

Miguel Díaz Rubio & Erik S. de Erice

GUÍA DE GIT

1. Bienvenida

Bienvenido a esta **Guía de Git en español**, donde aprenderás los conceptos más importantes de **Git**, así como algunos trucos y consejos que te harán la vida más fácil.

Git es a día de hoy la opción más reconocida y utilizada en el mercado para la gestión de código fuente. Si estás utilizando en estos momentos otro sistema, intentaremos hacer todo lo posible a lo largo de esta guía para convertirte a **Git**.

Antes de nada, quizás ya te estarás haciendo la pregunta: ¿necesito yo realmente un control de versiones para lo que hago? La respuesta es sí, por supuesto, seas quien seas, y hagas el tipo de proyectos que hagas.

Como verás a lo largo de esta guía, utilizar **Git** es algo muy sencillo y que te va a permitir trabajar de forma más organizada y segura. Te va a garantizar poder resolver las problemáticas clásicas que se suelen producir al desarrollar de forma individual, pero sobre todo cuando trabajas en equipos pequeños, medianos o grandes.

Aunque siempre que se habla de **Git** el autor se centra mucho en el mundo del **desarrollo de software**, todos los conceptos que vas a aprender aquí sirven exactamente igual para cualquier otro escenario. Y es que **no sólo los programadores pueden beneficiarse de su uso**, sino que también puedes utilizar Git para el control de versiones de tus archivos ofimáticos, documentos de diseño gráfico, etc.

En esta guía te vas a centrar en aprender algunos de los comandos Git más importantes, y para ello utilizarás siempre el terminal y su línea de comandos, ya que aunque en el futuro probablemente te ayudarás del uso de interfaces gráficas, creo que muy importante conocer la base y los comandos para estar preparados para utilizar **Git** en cualquier situación, sin necesidad de tener ningún software concreto.

El autor

Mi nombre es **Miguel Díaz Rubio** y soy **consultor informático** desde principios del año 2000 (sí, me perdí el efecto 2000). Desde entonces he estado trabajando principalmente en grandes empresas de consultoría, y dedicándome en los inicios a la programación y en la actualidad a la gestión de proyectos de desarrollo de software de gran envergadura.

Desde el año 2010 mantengo, en la medida en que mi vida personal y profesional me permiten, el blog www.migueldiazrubio.com dedicado a tutoriales sobre el mundo del desarrollo para iOS. Este blog empezó como algo totalmente personal y orientado a aprender enseñando, pero en estos momentos es una gran aventura que comparto con mi socio y amigo Erik Sebastián de Erice.

Una de las grandes ventajas que me ha supuesto esta aventura, es conservar y enriquecer mi entusiasmo y conocimientos sobre programación y tecnología en general.

Para cualquier cosa que necesites, puedes contactar conmigo en info@migueldiazrubio.com, o en twitter en [@migueldiazrubio](https://twitter.com/migueldiazrubio).

No te entretengo más y te dejo con esta guía de **Git**, que hemos preparado con todo el cariño del mundo, y que esperamos que te ayude a introducir **Git** en tu vida a partir de su lectura.

2. ¿Qué es Git?

Git es un sistema de gestión de código fuente, inventado en el año 2005 por Linus Torvalds, creador a su vez del conocido sistema operativo Linux.

La principal característica de Git frente a otras opciones del mercado como Subversion o CVS, es que es un sistema distribuido. Esto quiere decir que toda la historia de versiones del código fuente, no reside en un servidor centralizado, sino que lo tiene cada uno de los desarrolladores que utiliza dicho repositorio.

Instalando Git

Si estas leyendo esta guía desde un Mac estas de enhorabuena, porque MacOS viene con Git instalado por defecto. Podéis comprobarlo en vuestro equipo con el comando:

```
git --version
```

Si te encuentras en un sistema operativo **Windows**, instalar **Git** es muy sencillo y puedes hacerlo desde la siguiente dirección:

<https://git-for-windows.github.io>

Por último, si tu opción es **Linux**, puedes instalar **Git** dependiendo de tu distribución del sistema operativo, desde el instalador de paquetes que tengas: apt-get, dnf, yum, ...

