

Control de versiones

Caso práctico

Paco le comenta a **Juan** que una vez elegido el entorno de trabajo para el futuro proyecto, en las distintas reuniones de la empresa se van decidir que varios programadores trabajen en implementar código dentro un mismo fichero. Le informa que la empresa utiliza el sistema de control de versiones Git para no perder el control del desarrollo.

Juan le comenta a **Paco** que conoce este sistema, pero que repasará todos los aspectos importantes para ponerlo en práctica en la empresa.

Reflexiona

¿Qué ocurriría si todos los programadores empiezan a escribir código en el mismo fichero del proyecto sin ningún control?

[Mostrar retroalimentación](#)

Citas Para Pensar

“

Hay una fuerza motriz más poderosa que el vapor, la electricidad y la energía atómica: la voluntad.

Albert Einstein

1.- Introducción



Licencia: Dominio público

Paco pretende enseñar poco a poco a **Juan** como se organizan en la fase de codificación utilizando Git. **Juan** le informa que repasará primero los conceptos más importantes de un sistema de control de versiones para a continuación centrarse en Git y GitHub.

Obra publicada con [Licencia Creative Commons Reconocimiento Compartir igual 4.0](#)

1.1.- ¿Qué es un sistema de control de versiones?



Un sistema de control de versiones o **SVC (System Versión Control)** es una herramienta que registra los cambios realizados sobre un archivo o conjunto de archivos de un proyecto a lo largo del tiempo, de modo que puedas recuperar la versión de esos archivos en un estado anterior en el tiempo.

El sistema de control de versiones es de gran utilidad para entornos de desarrollo colaborativo donde varios diseñadores, maquetadores y/o programadores trabajan sobre un mismo proyecto. En todo momento tenemos los cambios que se realizan sobre cada uno de los ficheros por los diferentes programadores y podemos recuperar un fichero en un estado anterior.

Ejemplos de este tipo de herramientas son entre otros: CVS, Subversion, SourceSafe, ClearCase, Darcs, Bazaar, Plastic SCM, Git, SCCS, Mercurial, Perforce, Fossil SCM, Team Foundation Server.

El control de versiones se realiza principalmente en la industria informática para controlar las distintas versiones del código fuente dando lugar a los **sistemas de control de código fuente** o **SCM** (siglas del inglés *Source Code Management*). Sin embargo, los mismos conceptos son aplicables a otros ámbitos como en trabajo de oficina tales como documentos, imágenes, sitios web, etc.

Obra publicada con [Licencia Creative Commons Reconocimiento Compartir igual 4.0](#)

1.2.- Características de estos sistemas



Deberá ser un sistema que registra los cambios realizados sobre un archivo o conjunto de archivos a lo largo del tiempo, de modo que se pueda recuperar versiones específicas de los mismos en un determinado momento.

Un SVC posee tres capacidades importantes:

- **Reversibilidad:** retornar a un estado anterior del proyecto en caso de fallos.
- **Concurrencia:** Muchas personas modificando el mismo código o documento.
- **Anotación:** Adjuntar información relevante de los cambios realizados.

Un sistema de control de versiones debe proporcionar:

- Mecanismo de almacenamiento de los elementos que deba gestionar (ej. archivos de texto, imágenes, documentación...).
- Posibilidad de realizar cambios sobre los elementos almacenados (ej. modificaciones parciales, añadir, borrar, renombrar o mover elementos).
- Registro histórico de las acciones realizadas con cada elemento o conjunto de elementos (normalmente pudiendo volver o extraer un estado anterior del producto).

1.3.- Ventajas de estos sistemas

Las ventajas que se obtienen de utilizar un sistemas control de versiones serían las siguientes:

Creación de flujos de trabajo

Los flujos de trabajo de control de versiones impiden el caos de todos los usuarios que usan su propio proceso de desarrollo con herramientas diferentes e incompatibles. Los sistemas de control de versiones proporcionan permisos y cumplimiento de procesos para que todos permanezcan en la misma página.

Trabajo con versiones

Cada versión tiene una descripción para lo que hacen los cambios en la versión, como corregir un error o agregar una característica. Estas descripciones ayudan al equipo a seguir los cambios en el código por versión en lugar de por cambios de archivo individuales. El código almacenado en versiones se puede ver y restaurar desde el control de versiones en cualquier momento y según sea necesario. Las versiones facilitan la creación de nuevas versiones de cualquier versión del código.

Código juntos

El control de versiones sincroniza las versiones y se asegura de que los cambios no entren en conflicto con los cambios de otros usuarios. El equipo se basa en el control de versiones para ayudar a resolver y evitar conflictos, incluso cuando los usuarios realizan cambios al mismo tiempo.

Mantener un historial

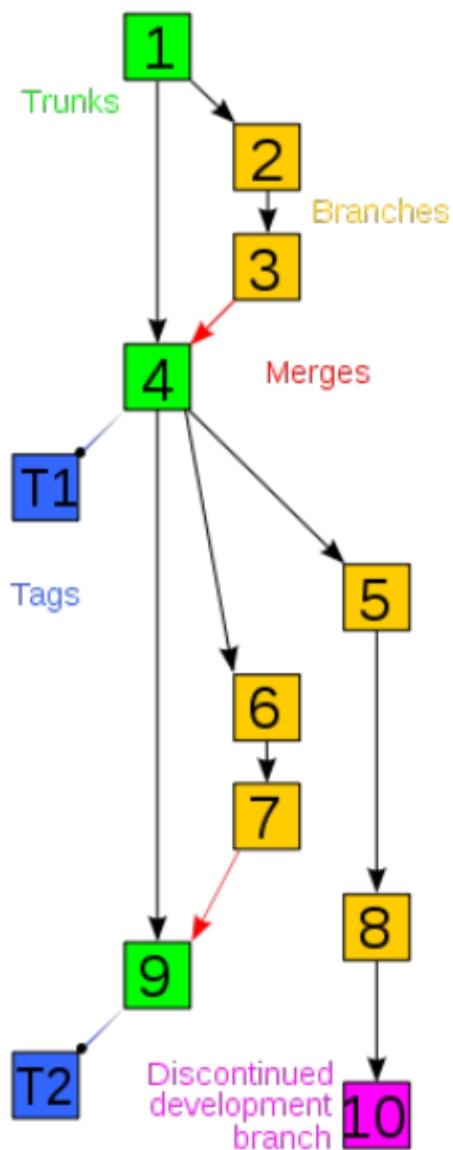
El control de versiones mantiene un historial de cambios a medida que el equipo guarda nuevas versiones de código. Los miembros del equipo pueden revisar el historial para averiguar quién, por qué y cuándo se realizaron cambios. El historial proporciona a los equipos la confianza de experimentar, ya que es fácil revertir a una versión correcta anterior en cualquier momento. El historial permite a cualquier persona basar el trabajo desde cualquier versión del código, por ejemplo, para corregir un error en una versión anterior.

Automatización de tareas

Las características de automatización del control de versiones ahorran tiempo y generan resultados coherentes. Automatizar las pruebas, el análisis de código y la implementación cuando se guardan nuevas versiones en el control de versiones son tres ejemplos.

Obra publicada con [Licencia Creative Commons Reconocimiento Compartir igual 4.0](#)

1.4.- Conceptos básicos



El sistema de control de versiones está formado por un conjunto de componentes:

- ✓ **Repositorio:** Lugar en el que se almacenan los datos actualizados e históricos de cambios (sistema de archivos en un disco duro, un banco de datos, etc).
- ✓ **Módulo:** en un directorio específico del repositorio. Puede identificar una parte del proyecto o ser el proyecto por completo.
- ✓ **Revisión:** es cada una de las versiones parciales o cambios en los archivos o repositorio completo. La evolución del sistema se mide en revisiones. Cada cambio se considera incremental.
- ✓ **Etiqueta (tag):** información textual que se añada a un conjunto de archivos o a un módulo completo para indicar alguna información importante.
- ✓ **Rama (branch):** revisiones paralelas de un módulo para efectuar cambios sin tocar la evolución principal. Se suele emplear para pruebas o para mantener los cambios en versiones antiguas.
- ✓ **Baseline:** Una revisión aprobada de un documento o fichero fuente, a partir del cual se pueden realizar cambios subsiguientes.

Las órdenes que se pueden ejecutar son:

- ✓ **checkout / clone:** Obtiene una copia del trabajo para poder trabajar con ella.
- ✓ **update:** Actualiza la copia con cambios recientes en el repositorio.
- ✓ **commit:** Almacena la copia modificada en el repositorio.
- ✓ **abort:** Abandona los cambios en la copia de trabajo.
- ✓ **merge:** Une dos grupos de cambios en un archivo (o grupo de archivos), generando una revisión unificada.
- ✓ **change set:** Conjunto de cambios realizados en un único commit.

Conflicto: Sigue cuando dos o más personas intentan realizar diferentes cambios en la misma porción de código.

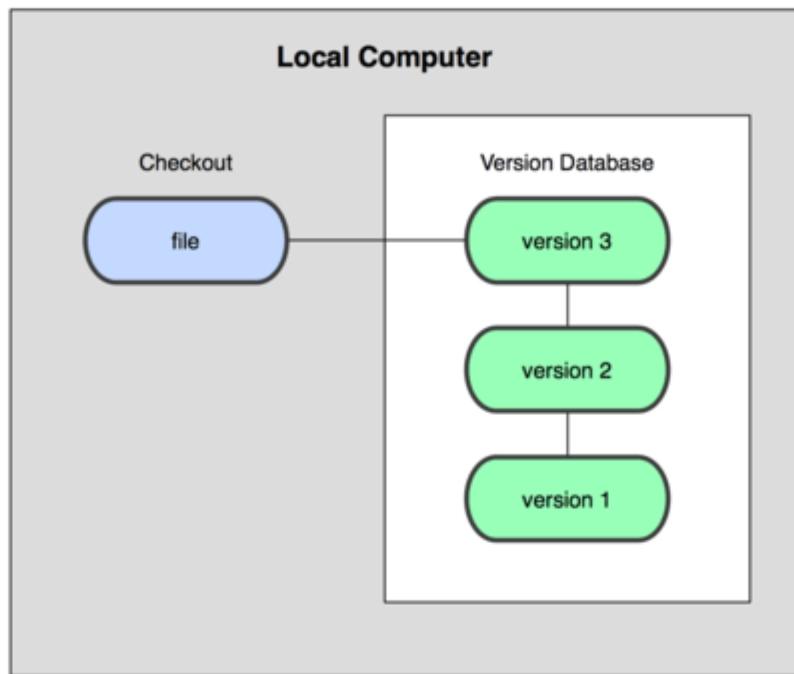
Obra publicada con [Licencia Creative Commons Reconocimiento Compartir igual 4.0](#)

1.5.- Clasificación de los sistemas

Podemos clasificar los sistemas de control de versiones según la arquitectura para almacenar la información en locales, centralizados o distribuidos.

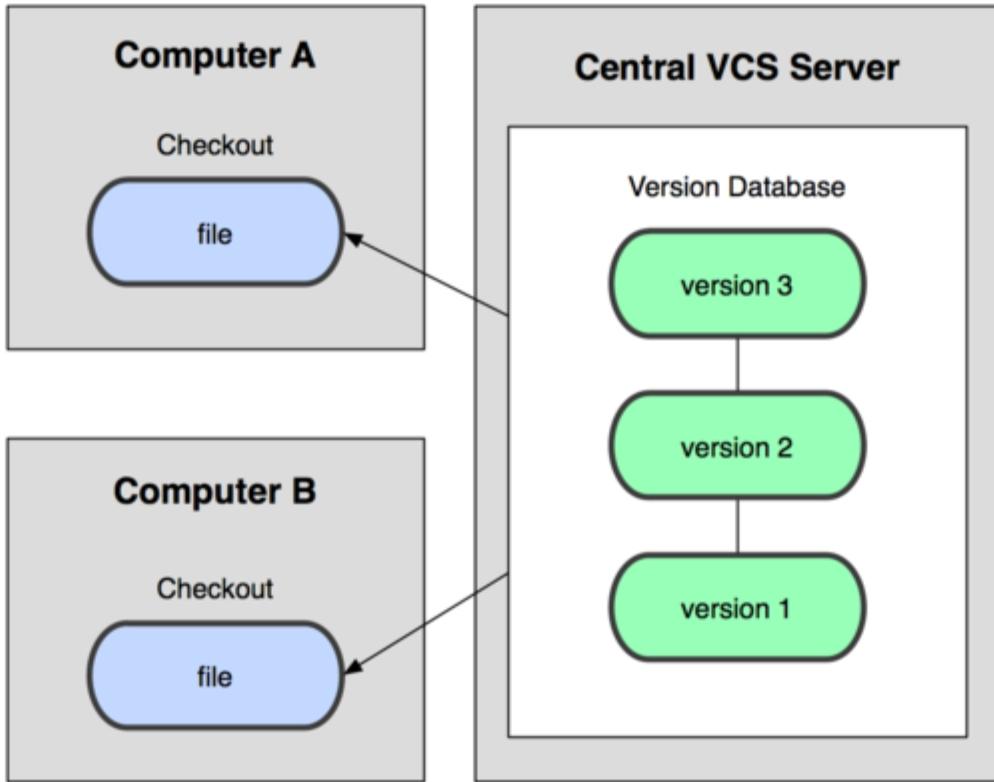
Locales

La información se guarda en un ordenador o repositorio local con lo que no sirve para trabajar en forma colaborativa. Ejemplo: RCS (Revision Control System).



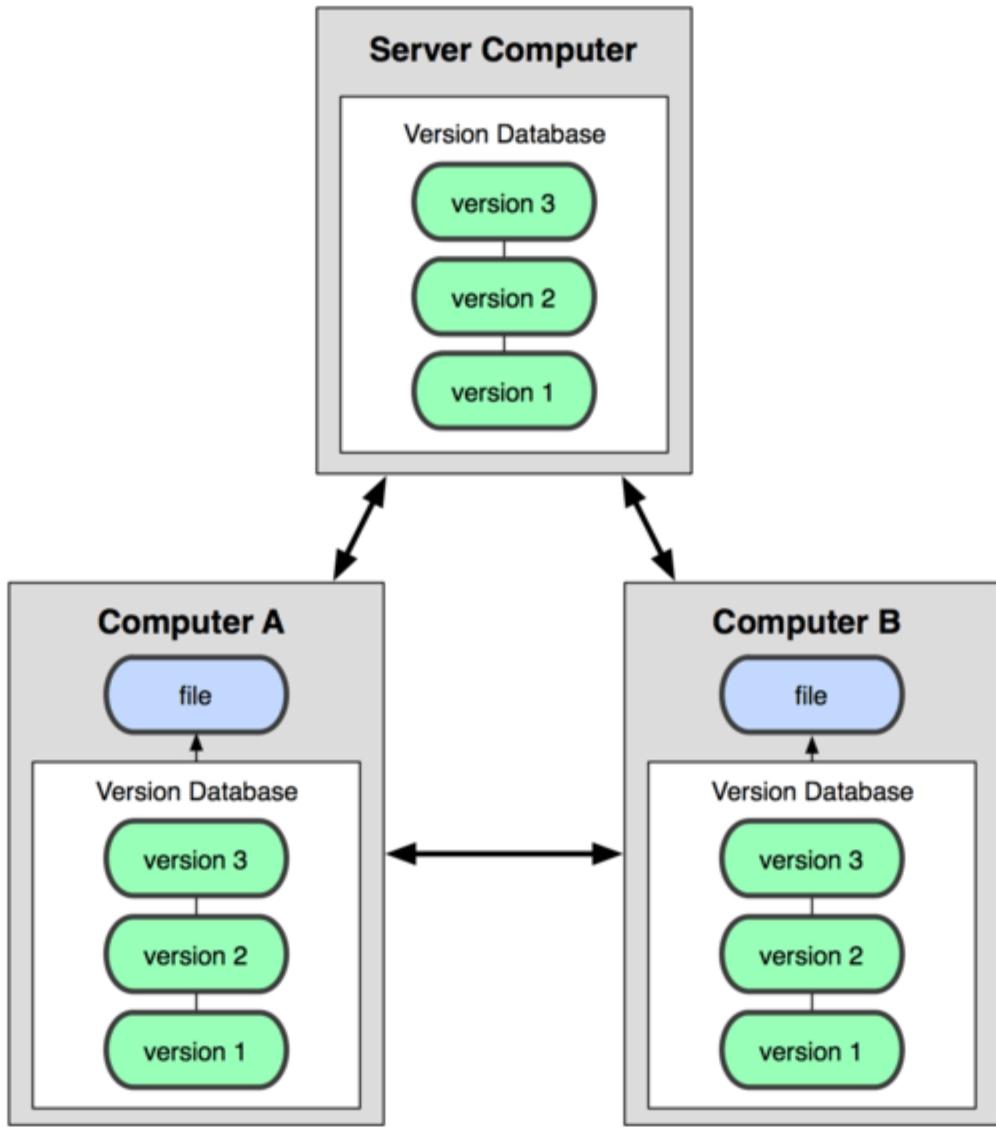
Centralizados

La información se guarda en un servidor dentro de un repositorio centralizado. Existe un usuario o usuarios responsables con capacidad de realizar tareas administrativas a cambio de reducir flexibilidad, necesitan la aprobación del responsable para realizar acciones, como crear una rama nueva. Ejemplos: Subversión y CVS.



