulr>” y en lugar de obtener simplemente una copia de su estado actual, Git se descarga todos los datos que el servidor contiene (incluido el historial de cambios completo de cada fichero). Si queremos, por ejemplo, podemos descargarnos el proyecto de React de su repositorio de Github:

$ git clone https://github.com/facebook/react

Este comando crea un directorio llamado “react” e inicializa el directorio “.git” dentro del mismo, descargándose toda la información disponible del servidor y obteniendo la última versión disponible de todos los ficheros.

Llegados a este punto, tenemos todo listo para poder trabajar sobre nuestro proyecto.

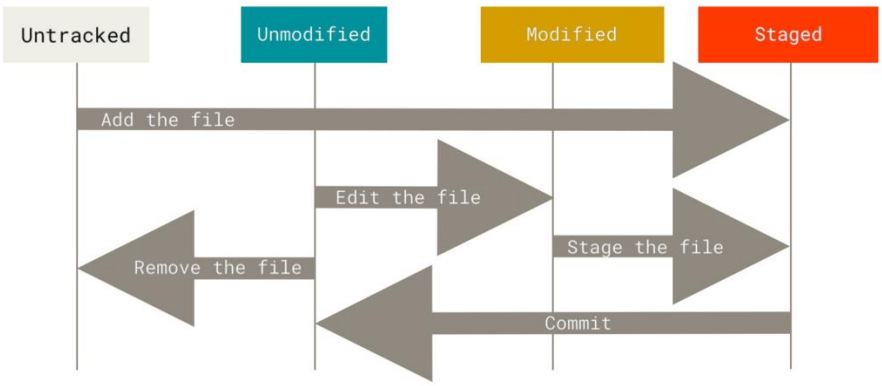
Trabajando sobre el Repositorio de Git

Para recapitular, recordemos los diferentes estados en los que puede estar un fichero dentro de un repositorio de Git:

Los dos estados principales en los que pueden estar un fichero son *tracked* o *untraked*. Los que son marcados como *tracked*, son los que únicamente Git hará seguimiento de sus cambios.

Cuando uno de los ficheros marcados como *tracked* es modificado, Git se dará cuenta de este cambio automáticamente, ya que el hash del fichero modificado no coincidirá con el de su anterior *snapshot*.

Cuando marcas dicho fichero modificado a la zona de *staged* y realizas el *commit*, una nueva versión de este fichero se introduce en histórico de cambios y el ciclo comienza de nuevo.



Conocer el estado de los ficheros en Git

Para saber el estado actual de nuestros ficheros usaremos el comando “git status”. Este comando nos dará información sobre la rama en la que nos encontramos actualmente (ya hablaremos más adelante sobre el sistema de ramas que tiene Git), y sobre los ficheros que están modificados, en la zona de *stage* y sin seguimiento por parte de git:

$ git status

```
AMAC02XJ55WJGH6:mi-fabuloso-proyecto d.a.somoza.paletta$ git status
En la rama master

No hay commits todavía

no hay nada para confirmar (crea/copia archivos y usa "git add" para hacerles seguimiento)
AMAC02XJ55WJGH6:mi-fabuloso-proyecto d.a.somoza.paletta$
```

En este caso Git nos informa que estamos actualmente en la rama principal de nuestro proyecto (rama *master*), y que aún no hemos realizado ningún *commit* sobre nuestro repositorio. Además, nos avisa de que todavía no hemos añadido ningún fichero para que Git haga un seguimiento de los mismos.

