# 

TEMAS:

* Conceptos básicos
  + Terminal o CLI
  + Servidores
  + Dominios
  + Protocolo http y https
* Entorno de desarrollo
  + Instalar nodeJs
* Git & Github
  + Crear cuenta Github
  + init, create, clone, commit, pull

OBJETIVO:

MARCO TEÓRICO

## Lectura 1: Pro Git

**Autor: Scott Chacon**

## Control de versiones

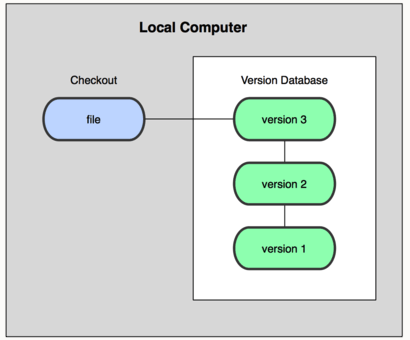
¿Qué es el control de versiones, y por qué debería ser

importarte? El control de versiones es un sistema que registra los cambios realizados sobre un archivo o conjunto de archivos a lo largo del tiempo, de modo que puedas recuperar versiones específicas más adelante. Los ejemplos de este libro utilizan el control de versiones para el código fuente, pero puedes emplearlo para casi cualquier tipo de archivo que encuentres en un ordenador.

## Sistemas de control de versiones locales

El método de control de versiones usado por mucha gente es copiar los archivos a otro directorio. Esta técnica es muy común porque es muy simple, pero también tremendamente propensa a cometer errores. Es fácil olvidar en qué directorio te encuentras, y guardar accidentalmente en el archivo equivocado o sobrescribir archivos que no querías.

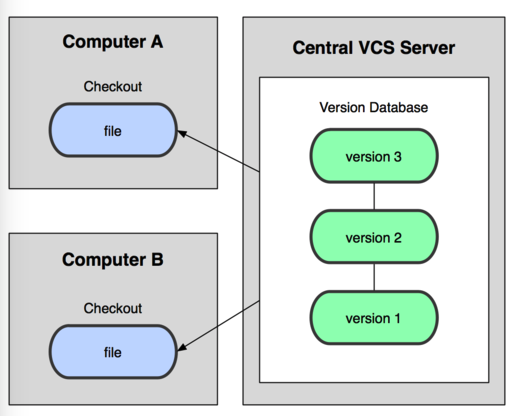
Para hacer frente a este problema, los programadores desarrollaron hace tiempo VCSs locales que contenían una simple base de datos en la que se llevaba registro de todos los cambios realizados sobre los archivos.



Una de las herramientas de control de versiones más popular fue un sistema llamado rcs, que todavía podemos encontrar en muchos de los ordenadores actuales. Hasta el famoso sistema operativo Mac OS X incluye el comando rcs cuando instalas las herramientas de desarrollo. Esta herramienta funciona básicamente guardando conjuntos de parches (es decir, las diferencias entre archivos) de una versión a otra en un formato especial en disco.

## Sistemas de control de versiones centralizados

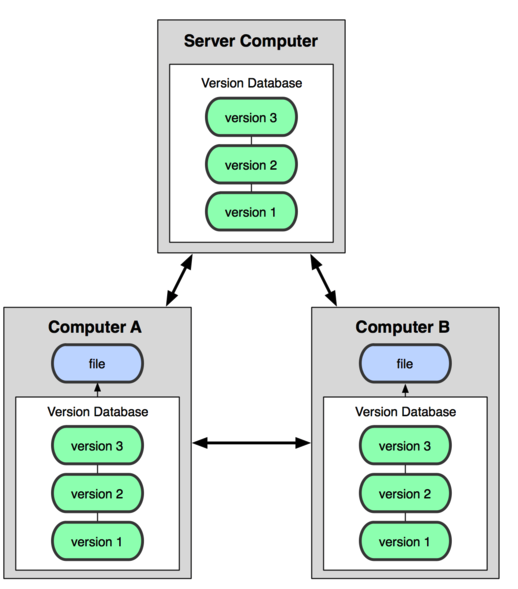
El siguiente gran problema que se encuentra la gente es que necesitan colaborar con desarrolladores en otros sistemas. Para solventar este problema, se desarrollaron los sistemas de control de versiones centralizados (Centralized Version Control Systems o CVCSs en inglés). Estos sistemas, como CVS, Subversion, y Perforce, tienen un único servidor que contiene todos los archivos versionados, y varios clientes que descargan los archivos de ese lugar central. Durante muchos años, éste ha sido el estándar para el control de versiones.



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

## Sistemas de control de versiones distribuidos

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



Es más, muchos de estos sistemas se las arreglan bastante bien teniendo varios repositorios con los que trabajar, por lo que puedes colaborar con distintos grupos de gente de maneras distintas simultáneamente dentro del mismo proyecto.

## Una breve historia de Git

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

En 2005, la relación entre la comunidad que desarrollaba el núcleo de Linux y la compañía que desarrollaba BitKeeper se vino abajo, y la herramienta dejó de ser 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 durante el uso de BitKeeper. Algunos de los objetivos del nuevo sistema fueron los siguientes:

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

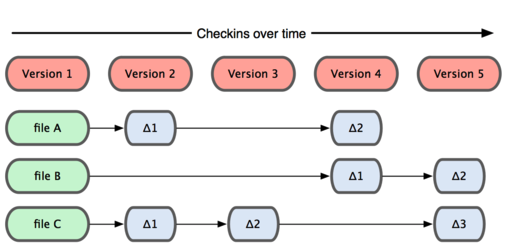
Desde su nacimiento en 2005, Git ha evolucionado y madurado para ser fácil de usar y aún así conservar estas cualidades iniciales. Es tremendamente rápido, muy eficiente con grandes proyectos, y tiene un increíble sistema de ramas (branching) para desarrollo no lineal.

## Fundamentos de Git

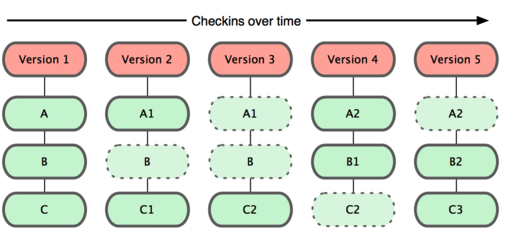
Entonces, ¿qué es Git en pocas palabras? Es muy importante asimilar esta sección, porque si entiendes lo que es Git y los fundamentos de cómo funciona, probablemente te sea mucho más fácil usar Git de manera eficaz. A medida que aprendas Git, intenta olvidar todo lo que puedas saber sobre otros VCSs, como Subversion y Perforce; hacerlo te ayudará a evitar confusiones sutiles a la hora de utilizar la herramienta. Git almacena y modela la información de forma muy diferente a esos otros sistemas, a pesar de que su interfaz sea bastante similar; comprender esas diferencias ayudará a evitar que te confundas a la hora de usarlo.

**Instantáneas, no diferencias**

La principal diferencia entre Git y cualquier otro VCS (Subversion y compañía incluidos) es cómo Git modela sus datos. Conceptualmente, la mayoría de los demás sistemas almacenan la información como una lista de cambios en los archivos. Estos sistemas (CVS, Subversion, Perforce, Bazaar, etc.) modelan la información que almacenan como un conjunto de archivos y las modificaciones hechas sobre cada uno de ellos a lo largo del tiempo



Git no modela ni almacena sus datos de este modo. En cambio, Git modela sus datos más como un conjunto de instantáneas de un mini sistema de archivos. Cada vez que confirmas un cambio, o guardas el estado de tu proyecto en Git, él básicamente hace una foto del aspecto de todos tus archivos en ese momento, y guarda una referencia a esa instantánea. Para ser eficiente, si los archivos no se han modificado, Git no almacena el archivo de nuevo — sólo un enlace al archivo anterior idéntico que ya tiene almacenado



Esta es una diferencia muy importante entre Git y prácticamente todos los demás VCSs. 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 a Git más como un mini sistema de archivos con algunas herramientas tremendamente potentes construidas sobre él, más que como un VCS.

