

Aprende git

Pablo Hinojosa y JJ Merelo

Licencia

Esta obra está sujeta a la licencia Reconocimiento-CompartirIgual 4.0 Internacional de Creative Commons. Para ver una copia de esta licencia, visite <http://creativecommons.org/licenses/by-sa/4.0/>.

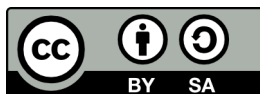


Figure 1: cc-by-sa

Prólogo y agradecimientos

Este libro se escribió originalmente para el [curso de git](#) impartido en el Centro de Enseñanzas Virtuales de la UGR en abril y mayo de 2014. Por lo tanto, agradecemos a los alumnos la retroalimentación ofrecida, especialmente a Manuel Cogolludo, que se revisó el material y nos hizo grandes sugerencias.

También es una primera edición, por lo que seguramente habrá algún (ciento de) fallo. Pero dado que es un libro libre, puedes corregir esos fallos y hacerte tu propia versión en GitHub o [hacernos sugerencias y comentarios](#) que atenderemos rápida y debidamente. Si te atreves una vez escrito el libro a hacer los cambios tú mismo y a solicitarnos la incorporación a versiones posteriores del libro, te lo agradeceremos aún más y aparecerás en este prólogo.

El libro está dirigido a una persona que no sepa programar, o que sepa sin que haga falta ningún lenguaje específico. Sí hace falta cierto manejo de la línea de órdenes y el uso de sistemas operativos adecuados para el desarrollo, lo que probablemente usas de todas formas si estás acostumbrado a usar la línea de órdenes. Sólo al final hay algunos programas con la idea de que se aprenda a modificarlos para usarlos en git, no tanto a dominarlos. En general, git va bien con cualquier lenguaje de programación.

Agradecimientos adicionales

Muchas personas han contribuido con [correcciones o aportaciones y están todas ahí, en el gráfico de GitHub](#) pero está bien nombrarlas; agradecemos a Miguel Ángel Pedregosa, Alfonso Romero y a [Juanmi](#), [Fernando Tricas](#) y [Diego](#) su revisión extensiva de los fuentes y las correcciones aportadas.

Introducción

Objetivos

- Saber qué es el Software Libre
- Aprender qué es un Sistema de Control de Versiones
- Ver los Sistema de Control de Versiones y sus tipos
- Conocer las principales características de git

Software Libre

En los últimos años, el software libre se ha ido extendiendo hasta abarcar la mayor parte de sistemas de Internet, supercomputadores o servicios de red, llegando finalmente a ordenadores personales, tablets, teléfonos e incluso electrodomésticos.

Llamamos [Software Libre](#) al que permite al usuario ejercer una serie de libertades, a saber:

- la libertad de usar el programa, con cualquier propósito.
- la libertad de estudiar cómo funciona el programa y modificarlo, adaptándolo a tus necesidades.
- la libertad de distribuir copias del programa, con lo cual puedes ayudar a tu prójimo.
- la libertad de mejorar el programa y hacer públicas esas mejoras a los demás, de modo que toda la comunidad se beneficie.

Para que un programa sea libre hacen falta dos requisitos fundamentales:

- Una licencia que permita el uso, modificación y distribución del programa
- Acceso al código fuente del programa

Además, en la comunidad de software libre se dan algunas peculiaridades, como la internacionalización o el trabajo colaborativo de gran cantidad de desarrolladores, que han promovido la creación de forjas, repositorios y sistemas que permitan compartir ese código de forma abierta y además, administrar y gestionar todo ese trabajo de una forma eficiente.

git es uno de estos sistemas, y es software libre.

Sistemas de control de Versiones

En cualquier proyecto de desarrollo es necesario gestionar los cambios, modificaciones, añadidos etc. que se han hecho a lo largo de la historia de dicho proyecto.

Si se trata de un trabajo pequeño, de corta duración y es llevado a cabo por un solo programador, quizás un archivo histórico de copias de seguridad pueda ser suficiente, pero esto se vuelve claramente insuficiente al crecer la complejidad del proyecto.

Algunos de los problemas más habituales a los que se enfrenta cualquier persona que participe en un desarrollo informático son:

- Dos o más programadores modifican el mismo archivo de código y hay que gestionar ese conflicto.
- Es necesario mantener (al menos) dos versiones del proyecto, una en producción y otra en desarrollo.
- Algún cambio ha sido desechado y es necesario regresar a una fase anterior del proyecto.
- Se inicia un “fork” o proyecto derivado a partir del estado actual del proyecto.

Llevar a cabo la administración de estos aspectos (y de muchos otros) manualmente es materialmente imposible, y es para ello para lo que se inventaron los Sistemas de Control de Versiones.

Debido al crecimiento en extensión y complejidad del software los Sistemas de Control de Versiones están tomando cada vez más importancia. En especial, los proyectos de software libre, que tienden a aunar los esfuerzos de un gran número de programadores trabajando simultáneamente en múltiples versiones del código, han estado a la vanguardia del uso de estas herramientas.

Estos sistemas nacieron para gestionar código fuente, pero pueden ser usados para cualquier tipo de documentación (aunque no pueden aprovechar todo su potencial en archivos que no sean de texto plano). Por ejemplo, este curso ha sido desarrollado bajo git y su código puede encontrarse en su repositorio oficial en github.com/oslugr/curso-git

Tipos de Sistemas de Control de Versiones

Existen muchos Sistemas de Control de Versiones como [CVS](#), [Subversion](#), [Bazaar](#), [Mercurial](#) o, por supuesto, [Git](#), pero todos pueden clasificarse en dos tipos fundamentales según su modo de trabajo.

Sistemas centralizados Los sistemas de control de versiones centralizados fueron los primeros en aparecer y son los de funcionamiento más simple e intuitivo.

En ellos, existe un repositorio central donde se archiva el código y toda la información asociada (marchas de tiempo, autores de los cambios, etc). Los distintos desarrolladores trabajan con copias de ese código que actualizan a partir del servidor central, a donde también envían sus cambios.

Es decir, la versión “oficial” de referencia es la que hay en ese repositorio, y todos los programadores sincronizan sus versiones con esa.

Los programas más conocidos de este tipo son CVS y Subversion.

Sistemas distribuidos. Los sistemas distribuidos son más complejos, pero a cambio ofrecen una mayor flexibilidad.

En estos sistemas no existe un servidor central, sino que cada programador tiene su propio repositorio que puede sincronizar con el de cada uno de los demás.

Hay que hacer notar que, aunque no es necesario en este tipo de arquitectura, en la práctica se suele definir un repositorio de referencia para albergar la versión “oficial” del software.

Los sistemas distribuidos más conocidos son Bazaar y, por supuesto, git.

git

Git nació para ser el Sistema de Control de Versiones del kernel de Linux (de hecho, fue originalmente programado por el mismo *Linus Torvalds*) y por ello está preparado para proyectos grandes, con muchos desarrolladores y un gran número de ramas.

Se trata, como ya hemos dicho, de un Sistema de Control de Versiones distribuido, por lo que cada programador cuenta con su propio repositorio. Esto hace que, salvo cuando llega el momento de sincronizar con otro repositorio, todo el trabajo se haga en local, con ventajas como la velocidad o que se pueda trabajar sin acceso a la red.

git es software libre, y su código está disponible en su [repositorio de GitHub](#).

El sitio oficial de git es git-scm.com

Git es también multiplataforma, y existen versiones para Linux, Mac, Windows y Solaris.

En este curso se usará Linux para los ejemplos y referencias, y se recomienda encarecidamente su uso. Otros sistemas operativos no están tan preparador para algunas tareas como administrar sesiones remotas, etc.

En cualquier caso, el uso del propio git en todos ellos es similar por lo que, salvo las particularidades y limitaciones que pueda tener cada uno, no hay ningún problema a la hora de seguir este curso desde otro sistema operativo.

Abrir una cuenta en Github

Más adelante en este curso se hablará de Github y se darán detalles sobre su uso pero, por ahora, será suficiente para nosotros con saber abrir una cuenta (que usaremos para enviar nuestros ejercicios).

Desde la propia [página principal de la web de GitHub](#) y como cualquier otro registro, se nos solicitan sólo tres datos: nombre o nick, e-mail y contraseña:

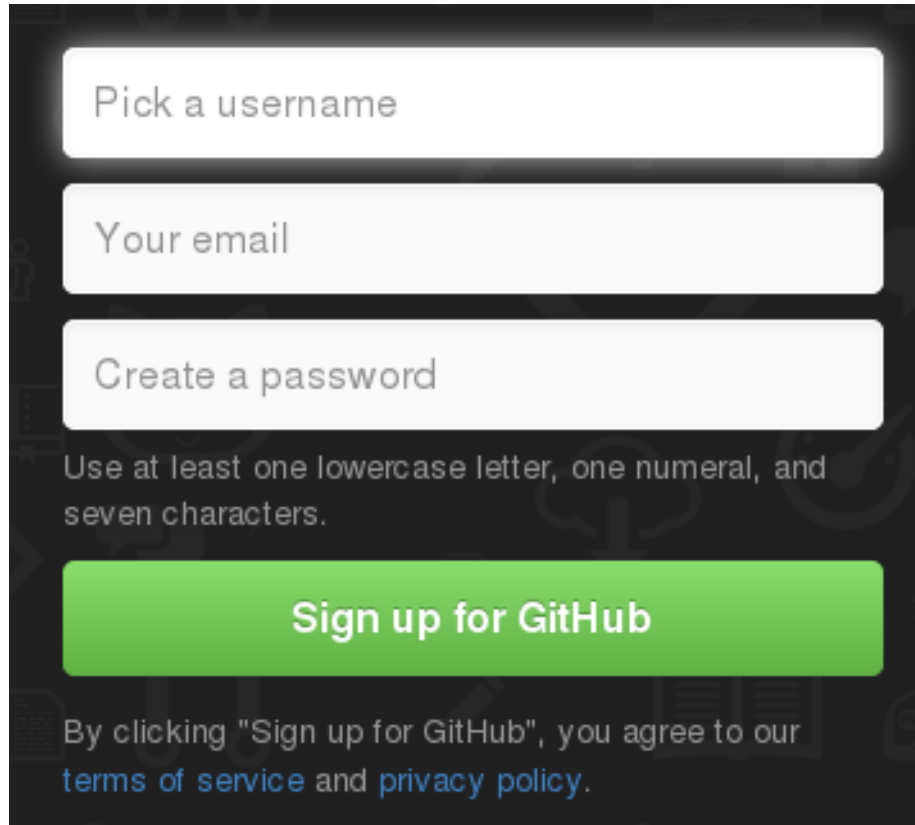
The image shows the GitHub sign-up interface. It features three white input fields on a dark background. The first field is labeled 'Pick a username', the second 'Your email', and the third 'Create a password'. Below the password field, there is a text requirement: 'Use at least one lowercase letter, one numeral, and seven characters.' A prominent green button with the text 'Sign up for GitHub' is positioned below the requirements. At the bottom, a line of text states: 'By clicking "Sign up for GitHub", you agree to our [terms of service](#) and [privacy policy](#).'

Figure 2: Formulario de GitHub

Uso básico de git

Objetivos

En este apartado veremos cómo se usa git de forma básica para trabajar en modo monousuario y con un repositorio centralizado.

- Instalar Git
- Crear un repositorio
- Mantener un control de cambios sobre nuestros archivos
- Sincronizar dos o más repositorios
- Tareas básicas de git

Instalar git

`git` es software libre y se puede instalar en cualquier sistema operativo. A continuación los más populares, empezando por el que aconsejamos para desarrollar software en general, Linux.

En Linux Instalar git en Linux es tan simple como usar tu gestor de paquetes favorito. Por ejemplo (recuerda que normalmente necesitarás privilegios de *root* para instalar cualquier programa):

En Arch Linux `# pacman -S git`

En sistemas Debian, Ubuntu, Mint... `# apt-get install git`

En Gentoo `# emerge --ask --verbose dev-vcs/git`

En sistemas Red Hat, Fedora: `# yum install git`

En Mac Hay dos maneras de instalar Git en Mac, la más fácil es utilizar el instalador gráfico:

[Git for OS X](#)

En Windows Para instalar git en Windows debemos descargar el programa instalador en su web oficial en <http://git-scm.com/downloads>.

Una vez descargado, sólo tenemos que ejecutarlo y se abrirá una ventana que nos irá solicitando paso a paso los datos necesarios para la instalación. Pulsaremos el botón “Next” para comenzar.

Nos pasará a la página de licencia (*es una licencia libre que permite copiar, modificar y distribuir el programa*).

La siguiente es una ventana que nos permite elegir el lugar de instalación. Si no tenemos especial interés en que sea otro, el que viene por defecto está bien.



Figure 3: Instalación de git en Windows (1)

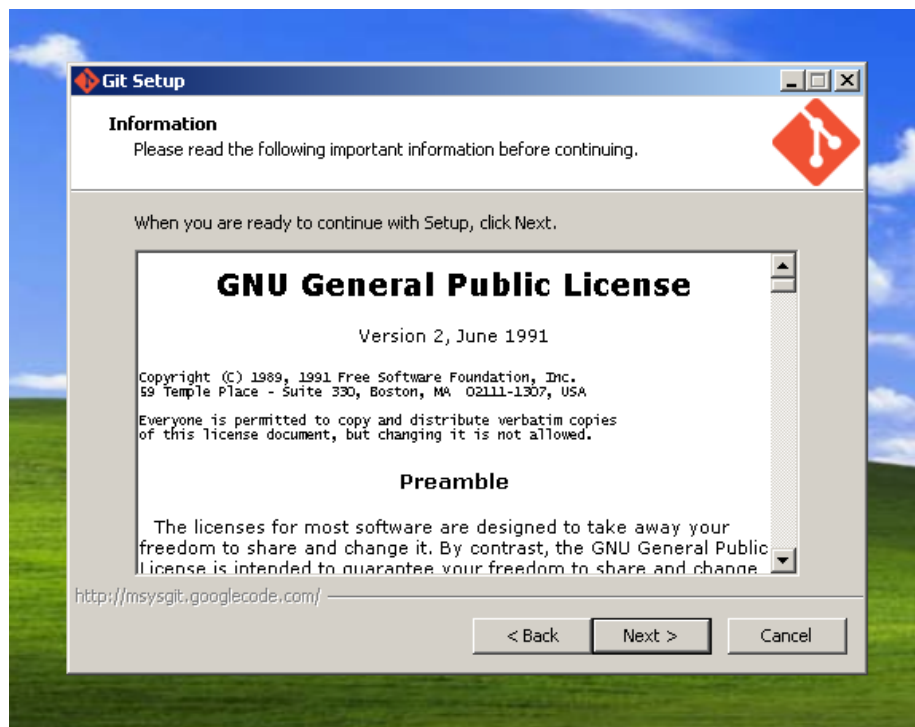


Figure 4: Instalación de git en Windows (2)

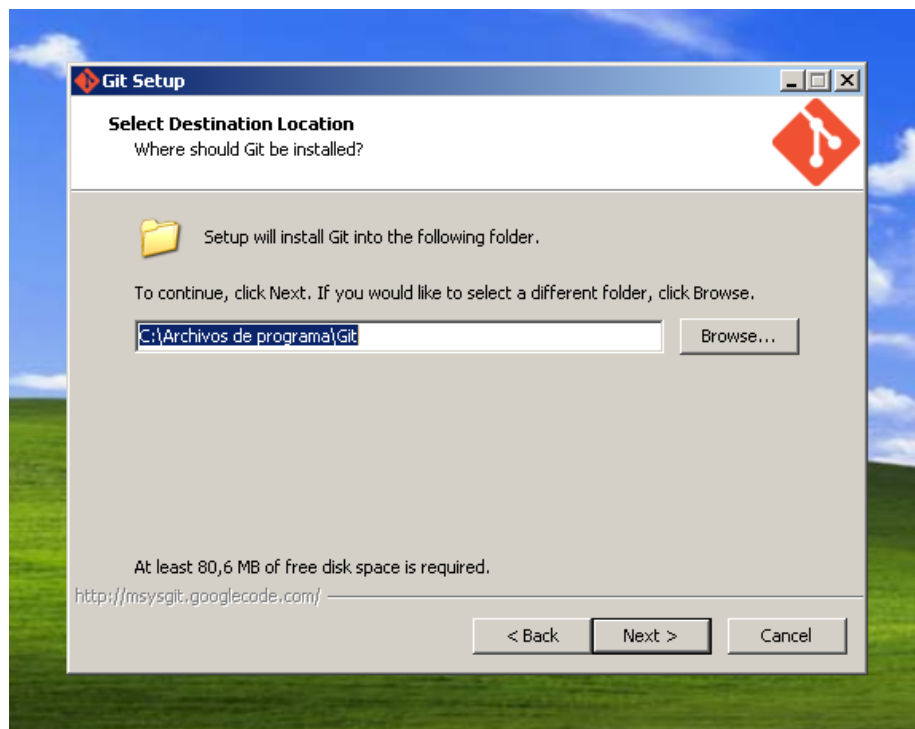


Figure 5: Instalación de git en Windows (3)

En la siguiente, podemos elegir una serie de cosas como el que aparezcan iconos de git en Inicio Rápido y el Escritorio o tener dos nuevas órdenes en el menú contextual (*el que aparece al hacer clic derecho con el ratón en una ventana*) para iniciar una ventana de git en esa carpeta.

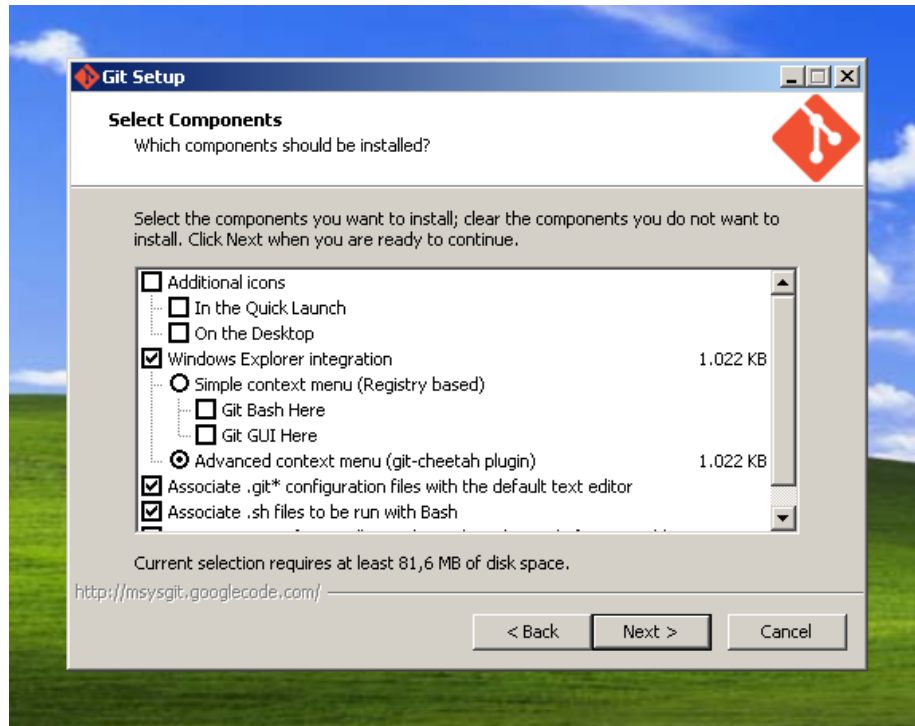


Figure 6: Instalación de git en Windows (4)

En la siguiente ventana se nos permite cambiar el nombre del grupo de programas que aparecerá en el menú Inicio.

Las siguientes dos opciones configuran aspectos avanzados de 'git', concretamente el uso del prompt y el manejo de retornos de carro. Para un usuario novel son adecuadas las opciones por defecto.

Y, por fin, hemos finalizado nuestra instalación.

A partir de este momento podemos ir al menú inicio como se indica en la imagen, y ejecutar "Git Bash", lo que abrirá una consola donde podremos interactuar con 'git' tal como se ve en este curso.