`apt-get install git-all`

`dnf install git-all`

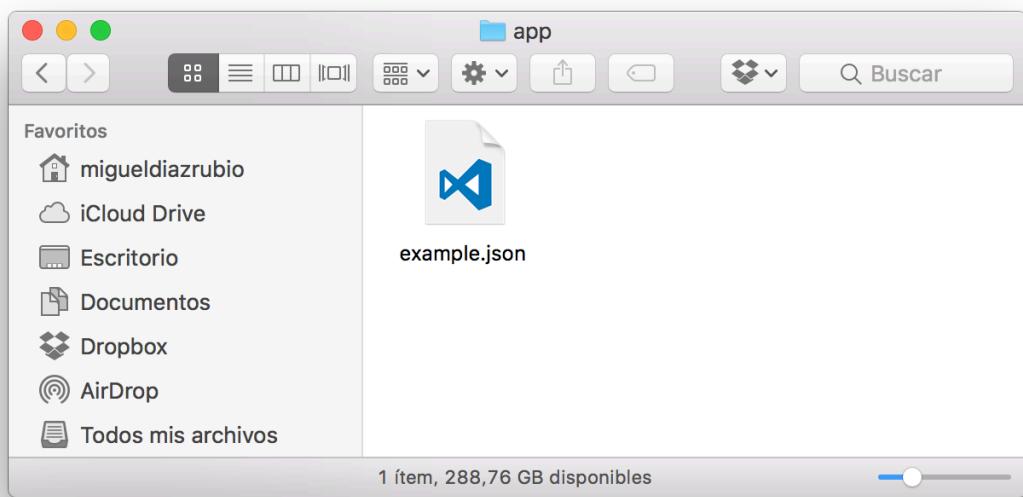
`yum install git`

En mi caso, uso un **Mac**, por lo que todas las capturas que verás dentro de esta guía se corresponden con un sistema **macOS 10.12.6 Sierra** y la versión **2.11.0** de **Git**.

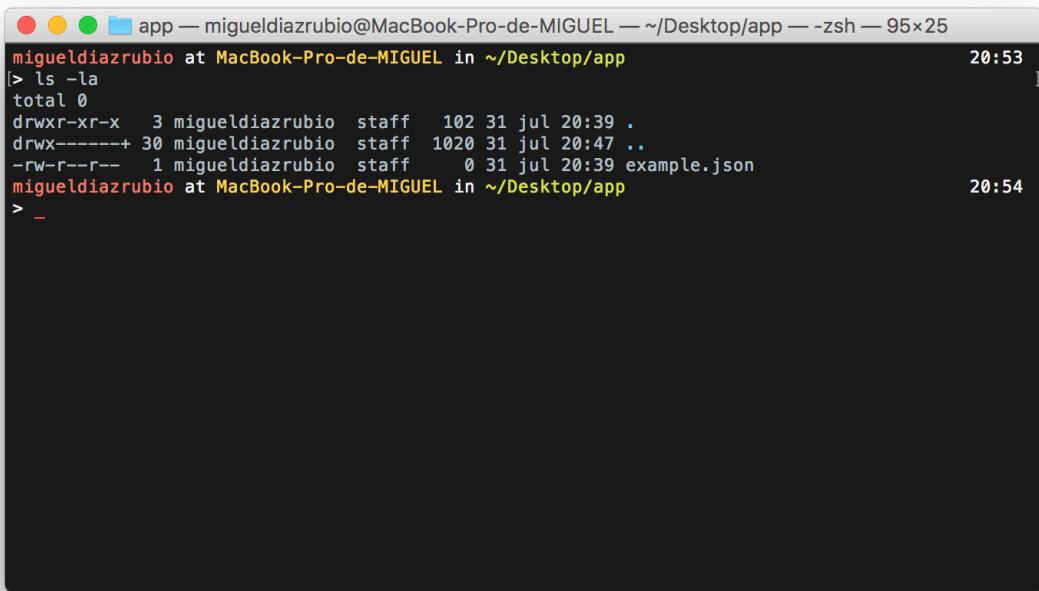
3. Repositorios

En el apartado anterior hemos tocado levemente un concepto muy importante en **Git**, el **repositorio**. Un repositorio es el elemento base en **Git**, y representa la biblioteca donde se almacenan todas las versiones de los archivos de una aplicación o proyecto. Una app, una página web, serían ejemplos de grupos de archivos que podrían ubicarse dentro de un repositorio. En el caso de las aplicaciones de gran volumen, normalmente éstas se encuentran distribuidas en múltiples repositorios. Un ejemplo es el código fuente de Linux por ejemplo.