**Casi cualquier operación es local**

La mayoría de las operaciones en Git sólo necesitan archivos y recursos locales para operar — por lo general no se necesita información de ningún otro ordenador de tu red —. Si estás acostumbrado a un CVCS donde la mayoría de las operaciones tienen esa sobrecarga del retardo de la red, este aspecto de Git te va a hacer pensar que los dioses de la velocidad han bendecido Git con poderes sobrenaturales. Como tienes toda la historia del proyecto ahí mismo, en tu disco local, la mayoría de las operaciones parecen prácticamente inmediatas.

Por ejemplo, para navegar por la historia del proyecto, Git no necesita acceder al servidor para obtener la historia y mostrártela — simplemente la lee directamente de tu base de datos local —. Esto significa que ves la historia del proyecto casi al instante. Si quieres ver los cambios introducidos entre la versión actual de un archivo y ese archivo hace un mes, Git puede buscar el archivo 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 del archivo del servidor remoto y hacerlo de manera local.

Esto también significa que hay muy poco que no puedas hacer si estás desconectado o sin VPN. Si te subes a un avión o a un tren y quieres trabajar un poco, puedes hacer tus cambios sin problemas hasta que consigas una conexión de red para subirlos. Si te vas a casa y no consigues que tu cliente VPN funcione correctamente, puedes seguir trabajando. En muchos otros sistemas, esto es imposible o muy costoso. En Perforce, por ejemplo, no puedes hacer mucho cuando no estás conectado al servidor; y en Subversion y CVS, puedes editar archivos, pero no puedes subir los cambios a tu base de datos (porque tu base de datos no tiene conexión). Esto puede no parecer gran cosa, pero te sorprendería la diferencia que puede suponer.

**Git tiene integridad**

Todo en Git es verificado mediante una suma de comprobación antes de ser almacenado, y es identificado a partir de ese momento mediante dicha suma (checksum en inglés). 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 en base a los contenidos del archivo o estructura de directorios en Git. Un hash SHA-1 tiene esta pinta:

24b9da6552252987aa493b52f8696cd6d3b00373

Verás estos valores hash por todos lados en Git, ya que los usa con mucha frecuencia. De hecho, Git guarda todo no por nombre de archivo, sino en la base de datos de Git por el valor hash de sus contenidos.

**Git generalmente sólo añade información**

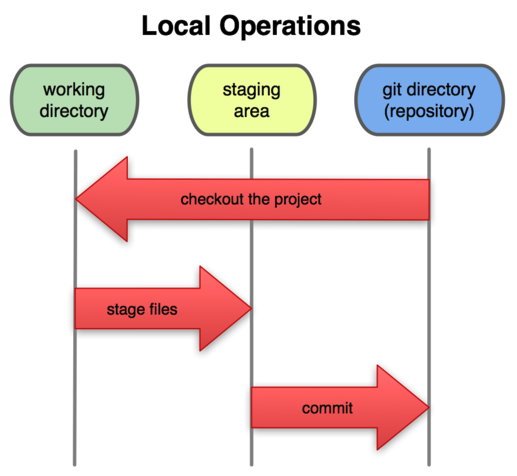
Cuando realizas acciones en Git, casi todas ellas sólo añaden información a la base de datos de Git. Es muy difícil conseguir que el sistema haga algo que no se pueda deshacer, o que de algún modo borre información. Como en cualquier VCS, puedes perder o estropear cambios que no has confirmado todavía; pero después de confirmar una instantánea en Git, es muy difícil de perder, especialmente si envías (push) tu base de datos a otro repositorio con regularidad.

Esto hace que usar Git sea un placer, porque sabemos que podemos experimentar sin peligro de fastidiar gravemente las cosas. Para un análisis más exhaustivo de cómo almacena Git su información y cómo puedes recuperar datos aparentemente perdidos

**Los tres estados**

Ahora presta atención. Esto es lo más importante a recordar acerca de Git si quieres que el resto de tu proceso de aprendizaje prosiga sin problemas. Git tiene tres estados principales en los que se pueden encontrar tus archivos: confirmado (committed), modificado (modified), y preparado (staged). Confirmado significa que los datos están almacenados de manera segura en tu base de datos local. Modificado significa que has modificado el archivo pero todavía no lo has confirmado a tu base de datos. Preparado significa que has marcado un archivo modificado en su versión actual para que vaya en tu próxima confirmación.

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



El directorio de Git es donde Git almacena 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.

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 sencillo archivo, generalmente contenido en tu directorio de Git, que almacena información acerca de lo que va a ir en tu próxima confirmación. A veces se denomina el índice, pero se está convirtiendo en estándar el referirse a ello como el área de preparación.

El flujo de trabajo básico en Git es algo así:

1. Modificas una serie de archivos en tu directorio de trabajo.
2. Preparas los archivos, añadiendo instantáneas de ellos 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 instantánea de manera permanente en tu directorio de Git.

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).

## Instalando Git

Vamos a empezar a usar un poco de Git. Lo primero es lo primero: tienes que instalarlo. Puedes obtenerlo de varias maneras; las dos principales son instalarlo desde código fuente, o instalar un paquete existente para tu plataforma.

**Instalando en Windows**

Instalar Git en Windows es muy fácil. El proyecto msysGit tiene uno de los procesos de instalación más sencillos. Simplemente descarga el archivo EXE del instalador desde la página de Google Code, y ejecútalo:

http://code.google.com/p/msysgit

Una vez instalado, tendrás tanto la versión de línea de comandos (incluido un cliente SSH que nos será útil más adelante) como la interfaz gráfica de usuario estándar.

**Configurando Git por primera vez**

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

Git trae una herramienta llamada git config que te permite obtener y establecer variables de configuración que controlan el aspecto y funcionamiento de Git. Estas variables pueden almacenarse en tres sitios distintos:

Archivo /etc/gitconfig: Contiene valores para todos los usuarios del sistema y todos sus repositorios. Si pasas la opción --system a git config, lee y escribe específicamente en este archivo.

Archivo ~/.gitconfig file: Específico a tu usuario. Puedes hacer que Git lea y escriba específicamente en este archivo pasando la opción --global.

Archivo config en el directorio de git (es decir, .git/config) del repositorio que estés utilizando actualmente: Específico a ese repositorio. Cada nivel sobrescribe los valores del nivel anterior, por lo que los valores de .git/config tienen preferencia sobre los de /etc/gitconfig.

En sistemas Windows, Git busca el archivo .gitconfig en el directorio $HOME (C:\Documents and Settings\$USER para la mayoría de usuarios). También busca en el directorio /etc/gitconfig, aunque esta ruta es relativa a la raíz MSys, que es donde quiera que decidieses instalar Git en tu sistema Windows cuando ejecutaste el instalador.

**Tu identidad**

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

**Tu editor**

Ahora que tu identidad está configurada, puedes elegir el editor de texto por defecto que se utilizará cuando Git necesite que introduzcas un mensaje. Si no indicas nada, Git usa el editor por defecto de tu sistema, que generalmente es Vi o Vim. Si quieres usar otro editor de texto, como Emacs, puedes hacer lo siguiente:



**Tu herramienta de diferencias**

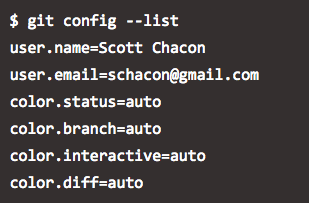
Otra opción útil que puede que quieras configurar es la herramienta de diferencias por defecto, usada para resolver conflictos de unión (merge). Digamos que quieres usar vimdiff:



Git acepta kdiff3, tkdiff, meld, xxdiff, emerge, vimdiff, gvimdiff, ecmerge, y opendiff como herramientas válidas.

**Comprobando tu configuración**

Si quieres comprobar tu configuración, puedes usar el comando git config --list para listar todas las propiedades que Git ha podido encontrar en ese punto:



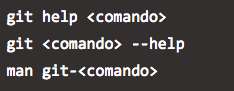
Puede que veas claves repetidas, porque Git lee la misma clave de distintos archivos (/etc/gitconfig y ~/.gitconfig, por ejemplo). En ese caso, Git usa el último valor para cada clave única que ve.

