

# Manual de Git



Miguel Angel Alvarez  
Israel Alcazar  
Joan León



[desarrolloweb.com/manuales/manual-de-git.html](http://desarrolloweb.com/manuales/manual-de-git.html)

# Introducción: Manual de Git

Esta es una compilación de artículos y vídeos destinados a aprender Git, el sistema de control de versiones más utilizado en la actualidad, popularizado en gran medida gracias al servicio de gitHub, el más popular de los hosting para repositorios Git.

Git es una de las herramientas fundamentales para cualquier equipo de desarrollo. "Ningún grupo de programadores debería desarrollar software sin usar un sistema de control de versiones". Existen muchos en el mercado, pero Git se destaca por ser un sistema de control de versiones distribuido que simplifica bastante el trabajo en cualquier tipo de entorno. No queremos decir que Git sea simple, sino que hace fáciles algunas tareas de control de versiones donde otros sistemas resultan bastante más complejos.

Git nos facilita llevar un control de los cambios de cualquier pieza de software, mediante el control de todos los archivos que forman parte de un proyecto. De este modo puedes saber qué estados han tenido tus archivos a lo largo del tiempo y permite que los componentes del equipo de desarrollo sincronicen sus cambios los unos con los otros.

Todos deberíamos saber manejarnos con Git y comenzar no cuesta mucho trabajo, porque las cosas son bastante sencillas. Otra cosa es cuando queramos realizar usos avanzados, pues a veces las cosas se pueden complicar, sobre todo al gestionar ramas y solucionar conflictos.

Estos artículos te sacarán de dudas acerca de las nociones más básicas y muchos de los conceptos avanzados que podrás encontrar en el día a día del uso de Git.

Encuentras este manual online en:

<http://desarrolloweb.com/manuales/manual-de-git.html>

# Autores del manual

Las siguientes personas han participado como autores escribiendo artículos de este manual.

---

## Miguel Angel Alvarez

Fundador de DesarrolloWeb.com y la plataforma de formación online EscuelaIT. Comenzó en el mundo del desarrollo web en el año 1997, transformando su hobby en su trabajo.



---

## Israel Alcázar

Freelance independiente, organizador de la conferencia internacional de Javascript SpainJS



---

## Joan Leon

Front-end developer y diseñador web



---

## Jon Torrado

---

CTO de GamersWalk. Entusiasta del desarrollo web con fuerte conocimiento en Symfony y los flujos de desarrollo actuales.



# Introducción a Git

Git es un sistema extremadamente útil, pero resulta muy confuso cuando uno está iniciando. En estos primeros artículos nuestros objetivos son los de aclarar conceptos necesarios para entender el sistema de control de versiones y algunas partes de su operativa fundamental.

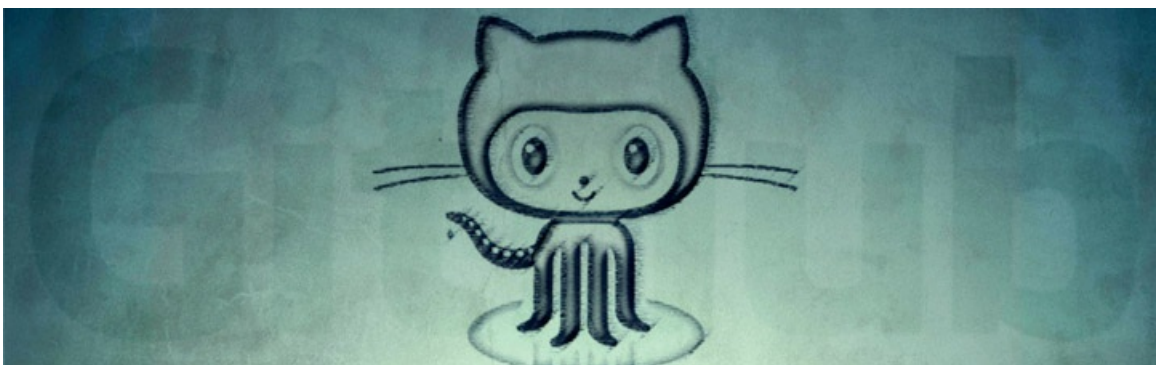
## Introducción a Git y Github

Una introducción a los sistemas de control de versiones, su importancia y sus características a lo largo de la historia. Descripción general de lo que son Git y Github.

Los sistemas de control de versiones son programas que tienen como objetivo controlar los cambios en el desarrollo de cualquier tipo de software, permitiendo conocer el estado actual de un proyecto, los cambios que se le han realizado a cualquiera de sus piezas, las personas que intervinieron en ellos, etc.

Este artículo sirve como introducción a este tipo de herramientas de manera global, pero también para conocer uno de los sistemas de control de versiones existentes en la actualidad que se ha popularizado tremendamente, gracias al sitio Github. Se trata de Git, el sistema de control de versiones más conocido y usado actualmente, que es el motor de Github. Al terminar su lectura entenderás qué es Git y qué es Github, dos cosas distintas que a veces resultan confusas de entender por las personas que están dando sus primeros pasos en el mundo del desarrollo.

Nota: Este texto es una transcripción libre de la primera media hora del [hangout donde presentamos Git y Github](#).



## Necesidad de un control de versiones

---

El control de versiones es una de las tareas fundamentales para la administración de un proyecto de desarrollo de software en general. Surge de la necesidad de mantener y llevar control del código que vamos programando, conservando sus distintos estados. Es absolutamente necesario para el trabajo en equipo, pero resulta útil incluso a desarrolladores independientes.

Aunque trabajemos solos, sabemos más o menos cómo surge la necesidad de gestionar los cambios entre distintas versiones de un mismo código. Prueba de ello es que todos los programadores, más tarde o más temprano, se han visto en la necesidad de tener dos o más copias de un mismo archivo, para no perder su estado anterior cuando vamos a introducir diversos cambios. Para ir solucionando nuestro día a día habremos copiado un fichero, agregándole la fecha o un sufijo como "antiguo". Aunque quizás esta acción nos sirva para salir del paso, no es lo más cómodo ni mucho menos lo más práctico.

En cuanto a equipos de trabajo se refiere, todavía se hace más necesario disponer de un control de versiones. Seguro que la mayoría hemos experimentado las limitaciones y problemas en el flujo de trabajo cuando no se dispone de una herramienta como Git: machacar los cambios en archivos hechos por otros componentes del equipo, incapacidad de comparar de manera rápida dos códigos, para saber los cambios que se introdujeron al pasar de uno a otro, etc.

Además, en todo proyecto surge la necesidad de trabajar en distintas ramas al mismo tiempo, introduciendo cambios a los programas, tanto en la rama de desarrollo como la que tenemos en producción. Teóricamente, las nuevas funcionalidades de tu aplicación las programarás dentro de la rama de desarrollo, pero constantemente tienes que estar resolviendo *bugs*, tanto en la rama de producción como en la de desarrollo.

Nota: Un sistema en producción se refiere a que está disponible para los usuarios finales. O sea, es una versión que está ya puesto en marcha y por lo tanto debería funcionar correctamente. Un sitio web, cuando lo estás creando, está en su etapa de desarrollo y cuando lo liberas en el dominio definitivo y lo pones a disposición de tu cliente, y/o los usuarios de Internet en general, se dice que está en producción.

Para facilitarnos la vida existen sistemas como Git, Subversion, CVS, etc. que sirven para controlar las versiones de un software y que deberían ser una obligatoriedad en cualquier desarrollo. Nos ayudan en muchos ámbitos fundamentales, como podrían ser:

- Comparar el código de un archivo, de modo que podamos ver las diferencias entre versiones
- Restaurar versiones antiguas
- Fusionar cambios entre distintas versiones
- Trabajar con distintas ramas de un proyecto, por ejemplo la de producción y desarrollo

En definitiva, con estos sistemas podemos crear y mantener repositorios de software que conservan todos los estados por el que va pasando la aplicación a lo largo del desarrollo del

---

proyecto. Almacenan también las personas que enviaron los cambios, las ramas de desarrollo que fueron actualizadas o fusionadas, etc. Todo este mundo de utilidades para llevar el control del software resulta complejo en un principio, pero veremos que, a pesar de la complejidad, con Git podremos manejar los procesos de una manera bastante simple.

## Alternativas y variantes de sistemas de control de versiones

Comenzaron a aparecer los sistemas de control del versionado del software allá por los años setenta, aunque al principio eran bastante elementales. Para hacerse una idea, en los primeros sistemas existía una restricción por la que sólo una persona podía estar a la vez tocando el mismo código. Es posible imaginarse que cosas semejantes provocaban retraso en los equipos de trabajo, por ello, a lo largo de los años fueron surgiendo nuevos sistemas de control de versiones, siempre evolucionando con el objetivo de resolver las necesidades de los equipos de desarrollo.

Tenemos principalmente dos tipos de variantes:

**Sistemas centralizados:** En estos sistemas hay un servidor que mantiene el repositorio y en el que cada programador mantiene en local únicamente aquellos archivos con los que está trabajando en un momento dado. Yo necesito conectarme con el servidor donde está el código para poder trabajar y enviar cambios en el software que se está programando. Ese sistema centralizado es el único lugar donde está todo el código del proyecto de manera completa. Subversion o CVS son sistemas de control de versiones centralizados.

**Sistemas distribuidos:** En este tipo de sistemas cada uno de los integrantes del equipo mantiene una copia local del repositorio completo. Al disponer de un repositorio local, puedo hacer *commit* (enviar cambios al sistema de control de versiones) en local, sin necesidad de estar conectado a Internet o cualquier otra red. En cualquier momento y en cualquier sitio donde esté puedo hacer un commit. Es cierto que es local de momento, luego podrás compartirlo con otras personas, pero el hecho de tener un repositorio completo me facilita ser autónomo y poder trabajar en cualquier situación. Git es un sistema de control de versiones distribuido.

Nota: Muchas personas, llegados a este punto, quieren saber dónde se encuentra Github y qué tiene que ver con Git, por lo que intentaremos aclarar algo en este sentido. Git es un sistema de control de versiones distribuido. Con Git hacemos repositorios de software. GitHub es un servicio para hacer *hosting* de repositorios de software que se administra con Git. Digamos que en GitHub mantienes una copia de tus repositorios en la nube, que además puedes hacer disponible para otros desarrolladores. De todos modos, sigue leyendo que más tarde en este mismo artículo te explicaremos todo esto con más detalle.

Tanto sistemas distribuidos como centralizados tienen ventajas e inconvenientes comparativas entre los unos y los otros. Para no hacer muy larga la introducción, cabe mencionar que los sistemas un poco más antiguos como CVS o también Subversion, por sus características son un poco más lentos y pesados para la máquina que hace de servidor central. En los sistemas distribuidos no es necesario que exista un servidor central donde enviar los cambios, pero en caso de existir se requiere menor capacidad de procesamiento y gestión, ya que muchas de las



---

operaciones para la gestión de versiones se hacen en local.

Es cierto que los sistemas de control de versiones distribuidos están más optimizados, principalmente debido al ser sistemas concebidos hace menos tiempo, pero no todo son ventajas. Los sistemas centralizados permiten definir un número de versión en cada una de las etapas de un proyecto, mientras que en los distribuidos cada repositorio local podría tener diferentes números de versión. También en los centralizados existe un mayor control del desarrollo por parte del equipo.

De todos modos, en términos comparativos nos podemos quedar con la mayor ventaja de los sistemas distribuidos frente a los sistemas centralizados: La posibilidad de trabajar en cualquier momento y lugar, gracias a que siempre se mandan cambios al sistema de versionado en local, permite la autonomía en el desarrollo de cada uno de los componentes del equipo y la posibilidad de continuar trabajando aunque el servidor central de versiones del software se haya caído.

## Sobre Git

Como ya hemos dicho, Git es un sistema de control de versiones distribuido. Git fue impulsado por Linus Torvalds y el equipo de desarrollo del Kernel de Linux. Ellos estaban usando otro sistema de control de versiones de código abierto, que ya por aquel entonces era distribuido. Todo iba bien hasta que los gestores de aquel sistema de control de versiones lo convirtieron en un software propietario. Lógicamente, no era compatible estar construyendo un sistema de código abierto, tan representativo como el núcleo de Linux, y estar pagando por usar un sistema de control de versiones propietario. Por ello, el mismo equipo de desarrollo del Kernel de Linux se tomó la tarea de construir desde cero un sistema de versionado de software, también distribuido, que aportase lo mejor de los sistemas existentes hasta el momento.

Así nació Git, un sistema de control de versiones de código abierto, relativamente nuevo que nos ofrece las mejores características en la actualidad, pero sin perder la sencillez y que a partir de entonces no ha parado de crecer y de ser usado por más desarrolladores en el mundo. A los programadores nos ha ayudado a ser más eficientes en nuestro trabajo, ya que ha universalizado las herramientas de control de versiones del software que hasta entonces no estaban tan popularizadas y tan al alcance del común de los desarrolladores.

Git es multiplataforma, por lo que puedes usarlo y crear repositorios locales en todos los sistemas operativos más comunes, Windows, Linux o Mac. Existen multitud de GUIs (Graphical User Interface o Interfaz de Usuario Gráfica) para trabajar con Git a golpe de ratón, no obstante para el aprendizaje se recomienda usarlo con línea de comandos, de modo que puedas dominar el sistema desde su base, en lugar de estar aprendiendo a usar un programa determinado.

Para usar Git debes instalarlo en tu sistema. Hay unas instrucciones distintas dependiendo de tu sistema operativo, pero en realidad es muy sencillo. La página oficial de descargas está en [gitscm.com](https://git-scm.com).

Allí encontrarás para descarga lo que se llama la "Git Bash" es decir, la interfaz de Git por línea de comandos. Tienes que descargar aquella versión adecuada para tu sistema (o si estás en



---

Linux instalarla desde los repositorios dependiendo de tu distribución). Aparte de Windows, Linux o Mac, también hay versión para Solaris. La instalación es muy sencilla, lo que resulta un poco más complejo al principio es el uso de Git. No obstante, en futuros artículos podremos ver cuáles son los primeros pasos con Git y veremos que para realizar unas cuantas funciones básicas el manejo resulta bastante asequible para cualquier persona. Otra cosa es dominar Git, para lo cual te hará falta más tiempo y esfuerzo.

## Sobre Github

Como se ha dicho, Github [github.com](https://github.com) es un servicio para alojamiento de repositorios de software gestionados por el sistema de control de versiones Git. Por tanto, Git es algo más general que nos sirve para controlar el estado de un desarrollo a lo largo del tiempo, mientras que Github es algo más particular: un sitio web que usa Git para ofrecer a la comunidad de desarrolladores repositorios de software. En definitiva, Github es un sitio web pensado para hacer posible el compartir el código de una manera más fácil y al mismo tiempo darle popularidad a la herramienta de control de versiones en sí, que es Git.

Nota: Queremos agradecer a Alejandro Morales de Honduras, estas explicaciones sobre Github en el hangout donde estuvo con nosotros como ponente. Su perfil de Github es [github.com/alejandro](https://github.com/alejandro).

Cabe destacar que Github es un proyecto comercial, a diferencia de la herramienta Git que es un proyecto de código abierto. No es el único sitio en Internet que mantiene ese modelo de negocio, pues existen otros sitios populares como Bitbucket que tienen la misma fórmula. No obstante, aunque Github tenga inversores que inyectan capital y esté movido por la rentabilidad económica, en el fondo es una iniciativa que siempre ha perseguido (y conseguido) el objetivo de hacer más popular el software libre. En ese sentido, en Github es gratuito alojar proyectos *Open Source*, lo que ha posibilitado que el número de proyectos no pare de crecer, y en estos momentos haya varios millones de repositorios y usuarios trabajando con la herramienta.

Pero ojo, para no llevarnos a engaño, al ser Git un sistema de control de versiones distribuido, no necesito Github u otro sitio de alojamiento del código para usar Git. Simplemente con tener Git instalado en mi ordenador, tengo un sistema de control de versiones completo, perfectamente funcional, para hacer todas las operaciones que necesito para el control de versiones. Claro que usar Github nos permite muchas facilidades, sobre todo a la hora de compartir código fuente, incluso con personas de cualquier parte del mundo a las que ni conoces.

Esa facilidad para compartir código del repositorio alojado en la nube con Github y la misma sencillez que nos ofrece el sistema de control de versiones Git para trabajar, ha permitido que muchos proyectos Open Source se hayan pasado a Github como repositorio y a partir de ahí hayan comenzado a recibir muchas más contribuciones en su código.

Quizás te estés preguntando ¿cómo obtienen dinero en Github si alojar proyectos en sus repositorios es gratis? Realmente solo es gratuito alojar proyectos públicos, de código abierto.

El servicio también permite alojar proyectos privados y para ello hay que pagar por una cuenta comercial o un plan de hosting que no es gratuito. Existen planes iniciales, para alojar hasta cinco proyectos privados a partir de 7 dólares por mes, lo que resulta bastante barato. Ese mismo plan es gratuito para los estudiantes universitarios.

Github además se ha convertido en una herramienta para los reclutadores de empleados, que revisan nuestros repositorios de Github para saber en qué proyectos contribuimos y qué aportaciones hemos realizado. Por ello, hoy resulta importante para los programadores no solo estar en Github sino además mantener un perfil activo.

## Conclusión

Esperamos que esta introducción al mundo de los sistemas de control de versiones te haya interesado y te haya ayudado a aclarar diversos conceptos. En futuros artículos os explicaremos otros detalles que tienen que ver con el uso de estos sistemas para versionar nuestro software. Si quieres acceder a formación especializada no dejes de consultar el [Curso de Git completo, intensivo y profesional que comenzamos en EscuelaIT](https://desarrolloweb.com/articulos/introduccion-git-github.html).

Este artículo es obra de *Israel Alcázar*  
Fue publicado / actualizado en 03/06/2014  
Disponible online en <https://desarrolloweb.com/articulos/introduccion-git-github.html>

## Curso de Git

El curso de Git comienza con esta primera clase, en la que damos una introducción a los comandos más importantes de Git y explicamos en qué consiste el servicio de GitHub.



Vamos a conocer los comandos principales de Git por medio de una práctica en la que trataremos de ilustrar cuál es el uso de la herramienta para gestionar las versiones de un proyecto. El objetivo es ver cómo se crea un repositorio, cómo se actualizan los archivos del repositorio y cómo podemos volver hacia atrás para recuperar el estado de los ficheros en versiones anteriores de desarrollo.

Posteriormente, veremos cómo se puede crear en Github un repositorio y cómo podemos

---

mandar los cambios de nuestro repositorio Git en local a Github, todo ello en el espacio de poco más de una hora de clase.

En el vídeo trabajamos siempre con el terminal, introduciendo los comandos a mano. Aunque existen diversos programas con interfaz gráfica que también te pueden facilitar las cosas, la única manera de explotar todas las posibilidades de Git, es a través de la línea de comando.

## Instalando Git

En el vídeo dedicamos unos momentos a mencionar las principales vías de instalación de Git en un ordenador en local, para disponer de la terminal donde podremos lanzar los comandos de Git. Es muy fácil de instalar, obteniendo la descarga desde su página oficial:

<http://git-scm.com/>

En Windows la descarga es un ejecutable que puedes instalar normalmente. En Mac también puedes instalarlo a través del archivo que descargas, así como bajarlo usando Home Brew. En Linux lo encuentras en todos los repositorios de las principales distribuciones, cuyos paquetes instalas con Apt-get, Yum, etc.

Nota: Con la versión de Git para Windows no puedes ejecutar los comandos Git directamente desde la consola de Windows, sino que tienes que usar el Git Bash, que obtendrás al instalar la herramienta.

No tenemos que asustarnos por usar la consola de comandos, pero si lo deseamos podemos usar herramientas visuales, que hay un montón, así como también se puede integrar Git en la mayoría de los IDE e incluso por medio de *plugins* en editores de código como Sublime Text.

## Creando un repositorio en local

Es tan sencillo como irse a un directorio de nuestra máquina desde la línea de comandos o desde Git Bash, si estás en Windows. Empezamos con un directorio vacío, ya que se supone que estamos comenzando nuestro proyecto y entonces podemos lanzar el comando:

```
git init
```

A partir de ese momento tienes todo un repositorio de software completo en tu máquina del proyecto que tenías en ese directorio. Puedes crear los archivos que quieras en el directorio (tu proyecto) y luego puedes enviarlos al repositorio haciendo un "commit".

Nota: La gran diferencia de Git con respecto a otros repositorios es que está distribuido. Todos los repositorios en cada máquina están completos, de modo que esa máquina podría ser perfectamente autónoma y desarrollar código en cualquier parte del proyecto. Por eso yo puedo hacer commit al repositorio local, independientemente de si el ordenador está o no conectado a una red.

---

En el vídeo conocemos varios comandos adicionales como "status", "log", etc.

## Añadir ficheros al repositorio

Antes de enviar cambios al repositorio, lo que se conoce como hacer commit, tienes que enviar los ficheros a un área intermedia llamada "staging area" en donde se almacenan los archivos que serían actualizados en el próximo commit. Éste es uno de los conceptos esenciales para aprender Git y para mandar los ficheros a este área tenemos que añadirlos al repositorio.

```
git add .
```

Con ese comando paso los ficheros al "staging area". En este caso "git add ." permite enviar todos los cambios en el working directory, incluyendo archivos nuevos y archivos que hayan sido modificados.

Una vez añadidos los archivos al staging area, ahora podremos hacer el commit sobre esos ficheros.

```
git commit -m "este es mi primer commit"
```

Ese comando permite enviar los archivos a una especie de base de datos donde puedes colocar los archivos cuyo estado queremos siempre memorizar.

Aquí en el vídeo hablamos brevemente de las ramas y de cómo se gestionan en Git y mencionamos que la rama principal, que se crea al iniciarse el repositorio; se llama "master". También vemos cómo localizar los commits a través de su identificador, lo que nos sirve para poder movernos entre las actualizaciones de software.

Cada vez que se modifica un fichero, se tiene que agregar al área intermedia con el comando "add" y luego hacer el commit para que se actualice el repositorio con los archivos nuevos o actualizados. Eso significa que en teoría cada vez que cambias un archivo tendrías que añadirlo al Staging area para poder hacer el commit, aunque más adelante veremos que hay maneras de automatizar las tareas un poco más y, mediante atajos, poder resumir el paso de hacer el "add" cuando se ha cambiado el código de un fichero.