Pues bien, vas a aprender tu primer comando **Git**, y es el necesario para crear un repositorio. Para ello, tengo una carpeta en mi disco local llamada **app**, que contiene un fichero llamado **example.json**. Crea tú también esta simple estructura antes de continuar.



Si te vas a una ventana de Terminal y haces un listado del directorio mostrando ficheros ocultos veras lo siguiente:



```
miguel diazrubio at MacBook-Pro-de-MIGUEL in ~/Desktop/app — zsh — 95x25
[> ls -la
total 0
drwxr-xr-x  3 miguel diazrubio  staff   102 31 jul 20:39 .
drwx-----+ 30 miguel diazrubio  staff  1020 31 jul 20:47 ..
-rw-r--r--  1 miguel diazrubio  staff     0 31 jul 20:39 example.json
miguel diazrubio at MacBook-Pro-de-MIGUEL in ~/Desktop/app
20:53 ]
```

> _

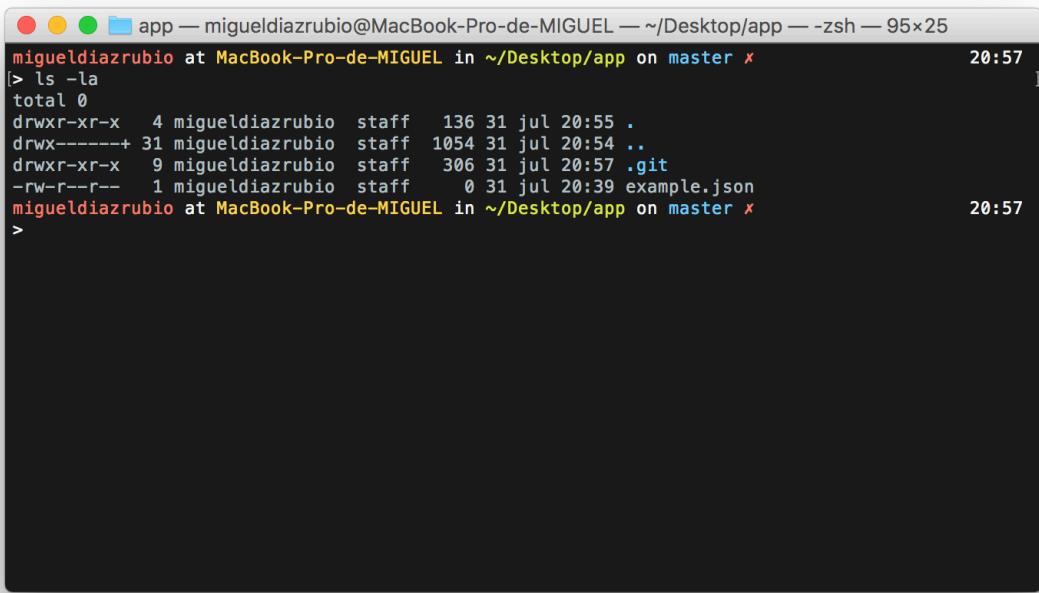
Para crear un repositorio para gestionar los archivos que contiene este directorio app, únicamente tendrás que utilizar un sencillo comando:

`git init`

Esto hará que se muestre en consola un mensaje similar al siguiente:

Initialized empty Git repository in /Users/miguel diazrubio/Desktop/app/.git/

Si ahora haces de nuevo un listado como anteriormente con `ls -la`, verás que se ha creado automáticamente una carpeta oculta `.git` dentro de tu directorio `app`.



```
miguel diazrubio at MacBook-Pro-de-MIGUEL in ~/Desktop/app on master x 20:57
[> ls -la
total 0
drwxr-xr-x  4 miguel diazrubio  staff   136 31 jul 20:55 .
drwxr-xr-x+ 31 miguel diazrubio  staff  1054 31 jul 20:54 ..
drwxr-xr-x  9 miguel diazrubio  staff   306 31 jul 20:57 .git
-rw-r--r--  1 miguel diazrubio  staff      0 31 jul 20:39 example.json
miguel diazrubio at MacBook-Pro-de-MIGUEL in ~/Desktop/app on master x 20:57
>
```

Dicha carpeta, es la encargada de almacenar toda la información relativa a nuestro repositorio, y **debes tratarla como una caja negra** donde no es necesario entrar en ningún caso (hay alguna excepción que veremos en la sección “*12. Trucos y consejos*”.