Añadiendo nuevos ficheros

Lo primero que tenemos que hacer es crear nuestros primeros ficheros:

```
AMAC02XJ55WJGH6:mi-fabuloso-proyecto d.a.somoza.paletta$ ls -la
total 0
drwxr-xr-x  3 d.a.somoza.paletta  staff   96 11 oct 11:22 .
drwxr-xr-x  4 d.a.somoza.paletta  staff  128 11 oct 11:21 ..
drwxr-xr-x  9 d.a.somoza.paletta  staff  288 11 oct 13:10 .git
AMAC02XJ55WJGH6:mi-fabuloso-proyecto d.a.somoza.paletta$ echo "este es nuestro primer fichero" > fichero1.txt
AMAC02XJ55WJGH6:mi-fabuloso-proyecto d.a.somoza.paletta$ echo "este es nuestro segundo fichero" > fichero2.txt
AMAC02XJ55WJGH6:mi-fabuloso-proyecto d.a.somoza.paletta$ echo "este es nuestro tercer fichero" > fichero3.txt
AMAC02XJ55WJGH6:mi-fabuloso-proyecto d.a.somoza.paletta$ ls -la
total 24
drwxr-xr-x  6 d.a.somoza.paletta  staff  192 11 oct 13:29 .
drwxr-xr-x  4 d.a.somoza.paletta  staff  128 11 oct 11:21 ..
drwxr-xr-x  9 d.a.somoza.paletta  staff  288 11 oct 13:10 .git
-rw-r--r--  1 d.a.somoza.paletta  staff   31 11 oct 13:29 fichero1.txt
-rw-r--r--  1 d.a.somoza.paletta  staff   32 11 oct 13:29 fichero2.txt
-rw-r--r--  1 d.a.somoza.paletta  staff   31 11 oct 13:29 fichero3.txt
AMAC02XJ55WJGH6:mi-fabuloso-proyecto d.a.somoza.paletta$ git status
En la rama master

No hay commits todavía

Archivos sin seguimiento:
  (usa "git add <archivo>..." para incluirlo a lo que se será confirmado)

        fichero1.txt
        fichero2.txt
        fichero3.txt

no hay nada agregado al commit pero hay archivos sin seguimiento presentes (usa "git add" para hacerles seguimiento)
AMAC02XJ55WJGH6:mi-fabuloso-proyecto d.a.somoza.paletta$
```

Nada más crear nuestros primeros 3 ficheros, ejecutamos el comando “git status”. Git nos informa que existen 3 ficheros en nuestro repositorio, pero que no han sido marcados para su seguimiento. Nos sugiere además que para incluirlos simplemente deberemos ejecutar el comando “git add <file_name>” (si queremos añadirlos todos de una vez: “git add *”).

$ git add <file_name>

Una vez ejecutado este comando, Git empezará a hacer un seguimiento de dicho fichero, comparando el hash del contenido actual con el del anterior *snapshot*. Si el hash no coincide, Git sabrá si el fichero ha sido modificado.

```
AMAC02XJ55WJGH6:mi-fabuloso-proyecto d.a.somoza.paletta$ git add fichero1.txt
AMAC02XJ55WJGH6:mi-fabuloso-proyecto d.a.somoza.paletta$ git add fichero2.txt
AMAC02XJ55WJGH6:mi-fabuloso-proyecto d.a.somoza.paletta$ git status
En la rama master

No hay commits todavía

Cambios a ser confirmados:
  (usa "git rm --cached <archivo>..." para sacar del área de stage)

        nuevo archivo:  fichero1.txt
        nuevo archivo:  fichero2.txt

Archivos sin seguimiento:
  (usa "git add <archivo>..." para incluirlo a lo que se será confirmado)

        fichero3.txt
```

Una vez ejecutado el comando "git add" con los dos primeros ficheros, estos pasarán al área de *stage*, pero no serán incluidos en el historial de cambios hasta que se ejecute el comando "git commit". El tercer fichero no ha sido incluido en el área de *stage*, por lo que Git lo ignorará hasta que el usuario decida incluirlo también usando el comando "git add"

Crear una nueva snapshot de nuestro proyecto

En este paso vamos a confirmar todos los cambios realizados que estén en la zona de stage (todos los ficheros modificados que no estén en esta zona no serán incluidos en la nueva *snapshot*). El comando para realizar esto es "git commit -m "descripción del cambio introducido"".