## Deshacer cambios

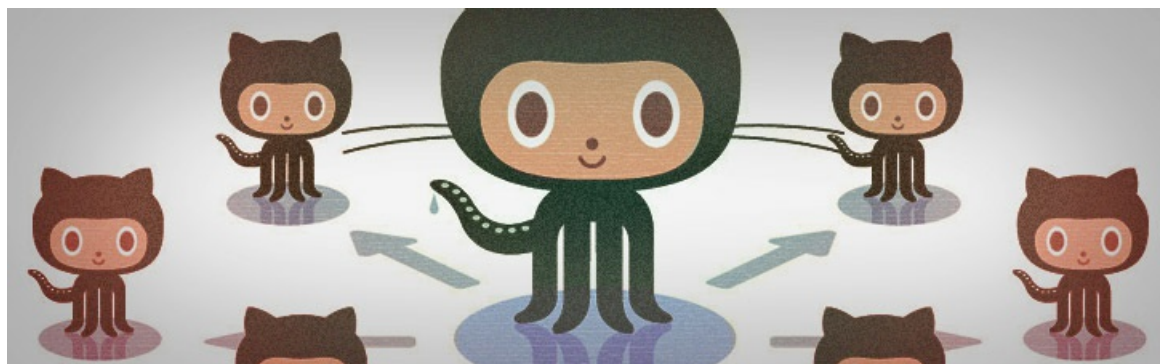
En cualquier momento puedo deshacer cambios que haya realizado en los archivos, recuperando versiones anteriores que podamos tener en el repositorio. Podría descartar los cambios que tengo en staging area.

```
git checkout -- nombre_fichero
```

En Git puedes tener código principalmente en tres lugares, el directorio de trabajo, en el Staging area o en una versión que tengas en un commit antiguo de ese repositorio. Existen diferentes comandos para hacer todas estas cosas, dependiendo del lugar de Git donde tengas el código que quieres recuperar. Pero lo verdaderamente útil y potente que tenemos con el control de versiones es poder recuperar archivos realizados commit anteriores, en el último commit, pero también en otros que tengas más antiguos.

Existen muchos comandos para recuperar código anterior como "reset", en el vídeo se ven varios ejemplos, se explican también como fusionar commits y muchas otras cosas interesantes.

## Github



En el vídeo se ve después cómo trabajar con Github, mostrando las posibilidades de este servicio de almacenamiento de repositorios en remoto. Vemos cómo se crea un repositorio en Git y cómo se puede clonar en nuestro sistema local.

Para trabajar con Github tenemos algunas características importantes como las claves de seguridad SSH, que tenemos que generar en local y llevarnos a Github para dar un nivel de seguridad adicional a las actualizaciones contra los repositorios.

En el vídeo se muestra cómo se crea el repositorio en Github y cómo se clona en local por medio del comando "clone". Una vez clonado se muestra cómo se actualizan los archivos en local, se añaden al área intermedia con "add" y se hace el "commit" al repositorio en local. Por último se envían los cambios al servidor remoto con "push".

Todo esto no es un paso inmediato, pero en el vídeo se explica fantásticamente.

## Curso de Git

Todo el material que se ve en el vídeo no es más que la punta del iceberg, es decir, una pequeña muestra de entre todas las enormes posibilidades de Git. Esta clase de una hora seguro que te servirá para dar los primeros pasos con Git, pero hay mucho más detrás de esta herramienta. Hay todavía muchas cosas que ver y conceptos fundamentales como hacer "fork", "pull", "merge", "checkout", "fetch", "branch"...

Todo ese material lo veremos con detalle en el Curso de Git de EscuelaIT, a lo largo de diez horas de clases en directo que se celebrarán durante algo más de una semana. Este vídeo es



solo una pequeña muestra de los contenidos que se verán allí con todo detalle.

Puedes obtener más información desde la [página del Curso de Git](https://desarrolloweb.com/articulos/curso-git.html).

De momento te dejamos con el vídeo que esperamos ¡puedas disfrutar mucho!

Para ver este vídeo es necesario visitar el artículo original en:  
<https://desarrolloweb.com/articulos/curso-git.html>

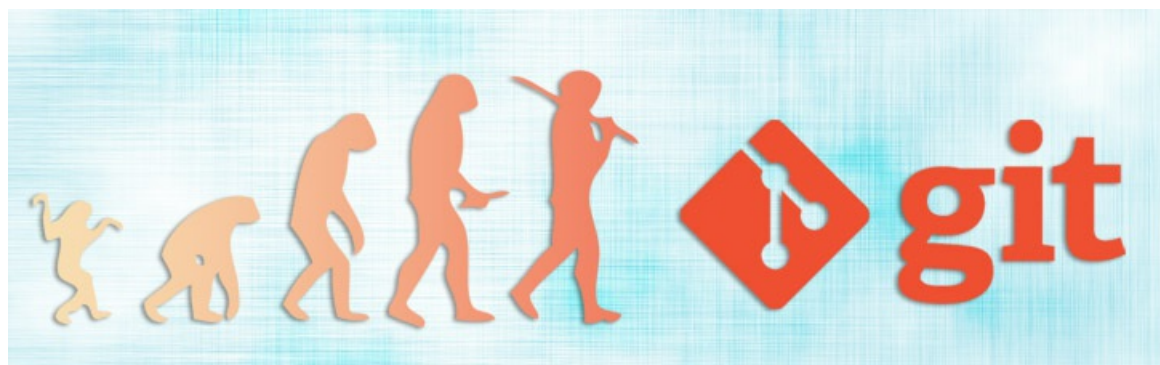
Este artículo es obra de *Miguel Angel Alvarez*  
Fue publicado / actualizado en 14/10/2020  
Disponible online en <https://desarrolloweb.com/articulos/curso-git.html>

## Entiende, instala y configura Git

Este artículo inicia una serie dedicada a dar los primeros pasos con Git, por la práctica, explicando algunos conceptos iniciales y la manera de instalar y configurar Git en tu ordenador local.

En este artículo te explicamos los primeros pasos con Git y GitHub, a partir de la transcripción de unas clases en directo que fueron emitidas en DesarrolloWeb.com antes del curso de Git que se impartió en [EscuelaIT](https://desarrolloweb.com).

Asimismo, encontrarás los vídeos de cada uno de los bloques de la clase junto con un resumen del vídeo a modo de transcripción de lo que se contó en esta clase. Son ideales para comenzar con el sistema de control de versiones Git. En este caso para entender unos conceptos básicos y crear tu entorno para poder trabajar con Git.



## Conceptos rápidos sobre Git y GitHub

Antes de meternos a fondo con Git y comenzar a practicar, nos interesa cubrir algunas de las típicas dudas que se tiene cuando se comienza con este sistema de control de versiones. Se trata de ver algunas claves rápidas sobre las características de Git, el flujo de trabajo y algunas

---

diferencias con GitHub.

En Git, cada desarrollador dispone de un repositorio completo instalado en su máquina; es algo inherente a los sistemas de control de versiones distribuidos, como vimos en el artículo sobre [Qué son Git y GitHub](#). Es condición indispensable. Todos los cambios de nuestros archivos a lo largo del desarrollo los vamos a tener en local. Opcionalmente, esos cambios los enviaremos a repositorios remotos, como puede ser GitHub o cualquier otro.

Esto quiere decir que mi máquina tendrá todo lo que necesito para trabajar. Tendremos una copia del repositorio completo, cada uno de los archivos de todo el proyecto. Con el repositorio Git en local, luego decidiré a qué otros servidores o máquinas mando mis cambios.

GitHub es un *hosting* de repositorios Git, por tanto, el uso que le daremos es como repositorio remoto. Pero debe quedar claro que primero debo tener los cambios en el repositorio local y luego podré "empujar" esos cambios al repositorio remoto.

Nota: Como convención, usaremos el verbo "empujar" refiriéndonos a la acción que se hace con el comando "push" de Git, que veremos dentro de poco. De momento, para que nos entienda todo el mundo y hacernos una idea mejor del funcionamiento de Git, podemos limitarnos a usar términos llanos del español, ya habrá momento de ponerse técnicos.

Por tanto, ya se ve la primera diferencia entre Git y GitHub: Git es la tecnología para hacer el control de versiones y GitHub simplemente es un *hosting* de repositorios Git, con una interfaz web que nos ofrece algunas utilidades basadas en el propio control de versiones Git.

En GitHub puedo tener repositorios diversos y si quiero trabajar con alguno de ellos, primero debo tenerlo en local. Ojo, no necesitamos tener todos los repositorios que has publicado en GitHub en local, solo aquellos con los que vayamos a trabajar. En el momento que los tenemos en local podremos hacer cambios, almacenarlos en nuestro repositorio local y cuando lo juzguemos oportuno, enviarlo (empujar, *push*) a tantos servidores o repositorios remotos como queramos.

Nota: Ten en cuenta también que al referirnos a GitHub queremos que se entienda como un repositorio remoto en general. Es decir, realmente lo que digamos de GitHub en líneas generales sirve para entender otros servicios de *hosting* de repositorios Git como Bitbucket.

Para concluir y tener más claro el flujo de trabajo con un control de versiones Git, imaginar un equipo de trabajo con varios componentes. Todos y cada uno de los desarrolladores deberán tener una copia completa de todo el repositorio de software que se está desarrollando. Luego el equipo decidirá a qué servidor con un repositorio remoto quiere enviar los cambios.

Cada desarrollador podrá enviar los cambios que tiene en local hacia el repositorio remoto cuando quiera y ese repositorio remoto lo usarán todos los componentes del equipo para sincronizarse y tener la versión más nueva del código en cada momento que lo deseen.



Este esquema podemos considerarlo como la base del trabajo con Git, aunque luego en la práctica la cosa se complica porque habrá conflictos y tendrás que resolverlos.

Para ver este vídeo es necesario visitar el artículo original en:  
<https://desarrolloweb.com/articulos/entiende-instala-configura-git.html>

## Instalar Git

Tener Git instalado en local es condición indispensable para trabajar con el sistema de control de versiones. El proceso para instalar Git es bien sencillo porque no difiere de la instalación de cualquier otro software que hayas hecho.

Te tienes que descargar la versión de tu sistema operativo en la página oficial (o si usas Linux lo bajarás de los repositorios de software que usas habitualmente en tu distribución).

[gitscm.com](https://git-scm.com)

Lo instalas como cualquier otro software. Si estás en Windows tendrás un asistente al que harás "siguiente, siguiente" hasta acabar el proceso. Puedes ver este vídeo que aclara algunos puntos sobre la instalación.

Para ver este vídeo es necesario visitar el artículo original en:  
<https://desarrolloweb.com/articulos/entiende-instala-configura-git.html>

## Para usuarios Windows, ¿Git Bash?

El único sitio donde puedes tener dudas es en el paso que te dice si quieres instalarlo como comando en la línea de comandos de tu consola o si simplemente quieres el "git bash".

Si lo instalas en la propia consola del Windows, la única ventaja es que lo tendrás disponible desde la ventana de línea de comandos de Windows y podrás hacer desde allí los comandos de Git. Si lo instalas solo en Git Bash no habrá más problemas, solo que cuando quieras usar Git tendrás que abrir la consola específica de Git que ellos llaman "git bash".

La manera más sencilla de saber si Git está instalado en tu sistema es a través del comando:

```
git version
```

Esto te mostrará en la pantalla el número de la versión de Git que tengas instalada. Si en Windows instalaste Git en consola (la ventana de línea de comandos), observarás que en cualquier consola de comandos de Windows tienes disponible Git. Si lo instalaste únicamente

---

en el Git Bash, no tendrás ese comando y Git no está disponible desde la consola de Windows.

Git Bash es la línea de comandos de Git para Windows, que además te permite lanzar comandos de Linux básicos, "ls -l" para listar archivos, "mkdir" para crear directorios, etc. Es la opción más común para usar Git en Windows.

Es indiferente si instalas Git en la línea de comandos del Windows o si lo instalas solamente en el Git Bash. Simplemente escoge la que te sea más cómoda.

Para ver este vídeo es necesario visitar el artículo original en:  
<https://desarrolloweb.com/articulos/entiende-instala-configura-git.html>

## Primera configuración de Git, primeros comandos que debes lanzar

Antes que nada, inmediatamente después de instalar Git, lo primero que deberías hacer es lanzar un par de comandos de configuración.

```
git config global user.name "Tu nombre aqui"  
git config global user.email "tu_email_aquí@example.com"
```

Con estos comandos indicas tu nombre de usuario (usas tu nombre y apellidos generalmente) y el *email*. Esta configuración sirve para que cuando hagas *commits* en el repositorio local, éstos se almacenen con la referencia a ti mismo, meramente informativa. Gracias a ello, más adelante cuando obtengas información de los cambios realizados en el los archivos del "repo" local, te va a aparecer como responsable de esos cambios a este usuario y correo que has indicado.

Para ver este vídeo es necesario visitar el artículo original en:  
<https://desarrolloweb.com/articulos/entiende-instala-configura-git.html>

Este artículo es obra de *Israel Alcázar*  
Fue publicado / actualizado en 16/06/2014  
Disponible online en <https://desarrolloweb.com/articulos/entiende-instala-configura-git.html>

# Operativa básica con Git

Ahora vamos a explicar los procedimientos básicos en la operativa del día a día con Git. Acciones como añadir código al repositorio, compartir el código a través de servidores remotos donde nos permitan sincronizar el proyecto entre los integrantes de un equipo y evitar que Git tenga en cuenta archivos que no deberían estar en el proyecto.

## Iniciar repositorio Git y primer commit

Operaciones iniciales para comenzar a trabajar con repositorios Git. Creamos un primer proyecto, inicializamos Git y hacemos el primer commit.

En los artículos anteriores de Git hemos visto los pasos iniciales que se deben realizar para poder usar Git, o mejor dicho, "comenzar a usarlo". Sin embargo, aún no hemos empezado a trabajar con repositorios y a controlar el estado de nuestro software a través de sus diversas versiones.

Si has llegado aquí y no sabes qué es este sistema de control de versiones, te recomendamos acceder al artículo donde explicamos [cómo instalar y configurar Git](#). Si ya has realizado esos primeros pasos, sigue leyendo.

A continuación te explicamos cómo crear tu primer repositorio de software y cómo enviar archivos al repositorio para que comiencen a ser rastreados por Git y que el sistema de control de versiones almacene sus diversos estados a lo largo del tiempo.



## Abrir una consola para trabajar con Git

Una vez instalado Git, vamos a trabajar con la consola de comandos para aprender de verdad a usar este sistema. Existen diversas aplicaciones de interfaz gráfica para trabajar con Git, pero no son realmente las mejores opciones para aprender a usar este sistema. Es por ello que te recomendamos aprender por línea de comandos, tal como te explicamos en estos artículos.

---

Si estamos en Windows, podremos abrir el "Git Bash" (ver artículo mencionado anteriormente) y si estamos en Linux / Mac, simplemente tendremos que abrir un terminal del sistema operativo.

Una vez dentro de tu consola de comandos, vamos a crear un directorio donde vamos a dejar todos nuestros repositorios. Aunque realmente puedes tenerlos dispersos en las carpetas que desees, eso es indiferente.

Nota: Si no sabes qué es un repositorio, podemos aclarar que es simplemente como una estantería de tu biblioteca donde guardas un software. Es un lugar, una carpeta, donde almacenas todos los archivos de un proyecto y cuando trabajas con Git tienes la posibilidad de tener no solo el estado actual de tus ficheros, sino el estado por el que han pasado en todas sus versiones a lo largo de la vida de ese software.

## Formas de comenzar a trabajar con Git

Para trabajar con Git o Github (sabemos las diferencias porque ya lo explicamos en el [artículo de conceptos iniciales](#)) tenemos dos formas de trabajar:

2) Clonar un repositorio de Github (u otro hosting de repositorios) para traernos a local el repositorio completo y empezar a trabajar con ese proyecto.

Vamos a elegir de momento la opción 1) que nos permitirá comenzar desde cero y con la que podremos apreciar mejor cuáles son las operaciones básicas con Git. En este sentido, cualquier operación que realizas con Git tiene que comenzar mediante el trabajo en local, por lo que tienes que comenzar por crear el repositorio en tu propia máquina. Incluso si tus objetivos son simplemente subir ese repositorio a Github para que otras personas lo puedan acceder a través del *hosting* remoto de repositorios, tienes que comenzar trabajando en local.

## Crear una carpeta para tu proyecto y colocar archivos

Entonces, estando en una carpeta de tu ordenador donde hemos dicho que vamos a crear todos nuestros repositorios, puedes crear una carpeta específica para tu primer proyecto con Git. La carpeta de tu proyecto la puedes crear con el explorador de archivos o si quieres por línea de comandos, es indiferente.

Una vez creada tu carpeta puedes crear archivos dentro. Como estamos hablando de desarrollo de software, los archivos que meterás dentro de tu carpeta contendrán el código de tu aplicación, página web, etc.

Nota: Para crear los archivos, en el flujo de trabajo común de un desarrollador usarás tu editor de código preferido: Eclipse, Sublime Text, Komodo, Notepad++, Gedit, PhpStorm, etc. Como prefieras. También, por supuesto, puedes crear tu archivo por línea de comandos y editarlo si estás en Linux con el conocido Vim o cualquier otro editor de interfaz solo texto.

---

## Inicializar el repositorio Git

Para crear tu repositorio Git, donde monitorizamos los archivos de un proyecto, tienes que realizar una operación previa. Es la inicialización del repositorio Git que queremos crear en la carpeta de tu proyecto y es una operación que deberás realizar una única vez para este proyecto.

Nota: Cada proyecto lo tendrás en una carpeta distinta y será un repositorio independiente. Esta operación de inicialización de tu repositorio, por tanto, la tendrás que realizar una única vez para cada proyecto que quieras controlar sus versiones por medio de Git.

Así pues, antes que nada (antes de enviar cualquier archivo al repositorio), creo el repositorio Git con el comando "git init".

```
git init
```

Una vez has inicializado el repositorio, podrás observar que se ha creado una carpeta en tu proyecto llamada ".git". Si ves esa carpeta en tu proyecto es que el repositorio se ha inicializado correctamente y que ya tienes convertida esa carpeta en un repositorio de software Git.

Nota: Los archivos o carpetas en Linux / Mac que comienzan por un punto están ocultos en el sistema de archivos. Para verlos tendrás que ejecutar el comando "ls -la".

En la carpeta .git es donde se va a guardar toda la información relativa al repositorio. Resultará obvio, pero conviene decir que nunca se debe borrar esa carpeta y tampoco deberíamos acceder de forma directa a sus contenidos, ya que nos vamos a comunicar siempre con el repositorio por medio de comandos.

## Guardar los archivos en el repositorio (commit)

Una vez que crees tus primeros archivos, puedes comenzar a trabajar con Git, enviando esos ficheros al repositorio. Aunque los archivos estén en la carpeta de tu proyecto y hayas iniciado el repositorio Git previamente, el sistema de control de versiones no los está monitorizando, por lo que a nivel de tu repositorio Git todavía es como si no estuvieran.

A esa acción de guardar los archivos en el repositorio se llama, en la jerga de Git, hacer un "commit". En este caso el commit lo estás haciendo en local, porque los archivos los estás enviando a tu repositorio local que tienes en tu máquina.

Un commit en Git se hace mediante dos pasos.

1) Tengo que añadir el fichero a una zona intermedia temporal que se llama "Zona de Index" o "Staging Area" que es una zona que utiliza Git donde se van guardando los ficheros que posteriormente luego se van a hacer un commit.

Cualquier archivo que quieras mandar a la zona de index lo haces con el comando "add".

```
git add nombre-del-fichero
```

Una vez añadido podrías ver que realmente tienes ese fichero en el "staging area", mediante el comando "status".

```
git status
```

Verás que, entre las cosas que te aparecen como salida de ese comando, te dice que tienes tu fichero añadido como "new file". Además te dice que estos cambios están preparados para hacer commit.

2) Tengo que enviar los archivos de la zona de Index al repositorio, lo que se denomina el commit propiamente dicho. Lo haces con el siguiente comando:

```
git commit -m "mensaje para el commit"
```

Con esto ya tenemos nuestro primer archivo dentro del repositorio. A partir de aquí podremos mantener y controlar los cambios que se hagan sobre este archivo (u otros que hayamos incluido por medio del commit).

Si haces de nuevo un comando "git status" podrás comprobar que no hay nada para enviar al repositorio. Eso quiere decir que la zona de Index está limpia y que todos los cambios están enviados a Git.

Hasta este punto hemos podido aprender a crear nuestro repositorio y enviar los primeros cambios por medio de la operación de commit. Ahora puedes probarlo en tu sistema para comenzar a experimentar Git. En las siguientes entregas continuaremos explorando el flujo de trabajo con Git, y mejorando esta primera aproximación con nuevos comandos que te permitan hacer mas cosas y facilitarte el trabajo resumiendo las operaciones ya comentadas en menos pasos.

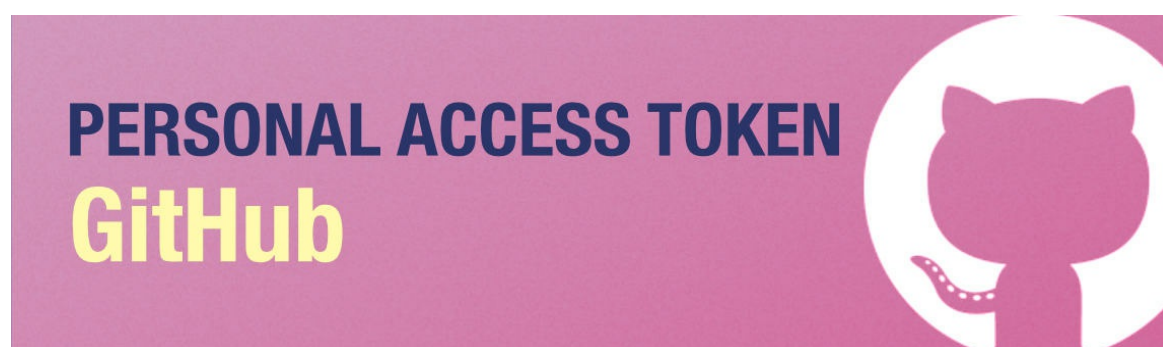
A continuación puedes ver un vídeo que hemos resumido en este artículo y donde podrás ver todas estas operaciones en directo y explicadas en directo y paso por paso.