Distribuido

Cada usuario tiene su propio repositorio. Los distintos repositorios pueden intercambiar y mezclar revisiones entre ellos. Es frecuente el uso de un repositorio central, que está normalmente disponible, que sirve de punto de sincronización de los distintos repositorios locales. Ejemplos: **Git** y Mercurial, Bazaar y Darcs.



Obra publicada con [Licencia Creative Commons Reconocimiento Compartir igual 4.0](#)

2.- Git

En esta sección aprenderemos a conocer el sistema de control de versiones Git ¿Estás listo? Adelante....

Nota importante: Toda la información que se ofrece en esta sección viene reflejada en las documentaciones oficiales <https://git-scm.com/book/es/v2>, <https://docs.github.com/es>, con pequeñas modificaciones adaptadas a las nuevas versiones que utilizaremos para las prácticas. Sólo se han incluido los conceptos y comandos que entendemos son claves para empezar a utilizar de forma profesional este sistema de control de versiones. Si en un momento determinado no aparece el procedimiento o comandos para realizar alguna acción, habría que investigarlo en la documentación oficial.

Obra publicada con [Licencia Creative Commons Reconocimiento Compartir igual 4.0](#)

2.1.- Fundamentos de Git



Hoy en día, Git es, con diferencia, el sistema de control de versiones moderno más utilizado del mundo. Git es un proyecto de código abierto maduro y con un mantenimiento activo que desarrolló originalmente Linus Torvalds, el famoso creador del kernel del sistema operativo Linux, en 2005. Un asombroso número de proyectos de software dependen de Git para el control de versiones, incluidos proyectos comerciales y de código abierto. Los desarrolladores que han trabajado con Git cuentan con una buena representación en la base de talentos disponibles para el desarrollo de software, y este sistema funciona a la perfección en una amplia variedad de sistemas operativos e IDE (entornos de desarrollo integrados).

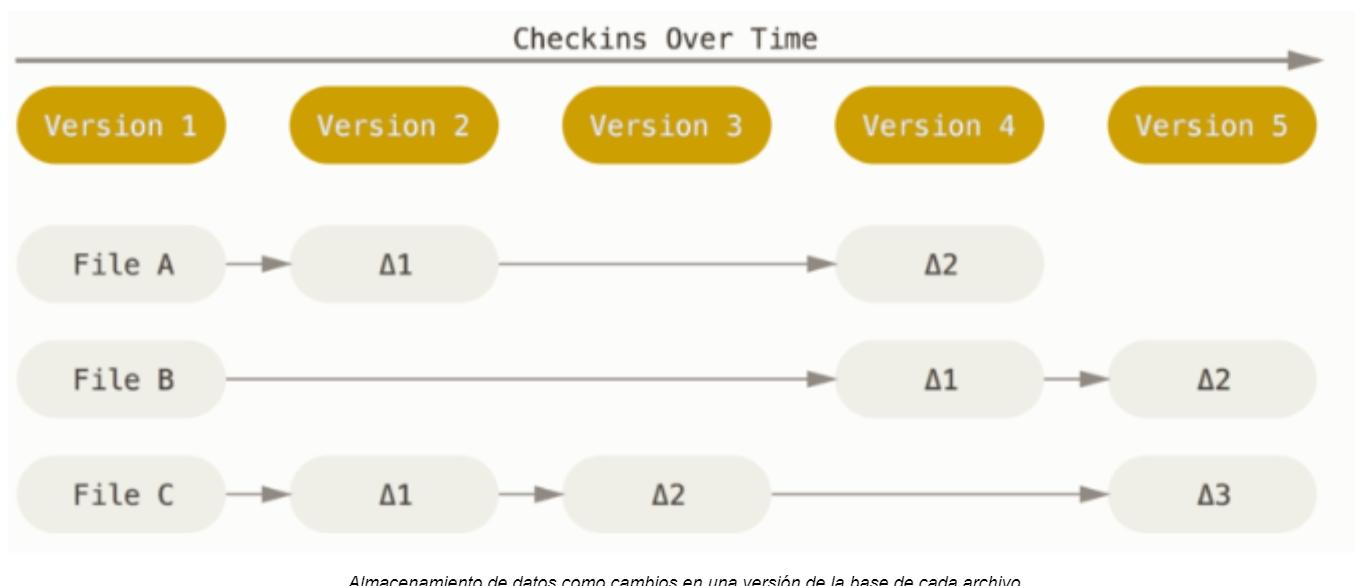
Git, que presenta una arquitectura distribuida, es un ejemplo de DVCS (sistema de control de versiones distribuido, por sus siglas en inglés). En lugar de tener un único espacio para todo el historial de versiones del software, como sucede de manera habitual en los sistemas de control de versiones antaño populares, como CVS o Subversion (también conocido como SVN), en Git, la copia de trabajo del código de cada desarrollador es también un repositorio que puede albergar el historial completo de todos los cambios.

Además de contar con una arquitectura distribuida, Git se ha diseñado teniendo en cuenta el rendimiento, la seguridad y la flexibilidad.

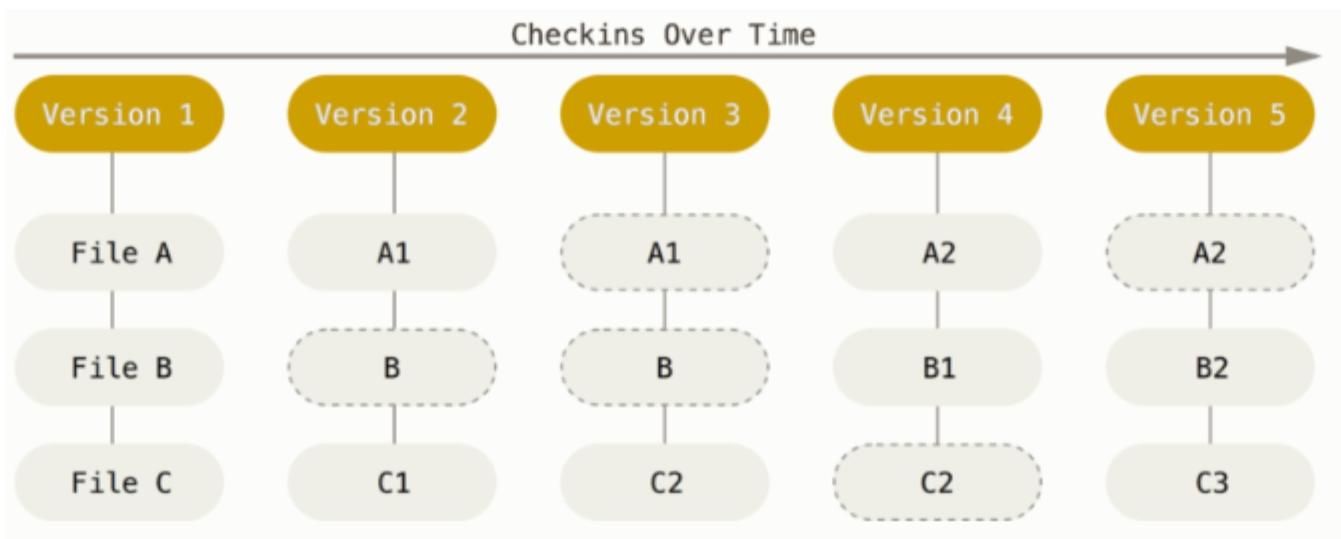
Es muy importante entender bien esta sección, porque si entiendes lo que es Git y los fundamentos de cómo funciona, probablemente te será mucho más fácil usar Git efectivamente. A medida que aprendas Git, intenta olvidar todo lo que posiblemente conoces acerca de otros VCS como Subversion y Perforce. Hacer esto te ayudará a evitar confusiones sutiles a la hora de utilizar la herramienta. Git almacena y maneja la información de forma muy diferente a esos otros sistemas, a pesar de que su interfaz de usuario es bastante similar. Comprender esas diferencias evitirá que te confundas a la hora de usarlo.

2.1.1.- Instantáneas no diferencias

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



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



Almacenamiento de datos como instantáneas del proyecto a través del tiempo

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

Obra publicada con [Licencia Creative Commons Reconocimiento Compartir igual 4.0](#)

2.1.2.- Algunos beneficios

Casi todas las operaciones son locales

La mayoría de las operaciones en Git sólo necesitan archivos y recursos locales para funcionar. Por lo general no se necesita información de ningún otro computador de tu red. Si estás acostumbrado a un CVCS donde la mayoría de las operaciones tienen el costo adicional del retardo de la red, este aspecto de Git hace que notes un buen rendimiento en las modificaciones. Debido a que 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 conectarse 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 instantáneamente. Si quieres ver los cambios introducidos en un archivo entre la versión actual y la de hace un mes, Git puede buscar el archivo de hace un mes y hacer un cálculo de diferencias localmente, en lugar de tener que pedirle a un servidor remoto que lo haga, u obtener una versión antigua desde la red y hacerlo de manera local.

Esto también significa que hay muy poco que no puedes hacer si estás desconectado o sin VPN. Si te subes a un avión o a un tren y quieres trabajar un poco, puedes confirmar tus cambios felizmente hasta que consigas una conexión de red para subirlos. 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 engorroso. En Perforce, por ejemplo, no puedes hacer mucho cuando no estás conectado al servidor. En Subversion y CVS, puedes editar archivos, pero no puedes confirmar 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.

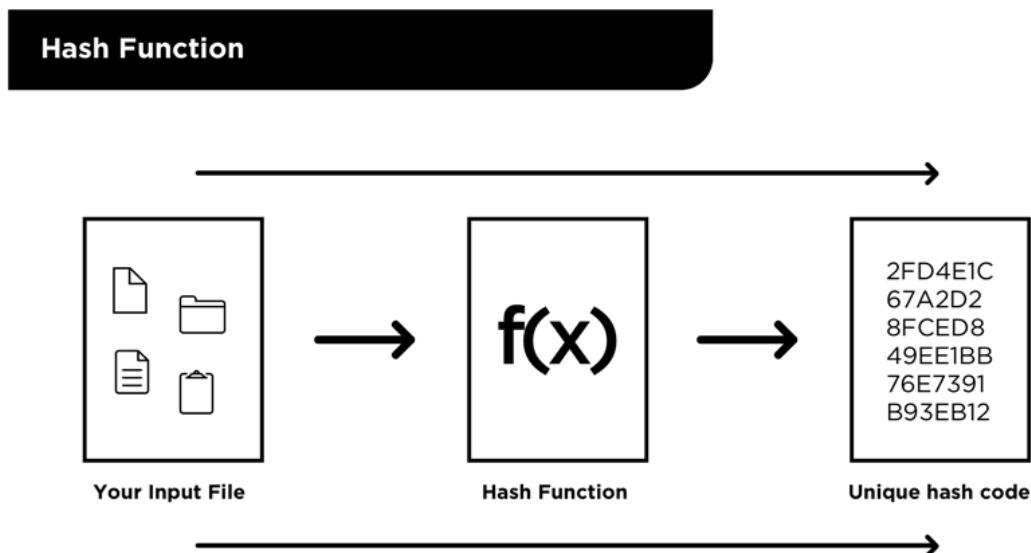
Git tiene integridad

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

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

24b9da6552252987aa493b52f8696cd6d3b00373

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



Git generalmente solo 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 enmendar, 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 copia instantánea en Git es muy difícil perderla, especialmente si envías 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 estropear gravemente las cosas.

Obra publicada con [Licencia Creative Commons Reconocimiento Compartir igual 4.0](#)

2.1.3.- Estados

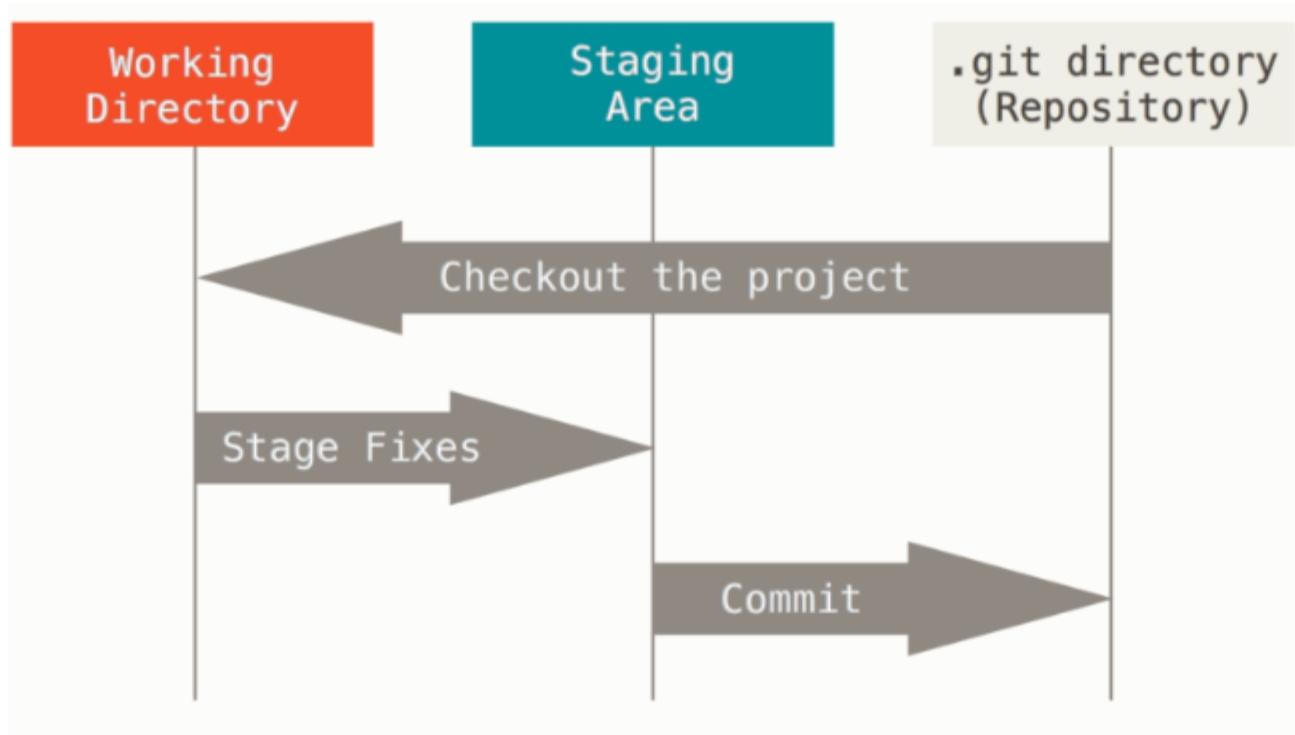
Ahora presta atención. Esto es lo más importante que debes recordar acerca de Git siquieres que el resto de tu proceso de aprendizaje prosiga sin problemas. Git tiene tres estados principales en los que se pueden encontrar tus archivos: confirmado (committed), modificado (modified), y preparado (staged).

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).



Directorio de trabajo, área de almacenamiento y el directorio Git

El directorio de Git es donde se almacenan los metadatos y la base de datos de objetos para tu proyecto. Es la parte más importante de Git, y es lo que se copia cuando clonas un repositorio desde otra computadora.

El directorio de trabajo es una copia de una versión del proyecto. Estos archivos se sacan de la base de datos comprimida en el directorio de Git, y se colocan en disco para que los puedas usar o modificar.

El área de preparación es un archivo, generalmente contenido en tu directorio de Git, que almacena información acerca de lo que va a ir en tu próxima confirmación. A veces se le denomina índice (“index”), pero se está convirtiendo en estándar el referirse a ella 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, añadiéndolos a tu área de preparación.
3. Confirmas los cambios, lo que toma los archivos tal y como están en el área de preparación y almacena esa copia instantánea de manera permanente en tu directorio de Git.

Si una versión concreta de un archivo está en el directorio de Git, se considera confirmada (committed). Si ha sufrido cambios desde que se obtuvo del repositorio, pero ha sido añadida al área de preparación, está preparada (staged). Y si ha sufrido cambios desde que se obtuvo del repositorio, pero no se ha preparado, está modificada (modified).

2.1.4.- Actividad de profundización

Introducción a la herramienta Git

Licencia: Dominio público

Obra publicada con [Licencia Creative Commons Reconocimiento Compartir igual 4.0](#)

2.2.- Instalación y configuración básica

En esta sección aprenderemos a instalar Git en distintas plataformas y desde distintas ubicaciones. ¿Estás listo? Adelante....

Obra publicada con [Licencia Creative Commons Reconocimiento Compartir igual 4.0](#)

2.2.1.- Instalación de Git

Instalación en Linux

Si quieres instalar Git en Linux a través de un instalador binario, en general puedes hacerlo mediante la herramienta básica de administración de paquetes que trae tu distribución. Si estás en Fedora por ejemplo, puedes usar yum:

```
$ yum install git
```

Si estás en una distribución basada en Debian como Ubuntu, puedes usar apt-get:

```
$ apt-get install git
```

Para opciones adicionales, la página web de Git tiene instrucciones de instalación en diferentes tipos de Unix. Puedes encontrar esta información en <http://git-scm.com/download/linux>.

Instalación en Mac

Hay varias maneras de instalar Git en un Mac. Probablemente la más sencilla es instalando las herramientas Xcode de Línea de Comandos. En Mavericks (10.9 o superior) puedes hacer esto desde el Terminal si intentas ejecutar *git* por primera vez. Si no lo tienes instalado, te preguntará si deseas instalarlo.