(Al terminar todos estos pasos, y como se ven en la imagen, también se instalará una versión gráfica "Git GUI", pero en este curso se seguirá la interfaz de línea de comandos)

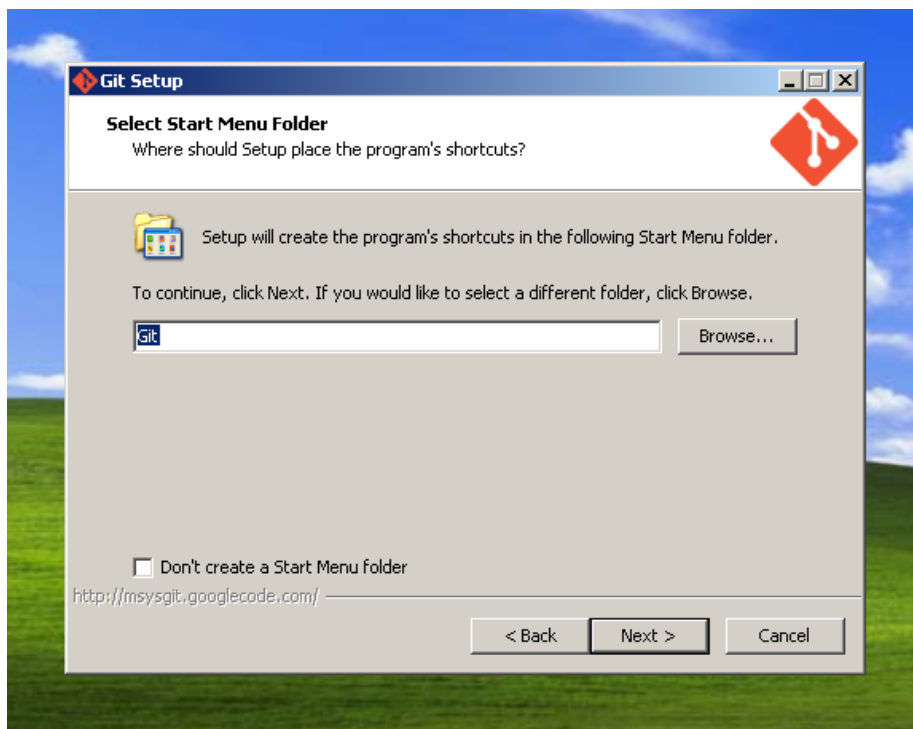


Figure 7: Instalación de git en Windows (5)

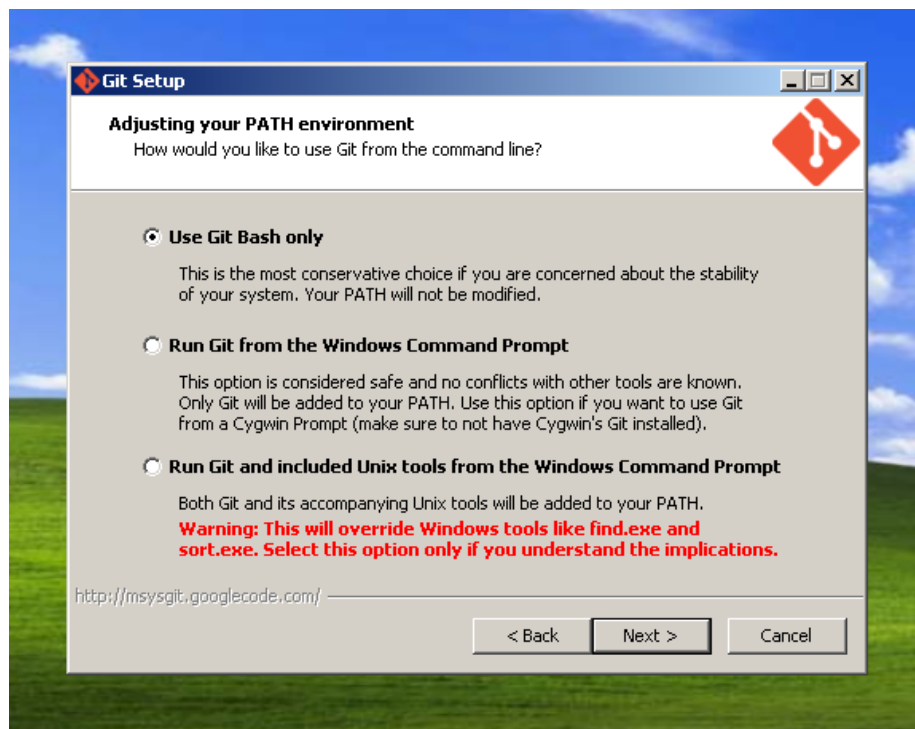


Figure 8: Instalación de git en Windows (6)

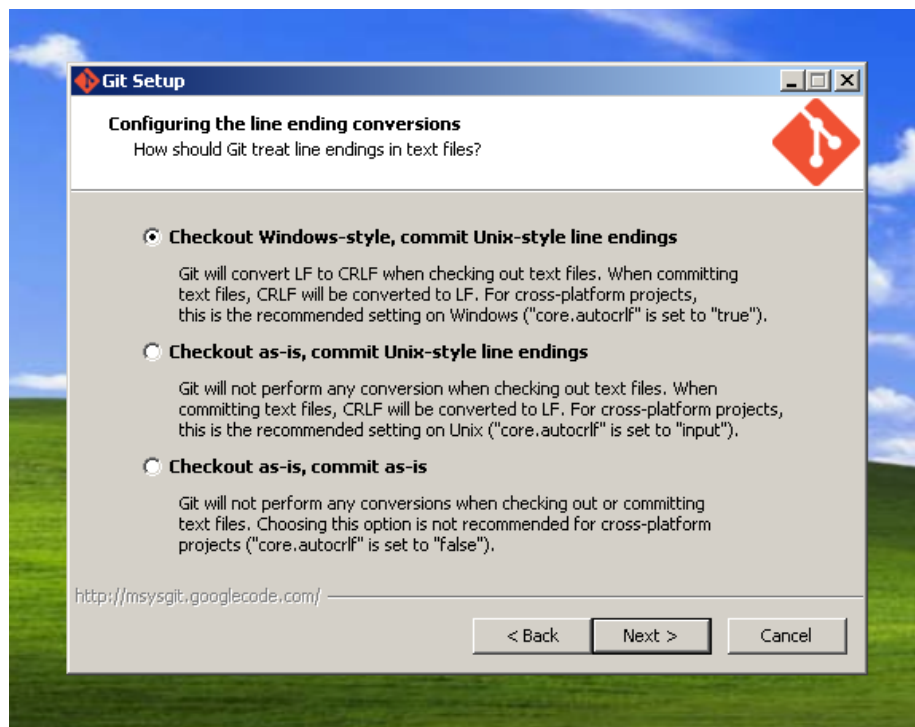


Figure 9: Instalación de git en Windows (7)



Figure 10: Instalación de git en Windows (8)



Figure 11: Instalación de git en Windows (9)

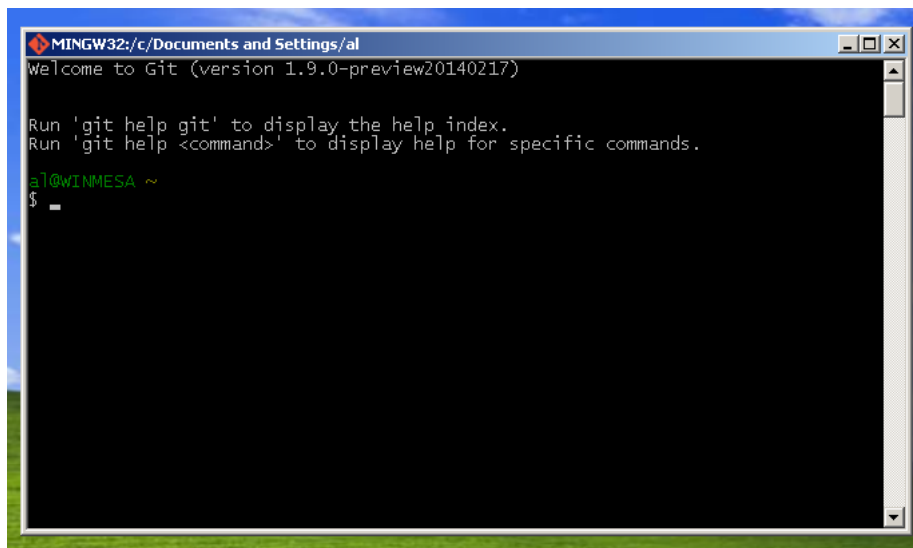


Figure 12: Instalación de git en Windows (10)

Los autores de este libro no somos partidarios del uso de Windows para desarrollo de software. Pero, si tienes que hacerlo porque no te queda otro remedio o porque te gusta, otra alternativa es usar [git for Windows](#) o [msysgit](#), un entorno completo que incluye un intérprete de órdenes y un entorno gráfico para trabajar desde él.

Clientes GUI para Linux, Windows y Mac

En este curso se seguirá la interfaz de *línea de comandos* (o *línea de órdenes*), pero existen varias aplicaciones para diversos sistemas operativos que permiten interactuar gráficamente (*Interfaz GUI*) con `git` de forma más o menos completa.

[GUI Mac](#)

[GUI Windows](#)

[GUI for Linux, Windows y Mac](#)

Empezando a usar git

Git es un programa en línea de comandos, y se te supone un conocimiento básico del manejo de esta (cosas como moverse por el árbol de directorios y poco más). No es necesario saber nada complejo, sólo los rudimentos básicos.

Configurar Lo primero que hay que hacer antes de empezar a usar git es configurar un par de parámetros básicos que nos identifican como usuario, que son nuestro correo electrónico y nuestro nombre.

Git usará estos datos para identificar nuestros aportes o modificaciones a la hora de mostrarlos en logs etc.

Configurar estos parámetros es muy fácil. Desde la línea de comandos escribimos las siguientes órdenes:

```
git config --global user.name "Nombre que quieras mostrar"
```

y

```
git config --global user.email "correo@electroni.com"
```

¿qué acabamos de hacer? Veámoslo, paso a paso:

Todos los comandos de git empiezan con la palabra **git**.

En este caso, el comando en sí mismo es **config**, que sirve para configurar varias opciones de git, en el primer caso **user.name** y en el segundo **user.email**.

Habrás notado que hay un parámetro **--global** en cada uno de los comandos. Este sirve para decirle a git que esos datos se aplican a todos los repositorios que abras.

Si quieres que algún repositorio concreto use unos datos distintos, puedes llamar al mismo comando, desde el directorio de ese repositorio, pero usando el parámetro **--local** en lugar de **--global**.

Las opciones que configures como **--global** se almacenarán en un archivo de tu carpeta home, llamado **.gitconfig**. Las opciones **--local** lo harán en un archivo **config** dentro del directorio **.git** de tu proyecto.

Hay más opciones que se pueden configurar, puedes verlas (y ver los valores que tienen) con el comando:

```
git config --list
```

Si te has equivocado al escribir alguno de estos datos o quieres cambiarlo, sólo tienes que volver a ejecutar el comando correspondiente de nuevo, y sobrescribirá los datos anteriores.

Una opción de configuración muy cómoda es **git config --global color.ui true**, que hace que el interfaz de git use (si es posible) colores para resaltar distintos aspectos en el texto de sus mensajes.

Iniciando un repositorio Un repositorio de git no es más que un directorio de nuestro ordenador que está bajo el control de git. En la práctica, esto significa que en el directorio raíz de nuestro proyecto hay otro directorio oculto llamado “.git” donde se guardan, por ejemplo, los archivos para el control de historiales y los cambios.

Para iniciar un repositorio sólo hay que situarse en el directorio de nuestro proyecto (el que contiene o va a contener los archivos que queremos controlar) y ejecutar la siguiente orden:

```
git init
```

Si todo va bien, este comando responderá algo parecido a “Initialized empty Git repository in /ruta/a/mi/proyecto/.git/”, que significa que ya tienes creado tu primer repositorio. Vacío, pero por algo hay que empezar.

Clonando un repositorio Un repositorio también puede iniciarse copiando (*clonando*) otro ya existente.

```
git clone REPOSITORIO
```

por ejemplo:

```
git clone https://github.com/oslugr/repo-ejemplo.git
```

Git usa su propio protocolo “git” para el acceso remoto (también se puede clonar un repositorio local, simplemente indicando el path), pero también soporta otros protocolos como **ssh**, **http**, **https**...

Al contrario que con **git init**, con **git clone** no es necesario crear un directorio para el proyecto. Al clonar se creará un directorio con el nombre del proyecto dentro del que te encuentres al llamar a la orden.

Clonar un repositorio significa copiarlo completamente. No sólo los archivos, sino todo su historial, cambios realizados, etc. Es decir que en tu repositorio local tendrás exactamente lo mismo que había en el repositorio remoto de donde lo has clonado.

Si has clonado el repositorio del ejemplo anterior (y si no, hazlo ahora), podemos ver un par de cosas interesantes. ¿Recuerdas las orden **git config --list**? Entra en el directorio del repositorio (para ello tendrás que hacer algo como **cd repo-ejemplo/**) y lista las opciones de configuración.

Verás, entre otras muchas, las **user.name** y **user.email** que ya conoces. Pero hay otra que es importante, y es **remote.origin.url**, que debe contener la dirección original del repositorio del que has clonado el tuyo.

Ahora mismo no nos sirve de mucho pero, cuando más adelante trabajemos en red con otros repositorios, nos va a venir bien recordarlo.

IMPORTANTE En adelante, a menos que se diga lo contrario, todos los comandos y órdenes que se indique se deberán ejecutar en

el directorio de nuestro proyecto (o uno de sus subdirectorios, lógicamente). Git reconoce el proyecto con el que está trabajando en función del lugar donde te encuentres al ejecutar los comandos

¿Cómo funciona git?

Antes de continuar, vamos a detenernos un momento para entender el funcionamiento de git.

Cuando trabajas con git lo haces, evidentemente, en un directorio donde tienes tus archivos, los modificas, los borras, creas nuevos, etc.

Ese directorio es lo que llamamos “Directorio de trabajo”, puede contener otros directorios y, de hecho es el que contiene el directorio `.git` del que hablábamos al principio.

Git sabe que tiene que controlar ese directorio, pero no lo hace hasta que se lo digas expresamente.

Más adelante veremos con algo más de detalle la orden `git add`, pero ya te adelanto que lo que hace es preparar los archivos que le indiques poniéndolos en una especie de lista virtual a la que llamamos el “Index”. En Index ponemos los archivos que hemos ido modificando, pero las cosas que están en el “Index” aun no han sido archivadas por git.

Ojo, que algo esté en el index no significa que se borre de tu directorio de trabajo ni nada parecido, el Index es sólo una lista de cosas que tendrás que actualizar en el repositorio porque han cambiado.

Por último, la instrucción `git commit`, que también veremos en breve, es la que realmente envía las cosas que hay en el Index al repositorio. Solo que, en lugar de “repositorio” lo vamos a llamar “HEAD”, porque el lugar exacto al que va puede significar cosas distintas en según que casos, como ya veremos cuando hablemos de ramas y esas cosas.

Lo sé, es todo un poco lioso ahora mismo, pero ya se irá aclarando conforme aprendamos más cosas.

Tú sólo mantén esta secuencia en la cabeza: Directorio de trabajo -> Index -> HEAD

Manteniendo nuestro repositorio al día

Tienes tu repositorio iniciado (o clonado) con una serie de archivos con los que empiezas a trabajar, creándolos, editándolos, modificándolos, etc.

Para que git sepa que tiene que empezar a tener en cuenta un archivo (a esto se le llama *preparar* un archivo), usamos la orden `git add` de este modo:

```
git add NOMBRE_DEL_ARCHIVO
```

Esto, como vimos antes, añadirá el archivo indicado con `NOMBRE_DEL_ARCHIVO` al Index. No lo archivará realmente en el sistema de control de versiones ni hará nada. Sólo le informa de que debe tener en cuenta ese archivo para futuras instrucciones (que es, básicamente, en lo que consiste el Index).

Si intentas añadir al Index un archivo que no existe te dará un error.

También puedes usar *comodines*, con cosas como:

```
git add miarchivo.*
```

(que reconocería, por ejemplo “miarchivo.txt”, “miarchivo.cosas” y “mi-archivo.png”)

o

```
git add miarchivo$.txt
```

(que identificaría cosas como “miarchivo1.txt”, “miarchivo2.txt” y “mi-archivoZ.txt”)

Y, en general, todos los comodines que permita usar tu sistema operativo.

Si, en lugar de un archivo, indicas un directorio, se agregarán al Index todos los archivos de ese directorio.

De este modo, la forma más fácil de agregar todos los archivos al Index es mediante la orden:

```
git add .
```

que añadirá el directorio en el que te encuentras y todo su contenido (incluyendo subdirectorios etc).

Un detalle importante es que, si mandas algo al Index con `git add` y luego lo modificas, no tendrás en Index la última versión, sino lo que hayas hecho hasta el momento del hacer el add.

Esto es muy útil (a veces tienes que hacer cambios que aún no quieres “archivar”) pero puede llevarte a alguna confusión.

Ahora vamos a ver una orden que será tu gran amiga:

```
git status
```

`git status` te da un resumen de cómo están las cosas ahora mismo respecto a la versión del repositorio (concretamente, respecto al HEAD). Qué archivos has modificado, que hay en el Index, etc (también te cuenta cosas como en qué rama estás, pero eso lo veremos más adelante). Cada vez que no tengas muy claro que has cambiado y qué no, consulta `git status`.

En principio, si no has modificado nada, el mensaje básico que te da `git status` es este:

```
## On branch master
nothing to commit (working directory clean)
```

Pero, y esa es una cosa que vas a ver a menudo en git, si hay algo que hacer te informa de las posibles acciones que puedes llevar a cabo dependiendo de las circunstancias actuales diciendo como, por ejemplo, (use "git add <file>..." to update what will be committed)".

Cuando ya has hecho los cambios que consideres necesarios y has puesto en el Index todo lo que quieras poner bajo el control de versiones, llega el momento de "hacer commit" (también se le llama "*confirmar*"). Esto significa mandar al HEAD los cambios que tenemos en el Index, y se hace de este modo:

```
git commit NOMBRE_DEL_ARCHIVO
```

Como te estarás imaginando, aquí también puedes usar comodines del mismo modo que vimos en `git add`. Además, si haces simplemente

```
git commit
```

Esto mandará todos los cambios que tengas en el Index.

AL hacer un `commit` se abre automáticamente el editor de texto que tengas por defecto en el sistema, para que puedas añadir un comentario a los cambios efectuados. Si no añades este comentario, recibirás un error y el commit no se enviará.

```
Puedes cambiar el editor por otro de tu gusto con git config
--global core.editor EDITOR', por ejemplo:git config --global
core.editor vim'
```

Si no quieres que se abra el editor puedes añadir el comentario en el mismo commit del siguiente modo:

```
git commit -m "Comentario al commit donde describo los cambios"
```

Recuerda lo que dijimos antes: si modificas un archivo después de haber hecho `git add`, esos cambios no estarán incluidos en tu `commit` (si quieres incluir la última versión, no tienes más que volver a hacer `git add` antes del `commit`).

Ahora nos puede surgir un problema:

Si sólo podemos confirmar con `commit` de un archivo que hayamos preparado con `add`, y sólo podemos hacer `add` de un archivo que existe en nuestro directorio de trabajo ¿Cómo le decimos a git que elimine un archivo del repositorio? Para ello tenemos la orden:

```
git add -u
```

que agregará al Index la información de los archivos que deben ser borrados.

Muy similar a la anterior, `git add -A` sirve para hacer `git add -u` (preparar los archivos eliminados) y `git add .` (preparar todos los archivos modificados) en una sola orden.

Un efecto parecido se puede conseguir con

```
git commit -a
```

Esta orden sirve para confirmar todos los cambios que haya en el directorio de trabajo *aunque no hayan sido preparados* (es decir, aunque no hayas hecho `add`). Esto incluye tanto los ficheros modificados como los eliminados, con lo que sería equivalente a hacer un `git add -A` seguido de un `git commit`.

Esta opción ahorra escribir órdenes, pero también te da más oportunidades de meter la pata. En general se recomienda usar por separado `adds` y `commits`, convenientemente salteados de `git status` para comprobar que todo va bien.

Sincronizando repositorios

Como sistema de control de versiones distribuido, una de las principales utilidades de `git` es poder mantener distintos repositorios sincronizados (es decir, que contengan la misma información), exportando e importando cambios.

Para importar (o exportar) cambios de un repositorio remoto se necesita, lógicamente, tener acceso de lectura a ese repositorio (En sentido estricto, ya hemos importado el estado de un repositorio cuando lo clonamos al hacer `git clone`).

Para sincronizar con uno o más repositorios remotos, debemos saber qué repositorios remotos son esos. Para ello tenemos `remote`, que se usa así:

```
git remote
```

Y, seguramente, te retornará algo parecido a

```
origin
```

Lo que nos dice que el repositorio es el que le indicamos como “origin” al hacer el `clone` y, la verdad, no es mucha información.

Para obtener algo más útil, prueba a hacerlo con el parámetro `-v` de este modo:

```
git remote -v
```

lo que te retornará algo parecido a esto:

```
origin https://github.com/oslugr/repo-ejemplo.git (fetch)
origin https://github.com/oslugr/repo-ejemplo.git (push)
```

Esto te dice que hay un repositorio llamado “origin” que se usará tanto para recibir (fetch) como para enviar (push) los cambios. “origin” es el nombre del

repositorio remoto por defecto, pero puedes tener muchos más y sincronizar con todos ellos.