Para ver este vídeo es necesario visitar el artículo original en:  
<https://desarrolloweb.com/articulos/iniciar-repositorio-git-primer-commit.html>

Este artículo es obra de *Israel Alcázar*  
Fue publicado / actualizado en 08/07/2014  
Disponible online en <https://desarrolloweb.com/articulos/iniciar-repositorio-git-primer-commit.html>

## Autenticar en GitHub con Personal Access Token

Cómo crear un Personal Access Token en el sitio web de GitHub para poder autenticarte correctamente, de modo que puedas realizar todo tipo de operativas, como clonar repositorios o enviar cambios a GitHub desde local a remoto.



En GitHub han declarado recientemente como obsoleta la autenticación por medio de usuario y clave y ahora los desarrolladores debemos autenticarnos a través de lo que llaman el Personal Access Token. Esta acción debes hacerla para poder conectarte y realizar cualquier tipo de operativa con el servicio de GitHub.

En el [Manual de Git](#) casi no hemos hablado de GitHub todavía, pero enseguida nos pondremos con operativas relacionadas con este servicio de Hosting de repositorios. No obstante, aunque parezca que estamos saltando algunos pasos previos al abordar este tema ahora, es importante explicarlo, para dejar claro cómo realizaremos el acceso a GitHub desde el terminal de línea de comandos, o cualquier otro programa cuando estemos en local. Por favor, si hay alguna cosa que no entiendes todavía, no te preocupes porque lo veremos todo a su debido momento en el manual.

## Por qué necesitamos el Personal Access Token

Anteriormente, cuando querías realizar cualquier operación en GitHub desde local y esa operación requería de acceso autenticado, ya sea mediante la línea de comandos o un programa de interfaz gráfica, podrías simplemente autenticarte usando el mismo usuario y contraseña que utilizas en el sitio web de GitHub. Ahora no es posible y tendrás que usar el mencionado Personal Access Token. Si todavía no has configurado tu equipo para adaptarte a este cambio o bien eres nuevo en GitHub, te explicamos en este artículo cómo has de hacer actualmente.



---

Este procedimiento lo tendrás que hacer en todo lugar donde necesites una clave para poder utilizar GitHub desde fuera del sitio web. Por ejemplo clonar un repositorio privado de GitHub en local, enviar cambios a GitHub con Push, traerte cambios con Pull, etc.

### Cómo crear un Personal Access Token (PAT)

Este token se crea desde el sitio web de GitHub. El procedimiento es muy sencillo. Una vez logueado con tu usuario y contraseña en el sitio web de GitHub, accedemos a la opción "Settings" y luego a "Developer settings" y por último a "Personal access tokens".

La opción de "Settings" la encuentras en el menú desplegable de tu avatar.



Signed in as  
**midesweb**



Set status

Your profile

Your repositories

Your codespaces

Your organizations

Your projects

Your stars

Your gists

Upgrade

Feature preview



Help

**Settings**

Sign out

## Explore repositories

### laravel/browser-kit

Provides back  
BrowserKit tes  
release.

● PHP ☆ 4

### coderello/lara

Language  
into 40+ lang  
language or u

● PHP ☆ 2

### Askedio/larav

Cascade Dele  
Laravel SoftD

● PHP ☆ 6

Explore more

Las siguientes opciones las accedes mediante el navegador de la izquierda. Entonces accederás a una página donde puedes administrar tus Personal access tokens.

Aquí vamos a crear un nuevo token con el botón "Generate new token". Entonces aparecerá un formulario que debes rellenar, indicando el nombre del token (para saber en qué lo vas a utilizar) y otros detalles como la expiración del token y los "scopes".

## New personal access token

Personal access tokens function like ordinary OAuth access tokens. They can be used instead of a password for Git over HTTPS, or can be used to [authenticate to the API over Basic Authentication](#).

### Note

Token para usar en local

What's this token for?

### Expiration \*

30 days

The token will expire on Fri, Aug 27 2021

### Select scopes

Scopes define the access for personal tokens. [Read more about OAuth scopes](#).

- |  |                                      |
|--|--------------------------------------|
| <input type="checkbox"/> <b>repo</b>     | Full control of private repositories |
| <input type="checkbox"/> repo:status     | Access commit status                 |
| <input type="checkbox"/> repo_deployment | Access deployment status             |
| <input type="checkbox"/> public_repo     | Access public repositories           |
| <input type="checkbox"/> repo:invite     | Access repository invitations        |
| <input type="checkbox"/> security_events | Read and write security events       |

Los scopes son simplemente las operaciones que estarán o no permitidas desde este token. Lo más común es que des permisos para acceder a los repositorios remotos alojados en GitHub, pero existen otras operativas que puedes hacer o no tú y que podrías permitir o no con este token que vas a crear.

## Copiar el PAT en un lugar seguro

Una vez generado el token aparecerá en la página de GitHub. Puedes copiarlo y ponerlo en un lugar seguro. Ten en cuenta que es como si fuera una clave de usuario! por lo que debes cerciorarte que solamente tú tengas acceso a este token.

## Personal access tokens

[Generate new token](#)[Revoke all](#)

Tokens you have generated that can be used to access the [GitHub API](#).

Make sure to copy your personal access token now. You won't be able to see it again!

✓ ghp\_1234567890abcdefghijklmnopqrstuvwxyz2

[Delete](#)

Personal access tokens function like ordinary OAuth access tokens. They can be used instead of a password for Git over HTTPS, or can be used to [authenticate to the API over Basic Authentication](#).

Por supuesto, no lo debes pegar en el código de ninguna aplicación, para no dejarlo visible para otras personas!! o tus accesos a GitHub podrían verse comprometidos.

Otro detalle importante es que este token solamente aparecerá una vez en la página de GitHub. Si lo necesitas recuperar porque lo has perdido simplemente no podrás hacerlo. Tendrás que generar uno nuevo.

## Cómo usar el Personal Access Token de GitHub

El mecanismo para usar el PAT es tan sencillo como el que anteriormente hacíamos con la clave del sitio web.

Simplemente, cuando la operación de GitHub te pida la clave de tu usuario, en lugar de la clave colocarás el token.

Por ejemplo, en la siguiente imagen vemos el comando de clonado de un repositorio. Cuando nos piden la clave, simplemente colocamos toda la cadena del token que hemos copiado en GitHub. Con esto habremos autenticado con el token correctamente y podremos acceder a los servicios de GitHub.

```
midesweb@MacBook-Pro ~/sites/sandbox$ git clone https://github.com/midesweb/preparar-php-apps-prof.git
Clonando en 'preparar-php-apps-prof'...
Username for 'https://github.com': midesweb
Password for 'https://midesweb@github.com':
```

No intentes clonar ese repositorio con el comando que ves en la imagen, ya que es un repositorio privado y te pedirá la clave. Solamente yo tengo permisos para clonarlo, dado que es privado. Si quieres aprender esta operativa tienes un artículo que te enseña a [clonar repositorios de GitHub](#).

## Cómo borrar credenciales previamente almacenadas en GitHub en MacOS

Para los usuarios de Mac, es posible que necesites eliminar credenciales previamente almacenadas en tu equipo. Para ello tienes que abrir una aplicación llamada "Keychain Access".

En ella buscas por "GitHub" y eliminas las credenciales existentes.



Después de realizar este paso, al intentar hacer alguna operativa de nuevo con GitHub se pedirá tu usuario y contraseña, e introducirás tu usuario y el token, tal como hemos indicado anteriormente.

## Cacheo del Personal Access Token

El cacheo del Access Token lo puedes realizar en cualquier sistema operativo por medio de Git. En MacOS se hace automáticamente con la aplicación de llaves. Para los usuarios de otros sistemas operativos como Linux y Windows en la documentación de GitHub encontramos algunas otras [indicaciones interesantes](#).

En Windows te indican que tienes que lanzar el siguiente comando:

```
git config --global credential.helper wincred
```

En Linux te informan de dos comandos útiles. Este te permite indicar que las credenciales se cacheen:

```
git config --global credential.helper cache
```

Y este otro comando te permite que las credenciales permanezcan cacheadas por el tiempo que tú estimes conveniente. El tiempo se indica en segundos. Por ejemplo, para cachear las credenciales por 3 horas lanzas el siguiente comando.

```
git config --global credential.helper 'cache --timeout=10800'
```

Eso es todo! espero que con estas indicaciones puedas realizar toda la configuración de GitHub para el acceso desde el terminal en local y poder administrar tus repositorios remotos.

Este artículo es obra de *Miguel Angel Alvarez*  
Fue publicado / actualizado en 28/07/2021  
Disponible online en <https://desarrolloweb.com/articulos/autenticar-github-personal-access-token>

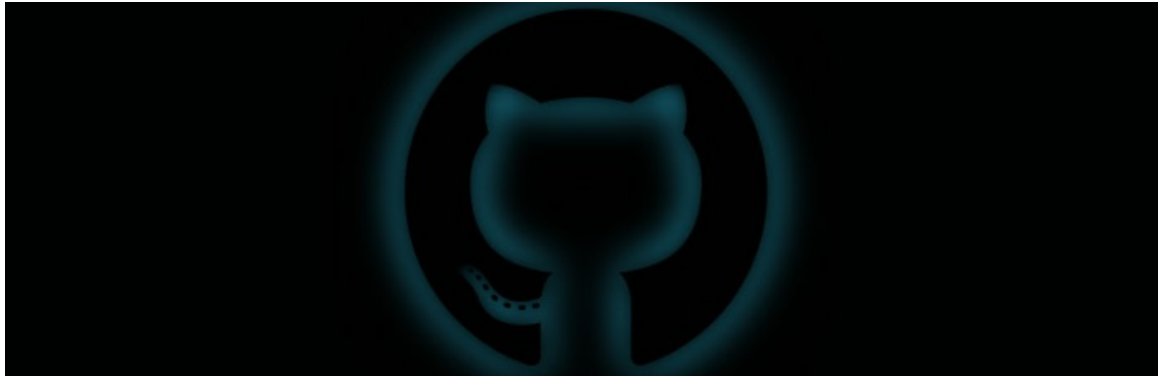
## Crear un repositorio Git en Github y subir el código

Explicaciones sobre el proceso de creación de un nuevo repositorio Git y cómo subir el código a un repositorio creado en Github.

En este artículo vamos a abordar una operativa básica del día a día del desarrollador y el trabajo con el sistema de control de versiones Git. Básicamente explicaremos los pasos para crear un nuevo repositorio en Github con el código de un proyecto.

Este artículo está motivado por una duda respondida a un compañero de la comunidad que tenía dudas con el sistema de control de versiones. La duda era la siguiente:

Me han pedido que suba un proyecto, el código, a Github en un nuevo repositorio que tengo que crear. Mi problema es que desconozco esta operativa y por miedo a parecer inexperto prefiero preguntaros a vosotros. Imagino que esto se hace con Git, pero no se muy bien como hacerlo... si tengo que hacerlo publico, privado, si subo el zip o como lo hago para que el código aparezca en Github.



Nuestra respuesta, viene a continuación, explicando toda la operativa paso a paso. De todos modos, esta respuesta te quedaría mucho más clara si echas un vistazo al [Manual de Git de DesarrolloWeb.com](#), o bien el [Curso de Git de EscuelaIT](#) que es muchísimo más avanzado.

### Crear un repositorio en Github

Realmente no sé muy bien en qué momento del proceso tienes la duda, pero parece un poco general del uso de Git, así que te explico en pocas palabras cómo debes de hacerlo.

Desde Github creas un repositorio con el botón "+" de arriba a la derecha. Obviamente tienes que haberte registrado en Github para comenzar. El registro es gratuito.



Apareces en una página para que indiques los datos del repositorio. Básicamente tienes que darle un nombre, elegir si es público o privado y opcionalmente una descripción. Yo la parte que dice "Initialize this repository with a README" siempre la dejo desmarcada. Luego comentaré algo más sobre el "README".

Owner

Repository name



/

Great repository names are short and memorable. Need inspiration? [Click here](#)

Description (optional)



**Public**

Anyone can see this repository. You choose who can commit.



**Private**

You choose who can see and commit to this repository.

☐ **Initialize this repository with a README**

This will let you immediately clone the repository to your computer. Skip this step if you don't want to initialize the repository.

El tema de hacer el repositorio público o privado te lo tienen que decir los que te han pedido realizar esta tarea. Si no me equivoco, para hacer repositorios en privado debes tener una cuenta de pago en Github, por lo que entiendo que ellos te habrán pedido hacerlo público. Tendrás que salir de dudas con ellos

Una vez que creas el repositorio mantén en el navegador de momento la página que te aparece en Github, porque tiene información importante que en seguida usaremos.

## Subir el proyecto a Github con Push

¿Sabes lo que tienes que hacer para subir los archivos? Es fácil. No lo subes por Zip ni nada de eso, sino que usas el propio Git, el sistema de control de versiones. La operación que tienes que realizar se llama "push".



Yo lo hago desde la línea de comandos, pero hay programas de interfaz gráfica que también te hacen estos pasos quizás más fácilmente. Si no tienes instalado Git, primero tendrás que ver cómo se hace (es descargar un ejecutable e instalarlo) en este artículo: [Instalar Git y primeros pasos](#).

Nota: Ten en cuenta que si estás en Linux o Mac usarás el terminal del propio sistema operativo, pero si estás en Windows usarás "Git Bash" que es el terminal de línea de comandos que se instala cuando instalas Git, que es mucho mejor que el terminal básico que viene con Windows. El comando para moverte entre carpetas es "cd" seguido del nombre de la carpeta donde te quieres meter.

Una vez tienes Git instalado, tienes que ir, en el terminal, a la carpeta de tu proyecto, entonces allí generas tu repositorio en local con la orden "init" (si es que no lo has hecho ya).

```
git init
```

Luego, desde la carpeta de haces el comando "add" para agregar todos los archivos al "staging area".

```
git add .
```

Luego lanzas el comando para el commit, que se lleva los archivos al repositorio para control de cambios. Es el siguiente:

```
git commit -m 'mi primer commit'
```

En vez de 'mi primer commit' pon algo que sea menos genérico y más relevante, relacionado con tu proyecto y el estado en el que estás ;)

Luego tienes que hacer el "push" desde tu repositorio local a remoto con los comandos que aparecen en la página de Github que hay justo después de haber creado el repositorio (allí donde te pedí que permanecieras con el navegador). Vuelve a tu navegador y los verás, abajo del todo, en la alternativa de subir un repositorio existente en local.

### **...or push an existing repository from the command line**

```
git remote add origin https://github.com/midesweb/test-de-repo-para-post.git  
git push -u origin master
```

Es algo como esto:

```
git remote add origin https://github.com/aqui-tu-repo.git
```

---

Y luego haces el propio push también con git:

```
git push -u origin master
```

Inisisto que estos dos comandos te aparecen justo en la página que llegas al crear un repositorio. Es bueno copiar y pegar de allí, porque aparecerá la URL de tu repositorio en GitHub y así no corres el riesgo de equivocarte al escribir.

## Pasos adicionales recomendados en la operativa con Git

Para acabar, o antes de subir el proyecto con push, por buenas prácticas, deberías ponerle un README.md a tu proyecto (generas el archivo en local y haces el git add . para que se agregue y luego el commit).

Ese archivo README.md tiene sintaxis Markdown, que permite escribir cosas como enlaces, encabezamientos, negritas, etc.

Luego también es recomendable tener un [archivo .gitignore](#), del que podemos hablar en otro momento, pero básicamente hace que se ignoren para Git ciertas carpetas o archivos que no deberían subirse al control de versiones.

Creo que es todo. Seguro que te he aclarado varios puntos. Inténtalo y peléate un poco, que así es como se aprende. Esta operativa puede parecer complicada al principio, pero cuando la haces unas cuantas veces la cosa va quedando más clara.

Este artículo es obra de *Miguel Angel Alvarez*  
Fue publicado / actualizado en 11/08/2016  
Disponible online en <https://desarrolloweb.com/articulos/crear-repositorio-git-codigo.html>

---

## Archivo .gitignore

Qué es el archivo gitignore, para qué sirve, cómo implementar el gitignore en un repositorio Git.

Git tiene una herramienta imprescindible casi en cualquier proyecto, el archivo "gitignore", que sirve para decirle a Git qué archivos o directorios completos debe ignorar y no subir al repositorio de código.

Su implementación es muy sencilla, por lo que no hay motivo para no usarlo en cualquier proyecto y para cualquier nivel de conocimientos de Git que tenga el desarrollador. Únicamente se necesita crear un archivo especificando qué elementos se deben ignorar y, a partir de entonces, realizar el resto del proceso para trabajo con Git de manera habitual.

En el gitignore se especificarán todas las rutas y archivos que no se requieren y con ello, el proceso de control de versiones simplemente ignorará esos archivos. Es algo tan habitual que

---

no debíamos de dejarlo pasar en el [Manual de Git](#).



## Por qué usar gitignore

Piensa que no todos los archivos y carpetas son necesarios de gestionar a partir del sistema de control de versiones. Hay código que no necesitas enviar a Git, ya sea porque sea privado para un desarrollador en concreto y no lo necesiten (o lo deban) conocer el resto de las personas. Pueden ser también archivos binarios con datos que no necesitas mantener en el control de versiones, como diagramas, instaladores de software, etc.

El ejemplo más claro que se puede dar surge cuando se trabaja con sistemas de gestión de dependencias, como npm, Bower, Composer. Al instalar las dependencias se descargan muchos archivos con documentos, tests, demos, etc. Todo eso no es necesario que se mantenga en el sistema de gestión de versiones, porque no forma parte del código de nuestro proyecto en concreto, sino que es código de terceros. Si Git ignora todos esos archivos, el peso total de proyecto será mucho menor y eso redundará en un mejor mantenimiento y distribución del código.

Otro claro ejemplo de uso de gitignore son los archivos que crean los sistemas operativos automáticamente, archivos que muchas veces están ocultos y no los vemos, pero que existen. Si no evitas que Git los procese, estarán en tu proyecto como cualquier otro archivo de código y generalmente es algo que no quieres que ocurra.

## Implementar el gitignore

Como hemos dicho, si algo caracteriza a gitignore es que es muy fácil de usar. Simplemente tienes que crear un archivo que se llama ".gitignore" en la carpeta raíz de tu proyecto. Como puedes observar, es un archivo oculto, ya que comienza por un punto ".".

Nota: Los archivos cuyo nombre comienza en punto "." son ocultos solamente en Linux y Mac. En Windows los podrás ver perfectamente con el explorador de archivos.

Dentro del archivo .gitignore colocarás texto plano, con todas las carpetas que quieres que Git simplemente ignore, así como los archivos.

La notación es muy simple. Por ejemplo, si indicamos la línea

```
bower_components/
```

Estamos evitando que se procese en el control de versiones todo el contenido de la carpeta "bower\_components".

Si colocamos la siguiente línea:

```
*.DS_Store
```

Estaremos evitando que el sistema de control de versiones procese todos los archivos acabados de .DS\_Store, que son ficheros de esos que crea el sistema operativo del Mac (OS X) automáticamente.

Hay muchos tipos de patrones aplicables a la hora de especificar grupos de ficheros, con comodines diversos, que puedes usar para poder indicar, de manera muy específica, lo que quieres que Git no procese al realizar el control de versiones. Puedes encontrar más información en la [documentación de gitignore](#), pero generalmente no lo necesitarás porque lo más cómodo es crear el código de este archivo por medio de unas plantillas que ahora te explicaremos.

## Cómo generar el código de tu .gitignore de manera automática

Dado que la mayoría de las veces los archivos que necesitas ignorar son siempre los mismos, atendiendo a tu sistema operativo, lenguajes y tecnologías que uses para desarrollar, es muy sencillo crear un archivo .gitignore por medio de una especie de plantillas.

Existe una herramienta online que yo uso siempre que se llama [gitignore.io](#). Básicamente permite escribir en un campo de búsqueda los nombres de todas las herramientas, sistemas, frameworks, lenguajes, etc. que puedas estar usando. Seleccionas todos los valores y luego generas el archivo de manera automática.

Por ejemplo una alternativa sería escribir las siguientes palabras clave: OSX, Windows, Node, Polymer, SublimeText.

Escribes cada una de las palabras y pulsas la tecla enter para ir creando los "chips". Te debe aparecer algo como esto:



Una vez generado el código de tu gitignore, ya solo te queda copiarlo y pegarlo en tu archivo

.gitignore, en la raíz de tu proyecto.

## Eliminar archivos del repositorio si te has olvidado el .gitignore

Nos ha pasado a todos más de una vez que se nos olvida generar el correspondiente .gitignore después de haber hecho un commit. Observerás que, por mucho que estés diciendo que ahora sí quieres ignorar ciertas carpetas o rutas, éstas continúan en tu repositorio. Básicamente, esto es así porque estaban allí antes de informar que se debían ignorar. En los siguientes commits serán ignoradas todas las modificaciones de las carpetas en cuestión, pero lo que había antes perdurará en el repositorio.

Por ejemplo, imagina que estás en un proyecto NodeJS. Olvidas de hacer el gitignore de la carpeta "node\_modules". Entonces haces un commit y metes un montón de dependencias a tu repositorio git, que no debían estar. Si ves ahora en un sistema de interfaz gráfica tu repositorio (o subiéndolo a Github) podrás observar que los archivos de "node\_modules" están ahí.

Luego creas tu .gitignore con el código para node, que puede ser muy grande pero donde querrás al menos ignorar los gestores de dependencias que puedas estar usando, como por ejemplo:

```
# Código gitignore para evitar procesar los directorios de las dependencias
node_modules
jspm_packages
```

Nota: Las líneas que comienzan por "#" en un .gitignore son simplemente comentarios.

Ahora haces nuevos commits pero los archivos no se borran. ¿Qué hacer entonces?

Básicamente lo solucionas con un comando de Git llamado "rm" que básicamente funciona igual que el comando del mismo nombre que usas para borrar archivos en una consola del estilo de Mac o Linux.

```
git rm -r --cached node_modules
```

Luego tendrás que hacer un commit para que esos cambios se apliquen al sistema de control de versiones.

```
git commit -m 'Eliminada carpeta node_modules del repo'
```

A partir de ahora esa carpeta no se verá en tu repositorio y gracias a tu .gitignore tampoco se tendrá en cuenta en las siguientes operaciones que realices mediante Git.