También puedes comprobar qué valor cree Git que tiene una clave específica ejecutando git config {clave}:



**Obteniendo ayuda**

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



Por ejemplo, puedes ver la página del manual para el comando config ejecutando:



Estos comandos están bien porque puedes acceder a ellos desde cualquier sitio, incluso sin conexión.

## 

## Fundamentos de Git

Si sólo puedes leer un capítulo para empezar a trabajar con Git, es éste. Este capítulo cubre todos los comandos básicos que necesitas para hacer la gran mayoría de las cosas a las que vas a dedicar tu tiempo en Git. Al final del capítulo, deberías ser capaz de configurar e inicializar un repositorio, comenzar y detener el seguimiento de archivos, y preparar (stage) y confirmar (commit) cambios. También te enseñaremos a configurar Git para que ignore ciertos archivos y patrones, cómo deshacer errores rápida y fácilmente, cómo navegar por la historia de tu proyecto y ver cambios entre confirmaciones, y cómo enviar (push) y recibir (pull) de repositorios remotos.

**Obteniendo un repositorio Git**

Puedes obtener un proyecto Git de dos maneras. La primera toma un proyecto o directorio existente y lo importa en Git. La segunda clona un repositorio Git existente desde otro servidor.

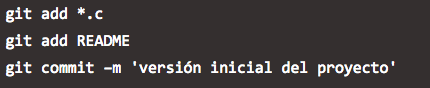
**Inicializando un repositorio en un directorio existente**

Si estás empezando el seguimiento en Git de un proyecto existente, necesitas ir al directorio del proyecto y escribir:



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

Si deseas empezar a controlar versiones de archivos existentes (a diferencia de un directorio vacío), probablemente deberías comenzar el seguimiento de esos archivos y hacer una confirmación inicial. Puedes conseguirlo con unos pocos comandos git add para especificar qué archivos quieres controlar, seguidos de un commit para confirmar los cambios:



Veremos lo que hacen estos comandos dentro de un minuto. En este momento, tienes un repositorio Git con archivos bajo seguimiento, y una confirmación inicial.

**Clonando un repositorio existente**

Si deseas obtener una copia de un repositorio Git existente — por ejemplo, un proyecto en el que te gustaría contribuir — el comando que necesitas es git clone. Si estás familizarizado con otros sistemas de control de versiones como Subversion, verás que el comando es clone y no 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 descargado 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 al servidor al estado en el que estaba cuando fue clonado.

Puedes clonar un repositorio con git clone [url]. Por ejemplo, si quieres clonar la librería Ruby llamada Grit, harías algo así:



Esto crea un directorio llamado "grit", 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 nuevo directorio grit, 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 grit, puedes especificarlo con la siguiente opción de línea de comandos:



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

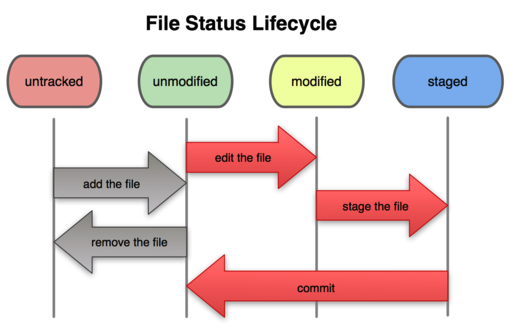
Git te permite usar distintos protocolos de transferencia. El ejemplo anterior usa el protocolo git://, pero también te puedes encontrar con http(s):// o usuario@servidor:/ruta.git, que utiliza el protocolo de transferencia SSH.

**Guardando cambios en el repositorio**

Tienes un repositorio Git completo, y una copia de trabajo de los archivos de ese proyecto. Necesitas hacer algunos cambios, y confirmar instantáneas de esos cambios a tu repositorio cada vez que el proyecto alcance un estado que desees grabar.

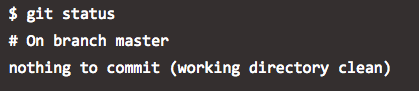
Recuerda que cada archivo de tu directorio de trabajo puede estar en uno de estos dos estados: bajo seguimiento (tracked), o sin seguimiento (untracked). Los archivos bajo seguimiento son aquellos que existían en la última instantánea; pueden estar sin modificaciones, modificados, o preparados. Los archivos sin seguimiento son todos los demás — cualquier archivo de tu directorio que no estuviese en tu última instantánea ni está en tu área de preparación —. La primera vez que clonas un repositorio, todos tus archivos estarán bajo seguimiento y sin modificaciones, ya que los acabas de copiar y no has modificado nada.

A medida que editas archivos, Git los ve como modificados, porque los has cambiado desde tu última confirmación. Preparas estos archivos modificados y luego confirmas todos los cambios que hayas preparado, y el ciclo se repite.



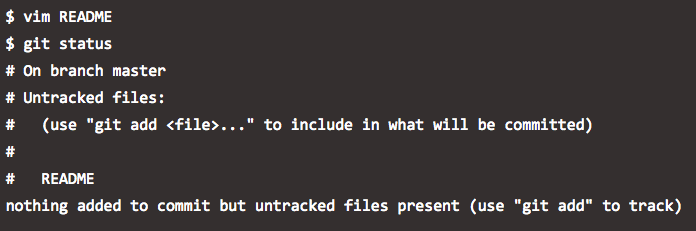
**Comprobando el estado de tus archivos**

Tu principal herramienta para determinar qué archivos están en qué estado es el comando git status. Si ejecutas este comando justo después de clonar un repositorio, deberías ver algo así:



Esto significa que tienes un directorio de trabajo limpio — en otras palabras, no tienes archivos bajo seguimiento y modificados —. Git tampoco ve ningún archivo que no esté bajo seguimiento, o estaría listado ahí. Por último, el comando te dice en qué rama estás. Por ahora, esa rama siempre es master, que es la predeterminada. No te preocupes de eso por ahora, el siguiente capítulo tratará los temas de las ramas y las referencias en detalle.

Digamos que añades un nuevo archivo a tu proyecto, un sencillo archivo README. Si el archivo no existía y ejecutas git status, verás tus archivos sin seguimiento así:



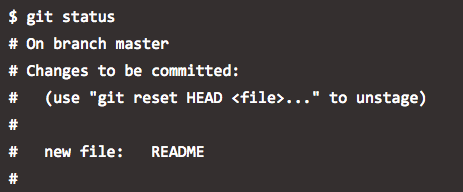
Puedes ver que tu nuevo archivo README aparece bajo la cabecera “Archivos sin seguimiento” (“Untracked files”) de la salida del comando. Sin seguimiento significa básicamente que Git ve un archivo que no estaba en la instantánea anterior; Git no empezará a incluirlo en las confirmaciones de tus instantáneas hasta que se lo indiques explícitamente. Lo hace para que no incluyas accidentalmente archivos binarios generados u otros archivos que no tenías intención de incluir. Sí que quieres incluir el README, así que vamos a iniciar el seguimiento del archivo.

**Seguimiento de nuevos archivos**

Para empezar el seguimiento de un nuevo archivo se usa el comando git add. Iniciaremos el seguimiento del archivo README ejecutando esto:



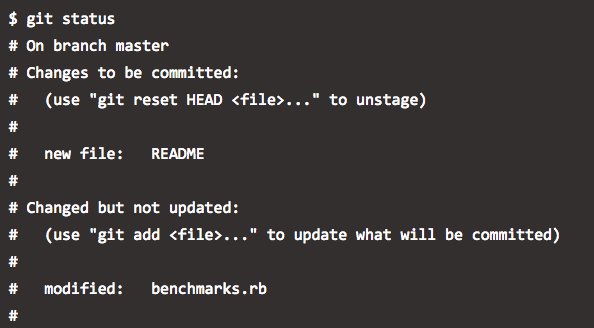
Si vuelves a ejecutar el comando git status, verás que tu README está ahora bajo seguimiento y preparado:



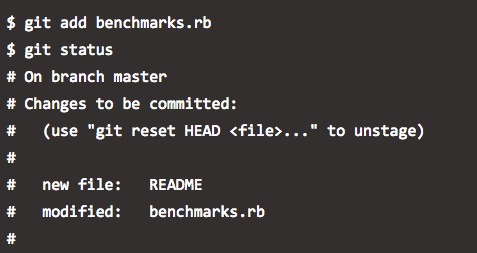
Puedes ver que está preparado porque aparece bajo la cabecera “Cambios a confirmar” (“Changes to be committed”). Si confirmas ahora, la versión del archivo en el momento de ejecutar git add será la que se incluya en la instantánea. Recordarás que cuando antes ejecutaste **git init**, seguidamente ejecutaste **git add** (archivos). Esto era para iniciar el seguimiento de los archivos de tu directorio. El comando git add recibe la ruta de un archivo o de un directorio; si es un directorio, añade todos los archivos que contenga de manera recursiva.