Para añadir otro repositorio remoto se hace con la misma instrucción `remote` de este modo:

```
git remote add ALIAS_DEL_REPOSITORIO DIRECCION_DEL_REPOSITORIO
```

Donde `ALIAS_DEL_REPOSITORIO` es un nombre corto para usar en las instrucciones de git (el equivalente al “origin” que hemos visto) y `DIRECCION_DEL_REPOSITORIO` la dirección donde se encuentra. Por ejemplo:

```
git remote add personal git://github.com/psicobyte/repo-ejemplo.git
```

Esto añade un repositorio remoto llamado “personal” con la dirección que se indica.

Si ahora hacemos un `git remote -v`, veremos algo como:

```
mio git://github.com/psicobyte/repo-ejemplo.git (fetch)
mio git://github.com/psicobyte/repo-ejemplo.git (push)
origin https://github.com/oslugr/repo-ejemplo.git (fetch)
origin https://github.com/oslugr/repo-ejemplo.git (push)
```

Para eliminar un repositorio tienes:

```
git remote rm NOMBRE
```

Y para cambiarle el nombre:

```
git remote rename NOMBRE_ANTERIOR NOMBRE_ACTUAL
```

Nota que git no comprueba si realmente existen los repositorios que agregas o si tienes permisos de lectura o escritura en ellos, de forma que el hecho de que estén ahí no significa que vayas a poder usarlos realmente.

Recibiendo cambios Ha llegado el momento de importar cambios desde un repositorio remoto. Para ello tenemos `git pull` que se usa así:

```
git pull REPOSITORIO_REMOTO RAMA
```

el `REPOSITORIO_REMOTO` es uno de los nombres de repositorio que hemos visto antes (si no pones ninguno, se supone “origin”). Sobre las ramas se hablará un poco más adelante, pero baste decir que, si no ponemos ninguna, se supone que es la rama “master”)

de este modo, la forma más usual de llamar esta orden es, simplemente:

```
git pull
```

(que significaría lo mismo que `git pull origin master`)

Esta instrucción trae del repositorio remoto indicado (o de “origin” si no indicas nada, como hemos visto), todos los cambios que haya respecto al tuyo (lógicamente, no se molesta en traer los que son iguales).

Si el repositorio del que tratas de importar no existe o no tienes permiso de lectura, te dará un mensaje de error advirtiéndote de ello.

Si hay archivos que tú has modificado pero el otro repositorio no, te quedarás con los tuyos. Cuando se trate de archivos que tú no has cambiado pero que sí son distintos en el remoto, actualizarás los tuyos a este último. Pero, si importas archivos que se han modificado en ambos repositorios ¿qué pasa con las diferencias? ¿Sobrescribirá tus archivos? ¿Perderás los del otro repositorio?

Ahí es donde entra la solución de problemas, y lo veremos dentro de poco.

En realidad, `git pull` es la unión de dos herramientas distintas, que son `git fetch`, que trae los cambios remotos creando una nueva rama, y `git merge`, que une esos cambios con los tuyos. En ocasiones te convendrá más usarlas por separado pero, como aún no hemos visto el manejo de las ramas, dejaremos esto por ahora.

Enviando cambios Si con `pull` importamos cambios desde otro repositorio, la instrucción `push` es la que nos permite enviar cambios a un repositorio remoto.

Se usa de un modo bastante parecido:

```
git push REPOSITORIO_REMOTO RAMA
```

Igual que hemos visto con `git pull`, los valores por defecto son “origin” para el repositorio y “master” para la rama, con lo que se puede poner simplemente:

```
git push
```

Lo que enviará nuestros cambios al servidor remoto.

Salvo que algo haya cambiado allí.

Si la versión que hay en el servidor es posterior a la última que sincronizamos (es decir, alguien más ha cambiado algo), git mostrará un error y no nos dejará hacer el push. Antes debemos hacer un pull.

Sólo cuando hayamos hecho el pull (y resuelto los conflictos, si es que hubiera alguno), nos dejará hacer el push y enviar nuestra versión.

Al hacer tu push, git te retornará información de los cambios realizados, número de archivos, etc.

Contraseñas Naturalmente, como ya hemos comentado, no puedes hacer push a un repositorio en el que no tengas permiso de escritura. Para eso puede ser que sea un repositorio abierto a todo el que conozca la dirección, pero eso

sería muy raro (e inseguro). Lo usual es que cuentes con un usuario y contraseña que te permitan acceder (normalmente por [ssh](#)) al servidor.

En otros repositorios (más raros), también necesitarás usuario y contraseña para acceder a la lectura y, por tanto, para hacer pull.

En ambos casos, git te solicitará el nombre de usuario y la contraseña cada vez que hagas push. No tiene por qué ser muy a menudo, pero puede ser un engorro.

En muchos sitios puedes ahorrarte ese trabajo usando pares de claves ssh. Básicamente consiste en que tu ordenador y el del repositorio se reconozcan entre ellos y no tengas que andar identificándote.

Las instrucciones para hacer esto en github están en [esta página de ayuda](#)

Comportamiento por defecto de push

Las versiones anteriores de git tenían un comportamiento por defecto a la hora de hacer push llamado 'matching'.

Este consiste en que, al hacer push, se sincronizan todas las ramas del proyecto con sendas ramas en el servidor con el mismo nombre (ya hablaremos en detalle de las ramas más adelante). Si en el servidor no existe una rama con el nombre de alguna local, se crea automáticamente.

La versión 2 de git cambiará ese comportamiento, que pasará a ser **simple**, lo que significa que se sube sólo la rama que tienes activa en este momento a la rama de la que has hecho el pull, pero te dará un error si el nombre de esa rama es distinto.

Mientras tanto, actualmente, git te avisa (a cada push) de que se va a hacer este cambio y te avisa de que puedes configurar este comportamiento por defecto con un mensaje como este:

```
warning: push.default is unset; its implicit value is changing in
Git 2.0 from 'matching' to 'simple'. To squelch this message
and maintain the current behavior after the default changes, use:
```

```
git config --global push.default matching
```

To squelch this message and adopt the new behavior now, use:

```
git config --global push.default simple
```

When push.default is set to 'matching', git will push local branches to the remote branches that already exist with the same name.

In Git 2.0, Git will default to the more conservative 'simple'

behavior, which only pushes the current branch to the corresponding remote branch that 'git pull' uses to update the current branch.

See 'git help config' and search for 'push.default' for further information. (the 'simple' mode was introduced in Git 1.7.11. Use the similar mode 'current' instead of 'simple' if you sometimes use older versions of Git)

Para elegir el comportamiento que prefieres sólo tienes que usar, como ya hemos visto para otras configuraciones, el comando `git config` de este modo:

```
git config --global push.default OPCION
```

Por ejemplo:

```
git config --global push.default matching
```

Usaría la opción `matching` en todos tus repositorios, pero:

```
git config --local push.default simple
```

Usaría la opción `simple` sólo en el repositorio en el que te encuentras.

Otras opciones posibles son:

- `current`: Sube los cambios de la rama activa a una rama remota del mismo nombre. Si no existe esa rama remota, se crea.
- `nothing`: Esta opción sólo tiene sentido para test, debugs y esas cosas. Al hacer push no se subirá nada a repositorio remoto.
- `upstream`: Al igual que `simple`, sube la rama que tienes activa a la rama de la que has hecho el pull pero, en este caso, *no* te dará error si el nombre de esa rama es distinto.

El archivo `.gitignore`

Cuando hacemos `git add .` o algo parecido, preparamos todos los archivos que hayan sido modificados. Esto es, sin duda, mucho más cómodo que ir añadiendo los archivos uno a uno. Pero muy a menudo hay montones de archivos en tu directorio de trabajo que no quieres que se añadan nunca. Archivos de contraseñas, temporales, borradores, binarios compilados, archivos de configuración local...

Por ejemplo, muchos editores de texto mantienen una copia temporal de los archivos que estás editando, con el mismo nombre pero terminado en el signo `~`. Si haces `git add .`, estos archivos se acabarán añadiendo a tu repositorio, cosa que no tiene demasiada utilidad.

Para evitar este problema y facilitarte el trabajo, git nos permite crear un archivo (varios, en realidad, como veremos enseguida) donde describir qué archivos quieres ignorar.

El archivo en cuestión debe llamarse “*.gitignore*” (empezando por un punto) y ubicarse en el directorio raíz de tu proyecto.

En este archivo podemos incluir los nombres de archivos que queramos ignorar. Por ejemplo, imaginemos que nuestro *.gitignore* tiene este (poco útil) contenido:

```
## Los archivos que se llamen "passwords.txt" serán ignorados
passwords.txt
```

Las líneas de *.gitignore* que comienzan con el signo “#” son comentarios (útiles para quién lo lea), y git las ignora.

Gracias a esto, git ignorará cualquier archivo que se llame passwords.txt, y no los incluirá en tus adds.

Esto es demasiado simple y no nos va a ser muy útil pero, afortunadamente, *.gitignore* permite comodines y otras herramientas útiles. Por ejemplo:

```
## Ignoramos todos los archivos que terminen en "~"
*~

## Ignoramos todos los archivos que terminen en ".temp"
*.temp

## Ignoramos todos los archivos que se llamen
## "passwords.txt", "passwords.c", "passwords.csv"...
passwords.*
```

El archivo *.gitignore* permite hacer cosas mucho más complejas, aunque para la mayoría de los casos con algo como lo visto arriba es suficiente.

Pese a que cada repositorio puede tener su propio *.gitignore*, puede ser útil tener además un archivo general para todos los repositorios.

Git busca por defecto este archivo general en el directorio “*.config/git/ignore*” de tu directorio “Home”, pero esto puede cambiarse con la siguiente orden:

```
git config --global core.excludesfile RUTA_AL_ARCHIVO_IGNORE
```

Por ejemplo, para usar un archivo llamado “ignorar” en mi directorio personal, pondría algo así:

```
git config --global core.excludesfile ~/ignorar
```

El símbolo “~” en un path significa “El directorio Home del usuario”

Puedes encontrar muchos ejemplos de archivos *.gitignore* en este [repositorio de GitHub](#).

Las opciones que se establecen con `git config` para el repositorio local se almacenan permanentemente en el fichero `.git/config`. Por lo pronto no nos preocupemos de este fichero, pero todas las variables anteriores (y alguna más) se pueden poner directamente en este fichero.

Solución de problemas con git

Objetivos

En todo desarrollo colaborativo aparecen problemas. No es difícil solucionarlos.

- Saber obtener información sobre git
- Conocer el estado e historial de nuestros proyectos
- Modificar y recuperar estados anteriores
- Solucionar conflictos entre repositorios

Obtener Ayuda

Lo primero que necesitamos a la hora de enfrentarnos a las dificultades es conocer nuestras herramientas.

Git dispone de una ayuda detallada que nos resultará muy útil. Para invocarla sólo hay que hacer

```
git help
```

también se puede obtener ayuda de un comando concreto con `git help COMANDO`, por ejemplo:

```
git help commit
```

Viendo el historial

Has hecho una serie de modificaciones seguidas de commits con sus comentarios ¿Cómo puedes ver todo eso? Para ello tienes la instrucción

```
git log
```

La orden `git log` te mostrará un listado de todos los cambios efectuados, con sus respectivos comentarios, empezando desde el más reciente hasta el más antiguo.

El formato de cada commit en la respuesta es como este:

```
commit c2ac7c356156177a50df5b4870c72ce01a88ae63
Author: psicobyte <psicobyte@gmail.com>
Date: Sun Mar 30 11:54:15 2014 +0200
    Cambios menores en las explicaciones de git add y git status
```

La primera línea es un *hash* único que identifica al commit (y que más adelante no será muy útil), seguida del autor, la fecha en que se hizo y el comentario que acompañó al commit.

Por defecto, `git log` nos mostrará todas las entradas del log. para ver sólo un número determinado sólo tienes que añadirle el parámetro `-NUMERO`, donde NUMERO es el número de entradas que quieres ver, por ejemplo:

```
git log -4
```

mostrará las últimas cuatro entradas en el log.

Otra opción posible, si quieres ver una versión más resumida y compacta de los datos, es `--oneline`, que te muestra una versión compacta.

Si, por el contrario, quieres más detalles, la opción `-p` te mostrará, para cada commit, todos los cambios que se realizaron en los archivos (en formato diff).

Otra ayuda visual es `--graph`, que dibuja (con caracteres ASCII) un árbol indicando las ramas del proyecto (ya veremos eso un poco más adelante).

`git log` tiene un montón de opciones más (para filtrar por autor o fecha, mostrar estadísticas...) que, además, se pueden usar en combinación. Por ejemplo, la instrucción

```
git log --graph --oneline
```

Mostrará los commits en versión compacta y dibujando las ramas (cuando las haya), dando una salida parecida a esta:

```
* 6ad05c1 Sólo una cosilla
* 0678363 Resuelve conflicto como ejemplo
|\
| * bf454ef Prueba para crear conflictos. Así mismo.
* | 8785174 Añade título nueva sección
|/
* afee5ab Acaba un-solo-usuario y listo para conflictos
* fd04eff Corregido (más o menos) el markdown
```

Para más detalles, recuerda que `git help log` es tu amigo.

Borrado de archivos

En git se pueden borrar archivos con la orden `git rm`.

```
git rm NOMBRE_DEL_FICHERO
```

Funciona como la propia orden del sistema operativo, con la salvedad de que *tambien* borra el archivo del Index, si estuviera allí. Esto lo hace muy útil en ocasiones.

Si necesitas borrar el archivo del Index pero sin borrarlo de directorio de trabajo (porque, por ejemplo, te has arrepentido y no quieres incluirlo en el próximo commit), se puede hacer con la opción `--cached` del siguiente modo:

```
git rm --cached NOMBRE_DEL_FICHERO
```

Otra opción para hacer esto mismo es con `git reset HEAD` que se usa del siguiente modo:

```
git reset HEAD NOMBRE_DEL_ARCHIVO
```

Rehacer un commit

Puedes rehacer el último commit usando la opción `--amend` de este modo:

```
git commit --amend
```

Si no has modificado nada en tus archivos, esto simplemente te permitirá reescribir el comentario del commit pero, si por ejemplo habías olvidado añadir algo al Index, puedes hacerlo antes del `git commit --amend` y se aplicará en el commit.

Deshacer cambios en un archivo

Has cambiado un archivo en tu directorio de trabajo, pero te arrepientes y quieres recuperar la versión del HEAD (la del último commit). Nada más fácil que:

```
git checkout -- NOMBRE_DEL_ARCHIVO
```

Resolviendo conflictos

Normalmente los conflictos suceden cuando dos usuarios han modificado la misma línea, o bien cuando han modificado un fichero binario; por eso los ficheros binarios **no** deben estar en un repositorio. Te aparecerá un conflicto de esta forma cuando vayas a hacer `push`

```
To git@github.com:oslugr/curso-git.git
! [rejected]          master -> master (non-fast-forward)
error: failed to push some refs to 'git@github.com:oslugr/curso-git.git'
consejo: Updates were rejected because the tip of your current branch is behind
```

consejo: its remote counterpart. Merge the remote changes (e.g. 'git pull')
consejo: before pushing again.
consejo: See the 'Note about fast-forwards' in 'git push --help' for details.

El error indica que la *punta* de tu rama está detrás de la rama remota (es decir, que hay modificaciones posteriores a tu última sincronización). Rechaza por lo tanto el **push**, pero vamos a hacer **pull** para ver qué es lo que ha fallado

```
remote: Counting objects: 7, done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 4 (delta 3), reused 0 (delta 0)
Unpacking objects: 100% (4/4), done.
De github.com:oslugr/curso-git
   afee5ab..bf454ef  master    -> origin/master
Auto-merging texto/mas-usos.md
CONFLICTO(contenido): conflicto de fusión en texto/mas-usos.md
Automatic merge failed; fix conflicts and then commit the result.
```

El conflicto de fusión te indica que no hay forma de combinar los dos ficheros porque hay cambios en la misma línea. Así que hay que arreglar los conflictos. En general, si se trata de este tipo de conflictos, no es complicado. Al mirar el fichero aparecerá algo así

```
<<<<<< HEAD
## Resolviendo conflictos en `git`
=====
## Vamos a ver cómo se resuelven los problemas en git
>>>>>> bf454eff1b2ea242ea0570389bc75c1ade6b7fa0
```

Lo que uno tiene está entre la primera línea y los signos de igual; lo que hay en la rama remota está a continuación y hasta la cadena que indica el número del commit en el cual está el conflicto. Resolver el conflicto es simplemente borrar las marcas de conflicto (los <<<<< y los >>>>> y los ==) y elegir el código con el que nos quedamos; en este caso, como se ve, es el que aparece efectivamente en este capítulo.

Una vez hecho eso, se puede ya hacer **push** directamente sin ningún problema.

Retrocediendo al pasado

Para recuperar el estado de tu directorio de trabajo tal como estaba en algún momento del pasado, primero necesitas saber qué momento es ese. Eso se consigue con **git log** que, como vimos, nos devuelve (entre otras cosas) un hash que identifica al commit.

Con ese hash ya podemos hacer un `git reset` del siguiente modo:

```
git reset --hard HASH_DEL_COMMIT_A_RECUPERAR
```

Eso, en cierto modo, volverá atrás en el tiempo, deshará los commits posteriores al indicado y traerá a tu directorio de trabajo los archivos tal y como estaban entonces. Todos los cambios posteriores desaparecerán, así que mucho cuidado.

Como consejo, recuerda hacer `push` antes de jugar con `git reset --hard`. De este modo, si quieres recuperar todo el trabajo posterior, no tienes más que hacer `pull` y los recuperarás de nuevo.

También te puedes salvaguardar usando otra rama para hacer el `git reset --hard` sobre ella, pero el uso de ramas es algo que veremos un poco más adelante.

Viendo (y recuperando) archivos antiguos

Puedes ver los cambios que hiciste en un commit si haces

```
git show HASH_DE_UN_COMMIT
```

Esto puede ser muy útil, pero aun hay más. Si haces `git show HASH_DE_UN_COMMIT:ruta/a/un/archivo` te mostrará el estado de ese archivo en aquel commit.

Esto nos va a servir para hacer un pequeño truco:

```
git show HASH_DE_UN_COMMIT:ruta/a/un/archivo > archivo_copia
```

La orden anterior nos permite redireccionar la salida de `git show` a un archivo llamado `archivo_copia`, con lo que obtendremos una copia del archivo tal y como estaba en el commit indicado.

Más usos de git

Objetivos

- Aprender patrones habituales de flujo de trabajo con `git`
- Aprender a trabajar con las ramas.
- Solucionar los conflictos cuando dos desarrolladores trabajan sobre la misma línea.
- Interpretar la historia en sus diferentes formatos

Flujos de desarrollo de software (y quizás de otras cosas)

Un sistema como `git` no es independiente de una organización del trabajo. Aunque a priori puedes trabajar como te dé la gana, el que te facilite el uso de

ciertas herramientas hace que sea más fácil usar una serie de prácticas que son habituales en el desarrollo de software (y de otras cosas, como documentación o novelas) para hacer más productivos a los equipos de trabajo y poder predecir con más precisión el desarrollo de los mismos. Por eso, aunque se puede usar cualquier metodología de desarrollo de software con el mismo, `git` funciona mejor con [metodologías ágiles](#) que tienen ciclos más rápidos de producción y de despliegue de nuevas características o de arreglo de las mismas. Las metodologías ágiles son iterativas y en todas las iteraciones están presentes la mayoría de los actores del desarrollo: clientes, desarrolladores, arquitectos; incluso en algunas puede que esté la peña de márketing, a ver si se enteran de lo que está haciendo el resto de la empresa para venderlo (y no al revés, vender cosas que luego obligan al resto de la empresa a desarrollar).

El desarrollo se divide por tanto en estas fases

1. Trabajo con el código. Modificar ficheros, añadir nuevos.
2. Prueba del código. La mayor parte de las metodologías de desarrollo hoy en día, o todas, incluyen una parte de prueba; en casi todos los casos esta prueba está automatizada e incluye test unitarios (que prueban características específicas), de integración y de cualquier otro tipo (calidad de código, existencia y calidad de la documentación).
3. Lanzamiento del producto. Cuando se han incorporado todas las características que se desean, se lanza el producto. El lanzamiento del producto, en el caso de web, incluye un *despliegue* (*deploy*) del mismo, y en el caso de tratarse de otro tipo de aplicación, de un *empaquetamiento*. Se suele hablar, en todo caso, de *despliegue* (aunque sea porque las aplicaciones web son más comunes hoy en día que las aplicaciones de escritorio).
4. Resolución de errores con el código en producción. Si surge algún error, se trata de resolver sobre la marcha (*hotfix*), por supuesto, incorporándolo al código que se va a usar para desarrollos posteriores.