Este artículo es obra de *Miguel Angel Alvarez*  
Fue publicado / actualizado en 16/12/2016  
Disponible online en <https://desarrolloweb.com/articulos/archivo-gitignore.html>

## Clonar un repositorio: Git clone

Aprende a clonar un repositorio Git que se encuentra alojado en GitHub. El comando es git clone, muy sencillo de usar. Sin embargo también es importante saber cómo restaurar las dependencias de los proyectos una vez clonado el repo.



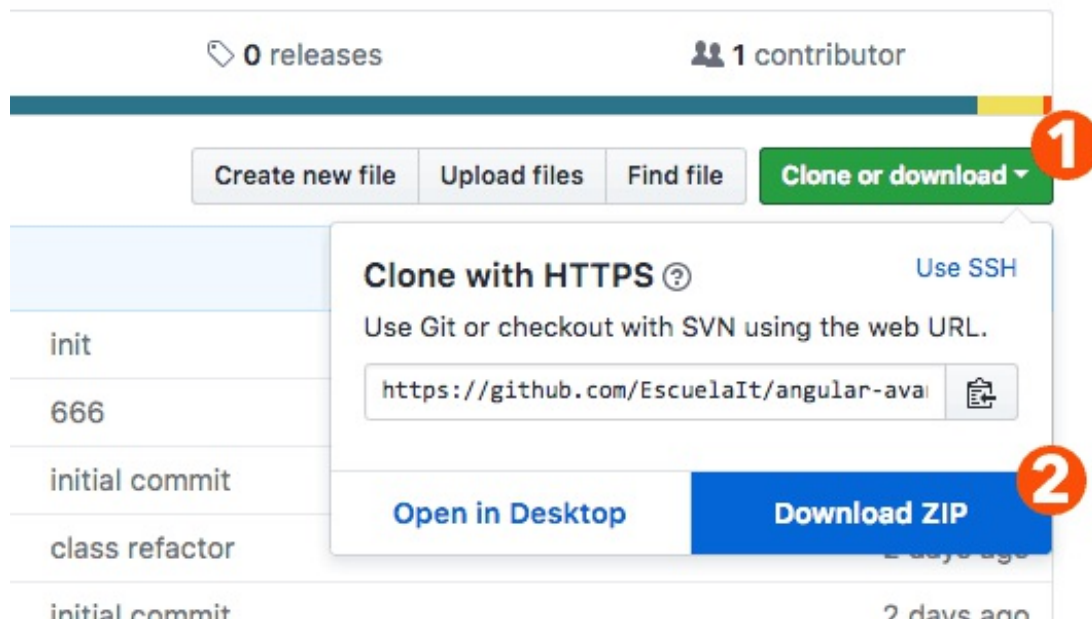
En este artículo del [Manual de Git](#) vamos a hablar de la operativa de clonado de un repositorio, el proceso que tienes que hacer cuando quieres traerte el código de un proyecto que está publicado en GitHub y lo quieres restaurar en tu ordenador, para poder usarlo en local, modificarlo, etc.

Este paso es bastante básico y muy sencillo de hacer, pero es esencial porque lo necesitarás realizar muchas veces en tu trabajo como desarrollador. Además intentaremos complementarlo con alguna información útil, de modo que puedas aprender cosas útiles y un poquito más avanzadas.

Ten en cuenta que el clonado de repositorios puede requerir que te autentiques en GitHub, especialmente si es un repositorio privado. La autenticación ahora se debe realizar mediante lo que se conoce como un Personal Access Token. Aprende a [crear y usar tus Personal Access Token de GitHub](#).

## Descargar vs Clonar

Al inicio de uso de un sitio como GitHub, si no tenemos ni idea de usar Git, también podemos obtener el código de un repositorio descargando un simple Zip. Esta opción la consigues mediante el enlace de la siguiente imagen.



Sin embargo, descargar un repositorio así, aunque muy sencillo no te permite algunas de las utilidades interesantes de clonarlo, como:

- No crea un repositorio Git en local con los cambios que el repositorio remoto ha tenido a lo largo del tiempo. Es decir, te descargas el código, pero nada más.
- No podrás luego enviar cambios al repositorio remoto, una vez los hayas realizado en local.

En resumen, no podrás usar en general las ventajas de Git en el código descargado. Así que es mejor clonar, ya que aprender a realizar este paso es también muy sencillo.

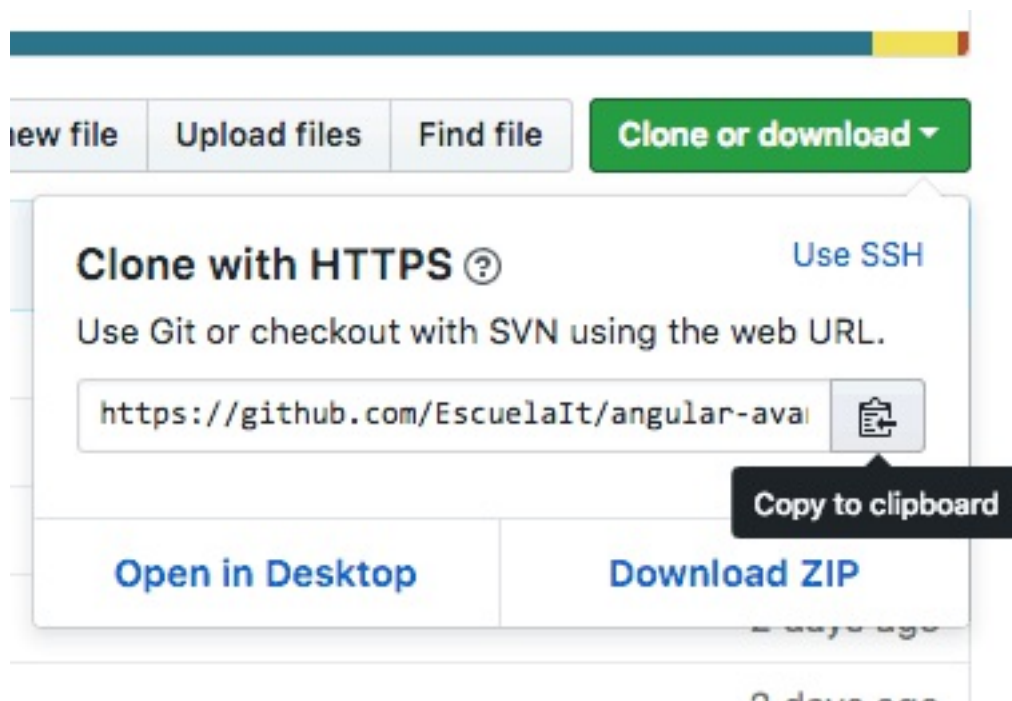
## Clonar el repositorio Git

Entonces veamos cómo debes clonar el repositorio, de modo que sí puedas beneficiarte de Git con el código descargado. El proceso es el siguiente.

Nota: Asumimos que tienes instalado Git en tu sistema local. Puedes aprender a hacerlo en este artículo: [Entiende, instala y configura Git](#).

Primero copiarás la URL del repositorio remoto que deseas clonar (ver el icono "Copy to clipboard en la siguiente imagen).





Luego abrirás una ventana de terminal, para situarte sobre la carpeta de tu proyecto que quieras clonar. Yo te recomendaría crear ya directamente una carpeta con el nombre del proyecto que estás clonando, o cualquier otro nombre que te parezca mejor para este repositorio. Te sitúas dentro de esa carpeta (cd, cd...) y desde ella lanzamos el comando para hacer el clon, que sería algo como esto:

git clone <https://github.com/EscuelaIt/angular-avanzado.git> . El último punto le indica que el clon lo vas a colocar en la carpeta donde estás situado, en tu ventana de terminal. La salida de ese comando sería más o menos como tienes en la siguiente imagen:

```
~/proyectos/angular-avanzado 11:28:37 100% midesweb
> git clone https://github.com/EscuelaIt/angular-avanzado.git .
Cloning into '.'...
remote: Counting objects: 341, done.
remote: Compressing objects: 100% (232/232), done.
remote: Total 341 (delta 136), reused 298 (delta 93), pack-reused 0
Receiving objects: 100% (341/341), 1.72 MiB | 898.00 KiB/s, done.
Resolving deltas: 100% (136/136), done.
master ~/proyectos/angular-avanzado 11:28:46 100% midesweb
>
```

## Instalar las dependencias

Habitualmente los desarrollos de Git tienen ignoradas las dependencias mediante el [archivo .gitignore](#), por ello es importante que las instalemos de nuevo en el repositorio clon, en local.

Para cada tipo de proyecto, con cada lenguaje, existirán comandos para instalar las dependencias. Este comando lo tendrías que conocer tú mismo, si es de [PHP dependerá de Composer](#), si es de NodeJS (o Javascript) [dependerá de npm](#).

Por ejemplo, así reinstalamos las dependencias en un proyecto Angular, que usa npm como gestor de dependencias.

```
npm install
```

```
master ~/proyectos/angular-avanzado 11:28:46 100% midesweb  
> npm install  
(( )) :: extract:@angular/common: sill extract is-descriptor@1.0.
```

Pero insisto que esto depende del gestor de dependencias, o gestores de dependencias del proyecto, si es que se están usando.

## Subir cambios al repositorio remoto realizados por nosotros

Una vez modificado el código en local podrás querer subir los cambios al repositorio remoto. Es algo muy sencillo, ya que al clonar el repositorio en local está asociado el origen remoto desde donde lo trajiste. Sin embargo, tenemos que llevar en consideración algunos puntos:

- Dependiendo de la empresa o del equipo de trabajo pueden haber algunas normas para enviar cambios, como por ejemplo trabajar siempre con [ramas](#) y enviar los cambios a una rama en concreto, que luego se puede fusionar con un [pull request](#).
- Si el repositorio que clonaste no es tuyo y no tienes permisos por no pertenecer a una organización que lo posea, obviamente, no podrás enviar cambios.

Nos vamos a quedar aquí en lo sencillo, pensando que: 1) El repositorio es tuyo y 2) que nadie se molesta por enviar cambios directamente a la rama master. Entonces, para enviar cambios a GitHub, o cualquier otro hosting de repositorios, haríamos:

```
git push
```

Con eso se envían los cambios en el instante y los podrás ver reflejados en el repo remoto.

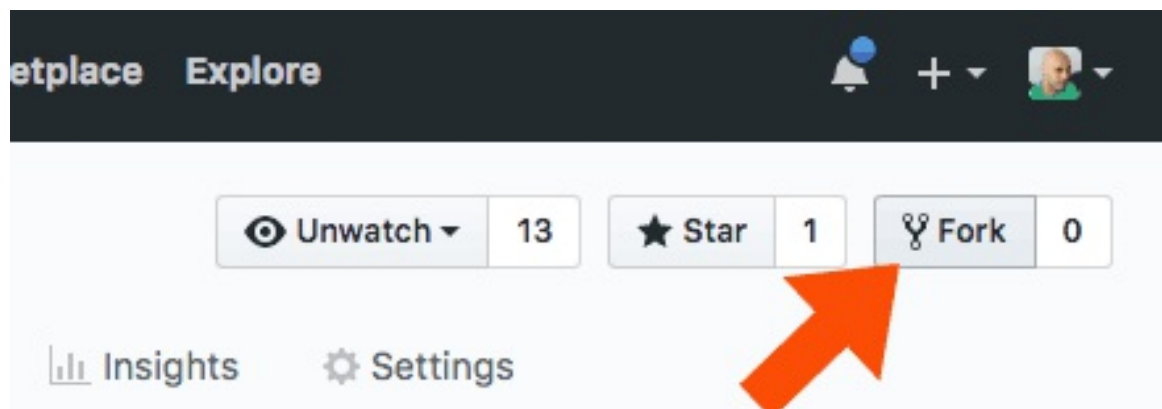
Nota: Obviamente, para que se envíen los cambios del repositorio, local tendrás que haber realizado alguna/s modificación/es en el código del proyecto, y por supuesto las tendrás que haber confirmado con el correspondiente/s commit. Pero esto suponemos que ya lo conoces, como lector del [Manual de Git](#). Puedes encontrar más información sobre esta operativa básica en el artículo sobre ti [primer commit](#).

Cómo enviar cambios a GitHub si el repositorio no es tuyo (vía fork)

Si el repositorio que vas a modificar no es tuyo y pretendes continuar el desarrollo, agregando cambios que querrás enviar a GitHub, tendrías que clonar el repositorio de otra manera.

Primero tienes que crear un "fork". Esto es, un repositorio que es copia de otro que ya está publicado en GitHub. Lo bueno de un fork es que el repositorio será exactamente igual, pero estará en tu cuenta de GitHub, con tu usuario, por lo que tendrás permisos para hacer lo que tú quieras con él.

Hay un botón para hacer el fork en la parte de arriba de la página del repositorio.



Una vez hecho el fork, el repositorio ya te pertenece. Entonces podrás clonar (el fork, no el repositorio original) y realizar los cambios que quieras, que podrás subir perfectamente a tu propio repositorio (tu fork).

En la parte de arriba puedes ver que el repositorio es un fork de otro repositorio existente en GitHub.



El resto del proceso, para descargar el repositorio, modificarlo y enviar los cambios es exactamente igual que lo descrito anteriormente, incluido el "git push" para enviar los cambios. Solo que ahora, ya que es tu propio repositorio, sí que te dejará actualizar su código en remoto.

Este artículo es obra de *Miguel Angel Alvarez*  
Fue publicado / actualizado en 02/05/2018  
Disponible online en <https://desarrolloweb.com/articulos/git-clone-clonar-repositorio.html>

## Git log

Ver el historial de commits de un proyecto con git log, cómo hacerlo, cuáles son las opciones más útiles para sacarle partido al log.

A veces necesitamos examinar la secuencia de commits (confirmaciones) que hemos realizado en la historia de un proyecto, para saber dónde estamos y qué estados hemos tenido a lo largo

---

de la vida del repositorio. Esto lo conseguimos con el comando de Git "log".

Una vez tenemos el listado en nuestro terminal podemos observar los mensajes de los commits y su identificador, pero también podríamos obtener más información de un commit en concreto, sus modificaciones en el código, o tener la referencia para moverse hacia atrás en el histórico de commits. En este artículo del [Manual de Git](http://desarrolloweb.com/manuales/manual-de-git.html) vamos a ver algunas de estas acciones más comunes para empezar a practicar con el log de Git.



## Comando git log

El comando que podemos usar para ver el histórico de commits, estando situados en la carpeta de nuestro proyecto, es:

```
git log
```

Con esto obtenemos una salida como la que sigue en la siguiente imagen:

```
commit b2c07b2f6bf6910c3a05aff3c3c2dc8efb096aa5
Author: Miguel Angel Alvarez <malvarez@desarrolloweb.com>
Date:   Fri Feb 24 19:53:11 2017 -0300
```

Cambios última clase

```
commit fa42f3ba4dc9b0112dd1e302f4a33b73be64e539
Author: Miguel Angel Alvarez <malvarez@desarrolloweb.com>
Date:   Fri Feb 17 18:11:19 2017 -0200
```

clase 2 de node

```
commit 35446abb25913d0bb383e9759a405de88fc46409
Author: Miguel Angel Alvarez <malvarez@desarrolloweb.com>
Date:   Fri Feb 17 16:00:58 2017 -0200
```

componentes antes de clase 2 nodejs

```
commit bda125319b4df54bc9fbf6596c83bcd5d0fe9a03
Author: Miguel Angel Alvarez <malvarez@desarrolloweb.com>
Date:   Mon Feb 13 18:23:03 2017 -0200
```

Clase 2 de polymerfire

Nota: El listado puede contener una cantidad de commits que no quepa en la ventana del terminal, en ese caso podemos ir a la siguiente página con la barra espaciadora. O puedes moverte mediante los cursores del teclado, arriba y abajo. Salimos de nuevo a la entrada de comandos con "CTRL + z".

Como puedes observar, el listado de commits está invertido, es decir, los últimos realizados aparecen los primeros.

De un commit podemos ver diversas informaciones básicas como:

- Identificador del commit
- Autor
- Fecha de realización
- Mensaje enviado

Podría ser algo como esto:

```
commit cd4bcc8bad230f895fcadd5baf258715466a8eaf
Author: Miguel Angel Alvarez <malvarez@desarrolloweb.com>
Date: Fri Feb 10 18:38:41 2017 -0200
```

```
ejemplos clase 1 polymerfire
```

## Log en una línea por commit

Es muy útil lanzar el log en una sola línea, lo que permite que veamos más cantidad de commits en la pantalla y facilita mucho seguir la secuencia, en vez de tener que ver un montón de páginas de commits.

Para ello usamos el mismo comando, pero con la opción "--oneline":

```
git log --oneline
```

Entonces la salida será algo como esto:

```
b2c07b2 Cambios última clase
fa42f3b clase 2 de node
35446ab componentes antes de clase 2 nodejs
bda1253 Clase 2 de polymerfire
cd4bcc8 ejemplos clase 1 polymerfire
d5a4479 Firebase-app integration
6b43e59 Iniciación a Polymer
```

Nota: Esta opción en realidad es un atajo de escribir estas dos opciones: "--pretty=oneline -abbrev-commit". La primera indica que justamente que pongas un commit por línea y la segunda te indica que muestres la cadena resumida con el identificador del commit, en vez de la cadena de 40 bytes hexadecimal normal. El identificador resumido es suficiente para referirse de manera inequívoca a un commit, ya que el resto del identificador contiene información sobre el usuario que hizo el commit y otros datos de control.

## Ver un número limitado de commits

Si tu proyecto ya tiene muchos commits, decenas o cientos de ellos, quizás no quieras mostrarlos todos, ya que generalmente no querrás ver cosas tan antiguas como el origen del repositorio. Para ver un número de logs determinado introducimos ese número como opción, con el signo "-" delante (-1, -8, -12...).

Por ejemplo, esto muestra los últimos tres commits:

```
git log -3
```



## Ver información extendida del commit

Si queremos que el log también nos muestre los cambios en el código de cada commit podemos usar la opción `-p`. Esta opción genera una salida mucho más larga, por lo que seguramente nos tocará movernos en la salida con los cursores y usaremos `CTRL + Z` para salir.

```
git log -p
```

Eso nos mostrará los cambios en el código de los últimos dos commits. Sin embargo, si quieres ver los cambios de cualquier otro commit que no sea tan reciente es mucho más cómodo usar otro comando de git que te explicamos a continuación `"git show"`.

## Comando "show": obtener mayor información de un commit

Podemos ver qué cambios en el código se produjeron mediante un commit en concreto con el comando `"show"`. No es algo realmente relativo al propio log del commit, pero sí que necesitamos hacer log primero para ver el identificador del commit que queremos examinar.

Realmente nos vale con indicarle el resumen de identificador del commit, que vimos con el modificador `--oneline`.

```
git show b2c07b2
```

Nota: Igual que en el log, para ver más cosas que no quepan en la pantalla, tenemos que usar los cursores del teclado, o pasar a la siguiente página con la barra espaciadora. Salimos con `"CTRL + z"`.

Podrás observar que al mostrar la información del commit te indica todas las líneas agregadas, en verde y con el signo `"+"` al principio, y las líneas quitadas en rojo y con el signo `"-"` al principio.

## Opción `--graph` para el log con diagrama de las ramas

Si tu repositorio tiene ramas (branch) y quieres que el log te muestre más información de las ramas existentes, sus uniones (merges) y cosas así, puedes hacer uso de la opción `--graph`.

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

Ese comando te mostrará los commit en una línea y las ramas en las que estabas, con sus diferentes operaciones. La salida del git log será algo como esto:

```
* e181272 Esto es un merge con mensaje
\
* 071d7ee commit en experimental para test
* | bc3b0c2 Yeah! git rules!
* | 27d0926 Merge branch 'experimental'
| \
| | /
| * a7c67c8 commit experimental
| * cb8877e Segundo cambio experimental
| * e2a99f8 cambio experimental
* | 03bbd72 cambio importante en master
| /
* 51aae71 primer commit
```

Nota: En el [Manual de Git](#) se explica en otro artículo el [trabajo con ramas](#). De hecho, en este diagrama se pueden ver las branch y merges que realizamos como práctica en aquel artículo.

## Conclusión

Hemos visto algunas de las opciones más útiles y recurrentes cuando quieres revisar el log de un repositorio Git. Pero hay decenas de otras configuraciones y opciones del comando existentes para conseguir infinidad de nuevos comportamientos.

La referencia completa de opciones del comando "git log" la puedes ver en esta dirección: <https://git-scm.com/docs/git-log>

Este artículo es obra de *Miguel Angel Alvarez*  
Fue publicado / actualizado en 28/02/2017  
Disponible online en <https://desarrolloweb.com/articulos/git-log.html>

## Modificar el último commit, con Git

Una operativa corriente del mantenimiento de un repositorio Git consiste en modificar el commit realizado por último con la opción --amend.

En el día a día de Git hay una situación que se produce bastante a menudo y que podemos

solucionar de una manera elegante con una opción determinada de la acción de commit. Básicamente se trata de modificar el último commit, en lugar de crear uno nuevo.

Imagina que estás editando tu proyecto, realizas una serie de cambios y luego haces el correspondiente commit para confirmarlos en el repositorio. Casi enseguida te das cuenta que cometiste un error. Entonces puedes arreglar tu código para solucionarlo y lanzar otro commit para confirmarlo. Está todo bien, lo que pasa es que la funcionalidad que has agregado es solo una, lo que pasa es que se generaron dos commits cuando lo más elegante es que hubiéramos creado solo uno, con la funcionalidad bien realizada.

La manera de solucionar esta situación es muy sencilla y merece la pena que la veamos, porque no supone ningún esfuerzo y nos permite gestionar nuestro repositorio un poco mejor. Como seguramente estemos empezando con Git, leyendo el [Manual de Git](http://desarrolloweb.com/manuales/manual-de-git.html) de DesarrolloWeb.com, vamos a hacer una práctica completa que a la vez nos servirá de repaso de todo lo aprendido.