**Preparando archivos modificados**

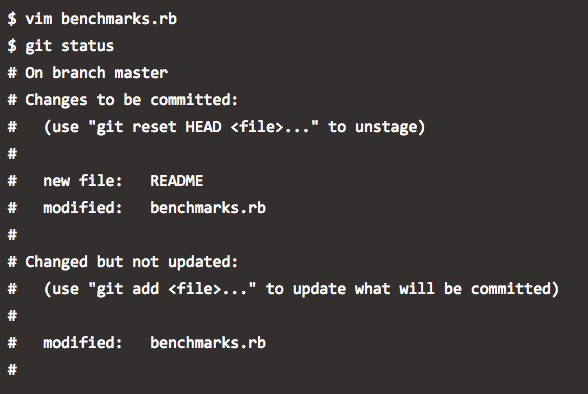
Vamos a modificar un archivo que estuviese bajo seguimiento. Si modificas el archivo benchmarks.rb que estaba bajo seguimiento, y ejecutas el comando status de nuevo, verás algo así:



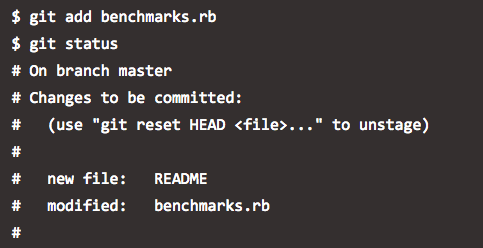
El archivo benchmarks.rb aparece bajo la cabecera “Modificados pero no actualizados” (“Changed but not updated”) — esto significa que un archivo bajo seguimiento ha sido modificado en el directorio de trabajo, pero no ha sido preparado todavía —. Para prepararlo, ejecuta el comando git add (es un comando multiuso — puedes utilizarlo para empezar el seguimiento de archivos nuevos, para preparar archivos, y para otras cosas como marcar como resueltos archivos con conflictos de unión —). Ejecutamos git add para preparar el archivo benchmarks.rb, y volvemos a ejecutar git status:



Ambos archivos están ahora preparados y se incluirán en tu próxima confirmación. Supón que en este momento recuerdas que tenías que hacer una pequeña modificación en benchmarks.rb antes de confirmarlo. Lo vuelves abrir, haces ese pequeño cambio, y ya estás listo para confirmar. Sin embargo, si vuelves a ejecutar git status verás lo siguiente:

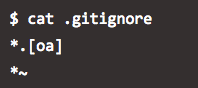


¿Pero qué...? Ahora benchmarks.rb aparece listado como preparado y como no preparado. ¿Cómo es posible? Resulta que Git prepara un archivo tal y como era en el momento de ejecutar el comando git add. Si haces git commit ahora, la versión de benchmarks.rb que se incluirá en la confirmación será la que fuese cuando ejecutaste el comando git add, no la versión que estás viendo ahora en tu directorio de trabajo. Si modificas un archivo después de haber ejecutado git add, tendrás que volver a ejecutar git add para preparar la última versión del archivo:



**Ignorando archivos**

A menudo tendrás un tipo de archivos que no quieras que Git añada automáticamente o te muestre como no versionado. Suelen ser archivos generados automáticamente, como archivos de log, o archivos generados por tu compilador. Para estos casos puedes crear un archivo llamado .gitignore, en el que listas los patrones de nombres que deseas que sean ignorados. He aquí un archivo .gitignore de ejemplo:



La primera línea le dice a Git que ignore cualquier archivo cuyo nombre termine en .o o .a — archivos objeto que suelen ser producto de la compilación de código —. La segunda línea le dice a Git que ignore todos los archivos que terminan en tilde (~), usada por muchos editores de texto, como Emacs, para marcar archivos temporales. También puedes incluir directorios de log, temporales, documentación generada automáticamente, etc. Configurar un archivo .gitignore antes de empezar a trabajar suele ser una buena idea, para así no confirmar archivos que no quieres en tu repositorio Git.

Las reglas para los patrones que pueden ser incluidos en el archivo .gitignore son:

Las líneas en blanco, o que comienzan por #, son ignoradas.

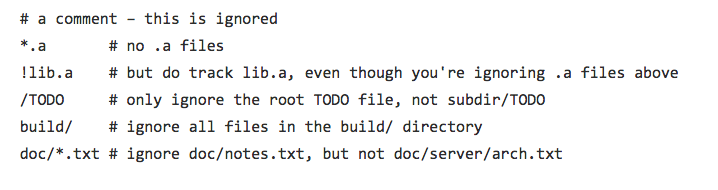
Puedes usar patrones glob estándar.

Puedes indicar un directorio añadiendo una barra hacia delante (/) al final.

Puedes negar un patrón añadiendo una exclamación (!) al principio.

Los patrones glob son expresiones regulares simplificadas que pueden ser usadas por las shells. Un asterisco (\*) reconoce cero o más caracteres; [abc] reconoce cualquier carácter de los especificados entre corchetes (en este caso, a, b, o c); una interrogación (?) reconoce un único carácter; y caracteres entre corchetes separados por un guión ([0-9]) reconoce cualquier carácter entre ellos (en este caso, de 0 a 9).

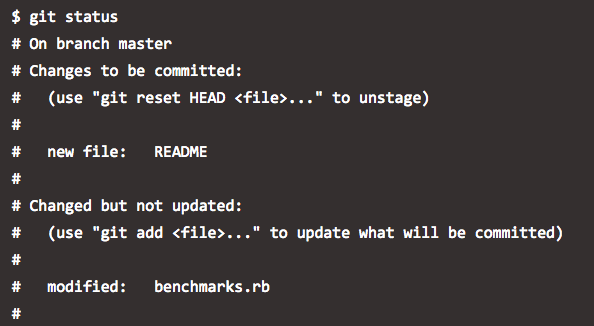
He aquí otro ejemplo de archivo .gitignore:



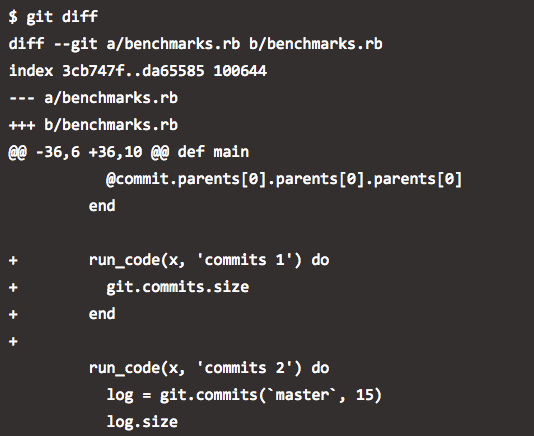
**Viendo tus cambios preparados y no preparados**

Si el comando git status es demasiado impreciso para ti — quieres saber exactamente lo que ha cambiado, no sólo qué archivos fueron modificados — puedes usar el comando git diff. Veremos git diff en más detalle después; pero probablemente lo usarás para responder estas dos preguntas: ¿qué has cambiado pero aún no has preparado?, y ¿qué has preparado y estás a punto de confirmar? Aunque git status responde esas preguntas de manera general, git diff te muestra exactamente las líneas añadidas y eliminadas — el parche, como si dijésemos —.