$ git commit -m "inital commit"

Por lo que en nuestro repositorio de ejemplo:

```
AMAC02XJ55WJGH6:mi-fabuloso-proyecto d.a.somoza.paletta$
AMAC02XJ55WJGH6:mi-fabuloso-proyecto d.a.somoza.paletta$ git commit -m "initial commit"
[master (commit-raíz) 8b9d518] initial commit
 2 files changed, 2 insertions(+)
 create mode 100644 fichero1.txt
 create mode 100644 fichero2.txt
AMAC02XJ55WJGH6:mi-fabuloso-proyecto d.a.somoza.paletta$ git status
En la rama master
Archivos sin seguimiento:
  (usa "git add <archivo>..." para incluirlo a lo que se será confirmado)

        fichero3.txt

no hay nada agregado al commit pero hay archivos sin seguimiento presentes (usa "git add" para hacerles seguimiento)
AMAC02XJ55WJGH6:mi-fabuloso-proyecto d.a.somoza.paletta$
```

De esta manera nuestros dos ficheros han sido incluidos. Podemos además fijarnos que en la salida de dicho comando, nos muestra la rama en la que han sido incluidos (rama master), el checksum SHA-1 del commit (8b9d518) y además el número de ficheros modificados y el número total de líneas.

Cuando ejecutamos "git status" ahora sólo nos muestra el fichero3.txt como sin seguimiento.

Con esta acción hemos realizado el primer cambio en nuestro proyecto.

Modificando ficheros

Ahora vamos a modificar uno de los ficheros que están siendo seguidos por Git. Por ejemplo, si modificamos el fichero2.txt y ejecutamos “git status” deberíamos ver un mensaje como este:

```
AMAC02XJ55WJGH6:mi-fabuloso-proyecto d.a.somoza.paletta$ cat fichero2.txt
este es nuestro segundo fichero
AMAC02XJ55WJGH6:mi-fabuloso-proyecto d.a.somoza.paletta$ echo "modificación del segundo fichero" > fichero2.txt
AMAC02XJ55WJGH6:mi-fabuloso-proyecto d.a.somoza.paletta$ cat fichero2.txt
modificación del segundo fichero
AMAC02XJ55WJGH6:mi-fabuloso-proyecto d.a.somoza.paletta$ git status
En la rama master
Cambios no rastreados para el commit:
  (usa "git add <archivo>..." para actualizar lo que será confirmado)
  (usa "git checkout -- <archivo>..." para descartar los cambios en el directorio de trabajo)

        modificado:     fichero2.txt

Archivos sin seguimiento:
  (usa "git add <archivo>..." para incluirlo a lo que se será confirmado)

        fichero3.txt

sin cambios agregados al commit (usa "git add" y/o "git commit -a")
AMAC02XJ55WJGH6:mi-fabuloso-proyecto d.a.somoza.paletta$
```

Hemos modificado el fichero2.txt y Git se ha dado cuenta automáticamente. Pero para que dicha modificación forme parte de la nueva *snapshot* de Git tenemos que mandarla al área de *stage* y *commitearlo*. Para añadirlo al área de stage, usaremos otra vez el comando de “git add” de la misma manera que en el paso anterior:

```
AMAC02XJ55WJGH6:mi-fabuloso-proyecto d.a.somoza.paletta$ git status
En la rama master
Cambios no rastreados para el commit:
  (usa "git add <archivo>..." para actualizar lo que será confirmado)
  (usa "git checkout -- <archivo>..." para descartar los cambios en el directorio de trabajo)

        modificado:     fichero2.txt

Archivos sin seguimiento:
  (usa "git add <archivo>..." para incluirlo a lo que se será confirmado)

        fichero3.txt

sin cambios agregados al commit (usa "git add" y/o "git commit -a")
AMAC02XJ55WJGH6:mi-fabuloso-proyecto d.a.somoza.paletta$ git add fichero2.txt
AMAC02XJ55WJGH6:mi-fabuloso-proyecto d.a.somoza.paletta$ git status
En la rama master
Cambios a ser confirmados:
  (usa "git reset HEAD <archivo>..." para sacar del área de stage)

        modificado:     fichero2.txt

Archivos sin seguimiento:
  (usa "git add <archivo>..." para incluirlo a lo que se será confirmado)

        fichero3.txt

AMAC02XJ55WJGH6:mi-fabuloso-proyecto d.a.somoza.paletta$
```