En casi todas estas fases puede intervenir, y de hecho lo hace, un sistema de control de fuentes como `git`; en muchos casos no se trata de órdenes de `git`, sino de funciones a las que se puede acceder directamente desde sitios de gestión como GitHub.

Organización de un repositorio de `git`

No hay reglas universales para la organización de un repositorio, aunque sí reglas sobre como *no* debe hacerse: todo en un sólo directorio. El repositorio debe estar organizado de forma que cada persona sólo tenga que *ver* los ficheros con los que tenga que trabajar y no se *distriga* con la modificación de ficheros con los cuales, en principio, no tiene nada que ver; también de forma que no se sienta tentado en modificar esos mismos ficheros. Vamos a exponer aquí algunas prácticas comunes, pero en cada caso el sentido común y la práctica habitual de la empresa deberá imponerse...

Qué poner en el directorio principal Cuando se crea un repositorio en GitHub te anima a crear un `README.md`. Es importante que lo hagas, porque va a ser lo que se muestre cuando entres a la página principal del proyecto en GitHub y, además, porque te permite explicar, en pocas palabras, de qué va el proyecto, cómo instalarlo, qué prerequisites tiene, la licencia, y todo lo demás necesario para navegar por él.

Otros ficheros que suelen ir en el directorio principal

- `INSTALL` por costumbre, suele contener las instrucciones para instalar. También por convención, hoy en día se suele escribir usando Markdown convirtiéndose, por tanto, en `INSTALL.md`.
- `.gitignore` posiblemente ya conocido, incluye los patrones y ficheros que no se deben considerar como parte del repositorio
- `LICENSE` incluye la licencia. También se crea automáticamente en caso desde Github en caso de que se haya hecho así. No hay que olvidar que también hay que incluir una cabecera en cada fichero que indique a qué paquete pertenece y cuál es la licencia.
- `TODO` es una ventana abierta a la colaboración, así como una lista para recordarnos a nosotros mismos qué tareas tenemos por delante.
- Otros ficheros de configuración, como `.travis.yml` para el sistema de integración continua Travis, `Makefile.PL` o `configure` u otros ficheros necesarios para configurar la librería, y ficheros similares que haga falta ejecutar o ver al instalar la librería. Se aconseja siempre que tengan los nombres que suelen ser habituales en el lenguaje de programación, si no el usuario no sabrá como usarlos.

En general se debe tratar de evitar cargar demasiados ficheros, fuera de esos, en el directorio principal. Siempre que se pueda, se usará un subdirectorio.

Una estructura habitual con directorio de test Los fuentes del proyecto deben ir en su propio directorio, que habitualmente se va a llamar `src`. Algunos lenguajes te van a pedir que tengan el nombre de la librería, en cuyo caso se usará el que más convenga. Si no se trata de una aplicación sino de una biblioteca, se usará `lib` en vez de `src`, como en esta [biblioteca llamada NodEO](#) Los tests unitarios irán aparte, en un directorio habitualmente llamado `test`. Finalmente, un directorio llamado `examples` o `apps` o `scripts` o `bin` o `exe` incluirá ejemplos de uso de la biblioteca o diferentes programas que puedan servir para entender mejor la aplicación o para ejecutarla directamente.

Estructura jerárquica con submódulos Un repositorio `git` tiene una estructura **plana**, en el sentido que se trata de un solo bloque de ficheros que se trata como tal, a diferencia de otros sistemas de gestión de fuentes centralizados como CVS o Subversion en los que se podía tratar cada subdirectorio como si fuera un proyecto independiente. Pero en algunos casos hace falta trabajar con proyectos en los cuales haya un repositorio que integre el resultado del desarrollo independiente de otros, por ejemplo, una aplicación que se desarrolle conjuntamente con una librería. En ese caso un repositorio `git` se puede dividir en **submódulos**, que son básicamente repositorios independientes pero que están incluidos en una misma estructura de directorios.

Por ejemplo, vamos a incluir el texto de este curso en el repositorio de ejemplo, para poder servirlo como una web también:

```
git submodule add git@github.com:oslugr/curso-git.git curso
Clonar en «curso»...
remote: Reusing existing pack: 14, done.
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 18 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (18/18), 17.26 KiB, done.
Resolving deltas: 100% (4/4), done.
```

Los submódulos no se clonan directamente al clonar el repositorio. Hay que dar dos comandos: `git submodule init` y `git submodule update` dentro del directorio correspondiente; esta última orden servirá para actualizar submódulos también cada vez que haya un cambio en el repo del que dependan (y queramos actualizar nuestra copia); tras ellos habrá que decir `git pull`, como siempre, para traerse los ficheros físicamente.

De esta forma, el repositorio queda (parcialmente) con esta estructura de directorios:

```
jmerelo@penny:~/txt/docencia/repo-tutoriales/repo-ejemplo$ tree
.
├── curso
│   ├── LICENSE
│   ├── README.md
│   └── texto
│       ├── ganchos.md
│       ├── GitHub.md
│       └── mas-usos.md
```

con el subdirectorio `curso` siendo, en realidad, otro repositorio.

Por ejemplo, podíamos tener una estructura que incluyera subdirectorios para `cliente` (un submódulo) y `servidor` (otro submódulo). Con ambos se puede

trabajar de forma independiente y, de hecho, *residen* en repositorios independientes, pero puede que, en caso de empaquetarlos o desplegarlos de alguna forma determinada (por ejemplo, a un PaaS), queramos hacerlo desde un solo repositorio, como en este caso.

Los submódulos pueden ser un ejemplo de flujos de trabajo: en este caso, hay un flujo desde “las fuentes del manantial” (que puede cambiar de forma independiente su proyecto) hasta “la desembocadura” (nuestro proyecto, que lo usa). Dividir un proyecto en módulos y dejar que personas independientes se encarguen de cada uno, integrándolo todo en un submódulo, por tanto, es una forma simple y sin demasiadas complicaciones de hacerlo.

Flujos de trabajo con git

Un *flujo de trabajo* es simplemente una forma de organizar las tareas de programación de forma que se conozca, de antemano, qué tareas van detrás de qué tareas y cuál es el destino, en cada momento, del código que se está haciendo. El tener un flujo de trabajo consistente hace que se eviten conflictos y también que el resultado del trabajo sea más predecible; también a evitar problemas y a identificarlos fácilmente.

El flujo de trabajo básico, de un solo usuario, cuando se trabaja con un sistema de control de fuentes y lo hace un solo usuario es el siguiente:

1. `git pull`
2. Trabajo con el código; añadir nuevos ficheros fuentes con `git add`
3. `git commit -a -m "[implícito: este commit] [hace] [Tal cosa]"`
(o `git -am`, que es lo mismo)
4. `git push`

Fijémonos en el tercer paso, el commit. Primero, conviene hacer siempre `-a`, es decir, `-all` por [varias razones](#):

1. Porque examina todos los ficheros que están siendo seguidos, no sólo los del directorio actual y los que hay por debajo.
2. Porque [hace automáticamente un `git rm` sobre los mismos](#), si es que ha sido borrados.

Lo segundo es decidir cuando se hace el *commit*; lo habitual es que se haga cada vez que se lleve a cabo un cambio significativo, pero ¿qué es un cambio significativo? Pues un cambio más o menos atómico, que incluya todos los ficheros afectados por ese cambio y, lo más importante, que deje, a priori, el código en un estado *sano*. No se debe hacer commit de un código sintácticamente incorrecto, dejarse a medias un cambio o, en general, antes de acabar lo que quiera que uno se propusiera hacer al empezar la sesión de trabajo. No es

imprescindible hacer un **push** por cada commit, pero tampoco estorba hacerlo, sobre todo por si, simultáneamente, alguien está modificando el mismo código.

También conviene escribir **-m** para insertar el mensaje directamente sin que salga el editor de textos (que puede que no sea nuestro editor favorito). En el mensaje podemos escribir cualquier cosa, pero hay que tener en cuenta que lo que escribamos es un predicado cuyo sujeto son los cambios que hemos hecho. Sobre qué hay que escribir hay [muchas versiones](#), pero conviene que el mensaje sea informativo, hable de porqués y de cómo más que de qué (no se puede decir “inserta una función”, eso ya se ve en el código, sino por qué se inserta esa función) y se aconseja también un formato similar al siguiente

Hace tal cosa arreglando el error en el issue #666

Siguiendo la sugerencia de @foodev hemos usado el algoritmo Vicentico para resolver ese error y ha quedado monísimo.

Este formato se denomina 50+72 y consiste en usar una primera línea con 50 caracteres (justos, en este caso) que incluya una explicación extendida que esté formateada en líneas de 72 caracteres. Esto es difícil de hacer desde la línea de órdenes, así que tienes dos opciones: usar un *script* que formatee este mensaje, o bien configurar tu editor de forma que use ese formato.

El mensaje de *commit* también admite markdown y de hecho se formateará de esa forma en GitHub (posiblemente también en otros repos) y dentro del mismo admite ciertos atajos como referirse a una tarea o *issue* por el número de la misma o a un desarrollador por el handle. En todo caso, se formatee o no de esa forma, es informativo saber por qué hace lo que hace.

En cuanto al **push** es un evento diferente que el **commit** y hace más cosas, por lo que no debe verse como algo automático tras el mismo. Un **commit** es una unidad mínima de cambio, un **push** envía los contenidos al repo remoto y activa una serie de actividades, como la integración continua, comprobación de código y otra serie de cosas interesantes. Además, hasta que no hacemos **push** no comparamos nuestro código con el que está en **HEAD**, que en ese momento puede estar más adelante o más atrás. Si tienes miedo de provocar un conflicto o de que te lo provoque, no hagas un pull hasta estar seguro de que el código no rompe los tests automáticos y hasta que estés seguro de poder resolver los conflictos que ocurran. ¿Cómo se pueden resolver estos conflictos? Lo veremos a continuación.

Igual que en el caso de los submódulos, no deja de ser simplemente una rutina de trabajo más que un flujo de trabajo si trabaja uno solo. Veremos cómo trabajar en diferentes ramas evitando conflictos.

Ramas

Las *ramas* son una característica de todos los sistemas de control de fuentes. A todos los efectos, una rama es un proyecto diferente que *surge* de un proyecto principal, aunque nada obliga a que contengan nada en común (por ejemplo, un repositorio puede incluir el código y las páginas web como una rama totalmente diferente). Sin embargo, las ramas, que más bien deberían llamarse *ramificaciones* o *caminos*, son *caminos divergentes* a partir de un tronco común que, eventualmente, pueden combinarse (aunque no es obligatorio) en uno sólo. En la práctica y como [dicen aquí](#) una rama es un nombre para un *commit* específico y todos los commits que son antecesores del mismo.

En **git** las ramas son también parte natural del desarrollo de un proyecto, dado que se trata de un sistema de control de fuentes distribuido. Cada usuario trabaja en su propia rama, que se *fusiona* con la rama principal en el repositorio compartido cuando se hace *push*. Por eso en alguna ocasión puede suceder que, cuando se hace *pull* o incluso *push*, si se encuentra que las ramas que hay en el repo con el que se fusiona y localmente contienen diferente número de *commits*, aparecerá un mensaje que te indicará que se está fusionando con la rama principal; todo esto, incluso aunque no se hayan creado ramas explícitamente. En realidad, *pull* es combinación de dos operaciones: **fetch** y **merge**, como ya se ha visto en [el capítulo de uso básico](#). De hecho [hay quien dice que no debe usarse nunca pull](#).

Por ejemplo, en caso de que se haya borrado un fichero (o, para el caso, hecho cualquier cambio) en un repositorio y se trate de hacer *push* desde el local, habrá un error de este estilo.

```
To git@github.com:oslugr/repo-ejemplo.git
! [rejected]          master -> master (non-fast-forward)
error: failed to push some refs to 'git@github.com:oslugr/repo-ejemplo.git'
consejo: Updates were rejected because the tip of your current branch is behind
consejo: its remote counterpart. Merge the remote changes (e.g. 'git pull')
consejo: before pushing again.
consejo: See the 'Note about fast-forwards' in 'git push --help' for details.
```

En este caso habrá dos ramas, en la *punta* de cada una de las cuales habrá un commit diferente. Se siguen instrucciones, es decir, **git pull**

```
jmerelo@penny:~/txt/docencia/repo-tutoriales/repo-ejemplo$ git pull
remote: Counting objects: 2, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 2 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (2/2), done.
De github.com:oslugr/repo-ejemplo
61253ec..2fd77db master -> origin/master
```

```

Eliminando Makefile
Merge made by the 'recursive' strategy.
 Makefile | 3 ---
 1 file changed, 3 deletions(-)
 delete mode 100644 Makefile

```

y aparece, efectivamente, el directorio borrado. Habrá que hacer el push de nuevo. Una vez hecho, el repositorio se ha estructurado como se muestra en la imagen:

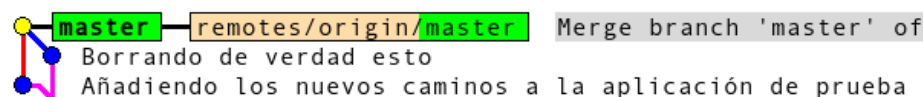


Figure 13: Fusión de dos ramas

Esta imagen, que [se puede ver también en GitHub con fecha 2 de abril](#), y que está obtenida del programa cliente `gitk`, muestra cómo se ha producido la fusión. La que aparece más cerca de la fusión es la que se hizo inicialmente, borrando el fichero, y la más alejada, que aparece más a la izquierda, es la hecha a continuación. El último *commit* fusiona las dos ramas y crea una sola dentro de la rama principal.

Por eso hablamos de enramamiento (bueno, debería ser ramificación, pero esto suena mejor) *natural* en `git`, porque se produce simplemente por que haya dos *commits* divergentes que procedan de la misma rama. Sin embargo, se pueden usar ramificaciones adrede y es lo que veremos a continuación.

Ramas ligeras: etiquetas Una *etiqueta* permite *guardar* el estado del repositorio en un momento determinado, siendo como una especie de *foto* del estado el proyecto. Se suele asociar a hitos en la historia del mismo: entrada en producción, despliegue de los resultados, o versión mayor o menor.

Para *etiquetar* se usa la orden `tag`

```
git tag v0.0.2
```

`tag` etiqueta el último *commit*, es decir, asigna una etiqueta al estado en el que estaba el repositorio tras el último *commit*. La etiqueta aparecerá de forma inmediata (sin necesidad de hacer *push*, puesto que se añade al último *commit* y se puede listar con

```

git tag
jmerelo@penny:~/txt/docencia/repo-tutoriales/repo-ejemplo$ git tag
v0.0.1
v0.0.2

```

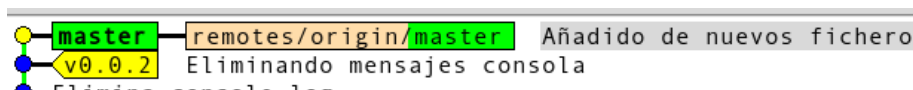


Figure 14: La etiqueta se muestra adosada a un commit

Como se ve, una etiqueta es, en realidad, un apodo para un commit determinado cuyo *hash* puede ser difícil de recordar. Pero por esa misma razón se pueden usar como salvaguarda del estado del repositorio en un momento determinado que, más adelante, se puede recuperar o fusionar con una rama.

Las ramas se pueden también anotar, lo que añade una explicación adicional a lo que ya esté almacenado en el commit correspondiente.

```
git tag -a v0.0.2.1
```

abrirá un editor (el que tengamos especificado por defecto) para añadir una anotación a esta etiqueta, lo que nos podemos ahorrar si usamos

```
git tag -a v0.0.2.1 -m "Estado estacionario del repositorio"
```

por ejemplo. Esta información aparecerá añadida al commit correspondiente (el último que hayamos hecho) cuando hagamos, por ejemplo, `git show v0.0.2.1`

```
tag v0.0.2.1
Tagger: JJ Merelo <jjmerelo@gmail.com>
Date:   Sun Apr 6 09:58:12 2014 +0200
Poco antes de pasar a producción el tema de tags
en realidad, sólo es un ejemplo
commit b958b16b8261fa3ca8159b3ae45e237ae1fa1dce
Author: JJ Merelo <jjmerelo@gmail.com>
Date:   Sun Apr 6 09:45:38 2014 +0200
    Añadido de nuevos ficheros al servidor
    Y edición del README para que sirva para algo
```

(Suprimidos espacios en blanco para que aparezca como un sólo mensaje). Que, como se ve, añade un pequeño mensaje (al principio) al propio del commit (a continuación).

Finalmente, `git describe` es una orden creada precisamente para trabajar con las etiquetas: te indica el camino que va desde la última etiqueta al commit actual o al que se le indique

```
git describe
v0.0.2.1-1-g6dd7a8c
```

que, de una forma un tanto críptica, indica que a partir de la etiqueta `v0.0.2.1` hay un commit `-1-` y el nombre del último objeto, en este caso el único `6dd7a8c`. Es otra forma de [etiquetar un punto en la historia de una rama](#), o simplemente otra forma de llamar a un commit. Es más descriptivo que simplemente el hash de un commit en el sentido que te indica de qué etiqueta has partido y lo lejos que estás de ella.

Por eso precisamente conviene, como una buena práctica, etiquetar la rama principal con estas *ramas ligeras* cuando suceda un hito importante en el desarrollo. Y también conviene recordar que, dado que son anotaciones locales, [hay que hacer explícitamente `git push --tags`](#) para que se comuniquen al repositorio remoto.

Creando y fusionando ramas Ya que hemos visto como se crean ramas de forma implícita y de forma *ligera* (con etiquetas), vamos a trabajar explícitamente con ramas. La [forma más rápida de crear una rama](#) es usar

```
git checkout -b get-dir
Switched to a new branch 'get-dir'
```

Esta orden hace dos cosas: crea la rama, copia todos los ficheros en la rama en la que estemos (que será la `master` si no hemos hecho nada) a la nueva rama y te cambia a la misma; a partir de ese momento estarás modificando ficheros en la nueva rama. Es decir, equivale a dos órdenes

```
git branch get-dir
git checkout get-dir
```

En esta rama se puede hacer lo que se desee: modificar ficheros, borrarlos, añadirlos o hacer algo totalmente diferente.

Un cambio de rama sobrescribirá los cambios que se hayan hecho a los ficheros sin hacer *commit*. Si existen tales cambios te avisará, pero puede que no quieras usar el *commit* para comprometer cambios y dejar el repositorio en un estado incorrecto; en ese caso se puede usar simplemente [git stash](#) que almacena los cambios en un fichero temporal que se puede recuperar más adelante usando `git stash apply --index`.

En todo momento

```
git status
```

```
# En la rama get-dir
```


nos dirá en qué rama estamos; los ficheros que físicamente encontraremos en el directorio de trabajo serán los correspondientes a esa rama. Conviene hacer siempre `git status` al principio de una sesión para saber dónde se encuentra uno para evitar cambios y sobre todo pulls sobre ramas no deseadas.

La rama que se ha creado sigue siendo rama local. Para crear esa rama en el repositorio remoto y a la vez sincronizar los dos repositorios haremos

```
git push --set-upstream origin get-dir
```

donde `get-dir` es el nombre de la rama que hemos creado. Esta orden establece un origen por defecto (*upstream*) para la rama en la que estamos y además le asigna un nombre a esa rama, `get-dir`.

Las ramas de trabajo se pueden listar con

```
git branch
* get-dir
master
```

con un asterisco diciéndonos en qué rama concreta estamos; si queremos ver todas las que tenemos se usa

```
git branch --all
* get-dir
master
remotes/heroku/master
remotes/origin/HEAD -> origin/master
remotes/origin/get-dir
remotes/origin/master
```

que, una vez más, nos muestra con un asterisco que estamos trabajando en la rama local `get-dir`; a la vez, nos muestra todas las ramas remotas que hay definidas y la relación que hay con las locales, pero más que nada por nombre. Si queremos ver la relación real entre ellas y los commits que hay en cada una

```
jmerelo@penny:~/txt/docencia/repo-tutoriales/repo-ejemplo$ git branch -vv
* get-dir 389b383 [origin/get-dir] Pasado a glob
master 1a93e3d [origin/master] Añade palabros al diccionario
```

con `-vv` indicando doble verbosidad, es decir, que imprima toda la información que tenga.