## Situación de nuestro repositorio

Partamos de un repositorio vacío:

```
git init
```

Ahora creamos un archivo llamado "README". En el que podemos colocar ya un poco de texto. Lo puedes hacer también con la consola con este comando (aunque también lo puedes hacer, claro está, con tu editor preferido).

```
echo "Esto es el README" > README
```

Luego añado los archivos al staging area, para que estén trackeados por Git.

```
git add .
```

Ahora hago un commit confirmando los cambios.

```
git commit -m 'creado readme'
```

## Hacer cambios en el archivo

Ahora te das cuenta que no colocaste algo en el README y lo editas de nuevo para colocar cualquier nuevo contenido, como tu nombre como autor del proyecto y tu email. Abres el archivo con tu editor y lo modificas. Podemos ver los cambios recién realizados con el comando status:

```
git status
```

Obtendrás una salida como esta:

```
mcMiguel:gitamend midesweb$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   README

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

Ahora podrías agregar ese archivo:

```
git add .
```

Y luego hacer la confirmación con:

```
git commit -m 'Indicado el nombre del autor'
```

Nota: Como el archivo ya se encuentra en el repositorio, Git lo está trackeando, podrías ahorrarte un paso, agregando el archivo y haciendo la confirmación de una única vez, con `git commit -am 'indicado nombre del autor'`.

Pero justamente es lo que queremos evitar. No necesito dos commits para esta operación, ya que he creado simplemente el README y me olvidé una cosa simple que no merece la pena tener en un commit separado.

## Modificar el commit con --amend

Entonces, vamos a alterar el commit anterior, ya que es lo lógico, en vez de producir un commit nuevo. Simplemente indicamos la opción --amend.

```
git commit --amend
```

Nota: Aunque lo hicimos en el anterior paso, no te olvides de hacer el "git add .", porque si no en el commit no habrá nada que enviar. En este caso no ponemos mensaje, ya que el mensaje ya lo habíamos indicado en el anterior commit. No obstante, Git nos abrirá un editor para que nosotros podamos editar el mensaje y cambiarlo por el que nos guste más. Puedes ahorrarte el (para algunos fastidioso) editor de línea de comandos indicando el mensaje en el commit con "-m". Aunque estés haciendo un "amend" y ya tengas mensaje lo puedes sobrescribirlo directamente sin tener en cuenta lo indicado anteriormente: git commit --amend -m 'Creado el readme editado el commit primero'

El editor puede variar dependiendo del sistema en el que te encuentres. Pero básicamente en todos los casos podrás editar el texto del mensaje del commit y luego activar la opción para salir de edición del texto.

Nota: A mi me lo abre con Vim, por lo que tengo que, desde el modo comando (pulsas ESC para asegurarte que estás en modo comando), escribes :wq y pulsas enter.

Listo, con esto hemos editado el anterior commit en vez de crear uno nuevo.

Por último, si haces "git log" podrás ver que efectivamente solo hay un commit en el repositorio, ya que el que hicimos para arreglar el problema en el README realmente solo modificó el commit anterior.

```
mcMiguel:gitamend midesweb$ git log
commit 1b102ddb2e6f98f50cf73f9c2d569ff7f1d5ffd3
Author: Miguel Angel Alvarez <malvarez@desarrolloweb.com>
Date: Sat Feb 25 09:43:42 2017 -0300

creado readme
```

Listo, eso es todo. Ya sabes modificar el último commit sin necesidad de hacer un commit nuevo e innecesario.

Este artículo es obra de *Miguel Angel Alvarez*  
Fue publicado / actualizado en 16/03/2017  
Disponible online en <https://desarrolloweb.com/articulos/modificar-ultimo-commit-git.html>

## Eliminar archivos de un repositorio git después de ignorarlos con .gitignore

Cómo eliminar archivos que no deberían estar en el repositorio y que se nos olvidó agregar al .gitignore, desde staging area, antes o después de hacer uno o varios commit.

En el [Manual de Git](#) hemos aprendido muchas operativas sencillas y complejas para aprender a usar este sistema de control de versiones. Después de haber abordado mucha información de base, en este artículo vamos a explicar una solución a un problema común a la hora de usar Git.

Se trata de un problema sencillo pero recurrente: borrar archivos de un repositorio GIT, que no deberían estar ahí por el motivo que sea.

Como sabes, existe un [archivo llamado .gitignore](#) con el que configuramos los archivos o carpetas que se deben pasar por alto al agregar contenido al repositorio. El típico contenido para meter en el .gitignore es archivos binarios, dependencias, variables de entorno, etc.



Gestionando correctamente el .gitignore conseguimos el efecto deseado, que solo entre en el repositorio aquel material que debería estar ahí. ¿Pero qué se puede hacer cuando hemos agregado al repositorio material que no queríamos trackear? Es decir, queremos que tales archivos continúen en el proyecto, pero no queremos que estén en el repositorio Git por cualquier motivo.

Cuando nos damos cuenta que estamos dando seguimiento a archivos que no deberíamos podemos encontrarnos en dos situaciones que vamos a resolver en este artículo:

- Que el archivo se haya agregado simplemente al staging area, pero que no se haya hecho commit todavía a estos archivos
- Que el archivo, o carpeta, esté ya en el repositorio, porque se haya hecho un commit anteriormente. Uno o varios, antes de darnos cuenta que el archivo no debería estar en el repositorio.

## Eliminar el seguimiento de archivos que sólo están en staging area

Supongo que sabrás qué es el "Staging Area", el área intermedia de Git donde residen los archivos a los que se está haciendo seguimiento, antes de confirmarlos con un commit. Es algo básico, por lo que si no sabes de lo que estamos hablando te recomiendo la lectura de artículos anteriores como el del [Curso de Git](#).

Cuando los archivos están en el staging area la operativa de eliminar los archivos del seguimiento con Git es bastante sencilla. Basta con lanzar un comando:

```
git reset HEAD nombre_de_archivo
```



---

Si quieres eliminar del staging area todos los ficheros del directorio donde nos encontramos.

```
git reset HEAD .
```

Recuerda que luego puedes lanzar un comando "`git status`" con el que puedes comprobar que los archivos ya no están en seguimiento. "Untracked files: ...".

## Eliminar del repositorio archivos que ya se han confirmado (se ha hecho commit)

El problema mayor te puede llegar cuando quieras quitarte de enmedio un archivo que hayas confirmado anteriormente. O una carpeta con un conjunto de archivos que no deberían estar en el repositorio.

Una vez confirmados los cambios, es decir, una vez has hecho commit a esos archivos por primera vez, los archivos ya forman parte del repositorio y sacarlos de allí requiere algún paso adicional. Pero es posible gracias a la instrucción "rm". Veamos la operativa resumida en una serie de comandos.

### Borrar los archivos del repositorio

El primer paso sería eliminar esos archivos del repositorio. Para eso está el comando "rm". Sin embargo, ese comando tal cual, sin los parámetros adecuados, borrará el archivo también de nuestro directorio de trabajo, lo que es posible que no desees.

Si quieres que el archivo permanezca en tu ordenador pero simplemente que se borre del repositorio tienes que hacerlo así.

```
git rm --cached nombre_archivo
```

Si lo que deseas es borrar un directorio y todos sus contenidos, tendrás que hacer algo así:

```
git rm -r --cached nombre_directorio
```

El parámetro "--cached" es el que nos permite mantener los archivos en nuestro directorio de trabajo.

### Asegurarse que estamos ignorando los archivos con .gitignore

Luego se trataría de asegurarse que nuestros archivos se encuentran correctamente ignorados, para que no los volvamos a meter en el repositorio cuando confirmemos cambios más adelante con commit.

Esto lo tienes que hacer en el archivo .gitignore y ya lo hemos explicado anteriormente, por lo que te recomiendo leer el artículo sobre [.gitignore](#).

---

Hacer el commit para confirmar los cambios

Una vez hemos quitado del repositorio los archivos que no queremos, y ahora sí están ignorados, tenemos que confirmar los cambios. Esta operación hará que ese archivo o esa carpeta desaparezca del estado actual de la rama de nuestro repositorio.

```
git commit -m 'Eliminados archivos no deseados'
```

Enviar esos cambios al repositorio remoto

Como último paso, en el caso que tengas un repositorio remoto donde envías tu código (GitHub o similar), tendrás que trasladar los datos allí.

Esto lo haces con "push", como de costumbre.

```
git push
```

O en el caso que tengas que especificar el repositorio remoto y la rama, sería algo como:

```
git push origin master
```

## Histórico de Git

Ten en cuenta que, aunque en un momento dado elimines archivos de tu repositorio, éstos seguirán en el histórico del repositorio y por tanto se podrán ver en commits anteriores, si es que alguien los busca. Es porque, cuando se confirmaron los cambios por primera vez, ya se indexaron por Git y quedarán registrados en los commits antiguos.

Quitarlo del histórico de commits también podría hacerse, pero ya requiere otra serie de pasos que no vamos a tratar ahora, porque realmente no es una operativa que resulte muy habitual y la dificultad de la acción es bastante superior.

Este artículo es obra de *Miguel Angel Alvarez*  
Fue publicado / actualizado en 14/06/2018  
Disponible online en <https://desarrolloweb.com/articulos/eliminar-archivos-git-gitignore.html>

---

## Descartar cambios de archivos con Git

Una operativa habitual en Git es descartar cambios realizados sobre archivos, para volver al estado del último commit.

Gracias a Git podemos mantener un control minucioso de los cambios en los archivos y una

operación bastante habitual es descartar cambios en los archivos que acabamos de modificar. Viene muy bien cuando, después de realizar una serie de cambios, nos damos cuenta que ese no era un buen camino y queremos volver atrás, para recuperar el estado que tenía nuestro código en la última confirmación (commit).

Realmente, esta operación tiene varias alternativas de funcionamiento, dependiendo de cómo se desean descartar los cambios realizados sobre los archivos, si los queremos descartar para siempre, si los queremos descartar pero almacenar provisionalmente, etc. En este artículo del [Manual de Git](#) vamos a dar un poco de luz y ayudar a encontrar la alternativa que necesites en cada momento.



## Descartar cambios en un archivo único, para siempre

Imagina que has modificado varios archivos desde el último commit. La cosa va bien, pero hay uno de ellos que tiene algún tipo de problema. Deseas dar marcha atrás con ese archivo en concreto pero dejar los cambios en el resto de los archivos modificados.

Para ello puedes usar el comando "git checkout", seguido del nombre del archivo que deseas hacer que vuelva al estado anterior.

```
git checkout -- index.html
```

Este comando dejará el resto de archivos de tu espacio de trabajo sin tocar, eliminando los cambios únicamente del archivo que necesitabas. Como puedes apreciar, en el comando indicas el nombre del archivo a descartar cambios, que en nuestro ejemplo se llamaba "index.html".

Nota: recuerda que el uso más común de checkout es para cambiar de ramas. Puedes obtener más información en el artículo de [Uso de Ramas en Git](#).

## Descartar todos los cambios en archivos modificados

Ahora imagina que has hecho cambios sobre varios archivos, pero no te gustaron y quieres volver al estado del último commit, eliminando de manera permanente todos los cambios a todos los archivos realizados.

Esta operación la consigues con el comando "git reset". La podrás realizar de la siguiente manera:

```
git reset --hard
```

La opción "--hard" provoca que cualquier cambio de archivos cuyo seguimiento llevamos con Git se elimine, volviendo al estado justo después del último commit. Eso quiere decir que, si hubiera archivos que no se han agregado todavía al repositorio (archivos nuevos desde el último commit), permanecerán en el espacio de trabajo.

## Descartar los cambios de un archivo que está en staging area

Pudiera darse el caso que hayas hecho "git add" de un archivo que ahora pretendes descartar. En ese caso ese archivo está almacenado provisionalmente, en el espacio que se conoce como "staging area".

La operativa anterior de reset funcionará también en este caso.

```
git reset --hard
```

Si haces "git reset --hard" esos cambios también se eliminarán, pero podría ocurrir que hayas hecho el add de varios archivos al staging area y sólo quieras resetear los cambios de uno de ellos. Entonces tendrías que usar el comando reset con una opción diferente (recuerda que "git reset --hard" descarta los cambios de todos los archivos). Para eliminar entonces estos cambios de un archivo único, primero tendríamos que sacar el archivo en concreto del staging area y luego descartar los cambios, con la combinación de estos dos comandos.

```
git reset HEAD archivo.txt  
git checkout -- archivo.txt
```

## Descartar los cambios, pero guardar provisionalmente los archivos modificados

Hay una alternativa todavía un poco más compleja con el comando "git stash" para descartar cambios de un archivo, pero sin perderlos del todo, pudiendo volver al estado anterior de esos archivos más adelante. Esto puede ser útil en muchos casos, como por ejemplo cuando tienes que cambiar de rama pero no quieres hacer commit de los cambios porque tu trabajo te lo has dejado a medias. En ese caso podrías descartar los cambios, pero guardarlos para que, más adelante, puedas volver a ellos y seguir por donde lo habías dejado.

El comando "stash" es suficientemente complejo para verlo en un artículo independiente, pero veamos una operativa muy sencilla para que podamos solucionar este caso particular.

```
git stash
```

Entonces nos aparecerá un mensaje como este "Saved working directory and index state WIP

---

on nombre\_rama". WIP son las siglas de "work in progress".

Ahora verás que los archivos modificados están otra vez como en el estado del último commit. Podrías cambiar de rama o hacer cualquier cosa y, cuando quieras volver al estado de tus archivos almacenado como "work in progress", lanzarás el siguiente comando:

```
git stash pop
```

Eso provocará que los cambios guardados en stash vuelvan a colocarse en tu espacio de trabajo.

Si quieres ver el status de tu proyecto, con información sobre archivos que tengas en "Work In Progress" con stash, puedes hacer un comando como este:

```
git status --show-stash
```

## Conclusión

Con lo que hemos visto en este artículo tienes un completo set de alternativas cuando quieres descartar cambios en uno o varios archivos de un repositorio y volver al estado anterior del espacio de trabajo, justo después del último commit.

Te servirán de bastante ayuda en multitud de casos. Sólo cerciérate de estar seguro que quieres descartar los cambios, puesto que la operación es permanente (ya que no se había realizado ninguna confirmación de esos archivos modificados). Recuerda que si quieres, por cualquier motivo, dejarte abierta la posibilidad de recuperar esos cambios descartados, tendrás que usar la operativa de stash. Consulta nuestro [Tutorial de Git](#) para seguir aprendiendo otras facetas del sistema de control de versiones.

Este artículo es obra de *Miguel Angel Alvarez*  
Fue publicado / actualizado en 09/10/2018  
Disponible online en <https://desarrolloweb.com/articulos/descartar-cambios-archivos-git.html>

# Operaciones más avanzadas con Git

Ahora que ya sabemos cómo subir código a los repositorios y sincronizarnos, en los siguientes artículos vamos a ver operativas menos frecuentes con Git, pero que son fundamentales también en la gestión habitual de proyectos. No son acciones que vas a realizar todos los días al usar el sistema de control de versiones, pero son importantes para administrar el código de los proyectos y colaborar entre componentes de equipos de trabajo o proyectos Open Source.

## Especificar versiones en Git con tag

Aprende a usar Git Tag. Los tag son una manera de etiquetar estados de tu repositorio, que se usa comúnmente para indicar las versiones o releases de un proyecto mantenido con Git.

Git tiene la posibilidad de marcar estados importantes en la vida de un repositorio, algo que se suele usar habitualmente para el manejo de las releases de un proyecto. A través del comando "git tag" podemos crear etiquetas, en una operación que se conoce comúnmente con el nombre de "tagging". Es una operativa que tiene muchas variantes y utilidades, nosotros veremos las más habituales que estamos seguros te agrada conocer.

Además de mantener informados a los usuarios del código de los proyectos y otros desarrolladores de las versiones de una aplicación, el etiquetado es una herramienta fundamental para que otros sistemas sepan cuándo un proyecto ha cambiado y se permitan desencadenar procesos a ejecutar cada vez que esto ocurre. De hecho es importante porque algunos gestores de paquetes te obligan a usarlo para poder publicar packages en ellos. Así pues, vamos a relatar las bases para trabajar con el sistema de tagging de modo que puedas usarlo en tu día a día en el trabajo con Git.



## Numeración de las versiones

Git es un sistema de control de versiones. Por tanto permite mantener todos los estados por los



que ha pasado cualquier de sus archivos. Cuando hablamos de Git tag no nos referimos a la versión de un archivo en particular, sino de todo el proyecto de manera global. Sirve para etiquetar con un tag el estado del repositorio completo, algo que se suele hacer cada vez que se libera una nueva versión del software.

Las versiones de los proyectos las define el desarrollador, pero no se recomienda crearlas de manera arbitraria. En realidad la recomendación sería darle un valor semántico. Esto no tiene nada que ver con Git, pero lo indicamos aquí porque es algo que consideramos interesante que sepas cuando empiezas a gestionar tus versiones en proyectos.

Generalmente los cambios se pueden dividir en tres niveles de "importancia": Mayor, menor y pequeño ajuste. Si tu versión de proyecto estaba en la 0.0.1 y haces un cambio que no altera la funcionalidad ni la interfaz de trabajo entonces lo adecuado es versionar tu aplicación como 0.0.2. Si el proyecto ya tiene alguna ampliación en funcionalidad, pero sigue manteniendo completa compatibilidad con la versión anterior, entonces tendremos que aumentar el número de enmedio, por ejemplo pasar de la 1.0.0 a la 1.1.0. Ahora bien, si los cambios introducidos en el proyecto son tales que impliquen una alteración sobre cómo se usará esa aplicación, haciendo que no sea completamente retrocompatible con versiones anteriores, entonces habría que aumentar en 1 la versión en su número más relevante, por ejemplo pasar de la 1.1.5 a la 2.0.0.

Realmente importa poco ahora el tema de la semántica, porque queremos hablar de Git. Sin embargo lo encuentras muy bien explicado en este documento [Versionamiento Semántico 2.0.0-rc.2](#), por lo que, si te interesa el tema, te recomendamos leerlo.

## Crear un tag con Git

Se supone que cuando comienzas con un repositorio no tienes ninguna numeración de versión y ningún tag, por lo que empezaremos viendo cómo se crean.

Supongamos que empezamos por el número de versión 0.0.1. Entonces para crear la correspondiente etiqueta lanzarás el subcomando de Git "git tag":

```
git tag v0.0.1 -m "Primera versión"
```

Como ves, es una manera de etiquetar estados del repositorio, en este caso para definir números de versión. Los acompañas con un mensaje, igual que se envían mensajes en el commit.

Nota: Este es el mecanismo que se conoce como "Etiquetas ligeras", existen otros tipos de etiquetado que es posible hacer mediante Git.

Generalmente, después de hacer cambios en tu repositorio y subirlos al sistema (después de hacer el commit), podrás generar otro número de versión etiquetando el estado actual.

```
git tag v0.0.2 -m "Segunda versión, cambios menores"
```

## Ver los estados de versión en el repositorio con Git tag

Después de haber creado tu primer tag, podrás lanzar el comando "git tag", a secas, sin más parámetros, que te informará sobre las versiones que has etiquetado hasta el momento.

```
git tag
```

Si tienes un repositorio donde has etiquetado ya tres números de versiones, podría arrojar una salida como la que ves en la siguiente imagen.

```
mcMiguel:dw-flagsprites midesweb$ git tag
v0.0.1
v0.0.2
v0.0.3
```

Otro comando interesante en el tema de versionado es "show" que te permite ver cómo estaba el repositorio en cada estado que has etiquetado anteriormente, es decir, en cada versión.

```
git show v0.0.2
```

Recibirás como salida un mensaje parecido a este:

```
tag v0.0.2
Tagger: Miguel Angel Alvarez <malvarez@desarrolloweb.com>
Date:   Fri Nov 13 16:23:00 2015 -0200

"

commit 8ef366190b73d56e267c5324aa8074db3c3f0ed9
Author: Miguel Angel Alvarez <malvarez@desarrolloweb.com>
Date:   Fri Nov 13 16:21:54 2015 -0200

...
```

## Enviar tags a GitHub

Si quieres que tus tags creados en local se puedan enviar al repositorio en GitHub, puedes lanzar el push con la opción --tags. Esto es una obligatoriedad, porque si no lo colocas, el comando push no va a enviar los nuevos tags creados.

```
git push --tags
```

En concreto la opción --tags, tal cual la hemos usado, envía todas las nuevas tag creadas, aunque podrías también enviar una en concreto mediante la especificación de la que quieres

enviar, tal como se puede ver en el siguiente comando.

```
git push origin v0.0.4
```

En este caso debemos especificar qué repositorio remoto es el destino de las tags que acabamos de crear ("origin" en este caso), pues si no se especifica el comando entenderá que el nombre de nuestro tag es el nombre del repositorio remoto que estamos queriendo usar para enviar los cambios locales, lo que nos dará un error.

Nota: Aparentemente, la opción `--tag` hace el mismo efecto que `--tags`. Las dos envían los tags que tengamos en local al repositorio remoto. Por eso puedes probar usar ambas, aunque en la documentación de Git usan siempre `--tags`.

```
git push --tag
```

Enviar tags y hacer push de los commits al mismo tiempo

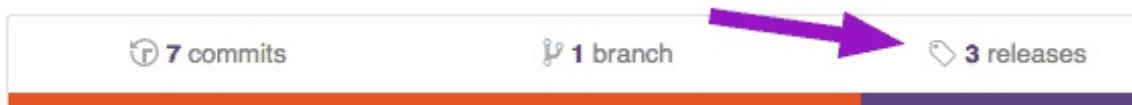
Solo un pequeño detalle relativo al comando push cuando lo usamos para enviar tags. Cuando en el comando push usamos la opción `--tags` en principio no se mandan los cambios que tengamos en el repositorio. Es decir, aunque hayamos hecho cambios en la rama master y se hayan realizado los correspondientes commits en local, si lanzamos `"git push --tags"`, únicamente los nuevos tags se van a enviar a remoto. No se enviarán los commits que se hayan podido realizar en cualquier rama.

Si queremos hacer un push de una rama en concreto, por ejemplo la rama master, y enviar los tags al mismo tiempo, entonces podríamos lanzar el siguiente comando.