Después de ejecutar “git add fichero2.txt” y “git status”, se nos muestra que ahora el fichero2.txt se encuentra en la zona de *stage*, además nos comunica que si quisiéramos sacar el fichero de la zona de *stage* mediante el comando “git reset HEAD fichero2.txt”.

$ git reset HEAD <fichero>

Para volver a añadirlo a la zona de stage ejecutamos nuevamente el comando “git add fichero2.txt”.

```
AMAC02XJ55WJGH6:mi-fabuloso-proyecto d.a.somoza.paletta$ git status
En la rama master
Cambios a ser confirmados:
  (usa "git reset HEAD <archivo>..." para sacar del área de stage)

        modificado:     fichero2.txt

Archivos sin seguimiento:
  (usa "git add <archivo>..." para incluirlo a lo que se será confirmado)

        fichero3.txt

AMAC02XJ55WJGH6:mi-fabuloso-proyecto d.a.somoza.paletta$ git reset HEAD fichero2.txt
Cambios fuera del área de stage tras el reset:
M       fichero2.txt
AMAC02XJ55WJGH6:mi-fabuloso-proyecto d.a.somoza.paletta$ git status
En la rama master
Cambios no rastreados para el commit:
  (usa "git add <archivo>..." para actualizar lo que será confirmado)
  (usa "git checkout -- <archivo>..." para descartar los cambios en el directorio de trabajo)

        modificado:     fichero2.txt

Archivos sin seguimiento:
  (usa "git add <archivo>..." para incluirlo a lo que se será confirmado)

        fichero3.txt

sin cambios agregados al commit (usa "git add" y/o "git commit -a")
AMAC02XJ55WJGH6:mi-fabuloso-proyecto d.a.somoza.paletta$ git add fichero2.txt
AMAC02XJ55WJGH6:mi-fabuloso-proyecto d.a.somoza.paletta$ git status
En la rama master
Cambios a ser confirmados:
  (usa "git reset HEAD <archivo>..." para sacar del área de stage)

        modificado:     fichero2.txt

Archivos sin seguimiento:
  (usa "git add <archivo>..." para incluirlo a lo que se será confirmado)

        fichero3.txt

AMAC02XJ55WJGH6:mi-fabuloso-proyecto d.a.somoza.paletta$
```

En algunos casos el comando “git status” no sea la mejor forma de ver los cambios que estamos introduciendo en el repositorio. Como alternativa podemos usar el comando “git diff”. Este comando nos proporciona información detallada de qué líneas se están añadiendo y modificando.

Para ver los cambios que no han sido marcados como stage:

$ git diff

Para ver los cambios que han sido marcados como stage:

$ git diff --staged

```
AMAC02XJ55WJGH6:mi-fabuloso-proyecto d.a.somoza.paletta$ git diff
AMAC02XJ55WJGH6:mi-fabuloso-proyecto d.a.somoza.paletta$ git diff --staged
diff --git a/fichero2.txt b/fichero2.txt
index f8f563e..85ee6f0 100644
--- a/fichero2.txt
+++ b/fichero2.txt
@@ -1 +1 @@
-este es nuestro segundo fichero
+modificación del segundo fichero
AMAC02XJ55WJGH6:mi-fabuloso-proyecto d.a.somoza.paletta$
```

Para confirmar dicho cambio solo nos falta realizar el commit:

```
AMAC02XJ55WJGH6:mi-fabuloso-proyecto d.a.somoza.paletta$ git commit -m "modificado el fichero2.txt"
[master bd46346] modificado el fichero2.txt
 1 file changed, 1 insertion(+), 1 deletion(-)
AMAC02XJ55WJGH6:mi-fabuloso-proyecto d.a.somoza.paletta$
```