Si deseas una versión más actualizada, puedes hacerlo a partir de un instalador binario. Un instalador de Git para OSX es mantenido en la página web de Git. Lo puedes descargar en <http://git-scm.com/download/mac>.

También puedes instalarlo como parte del instalador de Github para Mac. Su interfaz gráfica de usuario tiene la opción de instalar las herramientas de línea de comandos. Puedes descargar esa herramienta desde el sitio web de Github para Mac en <http://mac.github.com>.

Instalación en Windows

También hay varias maneras de instalar Git en Windows. La forma más oficial está disponible para ser descargada en el sitio web de Git. Solo tienes que visitar <http://git-scm.com/download/win> y la descarga empezará automáticamente.

Existe un proyecto conocido como Git para Windows (también llamado msysGit), el cual es diferente de Git. Para más información acerca de este proyecto visita <http://msysgit.github.io/>.

Otra forma de obtener Git fácilmente es mediante la instalación de GitHub para Windows. El instalador incluye la versión de línea de comandos y la interfaz de usuario de Git. Además funciona bien con Powershell y establece correctamente "caching" de credenciales y configuración CRLF adecuada. Aprenderemos acerca de todas estas cosas un poco más adelante, pero por ahora es suficiente mencionar que éstas son cosas que deseas. Puedes descargar este instalador del sitio web de GitHub para Windows en <http://windows.github.com>.

Instalación a partir del Código Fuente

Algunas personas desean instalar Git a partir de su código fuente debido a que obtendrán una versión más reciente. Los instaladores binarios tienden a estar un poco atrasados. Sin embargo, esto ha hecho muy poca diferencia a medida que Git ha madurado en los últimos años.

Para instalar Git desde el código fuente necesitas tener las siguientes librerías de las que Git depende: curl, zlib, openssl, expat y libiconv. Por ejemplo, si estás en un sistema que tiene yum (como Fedora) o apt-get (como un sistema basado en Debian), puedes usar estos comandos para instalar todas las dependencias:

```
$ yum install curl-devel expat-devel gettext-devel \
openssl-devel zlib-devel
$ apt-get install libcurl4-gnutls-dev libexpat1-dev gettext \
libz-dev libssl-dev
```

Cuando tengas todas las dependencias necesarias, puedes descargar la versión más reciente de Git en diferentes sitios. Puedes obtenerla a partir del sitio Kernel.org en <https://www.kernel.org/pub/software/scm/git>, o su "mirror" en el sitio web de GitHub en <https://github.com/git/git/releases>. Generalmente la más reciente versión en la página web de GitHub es un poco mejor, pero la página de kernel.org también tiene ediciones con firma en caso de que deseas verificar tu descarga.

Luego tienes que compilar e instalar de la siguiente manera:

```
$ tar -zxf git-2.0.0.tar.gz
$ cd git-2.0.0
$ make config
$ ./configure --prefix=/usr
$ make all doc info
$ sudo make install install-doc install-html install-info
```

Una vez hecho esto, también puedes obtener Git, a través del propio Git, para futuras actualizaciones:

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
```

Obra publicada con [Licencia Creative Commons Reconocimiento Compartir igual 4.0](#)

2.2.2.- Línea de comandos y GUI

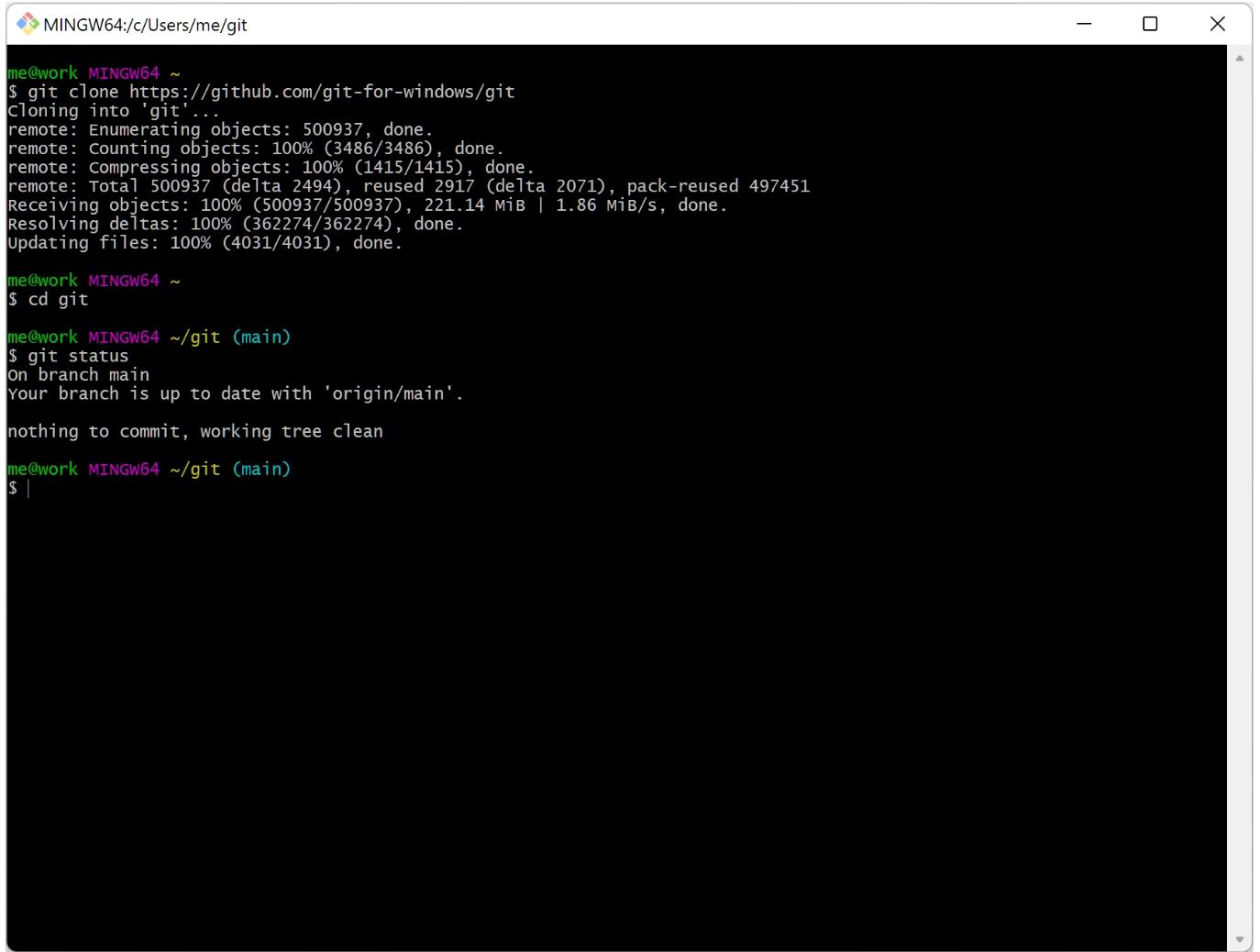
La Línea de Comandos

Existen muchas formas de usar Git. Por un lado tenemos las herramientas originales de línea de comandos, y por otro lado tenemos una gran variedad de interfaces de usuario con distintas capacidades. En este libro vamos a utilizar Git desde la línea de comandos. La línea de comandos es el único lugar en donde puedes ejecutar **todos** los comandos de Git - la mayoría de interfaces gráficas de usuario solo implementan una parte de las características de Git por motivos de simplicidad. Si tú sabes cómo realizar algo desde la línea de comandos, seguramente serás capaz de averiguar cómo hacer lo mismo desde una interfaz gráfica. Sin embargo, la relación opuesta no es necesariamente cierta. Así mismo, la decisión de qué cliente gráfico utilizar depende totalmente de tu gusto, pero *todos* los usuarios tendrán las herramientas de línea de comandos instaladas y disponibles.

Git para Windows

Git BASH

Git para Windows proporciona una emulación de BASH que se usa para ejecutar Git desde la línea de comandos. *Los usuarios de NIX deben sentirse como en casa, ya que la emulación BASH se comporta como el comando "git" en entornos LINUX y UNIX.



A screenshot of a terminal window titled "MINGW64:/c/Users/me/git". The window shows the following command-line session:

```
me@work MINGW64 ~
$ git clone https://github.com/git-for-windows/git
Cloning into 'git'...
remote: Enumerating objects: 500937, done.
remote: Counting objects: 100% (3486/3486), done.
remote: Compressing objects: 100% (1415/1415), done.
remote: Total 500937 (delta 2494), reused 2917 (delta 2071), pack-reused 497451
Receiving objects: 100% (500937/500937), 221.14 MiB | 1.86 MiB/s, done.
Resolving deltas: 100% (362274/362274), done.
Updating files: 100% (4031/4031), done.

me@work MINGW64 ~
$ cd git

me@work MINGW64 ~/git (main)
$ git status
On branch main
Your branch is up to date with 'origin/main'.

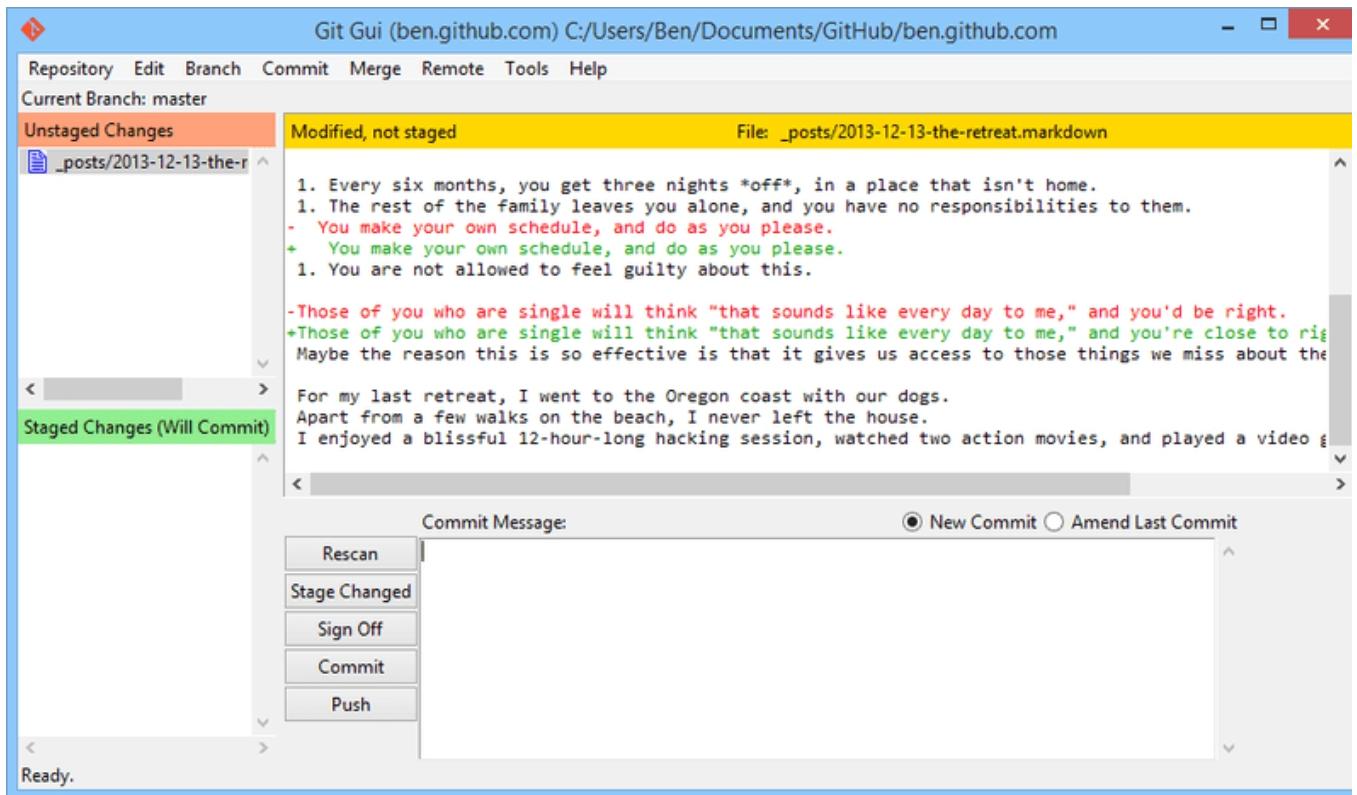
nothing to commit, working tree clean

me@work MINGW64 ~/git (main)
$ |
```

Terminal Git BASH

Interfaz gráfica de usuario Git

Como los usuarios de Windows suelen esperar interfaces gráficas de usuario, Git para Windows también proporciona la GUI de Git, una poderosa alternativa a Git BASH, que ofrece una versión gráfica de casi todas las funciones de la línea de comandos de Git, así como herramientas integrales de diferencias visuales.



GUI Git BASH para Windows

Integración de shell

Simplemente se puede hacer click derecho en una carpeta en el Explorador de Windows para acceder a BASH o GUI.

Obra publicada con [Licencia Creative Commons Reconocimiento Compartir igual 4.0](#)

2.2.3.- Configurando Git por primera vez

Linux

Ahora que tienes Git en tu sistema, vas a querer hacer algunas cosas para personalizar tu entorno de Git. Es necesario hacer estas cosas solamente una vez en tu computadora, y 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:

1. 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.
2. Archivo `~/.gitconfig` O `~/.config/git/config`: Este archivo es específico de tu usuario. Puedes hacer que Git lea y escriba específicamente en este archivo pasando la opción `--global`.
3. Archivo `config` en el directorio de Git (es decir, `.git/config`) del repositorio que estés utilizando actualmente: Este archivo es específico del repositorio actual.

Cada nivel sobrescribe los valores del nivel anterior, por lo que los valores de `.git/config` tienen preferencia sobre los de `/etc/gitconfig`.

Windows

En sistemas Windows, Git busca el archivo `.gitconfig` en el directorio `$HOME` (para mucha gente será `(C:\Users\$USER)`). También busca el archivo `/etc/gitconfig`, aunque esta ruta es relativa a la raíz MSys, que es donde decidiste instalar Git en tu sistema Windows cuando ejecutaste el instalador.

Tu Identidad

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

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

De nuevo, sólo necesitas hacer esto una vez si especificas la opción `--global`, ya que Git siempre usará esta información para todo lo que hagas en ese sistema. Si quieres sobrescribir esta información con otro nombre o dirección de correo para proyectos específicos, puedes ejecutar el comando sin la opción `--global` cuando estés en ese proyecto.

Muchas de las herramientas de interfaz gráfica te ayudarán a hacer esto la primera vez que las uses.

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 usará el editor por defecto de tu sistema, que generalmente es Vim. Si quieres usar otro editor de texto como Emacs, puedes hacer lo siguiente:

```
$ git config --global core.editor emacs
```

Comprobando tu Configuración

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

```
$ git config --list  
user.name=John Doe  
user.email=johndoe@example.com  
color.status=auto  
color.branch=auto  
color.interactive=auto  
color.diff=auto  
...  
Puede que veas claves repetidas, porque Git lee la misma clave de distintos archivos (/etc/gitconfig y ~/.gitconfig, por ejemplo). En estos casos, Git usa el último valor para cada clave única que ve.
```

También puedes comprobar el valor que Git utilizará para una clave específica ejecutando `git config`:

```
$ git config user.name  
John Doe
```

2.2.4.- ¿Cómo obtener ayuda?

¿Cómo obtener ayuda?

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

```
$ git help  
$ git --help  
$ man git-
```

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

```
$ git help config
```

Estos comandos son muy útiles porque puedes acceder a ellos desde cualquier sitio, incluso sin conexión. Si las páginas del manual y este libro no son suficientes y necesitas que te ayude una persona, puedes probar en los canales #git o #github del servidor de IRC Freenode (irc.freenode.net). Estos canales están llenos de cientos de personas que conocen muy bien Git y suelen estar dispuestos a ayudar.

Obra publicada con [Licencia Creative Commons Reconocimiento Compartir igual 4.0](#)

2.3.- Comandos básicos

Si pudieras leer solo un capítulo para empezar a trabajar con Git, este es el capítulo que debes leer. Este capítulo cubre todos los comandos básicos que necesitas para hacer la gran mayoría de cosas a las que eventualmente vas a dedicar tu tiempo mientras trabajas con Git. Al final del capítulo, deberás 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 enmendar 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.

Obra publicada con [Licencia Creative Commons Reconocimiento Compartir igual 4.0](#)

2.3.1.- Obteniendo un repositorio Git

Puedes obtener un proyecto Git de dos maneras. La primera es tomar un proyecto o directorio existente e importarlo en Git. La segunda es clonar un repositorio existente en Git desde otro servidor.

Inicializando un repositorio en un directorio existente

Si estás empezando a seguir un proyecto existente en Git, debes ir al directorio del proyecto y usar el siguiente comando:

```
$ git init
```

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

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 quieras controlar, seguidos de un `git commit` para confirmar los cambios:

```
$ git add *.c  
$ git add LICENSE  
$ git commit -m 'initial project version'
```

Veremos lo que hacen estos comandos más adelante. En este momento, tienes un repositorio de 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 familiarizado con otros sistemas de control de versiones como Subversion, verás que el comando es "checkout" en vez de "clone". Es una distinción importante, ya que Git recibe una copia de casi todos los datos que tiene el servidor. Cada versión de cada archivo de la historia del proyecto es descargada por defecto cuando ejecutas `git clone`. De hecho, si el disco de tu servidor se corrompe, puedes usar cualquiera de los clones en cualquiera de los clientes para devolver el servidor al estado en el que estaba cuando fue clonado (puede que pierdas algunos hooks del lado del servidor y demás, pero toda la información acerca de las versiones estará ahí) — véase Configurando Git en un servidor para más detalles.

Puedes clonar un repositorio con `git clone [url]`. Por ejemplo, si quieres clonar la librería de Git llamada `libgit2` puedes hacer algo así:

```
$ git clone https://github.com/libgit2/libgit2
```

Esto crea un directorio llamado `libgit2`, inicializa un directorio `.git` en su interior, descarga toda la información de ese repositorio y saca una copia de trabajo de la última versión. Si te metes en el directorio `libgit2`, 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 `libgit2`, puedes especificarlo con la siguiente opción de línea de comandos:

```
$ git clone https://github.com/libgit2/libgit2 mylibgit
```

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

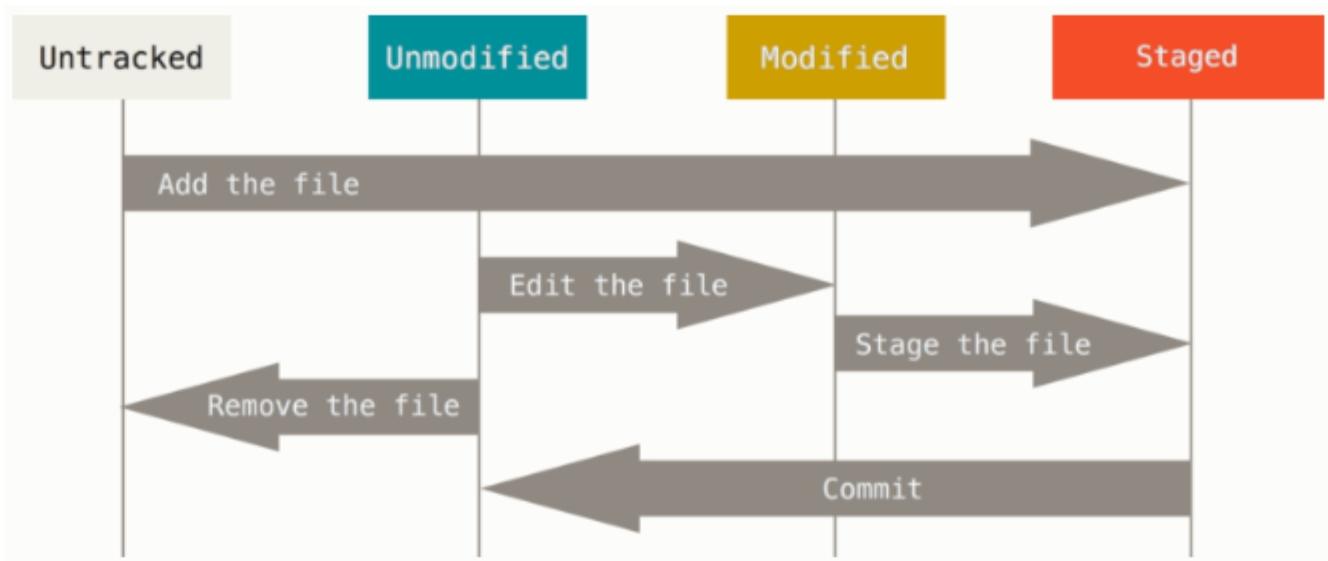
Git te permite usar distintos protocolos de transferencia. El ejemplo anterior usa el protocolo `https://`, pero también puedes utilizar `git://` o `usuario@servidor:ruta/del/repositorio.git` que utiliza el protocolo de transferencia SSH. En Configurando Git en un servidor se explicarán todas las opciones disponibles a la hora de configurar el acceso a tu repositorio de Git, y las ventajas e inconvenientes de cada una.

Obra publicada con [Licencia Creative Commons Reconocimiento Compartir igual 4.0](#)

2.3.2.- Revisando estados una vez clonado el repositorio

Si ya tienes un repositorio Git y un *checkout/clone* o copia de trabajo de los archivos de dicho proyecto. El siguiente paso es realizar algunos cambios y confirmar instantáneas de esos cambios en el repositorio **cada vez que el proyecto alcance un estado que quieras conservar**.

Recuerda que cada archivo de tu repositorio puede tener dos estados: rastreados y sin rastrear. Los archivos rastreados (*tracked files* en inglés) son todos aquellos archivos que estaban en la última instantánea del proyecto, pueden ser archivos sin modificar, modificados o preparados. Los archivos sin rastrear son todos los demás - cualquier otro archivo en tu directorio de trabajo que no estaba en tu última instantánea y que no está en el área de preparación (*staging area*). **Cuando clonas por primera vez un repositorio, todos tus archivos estarán rastreados y sin modificar pues acabas de sacarlos y aun no han sido editados.**



Ciclo de vida de los ficheros

Mientras editas archivos, Git los ve como modificados, pues han sido cambiados desde su último *commit*. Luego preparas estos archivos modificados y finalmente confirmas todos los cambios preparados, y repites el ciclo.

Revisando el Estado de tus Archivos

La herramienta principal para determinar qué archivos están en qué estado es el comando `git status`. Si ejecutas este comando inmediatamente después de clonar un repositorio, deberías ver algo como esto:

```
$ git status  
On branch master  
nothing to commit, working directory clean
```

Esto significa que tienes un directorio de trabajo limpio - en otras palabras, que **no hay archivos rastreados y modificados (las dos cosas a la vez)**. Además, Git no encuentra archivos sin rastrear, de lo contrario aparecerían listados aquí. Finalmente, el comando te indica en cuál rama estás y te informa que no ha variado con respecto a la misma rama en el servidor. Por ahora, la rama siempre será “master”, que es la rama por defecto, no le prestaremos atención de momento, más adelante se tratará en detalle las ramas y las referencias.

Supongamos que añades un nuevo archivo a tu proyecto, un simple README. Si el archivo no existía antes y ejecutas `git status`, verás el archivo sin rastrear de la siguiente manera:

```
$ echo 'My Project' > README  
$ git status  
On branch master  
Untracked files:  
(use "git add ..." to include in what will be committed)  
  
 README
```

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

Puedes ver que el archivo README está sin rastrear porque aparece debajo del encabezado “Untracked files” (“Archivos no rastreados” en inglés) en la salida. Sin rastrear significa que Git ve archivos que no tenías en el *commit* anterior. Git no los incluirá en tu próximo *commit* a menos que se lo indiques explícitamente. Se comporta así para evitar incluir accidentalmente archivos binarios o cualquier otro archivo que no quieras incluir. Como tú sí quieres incluir README, debes comenzar a rastrearlo.

2.3.3.- Rastrear nuevos archivos

Para comenzar a rastrear un archivo debes usar el comando `git add`. Para comenzar a rastrear el archivo README, puedes ejecutar lo siguiente:

```
$ git add README
```

Ahora si vuelves a ver el estado del proyecto, verás que el archivo README está siendo rastreado y está preparado para ser confirmado:

```
$ git status  
On branch master  
Changes to be committed:  
(use "git reset HEAD ..." to unstage)
```

```
new file: README
```

Puedes ver que está siendo rastreado porque aparece luego del encabezado “Cambios a ser confirmados” (“Changes to be committed” en inglés). Si confirmas en este punto, se guardará en el historial la versión del archivo correspondiente al instante en que ejecutaste `git add`. Anteriormente cuando ejecutaste `git init`, ejecutaste luego `git add (files)` - lo cual inició el rastreo de archivos en tu directorio. El comando `git add` puede recibir tanto una ruta de archivo como de un directorio, si es de un directorio, el comando añade recursivamente los archivos que están dentro de él.

Si lo que se quiere es rastrear todo el proyecto, se deberá ejecutar:

```
$ git add .
```

Obra publicada con [Licencia Creative Commons Reconocimiento Compartir igual 4.0](#)

2.3.4.- Preparar Archivos Modificados

Vamos a cambiar un archivo que esté rastreado. Si cambias el archivo rastreado llamado "CONTRIBUTING.md" y luego ejecutas el comando `git status`, verás algo parecido a esto:

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD ..." to unstage)

    new file: README

Changes not staged for commit:
  (use "git add ..." to update what will be committed)
  (use "git restore -- ..." to discard changes in working directory)

    modified: CONTRIBUTING.md
```

El archivo "CONTRIBUTING.md" aparece en una sección llamada "Changes not staged for commit" ("Cambios no preparado para confirmar" en inglés) - lo que significa que existe un archivo rastreado que ha sido modificado en el directorio de trabajo pero que aún no está preparado. Para prepararlo, ejecutas el comando `git add`. `git add` es un comando que cumple varios propósitos - lo usas para empezar a rastrear archivos nuevos, preparar archivos, y hacer otras cosas como marcar archivos en conflicto por combinación como resueltos. Es más útil que lo veas como un comando para "añadir este contenido a la próxima confirmación" más que para "añadir este archivo al proyecto". Ejecutemos `git add` para preparar el archivo "CONTRIBUTING.md" y luego ejecutemos `git status`:

```
$ git add CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD ..." to unstage)

    new file: README
    modified: CONTRIBUTING.md
```

Ambos archivos están preparados y formarán parte de tu próxima confirmación. En este momento, supongamos que recuerdas que debes hacer un pequeño cambio en `CONTRIBUTING.md` antes de confirmarlo. Abres de nuevo el archivo, lo cambias y ahora estás listos para confirmar. Sin embargo, ejecutemos `git status` una vez más:

```
$ vim CONTRIBUTING.md
```

```
$ git status
```

On branch master

Changes to be committed:

(use "git reset HEAD ..." to unstage)

new file: README

modified: CONTRIBUTING.md

Changes not staged for commit:

(use "git add ..." to update what will be committed)

(use "git checkout -- ..." to discard changes in working directory)

modified: CONTRIBUTING.md

¡¿Pero qué...?! Ahora `CONTRIBUTING.md` aparece como preparado y como no preparado. ¿Cómo es posible? Resulta que Git prepara un archivo de acuerdo al estado que tenía cuando ejecutas el comando `git add`. Si confirmas ahora, se confirmará la versión de `CONTRIBUTING.md` que tenías la última vez que ejecutaste `git add` y no la versión que ves ahora en tu directorio de trabajo al ejecutar `git status`. Si modificas un archivo luego de ejecutar `git add`, deberás ejecutar `git add` de nuevo para preparar la última versión del archivo:

```
$ git add CONTRIBUTING.md
```

```
$ git status
```

On branch master

Changes to be committed:

(use "git reset HEAD ..." to unstage)

new file: README

modified: CONTRIBUTING.md

Obra publicada con [Licencia Creative Commons Reconocimiento Compartir igual 4.0](#)

2.2.5.- Estados abreviados

Si bien es cierto que la salida de `git status` es bastante explícita, también es verdad que es muy extensa. Git ofrece una opción para obtener un estado abreviado, de manera que puedas ver tus cambios de una forma más compacta. Si ejecutas `git status -s` o `git status --short`, obtendrás una salida mucho más simplificada.

```
$ git status -s
M README
MM Rakefile
A lib/git.rb
M lib/simplegit.rb
?? LICENSE.txt
```

Los archivos nuevos que no están rastreados tienen un `??` a su lado, los archivos que están preparados tienen una `A` y los modificados una `M`. El estado aparece en dos columnas - la columna de la izquierda indica el estado preparado y la columna de la derecha indica el estado sin preparar. Por ejemplo, en esa salida, el archivo `README` está modificado en el directorio de trabajo pero no está preparado, mientras que `lib/simplegit.rb` está modificado y preparado. El archivo `Rakefile` fue modificado, preparado y modificado otra vez por lo que existen cambios preparados y sin preparar.

Si el comando `git status` es muy impreciso para ti - quieres ver exactamente que ha cambiado, no solo cuáles archivos lo han hecho - puedes usar el comando `git diff`. Hablaremos sobre `git diff` más adelante, pero lo usarás probablemente para responder estas dos preguntas: ¿Qué has cambiado pero aun no has preparado? y ¿Qué has preparado y está listo para confirmar? A pesar de que `git status` responde a estas preguntas de forma muy general listando el nombre de los archivos, `git diff` te muestra las líneas exactas que fueron añadidas y eliminadas, es decir, el parche.

Supongamos que editas y preparas el archivo `README` de nuevo y luego editas `CONTRIBUTING.md` pero no lo preparas. Si ejecutas el comando `git status`, verás algo como esto:

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD ..." to unstage)

    new file: README
```

```
Changes not staged for commit:
  (use "git add ..." to update what will be committed)
  (use "git restore -- ..." to discard changes in working directory)
```

```
modified: CONTRIBUTING.md
```

Para ver qué has cambiado pero aun no has preparado, escribe `git diff` sin más parámetros:

```
$ git diff
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
index 8ebb991..643e24f 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
@@ -65,7 +65,8 @@ branch directly, things can get messy.
Please include a nice description of your changes when you submit your PR;
if we have to read the whole diff to figure out why you're contributing
in the first place, you're less likely to get feedback and have your change
-merged in.
+merged in. Also, split your changes into comprehensive chunks if you patch is
+longer than a dozen lines.
```

If you are starting to work on a particular area, feel free to submit a PR
that highlights your work in progress (and note in the PR title that it's

Este comando compara lo que tienes en tu directorio de trabajo con lo que está en el área de preparación. El resultado te indica los cambios que has hecho pero que aun no has preparado.

Si quieres ver lo que has preparado y será incluido en la próxima confirmación, puedes usar `git diff --staged`. Este comando compara tus cambios preparados con la última instantánea confirmada.

```
$ git diff --staged
diff --git a/README b/README
new file mode 100644
index 0000000..03902a1
--- /dev/null
+++ b/README
@@ -0,0 +1 @@
+My Project
```

Es importante resaltar que al llamar a `git diff` sin parámetros no verás los cambios desde tu última confirmación - solo verás los cambios que aun no están preparados. Esto puede ser confuso porque si preparas todos tus cambios, `git diff` no te devolverá ninguna salida.

Pasemos a otro ejemplo, si preparas el archivo `CONTRIBUTING.md` y luego lo editas, puedes usar `git diff` para ver los cambios en el archivo que ya están preparados y los cambios que no lo están. Si nuestro ambiente es como este:

```
$ git add CONTRIBUTING.md
$ echo 'test line' >> CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD ..." to unstage)

    modified: CONTRIBUTING.md

Changes not staged for commit:
  (use "git add ..." to update what will be committed)
  (use "git restore -- ..." to discard changes in working directory)

    modified: CONTRIBUTING.md
```

Puedes usar `git diff` para ver qué está sin preparar

```
$ git diff  
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md  
index 643e24f..87f08c8 100644  
--- a/CONTRIBUTING.md  
+++ b/CONTRIBUTING.md  
@@ -119,3 +119,4 @@ at the  
## Starter Projects
```

See our [projects list](<https://github.com/libgit2/libgit2/blob/development/PROJECTS.md>).

```
+# test line
```

Y `git diff --cached` para ver que has preparado hasta ahora (--staged y --cached son sinónimos):

```
$ git diff --cached  
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md  
index 8ebb991..643e24f 100644  
--- a/CONTRIBUTING.md  
+++ b/CONTRIBUTING.md  
@@ -65,7 +65,8 @@ branch directly, things can get messy.  
Please include a nice description of your changes when you submit your PR;  
if we have to read the whole diff to figure out why you're contributing  
in the first place, you're less likely to get feedback and have your change  
-merged in.  
+merged in. Also, split your changes into comprehensive chunks if you patch is  
+longer than a dozen lines.
```

If you are starting to work on a particular area, feel free to submit a PR
that highlights your work in progress (and note in the PR title that it's

Obra publicada con [Licencia Creative Commons Reconocimiento Compartir igual 4.0](#)

2.2.6.- Confirmar los cambios

Ahora que tu área de preparación está como quieras, puedes confirmar tus cambios. Recuerda que cualquier cosa que no esté preparada - cualquier archivo que hayas creado o modificado y que no hayas agregado con `git add` desde su edición - no será confirmado. Se mantendrán como archivos modificados en tu disco. En este caso, digamos que la última vez que ejecutaste `git status` verificaste que todo estaba preparado y que estás listo para confirmar tus cambios. La forma más sencilla de confirmar es escribiendo `git commit`:

```
$ git commit
```

Al hacerlo, arrancará el editor de tu preferencia. (El editor se establece a través de la variable de ambiente `SEDEDITOR` de tu terminal - usualmente es vim o emacs, aunque puedes configurarlo con el editor que quieras usando el comando `git config --global core.editor`).

El editor mostrará el siguiente texto (este ejemplo corresponde a una pantalla de Vim):

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Changes to be committed:
#   new file: README
#   modified: CONTRIBUTING.md
#
~
~
~
~
".git/COMMIT_EDITMSG" 9L, 283C
```

Puedes ver que el mensaje de confirmación por defecto contiene la última salida del comando `git status` comentada y una línea vacía encima de ella. Puedes eliminar estos comentarios y escribir tu mensaje de confirmación, o puedes dejarlos allí para ayudarte a recordar qué estás confirmando. (Para obtener una forma más explícita de recordar qué has modificado, puedes pasar la opción `-v` a `git commit`. Al hacerlo se incluirá en el editor el diff de tus cambios para que veas exactamente qué cambios estás confirmando). Cuando sales del editor, Git crea tu confirmación con tu mensaje (eliminando el texto comentado y el diff).