Supongamos que quieres editar y preparar el archivo README otra vez, y luego editar el archivo benchmarks.rb sin prepararlo. Si ejecutas el comando status, de nuevo verás algo así:

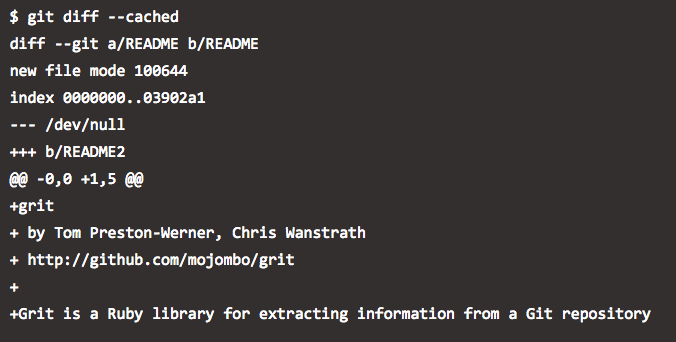


Para ver lo que has modificado pero aún no has preparado, escribe git diff:



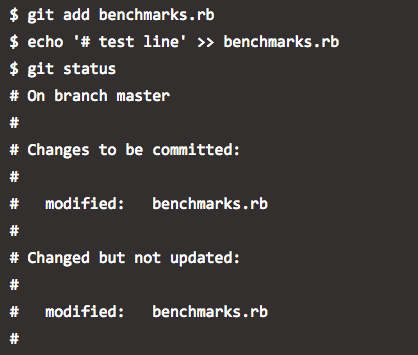
Ese comando compara lo que hay en tu directorio de trabajo con lo que hay en tu área de preparación. El resultado te indica los cambios que has hecho y que todavía no has preparado.

Si quieres ver los cambios que has preparado y que irán en tu próxima confirmación, puedes usar git diff –-cached. (A partir de la versión 1.6.1 de Git, también puedes usar git diff –-staged, que puede resultar más fácil de recordar). Este comando compara tus cambios preparados con tu última confirmación:

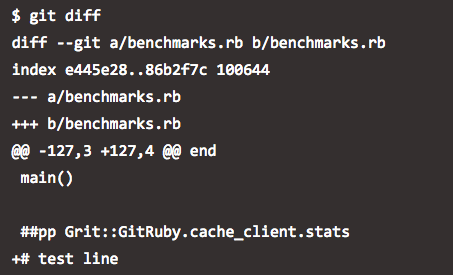


Es importante indicar que git diff por sí solo no muestra todos los cambios hechos desde tu última confirmación — sólo los cambios que todavía no están preparados —. Esto puede resultar desconcertante, porque si has preparado todos tus cambios, git diff no mostrará nada.

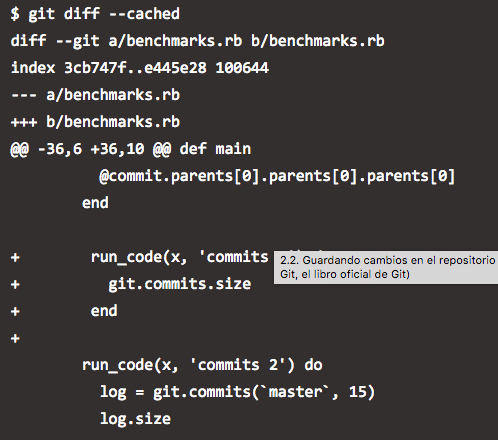
Por poner otro ejemplo, si preparas el archivo benchmarks.rb y después lo editas, puedes usar git diff para ver las modificaciones del archivo que están preparadas, y las que no lo están:



Ahora puedes usar git diff para ver qué es lo que aún no está preparado:



Ahora puedes usar git diff para ver qué es lo que aún no está preparado:



**Confirmando tus cambios**

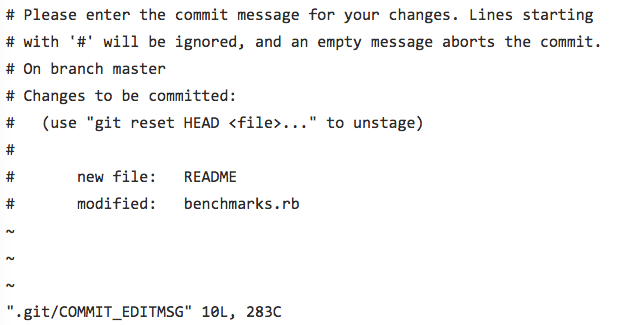
Ahora que el área de preparación está como tú quieres, puedes confirmar los cambios. Recuerda que cualquier cosa que todavía esté sin preparar — cualquier archivo que hayas creado o modificado, y sobre el que no hayas ejecutado git add desde su última edición — no se incluirá en esta confirmación. Se mantendrán como modificados en tu disco.

En este caso, la última vez que ejecutaste git status viste que estaba todo preparado, por lo que estás listo para confirmar tus cambios. La forma más fácil de confirmar es escribiendo git commit:



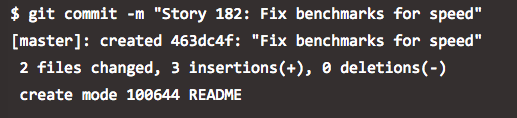
Al hacerlo, se ejecutará tu editor de texto.

El editor mostrará el siguiente texto (este ejemplo usa Vim):



Puedes ver que el mensaje de confirmación predeterminado contiene la salida del comando git status comentada, y una línea vacía arriba del todo. Puedes eliminar estos comentarios y escribir tu mensaje de confirmación, o puedes dejarlos para ayudarte a recordar las modificaciones que estás confirmando. (Para un recordatorio todavía más explícito de lo que has modificado, puedes pasar la opción -v a git commit. Esto provoca que se añadan también las diferencias de tus cambios, para que veas exactamente lo que hiciste.) Cuando sales del editor, Git crea tu confirmación con el mensaje que hayas especificado (omitiendo los comentarios y las diferencias).

Como alternativa, puedes escribir tu mensaje de confirmación desde la propia línea de comandos mediante la opción -m:



¡Acabas de crear tu primera confirmación! Puedes ver que el comando commit ha dado cierta información sobre la confirmación: a qué rama has confirmado (master), cuál es su suma de comprobación SHA-1 de la confirmación (463dc4f), cuántos archivos se modificaron, y estadísticas acerca de cuántas líneas se han añadido y cuántas se han eliminado.

Recuerda que la confirmación registra la instantánea de tu área de preparación. Cualquier cosa que no preparases sigue estando modificada; puedes hacer otra confirmación para añadirla a la historia del proyecto. Cada vez que confirmas, estás registrando una instantánea de tu proyecto, a la que puedes volver o con la que puedes comparar más adelante.

**Trabajando con repositorios remotos**

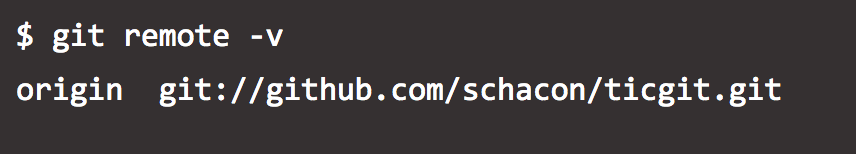
Para poder colaborar en cualquier proyecto Git, necesitas saber cómo gestionar tus repositorios remotos. Los repositorios remotos son versiones de tu proyecto que se encuentran alojados en Internet o en algún punto de la red. Puedes tener varios, cada uno de los cuales puede ser de sólo lectura, o de lectura/escritura, según los permisos que tengas. Colaborar con otros implica gestionar estos repositorios remotos, y mandar (push) y recibir (pull) datos de ellos cuando necesites compartir cosas.

Gestionar repositorios remotos implica conocer cómo añadir repositorios nuevos, eliminar aquellos que ya no son válidos, gestionar ramas remotas e indicar si están bajo seguimiento o no, y más cosas. En esta sección veremos todos estos conceptos.