Revisando el Histórico de Cambios

Ahora que hemos creado varios *commits* en nuestro proyecto de prueba, podemos ver todo el histórico de cambios realizados en nuestro proyecto. Para realizar esto usaremos el comando "git log"

$ git log

```
AMAC02XJ55WJGH6:mi-fabuloso-proyecto d.a.somoza.paletta$ git log
commit bd4634669852282e08cd73d5fc80faa3458ba9ca (HEAD -> master)
Author: Daniel Somoza Paletta <d.a.somoza.paletta@accenture.com>
Date:   Sun Oct 11 20:15:07 2020 +0200

    modificado el fichero2.txt

commit 8b9d518211793949697 59f913b1dcdbe75457aae
Author: Daniel Somoza Paletta <d.a.somoza.paletta@accenture.com>
Date:   Sun Oct 11 14:17:28 2020 +0200

    initial commit
AMAC02XJ55WJGH6:mi-fabuloso-proyecto d.a.somoza.paletta$
```

Como podemos ver, este comando nos muestra todos los *commits* realizados en orden cronológico inverso, es decir, los más recientes se muestran primero. En nuestro proyecto se han realizado 2 *commits*, en cada *uno* se muestra detalles relevantes del mismo, el autor, la fecha, el mensaje y el hash.

Este comando admite una gran variedad de opciones adicionales, por ejemplo si se ejecuta con la opción -p se muestran los cambios introducidos en cada fichero:

```
AMAC02XJ55WJGH6:mi-fabuloso-proyecto d.a.somoza.paletta$ git log -p
commit bd4634669852282e08cd73d5fc80faa3458ba9ca (HEAD -> master)
Author: Daniel Somoza Paletta <d.a.somoza.paletta@accenture.com>
Date:   Sun Oct 11 20:15:07 2020 +0200

    modificado el fichero2.txt

diff --git a/fichero2.txt b/fichero2.txt
index f8f563e..85ee6f0 100644
--- a/fichero2.txt
+++ b/fichero2.txt
@@ -1 +1 @@
-este es nuestro segundo fichero
+modificación del segundo fichero

commit 8b9d518211793949697 59f913b1dcdbe75457aae
Author: Daniel Somoza Paletta <d.a.somoza.paletta@accenture.com>
Date:   Sun Oct 11 14:17:28 2020 +0200

    initial commit

diff --git a/fichero1.txt b/fichero1.txt
new file mode 100644
index 0000000..12862c3
--- /dev/null
+++ b/fichero1.txt
@@ -0,0 +1 @@
+este es nuestro primer fichero
diff --git a/fichero2.txt b/fichero2.txt
new file mode 100644
index 0000000..f8f563e
--- /dev/null
+++ b/fichero2.txt
@@ -0,0 +1 @@
+este es nuestro segundo fichero
AMAC02XJ55WJGH6:mi-fabuloso-proyecto d.a.somoza.paletta$
```

Otra opción útil es mostrar los ficheros modificados en cada *commit*, así como el número de líneas afectadas:

$ git log --stat

Para cambiar el formato de lo que te devuelve el comando se puede usar la opción --pretty que admite *oneline*, *short*, *full*, and *fuller*

$ git log --pretty=oneline

Para limitar el número de *commits* a visualizar simplemente le pasamos un -n, siendo n el número de *commits* que queremos visualizar, por ejemplo el comando “git log --stat -5” mostrará sólo los último 5 *commits* realizados.

¡Me he olvidado de commitear un fichero!