Otra alternativa es escribir el mensaje de confirmación directamente en el comando `commit` utilizando la opción `-m`:

```
$ git commit -m "Story 182: Fix benchmarks for speed"
[master 463dc4f] Story 182: Fix benchmarks for speed
2 files changed, 2 insertions(+)
create mode 100644 README
```

¡Has creado tu primera confirmación (o *commit*)! Puedes ver que la confirmación te devuelve una salida descriptiva: indica cuál rama has confirmado (`master`), que `checksum` SHA-1 tiene el `commit` (`463dc4f`), cuántos archivos han cambiado y estadísticas sobre las líneas añadidas y eliminadas en el `commit`.

Recuerda que la confirmación guarda una instantánea de tu área de preparación. Todo lo que no hayas preparado sigue allí modificado; puedes hacer una nueva confirmación para añadirlo a tu historial. Cada vez que realizas un `commit`, guardas una instantánea de tu proyecto la cual puedes usar para comparar o volver a ella luego.

2.2.7.- Ver historial de confirmaciones

Después de haber hecho varias confirmaciones, o si has clonado un repositorio que ya tenía un histórico de confirmaciones, probablemente quieras mirar atrás para ver qué modificaciones se han llevado a cabo. La herramienta más básica y potente para hacer esto es el comando `git log`.

Estos ejemplos usan un proyecto muy sencillo llamado “simplegit”. Para clonar el proyecto, ejecuta:

```
git clone https://github.com/schacon/simplegit-progit
```

Cuando ejecutes `git log` sobre este proyecto, deberías ver una salida similar a esta:

```
$ git log  
commit ca82a6dff817ec66f44342007202690a93763949  
Author: Scott Chacon  
Date: Mon Mar 17 21:52:11 2008 -0700
```

changed the version number

```
commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7  
Author: Scott Chacon  
Date: Sat Mar 15 16:40:33 2008 -0700
```

removed unnecessary test

```
commit a11bef06a3f659402fe7563abf99ad00de2209e6  
Author: Scott Chacon  
Date: Sat Mar 15 10:31:28 2008 -0700
```

first commit

Por defecto, si no pasas ningún parámetro, `git log` lista las confirmaciones hechas sobre ese repositorio en orden cronológico inverso. Es decir, las confirmaciones más recientes se muestran al principio. Como puedes ver, este comando lista cada confirmación con su suma de comprobación SHA-1, el nombre y dirección de correo del autor, la fecha y el mensaje de confirmación.

El comando `git log` proporciona gran cantidad de opciones para mostrarte exactamente lo que buscas. Aquí veremos algunas de las más usadas.

Una de las opciones más útiles es `-p`, que muestra las diferencias introducidas en cada confirmación. También puedes usar la opción `-2`, que hace que se muestren únicamente las dos últimas entradas del historial:

```
$ git log -p -2
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon
Date: Mon Mar 17 21:52:11 2008 -0700
```

changed the version number

```
diff --git a/Rakefile b/Rakefile
index a874b73..8f94139 100644
--- a/Rakefile
+++ b/Rakefile
@@ -5,7 +5,7 @@ require 'rake/gempackagetask'
spec = Gem::Specification.new do |s|
  s.platform = Gem::Platform::RUBY
  s.name      = "simplegit"
- s.version   = "0.1.0"
+ s.version   = "0.1.1"
  s.author    = "Scott Chacon"
  s.email     = "schacon@gee-mail.com"
  s.summary   = "A simple gem for using Git in Ruby code."
```

```
commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon
Date: Sat Mar 15 16:40:33 2008 -0700
```

removed unnecessary test

```
diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index a0a60ae..47c6340 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -18,8 +18,3 @@ class SimpleGit
end

end
-
-if $0 == __FILE__
- git = SimpleGit.new
- puts git.show
-end
\ No newline at end of file
```

Esta opción muestra la misma información, pero añadiendo tras cada entrada las diferencias que le corresponden. Esto resulta muy útil para revisiones de código, o para visualizar rápidamente lo que ha pasado en las confirmaciones enviadas por un colaborador. También puedes usar con `git log` una serie de opciones de resumen. Por ejemplo, si quieras ver algunas estadísticas de cada confirmación, puedes usar la opción `--stat`:

```
$ git log --stat
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon
Date: Mon Mar 17 21:52:11 2008 -0700
```

changed the version number

```
Rakefile | 2 +-
1 file changed, 1 insertion(+), 1 deletion(-)
```

```
commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon
Date: Sat Mar 15 16:40:33 2008 -0700
```

removed unnecessary test

```
lib/simplegit.rb | 5 -----
1 file changed, 5 deletions(-)
```

```
commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon
Date: Sat Mar 15 10:31:28 2008 -0700
```

first commit

```
README      | 6 ++++++
Rakefile    | 23 ++++++++++++++++++++++
lib/simplegit.rb | 25 ++++++++++++++++++++++
3 files changed, 54 insertions(+)
```

Como puedes ver, la opción `--stat` imprime tras cada confirmación una lista de archivos modificados, indicando cuántos han sido modificados y cuántas líneas han sido añadidas y eliminadas para cada uno de ellos, y un resumen de toda esta información.

Otra opción realmente útil es `--pretty`, que modifica el formato de la salida. Tienes unos cuantos estilos disponibles. La opción `oneline` imprime cada confirmación en una única línea, lo que puede resultar útil si estás analizando gran cantidad de confirmaciones. Otras opciones son `short`, `full` y `fuller`, que muestran la salida en un formato parecido, pero añadiendo menos o más información, respectivamente:

```
$ git log --pretty=oneline
ca82a6dff817ec66f44342007202690a93763949 changed the version number
085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7 removed unnecessary test
a11bef06a3f659402fe7563abf99ad00de2209e6 first commit
```

La opción más interesante es `format`, que te permite especificar tu propio formato. Esto resulta especialmente útil si estás generando una salida para que sea analizada por otro programa — como específicas el formato explícitamente, sabes que no cambiará en futuras actualizaciones de Git—:

```
$ git log --pretty=format:"%h - %an, %ar : %s"
ca82a6d - Scott Chacon, 6 years ago : changed the version number
085bb3b - Scott Chacon, 6 years ago : removed unnecessary test
a11bef0 - Scott Chacon, 6 years ago : first commit
```

Opciones útiles de `git log --pretty=format` lista algunas de las opciones más útiles aceptadas por `format`.

Tabla 1. Opciones útiles de `git log --pretty=format`

Opción	Descripción de la salida
<code>%H</code>	Hash de la confirmación
<code>%h</code>	Hash de la confirmación abreviado
<code>%T</code>	Hash del árbol
<code>%t</code>	Hash del árbol abreviado
<code>%P</code>	Hashes de las confirmaciones padre
<code>%p</code>	Hashes de las confirmaciones padre abreviados
<code>%an</code>	Nombre del autor
<code>%ae</code>	Dirección de correo del autor
<code>%ad</code>	Fecha de autoría (el formato respeta la opción <code>--date</code>)
<code>%ar</code>	Fecha de autoría, relativa
<code>%cn</code>	Nombre del confirmador
<code>%ce</code>	Dirección de correo del confirmador
<code>%cd</code>	Fecha de confirmación
<code>%cr</code>	Fecha de confirmación, relativa
<code>%s</code>	Asunto

Puede que te estés preguntando la diferencia entre *autor* (*author*) y *confirmador* (*committer*). El autor es la persona que escribió originalmente el trabajo, mientras que el confirmador es quien lo aplicó. Por tanto, si mandas un parche a un proyecto, y uno de sus miembros lo aplica, ambos recibiréis reconocimiento —tú como autor, y el miembro del proyecto como confirmador—. Veremos esta distinción con mayor profundidad en Git en entornos distribuidos.

Las opciones `oneline` y `format` son especialmente útiles combinadas con otra opción llamada `--graph`. Esta añade un pequeño gráfico ASCII mostrando tu historial de ramificaciones y uniones:

```
$ git log --pretty=format:"%h %s" --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 xmllschema
* 11d191e Merge branch 'defunkt' into local
```

Este tipo de salidas serán más interesantes cuando empecemos a hablar sobre ramificaciones y combinaciones en el próximo capítulo.

Éstas son sólo algunas de las opciones para formatear la salida de `git log` —existen muchas más. Opciones típicas de `git log` lista las opciones vistas hasta ahora, y algunas otras opciones de formateo que pueden resultarte útiles, así como su efecto sobre la salida.

Tabla 2. Opciones típicas de `git log`

Opción	Descripción
<code>-p</code>	Muestra el parche introducido en cada confirmación.
<code>--stat</code>	Muestra estadísticas sobre los archivos modificados en cada confirmación.
<code>--shortstat</code>	Muestra solamente la línea de resumen de la opción <code>--stat</code> .
<code>--name-only</code>	Muestra la lista de archivos afectados.
<code>--name-status</code>	Muestra la lista de archivos afectados, indicando además si fueron añadidos, modificados o eliminados.
<code>--abbrev-commit</code>	Muestra solamente los primeros caracteres de la suma SHA-1, en vez de los 40 caracteres de que se compone.
<code>--relative-date</code>	Muestra la fecha en formato relativo (por ejemplo, “2 weeks ago” (“hace 2 semanas”)) en lugar del formato completo.
<code>--graph</code>	Muestra un gráfico ASCII con la historia de ramificaciones y uniones.
<code>--pretty</code>	Muestra las confirmaciones usando un formato alternativo. Posibles opciones son <code>oneline</code> , <code>short</code> , <code>full</code> , <code>fuller</code> y <code>format</code> (mediante el cual puedes especificar tu propio formato).

Limitar la Salida del Historial

Además de las opciones de formateo, `git log` acepta una serie de opciones para limitar su salida —es decir, opciones que te permiten mostrar únicamente parte de las confirmaciones—. Ya has visto una de ellas, la opción `-2`, que muestra sólo las dos últimas confirmaciones. De hecho, puedes hacer `-n`, siendo `n` cualquier entero, para mostrar las últimas `n` confirmaciones. En realidad es poco probable que uses esto con frecuencia, ya que Git por defecto pagina su salida para que veas cada página del historial por separado.

Sin embargo, las opciones temporales como `--since` (desde) y `--until` (hasta) sí que resultan muy útiles. Por ejemplo, este comando lista todas las confirmaciones hechas durante las dos últimas semanas:

```
$ git log --since=2.weeks
```

Este comando acepta muchos formatos. Puedes indicar una fecha concreta ("2008-01-15"), o relativa, como "2 years 1 day 3 minutes ago" ("hace 2 años, 1 día y 3 minutos").

También puedes filtrar la lista para que muestre sólo aquellas confirmaciones que cumplen ciertos criterios. La opción `--author` te permite filtrar por autor, y `--grep` te permite buscar palabras clave entre los mensajes de confirmación. (Ten en cuenta que si quieras aplicar ambas opciones simultáneamente, tienes que añadir `--all-match`, o el comando mostrará las confirmaciones que cumplan cualquiera de las dos, no necesariamente las dos a la vez.)

Otra opción útil es `-s`, la cual recibe una cadena y solo muestra las confirmaciones que cambiaron el código añadiendo o eliminando la cadena. Por ejemplo, si quieras encontrar la última confirmación que añadió o eliminó una referencia a una función específica, puede ejecutar:

```
$ git log -Sfunction_name
```

La última opción verdaderamente útil para filtrar la salida de `git log` es especificar una ruta. Si especificas la ruta de un directorio o archivo, puedes limitar la salida a aquellas confirmaciones que introdujeron un cambio en dichos archivos. Ésta debe ser siempre la última opción, y suele ir precedida de dos guiones (--) para separar la ruta del resto de opciones.

En Opciones para limitar la salida de `git log` se listan estas opciones, y algunas otras bastante comunes a modo de referencia.

Tabla 3. Opciones para limitar la salida de `git log`

Opción	Descripción
<code>-n</code>	Muestra solamente las últimas n confirmaciones
<code>--since</code> , <code>--after</code>	Muestra aquellas confirmaciones hechas después de la fecha especificada.
<code>--until</code> , <code>--before</code>	Muestra aquellas confirmaciones hechas antes de la fecha especificada.
<code>--author</code>	Muestra sólo aquellas confirmaciones cuyo autor coincide con la cadena especificada.
<code>--committer</code>	Muestra sólo aquellas confirmaciones cuyo confirmador coincide con la cadena especificada.
<code>-s</code>	Muestra sólo aquellas confirmaciones que añaden o eliminan código que corresponda con la cadena especificada.

Por ejemplo, si quieras ver cuáles de las confirmaciones hechas sobre archivos de prueba del código fuente de Git fueron enviadas por Junio Hamano, y no fueron uniones, en el mes de octubre de 2008, ejecutarías algo así:

```
$ git log --pretty="%h - %s" --author=gitster --since="2008-10-01" \
--before="2008-11-01" --no-merges -- t/
```

5610e3b - Fix testcase failure when extended attributes are in use

acd3b9e - Enhance hold_lock_file_for_{update,append}() API

f563754 - demonstrate breakage of detached checkout with symbolic link HEAD

d1a43f2 - reset --hard/read-tree --reset -u: remove unmerged new paths

51a94af - Fix "checkout --track -b newbranch" on detached HEAD

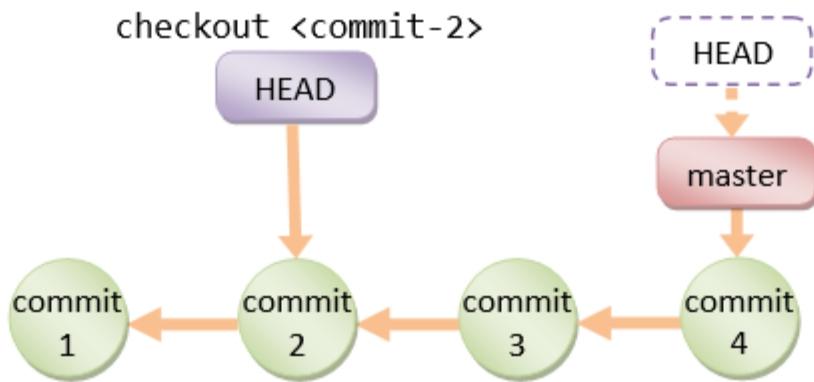
b0ad11e - pull: allow "git pull origin \$something:\$current_branch" into an unborn branch

De las casi 40.000 confirmaciones en la historia del código fuente de Git, este comando muestra las 6 que cumplen estas condiciones.

Obra publicada con [Licencia Creative Commons Reconocimiento Compartir igual 4.0](#)

2.2.8.- Restaurar confirmaciones

El comando **git checkout** te permite volver a una versión anterior de un *commit* o momento en el tiempo del historial de las versiones de tu programa. Para ello, deberás extraer una parte del número HASH e ingresarla al comando de git *checkout* de la siguiente forma: **git checkout [número o parte del número HASH]**. Para volver a la versión actual se puede utilizar el formato anterior o simplemente **git checkout master**.



El **git reset --hard** es un comando de Git que se usa para restablecer la rama actual a una confirmación anterior, descartando cualquier cambio local y modificación realizada en los archivos.

La opción **--hard** significa "restablecimiento completo" y se usa para descartar por la fuerza cualquier cambio local y modificación realizada en los archivos. Cuando ejecutas **git reset --hard**, Git restablece el puntero de rama a la confirmación especificada y cualquier cambio realizado en los archivos desde entonces se pierde de forma permanente.

Aquí hay un ejemplo de cómo usar **git reset --hard**:

```
git reset --hard HEAD~2
```

Este comando restablecerá la rama actual a la confirmación que está dos pasos atrás del puntero **HEAD** actual. La opción **--hard** descartará cualquier cambio local realizado en los archivos.

Es importante usar **git reset --hard** con precaución, ya que puede eliminar de forma permanente cualquier cambio no confirmado realizado en los archivos. Siempre debe asegurarse de hacer una copia de seguridad de los cambios importantes antes de usar este comando.

Otro ejemplo:

```
$ git log --oneline  
.....  
ca82a6d changed the version number  
085bb3b removed unnecessary test  
a11bef0 first commit
```

Si ejecutamos

```
git reset --hard a11bef0
```

Restablecerá el puntero **HEAD** a la instantánea *first commit*, ¡ojo! se perderán todas las anteriores.

Obra publicada con [Licencia Creative Commons Reconocimiento Compartir igual 4.0](#)

2.2.9.- Actividad de ampliación

Creando un repositorio local

*Ojo no es lo mismo obtener un repositorio externo y añadir ficheros al área de trabajo de git para hacerles un rastreo que editar directamente los ficheros en el directorio real (este es el caso de la actividad de ampliación), en este caso cuando creamos e inicializamos el repositorio con git init, todos los archivos se consideran como archivos nuevos sin rastrear ??.

Obra publicada con [Licencia Creative Commons Reconocimiento Compartir igual 4.0](#)

2.4.- Otros comandos

En esta sección aprenderemos a utilizar otros comandos más avanzados. ¿Estás listo? Adelante....

Obra publicada con [Licencia Creative Commons Reconocimiento Compartir igual 4.0](#)

2.4.1.- Cambiar el nombre de los archivos

Al contrario que muchos sistemas VCS, Git no rastrea explícitamente los cambios de nombre en archivos. Si renombras un archivo en Git, no se guardará ningún metadato que indique que renombraste el archivo. Sin embargo, Git es bastante listo como para detectar estos cambios luego que los has hecho - más adelante, veremos cómo se detecta el cambio de nombre.