En este ejemplo se ha mostrado un patrón habitual de uso de las ramas: para probar nuevas características que no sabes si van a funcionar o no y que, cuando

funcionen, se pasan a la rama principal. En este caso se trataba de trabajar con *todos* los ficheros del directorio en vez de los ficheros que le pasemos explícitamente. Estas ramas se suelen denominar *ramas de características o feature branches* y forman parte de un flujo de trabajo habitual en git. Sobre un repositorio central se creará una rama si quieres probar algo que no sabes si estará bien eventualmente o si realmente será útil. De esta forma no se *estorba* a la rama principal, que puede estar desarrollando o arreglando errores por otro lado. En este flujo de trabajo, eventualmente se integra la rama desarrollada en la principal, para lo que se usa pull de nuevo. El concepto de pull es usar primero **fetch** (descargarse los cambios al árbol) y posteriormente **merge** (incorporar los cambios del árbol al índice). En casos complicados esta división te da flexibilidad para escoger qué cambios quieres hacer, pero en un flujo de trabajo como este se puede usar simplemente. Supongamos, por ejemplo, que estamos en la rama **get-dir** y se han hecho cambios en la rama principal.

```
jmerelo@penny:~/txt/docencia/repo-tutoriales/repo-ejemplo$ git pull origin master
De github.com:oslugr/repo-ejemplo
* branch          master      -> FETCH_HEAD
Merge made by the 'recursive' strategy.
 .aspell.es.pws   | 3 +++
 README.md        | 5 +++--
2 files changed, 6 insertions(+), 2 deletions(-)
```

Este mensaje te muestra que se ha fusionado usando una estrategia determinada. git examina los commits que diferencian una rama de la otra y te los aplica; al hacer **pull** aparecerá el editor, en el que pondremos el mensaje de fusión. Los cambios se propagarán a la rama remota haciendo **git push** y las ramas quedarán como aparece en [la visualización de la red con fecha 6 de abril de 2014](#); **master** se ha fusionado con **get-dir**.

También podemos hacer la operación inversa. Visto que los cambios de **master** no afectan a la funcionalidad nueva que hemos creado, fusionemos la rama **get-dir** en la principal. Cambiamos primero a ésta

```
git checkout master
```

git checkout *saca* del árbol los ficheros correspondientes (lo que puede afectar a los editores y a las fechas de los mismos, que mostrarán la del último checkout si no se han modificado) y nos deposita en la rama principal, desde la cual podemos fusionar, usando también pull

```
jmerelo@penny:~/txt/docencia/repo-tutoriales/repo-ejemplo$ git pull origin get-dir
De github.com:oslugr/repo-ejemplo
* branch          get-dir      -> FETCH_HEAD
Updating df46a37..3705af0
```

```
Fast-forward
package.json | 5 +++--
web.js       | 18 ++++++++-----
2 files changed, 15 insertions(+), 8 deletions(-)
```

que, dado que no hemos hecho ningún cambio en el mismo fichero, fusiona sin más problema la rama. En caso de que se hubiera modificado las mismas líneas, es decir, que los *commits* hubieran creado una divergencia, se habría provocado un conflicto que se puede solucionar como se ha visto en el apartado correspondiente. Pero, dado que no la ha habido, el resultado final será el que se muestra en el gráfico.

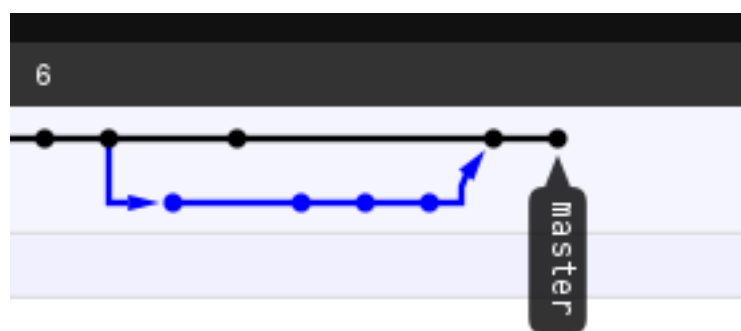


Figure 15: Volviendo al redil del master

La rama, una vez fusionada con el tronco principal, se puede considerar una rama muerta, así que nos la cargamos

```
jmerelo@penny:~/txt/docencia/repo-tutoriales/repo-ejemplo$ git branch -d get-dir
Deleted branch get-dir (was 3705af0).
```

Pero eso borra solamente la rama local. Para [borrarla remotamente](#):

```
jmerelo@penny:~/txt/docencia/repo-tutoriales/repo-ejemplo$ git push origin :get-dir
To git@github.com:oslugr/repo-ejemplo.git
- [deleted]          get-dir
```

Una sintaxis con `:` que es ciertamente poco lógica, pero efectiva. Con eso tenemos la rama borrada tanto local como remotamente.

Los misterios del rebase

`git` tiene múltiples formas de reescribir la historia, como si de un régimen totalitario se tratara. Una de las más simples es *aplanar* la historia como si todos los *commits* hubieran sucedido unos detrás de otros, en vez de en múltiples ramas como es la forma habitual de trabajar (en un flujo de trabajo *rama por característica* como hemos visto anteriormente). Por ejemplo, podemos crear una rama `img-dir` (sobre el repositorio de ejemplo, añade a la aplicación de forma que se pueda trabajar con las imágenes del tutorial) dejando el repo en el estado que se muestra a continuación.

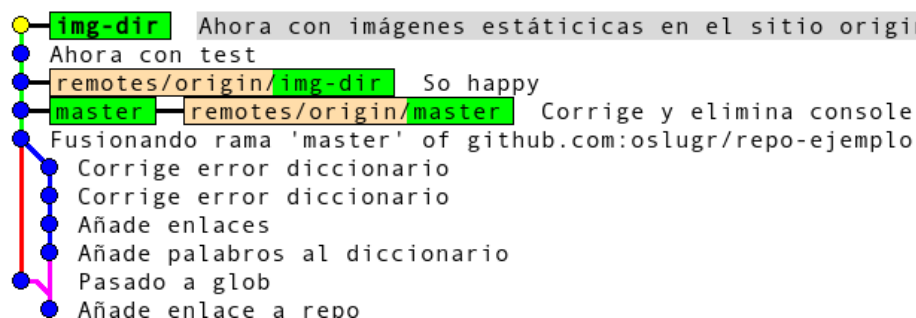


Figure 16: Antes del rebase

tomada desde la propia rama, donde hay una rama `img-dir` con un par de *commits* a partir del máster (dos puntitos azules más abajo).

Una vez acabado el trabajo con la rama, cambiamos a `master` (`git checkout master`) y podemos hacer `rebase`

```
git checkout master
Switched to branch 'master'
jmerelo@penny:~/txt/docencia/repo-tutoriales/repo-ejemplo$ git rebase img-dir
First, rewinding head to replay your work on top of it...
Fast-forwarded master to img-dir.
```

Dejando el repositorio en el estado siguiente

El último commit es ahora parte de la rama `master`. No sólo se han fusionado los cambios en la rama principal, como se ve más abajo en la misma imagen e hicimos con la rama creada anteriormente, `get-dir`. En este caso, y a todos los efectos, se ha *reescrito la historia*, pasando los commits hechos sobre la rama anterior a formar parte de la rama principal. Una vez hecho esto, se limpia eliminando la rama creada. Sin embargo, un rebase no elimina una rama, que sigue ahí, sólo que en una parte diferente del árbol como se muestra a continuación

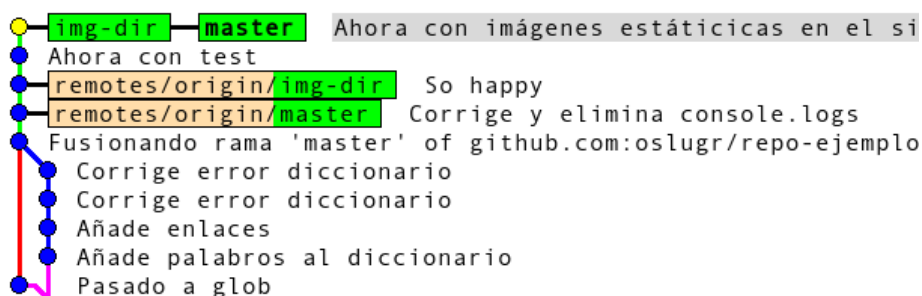


Figure 17: Después del rebase

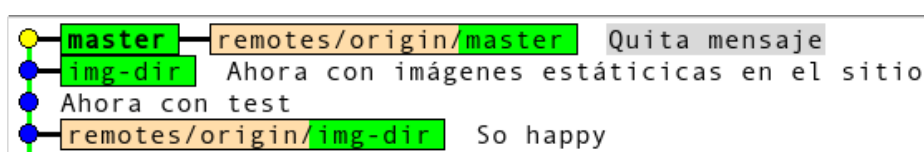


Figure 18: Despegando master de la rama

Sin embargo, ahora la rama es poco menos que un *tag* como el que hemos visto antiguamente. No estorba así que no hace falta borrarla.

Quién hizo qué

Con todas estas ramificaciones es posible que, en un momento determinado, sea difícil saber quién ha hecho qué cambio. Esto puede ser importante no sólo para repartir las culpas cuando algo falle, sino también para ver quién se responsabiliza de cada rama o característica y, eventualmente, también para asignar méritos. La herramienta `gitk` que hemos usado hasta ahora te presenta en forma de árboles los cambios que se han venido haciendo en el repositorio, con un panel a la derecha que muestra quién ha hecho cada commit:

En esta imagen se ve como cada commit está asignado a uno de los autores de este tutorial, junto con los mensajes correspondientes. Con `git log --pretty=short` se puede conseguir un efecto similar en la línea de órdenes:

```
commit 3b89bd2fffbf7f5988de16b9911b14d70c9197bd
Author: JJ Merelo <jjmerelo@gmail.com>
Añadido texto sobre rebase
commit c09756d4d296fbacd9541d2d7c23e7710a5d1f09
Author: JJ Merelo <jjmerelo@gmail.com>
Añadiendo el capítulo de ramas y tags y esas cosas
commit 8e4559325032fe1425288c4d1ab51fb7072f79b1
Author: psicobyte <psicobyte@gmail.com>
```

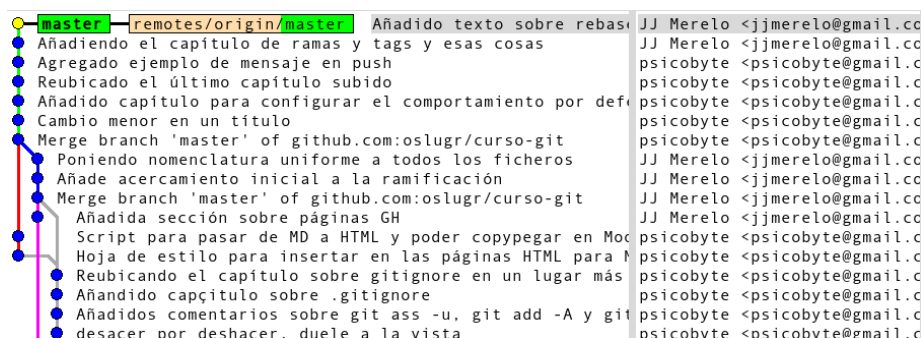


Figure 19: Quien ha hecho qué

Agregado ejemplo de mensaje en push

(una vez más, sin líneas en blanco), con una muestra del mensaje corto y del commit junto con el autor. `log` es muy flexible y permite poner cualquier tipo de formato, pero hay todavía más herramientas. `git blame` permite hacer lo mismo sobre un fichero, viendo quién ha modificado cada una de las líneas. Por ejemplo, `git blame uso_basico.md` devolvería, entre otras cosas, estas líneas

```
6017d70c (Manu      2014-03-28 22:30:40 +0100 70) [GUI Mac](http://mac.github.c
2feb1052 (psicobyte 2014-03-10 03:35:49 +0100 71)
6017d70c (Manu      2014-03-28 22:30:40 +0100 72) [GUI Windows](http://windows.
2feb1052 (psicobyte 2014-03-10 03:35:49 +0100 73)
01b9da5a (Manu      2014-03-28 22:36:46 +0100 74) [GUI for Linux, Windows y Mac
75b8e467 (Manu      2014-03-28 22:29:29 +0100 75)
2feb1052 (psicobyte 2014-03-10 03:35:49 +0100 76) ##Empezando a usar git
```

que muestran que la línea 70 y la 71 han sido modificadas por [Manu](#) mientras que el resto lo han sido por [Pablo](#). El formato que siguen es el hash del commit seguidos por el nombre del usuario, la fecha, el número de línea; finalmente está el contenido de la línea. Algo un poco más vistoso se puede ver en algunos repositorios como GitHub, pulsando sobre el botón *Blame* que aparece en cada uno de los ficheros

Esta es simplemente una visualización del comando anterior, que presenta además un enlace al usuario en GH en caso de serlo (porque, recordemos, git es un DVCS cuyos cambios pueden haberse fusionado en local por parte de cualquier tipo de usuario, que no tiene por qué estar necesariamente en GitHub). Con `blame` se puede saber [incluso quien modificó una línea en particular](#). pero, para un uso básico, basta lo anterior.

2feb1052 » psicobyte 2014-03-10 Segundo módulo, borrador	61	###En Mac
01b9da5a » Makova 2014-03-28 Update uso_basico.md	62	Hay dos maneras de instalar Git en Mac, la más fácil es utilizar el instalado
75b9e467 » Makova 2014-03-28 Añadir Guis SO y Git para Mac	63	
6738e67d » Makova 2014-03-28 Git Mac	64	[Git for OS X](https://code.google.com/p/git-osx-installer/)
2feb1052 » psicobyte 2014-03-10 Segundo módulo, borrador	65	
	66	##Clientes GUI para Linux, Windows y Mac
	67	

Figure 20: Visualización de culpabilidades en GitHub

Usando git como los profesionales: GitHub

Objetivos

- Conocer las características generales de los repositorios públicos de GitHub
- Conocer las características específicas de GitHub
- Aprovechar esas características específicas en el trabajo de desarrollo.

Por qué GitHub

GitHub se ha convertido en el sitio más popular, a pesar de encontrarse entre una cantidad de sitios que alojan también proyectos y permiten usar Git como Gitorious, BitBucket o incluso el venerable [SourceForge](#); [Gitorious](#) tiene la ventaja de que está a su vez basado en software libre, por lo que te puedes instalar tu propia copia del repositorio bajo tu control. Sea con este software o con [Git-Lab](#) te puedes hacer tu propia instalación de git si tienes disponible un servidor para ello. Evidentemente, para trabajar con grandes proyectos privados son una buena opción y lo más aceptable.

Sin embargo, hay buenas razones para usar GitHub. Aparte del uso de git, GitHub en realidad funciona como una comunidad de programadores que te permite interaccionar muy fácilmente con otros programadores que usen las mismas herramientas que uno. Ninguno de los otros repositorios tiene esas características; aunque todos tienen ciertas capacidades *sociales*, lo cierto es que la mayoría de los programadores de software libre usan hoy GitHub. De hecho, GitHub con sus métricas, características de *gamificación* (poner estrellas a proyectos, por ejemplo), continuo desarrollo, enganche a plataformas de programación diversas y cliente móvil es hoy en día la mejor opción para desarrollar software libre (y novelas y [cursos de Git](#)).

Adicionalmente, GitHub se usa como repositorio por defecto para módulos de diferentes lenguajes de programación; por ejemplo, [npm](#), el gestor de módulos de

Node, lo usa para alojar y descargar las fuentes; otros marcos como Joomla lo usan también para definir sus plugins. Incluso GitHub se incluye ya en flujos de trabajo de ciertos entornos, como [la \(futura o pasada\) revista científica Push](#).

En resumen, GitHub es la primera, segunda y tercera mejor opción de un programador de software libre a la hora de alojar sus proyectos. La cuarta sería gitorious, por el hecho de que efectivamente todo el software que usa está liberado y la quinta Bitbucket. Más atrás estarían el resto. En este curso veremos principalmente, por esa razón, GitHub y sus características específicas, algunas de las cuales se han venido usando ya el resto del curso.

Repositorios públicos y privados

Por omisión, los repositorios de GitHub son públicos. Para conseguir un repositorio privado hay que hacer una de dos cosas

- Pagar. Por 7 dólares al mes puedes tener [cinco repositorios privados](#).
- Conseguir [una cuenta para educación](#) que permite, generalmente de forma limitada, tener un número determinado de cuentas gratuitas.

En realidad, las cuentas privadas son útiles para empresas que quieran desarrollar sus propias aplicaciones en privado; para software libre, aplicaciones que se creen para un proyecto las cuentas tienen espacio de almacenamiento ilimitado y un número ilimitado de colaboradores, por lo que no hace falta adquirir una cuenta ni tener un repositorio privado.

El GitHub *social*

En realidad, GitHub es una red social de gente que hace cosas y escribe texto como [este](#) o aplicaciones en diferentes lenguajes de programación. Cada usuario tiene [un perfil](#) y en él puede contar cosas como dónde se encuentra, su web y poco más. Lo que importa es lo que se hace, que aparece justo al lado del logo. GitHub usa [Gravatar](#) para las fotos de perfil, por lo que tendrás que darte de alta en ese servicio *con la misma dirección de email que uses en GH* para que aparezca tu avatar junto a tus contribuciones. Se puede seguir la [actividad](#) de un usuario, pero también se puede ir un paso más allá y pulsar el botón de *Follow* con lo que, al entrar en la [página principal de GitHub](#) se te mostrará, junto con la actividad propia, la de esta persona. Una persona puede ser también añadida a un repositorio, lo que le dará privilegios para realizar todo tipo de acciones sobre él. Conviene usar esto con moderación y sólo cuando se trate de una persona ya involucrada en el proyecto.

El componente *social* también se ve en repositorios específicos: los repositorios, como [este de ejemplo del curso](#), se pueden *vigilar* (*Watch*) y también poner una

especie de “Me gusta”, *Star*, uno al lado del otro. Finalmente, se puede hacer un *Fork* del repositorio, lo que copia el contenido completo del repositorio al propio y permite trabajar con él; equivale a una *rama* tal como la que hemos visto anteriormente. Este *fork* respeta la autoría original que sigue apareciendo en todos los *commits* que se hayan hecho originalmente, pero permite añadir uno sus propias modificaciones *sin que el autor original tenga que aprobarlas*. Los *pull requests* permiten colaboración esporádica, ya que las modificaciones que se soliciten pueden aprobarse o no por parte del autor principal del repositorio; la persona que las haga, sin embargo, no tiene por qué estar añadida como colaborador permanente al mismo.

GitHub también permite comentar a diferentes niveles: se puede comentar un *commit*, se pueden comentar líneas de código y finalmente se pueden hacer solicitudes y comentarios a un repo completo usando *issues* o *solicitudes* (*to have an issue* significa literalmente tener un *asunto* o *problema*). Todas estas formas de interacción permiten tener una vida *social* más o menos rica y que haya muchas formas posibles de interaccionar con los autores de un proyecto y por supuesto también de que esos autores aumenten su *karma* a base de las conexiones, estrellas que reciban sus repositorios y comentarios, así como los *issues* resueltos. Los *issues* se agrupan en *milestones* o hitos, que son simplemente grupos de temas que deben ser resueltos antes de pasar a otra fase del desarrollo. Agrupar los *issues* en hitos permite ver el progreso del mismo, ya que te va mostrando cuál es el grado de terminación de dicho hito. Los hitos, además, pueden fecharse con lo que se puede ver si se ha pasado uno de fecha o no.

Finalmente, los usuarios se pueden agrupar en *organizaciones*. Una organización es en muchos aspectos similar a un usuario; tiene las mismas limitaciones y las mismas ventajas, pero en una organización se definen *equipos* y los permisos para trabajar por repositorios se hacen usando estos *equipos*; cada repositorio puede tener uno o más equipos con diferentes niveles de privilegios y el repositorio en sí tendrá también un equipo que será el que pueda realizar ciertas acciones sobre el mismo. Generalmente se crea un equipo por repositorio, pero puede organizarse de cualquier otra forma.

Cuando uno es añadido a un equipo de una organización, se convierte en otra “personalidad” que te permite, por ejemplo, hacer *fork* como miembro de tal organización. Para adoptar esa personalidad se selecciona el nombre de la organización de un menú que está en el panel de control de [GitHub \(página principal\)](#) justo debajo del logotipo del octocat.

En resumen, la facilidad que tiene GitHub para manejar todo tipo de situaciones de desarrollo y la *gamificación* y *socialización* de la experiencia de desarrollo es lo que ha hecho que hoy en día tenga tanto éxito hasta el punto de que el perfil de uno en GitHub es su mejor carta de presentación a la hora de conseguir un trabajo en desarrollo y programación.

Creando páginas para GitHub pages

Una de las partes interesantes de GitHub y en lo que coincide con otros más clásicos como SourceForge es en la posibilidad de publicar páginas relacionadas con el proyecto o, para el caso, sobre lo que uno quiera. A diferencia de otros sitios, son páginas estáticas (lo que permite, imagino, ser más rápido y eficiente a la hora de servir las).