**Mostrando tus repositorios remotos**

Para ver qué repositorios remotos tienes configurados, puedes ejecutar el comando git remote. Mostrará una lista con los nombres de los remotos que hayas especificado. Si has clonado tu repositorio, deberías ver por lo menos "origin" — es el nombre predeterminado que le da Git al servidor del que clonaste —:

También puedes añadir la opción -v, que muestra la URL asociada a cada repositorio remoto:



Si tienes más de un remoto, este comando los lista todos. Por ejemplo, mi repositorio Grit tiene esta pinta:



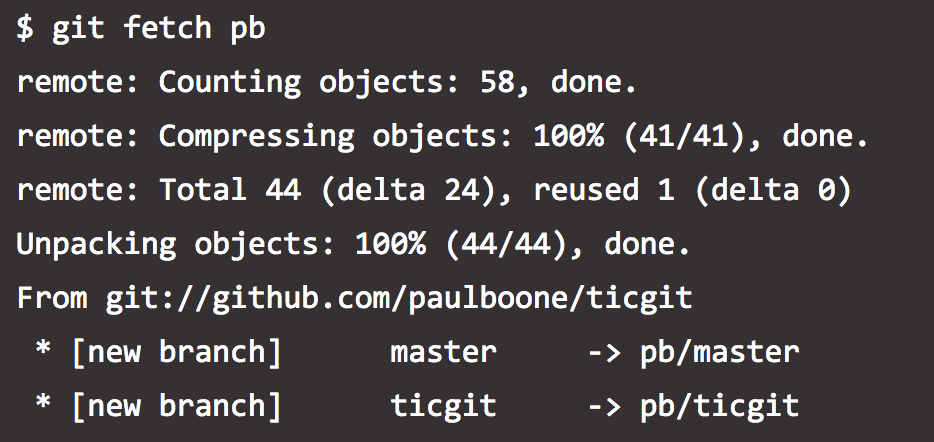
Esto significa que podemos recibir contribuciones de cualquiera de estos usuarios de manera bastante fácil.

**Añadiendo repositorios remotos**

Ya he mencionado y he dado ejemplos de repositorios remotos en secciones anteriores, pero a continuación veremos cómo añadirlos explícitamente. Para añadir un nuevo repositorio Git remoto, asignándole un nombre con el que referenciarlo fácilmente, ejecuta git remote add [nombre] [url]:



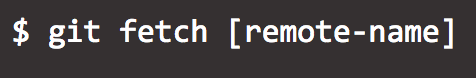
Ahora puedes usar la cadena "pb" en la línea de comandos, en lugar de toda la URL. Por ejemplo, si quieres recuperar toda la información de Paul que todavía no tienes en tu repositorio, puedes ejecutar git fetch pb:



La rama maestra de Paul es accesible localmente como pb/master — puedes unirla a una de tus ramas, o copiarla localmente para inspeccionarla —.

**Recibiendo de tus repositorios remotos**

Como acabas de ver, para recuperar datos de tus repositorios remotos puedes ejecutar:



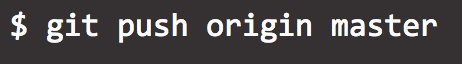
Este comando recupera todos los datos del proyecto remoto que no tengas todavía. Después de hacer esto, deberías tener referencias a todas las ramas del repositorio remoto, que puedes unir o inspeccionar en cualquier momento.

Si clonas un repositorio, el comando añade automáticamente ese repositorio remoto con el nombre de "origin". Por tanto, git fetch origin recupera toda la información enviada a ese servidor desde que lo clonaste (o desde la última vez que ejecutaste fetch). Es importante tener en cuenta que el comando fetch sólo recupera la información y la pone en tu repositorio local — no la une automáticamente con tu trabajo ni modifica aquello en lo que estás trabajando. Tendrás que unir ambos manualmente a posteriori —.

Si has configurado una rama para seguir otra rama remota, puedes usar el comando git pull para recuperar y unir automáticamente la rama remota con tu rama actual. Éste puede resultarte un flujo de trabajo más sencillo y más cómodo; y por defecto, el comando git clone automáticamente configura tu rama local maestra para que siga la rama remota maestra del servidor del cual clonaste (asumiendo que el repositorio remoto tiene una rama maestra). Al ejecutar git pull, por lo general se recupera la información del servidor del que clonaste, y automáticamente se intenta unir con el código con el que estás trabajando actualmente.

**Enviando a tus repositorios remotos**

Cuando tu proyecto se encuentra en un estado que quieres compartir, tienes que enviarlo a un repositorio remoto. El comando que te permite hacer esto es sencillo: git push [nombre-remoto][nombre-rama]. Si quieres enviar tu rama maestra (master) a tu servidor origen (origin), ejecutarías esto para enviar tu trabajo al servidor:



Este comando funciona únicamente si has clonado de un servidor en el que tienes permiso de escritura, y nadie ha enviado información mientras tanto. Si tú y otra persona clonais a la vez, y él envía su información y luego envías tú la tuya, tu envío será rechazado. Tendrás que bajarte primero su trabajo e incorporarlo en el tuyo para que se te permita hacer un envío.

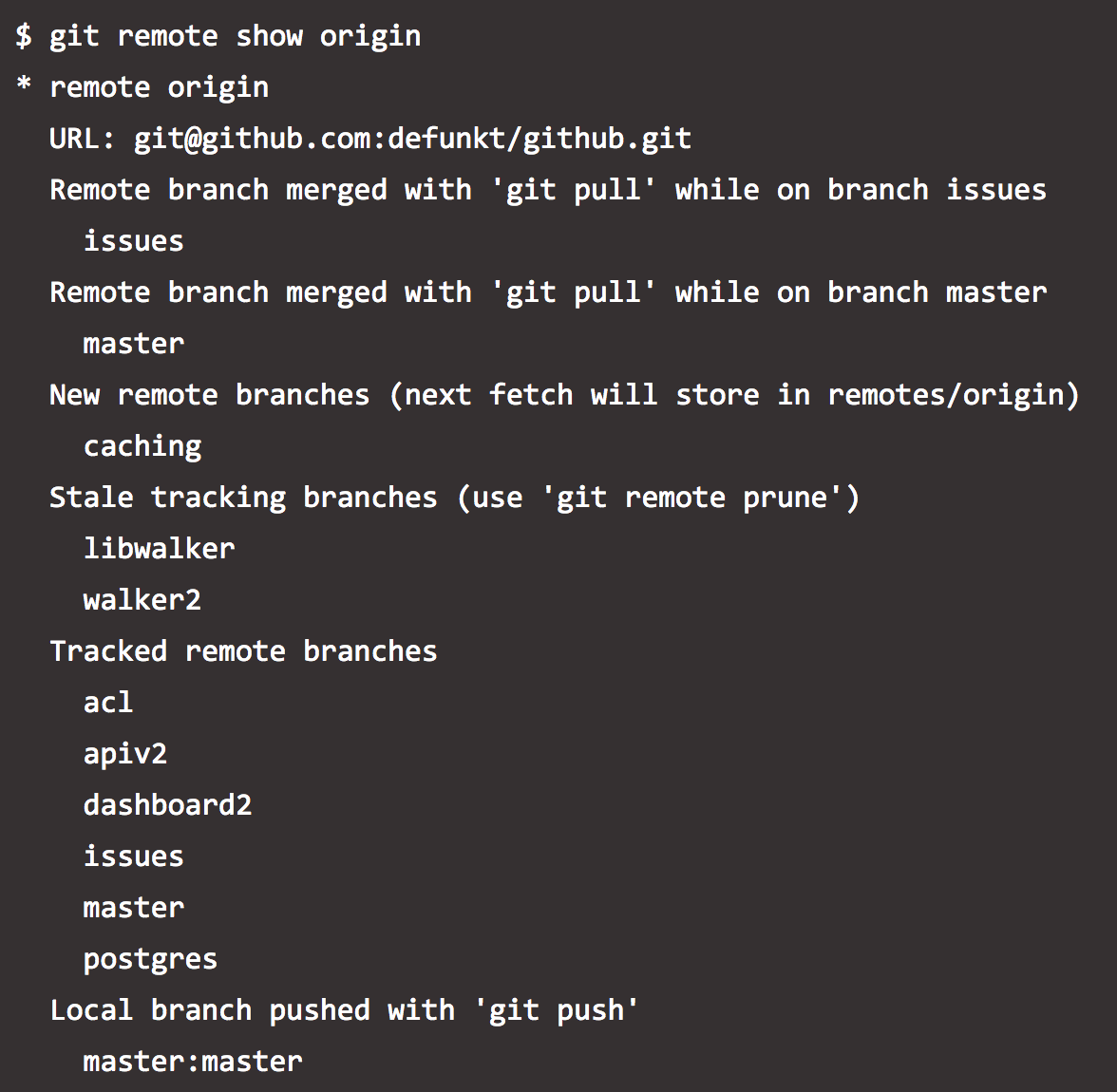
**

Esto lista la URL del repositorio remoto, así como información sobre las ramas bajo seguimiento. Este comando te recuerda que si estás en la rama maestra y ejecutas git pull, automáticamente unirá los cambios a la rama maestra del remoto después de haber recuperado todas las referencias remotas. También lista todas las referencias remotas que ha recibido.