Por esto, resulta confuso que Git tenga un comando `mv`. Si quieras renombrar un archivo en Git, puedes ejecutar algo como

```
$ git mv file_from file_to
```

y funcionará bien. De hecho, si ejecutas algo como eso y ves el estado, verás que Git lo considera como un renombramiento de archivo:

```
$ git mv README.md README
```

```
$ git status
```

On branch master

Changes to be committed:

(use "git reset HEAD ..." to unstage)

renamed: README.md -> README

Sin embargo, eso es equivalente a ejecutar algo como esto:

```
$ mv README.md README
```

```
$ git rm README.md
```

```
$ git add README
```

Git se da cuenta que es un renombramiento implícito, así que no importa si renombras el archivo de esa manera o a través del comando `mv`. La única diferencia real es que `mv` es un solo comando en vez de tres - existe por conveniencia. De hecho, puedes usar la herramienta que quieras para renombrar un archivo y luego realizar el proceso `rm/add` antes de confirmar.

Obra publicada con [Licencia Creative Commons Reconocimiento Compartir igual 4.0](#)

2.4.2.- Deshacer

En cualquier momento puede que quieras deshacer algo. Aquí repasaremos algunas herramientas básicas usadas para deshacer cambios que hayas hecho. Ten cuidado, a veces no es posible recuperar algo luego que lo has deshecho. Esta es una de las pocas áreas en las que Git puede perder parte de tu trabajo si cometes un error.

Uno de las acciones más comunes a deshacer es cuando confirmas un cambio antes de tiempo y olvidas agregar algún archivo, o te equivocas en el mensaje de confirmación. Si quieres rehacer la confirmación, puedes reconfirmar con la opción `--amend`:

```
$ git commit --amend
```

Este comando utiliza tu área de preparación para la confirmación. Si no has hecho cambios desde tu última confirmación (por ejemplo, ejecutas este comando justo después de tu confirmación anterior), entonces la instantánea lucirá exactamente igual y lo único que cambiarás será el mensaje de confirmación.

Se lanzará el mismo editor de confirmación, pero verás que ya incluye el mensaje de tu confirmación anterior. Puedes editar el mensaje como siempre y se sobreescibirá tu confirmación anterior.

Por ejemplo, si confirmas y luego te das cuenta que olvidaste preparar los cambios de un archivo que querías incluir en esta confirmación, puedes hacer lo siguiente:

```
$ git commit -m 'initial commit'  
$ git add forgotten_file  
$ git commit --amend
```

Al final terminarás con una sola confirmación - la segunda confirmación reemplaza el resultado de la primera.

Deshacer un Archivo Preparado

Las siguientes dos secciones demuestran cómo lidiar con los cambios de tu área de preparación y tú directorio de trabajo. Afortunadamente, el comando que usas para determinar el estado de esas dos áreas también te recuerda cómo deshacer los cambios en ellas. Por ejemplo, supongamos que has cambiado dos archivos y que quieres confirmarlos como dos cambios separados, pero accidentalmente has escrito `git add *` y has preparado ambos. ¿Cómo puedes sacar del área de preparación uno de ellos? El comando `git status` te recuerda cómo:

```
$ git add .
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD ..." to unstage)

    renamed: README.md -> README
    modified: CONTRIBUTING.md
```

Justo debajo del texto “Changes to be committed” (“Cambios a ser confirmados”, en inglés), verás que dice que uses `git reset HEAD ...` para deshacer la preparación. Por lo tanto, usemos el consejo para deshacer la preparación del archivo `CONTRIBUTING.md`:

```
$ git reset HEAD CONTRIBUTING.md
Unstaged changes after reset:
M  CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD ..." to unstage)
```

```
renamed: README.md -> README

Changes not staged for commit:
  (use "git add ..." to update what will be committed)
  (use "git checkout -- ..." to discard changes in working directory)
```

```
modified: CONTRIBUTING.md
```

El comando es un poco raro, pero funciona. El archivo `CONTRIBUTING.md` está modificado y, nuevamente, no preparado.

Nota `git reset` puede ser un comando peligroso, especialmente si lo llamas con la opción `--hard`. Sin embargo, en el escenario descrito anteriormente, el archivo que está en tu directorio de trabajo no se toca, por lo que es relativamente seguro.

Por ahora lo único que necesitas saber sobre el comando `git reset` es esta invocación mágica. Entraremos en mucho más detalle sobre qué hace `reset` y cómo dominarlo para que haga cosas realmente interesantes en Reiniciar Desmitificado.

Deshacer un Archivo Modificado

¿Qué tal si te das cuenta que no quieres mantener los cambios del archivo `CONTRIBUTING.md`? ¿Cómo puedes restaurarlo fácilmente - volver al estado en el que estaba en la última confirmación (o cuando estaba recién clonado, o como sea que haya llegado a tu directorio de trabajo)? Afortunadamente, `git status` también te dice cómo hacerlo. En la salida anterior, el área no preparada lucía así:

Changes not staged for commit:

(use "git add ..." to update what will be committed)

(use "git restore -- ..." to discard changes in working directory)

modified: CONTRIBUTING.md

Allí se te indica explícitamente como descartar los cambios que has hecho. Hagamos lo que nos dice:

```
$ git checkout -- CONTRIBUTING.md
```

```
$ git status
```

On branch master

Changes to be committed:

(use "git reset HEAD ..." to unstage)

renamed: README.md -> README

Ahora puedes ver que los cambios se han revertido.

Obra publicada con [Licencia Creative Commons Reconocimiento Compartir igual 4.0](#)

2.4.3.- Eliminar archivos

Para eliminar archivos de Git, debes eliminarlos de tus archivos rastreados (o mejor dicho, eliminarlos del área de preparación) y luego confirmar. Para ello existe el comando `git rm`, que además elimina el archivo de tu directorio de trabajo de manera que no aparezca la próxima vez como un archivo no rastreado.

Si simplemente eliminas el archivo de tu directorio de trabajo, aparecerá en la sección "Changes not staged for commit" (esto es, *sin preparar*) en la salida de `git status`:

```
$ rm PROJECTS.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add/rm ..." to update what will be committed)
    (use "git checkout -- ..." to discard changes in working directory)

          deleted:  PROJECTS.md
```

no changes added to commit (use "git add" and/or "git commit -a")

Ahora, si ejecutas `git rm`, entonces se prepara la eliminación del archivo:

```
$ git rm PROJECTS.md
rm 'PROJECTS.md'
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD ..." to unstage)
```

deleted: PROJECTS.md

Con la próxima confirmación, el archivo habrá desaparecido y no volverá a ser rastreado. Si modificaste el archivo y ya lo habías añadido al índice, tendrás que forzar su eliminación con la opción `-f`. Esta propiedad existe por seguridad, para prevenir que elimines accidentalmente datos que aun no han sido guardados como una instantánea y que por lo tanto no podrás recuperar luego con Git.

Otra cosa que puedes querer hacer es mantener el archivo en tu directorio de trabajo pero eliminarlo del área de preparación. En otras palabras, quisieras mantener el archivo en tu disco duro pero sin que Git lo siga rastreando. Esto puede ser particularmente útil si olvidaste añadir algo en tu archivo `.gitignore` y lo preparaste accidentalmente, algo como un gran archivo de trazas a un montón de archivos compilados `.a`. Para hacerlo, utiliza la opción `--cached`:

```
$ git rm --cached README
```

Al comando `git rm` puedes pasarle archivos, directorios y patrones glob. Lo que significa que puedes hacer cosas como

```
$ git rm log/*.*log
```

Fíjate en la barra invertida (`\`) antes del asterisco `*`. Esto es necesario porque Git hace su propia expansión de nombres de archivo, aparte de la expansión hecha por tu terminal. Este comando elimina todos los archivos que tengan la extensión `.log` dentro del directorio `log/`. O también puedes hacer algo como:

```
$ git rm \*~
```

Este comando elimina todos los archivos que acaben con `~`.

Obra publicada con [Licencia Creative Commons Reconocimiento Compartir igual 4.0](#)

2.5.- GitHub

En esta sección aprenderemos a utilizar la plataforma GitHub. ¿Estás listo? Adelante....

Nota importante: Toda la información que se ofrece en esta sección viene reflejada en las documentaciones oficiales que se encuentra en la página <https://docs.github.com/es>, están adaptadas a las nuevas versiones que utilizaremos para las prácticas. Sólo se han incluido los conceptos y comandos que entendemos son claves para empezar a utilizar de forma profesional este sistema de control de versiones. Si en un momento determinado no aparece el procedimiento o comandos para realizar alguna acción, habría que investigarlo en la documentación oficial.

Obra publicada con [Licencia Creative Commons Reconocimiento Compartir igual 4.0](#)

2.5.1.- ¿Qué es GitHub?



Github es un portal creado para alojar el código de las aplicaciones de cualquier desarrollador, y que fue comprada por Microsoft en junio del 2018. La plataforma está creada para que **los desarrolladores suban el código de sus aplicaciones y herramientas**, y que como usuario no sólo puedas descargar la aplicación, sino también entrar a su perfil para leer sobre ella o colaborar con su desarrollo.

Obra publicada con [Licencia Creative Commons Reconocimiento Compartir igual 4.0](#)

2.5.2.- Crear una cuenta

Picha el siguiente enlace para obtener información de la página oficial:

[Crea una cuenta en la plataforma](#)

Obra publicada con [Licencia Creative Commons Reconocimiento Compartir igual 4.0](#)

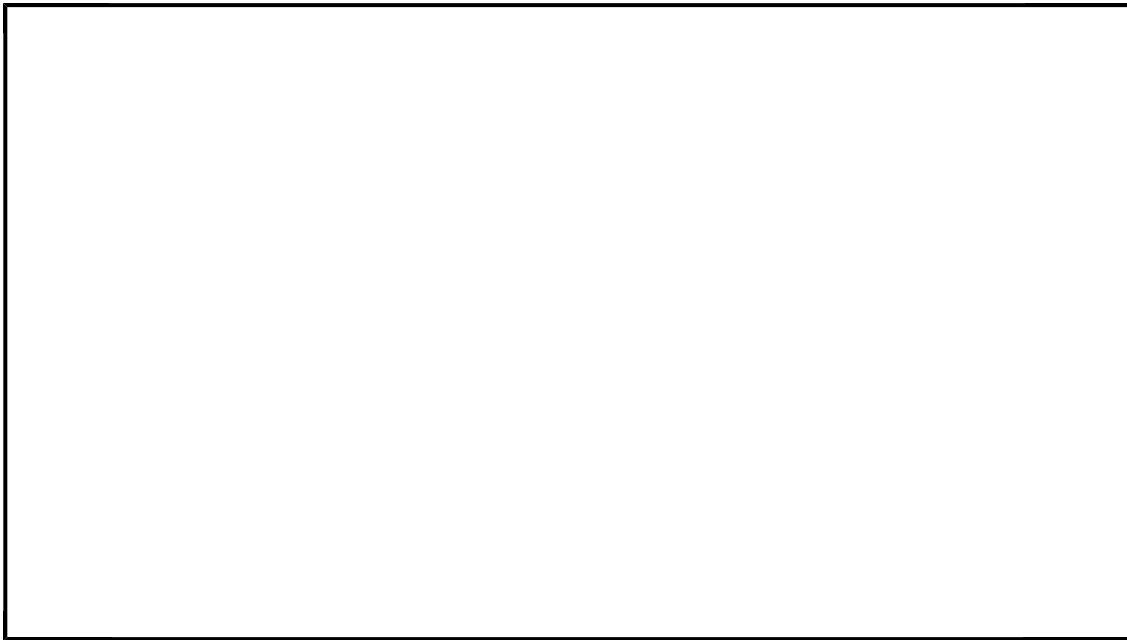
2.5.3.- Crear un repositorio

Picha el siguiente enlace para obtener información de la página oficial:

[Crear un repositorio en la plataforma](#)

Obra publicada con [Licencia Creative Commons Reconocimiento Compartir igual 4.0](#)

2.5.4.- Actividad de ampliación



Licencia: Dominio público

Licencia: Dominio público

Obra publicada con [Licencia Creative Commons Reconocimiento Compartir igual 4.0](#)

2.6.- Ramificaciones en Git

En esta sección aprenderemos el concepto de ramificación. ¿Estás listo? Adelante....

Obra publicada con [Licencia Creative Commons Reconocimiento Compartir igual 4.0](#)

2.6.1.- ¿Qué es una rama?

Cualquier sistema de control de versiones moderno tiene algún mecanismo para soportar el uso de ramas. Cuando hablamos de ramificaciones, significa que tú 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 requiere crear una nueva copia del código, lo cual puede tomar mucho tiempo cuando se trata de proyectos grandes.

Algunas personas resaltan que uno de los puntos más fuertes de Git es su sistema de ramificaciones y lo cierto es que esto le hace resaltar 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 tremadamente 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 desarollas.

¿Qué es una rama?

Para entender realmente cómo ramifica Git, previamente hemos de examinar la forma en que almacena sus datos.

Recordando lo citado en introducción, 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 una instantánea de tu trabajo preparado. Dicha instantánea contiene además 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 parent en los casos de confirmación normal, y múltiples padres en los casos de estar confirmando una fusión (merge) de dos o más 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 (un resumen SHA-1, tal y como se mencionaba en introducción, 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):

```
$ git add README test.rb LICENSE  
$ git commit -m 'initial commit of my project'
```

Cuando creas una confirmación con el comando `git commit`, Git realiza sumas de control de cada subdirectorios (en el ejemplo, solamente tenemos el directorio principal del proyecto), y las guarda como objetos árbol en el repositorio Git. Después, Git crea un objeto de confirmación con los metadatos pertinentes y un apuntador al objeto árbol raíz del proyecto.

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 del directorio (más sus respectivas relaciones con los "blobs"), y una confirmación de cambios (commit) apuntando a la raíz de ese árbol y conteniendo el resto de metadatos pertinentes.

Figura 9. Una confirmación y sus árboles

Si haces más cambios y vuelves a confirmar, la siguiente confirmación guardará un apuntador a su confirmación precedente.

Figura 10. Confirmaciones y sus predecesoras

Una rama Git es simplemente un apuntador móvil (dinámico) 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.

Nota: La rama “`master`” en Git, no es una rama especial. Es como cualquier otra rama. La única razón por la cual aparece en casi todos los repositorios es porque es la que crea por defecto el comando `git init` y la gente no se molesta en cambiarle el nombre.



Figura 11. Una rama y su historial de confirmaciones

Crear una Rama Nueva

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

```
$ git branch testing
```

Esto creará un nuevo apuntador apuntando a la misma confirmación donde estés actualmente.

Figura 12. Dos ramas apuntando al mismo grupo de confirmaciones

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 la rama `master`; pues el comando `git branch` solamente crea una nueva rama, pero no salta a dicha rama.

Figura 13. Apuntador HEAD a la rama donde estás actualmente

Esto puedes verlo fácilmente al ejecutar el comando `git log` para que te muestre a dónde apunta cada rama. Esta opción se llama `--decorate`.

```
$ git log --oneline --decorate
f30ab (HEAD, master, testing) add feature #32 - ability to add new
34ac2 fixed bug #1328 - stack overflow under certain conditions
98ca9 initial commit of my project
```

Puedes ver que las ramas “`master`” y “`testing`” están junto a la confirmación `f30ab`.

Cambiar de 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`.

Figura 14. El apuntador HEAD apunta a la rama actual

¿Cuál es el significado de todo esto? Buen, lo veremos tras realizar otra confirmación de cambios:

```
$ vim test.rb  
$ git commit -a -m 'made a change'
```


Figura 15. La rama apuntada por HEAD avanza con cada confirmación de cambios

Observamos algo interesante: la rama `testing` avanza, mientras que la rama `master` permanece en la confirmación donde estaba cuando lanzaste el comando `git checkout` para saltar. Volvamos ahora a la rama `master`:

```
$ git checkout master
```

Figura 16. HEAD apunta a otra rama cuando hacemos un salto

Este comando realiza dos acciones: Mueve el apuntador HEAD de nuevo a la rama `master`, y revierte los archivos de tu directorio de trabajo; dejándolos tal y como estaban en la última instantánea confirmada en dicha rama `master`. Esto supone que los cambios que hagas desde este momento en adelante, divergirán de la antigua versión del proyecto. Básicamente, lo que se está haciendo es rebobinar el trabajo que habías hecho temporalmente en la rama `testing`; de tal forma que puedas avanzar en otra dirección diferente.

Nota: Saltar entre ramas cambia archivos en tu directorio de trabajo

Es importante destacar que cuando saltas a una rama en Git, los archivos de tu directorio de trabajo cambian. Si saltas a una rama antigua, tu directorio de trabajo retrocederá para verse como lo hacía la última vez que confirmaste un cambio en dicha rama. Si Git no puede hacer el cambio limpiamente, no te dejará saltar.

Haz algunos cambios más y confírmalos:

```
$ vim test.rb  
$ git commit -a -m 'made other changes'
```

Ahora el historial de tu proyecto diverge (observar que los registros de las ramas divergen). Has creado una rama y saltado a ella, has trabajado sobre ella; has vuelto a la rama original, y has trabajado también sobre ella. Los cambios realizados en ambas sesiones de trabajo están aislados en ramas independientes: puedes saltar libremente de una a otra según estimes oportuno. Y todo ello simplemente con tres comandos: `git branch`, `git checkout` y `git commit`.