También se pueden crear muy fácilmente: *Settings*-> se baja a la página donde pone “GitHub Pages”, y se pulsa en *Automatic Page Generator* que te permitirá elegir entre unos pocos (la verdad, no hay muchos) *temas* el que más te guste.

Lo que hace este generador automático es lo siguiente:

- Generar una rama **gh-pages** de tu repositorio principal
- A partir del fichero **README.md** del directorio principal de tu proyecto, genera un fichero **index.html** usando la plantilla seleccionada
- Genera un dominio **usuario.github.io/proyecto** desde el cual se puede acceder a las páginas publicadas

El generador automático sólo funciona una vez. A partir de ese momento, sólo se reflejarán en el sitio general los cambios que se hagan desde la rama **gh-pages**. Se puede trabajar directamente con ella o bien usar algún tipo de **hook** para generar contenido a partir de la rama **master** y copiarlo a esa rama.

Tampoco es obligatorio usar el generador, que está basado en [Jekyll](#), un motor de plantillas y generador estático de páginas basado en Markdown u otros lenguajes simplificados. Jekyll es muy potente y te permite hasta [montar un blog](#), pero no nos vamos a meter en el funcionamiento del mismo. Tampoco es necesario; para crear una página de proyecto sólo hay que hacer dos pasos:

```
git checkout -b gh-pages
touch index.html
git add index.html
```

(Aquí tendría que editarse el fichero HTML y meter algo, y a continuación)

```
git commit -am "Creada página del sitio"
git push origin gh-pages
```

Con esto se transmite la rama al repositorio y automáticamente se publica.

Adicionalmente a las páginas de proyecto, cada organización y cada usuario puede crear también su página. Un usuario **nombredeusuario** tendrá una página **nombredeusuario.github.io** de la que “colgarán” el resto de las páginas (aunque el realidad se tratará de repositorios diferentes, y sólo estarán las

que se hayan generado, claro). Para crear tanto una página de usuario como de organización simplemente se crea el repositorio y se pone el contenido en la rama principal, la **master**. Al hacer push se publica automáticamente, como la de la [organización que se ha creado para este curso](#)

Cómo usar los hooks

Los *hooks* o *ganchos* son eventos que se activan cuando se produce algún tipo de acción por parte de git. En general, se usan para integrar el sistema de gestión de fuentes de git con otra serie de sistemas, principalmente de [integración continua](#) o [entrega continua](#) o, en general, cualquier tipo de sistema de notificaciones o de trabajo en grupo.

GitHub puede integrar cualquier tipo de servicio que acepte una petición REST con una serie de características (esencialmente, datos sobre el repositorio y sobre el último commit), pero tiene ya una serie de servicios, casi un centenar, configurables directamente desde el panel de control yendo a *Settings* -> *Webhooks & Services* -> *Configure services*. Todos los servicios se activan cuando se hace un push a GitHub. Evidentemente, en local no se enteran, salvo que los configuremos explícitamente como vamos a ver en el tema siguiente.

Vamos a dividir los servicios que hay en varios grupos:

- Integración continua. Servicios como TravisCI, CircleCI, Jenkins o Shipable. Los tres primeros se pueden configurar directamente desde GH, para el último hay que entrar en [su web](#) y activar el repositorio que haga falta. Estos servicios realizan una serie de tests o generación de código sobre el proyecto y dan un resultado indicando qué tests se han pasado o no. Para indicar qué tests se hacen y los parámetros del repositorio, cada uno usa un formato diferente, aunque son habituales los ficheros de formato YAML o XML. En [el repo de ejemplo](#) se han activado Travis y Shippable, y en el directorio principal se pueden ver los ficheros de configuración (del mismo nombre que el sitio).
- Servicios de mensajería diversos, que envían mensajes cuando sucede algo. Entre estos últimos está [Twitter](#), que se puede configurar para que se cree un tweet con el mensaje del commit cada vez que se haga uno. Puede ser bastante útil, si se usa este sitio, para mantenerte al día de la actividad de un grupo de trabajo. También hay otros servicios como Jabber o Yammer o comerciales como Amazon SNS.
- Entrega continua: a veces integrados con los de, valga la redundancia, integración continua, pero que permiten directamente, cuando se hace un push sobre una rama determinada, se despliegue en el sitio definitivo. Servicios como Azure lo permiten, pero también [CodeShip](#) o [Jenkins](#). Generalmente en este caso hay que configurar algún tipo de *secret* o *clave* que permita a GitHub acceder al sitio y depositar la *carga* que, inmediatamente, estará

disponible. De hecho, es muy fácil trabajar con esto [directamente desde el editor como Eclipse](#)

- Sistemas de trabajo en grupo, que integran GitHub con los sistemas que tengan de asignación de tareas, de resolución de incidencias incluidas por parte de clientes. Por ejemplo, Basecamp, Bugzilla o Zendesk. De hecho, el propio GitHub integra un sistema de incidencias que se puede usar fácilmente.
- Análisis del código como CodeClimate (que analiza una serie de parámetros del código), Depending (que analiza dependencias en PHP) o David-DM (que analiza dependencias para nodejs).

Lo interesante es que se puede trabajar con la mayoría de estos sistemas de forma gratuita, aunque algunos tienen un modelo *freemium* que te cobra a partir de un nivel determinado de uso (lo que es natural, si no no podrían ofrecértelo de forma gratuita). Además, integra la mayor parte de los sistemas que se usan habitualmente en la industria del software.

Algunos *hooks* interesantes: sistemas de integración continua

GitHub resulta ideal para trabajar con cualquier sistema de integración continua, sea alojado o propio. Los sistemas de integración continua funcionan de la forma siguiente:

- Provisionan una máquina virtual con unas características determinadas para ejecutar pruebas o compilar código.
- Instalan el software necesario para llevar a cabo dichas pruebas.
- Ejecutan las pruebas, creando finalmente un informe que indique cuantas han fallado o acertado.
- Crear un *artefacto*, que puede ir desde un fichero con el informe en un formato estándar (suele ser XUnit o JUnit) hasta el ejecutable que se podrá descargar directamente del sitio; esto último puede incluir también su despliegue en la *nube*, un IaaS (Infrastructure as a Service) o PaaS (Platform as a Service) en caso de que haya pasado todos los tests satisfactoriamente.

La integración continua forma parte de una metodología de [desarrollo basado en test o guiado por pruebas](#) que consiste en crear primero las pruebas que tiene que pasar un código antes de, efectivamente, escribir tal código. Las pruebas son tests unitarios y también de integración, que prueban las capas de la aplicación a diferentes niveles (por ejemplo, acceso a datos, procesamiento de los datos, UI). Todos los lenguajes de programación moderno incluyen una aplicación que crea un protocolo para llevar a cabo los test e informar del resultado y estos sistemas

van desde el humilde Makefile que se usa en diferentes lenguajes compilados hasta el complejo Maven, pasando por sistemas como los tests de Perl o los Rakefiles de Ruby. En cualquier caso, cada lenguaje suele tener una forma estándar de pasar los tests (`make test`, `npm test` o `mocha`) y los sistemas de integración continua hacen muy simple trabajar con estos tests estándar, pero también son flexibles en el sentido que se puede adaptar a todo tipo de programa.

Veamos como trabajar con [Travis](#). Se hace siguiendo estos pasos

1. [Darse de alta en Travis CI](#) usando la propia cuenta de GitHub
2. Activar el *hook* en [tu perfil de Travis](#).
3. Se añade el fichero `.travis.yml` a tu repositorio. Este dependerá del lenguaje que se esté usando, aunque si lo único que quieres es comprobar la ortografía de tus documentos, lo puedes hacer [como en el repositorio ejemplo](#)
4. Hacer push.

La mayoría de estos repositorios suelen usar un fichero en formato estándar, YAML, XML o JSON. Veamos qué hace el fichero que hemos usado para el repo ejemplo:

```
branches:
  except:
    - gh-pages
language: C
compiler:
  - gcc
before_install:
  - sudo apt-get install aspell-es
script: OUTPUT=`cat README.md | aspell list -d es -p ./aspell.es.pws`; if [ -n "$OUTPUT" ] ;
```

La estructura de YAML permite expresar vectores y matrices asociativas fácilmente. En general, nos vamos a encontrar con algo del tipo **variable: valor** que será una clave y el valor correspondiente; el valor, a su vez, puede incluir otras estructuras similares. Por ejemplo, la primera

```
branches:
  except:
    - gh-pages
```

es una clave, **branches**, que incluye otra clave, **except**, que a su vez apunta a un vector (diferentes valores precedidos por -) con las ramas que vamos a excluir. En este caso, **gh-pages**, la de las páginas. Si hubiera otra rama a excluir, iría de esta forma

```
branches:
  except:
    - gh-pages
    - a-excluir
```

es decir, como un array de dos componentes. Esto hará que, sobre nuestro repositorio, se prueben todas las ramas, inclusive por supuesto la máster. A continuación expresamos el lenguaje que se va a usar; Travis no admite cualquier lenguaje, pero algunos como C, Perl o nodejs los acepta sin problemas. Como hay varios compiladores posibles, a continuación le decimos qué compilador se va a usar (en realidad, no se usa ninguno). En otros lenguajes habría que decir qué intérprete o qué versión; estas claves son específicas del lenguaje.

```
before_install:
  - sudo apt-get install aspell-es
script: OUTPUT=`cat README.md | aspell list -d es -p ./aspell.es.pws`; if [ -n "$OUTPUT" ] ;
```

Como lo único que vamos a hacer en este caso es comprobar la ortografía del texto del fichero `README.md`, instalamos con `apt-get` (herramienta estándar para Linux) un diccionario en español; este instalará todas las dependencias a su vez. Finalmente, la orden marcada `script` es la que lleva a cabo la comprobación. Para un programa normal sería suficiente hacer `make test` (y definir las dependencias para este objetivo, claro). No nos preocupemos mucho por lo que es, sino por lo que hace: si hay alguna palabra que no pase el test ortográfico, [fallará y enviará un mensaje de correo electrónico a la persona que haya hecho un commit indicándolo](#). Si lo pasa sin problemas, [también enviará el mensaje indicando que todo está correcto](#). Este tipo de cosas resulta útil sólo por el hecho de que se ejecuten automáticamente, pero pueden servir también para hacer despliegues continuos.

Travis también proporciona un *badge* que puedes incluir en tu repositorio para indicar si pasa los tests o no, que puedes incluir en tu fichero `README.md`(o donde quieras) con este código

```
[![Build Status](https://travis-ci.org/oslugr/repo-ejemplo.svg?branch=master)](https://tr
```

sustituyendo el nombre de usuario y el nombre del repo por el correspondiente, claro. Este código está escrito en Markdown, y GitHub lo interpretará directamente sin problemas, aunque lo mejor es que pinches en la imagen que aparece arriba a la derecha que te dará el código correspondiente.

Cliente de GitHub

GitHub también mantiene un [cliente de GitHub](#), escrito en Ruby y llamado `hub`, que se puede usar para sustituir a `git` o por sí mismo. En realidad, es como `git`

salvo que tiene ya definidos por omisión una serie de características específicas de GitHub, como los nombres de los repositorios o los usuarios de los mismos. Tras [instalarlo](#) puedes usarlo, por ejemplo, para clonar el repo de ejemplo usado aquí con `hub clone oslugr/repo-ejemplo` en vez de usar el camino completo a git; el formato sería siempre `usuario/nombre-del-repo`. Más órdenes que añade a git (y que se pueden usar directamente desde git si se usa, como se indica en las instrucciones, git como un alias de `hub`):

- `hub browse`: abre un navegador en la página del repositorio correspondiente. Por ejemplo `hub browse -- issues` lo abriría en la página correspondiente a las solicitudes de ese proyecto.
- `hub fork`: una vez clonado un repositorio de otro usuario, no hace falta hacer *fork* desde la web, se puede hacer directamente desde el repo. Se crea un origen remoto con el nombre de tu usuario, al que se puede hacer *push* de la forma normal. Desde la misma línea de órdenes se puede hacer un *pull request* al repositorio original también.
- Como usuario puedes aplicar también los *pull requests* desde línea de órdenes.
- `hub compare` permite comparar entre diferentes tags o versiones o ramas.

En general, ya que se tiene GitHub, conviene usar este cliente, sea o no con un alias a git. Por lo menos su uso es conveniente.

Listo para el lanzamiento: publicación en GitHub

GitHub, como cualquier otro repositorio git, permite usar una rama específica para depositar las versiones descargables; una forma de hacerlo, por ejemplo, es usar la rama `gh-pages` para no mezclarlo con el resto de los ficheros que están en el repositorio y, por tanto, versionados. Sin embargo, no es una buena idea poner ficheros binarios bajo control de git, porque es muy fácil provocar conflictos con ellos y no tan fácil resolverlos (o es un fichero o es otro, los algoritmos de diferencias de texto no trabajan sobre ficheros que no sean binarios); además, en general, estos ficheros se generan a partir del fuente de una forma más o menos automática: usando `Makefiles`, por ejemplo, en C, o en general compilando; si se trata de paquetes, cada lenguaje tiene mecanismos específicos para crearlos a partir de los fuentes, por lo tanto no es necesario colocar tales ficheros en el repositorio. Por eso es mejor colocarlos *fuera* del repositorio, y es lo que hace GitHub, en un apartado llamado *archivos*.

Crear un lanzamiento es fácil en GitHub: simplemente se crea una etiqueta como se ha visto anteriormente, con `git tag`. Por ejemplo, [este es el fichero correspondiente a la etiqueta v0.0.1 que se definió en el repositorio de ejemplo](#). Al pinchar en [Releases](#) te aparecen las versiones que ya has creado o un botón con *Draft a new release* para crear una nueva.

Desde esta interfaz web se puede añadir alguna información más que desde la línea de comandos: se puede crear la etiqueta si no existe y se pueden añadir imágenes, ficheros binarios generados de cualquier otra forma (o automáticamente) y, en general, lo que uno desee. También se puede marcar como *pre-release* y darle un título como a las versiones de Ubuntu, con animalitos o nombres de días de la semana o lo que sea.

En general, si no se usa ningún repositorio de módulos o aplicaciones para publicar la aplicación, o simplemente se quiere publicar junto con los fuentes, manuales y lo que se desee, es conveniente usar esta característica de GitHub para mantener un archivo de versiones descargables de la misma y también para que puedan acceder a ella fácilmente quienes no quieran usar simplemente `git` para descargársela.

Vais a decir que ya podían instalarse `git` y demás herramientas necesarias para compilar o ejecutar la aplicación, pero en muchos casos no tiene por qué ser fácil o factible; no se va a instalar uno un compilador de fortran simplemente para compilar una aplicación nuestra, por ejemplo.

Hooks: ejecutando código tras una orden git

Objetivos

- Entender el concepto de *fontanería* y *loza*
- Entender el concepto de *hooks* o *puntos de enganche*
- Entender las órdenes menos usuales de git usadas desde los *hooks*
- Saber adaptar *hooks* para una labor determinada

Viendo las cañerías: estructura de un repositorio git

Cuando se crea por primera vez un repositorio veremos que aparecen misteriosamente una serie de ficheros con esta estructura dentro del directorio `.git`.

`branches` lo dejamos de lado porque ya no se usa (aunque por alguna razón se sigue creando). `config`, `HEAD`, `refs` y `objects` son ficheros o directorios que almacenan información dinámica, por ejemplo `config` almacena las variables de configuración (que se han visto anteriormente). El resto de los ficheros y directorios se copian de una *plantilla*; esta plantilla se instala con `git` y se usa cada vez que hacemos `clone` o `init`, y contiene `hooks`, `description`, `branches` e `info` y los ficheros que se encuentran dentro de ellos.

Esta plantilla la podemos modificar y cambiar. La plantilla que se usa por omisión se encuentra en `/usr/share/git-core/templates/` y contiene una serie de ficheros junto con ejemplos (*samples*) para ganchos. Sin embargo, podemos personalizar nuestra plantilla editando (con permiso de superusuario) estos

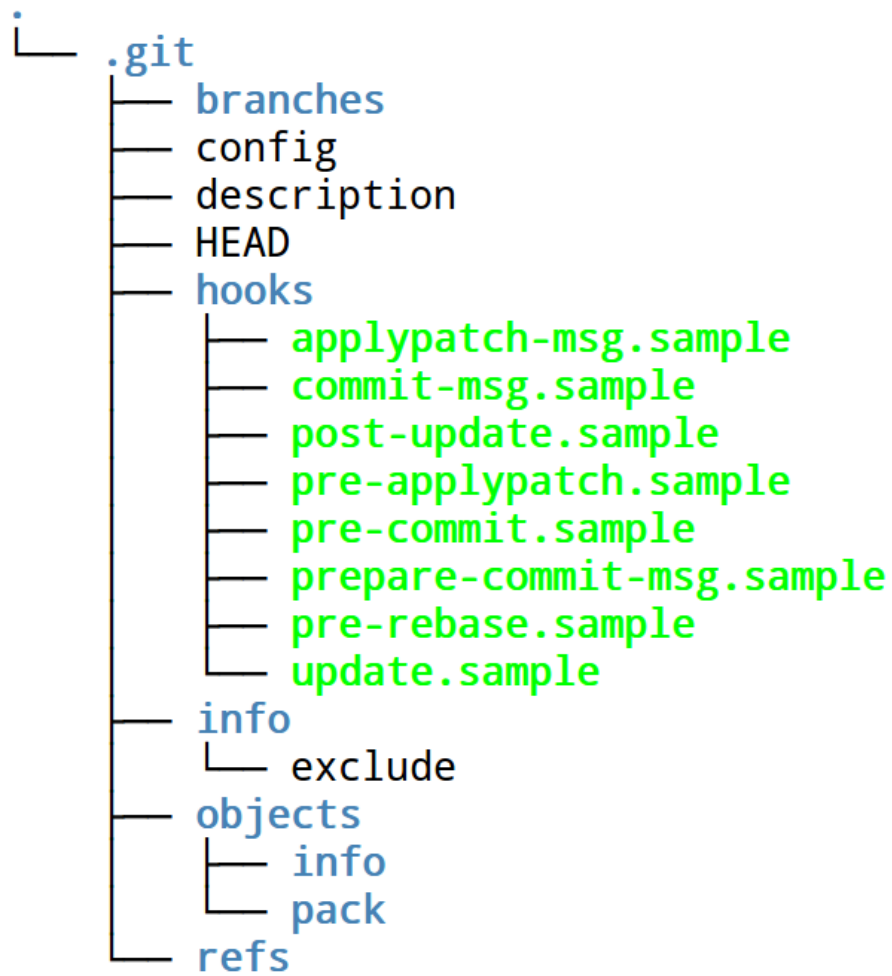


Figure 21: Estructura básica de un repositorio git

ficheros o bien usando la opción `--template <nombre de directorio>` de `clone` o `init`. En ese caso, en vez de copiar los ficheros por omisión, copiará los contenidos en ese directorio.

Por ejemplo, se puede usar [esta plantilla](#) que elimina los ficheros de ejemplo, sustituye por otro y traduce los contenidos de los otros ficheros al castellano; también mete en los patrones ignorados (sin necesidad de usar `.gitignore`) los ficheros que terminan en `~`, que produce Emacs como copia de seguridad.

Estos ficheros forman parte de las cañerías de `git` y podemos cambiar su comportamiento editando `config` como ya se ha visto en el capítulo de uso básico; de hecho, existe también un fichero de configuración a nivel global, `.gitconfig` que sigue el mismo formato y que ya hemos visto

```
[alias]
    ci = commit
    st = status
[core]
    editor = emacs
```

Estos ficheros de configuración siguen un formato similar al de los ficheros `.ini`, es decir, bloques definidos entre corchetes y variables con valor, dentro de ese bloque, a las que se le asigna usando `=`. En este caso [definimos dos alias](#) y un editor o, mejor dicho, *el* editor. Esto podemos hacerlo tanto en el fichero global como en el local si queremos que afecte sólo a nuestro repositorio.

Otro fichero dentro de este directorio que se puede modificar es `.git/info/exclude`; es similar a `.gitignore`, salvo que en este caso afectará solamente a nuestra copia local del repositorio y no a todas las copias del mismo. Por ejemplo, podemos editarlo de esta forma

```
# git ls-files --others --exclude-from=.git/info/exclude
# Lines that start with '#' are comments.
# For a project mostly in C, the following would be a good set of
# exclude patterns (uncomment them if you want to use them):
# *.loa
*~
```

para excluir los ficheros de copia de seguridad de Emacs (que hemos definido antes como editor) que nos interesa evitar a nosotros, pero que puede que tengan un significado especial para otro usuario del repo y que por tanto no quiera evitar.