A veces con las prisas podemos olvidarnos de añadir uno de los ficheros que hemos modificado en nuestro último *commit*. Para evitar *commits* innecesarios del estilo “ooops, he olvidado añadir el fichero x al último *commit*”, podemos hacer uso del comando “git commit --amend”de esta manera:

$ git commit -m “mi commit incompleto”

$ git add mi_fichero_olvidado.txt

$ git commit --amend

Con este comando, sustituiremos el anterior *commit* por uno nuevo, añadiéndole el fichero mi_fichero_olvidado.txt. Trata de usar este comando antes de propagar los cambios a través de un repositorio remoto, porque podría causar problemas en otros colaboradores remotos.

Descartar las modificaciones de un fichero

Si hemos realizado modificaciones en un fichero y queremos descartar dichos cambios (porque por ejemplo no nos gustan los cambios realizados o porque ya no hacen falta), simplemente ejecutamos el comando "git checkout -- <fichero>".

$ git checkout -- fichero_a_reestablecer.txt

Este comando es peligroso, los cambios realizados en el fichero se pierden para siempre. Ya que lo que hace Git realmente es reemplazar el fichero modificado por el del anterior *snapshot*. Al usar este comando asegúrate bien de que los cambios que has realizado son descartables, porque se perderán para siempre.

Alternativamente puedes usar el comando "git restore <file>" para este propósito.

Trabajando con repositorios remotos

Hasta ahora nos hemos centrado en la gestión local del repositorio, pero como ya hemos comentado, Git es un sistema de control de versiones **distribuido**, por lo que está preparado para ser usado en proyectos cuyos desarrolladores están repartidos a lo largo del mundo.

Por lo que si queremos colaborar en un proyecto de este tipo, necesitamos saber cómo manejar los repositorios remotos.

En esencia, un repositorio remoto no es nada más que la copia completa de nuestro proyecto disponible a través de internet. Puedes tener el número de repositorios remotos que quieras para un mismo proyecto. Básicamente, las operaciones a realizar se resumen en dos, la de propagar y obtener los cambios realizados en el proyecto.

Para ver la lista de los servidores remotos definidos en un repositorio usaremos el comando:

$ git remote -v

Si has clonado el repositorio desde un servidor, debería aparecer en la lista con la etiqueta "origin", que es el nombre por defecto que asigna Git al servidor desde el que has realizado la clonación.

Añadiendo un nuevo repositorio remoto

Para añadir un nuevo repositorio remoto a Git simplemente usa el comando "git remote add <nombre> <url>"

$ git remote add origin https://github.com/usuario/mi-fabuloso-proyecto.git

Obtener los cambios del repositorio remoto

Para obtener los cambios del repositorio remoto simplemente hay que ejecutar:

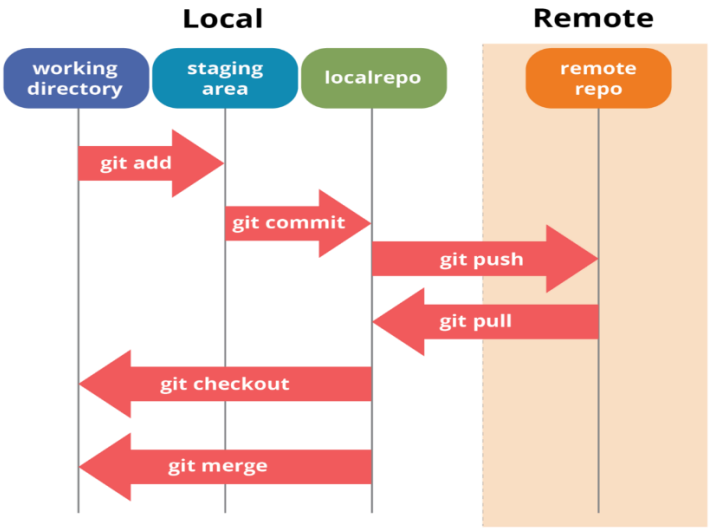
$ git fetch <servidor_remoto>

Este comando se trae todos los datos que todavía no tengas del servidor remoto. Pero es importante tener en cuenta que sólo se trae los cambios, no los incluye en tu rama actual.

Realmente es más útil el comando "git pull".