```
git push origin master --tags
```

## Conclusión Git Tag

Además, en el caso de GitHub también puedes crear tags en los repositorios directamente desde la interfaz web del servicio. Desde la página principal del repositorio, en el enlace que pone "releases", puedes acceder a la información sobre las versiones etiquetadas en tu proyecto, así como etiquetar nuevas versiones.



Hemos aprendido a etiquetar estados de un proyecto con Git, algo que se realiza comúnmente para informar en el sistema de control de versiones de las releases principales durante la vida

---

de un software.

Trabajar con Git Tag como has visto no resulta nada complicado, puesto que en principio solamente son un par de comandos nuevos los que te tienes que aprender. Puedes obtener más ayuda sobre opciones para el comando git tag con "git tag -h".

Estamos seguros que con esta información tendrás suficiente para comenzar, pero si te queda alguna duda puedes consultar este [artículo de la documentación de Git](https://desarrolloweb.com/articulos/especificar-versiones-git-tag.html).

Este artículo es obra de *Miguel Angel Alvarez*  
Fue publicado / actualizado en 08/01/2020  
Disponible online en <https://desarrolloweb.com/articulos/especificar-versiones-git-tag.html>

---

## Trabajar con ramas en Git: git branch

En este artículo aprenderás a trabajar con ramas en Git. Usaremos el comando git branch para crear ramas dentro del proyecto, movernos entre ramas con checkout, fusionarlas con merge, así como trabajar con ramas de repositorios remotos.



En el día a día del trabajo con Git una de las cosas útiles que podemos hacer es trabajar con ramas. Las ramas son caminos que puede tomar el desarrollo de un software, algo que ocurre naturalmente para resolver problemas o crear nuevas funcionalidades. En la práctica permiten que nuestro proyecto pueda tener diversos estados y que los desarrolladores sean capaces de pasar de uno a otro de una manera ágil.

Ramas podrás usar en muchas situaciones. Por ejemplo imagina que estás trabajando en un proyecto y quieres implementar una nueva funcionalidad en la que sabes que quizás tengas que invertir varios días. Posiblemente sea algo experimental, que no sabes si llegarás a incorporar, o bien es algo que tienes claro que vas a querer completar, pero que, dado que te va a ocupar un tiempo indeterminado, es posible que en medio de tu trabajo tengas que tocar tu código, en el estado en el que lo tienes en producción.

Bajo el supuesto anterior lo que haces es crear una rama. Trabajas dentro de esa rama por un tiempo, pero de repente se te cuelga el servidor y te das cuenta que hay cosas en el proyecto que no están funcionando correctamente. Los cambios de la rama no están completos, así que no los puedes subir. En ese instante, lo que las ramas Git te permiten es que, lanzando un

sencillo comando, poner de nuevo el proyecto en el estado original que tenías, antes de empezar a hacer esos cambios que no has terminado. Perfecto! solucionas la incidencia, sobre el proyecto original y luego puedes volver a la rama experimental para seguir trabajando en esa idea nueva.

Llegará un momento en el que, quizás, aquellos cambios experimentales los quieras subir a producción. Entonces harás un proceso de fusión entre la rama experimental y la rama original, operación que se conoce como merge en Git.

Las aplicaciones de las ramas son, como puedes imaginar, bastante grandes. Espero haber podido explicar bien una de ellas y que te hayas podido hacer una idea antes de seguir la lectura de este artículo.

Nota: Ten en cuenta que este artículo es un tanto avanzado, pensado para personas que ya trabajan con Git. Puedes aprender cosas más básicas que damos por sabidas aquí en el [Manual de Git](#).

## Git branch

El comando `git branch` es el que usaremos principalmente para trabajar con la creación de ramas, borrado de ramas y demás. Sin embargo, no es el único comando para la operativa que veremos en este artículo, ya que existen otros subcomandos de Git útiles y necesarios para trabajar con ramas, como `checkout` para moverse entre ramas o `merge` para fusionar ramas.

Puedes comenzar tu primera práctica para trabajar con ramas. Haremos algo tan sencillo como lanzar el comando "`git branch`" a secas. Esto nos dará el listado de ramas que tengamos en un proyecto. Pero hay que advertir que las ramas de un repositorio local pueden ser distintas de las ramas de un repositorio remoto. Por ejemplo, cuando clonas un repositorio de GitHub generalmente estás clonando únicamente la rama master y no todas las ramas que se hayan creado a lo largo del tiempo. Otro ejemplo es cuando creas una rama en tu repositorio local. En este caso la rama la tendrás simplemente en tu proyecto local y no se subirá al repositorio remoto hasta que no lo especifiques. Oviamente, todas estas cosas, y otras, son las que vamos a ver en este artículo.

## La rama master

Cuando inicializamos un proyecto con Git automáticamente nos encontramos en una rama a la que se denomina "master".

Puedes ver la rama en la que te encuentras en cada instante con el comando:

```
git branch
```

Esta rama es la principal de tu proyecto y a partir de la que podrás crear nuevas ramas cuando lo necesites.

Si has hecho algún commit en tu repositorio observarás que después de lanzar el comando "git branch" nos informa el nombre de la rama como "master".

```
mcMiguel:gittest midesweb$ git branch
* master
```

Nota: Si no has hecho un commit en tu proyecto observarás que no se ha creado todavía ninguna rama y que el comando branch no produce ninguna salida.

## Crear una rama nueva

El procedimiento para crear una nueva rama es bien simple. Usando el comando branch, seguido del nombre de la rama que queremos crear.

```
git branch experimental
```

Este comando en sí no produce ninguna salida, pero podrías ver las "branches" de un proyecto con el comando "git branch", u obtener una descripción más detallada de las ramas con este otro comando:

```
git show-branch
```

Esto nos muestra todas las ramas del proyecto con sus commits realizados. La salida sería como la de la siguiente imagen.

```
mcMiguel:gittest midesweb$ git show-branch
! [experimental] primer commit
* [master] primer commit
--
+* [experimental] primer commit
```

## Pasar de una rama a otra

Para moverse entre ramas usamos el comando "git checkout" seguido del nombre de la rama que queremos que sea la activa.

```
git checkout experimental
```

esta sencilla operación tiene mucha potencia, porque nos cambiará automáticamente todos los archivos de nuestro proyecto, los de todas las carpetas, para que tengan el contenido en el que se encuentren en la correspondiente rama.



De momento en nuestro ejemplo las dos ramas tenían exactamente el mismo contenido, pero ahora podríamos empezar a hacer cambios en el proyecto experimental y sus correspondientes commit y entonces los archivos tendrán códigos diferentes, de modo que puedas ver que al pasar de una rama a otra hay cambios en los archivos.

Si estando en la rama experimental haces un par de commit, observarás que al hacer el show-branches te mostrará nuevos datos:

```
mcMiguel:gittest midesweb$ git show-branch
* [experimental] Segundo cambio experimental
! [master] primer commit
--
* [experimental] Segundo cambio experimental
* [experimental^] cambio experimental
*+ [master] primer commit
```

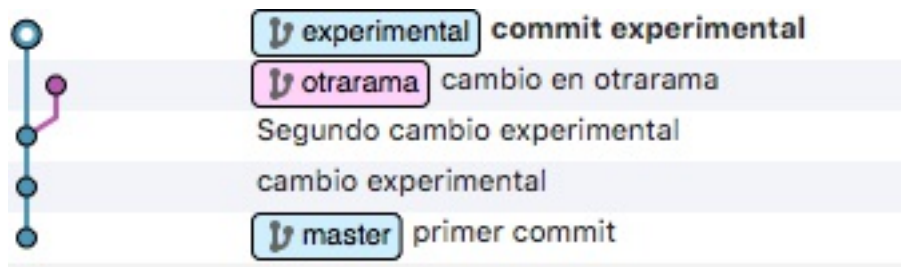
El comando checkout tiene la posibilidad de permitirte crear una rama nueva y moverte a ella en un único paso. Para crear una nueva rama y situarte sobre ella tendrás que darle un nombre y usar el parámetro -b.

```
git checkout -b otrarama
```

Como salida obtendrás el mensaje Switched to a new branch 'otrarama'. Eso quiere decir que, además de crear la rama, nuestra cabecera está apuntando hacia esta nueva branch.

Si te dedicas a editar tus ficheros, crear nuevos archivos y demás en las distintas ramas entonces podrás observar que al moverte de una a otra con `checkout` el proyecto cambia automáticamente en tu editor, mostrando el estado actual en cada una de las ramas donde te estás situando. Es algo divertido y, si eres nuevo en Git verás que es una magia que resulta bastante sorprendente.

Como estás entendiendo, el proyecto puede tener varios estados en un momento dado y tú podrás moverte de uno a otro con total libertad y sin tener que cambiar de carpeta ni nada parecido. Si usas un programa de interfaz gráfica de Git, como SourceTree o cualquier otro, podrás ver las ramas en un esquema gráfico más entendible que en la consola de comandos.



## Fusionar ramas

A medida que crees ramas y cambies el estado de las carpetas o archivos tu proyecto empezará

a divergir de una rama a otra. Llegará el momento en el que te interese fusionar ramas para poder incorporar el trabajo realizado a la rama master.

El proceso de fusionado se conoce como "merge" y puede llegar a ser muy simple o más complejo si se encuentran cambios que Git no pueda procesar de manera automática. Git para procesar los merge usa un antecesor común y comprueba los cambios que se han introducido al proyecto desde entonces, combinando el código de ambas ramas.

Para hacer un merge nos situamos en una rama, en este caso la "master", y decimos con qué otra rama se debe fusionar el código.

El siguiente comando, lanzado desde la rama "master", permite fusionarla con la rama "experimental".

```
git merge experimental
```

Un merge necesita un mensaje, igual que ocurre con los commit, por lo que al realizar ese comando se abrirá "Vim" (o cualquier otro editor de consola que tengas configurado) para que introduzcas los comentarios que juzgues oportuno. Salir de Vim lo consigues pulsando la tecla ESC y luego escribiendo ":q" y pulsando enter para aceptar ese comando. Esta operativa de indicar el mensaje se puede resumir con el comando:

```
git merge experimental -m 'Esto es un merge con mensaje'
```

En la siguiente imagen puedes ver una secuencia de comandos y su salida. Primero el cambio a la rama master "git checkout master", luego el "git branch" para confirmar en qué rama nos encontramos y por último el merge para fusionarla con la rama experimental.

```
mcMiguel:gittest midesweb$ git checkout master
Switched to branch 'master'
mcMiguel:gittest midesweb$ git branch
  experimental
* master
  otrarama
mcMiguel:gittest midesweb$ git merge experimental
Auto-merging miarchivo.txt
Merge made by the 'recursive' strategy.
 miarchivo.txt      | 4 ++++
 otro_archivo.txt  | 1 +
 2 files changed, 5 insertions(+)
 create mode 100644 otro_archivo.txt
```

Luego podremos comprobar que nuestra rama master tiene todo el código nuevo de la rama experimental y podremos hacer nuevos commits en master para seguir el desarrollo de nuestro proyecto ya con la rama principal, si es nuestro deseo.

Si tenemos un programa de Git por interfaz gráfica podremos ver el diagrama con el combinado de las ramas.



Fusionar los cambios de master en la rama en desarrollo

Durante tu trabajo en el desarrollo del proyecto gestionado con Git también puede ser normal que se vayan haciendo cambios en la rama master, o en otras ramas en desarrollo, y quieras traerlos para tu rama actual. Por ejemplo, la rama experimental está tardando varios días o semanas en completarse y mientras tanto han agregado nuevas features que quieras que esté disponibles también en la rama experimental.

Entonces seguramente querrás traerte los cambios de la rama master. Para ello, estando en la rama experimental, puedes lanzar el siguiente comando.

```
git merge master -m 'Un mensaje del merge de master en el branch experimental'
```

Ya lo tienes! ahora tu rama está actualizada con todos los cambios en master. Puedes seguir desarrollando tu rama experimental sabiendo que tienes el proyecto actualizado.

## Subir una rama al repositorio remoto (Github o similares)

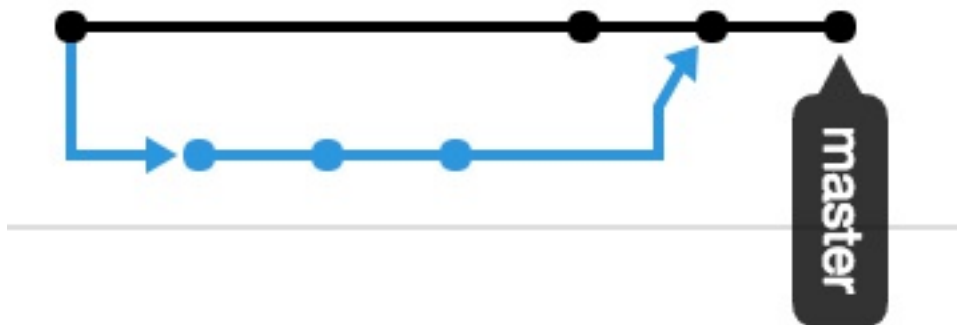
Como habíamos dicho anteriormente, por mucho que hagas la operativa descrita para crear ramas en tu ordenador, y las puedas ver en tu repositorio local con `git branch`, las ramas no se publicarán en Github o cualquier otro hosting de repositorios remoto. Para que esto ocurra tienes que realizar específicamente la acción de subir una rama determinada.

La operativa de publicar una rama en remoto la haces mediante el comando `push`, indicando la opción `-u` y el nombre de la rama que deseas subir. Por ejemplo de esta manera:

```
git push -u origin experimental
```

Así estamos haciendo un push, empujando hacia origin (que es el nombre que se suele dar al repositorio remoto), la rama con nombre "experimental".

Por cierto, si subimos el proyecto a Github podremos ver también un diagrama de las ramas que hemos ido creando y fusionando a master, en la sección Graps / Network.



## Borrar una rama

En ocasiones puede ser necesario eliminar una rama del repositorio, por ejemplo porque nos hayamos equivocado en el nombre al crearla. Aquí la operativa puede ser diferente, dependiendo de si hemos subido ya esa rama a remoto o si todavía solamente está en local.

### Borrado de la rama en local

Esto lo conseguimos con el comando `git branch`, solamente que ahora usamos la opción `"-d"` para indicar que esa rama queremos borrarla.

```
git branch -d rama_a_borrar
```

Sin embargo, puede que esta acción no nos funcione porque hayamos hecho cambios que no se hayan salvado en el repositorio remoto, o no se hayan fusionado con otras ramas. En el caso que queramos forzar el borrado de la rama, para eliminarla independientemente de si se ha hecho el push o el merge, tendrás que usar la opción `"-D"`.

```
git branch -D rama_a_borrar
```

Debes prestar especial atención a esta opción `"-D"`, ya que al eliminar de este modo pueden haber cambios que ya no se puedan recuperar. Como puedes apreciar, es bastante fácil de confundir con `"-d"`, opción más segura, ya que no permite borrado de ramas en situaciones donde se pueda perder código.

### Eliminar un branch en remoto

Si la rama que queremos eliminar está en el repositorio remoto, la operativa es un poco diferente. Tenemos que hacer un push, indicando la opción `--delete`, seguida de la rama que se desea borrar.

```
git push origin --delete rama_a_borrar
```

### Descargar una rama de remoto

A veces ocurre que se generan ramas en remoto, por ejemplo cuando han sido creadas por otros usuarios y subidas al hosting de repositorios, como GitHub o similares, y necesitamos acceder a ellas en local para verificar los cambios o continuar el trabajo. En principio esas ramas en remoto creadas por otros usuarios no están disponibles para nosotros en local, pero las podemos descargar.

El proceso para obtener una rama del repositorio remoto es bien sencillo. Lo conseguimos con el comando fetch.

```
git fetch
```

Lanzado ese comando hemos podido descargar la rama git de remoto. Ahora podemos acceder a ella con los comandos que ya conoces.