Por supuesto, el tema principal de este capítulo está en el otro directorio, *hooks*, cuyo contenido tendremos que cambiar si queremos añadir ganchos al repositorio. Pero para usarlo necesitamos también conocer algunos conceptos más de `git`, empezando por cómo se accede a más cañerías.

Paso a paso

Si miramos en el directorio `.git/objects` encontraremos una serie de directorios con nombres de dos letras, dentro de los cuales están los objetos de git, que tienen otras 38 letras para componer las 40 letras que componen el nombre único de cada objeto; este nombre se genera a partir del contenido usando [SHA1](#).

git almacena toda la información en ese directorio y de hecho está [organizado para acceder a la información almacenada por contenido](#). Por eso tenemos que imaginarlo como un sistema de ficheros normal, con una raíz (que es HEAD, el punto en el que se encuentra el repositorio en este momento) y una serie de ramas que apuntan a ficheros y a diferentes versiones de los mismos.

git, entonces, [procede de la forma siguiente](#).

1. Crea un SHA1 a partir del contenido del fichero cambiado o añadido. Este fichero se almacena en la zona temporal en forma de *blob*.
2. El nombre del fichero se almacena en un árbol, junto con el nombre de otros ficheros que estén, en ese momento, en la zona temporal. Un árbol almacena los nombres de varios ficheros y apunta al contenido de los mismos, almacenado como *blob*. De este objeto, que tiene un formato fijo, se calcula también el SHA1 y se almacena en `.git/objects`. Un árbol, a su vez, puede apuntar a otros árboles creados de la misma forma.
3. Cuando se hace *commit*, se crea un tercer tipo de objeto con ese nombre. Un *commit* contiene enlaces a un árbol (el de más alto nivel) y metadatos adicionales: quién lo ha hecho, cuándo y por supuesto el mensaje de commit.

Veremos más adelante cómo se listan ficheros de todos estos tipos, pero por lo pronto la idea es que un comando de git de alto nivel involucra varias órdenes de bajo nivel que, eventualmente, van a parar a información que se almacena en un directorio determinado con nombres de fichero que se calculan usando SHA1, aparte de que se puede actuar a diferentes niveles, desde el más bajo de almacenar un objeto directamente en un árbol o crear un commit “a mano” hasta el más alto (que es el que estamos acostumbrados). Este sistema, además, asegura que no se pierda ninguna información y que podamos acceder al contenido de un fichero determinado hecho en un momento determinado de forma fácil y eficiente. Pero para poder hacerlo debe haber una forma única y también compacta de referirse a un elemento determinado dentro de ese repositorio. Es lo que explicaremos a continuación.

El nombre de las cosas: refiriéndonos a objetos en git.

Como ya hemos visto antes, todos los objetos (sean *blobs*, árboles o *commits*) están representados por un SHA1. Si conocemos el SHA1, se puede usar `show`,

por ejemplo, para visualizarlo. Haciendo `git log` veremos, por ejemplo, los últimos commits y si hacemos `show` sobre uno de ellos,

```
git show fe88e5eefff7f3b7ea95be510c6dcb87054bbcb
commit fe88e5eefff7f3b7ea95be510c6dcb87054bbcb0
Author: JJ Merele <jjmerele@gmail.com>
Date: Thu Apr 17 18:29:11 2014 +0200
    Añade layout
diff --git a/views/layout.jade b/views/layout.jade
new file mode 100644
index 0000000..36cc059
--- /dev/null
+++ b/views/layout.jade
@@ -0,0 +1,6 @@
[....]
```

El mismo resultado que obtendríamos si hacemos `git show HEAD`, que recordemos que es una referencia que apunta al último commit. También obtendremos lo mismo si hacemos `git show master`. En cualquiera de los casos, lo que está mostrando es un objeto de tipo *commit*, el último realizado.

Pero veremos como funciona este último ejemplo. Al lado del directorio `objects` está el directorio `refs`, que almacena referencias y que es como `git` sabe a qué commit corresponde cada cosa. Este comando:

```
~/repos-git/repo-ejemplo<master>$ tree .git/refs/ .git/refs/
heads      img-dir      master  remotes  heroku
master     origin      HEAD    img-dir  master  tags
v0.0.1     v0.0.2     v0.0.2.1 v0.0.3 5 directories, 10 files
```

muestra todo lo que hay almacenado en este directorio: referencia a las ramas locales en `heads` y a las remotas en `remotes`. Si mostramos el contenido de los ficheros:

```
~/repos-git/repo-ejemplo<master>$ cat .git/refs/heads/master
fe88e5eefff7f3b7ea95be510c6dcb87054bbcb0
```

Que muestra que, efectivamente, el hash del commit es el que corresponde

Podemos mirar en `.git/objects/fe` a ver si efectivamente se encuentra; puedes hacerlo sobre tu copia del repositorio `repo-ejemplo`, ya que los hash son iguales en todos lados.

Como hemos visto anteriormente, un *commit* apunta a un árbol. Podemos indicarle a `show` que nos muestre este árbol de esta forma:

```
~/repos-git/repo-ejemplo<master>$ git show master^{tree}
tree master^{tree}
.aspell.es.pws
.gitignore
.gitmodules
.travis.yml
LICENSE
Procfile
README.md
curso
package.json
shippable.yml
test/
views/
web.js
```

En este caso el [formato es rama \(circunflejo o *caret*\) {tree}](#); el circunflejo se usa en la selección de referencias de `git` para cualificar lo que se encuentra antes de ella, pero no hay muchas más opciones aparte de `tree`, pero sí podemos acceder a versiones anteriores del repositorio y a sus ficheros.

```
~/repos-git/repo-ejemplo<master>$ git show master~1
commit 5be23bb2a610260da013fcea807be872a4bd6981
Author: JJ Merelo <jjmerelo@gmail.com>
Date: Thu Apr 17 17:42:39 2014 +0200
    Aclara una frase
[...]
```

La [tilde ~](#) indica un ancestro, es decir, el *padre* del commit anterior, que, como vemos [corresponde al commit 5be23bb](#).

La lista completa de opciones para especificar revisiones está, curiosamente, en [la página de referencia del comando `rev-parse`](#). Hay un número excesivo de ellas, pero si en algún momento no se entiende qué es lo que se está usando, conviene ir ahí.

Podemos ir más allá hasta que nos aburramos: `~2` accederá al padre de este y así sucesivamente. Y, por supuesto, podemos cualificarlo con `^{tree}` para que nos muestre el árbol en el estado que estaba en ese commit. Y también para que nos muestre un fichero sin necesidad de sacarlo del repositorio:

```
~/repos-git/repo-ejemplo<master>$ git show master~2:README.md
repo-ejemplo
=====
```

Ejemplo de repositorio para trabajar en el
[curso de `git`](http://cevug.ugr.es/git) el contenido del cual está
[...]

En esta sección hemos usado `show` para mostrar las capacidades de los diferentes selectores de `git`, pero se pueden usar con cualquier otra orden, como `checkout` o cualquiera que admita, en general, una referencia a un objeto.

Comandos de alto y bajo nivel: *fontanería y loza*

Para entendernos, todas las órdenes que hemos usado hasta ahora son *loza*. Es decir, es el *interfaz* del usuario de toda la instalación de fontanería que lleva a cabo realmente la labor de quitar de enmedio lo que uno deposita en las instalaciones sanitarias. Pero por debajo de la loza y pegado a ella, están las cañerías y toda la instalación de fontanería.

Los comandos de `git` se dividen en **dos tipos**: *fontanería* o *cañería*, que son comandos que *generalmente* no ve el usuario y *loza*, que son los que ve y los que usa. Sin embargo, este capítulo trata realmente de esa fontanería, porque van a ser una serie de órdenes que se van a llevar a cabo *después* de que se ejecuten las órdenes de *loza*, o, quizás *dentro* de esas órdenes de loza.

Pero antes de usar esas órdenes de fontanería tenemos que entender cómo son las cañerías. Una parte se ha visto anteriormente: el *index* o índice que contiene todos los objetos a los que `git` debe prestarles atención a la hora de hacer un commit. Pero existen además **los objetos y las referencias**.

Los *objetos* son, en general, información que está almacenada en el repositorio. Incluyen, por supuesto, los ficheros que almacenamos en el mismo, pero también los mensajes de commit, las etiquetas y los *árboles*. Los ficheros almacenados están *divididos*: el *contenido* del fichero se almacena en un *blob* y el nombre del fichero se almacena en el árbol. Hay, pues, cuatro tipos de objetos: *blob*, *tree*, *commit* y *tag*.

La orden `ls-tree` nos permite ver qué tipos de objetos tenemos almacenados y sus códigos SHA1, que son los nombres de ficheros calculados a partir del contenido del mismo. Aunque todos están almacenados en el directorio `.git/objects`, esta orden nos permite ver también de qué tipo son:

```
~/repos-git/repo-ejemplo<master>$ git ls-tree HEAD
100644 blob a6f69e4284566cd84272c6a4e4996f64643afbea  .aspell.es.pws
100644 blob a72b52ebe897796e4a289cf95ff6270e04637aad  .gitignore
100644 blob cc5411b5557f43c7ba2f37ad31f8dc34ccda075   .gitmodules
100644 blob 4e7b6c1b5a6cb3a962ea05874d10c943c1923f39  .travis.yml
100644 blob d5445e7ac8422305d107420de4ab8e1ee6227cca  LICENSE
100644 blob d1913ebe4d9e457be617ee0e786fc8c30a237902  Procfile
```

```

100644 blob c5badda0c484c989e958ea4e27dfe11d69f3c8ef    README.md
160000 commit fa8b7521968bddf235285347775b21dd121b5c11  curso
100644 blob f8c35adaf57066d4329737c8f6ec7ce6179cc221  package.json
100644 blob 08827778af94ea4c0ddbc28194ded3081e7b0f87  shippable.yml
040000 tree 39da6b155c821af1e6a304daca9b66efb1ac651f    test
100644 blob 94f151d9ef9340c81989b0c3fa8c517c068e1864    web.js

```

En este caso tenemos objetos de tres tipos: blob, commit y tree. a `ls-tree` se le pasa un *tree-ish*, es decir, algo que apunte a dónde esté almacenado un árbol pero, para no preocuparnos de qué se trata esto, usaremos simplemente `HEAD`, que apunta como sabéis a la punta de la rama en la que nos encontramos ahora mismo. También nos da el SHA1 de 40 caracteres que representa cada uno de los ficheros. Si queremos que se expandan los `tree` para mostrar los ficheros que hay dentro también usamos la opción `-r`

```

~/repos-git/repo-ejemplo<master>$ git ls-tree -r HEAD
100644 blob a6f69e4284566cd84272c6a4e4996f64643afbea    .aspell.es.pws
100644 blob a72b52ebe897796e4a289cf95ff6270e04637aad    .gitignore
100644 blob cc5411b5557f43c7ba2f37ad31f8dc34ccda075    .gitmodules
100644 blob 4e7b6c1b5a6cb3a962ea05874d10c943c1923f39    .travis.yml
100644 blob d5445e7ac8422305d107420de4ab8e1ee6227cca    LICENSE
100644 blob d1913ebe4d9e457be617ee0e786fc8c30a237902    Procfile
100644 blob da5b5121adb42e990b9e990c3edb962ef99cb76a    README.md
160000 commit fa8b7521968bddf235285347775b21dd121b5c11  curso
100644 blob f8c35adaf57066d4329737c8f6ec7ce6179cc221  package.json
100644 blob 08827778af94ea4c0ddbc28194ded3081e7b0f87  shippable.yml
100644 blob 9920d80438d42e3b0a6924a0fcace2d53a6af602  test/route.js
100644 blob 36cc059186e7cb247eaf7bfd6a318be6cfff9ea3  views/layout.jade
100644 blob 97c32024cda29e0fb6abebf48d3f6740f0acb9e2    web.js

```

que muestra solo los objetos de tipo `blob` (y un `commit`) con el camino completo que llega hasta ellos.

Si editamos un fichero tal como el `README.md`, tras hacer el `commit` tendrá esta apariencia:

```

~/repos-git/repo-ejemplo<master>$ git ls-tree HEAD
100644 blob a6f69e4284566cd84272c6a4e499
[...]
100644 blob da5b5121adb42e990b9e990c3edb962ef99cb76a    README.md

```

Como vemos, ha cambiado el SHA1. Pero `ls-tree` va más allá y te puede mostrar también cuál es el estado del repositorio hace varios `commits`. Por ejemplo, podemos usar `HEAD^` para referirnos al `commit` anterior y `git ls-tree HEAD^` nos devolvería exactamente el mismo estado en el que estaba antes de hacer la modificación a `README.md`. De hecho, podemos usar también la

abreviatura del commit de esta forma `git ls-tree 5be23bb`, siendo este último una parte del SHA1 (o hash) del último commit; nos devolvería el último resultado.

Pero podemos ir todavía más profundamente dentro de las tuberías. `ls-tree` sólo lista los objetos que ya forman parte del árbol, del principal o de alguno de los secundarios. Puede que necesitemos acceder a aquellos objetos que se han añadido al índice, pero todavía no han pasado a ningún árbol. Para eso usamos `ls-files`. Tras añadir un fichero que está en un subdirectorio `views` con `add`, podemos hacer:

```
git ls-files --stage
100644 a6f69e4284566cd84272c6a4e4996f64643afbea 0      .aspell.es.pws
100644 a72b52ebe897796e4a289cf95ff6270e04637aad 0      .gitignore
100644 cc5411b5557f43c7ba2f37ad31f8dc34ccda075 0      .gitmodules
100644 4e7b6c1b5a6cb3a962ea05874d10c943c1923f39 0      .travis.yml
100644 d5445e7ac8422305d107420de4ab8e1ee6227cca 0      LICENSE
100644 d1913ebe4d9e457be617ee0e786fc8c30a237902 0      Procfile
100644 da5b5121adb42e990b9e990c3edb962ef99cb76a 0      README.md
160000 fa8b7521968bddf235285347775b21dd121b5c11 0      curso
100644 f8c35adaf57066d4329737c8f6ec7ce6179cc221 0      package.json
100644 08827778af94ea4c0ddbc28194ded3081e7b0f87 0      shippable.yml
100644 9920d80438d42e3b0a6924a0fcace2d53a6af602 0      test/route.js
100644 36cc059186e7cb247eaf7bfd6a318be6cffb9ea3 0      views/layout.jade
100644 94f151d9ef9340c81989b0c3fa8c517c068e1864 0      web.js
```

Que nos devuelve, en penúltimo lugar, un fichero que todavía no ha pasado al árbol. Evidentemente, tras el commit:

```
~/repos-git/repo-ejemplo<master>$ git ls-tree HEAD
[...]
040000 tree fd3846c0d6089437598004131184c61aea2b6514      views
```

Este listado nos muestra el nuevo objeto de tipo `tree` que se ha creado y nos da su SHA1, que podemos usar para examinarlo con `ls-tree`

```
~/repos-git/repo-ejemplo<master>$ git ls-tree fd3846c
100644 blob 36cc059186e7cb247eaf7bfd6a318be6cffb9ea3      layout.jade
```

que, si queremos ver en una vista más normal, hacemos lo mismo con `ls-file`

```
~/repos-git/repo-ejemplo<master>$ git ls-files views
views/layout.jade
```


Hay un tercer comando relacionado con el examen de directorios y ficheros locales, `cat-file`, que muestra el contenido de un objeto, en general. Por ejemplo, en este caso, para listar el contenido de un objeto de tipo `tree`

```
~/repos-git/repo-ejemplo<master>$ git cat-file -p fd3846c
100644 blob 36cc059186e7cb247eaf7bfd6a318be6cffb9ea3    layout.jade
```

nos muestra que ese objeto contiene un solo fichero, `layout.jade`, y sus características. Pero más curioso aún es cuando se usa sobre objetos de tipo `commit` (no sobre el objeto `commit` que aparece arriba, que se trata de un `commit` de otro repositorio al contener el directorio `curso` un submódulo de git. Por ejemplo, podemos hacer:

```
~/repos-git/repo-ejemplo<master>$ git cat-file -p HEAD
tree 1c40899a32c2b5ec7f930bd943e5d5bb98562d373
parent 5be23bb2a610260da013fcea807be872a4bd6981
author JJ Merelo <jjmerelo@gmail.com> 1397752151 +0200
committer JJ Merelo <jjmerelo@gmail.com> 1397752151 +0200
Añade layout
```

que, dado que `HEAD` apunta al último commit, nos muestra en modo *pretty-print* toda la información sobre el último `commit` y muestra el árbol de ficheros correspondiente, que podemos listar con

```
~/repos-git/repo-ejemplo<master>$ git cat-file -p 1c40899a
100644 blob a6f69e4284566cd84272c6a4e4996f64643afbea    .aspell.es.pws
100644 blob a72b52ebe897796e4a289cf95ff6270e04637aad    .gitignore
100644 blob cc5411b5557f43c7ba2f37ad31f8dc34ccda075     .gitmodules
100644 blob 4e7b6c1b5a6cb3a962ea05874d10c943c1923f39    .travis.yml
100644 blob d5445e7ac8422305d107420de4ab8e1ee6227cca    LICENSE
100644 blob d1913ebe4d9e457be617ee0e786fc8c30a237902    Procfile
100644 blob da5b5121adb42e990b9e990c3edb962ef99cb76a    README.md
160000 commit fa8b7521968bddf235285347775b21dd121b5c11  curso
100644 blob f8c35adaf57066d4329737c8f6ec7ce6179cc221    package.json
100644 blob 08827778af94ea4c0ddbc28194ded3081e7b0f87    shippable.yml
040000 tree 39da6b155c821af1e6a304daca9b66efb1ac651f    test
040000 tree fd3846c0d6089437598004131184c61aea2b6514    views
100644 blob 97c32024cda29e0fb6abebf48d3f6740f0acb9e2    web.js
```

En general, si queremos ahondar en las entrañas de un punto determinado en la historia del repositorio, trabajar con `ls-files`, `cat-file` y `ls-tree` permite obtener toda la información contenida en el mismo. Esto nos va a resultar útil un poco más adelante.