¡Ya tienes tu primer repositorio creado! ¡Que fácil!, ¿no?

Repositorios remotos

En caso de que no estés comenzando un nuevo proyecto, sino que quieras colaborar en un proyecto que ya tiene un repositorio **Git**, normalmente alguien te facilitará la URL de dicho repositorio, que suele ser del tipo:

```
https://github.com/migueliazrubio/Workapp.git
```

Son fáciles de distinguir, porque la extensión final de la URL siempre es **.git**.

Pues si quieres trabajar sobre dicho repositorio, tienes que hacer una copia de dicho repositorio en tu máquina, o lo que técnicamente se denomina **clonar el repositorio**, que se realiza con el siguiente comando:

```
git clone https://github.com/migueliazrubio/Workapp.git
```

Si se trata de un **servidor que requiere credenciales** para consultar el código, o lo que se denomina un repositorio privado, tienes que modificar ligeramente dicho comando:

```
git clone https://TU_USUARIO@github.com/migueliazrubio/  
Workapp.git
```

Esto hará que **Git** te solicite la contraseña de dicho usuario antes de proceder con la clonación del repositorio.

4. Áreas en Git

Trabajar con **Git** requiere que entiendas primero el **flujo de trabajo** en el que se basa, y en cómo los cambios que realizas en el código van a ir pasando por una serie de áreas antes de llegar a estar guardados dentro de la historia de tu repositorio, como si de una **cadena de montaje industrial** se tratase.

Cada una de esas áreas tiene una función específica, y no hay pasos en falso, ya que todo está estructurado para **trabajar de forma organizada y segura**.

En **Git** existen principalmente tres áreas por las cuales van a ir pasando tus cambios:

- **Área de trabajo o *working directory*:** es la primera etapa en el proceso y, básicamente, se refiere al propio directorio físico donde se encuentra ubicado nuestro código en el sistema operativo, y que será donde hagamos los cambios.

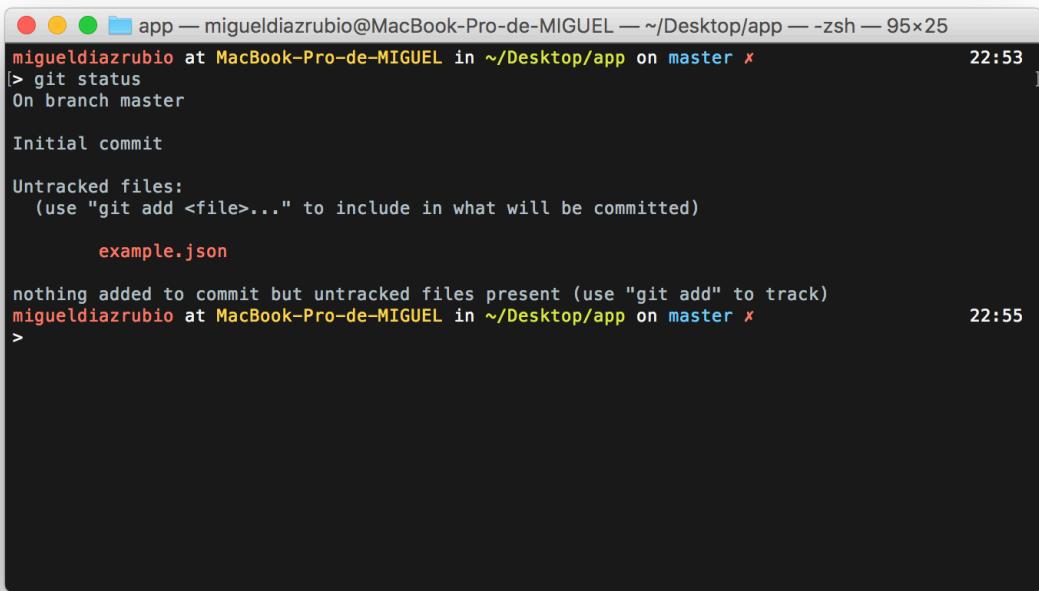
- **Área de preparación o *staging area***: según vas modificando archivos que van a formar parte de una nueva versión de tu proyecto, éstos se van a ir trasladando manualmente a este área. El objetivo de la misma es ir almacenando todos aquellos archivos y modificaciones que van a conformar un cambio en el proyecto.
- **Repositorio git**: cuando ya tenemos en el área anterior todos los cambios que quieras que formen parte de esta nueva versión, puedes confirmar todos esos cambios mediante el uso del comando **commit**. Con ello, dichos cambios pasan a formar parte de ésta tercera y última área, confirmándose como parte de la historia de tu repositorio.