```
git checkout mi_rama_remota_descargada
```

## Conclusión

Espero que estas notas te hayan ayudado a entender las ramas de Git y puedas experimentar por tu cuenta para hacer algunos ejemplos e ir cogiendo soltura. Realmente trabajar con ramas te dará la posibilidad de sacar mucho partido a Git y organizar mejor tus prepositorios, facilitando también el trabajo en grupo.

Para acabar te dejamos un enlace muy interesante y más detallado donde [aprender más del proceso de fusión de ramas](#).

En el [Manual de Git](#) encontrarás también muchas otras informaciones y tutoriales para extraer lo mejor de esta imprescindible herramienta.

Este artículo es obra de *Miguel Angel Alvarez*  
Fue publicado / actualizado en 21/05/2021  
Disponible online en <https://desarrolloweb.com/articulos/trabajar-ramas-git.html>

## Fork en Git

Qué es un fork y para qué sirve, cuáles son los pasos para crear un fork de un repositorio Git en GitHub.



El "fork" es una de las operativas comunes con el trabajo en Git y GitHub. Básicamente sirve para crear una copia de un repositorio en tu cuenta de usuario. Ese repositorio copiado será básicamente un clon del repositorio desde el que se hace el fork, pero a partir de entonces el fork vivirá en un espacio diferente y podrá evolucionar de manera distinta, a tu propio cargo.

El fork lo podemos entender como una *rama externa de un repositorio*, colocando esa rama en un nuevo repositorio controlado por otros usuarios. Una vez hecho el fork existirán dos repositorios distintos. Inicialmente uno era copia exacta del otro, pero a medida que se vaya desarrollando y publicando cambios en uno u otro repo, ambos repositorios podrán tender a ser tan distintos como quieran cada uno de los equipos de desarrollo que los mantengan.

### Para qué necesito un Fork

Un fork es una copia de un repositorio, pero ¿por qué no clonamos el repositorio que queremos copiar y listo?

Si [haces un clon](#) normal de un repositorio, el espacio en GitHub de ese clon seguirá asociado al repositorio que has clonado. De este modo, si realizas cambios sobre el clon y los quieres publicar en GitHub, probablemente no los podrás subir.

Obviamente, si clonas un repositorio que era tuyo, podrás realizar cambios en local y subirlos a GitHub siempre que quieras. Pero si el repositorio era de otro desarrollador y tú no tenías permisos de escritura sobre él, entonces no podrás subir cambios, porque GitHub no te lo permitirá. Para este caso es donde necesitas un fork.

Como hemos dicho, un fork es una copia de un repositorio, pero creado en tu propia cuenta de GitHub, donde sí que tienes permisos de escritura. Por tanto, si tienes intención de bajarte un repositorio de GitHub para hacer cambios en él y ese repositorio no te pertenece, lo más normal es que crees un fork primero y luego clones en local tu propio fork.

#### Operativa de Pull request

El fork es un paso inicial para conseguir participar en los proyectos de las otras personas que publican código en GitHub. Cualquier contribución comienza por la realización de un fork del repositorio en el que quieres colaborar.

Una vez creado el fork, puedes realizar cambios y solicitar el pull request a través de la página



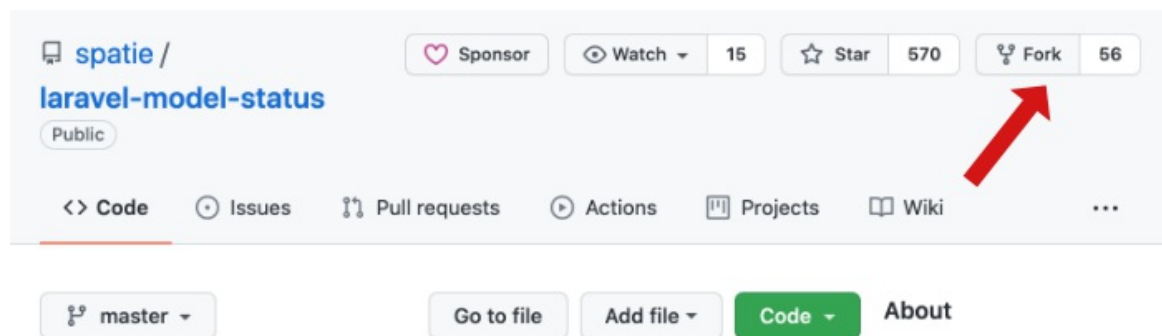
de GitHub, que es básicamente una solicitud para que tu código se fusione con el código del repositorio donde has colaborado.

Esta operativa de Pull Request es un poco más complicada que realizar un simple fork y de lo que hemos resumido en las anteriores líneas. Si quieres saber más te recomendamos leer el artículo sobre [Pull Request](#).

## Cómo hacer un fork en GitHub

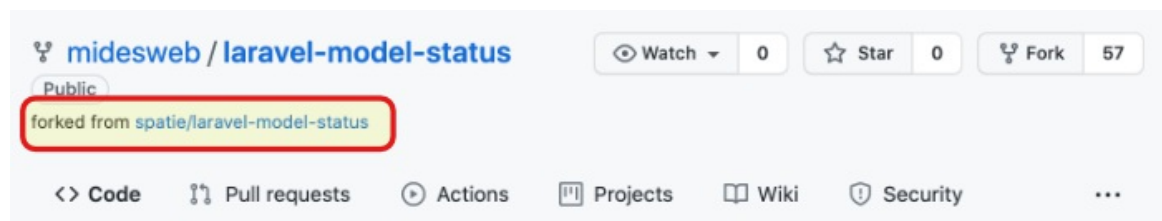
Ahora que ya sabes qué es un fork y cuándo lo puedes necesitar, vamos a aprender a hacer un fork en GitHub. El procedimiento es tan sencillo como pulsar un botón!

Simplemente accedes al repositorio que deseas forkear, y pulsas el botón "Fork".



Tendrás que hacer login en GitHub con tu cuenta para crear un fork. Si estás dentro de alguna organización te podrá aparecer una imagen para que selecciones en qué lugar quieres crear el fork, en tu cuenta personal o en alguna de tus organizaciones.

Una vez realizado el fork se creará un nuevo repositorio en tu cuenta, con una copia del repo original. En la página de GitHub se mostrará además que este repositorio es un fork de otro repositorio, el original.



Ahora podrás perfectamente hacer el clon de este fork en local, para descargarlo en tu ordenador de desarrollo. Podrás realizar cambios en el proyecto y luego hacer el commit, para seguidamente subir los cambios a GitHub.

Como los cambios los estás subiendo en un repositorio de tu propiedad, ya que el fork está

---

realizado en tu propia cuenta, GitHub te permitirá publicar las modificaciones realizadas.

## Videotutorial de un fork en GitHub

En el siguiente vídeo verás la operativa de realización de un fork en GitHub. Verás cómo realizamos un fork de un repositorio que no nos pertenece. Luego clonaremos el repositorio en local y realizaremos algunos cambios en su código.

Finalmente en el vídeo verás cómo se suben esos cambios a GitHub y cómo el repositorio nuestro, el fork, se ha actualizado correctamente.

Es un proceso sencillo, que es la antesala del procedimiento que tienes que realizar para colaborar con tu código en los proyectos de otras personas. Todo eso lo veremos más adelante, en el artículo de [Pull Request en GitHub](#).

Para ver este vídeo es necesario visitar el artículo original en:  
<https://desarrolloweb.com/articulos/fork-git>

Este artículo es obra de *Miguel Angel Alvarez*  
Fue publicado / actualizado en 10/09/2021  
Disponible online en <https://desarrolloweb.com/articulos/fork-git>

---

## Pull Request con Git

Cómo contribuir con tu código en proyectos Open Source. Realizar la operativa de Pull Request al repositorio de código de un proyecto publicado en GitHub.

En este artículo te vamos a explicar el mecanismo para contribuir en un proyecto Open Source que está publicado en GitHub, enviando código para la inclusión en el repositorio mediante el mecanismo conocido como Pull Request.

Es un proceso bastante sencillo, pero incluye varios pasos, comandos de consola, etc. Los describiremos, así como algunos de los conceptos que debes conocer para poder entender los procedimientos de Git para la gestión del código.

Al final del artículo además encontrarás un vídeo donde se demuestra todo este proceso, en vivo, paso por paso.



Nota: Pull Request es la acción básica con la que podrás aportar código para un proyecto. Recuerda además que GitHub es un escaparate excelente para un programador, pues es el mejor sitio donde se puede demostrar los conocimientos, compartiendo código de calidad y contribuyendo activamente a proyectos Open Source. Si quieres aprender a usar Git desde un punto de vista muy práctico infórmate de nuestro [curso de Git que comienza esta semana](#).

## Fork de un proyecto

Como sabes, en Git se trabaja siempre en local, con la copia del repositorio en nuestro ordenador. Fíjate que el repositorio que tienes en local no debe ser un clon de aquel con el que deseas contribuir, sino un fork propio.

Puedes hacer un Fork desde la propia página de GitHub, entrando en un repositorio y pulsando el botón "Fork", arriba a la derecha. Inmediatamente se creará en tu cuenta de GitHub un nuevo repositorio, el fork, que es la copia de estado actual del repositorio original. Podrás ver la información del repositorio, identificando que es un fork, en un área arriba de la página.



Puedes aprender más detalles del proceso de hacer un fork en este artículo: [Fork en Github](#).

En tu fork se supone que habrás realizado diversos cambios en el código, en uno o más ficheros. Tendrás entonces que hacer el commit para enviarlos a tu copia del repositorio en local.

Nota: Ya debes conocer la operación commit, si no, mira en el artículo [Primer commit](#) del Manual de Git.

Luego tendrás que enviar esos cambios a tu propio repositorio, el fork, publicado en GitHub. Eso lo consigues con el comando git push, indicando el repo al que quieres enviar los cambios y la rama donde enviarlos.

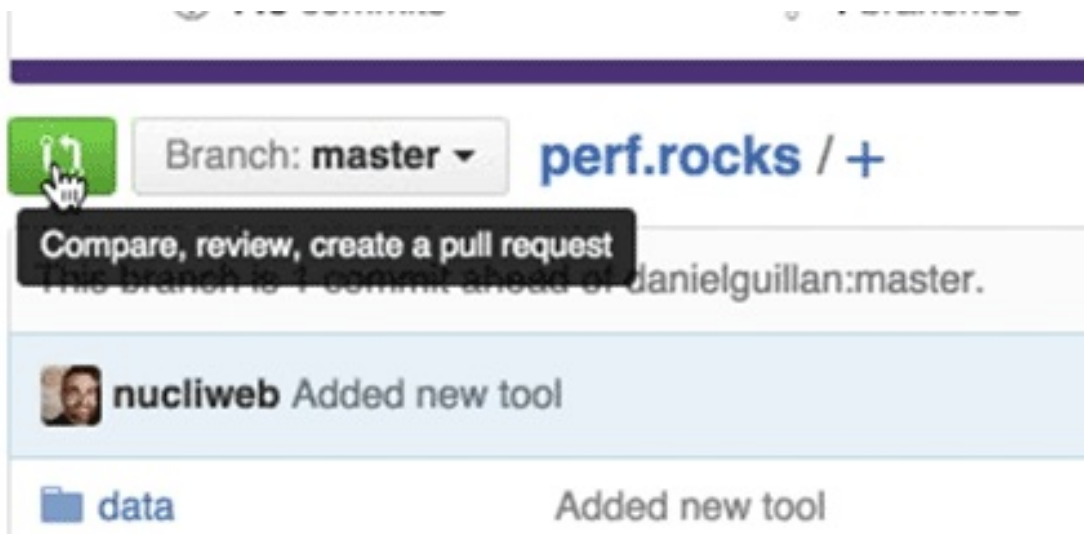
```
git push origin master
```

Con ese comando estás diciendo a Git: "haz un push a origin, que es el origen del repositorio en el servidor remoto (GitHub), especificando la rama master".

Esto conectará por Git para subir aquellos cambios que acabas de realizar y podrás verlos en GitHub en tu propio Fork.

## Hacer el Pull Request

El siguiente paso, una vez subido el commit, es realizar el Pull Request. En tu fork observarás un icono en verde que sirve para comparar y revisar para crear un Pull Request.



Hacemos clic sobre ese botón y accedemos a una página donde está realizando la comparación contra el repositorio original. Te aparecerá un detalle de todos los archivos que se han cambiado con los cambios coloreados para que los puedas entender mejor, junto con un informe del número de commit realizados y los colaboradores que los han enviado.

## Comparing changes

Choose two branches to see what's changed or to start a new pull request. If you need to, you can also compare [across forks](#).

base fork: danielguillan/perf.rocks

base: master

head fork: nucliweb/perf.rocks

compare: master

✓ Able to merge. These branches can be automatically merged.

Create pull request

Discuss and review the changes in this comparison with others.

1 commit

1 file changed

0 commit comments

1 contributor

Commits on Nov 04, 2015

nucliweb

Added new tool

3559f97

Showing 1 changed file with 6 additions and 0 deletions.

Unified

Split

6 data/tools.yml

View


		@@ -543,3 +543,9 @@
543	543	website: https://www.chromestatus.com/metrics/css/
544	544	category: 'CSS Analysis'
545	545	tags: ['metrics', 'analysis', 'css']
546	++	
547	+	name: Chrome DevTools Timeline

Luego puedes hacer clic sobre el icono "Create pull request" para lanzar la solicitud de fusión de nuestro fork con el repositorio original.

Aparecemos entonces en una página donde encontramos un formulario que nos permite indicar un mensaje sobre los motivos por los que quieres hacer el pull request y que comentes el detalle de los cambios que has realizado, qué funcionalidad has agregado, que bug se ha solucionado, etc.

## Open a pull request

Create a new pull request by comparing changes across two branches. If you need to, you can also [compare across forks](#).



base fork: **danielguillan/perf.rocks** ▾

base: **master** ▾


...

head fork: **nucliweb/perf.rocks** ▾

compare: **master** ▾

✓ **Able to merge.** These branches can be automatically merged.

Please review the [guidelines for contributing](#) to this repository.




Added new tool

I

Write

Preview

 Styling with Markdown is supported

Leave a comment

Attach files by dragging & dropping, [selecting them](#), or pasting from the clipboard.

Create pull request

Nota: si en el proyecto donde estás haciendo un pull request tienen el CONTRIBUTING.md te saldrá un enlace para conocer las políticas necesarias a seguir para poder hacer aportaciones al código. "Guidelines for contributing". Puedes ver ese enlace en la imagen anterior.

Una vez enviado el formulario con las notas del pull request, que deben servir a los dueños del repositorio original para entender tu código y valorar la posibilidad de aceptar la contribución. Al enviar el formulario nos redirige a la página del pull request, dentro del repositorio original, donde encontrarás que tu usuario de GitHub ha solicitado un merge al proyecto, junto con una serie de informaciones relacionadas, el repositorio original, el fork que ha pedido el pull request, etc. En esa página no puedes hacer nada más, salvo realizar alguna comunicación con los administradores del repositorio original, simplemente enviando algún mensaje.



## Added new tool #91

Open nucliweb wants to merge 1 commit into danielguillan:master from nucliweb:master

Edit

Conversation 0 Commits 1 Files changed 1

+6 -0



nucliweb commented just now

No description provided.

Added new tool

3559f97

Labels

None yet

Milestone

No milestone

Assignee

No one assigned

Notifications

Unsubscribe

You're receiving notifications because you authored the thread.

1 participant

Add more commits by pushing to the master branch on nucliweb/perf.rocks.



This branch is up-to-date with the base branch

Only those with write access to this repository can merge pull requests.



Write

Preview

Styling with Markdown is supported

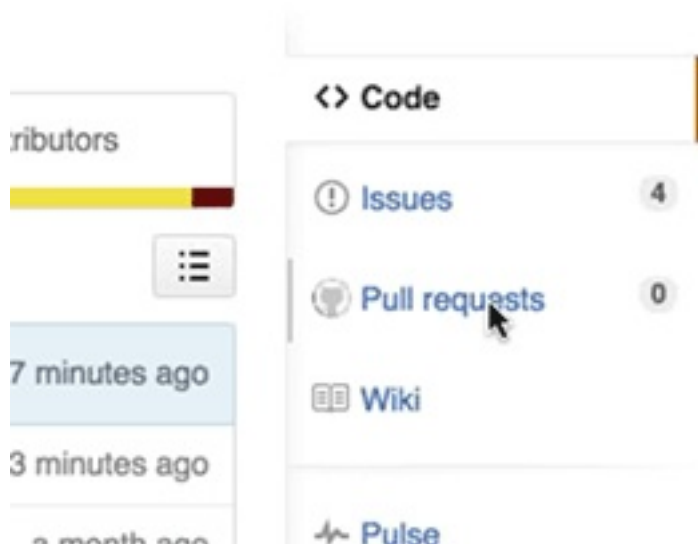
Leave a comment

De manera automática, a los administradores les llegará un correo y una notificación a través del sitio de GitHub para que revisen el pull request y acepten o no los cambios al código. Si además entras en la home del repositorio verás que en la parte de la derecha aparece el número de pull request, donde debería haber al menos uno, el que acabas de hacer.

## Pull Request aceptado

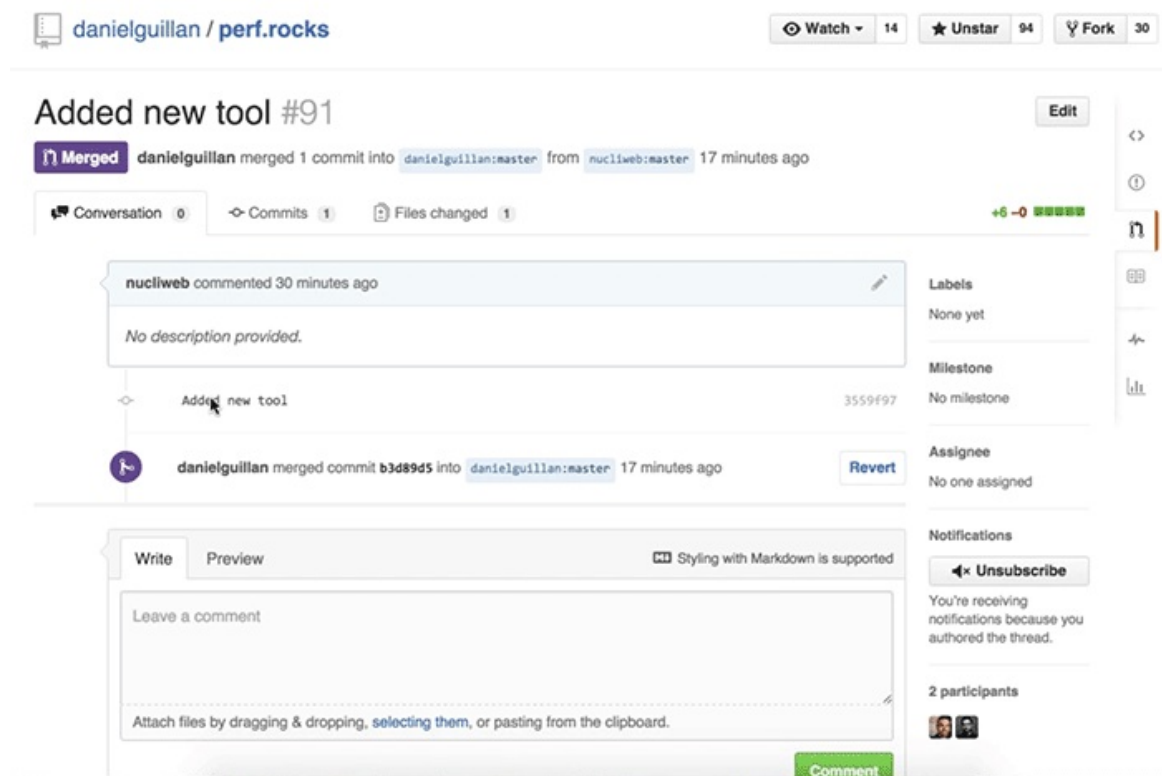
En el caso que los administradores acepten el pull request que acabas de realizar, recibirás las correspondientes notificaciones. Si ha sido aceptado el Pull Request se fusionará el código de tu propio fork con el proyecto al que estás contribuyendo.

En la parte de la derecha observarás que el pull request ya no está, pero si pulsas sobre el enlace "Pull request" encontrarás un histórico de las operaciones realizadas.



Si vas a los pull request cerrados (closed) encontrarás un listado de la actividad. Deberías localizar el tuyo y un enlace que nos lleva a una página con algunas informaciones útiles del pull request cerrado.

El estado de tu Pull Request observarás que está en "Merged" y que el proceso está completado. Desde allí podrás comunicar nuevamente con los gestores del repo original, enviando si lo deseas mensajes.



Con esta serie de pasos hemos aprendido a realizar una operación de Pull Request, aportando código en un repositorio Open Source. Esperamos que puedas poner en práctica esta actividad que sin duda resultará enriquecedora tanto para el software libre como para tu propio perfil de GitHub y tu formación profesional.

En el vídeo que puedes ver a continuación encontrarás todo este proceso paso por paso, relatado en mayor detalle. Recuerda además que esta es una de las muchas cosas que vas a aprender y practicar en el próximo [Curso de Git Práctico para desarrolladores y diseñadores](#).

Para ver este vídeo es necesario visitar el artículo original en:  
<https://desarrolloweb.com/articulos/pull-request-git.html>

Este artículo es obra de \_Joan Leon\_  
Fue publicado / actualizado en 09/11/2015  
Disponible online en <https://desarrolloweb.com/articulos/pull-request-git.html>



# Herramientas basadas en Git

Existen en el mercado numerosas herramientas basadas en Git que merece la pena conocer, ya que nos ayudan a desarrollar muchas de las tareas habituales en el desarrollo de software. Muchos de estos programas o servicios alientan el uso de buenas prácticas y proponen flujos de trabajo deseables para el desarrollo de software moderno.

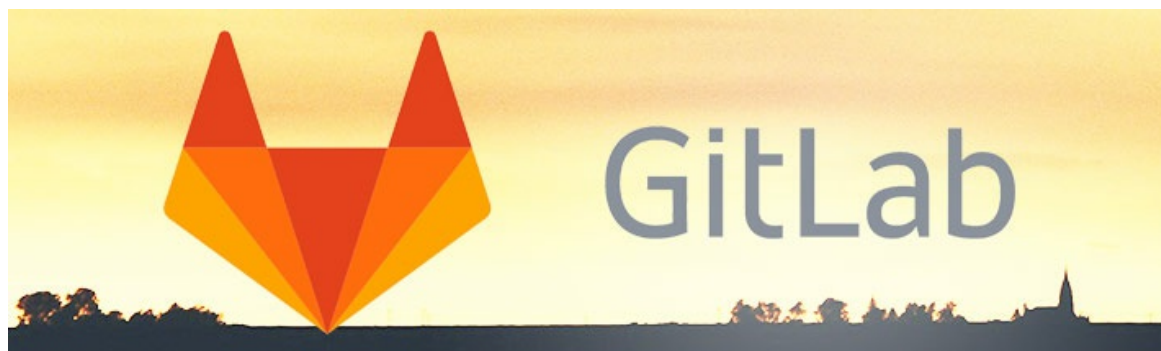
## Introducción a GitLab

Qué es GitLab, qué le diferencia de sus competidores como GitHub o Bitbucket y qué herramientas engloba, además de los repositorios remotos Git, para gestionar los proyectos de desarrollo.

GitLab nació como un sistema de alojamiento de repositorios Git, es decir, un hosting para proyectos gestionados por el sistema de versiones Git. Sin embargo, alrededor de esta herramienta han surgido muchas otras herramientas muy interesantes para programadores y equipos de desarrollo, que envuelven todo el flujo del desarrollo y el despliegue de aplicaciones, test, etc.

Sin duda, para dar una idea de lo que es GitLab, lo más rápido sería compararlo con uno de sus competidores, GitHub, pues éste último es especialmente conocido en el mundo del desarrollo de software. Todos más o menos estamos familiarizados con lo que ofrece, la gestión de repositorios, las issues, los pull request, GitHub Pages, etc. GitLab sería algo muy similar a lo que encontramos en GitHub, aunque a veces con otros nombres. Otras alternativas de programas o servicios para Git como Bitbucket están muy por detrás en posibilidades.

Aunque GitHub es un monstruo, en cuanto a número de repositorios y funcionalidad, GitLab ha conseguido llegar aún más lejos, ofreciendo un conjunto más amplio de servicios que harán las delicias no solo de desarrolladores, sino también de devops.



## Instalar en tu servidor o usarlo como servicio web

La principal diferencia entre GitLab y sus competidores es que GitLab se ofrece como un

---

software libre que puedes descargar e instalar en cualquier servidor. Esta posibilidad permite usar GitLab para una empresa, profesional u organización en sus propios servidores, sin ningún coste adicional.

Nota: Obviamente, si instalas tu propio GitLab para usarlo en tu propia empresa, te obliga a mantener el servidor, configurarlo, actualizar el software, etc. Trabajo que no deja de representar un gasto de tiempo, lo que al final se traduce en dinero. Es el motivo por lo que muchas empresas prefieren acabar pagando para disfrutar de GitLab como servicio.

La otra alternativa es usar GitLab directamente de [GitLab.com](https://gitlab.com), pagando por el servicio. Esto permite disponer de todo el poder de GitLab y sus herramientas colindantes, sin invertir tiempo en configuración, aprovechando sus ventajas desde el primer minuto. Además, las versiones del servicio "en la nube" tienen muchas herramientas adicionales, funcionalidades que superan con diferencia a la versión que se ofrece para instalar como software libre.

Por último, GitLab también se ofrece sin coste para publicar repositorios de software libre, igual que su competidor GitHub. En este caso, aunque GitLab pueda disponer de algunos servicios extra que justifiquen trabajar con la herramienta, lo cierto es que GitHub sigue siendo el sitio preferido donde ubicar un proyecto, dado que diversos sistemas de gestión de dependencias, como npm, Composer, etc., trabajan directamente contra ellos.

Nota: Para quien desee aprovechar las ventajas de GitLab en un proyecto publicado en GitHub existe la posibilidad de hacer un espejo del repositorio. GitLab, cada vez que el repo se actualiza en GitHub es capaz de traerse los cambios. Los desarrolladores desde GitLab, con el repositorio siempre actualizado, pueden realizar uso de los servicios extra, como los procesos de integración continua o despliegue continuo.

En resumen, si queremos usar GitLab gratis, para alojar en remoto un repositorio Git en general, la mejor alternativa es que lo instalemos gratuitamente en una de nuestras máquinas. Sin embargo, si queremos ahorrarnos tiempo y no nos importa pagar un poco, la versión en la nube está mucho más completa y nos permite olvidarnos del servicio y concentrarnos en el desarrollo de nuestros programas.

## Cómo usar GitLab

GitLab es una herramienta basada en Git, que usas de la misma manera que cualquier otra herramienta similar. Generalmente usas Git a través de la línea de comandos, o a través de programas de interfaz gráfica, o del propio editor de código. Toda esa operativa que ya conoces y que hemos explicado en el [Manual de Git](#), no cambia.

Además del hosting remoto para repositorios GitLab ofrece una interfaz web para controlar el repositorio y muchas otras herramientas. Ofrece la posibilidad de examinar el código en cualquiera de sus versiones, realizar acciones relacionadas con el sistema de repositorios como mergear el código de versiones de proyecto o gestionar las "pull request" (que en GitLab se

---

llaman "merge request"), gestionar problemática de tu software diversa, automatizar procesos como el despliegue o la ejecución de pruebas del software, etc. Toda esta operativa la realizas, o configuras, en GitLab por medio de una web.

Por tanto, para usar GitLab simplemente necesitas las mismas herramientas que ya utilizas en tu día a día, el terminal o un programa de interfaz gráfica para gestionar tu repositorio, así como el navegador web para acceder a el ecosistema de herramientas disponible en el sitio de GitLab. Por supuesto, todas estas herramientas las puedes usar desde cualquier ordenador conectado a Internet, independientemente de su sistema operativo.

Nota: ya, si nos referimos a cómo usar nuestra propia instalación de GitLab, y qué necesitamos para instalar el software libre en nuestros propios servidores, la respuesta es que lo más normal es que instales GitLab en un servidor Linux, ya que es su entorno natural. Generalmente instalar GitLab es tan sencillo como instalar cualquier otro software en Linux, solo que necesitarás configurar además una serie de programas adicionales, que usa GitLab por debajo, para que el servicio funcione de manera fina. Esta parte puede que no sea tan fácil para una persona que no tenga conocimientos sólidos de administración de sistemas.

## Funcionalidades de GitLab

En GitLab podemos gestionar principalmente proyectos, Grupos y Snippets. Los proyectos son los protagonistas del sistema, básicamente repositorios de software gestionados por GitLab y todo el ecosistema GitLab. Los grupos son básicamente empresas y usuarios. Los snippets por su parte son como pedazos de código que puedes dejar para hacer cualquier cosa.

Como decimos, dentro de los proyectos es donde se aglutinan la mayoría de las funcionalidades que vamos a resumir:

### Overview:

Es un listado de todo el proyecto, los archivos, los README.md. Es parecido a lo que vemos cuando accedemos a un proyecto con GitHub. Te da el resumen del repositorio, archivos, commits, etc.

Luego tiene dos subsecciones: En Activity del proyecto te ofrece toda la actividad, de una manera estadística. En Cycle Analytics además te ofrece algo muy novedoso, no disponible en otras herramientas. Básicamente informa el tiempo que se tarda en realizar una funcionalidad, desde que tienes la idea hasta que se incorpora al software, de modo que cualquier persona, incluso sin conocimientos de programación, puede saber el tiempo que ocupó el hacer las tareas. Una información muy valiosa que puede ayudar a futuro a estimar mejor el tiempo de trabajo necesario para nuevas funcionalidades. Obviamente, cuantas más issues tengas en el sistema, más datos tendrás para saber el tiempo que necesitas para las próximas tareas.

### Repository:



---

Dentro de la sección "Repository" tenemos varias opciones diversas que afectan al repositorio del proyecto.

Tenemos "Files", donde se puede navegar por los directorios y archivos, cuyo código podemos ver, e incluso editar los ficheros. Está disponible una visualización por ramas y dispone de utilidades diversas para poder hacer cosas relacionadas con el repositorio remoto, ahorrando la necesidad de lanzar comandos. Tiene un buscador de archivos muy potente.

En "Commits" encontramos un listado de todos los commits realizados contra el repositorio, junto con datos específicos de cada uno de ellos.

Las ramas "Branches" sirven para ver las ramas que tenemos en el repositorio.

La siguiente sección, "Tags", es importante también, pues es el mecanismo disponible en Git para definir puntos del estado del código, correspondientes a cada release.

Además esta sección tiene otras áreas también importantes, que os dejamos para vuestra propia investigación. Especialmente sería destacable la parte de "Locked files", disponible solo en GitLab como servicio, que es algo que no ofrece el propio sistema de control de versiones Git pero que han implementado dentro de GitLab, que permite bloquear un fichero para que solo ciertas personas lo puedan editar.

#### Issues:

Este es otra de las grandes utilidades de GitLab, que permite definir cualquier problema que se detecta en el software y darle seguimiento. Seguro que las conocemos porque es una de las partes fundamentales de GitHub y habremos navegado por ellas en decenas de ocasiones.

Básicamente nos permite ver las issues generadas en un proyecto, mantener discusiones sobre ellas, y controlar los flujos de trabajo para su resolución, permitiendo definir las personas que deben resolverla, el tiempo estimado y el usado, la fecha límite, el peso de las tareas, etc.

En GitLab han publicado otra interesante innovación que es un tablero de issues (Issue Boards), que permite visualizar las tareas, de una manera similar a los boards de Trello. Como gestores somos capaces de definir los tableros y las etiquetas. GitLab, por medio de la gestión de las Issues, es capaz actualizar el estado de las tareas, permitiendo visualizar su evolución por medio de los tableros.

Otra cosa muy interesante es el "Service desk", que te ofrece un email que lo puedes proporcionar al cliente. Sin que el cliente se registre en GitLab, ni tenga acceso al proyecto, puede enviar mensajes a ese email, adjuntando texto, imágenes y archivos. GitLab, al recibir el correo, da de alta automáticamente una issue con ese contenido.

#### Merge Request:

Son como las Pull Request de GitHub. Te permiten controlar todas las solicitudes de combinación o unión de código de distintas ramas o forks. Es muy importante que los merges se resuelvan mediante la interfaz gráfica, ya que nos ofrece muchas posibilidades interesantes,

---

como automatización de tests, la posibilidad de revisión de los cambios por parte de componentes del equipo, implementar diversas políticas de control sobre el código del proyecto, etc.

CI/CD:

Es una de las maravillas que dispone GitLab, una herramienta sencilla y muy útil para los procesos de integración continua y despliegue continuo. Existen muchas herramientas que se pueden integrar para automatizar los procesos y llegar a crear flujos de trabajo completamente automatizados. De modo que se lancen los test y si todo va bien se puedan realizar una serie de tareas definida, que pueden llegar a producir el despliegue automático de las aplicaciones.

Solo disponer de esta sección es suficiente motivo para pasarse a GitLab. No llega la complejidad de herramientas específicas como Jenkins, pero resuelve de manera muy potente problemas similares.

## GitLab es mucho más

Gitlab no se queda aquí, existen decenas de herramientas para hacer un montón de procesos, plugins, integraciones con diversos servicios útiles en el día a día de los equipos de desarrollo, etc.

Tenemos un vídeo en el que explicamos con mucho más detalle todas las posibilidades de GitLab y lo comparamos con otros sistemas similares. Además estamos impartiendo el [Curso de GitLab en EscuelaIT](#), en el que podrás aprender lo básico para comenzar con Git, pero sobre todo la manera de sacar todo el partido de las herramientas disponibles en GitLab.

Puedes dar una vuelta por GitLab y examinar las opciones y posibilidades. Al ritmo que van, seguro que cuando entres habrán sacado novedades representativas. Pero si quieres que te guiemos en esa visita, puedes consultar el siguiente vídeo. También encontrarás en este vídeo un resumen de lo que se denomina el "[GitLab Flow](#)", un conjunto de prácticas que son aconsejables para mantener un flujo adecuado en todo lo que sería el desarrollo de software, que nos asegure la calidad y facilidad de mantenimiento y trabajo en equipo.

Para ver este vídeo es necesario visitar el artículo original en:  
<https://desarrolloweb.com/articulos/introduccion-gitlab.html>

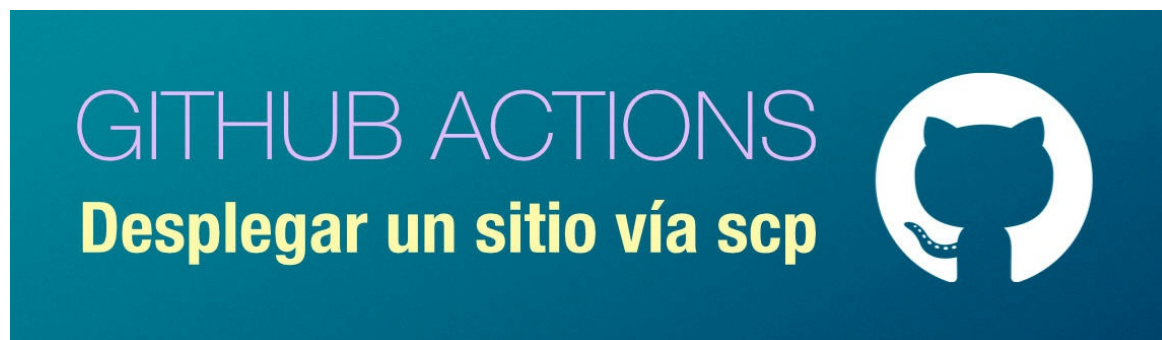
Este artículo es obra de *Jon Torrado*  
Fue publicado / actualizado en 12/10/2017  
Disponible online en <https://desarrolloweb.com/articulos/introduccion-gitlab.html>

---

## Desplegar un sitio web con GitHub Actions

---

Usar GitHub Actions para desplegar un sitio web en cualquier servidor al hacer un push en el repositorio.



En este artículo vamos a ver una pequeña práctica con GitHub Actions, que nos permite desplegar un sitio web de manera automatizada, cada vez que se actualice una rama de un repositorio en GitHub.

Para acometer esta acción desde GitHub Actions crearemos un workflow sencillo que se ejecute al hacer un push y que mande los archivos al servidor por [SCP](#). Por supuesto, necesitaremos un acceso ssh al servidor para que esta acción sea posible, con nuestro usuario y la correspondiente llave ssh para poder realizar el proceso de autenticación sobre el servidor.

## Qué son GitHub Actions

Antes que nada y por si alguien no lo sabe todavía, GitHub Actions es uno de los servicios recientes de GitHub que permite crear todo tipo de flujos de trabajo para la automatización de tareas cuando ocurran cosas en el repositorio.

Por ejemplo, cuando se realiza un push a una determinada rama del repositorio se pueden correr las pruebas o validar si el código tiene los parámetros de calidad deseados. Las acciones pueden ser sencillas o muy complejas, como correr las pruebas, compilar un software, crear un servidor nuevo para desplegar una aplicación en preproducción y en general cualquier flujo de trabajo totalmente personalizado que requiera un proyecto.

GitHub Actions es un servicio gratuito para cualquier usuario, aunque solamente por una cuota de uso limitada. Si necesitamos más minutos de procesamiento de GitHub Actions entonces sería necesario contar con una cuenta de pago.

## Crear un workflow en GitHub Actions

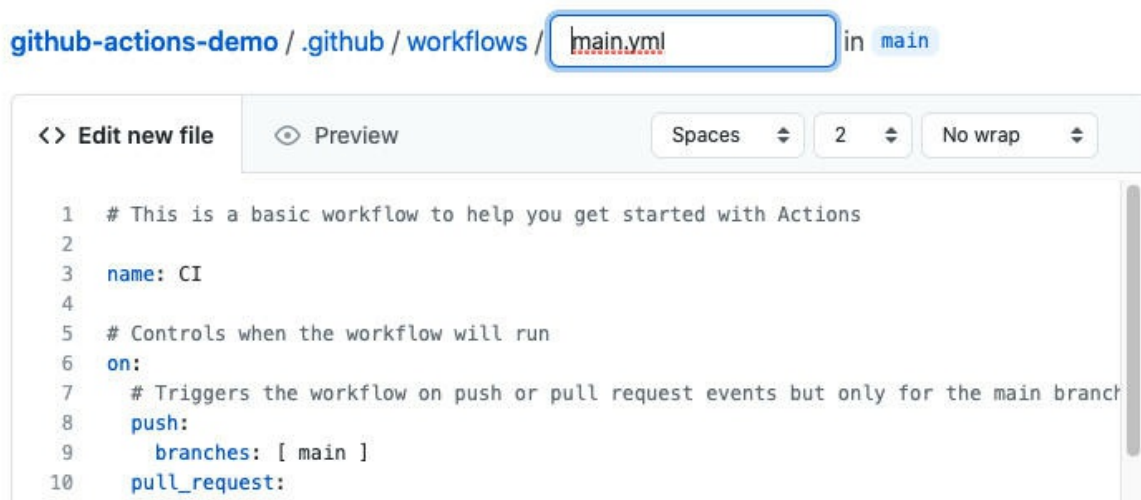
En un repositorio GitHub encontramos una sección llamada "Actions". Desde aquí podemos crear nuevas acciones fácilmente. Si pulsamos el enlace "Actions" y no hemos creado ninguna acción todavía nos llevará a una página donde se explicarán algunas de las posibilidades de esta herramienta.

Podemos empezar pulsando el enlace que dice "set up a workflow yourself". Esto nos llevará a otra página desde donde podemos editar un código de la acción o "workflow".

## Archivos yml para definir los workflows

Los flujos de trabajo en GitHub Actions se definen en archivos de código con extensión "yaml". Son archivos en un formato llamado Yaml que quizás ya conozcas. Si no es así, puedes entenderlos como un JSON pero más ligero, ya que no requieren el uso de llaves. En vez de colocar llaves de inicio y de cierre en Yaml lo que se hace es indentar el código.

Al crear tu primer workflow ya te aportan un archivo llamado "main.yml" de ejemplo, que puedes observar para aprender algunas cosillas. El archivo viene con muchos comentarios, por lo que es sencillo entender qué se hace en cada punto.



Ese código del main.yml de ejemplo lo podemos borrar para crear nuestra propia acción personalizada. Y por supuesto también podemos cambiar el nombre al archivo, para llamarlo algo como "deploy.yml" en vez de "main.yml".

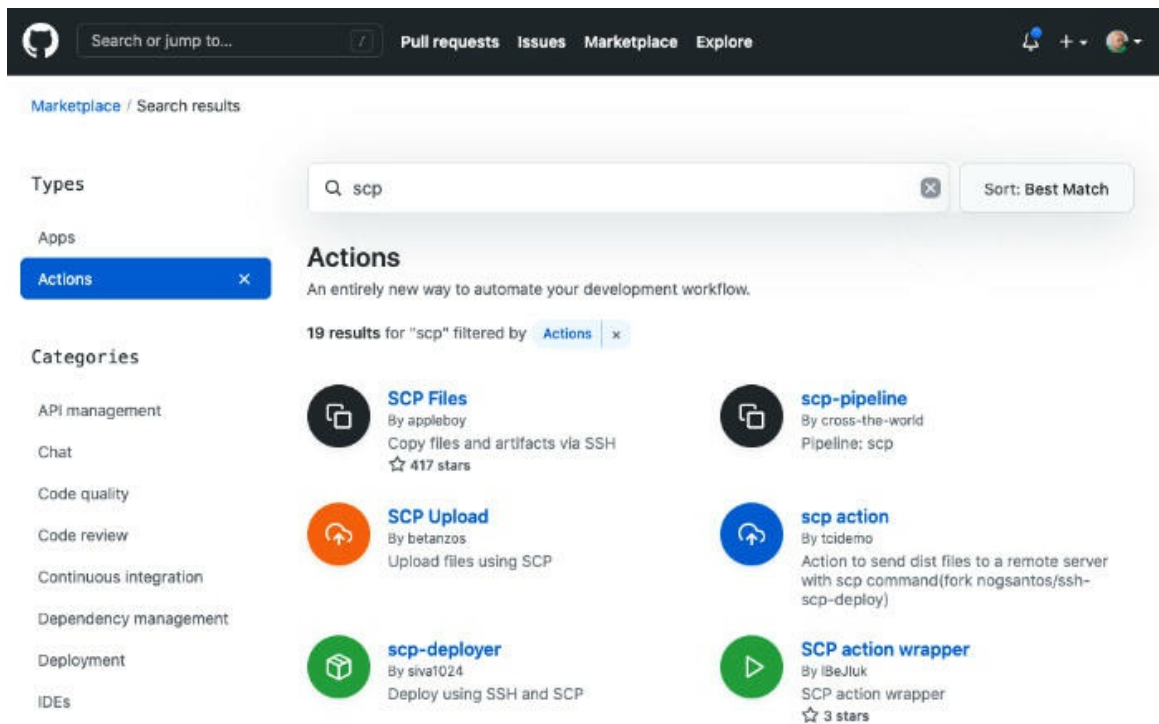
Enseguida veremos el código de nuestra acción personalizada, pero antes vamos a explicar cómo conseguir fácilmente workflows para hacer todo tipo de procedimientos imaginables.

## Marketplace de GitHub Actions

GitHub tiene un marketplace de actions desde donde puedes obtener trabajos ya listos para realizar todo tipo de workflows. Gracias a estos códigos puedes ahorrarte mucho tiempo de aprendizaje e investigación, yendo directamente al grano.

<https://github.com/marketplace?type=actions>

Desde aquí podemos hacer una búsqueda por "scp" y encontraremos varias acciones para hacer cosas que tengan que ver con este comando de Linux.



Nosotros vamos a ir a lo fácil, escogiendo una de las acciones que tienen más estrellitas en GitHub. La acción que vamos a usar se llama "SCP Files" y está creada por un desarrollador con el nick "appleboy".



## SCP for GitHub Actions

<https://github.com/marketplace/actions/scp-files>

En la propia página de la acción encontramos ya una serie de ejemplos que nos sirven para entender cómo se usa. Desde aquí podemos fácilmente encontrar código de base para nuestro workflow en el archivo Yaml.

### Creando el Workflow personalizado

Ahora que sabemos de dónde tirar, vamos a crear ya nuestro propio Workflow de GitHub Actions.

El código sería como este:

```
name: Deploy

on: [push]

jobs:
  deploy:

    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@master
      - name: Copiar el contenido del repositorio con scp
        uses: appleboy/scp-action@master
        env:
          HOST: ${ secrets.HOST }
          USERNAME: ${ secrets.USERNAME }
          PORT: ${ secrets.PORT }
          KEY: ${ secrets.SSHKEY }
      with:
        source: "site/"
        target: "/var/www/html"
```

- name: Deploy nos sirve para indicar cómo se llama esta action
- on: [push] quiere decir que se ejecutará siempre que hagamos push al repositorio
- uses: actions/checkout@master sirve para enviar el contenido de nuestro repositorio a el entorno de trabajo de GitHub, de modo que el workflow pueda acceder a nuestros archivos.
- name: Copiar el contenido del repositorio con scp es el nombre de nuestra acción
- uses: appleboy/scp-action@master indica el código de la acción que hemos encontrado en el marketplace, que hace el trabajo duro para enviar los archivos por scp.
- env: indica toda la serie de datos que requiere esta acción, como el host, el nombre de usuario, etc. Luego te explicamos cómo definir esos datos.
- with: sirve para indicar la configuración extra, en la que definimos dónde están los archivos de origen, que se van a llevar a producción y la carpeta del servidor donde los vamos a colocar.

Si guardamos el archivo con el botón verde "Start commit" habremos hecho todo lo necesario para que las acciones se ejecuten, porque ese botón realiza el commit y luego el push al repositorio. Tal como hemos configurado la acción, al procesarse el push se ejecutará.

Sin embargo, si accedemos a la pestaña "Actions" de nuestro repositorio observaremos que la acción habrá fallado. Si está en ejecución todavía veremos que fallará en unos instantes.

## Create deploy.yml

Deploy #1: Commit 68848f9 pushed by midesweb

Cuando están en ejecución las actions nos aparecerán en naranja. En esos momentos podemos hacer clic y ver todo el proceso de ejecución actual, examinando cómo van cada



---

una de las tareas planificadas. También podremos ver el log de las actions una vez acaben tanto si fallan como si van bien.

No te preocupes, es normal que haya fallado, dado que todavía no hemos definido los secretos para poder saber a qué servidor deben enviarse los archivos y las credenciales de acceso.

Una vez creado el código del workflow puedes hacer el pull en tu repositorio local para editarlo en local con tu propio editor y subirlo con el correspondiente commit y luego el push. También podrías editarlo directamente desde el sitio de GitHub accediendo a la carpeta y pulsando el botón para editar el archivo.

## Secrets en GitHub

Todos los archivos yaml de las acciones se colocan en el repositorio, en una carpeta que se ha creado dentro de ".github/workflows". Ese código estará visible para todas las personas que tengan acceso al repositorio, por ejemplo todos los desarrolladores, o si el repositorio es público, todas las personas que accedan al repo.

Dado que muchas personas pueden tener acceso al código yaml de las GitHub Actions es importante no publicar en ese archivo ninguna información sensible, como claves de los servidores, especialmente cuando el repositorio es público.

Por supuesto, GitHub tiene una zona segura donde colocar los denominados secretos, es decir, todo lo que son las llaves ssh, usuarios y contraseñas. Esta zona está en la parte de "settings" y luego "secrets".

Aquí pulsamos el botón "New repository secret" y nos llevará a una sección donde podemos crear los secretos que vamos a necesitar. Cada uno de ellos tiene un nombre y un valor. Por ejemplo "HOST" y el valor será la IP del servidor que queramos usar para desplegar el proyecto.

## Actions secrets / New secret

Name

HOST

Value





0.1.2.3|

Add secret

Para nuestro workflow `deploy.yml` tenemos que crear los siguientes secretos:

- **HOST:** La IP del servidor
- **USERNAME:** el nombre de usuario con el que accedemos al servidor
- **PORT:** el puerto de conexiones ssh, generalmente 22
- **SSHKEY:** la llave ssh que usamos para conectar

Una vez creados estos secretos los podremos ver en la página de "secrets" y los podremos editar si fuera necesario, en caso que nos hayamos equivocado con sus valores o nombres.

Repository secrets		
 HOST	Updated 2 minutes ago	<button>Update</button> <button>Remove</button>
 PORT	Updated 10 minutes ago	<button>Update</button> <button>Remove</button>
 SSHKEY	Updated now	<button>Update</button> <button>Remove</button>
 USERNAME	Updated 2 minutes ago	<button>Update</button> <button>Remove</button>

## Ejecutar de nuevo las acciones

Cada vez que hagamos un push al repositorio se ejecutarán de nuevo las acciones. De modo que para poder correrlas de nuevo simplemente actualizamos el repositorio y hacemos un push.

Ahora, si hemos colocado correctamente los secretos, veremos que la acción se ejecuta correctamente.

Durante la ejecución es posible ver el progreso haciendo clic en la acción que se está ejecutando.

El despliegue podría fallar por diferentes razones, algunas de las más típicas podrían ser:

- No hemos colocado correctamente los secretos, ya sea porque hemos escrito mal los nombres de los secretos o sus valores
- No hemos colocado bien la ruta donde están los archivos que se van a desplegar (source)
- No hemos colocado bien la ruta del servidor donde colocar los archivos (target)
- No hemos indentado correctamente el archivo .yaml
- No hemos escrito bien los nombres de las actions que estamos usando.

Si ha ido mal, aparecerá en rojo la acción y podremos ver, si hacemos clic, el motivo del fallo.

Si ha ido bien, la acción aparecerá en verde y podremos comprobar si los archivos han subido al servidor.


**quitar carpeta site**

Deploy #6: Commit d5f80ab pushed by midesweb

main

 1 hour ago

 25s

...

En el caso que haya ido correctamente también podremos hacer una consulta de la salida de la acción haciendo click en ella.

Quizás habrás observado que, al subir los archivos al servidor se ha creado una carpeta llamada "site", que es donde están los archivos que queremos llevar a producción, en la estructura de carpetas del repo original.

Quizás los archivos los queríamos situar en la carpeta raíz de publicación del servidor y no en la carpeta "site" como están en el repositorio. Esto se puede configurar añadiendo la línea "strip\_components: 1" en la parte del "with".

El código quedará así:

```
name: Deploy

on: [push]

jobs:
  deploy:

    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@master
      - name: Copiar el contenido del repositorio con scp
        uses: appleboy/scp-action@master
        env:
          HOST: ${ secrets.HOST }
          USERNAME: ${ secrets.USERNAME }
          PORT: ${ secrets.PORT }
          KEY: ${ secrets.SSHKEY }
        with:
          source: "site/"
          target: "/var/www/html"
          strip_components: 1
```

Ahora los archivos de la carpeta "site" se colocarán en el servidor directamente sobre /var/www/html.

## Conclusión

Hemos aprendido qué son las GitHub Actions y hemos practicado con una primera acción sencilla pero muy útil para automatizar el despliegue de un proyecto en el servidor, vía el comando scp.

El proceso ha sido bastante sencillo gracias al marketplace de actions de GitHub, que nos da realizada la parte más compleja de generar las acciones. Seguramente no funcione a la primera, pero con un poco de prueba y error verás que consigues despegar el sitio de manera automatizada cada vez que hagas push en el repositorio.

Si quieres aprender más sobre GitHub Actions te recomendamos acceder al [curso de CI/CD](#) de EscuelaIT, donde se explican workflows no solo en GitHub, sino en muchos otros servicios populares.

El repositorio de este demo de GitHub Actions lo tienes en GitHub:

---

<https://github.com/deswebcom/github-actions-demo>

Este artículo es obra de *Miguel Angel Alvarez*

Fue publicado / actualizado en 25/06/2021

Disponible online en <https://desarrolloweb.com/articulos/desplegar-github-actions>