Viva la diferencia En muchos casos para procesar los cambios dentro de un gancho necesitaremos saber cuál es la diferencia con versiones anteriores del fichero. Hay que tener en cuenta que esas diferencias, dependiendo del estado en el que estemos, estarán en el árbol o en el índice preparadas para ser enviadas al repositorio. En general, son una serie de órdenes con **diff** en ellas. La más simple, **git diff**, nos mostrará la diferencia entre los archivos en el índice y el último commit.

```
~/repos-git/repo-ejemplo<master>$ git diff
diff --git a/views/layout.jade b/views/layout.jade
index 36cc059..2a66d58 100644
--- a/views/layout.jade
+++ b/views/layout.jade
@@ -1,6 +1,11 @@
-!!! 5
+doctype html
+html(lang="en")

-body
+html
+  head
+    title #{title}
+  body

-#wrapper
-  block content
\ No newline at end of file
+    h1 Curso de git
+
+    block content
\ No newline at end of file
diff --git a/web.js b/web.js
index 97c3202..93b6255 100644
--- a/web.js
+++ b/web.js
@@ -27,6 +27,12 @@ app.get('/', function(req, res) {
    res.send(routes['README']);
  });

+app.get('/curso/:ruta', function(req, res) {
+  var ruta = "curso/texto/"+req.params.ruta;
+  console.log("Request "+req.params.ruta + " doc " + ruta + " contenido " + file_contenido);
+  res.render('doc', { content: routes[ruta], title: ruta });
+})
+
+app.get('/curso/texto/:ruta', function(req, res) {
```

```
//      console.log("Request "+req.params.ruta);
      var ruta_toda = "curso/texto/"+req.params.ruta;
```

Esta vista de [diff](#) las diferencias sigue el formato habitual en [la utilidad diff](#), que permite generar parches para aplicarlos a conjuntos de ficheros. En concreto, muestra qué ficheros se están comparando (pueden ser diferentes ficheros, si se ha cambiado el nombre) los SHA1 de los contenidos correspondientes, y luego un + o - delante de cada una de las líneas que hay de diferencia. Este fichero se podría usar directamente con la utilidad `diff` de Linux, pero realmente no nos va a ser de mucha utilidad a la hora de saber, por ejemplo, qué ficheros se han modificado. Para hacer esto, [simplemente](#):

```
~/repos-git/repo-ejemplo<master>$ git diff --name-only
views/layout.jade
web.js
```

que se puede hacer un poco más completa con `--name-status`:

```
~/repos-git/repo-ejemplo<master>$ git diff --name-status
M      views/layout.jade
M      web.js
~/repos-git/repo-ejemplo<master>$ git diff --name-status --cached
A      views/doc.jade
```

que nos muestra, usando una sola letra, qué tipo de cambio han sufrido. En el primer caso nos muestra que han sido Modificados, y en el segundo caso, además usamos otra opción, `--cached` que, en este caso, nos muestra los ficheros que han sido preparados para el commit; es decir, la [diferencia que hay entre la cabeza y el índice](#); en este caso, “A” indica que se trata de un fichero añadido. Podemos ver todo junto con

```
~/repos-git/repo-ejemplo<master>$ git diff --name-status HEAD
A      views/doc.jade
M      views/layout.jade
M      web.js
```

que nos muestra la diferencia entre el commit más moderno (HEAD) y el área de trabajo ahora mismo: hemos cambiado dos ficheros y añadido uno. Un resultado similar obtendremos con `diff-index`, cuya principal diferencia es que compara siempre el índice con algún *árbol*, sin tener ningún valor por omisión:

```
~/repos-git/repo-ejemplo<master>$ git diff-index HEAD
:000000 100644 0000000000000000000000000000000000 67e6d7e1ecbb64ff7d467dc2103fa2b2f
:100644 100644 36cc059186e7cb247eaf7bfd6a318be6cffb9ea3 00000000000000000000000000000000
:100644 100644 97c32024cda29e0fb6abebf48d3f6740f0acb9e2 00000000000000000000000000000000
```

Además del estado muestra el hash inicial y final de cada uno de los ficheros. En este caso, como todavía no le hemos hecho commit, muestra 0. `diff-tree`, sin embargo, sí muestra el SHA1 puesto que trabaja con el árbol:

```
~/repos-git/repo-ejemplo<master>$ git diff-tree HEAD
fe88e5eefff7f3b7ea95be510c6dcb87054bbcb0
:000000 040000 00000000000000000000000000000000000000000000000000000000 fd3846c0d6089437598004131184c61aea2b6514 6bb4560a218c008bbc468f23f36f26ff6
:100644 100644 94f151d9ef9340c81989b0c3fa8c517c068e1864 97c32024cda29e0fb6abebf48d3f6740f0acb9e2 93b625533c2d1752d9a8e789878512919
```

Aunque en este caso muestra un árbol, `views`, que ha sido cambiado porque se le ha añadido un fichero nuevo, `views/doc.jade`. En el momento que se haga el commit y pase por tanto del índice al la zona de *staging*, los hash ya están calculados y cambia la salida:

```
~/repos-git/repo-ejemplo<master>$ git diff-tree HEAD
637c2820013188f1c4951aef0c21de20440a6fbb
:040000 040000 fd3846c0d6089437598004131184c61aea2b6514 6bb4560a218c008bbc468f23f36f26ff6
:100644 100644 97c32024cda29e0fb6abebf48d3f6740f0acb9e2 93b625533c2d1752d9a8e789878512919
```

`diff-index`, sin embargo, no devolverá nada puesto que todos los cambios que se habían hecho han pasado al árbol.

En general, lo que más nos va a interesar a la hora de hacer un gancho es qué ficheros han cambiado. Pero conviene conocer toda la gama de posibilidades que ofrece `git`, sobre todo para poder entender su estructura interna.

Los dueños de las tuberías No todo el contenido que hay en el repositorio son los ficheros que forman parte del mismo. Hay una parte importante de la fontanería que son los metadatos del repositorio. Hay dos órdenes importantes, `var` y `config`. Con `-l` nos listan todas las variables o variables de configuración disponibles

```
~/repos-git/curso-git/texto<master>$ git var -l
user.email=jjmerelo@gmail.com
user.name=JJ Merelo
filter.obj-add.smudge=cat
push.default=simple
rerere.enabled=true
core.repositoryformatversion=0
core.filemode=true
core.bare=false
core.logallrefupdates=true
remote.origin.url=git@github.com:oslugr/curso-git.git
remote.origin.fetch=+refs/heads/*:refs/remotes/origin/*
```

```
branch.master.remote=origin
branch.master.merge=refs/heads/master
GIT_COMMITTER_IDENT=JJ Merelo <jjmerelo@gmail.com> 1399019391 +0200
GIT_AUTHOR_IDENT=JJ Merelo <jjmerelo@gmail.com> 1399019391 +0200
GIT_EDITOR=editor
GIT_PAGER=pager
```

Todas excepto las cuatro últimas variables son variables de configuración que, por tanto, se pueden obtener también con `git config -l`. Por sí sólo, `config` o `var` listan el valor de una variable:

```
~/repos-git/curso-git/texto<master>$ git config user.name
JJ Merelo
```

La mayoría de estos valores están disponibles o como variables de entorno o en ficheros; sin embargo estas órdenes dan un interfaz común para todos los sistemas operativos.

Todavía nos hacen falta una serie de órdenes para tomar decisiones sobre ficheros y sobre dónde estamos en el repositorio. La veremos a continuación

Simplemente, `rev-parse` La [tersa descripción del comando `rev-parse`](#), “[recoge y procesa parámetros](#)” esconde la complejidad del mismo y su potencia, que va desde el procesamiento de parámetros hasta la especificación de objetos, pasando por la búsqueda de diferentes directorios dentro del repositorio git. Por ejemplo, se puede usar para verificar si un objeto existe o no:

```
~/repos-git/repo-ejemplo<master>$ git rev-parse --verify HEAD
637c2820013188f1c4951aef0c21de20440a6fbb
```

Nos muestra el SHA1 de la cabeza actual del repositorio de ejemplo, verificando que actualmente existe. No lo hará si acabamos de crear el repositorio, por ejemplo

```
/tmp/pepe<>$ git init
Initialized empty Git repository in /tmp/pepe/.git/
/tmp/pepe<>$ git rev-parse --verify HEAD
fatal: Needed a single revision
```

De hecho, con él podemos encontrar todo tipo de objetos usando la notación que permite especificar revisiones

```
~/repos-git/repo-ejemplo<master>$ git rev-parse HEAD@{1.month}
61253ecba351921c96a1553f6c5b7f9910f286f3
```

que correspondería a [este commit del 9 de marzo](#) y que podemos listar usando `show`, `describe` o cualquiera de los otros comandos que se pueden aplicar a objetos. En general, para esto se usará dentro de los *garfios*: para poder acceder a un objeto determinado o a sus metadatos a la hora de ver las diferencias con el objeto actual.

Concepto de *hooks*

Un *hook* literalmente *garfio* o *gancho* es un programa que se ejecuta cuando sucede un evento determinado en el repositorio. Los *webhooks* de GitHub, por ejemplo, son un ejemplo: cuando se lleva a cabo un *push*, se envía información al sitio configurado para que ejecute un programa determinado: pase unos tests, publique un tweet, o lleve a cabo una serie de comprobaciones.

Los *ganchos* no son estrictamente necesarios en todo tipo de instalaciones; se puede trabajar con un repositorio sin tener la necesidad de usarlos. Sin embargo, [son tremendamente útiles para automatizar una serie de tareas](#) (como los tests que se usan en integración continua), implementar una serie de políticas para todos los usuarios de un repositorio (formato de los mensajes de *commit*, por ejemplo) y añadir información al repositorio de forma automática.

Los *hooks* son, por tanto, programas ejecutables. Cualquier programa que se pueda lanzar puede servir, pero generalmente se usa o guiones del *shell* (si uno es suficientemente masoquista) o lenguajes de *scripting* tales como Perl, Python, Ruby, Javascript o, si uno es *realmente* masoquista, PHP. En realidad a git le da igual qué lenguaje se use.

Los *hooks* van en su propio directorio, `.git/hooks` que se crea automáticamente y que tiene, siempre, una serie de *scripts* ejemplo, ninguno de ellos activados. Sólo se admite un *hook* por evento, y ese *hook* tendrá el nombre del evento asociado; es decir, un programa llamado `post-merge` en ese directorio se ejecutará siempre cada vez que se termine un *merge* con éxito. Como generalmente uno quiere que los scripts tengan un nombre razonable, la estrategia más general es usar un *enlace simbólico* de esta forma

```
ln -s nombre-real-del-script.sh post-merge
```

y, en todo caso, no se debe olvidar

```
chmod +x nombre-real-del-script.sh
```

para hacerlo ejecutable.

Windows seguramente tendrá su forma particular de hacer lo mismo, o ninguna. Por cualquiera de esas dos razones, nunca recomendamos Windows como una plataforma para desarrollo. Úsala para

la declaración de la renta o para jugar al Unreal, pero para desarrollar usa una plataforma para desarrolladores: Linux (o Mac, que tiene un núcleo Unix por debajo).

Los *hooks* se activarán cuando se ejecute un comando determinado y recibirán una serie de parámetros como argumento o en algún caso como entrada estándar. Este [cuadro](#) resume cuando se ejecutan y también qué reciben como parámetro. En general, también tendrán influencia en si tiene éxito o no el comando determinado: salir con un valor no nulo, en algunos casos, parará la ejecución del comando con un mensaje de error. Por ejemplo, un *hook applypatch-msg*, que se aplica desde el comando `git am` antes de que se ejecute, parará la aplicación del parche si se sale con un valor 1.

De todos los *hooks* posibles sólo veremos los que se refieren al *commit*. Son los que se pueden usar en local (los referidos a *push* sólo se programan en remoto, y los que se aplican a `git am` o `git gc` quedan fuera de los temas de este libro. Hay sólo cuatro de estos, que veremos a continuación.

Programando un *hook* básico

En general, un *hook* hará lo siguiente

1. Examinar el entorno y los parámetros de entrada
2. Hacer cambios en el entorno, los ficheros o la salida
3. Salir con un mensaje si hay algún error, o ninguno.

No son diferentes de ningún otro programa, en realidad, salvo que los parámetros de entrada y cómo se debe salir está establecido de antemano.

Miremos un script simple, que actúa como gancho para preparación de un mensaje de commit (`prepare-commit-msg`, incluido en el [repositorio de ejemplo de este curso](#))

```
#!/bin/sh
SOB=$(git config github.user)
grep -qs "^$SOB" "$1" || echo ". Cambio por @$SOB" >> "$1"
```

Este script tiene toda la simplicidad de estar en dos líneas y toda la complicación de estar escrito para el *shell*. [Esta introducción](#) venerable te puede ayudar a empezar a trabajar con él, pero en este capítulo no pretendemos que aprendas a programar, sólo que tengas las nociones básicas para echar a andar y posiblemente modificar ligeramente un gancho.

Empecemos por la primera línea: es común a todos los guiones del shell. Simplemente indica el camino en el que se encuentra el mismo, con `#!` indicando

que se trata de un fichero ejecutable (junto con el `chmod +x`, que se lo indica al sistema de ficheros).

La siguiente línea define una variable, SOB, que no es acrónimo de nada, cuidado. Esa variable usa el formato de ejecución de un comando del shell `$()` para asignar la salida de dicho comando a la variable.

La tercera línea, en resumen, comprueba si el nombre del usuario ya está en el mensaje y si no lo está le añade una “firma” que lo incluye. Lo hace de la forma siguiente: `grep -qs "^$SOB" "$1"` comprueba si el valor de la variable no (de ahí el caret `^`) está ya en el mensaje (cuyo nombre de fichero está en el primer argumento, `$1`, según vemos en la [chuleta de ganchos de git](#). Si no lo está (de ahí el `||`, es decir, *O*, se escribe (`echo`) el valor de la variable añadiéndose (`>>`) al final del fichero cuyo nombre está en la variable `$1`.

La variable `github.user` no tiene por qué estar definida siempre. Se puede sustituir por `user.email`, por ejemplo, o por `user.name`, que sí se suele definir siempre cuando se crea un repositorio.

Este [otro ejemplo](#) es todavía más minimalista, y también añade información al commit (que, como en el caso anterior, se puede eliminar si se edita el fichero de commit):

```
STATS=$(git diff --cached --shortstat)
echo ". Cambios en este commit\n ${STATS}" >> "$1"
```

Es interesante notar, en este caso, que se usa `diff --cached` ya que en este caso los cambios estarán ya *staged* o *cached* y la diferencias (que suelen aparecer de todas formas en el mensaje que da el comando) serán entre HEAD y lo que hay ya almacenado el área de preparación de los ficheros, no entre HEAD y lo que hay en el sistema de ficheros; esto es así porque ya estamos *dentro* del commit y los ficheros están ya preparados para ser procesados en las tuberías de `git` por sus comandos-tubería. En resumen, esta orden da una mini-estadística que dice el número de ficheros y líneas cambiadas, produciendo [resultados sobre este mismo fichero tales como este](#):

Mini-corrección

```
. Cambios en este commit
  1 file changed, 1 insertion(+)
```

Otro *hook* puede servir para comprobar que los mensajes de *commit* son correctos. Como se ha visto anteriormente, una buena práctica es usar una primera línea de 50 caracteres (que aparecerán como título) seguida por una línea vacía y el resto del mensaje. Esto se puede aplicar mediante [un programa en Python](#) o el siguiente en [Node](#), una versión de Javascript.


```
#!/usr/bin/env node

var fs = require('fs');
var msg_file = process.argv[2];

fs.readFile( msg_file, 'utf8', function( err, data ) {
    if ( err ) throw err;
    var lines = data.split("\n");
    if ( lines[0].length > 50 ) {
        console.log("[FORMATO] Primera línea > 50 caracteres");
        process.exit(1);
    }
})
```

La primera línea es común a los scripts, y a continuación se carga el módulo necesario para usar el sistema de ficheros (**fs**) y se lee el argumento que se pasa por línea de órdenes, el nombre del fichero que contiene el mensaje del commit (este mensaje estará ahí aunque lo hayamos pasado con **-m** desde la línea de órdenes).

El resto, usando el modo asíncrono que es común en Node, lee el fichero (creando una excepción si hay algún error), lo divide en líneas usando **split** y a continuación comprueba si la primera línea (**lines[0]**) tiene más de 50 caracteres, en cuyo caso sale del proceso con un código de error (1), de esta forma

```
~/repos-git/curso-git<master>$ git commit -am "Añadiendo un montón de cosas al capítulo de 1
[FORMATO] Primera línea > 50 caracteres
```

Si no es así, simplemente deja pasar el mensaje. En este *hook*, curiosamente, no se usa más comando de git que el mensaje almacenado en el fichero; realmente, no es imprescindible usarlo, sólo cuando vayamos a usar información de git en el mismo.

El programa anterior es un ejemplo de cómo se pueden implementar políticas de formato o de cualquier otro tipo sobre un repositorio; sin embargo, la capacidad que tienen es limitada, ya que se aplican sólo sobre los mensajes. En realidad, el que actúen de esa forma es convencional, porque los programas que ejecutan los *ganchos* se diferencian solamente en el momento en el que actúan, no en lo que pueden hacer. Sin embargo, si queremos [implementar una política](#) sobre los nombres de ficheros o el contenido de los mismos, [hay que usar un gancho que actúe cuando se añadan al repositorio](#) como [el siguiente gancho pre-commit](#) escrito en Perl:

```
#!/usr/bin/env perl
my $is_head = `git rev-parse --verify HEAD`;
my $last_commit = $is_head?"HEAD":"4b825dc642cb6eb9a060e54bf8d69288fbee4904";
```

```

my @changes = `git diff-index --name-only $last_commit`;
my %policies = ( no_underscore => qr/_/ );
for my $f (@changes) {
    for my $p ( keys %policies ) {
        if ($f =~ /$policies{$p}/) {
            die "[FORMATO]: $p ";
        }
    }
}
}

```

Este programa es similar a los anteriores, pero para empezar usa un comando *cañería* que se encuentra mucho: `rev-parse`. Las dos primeras órdenes comprueban si nos encontramos en el primer *commit* de un repositorio (en cuyo caso `HEAD` no existe y tendremos que usar el *commit primigenio* que es igual en todos los repositorios de `git`; lo que pretenden esas dos líneas es encontrar el “último commit” para ver cuál es la diferencia con respecto a él. Como vamos a tratar con ficheros que acaban de ser añadidos al índice, usamos `diff-index` en la siguiente línea con un formato que nos devolverá solamente los ficheros cambiados desde el último commit (o desde el primero) sin que nos interese nada más sobre ellos.

La siguiente línea define un *hash* con un nombre y una expresión regular que será la que se tiene que implementar. Si no conoces el lenguaje no te preocupes, pero si lo conoces (ese u otro) es relativamente fácil añadir políticas nuevas, como por ejemplo que no se permitan `.pdfs`, simplemente añadiéndole una línea.

El bucle `for` posterior es que va recorriendo cada uno de los cambios y cada una de las políticas (aunque en este caso habrá una sola) y saldrá con `die` si alguno de los ficheros tiene un nombre incorrecto.

```

~/repos-git/repo-plantilla<master>$ git commit -am "A ver si me deja"
[FORMATO]: no_underscore at .git/hooks/pre-commit line 13.
~/repos-git/repo-plantilla<master>$ git status
# En la rama master
# Cambios para hacer commit:
#   (use «git reset HEAD <archivo>...«para eliminar stage)
#
#   archivo nuevo:    uno_que_no

```

Lo que se puede hacer ahora no es tan trivial: puedes cambiar el fichero de nombre a pelo, pero ya está en el índice, por eso es mejor hacer algo como `git mv` (o `git rm --force`).

Algunos *hooks* útiles explicados

Hay múltiples posts en blogs que [explican diferentes ejemplos de hooks](#) y lo que se puede hacer con ellos. Por ejemplo, en [este conjunto](#) usa programas externos y su

código de salida (\$?) para comprobar si un programa es correcto o no; también usa extensivamente `git stash` para almacenar todos los ficheros que se hayan modificado y luego los recupera con `git stash pop`. [Este, por ejemplo](#), crea una plantilla para usarla en los mensajes de commit, Pero [este gist](#) incluye algunos *hooks* útiles que vamos a ver. Por ejemplo, [el siguiente](#) comprueba si se encuentra la palabra `debugger` en el fichero (es decir, comentarios de depuración):

```
if git-rev-parse --verify HEAD >/dev/null 2>&1; then
    against=HEAD
else
    against=4b825dc642cb6eb9a060e54bf8d69288fbee4904
fi
for FILE in `git diff-index --check --name-status $against -- | cut -c3-` ; do
# Check if the file contains 'debugger'
if [ "grep 'debugger' $FILE" ]
then
echo $FILE ' contains debugger!'
exit 1
fi
done
```

Lo más complicado que tiene este fichero es el uso de filtros del shell (algo que, por otro lado, debería aprenderse); esos filtros hacen algo similar a lo que se ha hecho antes con Perl: recorre el nombre de los ficheros y le aplica el buscador de cadenas, `grep`, buscando la palabra `debugger`; si la encuentra, sale. Quizás está mejor explicado en [la historia original](#)

Contents

Aprende git	1
Pablo Hinojosa y JJ Merelo	1
Licencia	1
Prólogo y agradecimientos	1
Agradecimientos adicionales	1
Introducción	2
Objetivos	2
Software Libre	2
Sistemas de control de Versiones	2

Tipos de Sistemas de Control de Versiones	3
git	4
Abrir una cuenta en Github	5
Uso básico de git	5
Objetivos	5
Instalar git	6
Clientes GUI para Linux, Windows y Mac	15
Empezando a usar git	15
¿Cómo funciona git?	18
Manteniendo nuestro repositorio al día	18
Sincronizando repositorios	21
Comportamiento por defecto de push	24
El archivo .gitignore	25
Solución de problemas con git	27
Objetivos	27
Obtener Ayuda	27
Viendo el historial	27
Borrado de archivos	28
Rehacer un commit	29
Deshacer cambios en un archivo	29
Resolviendo conflictos	29
Retrocediendo al pasado	30
Viendo (y recuperando) archivos antiguos	31
Más usos de git	31
Objetivos	31
Flujos de desarrollo de software (y quizás de otras cosas)	31
Organización de un repositorio de git	32
Flujos de trabajo con git	35
Ramas	37
Los misterios del rebase	44
Quién hizo qué	45

Usando git como los profesionales: GitHub	47
Objetivos	47
Por qué GitHub	47
Repositorios públicos y privados	48
El GitHub <i>social</i>	48
Creando páginas para GitHub pages	50
Cómo usar los hooks	51
Algunos <i>hooks</i> interesantes: sistemas de integración continua . . .	52
Cliente de GitHub	54
Listo para el lanzamiento: publicación en GitHub	55
<i>Hooks</i> : ejecutando código tras una orden git	56
Objetivos	56
Viendo las cañerías: estructura de un repositorio git	56
Paso a paso	59
El nombre de las cosas: refiriéndonos a objetos en git.	59
Comandos de alto y bajo nivel: <i>fontanería</i> y <i>loza</i>	62
Concepto de <i>hooks</i>	70
Programando un <i>hook</i> básico	71
Algunos <i>hooks</i> útiles explicados	74