$ git pull <servidor_remoto>

Este comando automáticamente se traerá los cambios del repositorio remoto y los *mergeará* en tu rama actual.



Subir los cambios a un repositorio remoto

Para subir los cambios que hayas realizado al repositorio remoto simplemente ejecuta el comando “git push <servido_remoto> <rama>”:

$ git push origin master

Obtener detalle de un repositorio remoto

Si quieres obtener información detallada de un repositorio remoto simplemente ejecuta:

$ git remote show origin

Git Branching

Las ramas (*branches*) en Git nos permiten bifurcar nuestro desarrollo para poder experimentar sin mezclarlo con la línea principal de nuestro código.

Creación de una rama

En Git siempre se trabaja con ramas. La rama que se crea por defecto es llama rama master, no tiene nada de especial con respecto a otras ramas sólo que es la que Git crea por defecto al ejecutarse el comando “git init”. Para crear otra rama deberemos usar el comando “git branch <nombre_de_la_rama>”:

$ git branch develop

Cambiando de rama

Para cambiarnos de una rama a otra usaremos el comando “git checkout”:

$ git checkout develop

Cuando nos cambiamos de rama, nuestro directorio de trabajo también cambiará. Si te mueves a una rama antigua que hace tiempo que no tocas, tu directorio de trabajo tendrá el mismo aspecto que tenía cuando hiciste el último *commit* en dicha rama.

Si tienes ficheros modificados, puede que Git no te deje moverte de rama.

Flujo de trabajo con Ramas

Básicamente, el flujo típico de trabajo que se usa en el mundo real suele seguir más o menos los siguientes pasos:

1. Existe una necesidad de añadir una funcionalidad a un proyecto.
2. Creas una rama a partir de la rama principal (rama master) para introducir dicho cambio en el proyecto.
3. Realizas todo tu trabajo en dicha rama.

Cuando estás en el paso 3, trabajando en esa nueva funcionalidad, de repente aparece alguien y te dice que se ha descubierto un fallo en el proyecto y que hay que arreglarlo urgentemente. Como estás trabajando sobre ramas, simplemente:

1. Cambiarias a la rama principal (rama master).
2. Creas una rama a partir de la rama principal (rama master) para arreglar dicho error.
3. Una vez arreglado el fallo, mezclas la rama con la solución (mergeas) con la rama principal.
4. Regresas a la rama original y continúas tu trabajo donde lo dejaste la última vez.

Mezclar dos ramas (Merge)

Cuando terminas tu trabajo sobre una rama, querrás introducirlo en la rama principal de desarrollo, para simplemente hay que colocarse sobre la rama que quieras introducir los cambios y ejecutar el comando “git merge <nombre_de_la_rama>”:

$ git checkout master

$ git merge mi_rama_acabada

Visualizar las ramas creadas

Con el comando “git branch” sin argumentos puedes visualizar la lista de las ramas actuales, con un * indicando la rama actual en la que te encuentras.

Una opción que puede ser útil es la de especificar si listar las ramas ya mergeadas o las pendientes por mergear:

$ git branch --merged

$ git branch --no-merged

Subir una rama al repositorio remoto

Para subir una rama al repositorio remoto por primera vez ejecutaremos el siguiente comando:

$ git push --set-upstream origin nombre_de_mi_rama

A partir de ahí con el comando “git push” será suficiente.

Bibliografía

Enlaces al contenido utilizado en este documento:

https://git-scm.com/book/es/v2

https://es.wikipedia.org/wiki/N%C3%BAcleo_Linux#Historia

https://es.wikipedia.org/wiki/Historia_de_Linux

Enlaces de Interés

Libro de Git:
https://git-scm.com/book/es/v2

Generador on-line de hashes sha-1:
https://passwordsgenerator.net/sha1-hash-generator/

Github:
https://github.com/

Bitbucket:
https://bitbucket.org

Gitlab:
https://gitlab.com/

Sourcetree:
https://www.sourcetreeapp.com/