El anterior es un sencillo ejemplo que te encontrarás con frecuencia. Sin embargo, cuando uses Git de forma más avanzada, puede que git remote show muestre mucha más información:

**Inspeccionando un repositorio remoto**

Si quieres ver más información acerca de un repositorio remoto en particular, puedes usar el comando git remote show [nombre]. Si ejecutas este comando pasándole el nombre de un repositorio, como origin, obtienes algo así:



Este comando muestra qué rama se envía automáticamente cuando ejecutas git push en determinadas ramas. También te muestra qué ramas remotas no tienes todavía, qué ramas remotas tienes y han sido eliminadas del servidor, y múltiples ramas que serán unidas automáticamente cuando ejecutes git pull.

## Trabajando con ramas en Git

Cualquier sistema de control de versiones moderno tiene algún mecanismo para soportar distintas ramas. Cuando hablamos de branching o trabajo con ramas, significa que tu has tomado la rama principal de desarrollo (master) y a partir de ahí has continuado trabajando sin seguir la rama principal de desarrollo. En muchos sistemas de control de versiones este proceso es costoso, pues a menudo significa crear una nueva copia del código, lo cual puede requerir mucho tiempo cuando se trata de proyectos grandes.

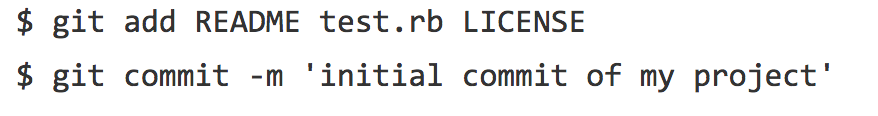
Según muchos desarrolladores, uno de los puntos más fuertes de Git es su sistema de ramificaciones y lo cierto es que esto le hace destacar sobre los otros sistemas de control de versiones. ¿Por qué esto es tan importante? La forma en la que Git maneja las ramificaciones es increíblemente rápida, haciendo así de las operaciones de ramificación algo casi instantáneo, al igual que el avance o el retroceso entre distintas ramas, lo cual también es tremendamente rápido. A diferencia de otros sistemas de control de versiones, Git promueve un ciclo de desarrollo donde las ramas se crean y se unen ramas entre sí, incluso varias veces en el mismo día. Entender y manejar esta opción te proporciona una poderosa y exclusiva herramienta que puede, literalmente, cambiar la forma en la que desarrollas.

**¿Qué es una rama?**

Para entender realmente cómo ramifica Git, previamente hemos de examinar la forma en que almacena sus datos. Git no los almacena de forma incremental (guardando solo diferencias), sino que los almacena como una serie de instantáneas (copias puntuales de los archivos completos, tal y como se encuentran en ese momento).

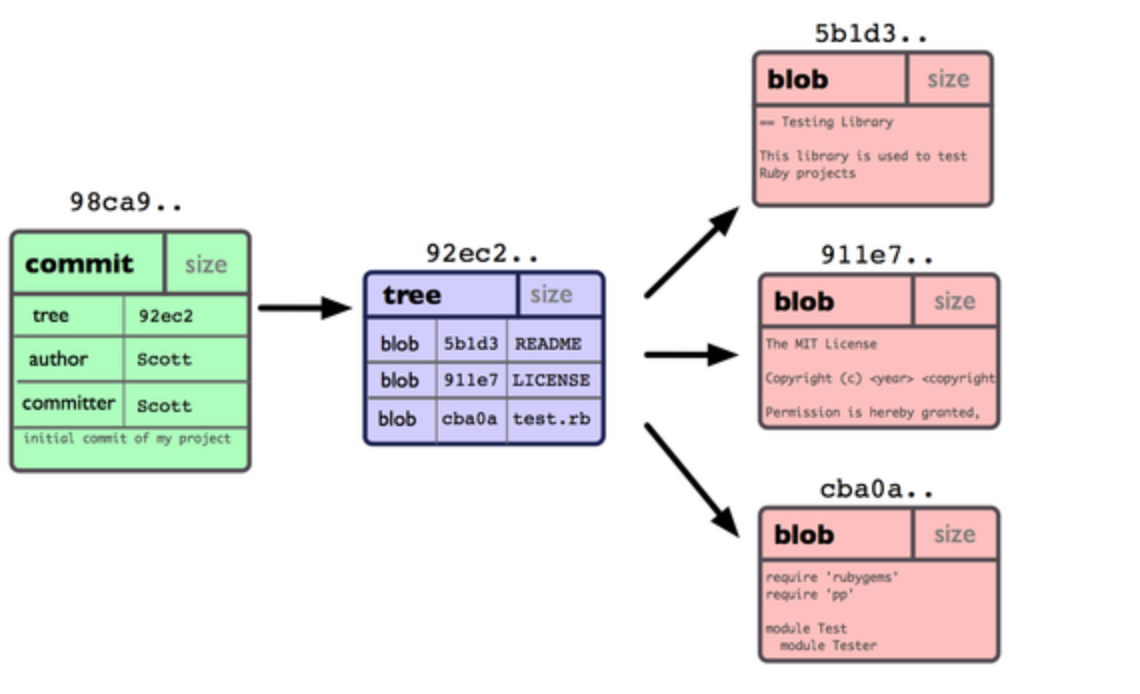
En cada confirmación de cambios (commit), Git almacena un punto de control que conserva: un apuntador a la copia puntual de los contenidos preparados (staged), unos metadatos con el autor y el mensaje explicativo, y uno o varios apuntadores a las confirmaciones (commit) que sean padres directos de esta (un padre en los casos de confirmación normal, y múltiples padres en los casos de estar confirmando una fusión (merge) de dos o mas ramas).

Para ilustrar esto, vamos a suponer, por ejemplo, que tienes una carpeta con tres archivos, que preparas (stage) todos ellos y los confirmas (commit). Al preparar los archivos, Git realiza una suma de control de cada uno de ellos, almacena una copia de cada uno en el repositorio (estas copias se denominan "blobs"), y guarda cada suma de control en el área de preparación (staging area):

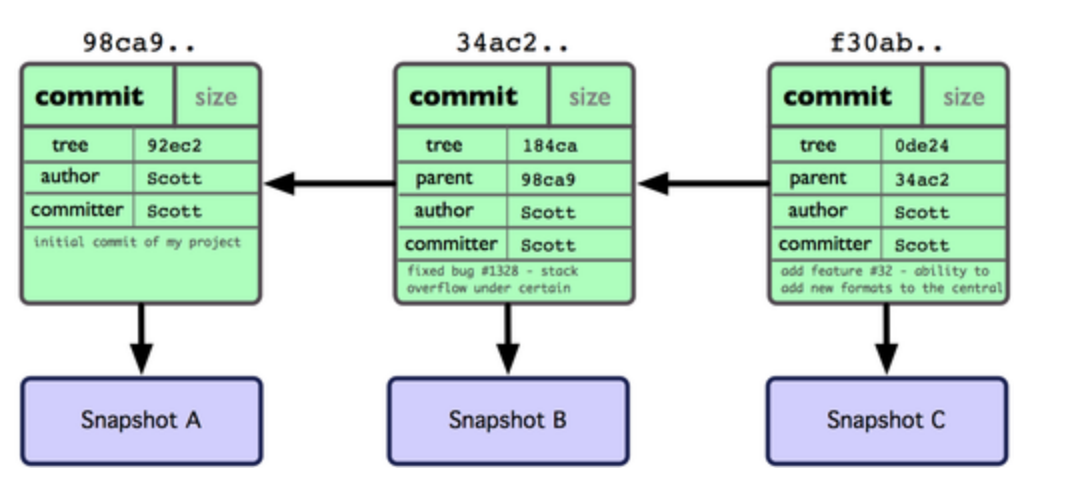


Cuando creas una confirmación con el comando git commit, Git realiza sumas de control de cada subcarpeta (en el ejemplo, solamente tenemos la carpeta principal del proyecto), y guarda en el repositorio Git una copia de cada uno de los archivos contenidos en ella/s. Después, Git crea un objeto de confirmación con los metadatos pertinentes y un apuntador al nodo correspondiente del árbol de proyecto. Esto permitirá poder regenerar posteriormente dicha instantánea cuando sea necesario.