Figura 17. Los registros de las ramas divergen

También puedes ver esto fácilmente utilizando el comando `git log`. Si ejecutas `git log --oneline --decorate --graph --all` te mostrará el historial de tus confirmaciones, indicando dónde están los apuntadores de tus ramas y como ha divergido tu historial.

```
$ git log --oneline --decorate --graph --all
* c2b9e (HEAD, master) made other changes
| * 87ab2 (testing) made a change
|
* f30ab add feature #32 - ability to add new formats to the
* 34ac2 fixed bug #1328 - stack overflow under certain conditions
* 98ca9 initial commit of my project
```

Debido a que una rama Git es realmente un simple archivo que contiene los 40 caracteres de una suma de control SHA-1, (representando la confirmación de cambios a la que apunta), no cuesta nada el crear y destruir ramas en Git. Crear una nueva rama es tan rápido y simple como escribir 41 bytes en un archivo, (40 caracteres y un retorno de carro).

Esto contrasta fuertemente con los métodos de ramificación usados por otros sistemas de control de versiones, en los que crear una rama nueva supone el copiar todos los archivos del proyecto a un directorio adicional nuevo. Esto puede llevar segundos o incluso minutos, dependiendo del tamaño del proyecto, mientras que en Git el proceso es siempre instantáneo. Y además, debido a que se almacenan también los nodos padre para cada confirmación, el encontrar las bases adecuadas para realizar una fusión entre ramas es un proceso automático y generalmente sencillo de realizar. Animando así a los desarrolladores a utilizar ramificaciones frecuentemente.

Vamos a ver el por qué merece la pena hacerlo así.

2.6.2.- Procedimientos Básicos para Ramificar y Fusionar

Vamos a presentar un ejemplo simple de ramificar y de fusionar, con un flujo de trabajo que se podría presentar en la realidad. Imagina que sigues los siguientes pasos:

1. Trabajas en un sitio web.
2. Creas una rama para un nuevo tema sobre el que quieres trabajar.
3. Realizas algo de trabajo en esa rama.

En este momento, recibes una llamada avisándote de un problema crítico que has de resolver. Y sigues los siguientes pasos:

1. Vuelves a la rama de producción original.
2. Creas una nueva rama para el problema crítico y lo resuelves trabajando en ella.
3. Tras las pertinentes pruebas, fusionas (merge) esa rama y la envías (push) a la rama de producción.
4. Vuelves a la rama del tema en que andabas antes de la llamada y continúas tu trabajo.

Procedimientos Básicos de Ramificación

Imagina que estás trabajando en un proyecto y tienes un par de confirmaciones (commit) ya realizadas.

Figura 18. Un registro de confirmaciones corto y sencillo

Decides trabajar en el problema #53, según el sistema que tu compañía utiliza para llevar el seguimiento de los problemas. Para crear una nueva rama y saltar a ella, en un solo paso, puedes utilizar el comando `git checkout` con la opción `-b`:

```
$ git checkout -b iss53
Switched to a new branch "iss53"
```

Esto es un atajo para:

```
$ git branch iss53  
$ git checkout iss53
```

Figura 19. Crear un apuntador a la rama nueva

Trabajas en el sitio web y haces algunas confirmaciones de cambios (commits). Con ello avanzas la rama `iss53`, que es la que tienes activada (checked out) en este momento (es decir, a la que apunta HEAD):

```
$ vim index.html  
$ git commit -a -m 'added a new footer [issue 53]'
```

Figura 20. La rama `iss53` ha avanzado con tu trabajo

Entonces, recibes una llamada avisándote de otro problema urgente en el sitio web y debes resolverlo inmediatamente. Al usar Git, no necesitas mezclar el nuevo problema con los cambios que ya habías realizado sobre el problema #53; ni tampoco perder tiempo revirtiendo esos cambios para poder trabajar sobre el contenido que está en producción. Basta con saltar de nuevo a la rama `master` y continuar trabajando a partir de allí.

Pero, antes de poder hacer eso, hemos de tomar en cuenta que si tenemos cambios aún no confirmados en el directorio de trabajo o en el área de preparación, Git no nos permitirá saltar a otra rama con la que podríamos tener conflictos. Lo mejor es tener siempre un estado de trabajo limpio y despejado antes de saltar entre ramas. Y, para ello, tenemos algunos procedimientos (stash y corregir confirmaciones), que vamos a ver más adelante en Guardado rápido y Limpieza. Por ahora, como tenemos confirmados todos los cambios, podemos saltar a la rama **master** sin problemas:

```
$ git checkout master  
Switched to branch 'master'
```

Tras esto, tendrás el directorio de trabajo exactamente igual a como estaba antes de comenzar a trabajar sobre el problema #53 y podrás concentrarte en el nuevo problema urgente. Es importante recordar que Git revierte el directorio de trabajo exactamente al estado en que estaba en la confirmación (commit) apuntada por la rama que activamos (checkout) en cada momento. Git añade, quita y modifica archivos automáticamente para asegurar que tu copia de trabajo luce exactamente como lucía la rama en la última confirmación de cambios realizada sobre ella.

A continuación, es momento de resolver el problema urgente. Vamos a crear una nueva rama **hotfix**, sobre la que trabajar hasta resolverlo:

```
$ git checkout -b hotfix  
Switched to a new branch 'hotfix'  
$ vim index.html  
$ git commit -a -m 'fixed the broken email address'  
[hotfix 1fb7853] fixed the broken email address  
1 file changed, 2 insertions(+)
```

Figura 21. Rama `hotfix` basada en la rama `master` original

Puedes realizar las pruebas oportunas, asegurarte de que la solución es correcta, e incorporar los cambios a la rama `master` para ponerlos en producción. Esto se hace con el comando `git merge`:

```
$ git checkout master
$ git merge hotfix
Updating f42c576..3a0874c
Fast-forward
index.html | 2 ++
1 file changed, 2 insertions(+)
```

Notarás la frase “Fast forward” (“Avance rápido”, en inglés) que aparece en la salida del comando. Git ha movido el apuntador hacia adelante, ya que la confirmación apuntada en la rama donde has fusionado estaba directamente arriba respecto a la confirmación actual. Dicho de otro modo, cuando intentas fusionar una confirmación con otra confirmación accesible siguiendo directamente el historial de la primera, Git simplifica las cosas avanzando el puntero, ya que no hay ningún otro trabajo divergente a fusionar. Esto es lo que se denomina “avance rápido” (“fast forward”).

Ahora, los cambios realizados están ya en la instantánea (snapshot) de la confirmación (commit) apuntada por la rama `master`. Y puedes desplegarlos.

Figura 22. Tras la fusión (merge), la rama `master` apunta al mismo sitio que la rama `hotfix`.

Tras haber resuelto el problema urgente que había interrumpido tu trabajo, puedes volver a donde estabas. Pero antes, es importante borrar la rama `hotfix`, ya que no la vamos a necesitar más, puesto que apunta exactamente al mismo sitio que la rama `master`. Esto lo puedes hacer con la opción `-d` del comando `git branch`:

```
$ git branch -d hotfix
Deleted branch hotfix (3a0874c).
```

Y, con esto, ya estás listo para regresar al trabajo sobre el problema #53.

```
$ git checkout iss53
Switched to branch "iss53"
$ vim index.html
$ git commit -a -m 'finished the new footer [issue 53]'
[iss53 ad82d7a] finished the new footer [issue 53]
1 file changed, 1 insertion(+)
```

Figura 23. La rama `iss53` puede avanzar independientemente

Cabe destacar que todo el trabajo realizado en la rama `hotfix` no está en los archivos de la rama `iss53`. Si fuera necesario agregarlos, puedes fusionar (merge) la rama `master` sobre la rama `iss53` utilizando el comando `git merge master`, o puedes esperar hasta que decidas fusionar (merge) la rama `iss53` a la rama `master`.

Procedimientos Básicos de Fusión

Supongamos que tu trabajo con el problema #53 ya está completo y listo para fusionarlo (merge) con la rama `master`. Para ello, de forma similar a como antes has hecho con la rama `hotfix`, vas a fusionar la rama `iss53`. Simplemente, activa (checkout) la rama donde deseas fusionar y lanza el comando `git merge`:

```
$ git checkout master
Switched to branch 'master'
$ git merge iss53
Merge made by the 'recursive' strategy.
index.html | 1 +
1 file changed, 1 insertion(+)
```

Es algo diferente de la fusión realizada anteriormente con `hotfix`. En este caso, el registro de desarrollo había divergido en un punto anterior. Debido a que la confirmación en la rama actual no es ancestro directo de la rama que pretendes fusionar, Git tiene cierto trabajo extra que hacer. Git realizará una fusión a tres bandas, utilizando las dos instantáneas apuntadas por el extremo de cada una de las ramas y por el ancestro común a ambas.

Figura 24. Git identifica automáticamente el mejor ancestro común para realizar la fusión de las ramas

En lugar de simplemente avanzar el apuntador de la rama, Git crea una nueva instantánea (snapshot) resultante de la fusión a tres bandas, y crea automáticamente una nueva confirmación de cambios (commit) que apunta a ella. Nos referimos a este proceso como "fusión confirmada" y su particularidad es que tiene más de un parent.

Figura 25. Git crea automáticamente una nueva confirmación para la fusión

Vale la pena destacar el hecho de que es el propio Git quien determina automáticamente el mejor ancestro común para realizar la fusión; a diferencia de otros sistemas tales como CVS o Subversion, donde es el desarrollador quien ha de determinar cuál puede ser dicho mejor ancestro común. Esto hace que en Git sea mucho más fácil realizar fusiones.

Ahora que todo tu trabajo ya está fusionado con la rama principal, no tienes necesidad de la rama `iss53`. Por lo que puedes borrarla y cerrar manualmente el problema en el sistema de seguimiento de problemas de tu empresa.

```
$ git branch -d iss53
```

Principales Conflictos que Pueden Surgir en las Fusiones

En algunas ocasiones, los procesos de fusión no suelen ser fluidos. Si hay modificaciones dispares en una misma porción de un mismo archivo en las dos ramas distintas que pretendes fusionar, Git no será capaz de fusionarlas directamente. Por ejemplo, si en tu trabajo del problema #53 has modificado una misma porción que también ha sido modificada en el problema `hotfix`, verás un conflicto como este:

```
$ git merge iss53
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
```

Automatic merge failed; fix conflicts and then commit the result.

Git no crea automáticamente una nueva fusión confirmada (merge commit), sino que hace una pausa en el proceso, esperando a que tú resuelvas el conflicto. Para ver qué archivos permanecen sin fusionar en un determinado momento conflictivo de una fusión, puedes usar el comando `git status`:

```
$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")
```

Unmerged paths:

(use "git add ..." to mark resolution)

both modified: index.html

no changes added to commit (use "git add" and/or "git commit -a")

Todo aquello que sea conflictivo y no se haya podido resolver, se marca como "sin fusionar" (~~unmerged~~) a los archivos conflictivos unos marcadores especiales de resolución de conflictos que te guiarán cuando abras manualmente los archivos implicados y los edites para corregirlos. El archivo conflictivo contendrá algo como:

```
contact : email.support@github.com
```

```
=====
```

```
please contact us at support@github.com
```

```
>>>>> iss53:index.html
```

Donde nos dice que la versión en HEAD (la rama `master`, la que habías activado antes de lanzar el comando de fusión) contiene lo indicado en la parte superior del bloque (todo lo que está encima de `=====`) y que la versión en `iss53` contiene el resto, lo indicado en la parte inferior del bloque. Para resolver el conflicto, has de elegir manualmente el contenido de uno o de otro lado. Por ejemplo, puedes optar por cambiar el bloque, dejándolo así:

```
please contact us at email.support@github.com
```

Esta corrección contiene un poco de ambas partes y se han eliminado completamente las líneas `>>>>>`, `=====` y `>>>>>`. Tras resolver todos los bloques conflictivos, has de lanzar comandos `git add` para marcar cada archivo modificado. Marcar archivos como preparados (staged) indica a Git que sus conflictos han sido resueltos.

Si en lugar de resolver directamente prefieres utilizar una herramienta gráfica, puedes usar el

Si en lugar de resolver directamente preferes utilizar una herramienta gráfica, puedes usar el comando `git mergetool`, el cual arrancará la correspondiente herramienta de visualización y te permitirá ir resolviendo conflictos con ella:

```
$ git mergetool
```

This message is displayed because 'merge.tool' is not configured.

See 'git mergetool --tool-help' or 'git help config' for more details.

'git mergetool' will now attempt to use one of the following tools:

opendiff kdiff3 tkdiff xxdiff meld tortoisemerge gvimdiff diffuse diffmerge ecmerge p4merge araxis bc3 codecompare vimdiff emerge

Merging:

index.html

Normal merge conflict for 'index.html':

{local}: modified file

{remote}: modified file

Hit return to start merge resolution tool (opendiff):

Si deseas usar una herramienta distinta de la escogida por defecto (en mi caso `opendiff`, porque estoy lanzando el comando en Mac), puedes escogerla entre la lista de herramientas soportadas mostradas al principio ("merge tool candidates") tecleando el nombre de dicha herramienta.

Tras salir de la herramienta de fusión, Git preguntará si hemos resuelto todos los conflictos y la fusión ha sido satisfactoria. Si le indicas que así ha sido, Git marca como preparado (staged) el archivo que acabamos de modificar. En cualquier momento, puedes lanzar el comando `git status` para ver si ya has resuelto todos los conflictos:

```
$ git status
```

On branch master

All conflicts fixed but you are still merging.

(use "git commit" to conclude merge)

Changes to be committed:

modified: index.html

Si todo ha ido correctamente, y ves que todos los archivos conflictivos están marcados como preparados, puedes lanzar el comando `git commit` para terminar de confirmar la fusión. El mensaje de confirmación por defecto será algo parecido a:

```
Merge branch 'isscc3'
```

```
Conflicts:
```

```
  index.html
#
# It looks like you may be committing a merge.
# If this is not correct, please remove the file
# .git/  MERGE_HEAD
# and try again.
```

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
```

```
# On branch master
```

```
# All conflicts fixed but you are still merging.
```

```
#
```

```
# Changes to be committed:
```

```
#   modified: index.html
```

```
#
```

Puedes modificar este mensaje añadiendo detalles sobre cómo has resuelto la fusión, si lo consideras útil para que otros entiendan esta fusión en un futuro. Se trata de indicar por qué has hecho lo que has hecho; a no ser que resulte obvio, claro está.

2.6.3.- Gestión de ramas

Ahora que ya has creado, fusionado y borrado algunas ramas, vamos a dar un vistazo a algunas herramientas de gestión muy útiles cuando comienzas a utilizar ramas de manera avanzada.

El comando `git branch` tiene más funciones que las de crear y borrar ramas. Si lo lanzas sin parámetros, obtienes una lista de las ramas presentes en tu proyecto:

```
$ git branch
iss53
* master
  testing
```

Fíjate en el carácter * delante de la rama `master`: nos indica la rama activa en este momento (la rama a la que apunta `HEAD`). Si hacemos una confirmación de cambios (`commit`), esa será la rama que avance. Para ver la última confirmación de cambios en cada rama, puedes usar el comando `git branch -v`:

```
$ git branch -v
iss53 93b412c fix javascript issue
* master 7a98805 Merge branch 'iss53'
  testing 782fd34 add scott to the author list in the readmes
```

Otra opción útil para averiguar el estado de las ramas, es filtrarlas y mostrar solo aquellas que han sido fusionadas (o que no lo han sido) con la rama actualmente activa. Para ello, Git dispone de las opciones `--merged` y `--no-merged`. Si deseas ver las ramas que han sido fusionadas con la rama activa, puedes lanzar el comando `git branch --merged`:

```
$ git branch --merged
iss53
* master
```

Aparece la rama `iss53` porque ya ha sido fusionada. Las ramas que no llevan por delante el carácter * pueden ser eliminadas sin problemas, porque todo su contenido ya ha sido incorporado a otras ramas.

Para mostrar todas las ramas que contienen trabajos sin fusionar, puedes utilizar el comando `git branch --no-merged`:

```
$ git branch --no-merged
  testing
```

Esto nos muestra la otra rama del proyecto. Debido a que contiene trabajos sin fusionar, al intentar borrarla con `git branch -d`, el comando nos dará un error:

```
$ git branch -d testing
error: The branch 'testing' is not fully merged.
If you are sure you want to delete it, run 'git branch -D testing'.
```

Si realmente deseas borrar la rama y perder el trabajo contenido en ella, puedes forzar el borrado con la opción `-D`, tal y como indica el mensaje de ayuda.

2.6.4.- Flujos de trabajo ramificados

Ahora que ya has visto los procedimientos básicos de ramificación y fusión, ¿qué puedes o qué debes hacer con ellos? En este apartado vamos a ver algunos de los flujos de trabajo más comunes, de tal forma que puedas decidir si te gustaría incorporar alguno de ellos a tu ciclo de desarrollo.

Ramas de Largo Recorrido

Por la sencillez de la fusión a tres bandas de Git, el fusionar una rama a otra varias veces a lo largo del tiempo es fácil de hacer. Esto te posibilita tener varias ramas siempre abiertas, eirlas usando en diferentes etapas del ciclo de desarrollo; realizando fusiones frecuentes entre ellas.

Muchos desarrolladores que usan Git llevan un flujo de trabajo de esta naturaleza, manteniendo en la rama `master` únicamente el código totalmente estable (el código que ha sido o que va a ser liberado) y teniendo otras ramas paralelas denominadas `desarrollo` o `siguiente`, en las que trabajan y realizan pruebas. Estas ramas paralelas no suelen estar siempre en un estado estable; pero cada vez que sí lo están, pueden ser fusionadas con la rama `master`. También es habitual el incorporarle (pull) ramas puntuales (ramas temporales, como la rama `iss53` del ejemplo anterior) cuando las completamos y estamos seguros de que no van a introducir errores.

En realidad, en todo momento estamos hablando simplemente de apuntadores moviéndose por la línea temporal de confirmaciones de cambio (commit history). Las ramas estables apuntan hacia posiciones más antiguas en el historial de confirmaciones, mientras que las ramas avanzadas, las que van abriendo camino, apuntan hacia posiciones más recientes.

Figura 26. Una vista lineal del ramificado progresivo estable

Podría ser más sencillo pensar en las ramas como si fueran silos de almacenamiento, donde grupos de confirmaciones de cambio (commits) van siendo promocionados hacia silos más estables a medida que son probados y depurados.

Figura 27. Una vista tipo “silo” del ramificado progresivo estable

Este sistema de trabajo se puede ampliar para diversos grados de estabilidad. Algunos proyectos muy grandes suelen tener una rama denominada `propuestas` o `pu` (del inglés “proposed updates”, propuesta de actualización), donde suele estar todo aquello que es integrado desde otras ramas, pero que aún no está listo para ser incorporado a las ramas `siguiente` o `master`. La idea es mantener siempre diversas ramas en diversos grados de estabilidad; pero cuando alguna alcanza un estado más estable, la fusionamos con la rama inmediatamente superior a ella. Aunque no es obligatorio el trabajar con ramas de larga duración, realmente es práctico y útil, sobre todo en proyectos largos o complejos.

Ramas Puntuales

Las ramas puntuales, en cambio, son útiles en proyectos de cualquier tamaño. Una rama puntual es aquella rama de corta duración que abres para un tema o para una funcionalidad determinada. Es algo que nunca habrías hecho en otro sistema VCS, debido a los altos costos de crear y fusionar ramas en esos sistemas. Pero en Git, por el contrario, es muy habitual el crear, trabajar con, fusionar y eliminar ramas varias veces al día.

Tal y como has visto con las ramas `iss53` y `hotfix` que has creado en la sección anterior. Has hecho algunas confirmaciones de cambio en ellas, y luego las has borrado tras fusionarlas con la rama principal. Esta técnica te posibilita realizar cambios de contexto rápidos y completos y, debido a que el trabajo está claramente separado en silos, con todos los cambios de cada tema en su propia rama, te será mucho más sencillo revisar el código y seguir su evolución. Puedes mantener los cambios ahí durante minutos, días o meses; y fusionarlos cuando realmente estén listos, sin importar el orden en el que fueron creados o en el que comenzaste a trabajar en ellos.

Por ejemplo, puedes realizar cierto trabajo en la rama `master`, ramificar para un problema concreto (rama `iss91`), trabajar en él un rato, ramificar una segunda vez para probar otra manera de resolverlo (rama `iss92v2`), volver a la rama `master` y trabajar un poco más, y, por último, ramificar temporalmente para probar algo de lo que no estás seguro (rama `dumbidea`). El historial de confirmaciones (commit history) será algo parecido esto:

Figura 28. Múltiples ramas puntuales

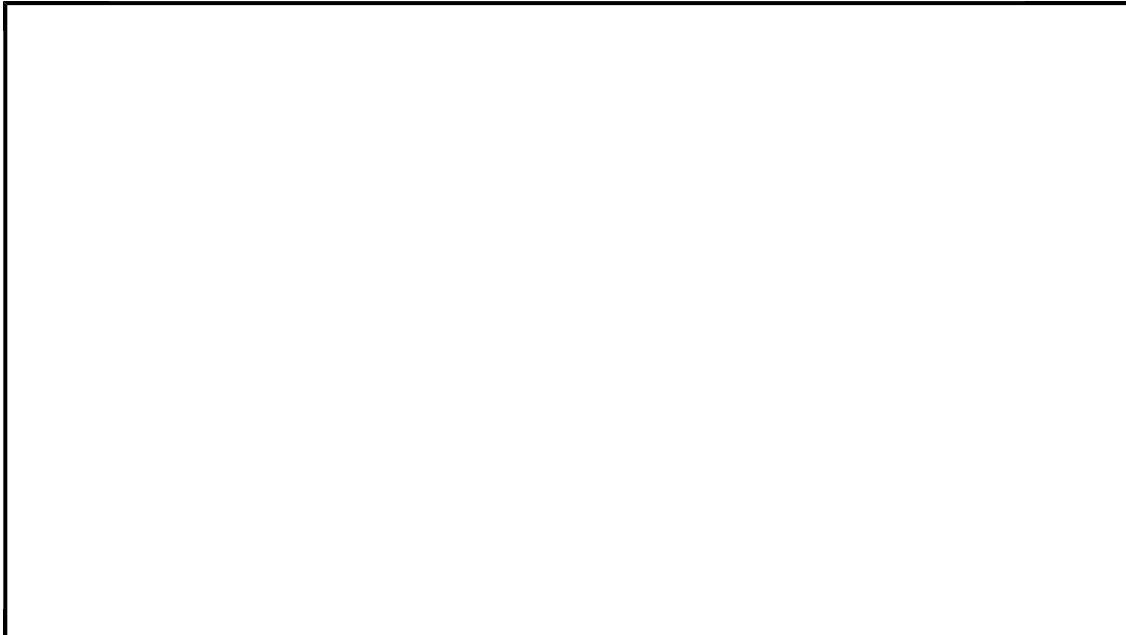
En este momento, supongamos que te decides por la segunda solución al problema (rama `iss92v2`); y que, tras mostrar la rama `dumbidea` a tus compañeros, resulta que les parece una idea genial. Puedes descartar la rama `iss91` (perdiendo las confirmaciones C5 y C6), y fusionar las otras dos. El historial será algo parecido a esto:

Figura 29. El historial tras fusionar `dumbidea e iss91v2`

Hablaremos un poco más sobre los distintos flujos de trabajo de tu proyecto Git en Git en entornos distribuidos, así que antes de decidir qué estilo de ramificación usará tu próximo proyecto, asegúrate de haber leído ese capítulo.

Es importante recordar que, mientras estás haciendo todo esto, todas las ramas son completamente locales. Cuando ramificas y fusionas, todo se realiza en tu propio repositorio Git. No hay ningún tipo de comunicación con ningún servidor.

2.6.5.- Actividad de ampliación



Licencia: Dominio público

Obra publicada con [Licencia Creative Commons Reconocimiento Compartir igual 4.0](#)

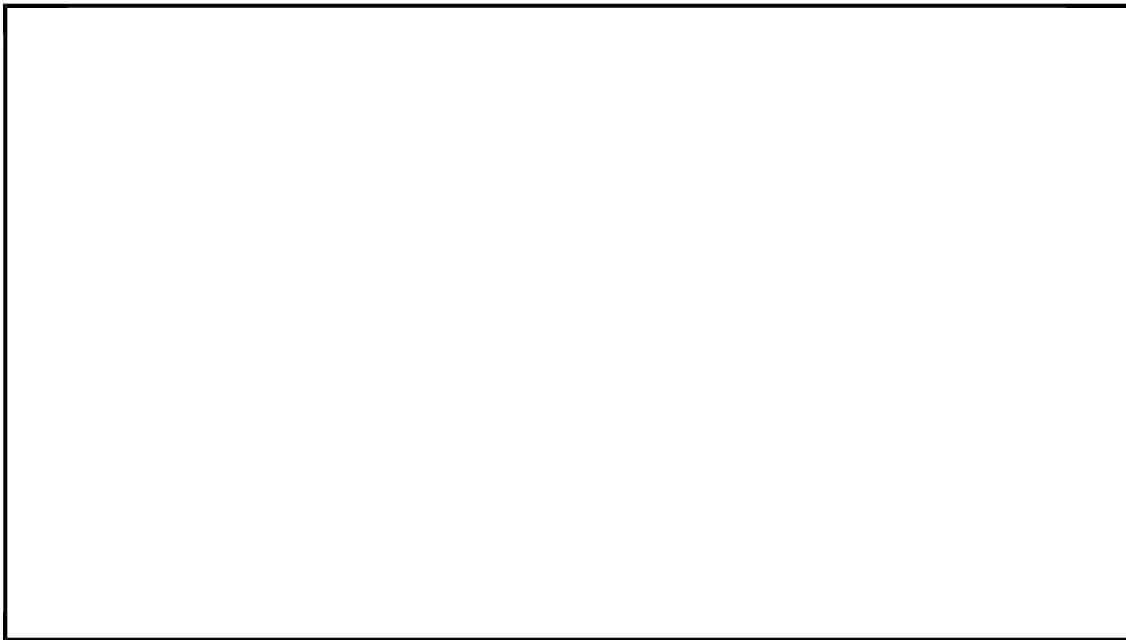
3.- Visual Estudio Code con Git y GitHub

En esta sección aprenderemos a integrar el sistema de control de versiones Git con Visual Estudio Code. ¿Estás listo? Adelante....

Obra publicada con [Licencia Creative Commons Reconocimiento Compartir igual 4.0](#)

3.1.- Ramas o branches

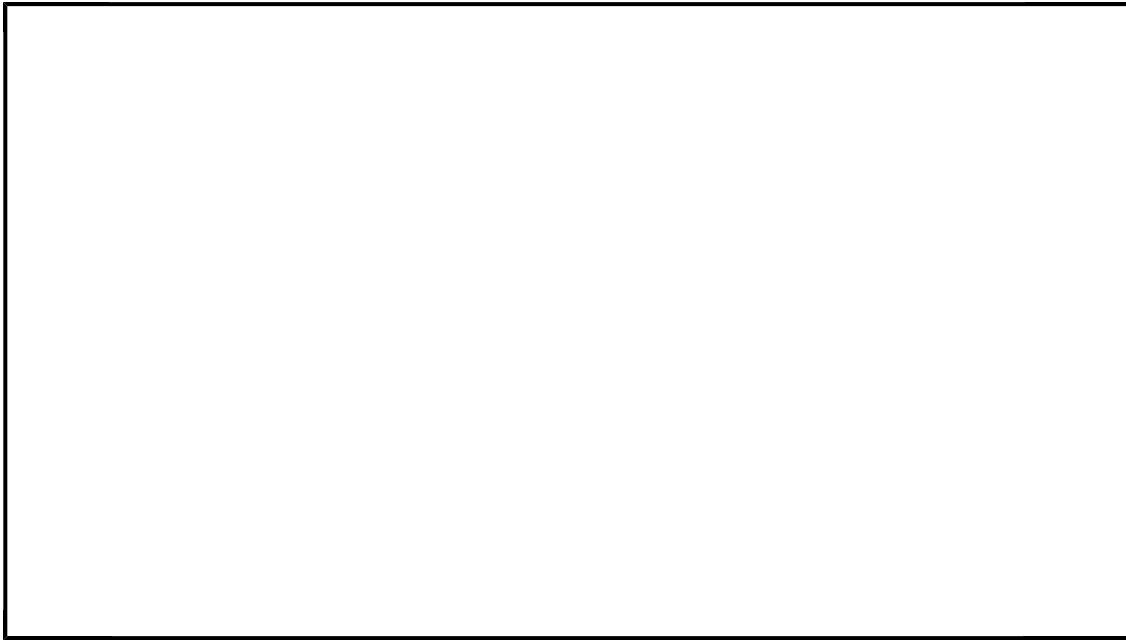
Licencia: Dominio público



Licencia: Dominio público

Obra publicada con [Licencia Creative Commons Reconocimiento Compartir igual 4.0](#)

3.2- Subiendo y sincronizando proyectos en GitHub



Licencia: Dominio público

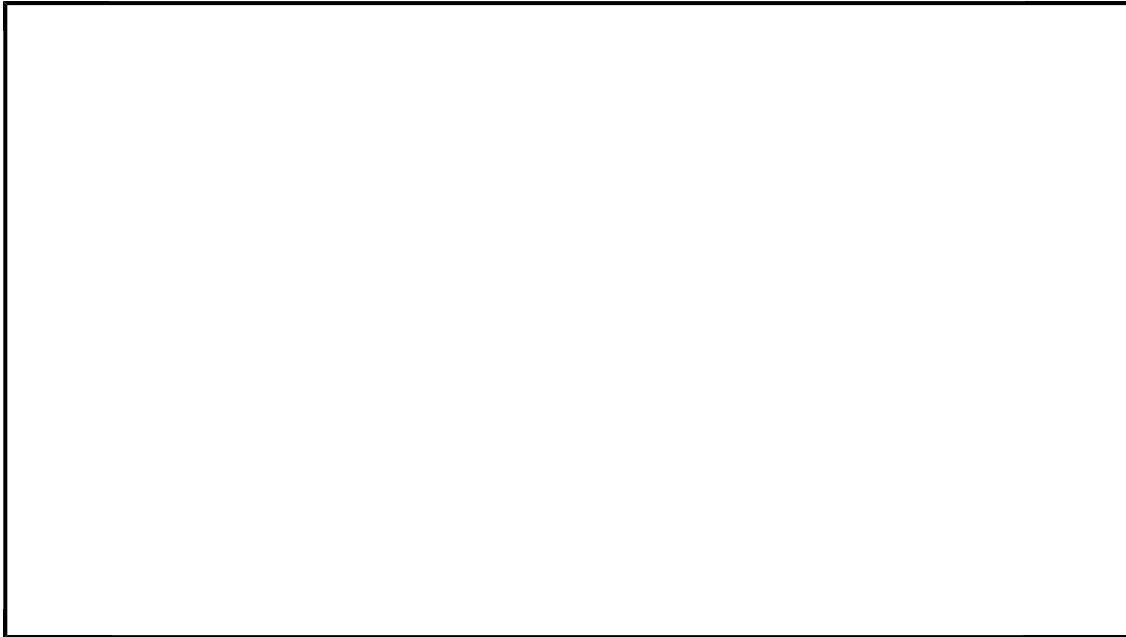
Obra publicada con [Licencia Creative Commons Reconocimiento Compartir igual 4.0](#)

3.3.- Ramas en GitHub

Licencia: Dominio público

Obra publicada con [Licencia Creative Commons Reconocimiento Compartir igual 4.0](#)

3.4.- Fork

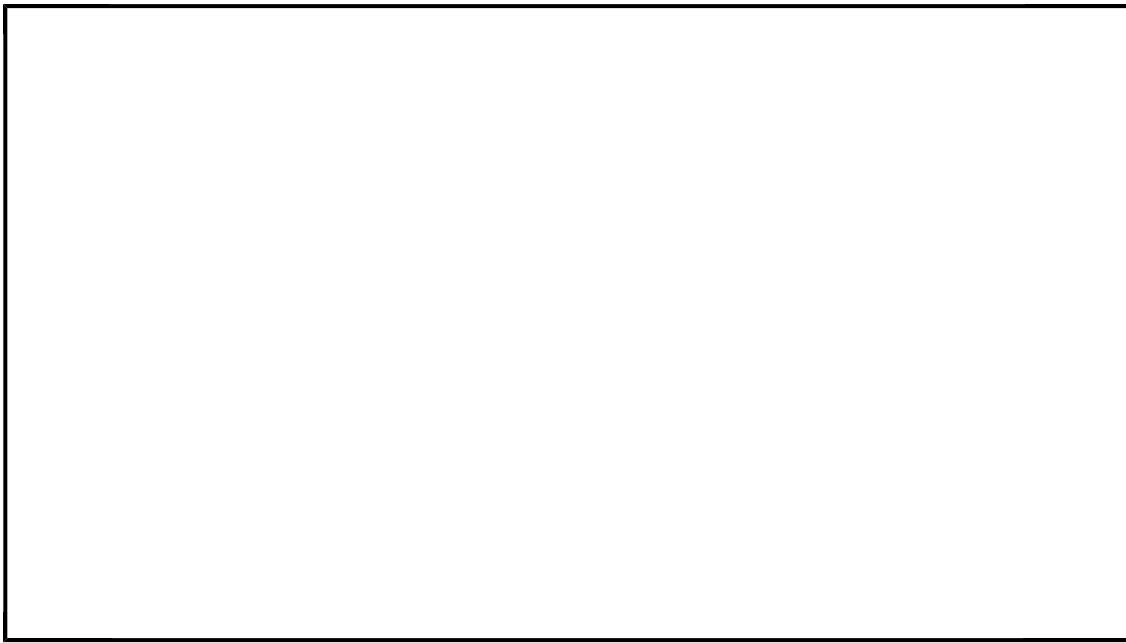


Licencia: Dominio público

Obra publicada con [Licencia Creative Commons Reconocimiento Compartir igual 4.0](#)

4.- Git con Eclipse

En esta sección aprenderemos a integrar el sistema de control de versiones Git con Eclipse.
¿Estás listo? Adelante....



Licencia: Dominio público

Obra publicada con [Licencia Creative Commons Reconocimiento Compartir igual 4.0](#)