En este momento, el repositorio de Git contendrá cinco objetos: un "blob" para cada uno de los tres archivos, un árbol con la lista de contenidos de la carpeta (más sus respectivas relaciones con los "blobs"), y una confirmación de cambios (commit) apuntando a la raiz de ese árbol y conteniendo el resto de metadatos pertinentes.



Si haces más cambios y vuelves a confirmar, la siguiente confirmación guardará un apuntador a esta su confirmación precedente. Tras un par de confirmaciones más, el registro ha de ser algo parecido a:

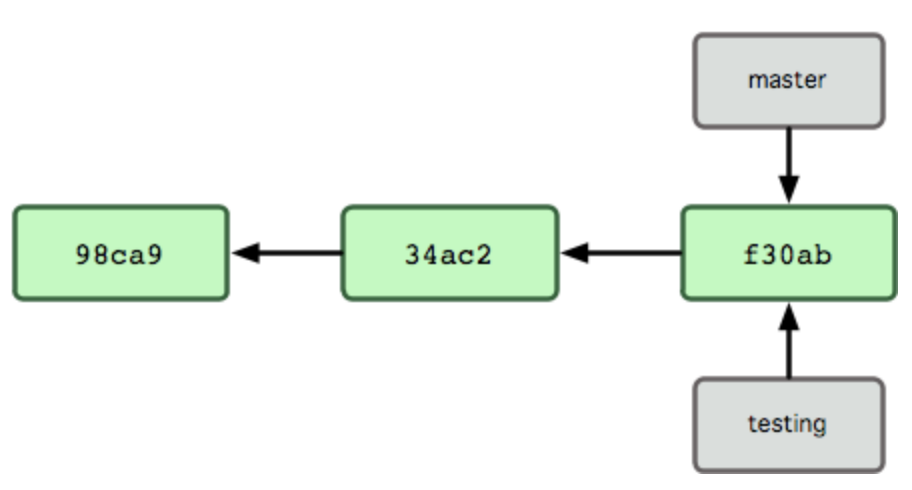


Una rama Git es simplemente un apuntador móvil apuntando a una de esas confirmaciones. La rama por defecto de Git es la rama master. Con la primera confirmación de cambios que realicemos, se creará esta rama principal master apuntando a dicha confirmación. En cada confirmación de cambios que realicemos, la rama irá avanzando automáticamente. Y la rama master apuntará siempre a la última confirmación realizada

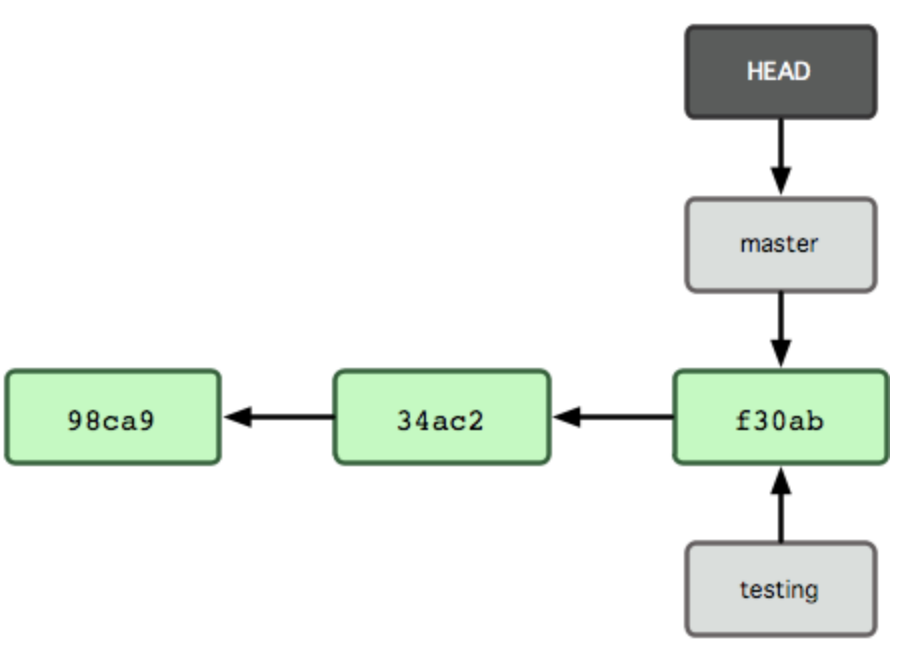
¿Qué sucede cuando creas una nueva rama? Pues que simplemente se crea un nuevo apuntador para que lo puedas mover libremente. Por ejemplo, si quieres crear una nueva rama denominada testing usarás el comando git branch:

git branch testing

Esto creará un nuevo apuntador apuntando al mismo commit donde estés actualmente



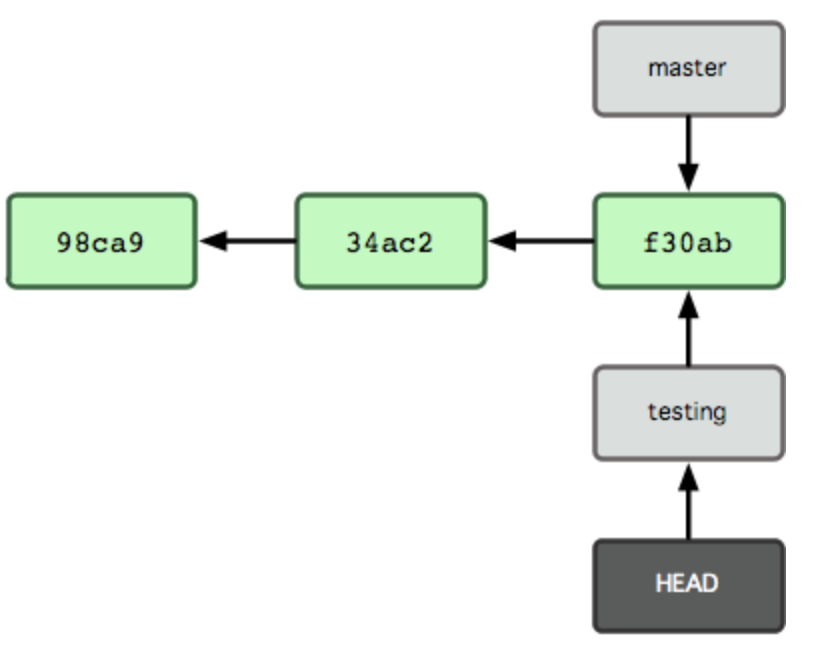
Y, ¿cómo sabe Git en qué rama estás en este momento? Pues mediante un apuntador especial denominado HEAD. Aunque es preciso comentar que este HEAD es totalmente distinto al concepto de HEAD en otros sistemas de control de cambios como Subversion o CVS. En Git, es simplemente el apuntador a la rama local en la que tú estés en ese momento. En este caso, en la rama master. Puesto que el comando git branch solamente crea una nueva rama, y no salta a dicha rama.



Para saltar de una rama a otra, tienes que utilizar el comando git checkout. Hagamos una prueba, saltando a la rama testing recién creada:

$ git checkout testing

Esto mueve el apuntador HEAD a la rama testing



Herramientas

Se recomienda usar como herramienta para el ejecicio https://github.com/.

Ejercicios

Crear cuenta y clonar repositorio que el profesor asigne a cada grupo Github, ademas agregar descripción del proyecto en el archivo README.