Este proceso de paso por las diferentes áreas, es un ciclo completamente iterativo, que se va produciendo una y otra vez durante la vida del desarrollo de tu proyecto.

Git te ofrece un comando para saber en todo momento, qué ficheros o modificaciones se encuentran en cada uno de éstas áreas. Basta con ejecutar el comando **status**:

`git status`

En tu caso, verás lo siguiente como resultado del comando:



```
miguel diazrubio at MacBook-Pro-de-MIGUEL in ~/Desktop/app on master x 22:53
[> git status
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

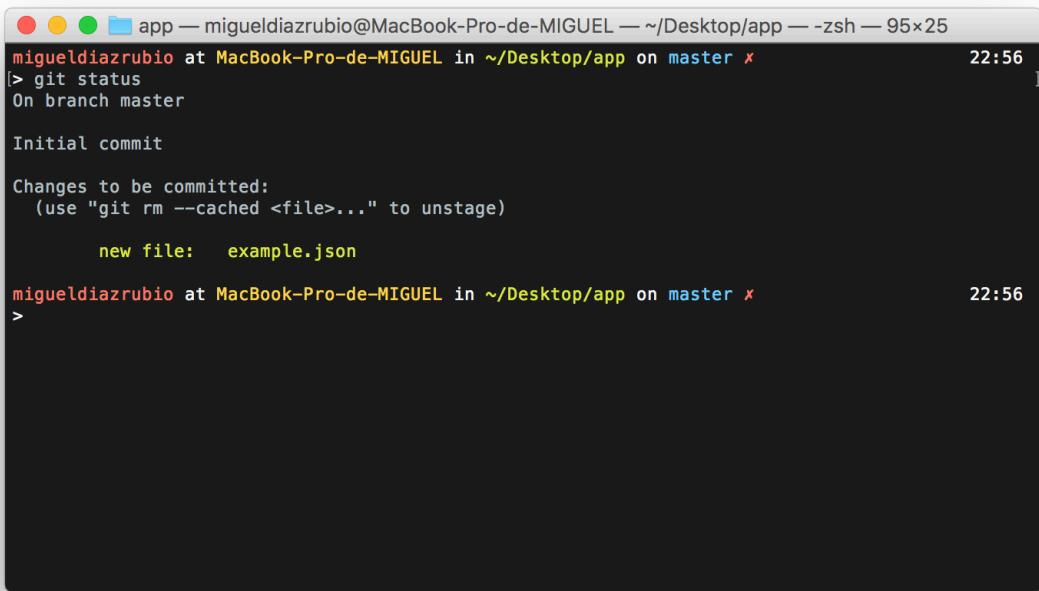
    example.json

nothing added to commit but untracked files present (use "git add" to track)
miguel diazrubio at MacBook-Pro-de-MIGUEL in ~/Desktop/app on master x 22:55
>
```

Los cambios realizados en el fichero `example.json` aún no pueden ser parte de un `commit`, porque se encuentran en el **área de trabajo o *working directory*** marcados como *untracked files*. Para poder trasladarlos al **área de preparación o *staging area*** tienes que utilizar el comando `add`:

```
git add example.json
```

Una vez hecho esto, puedes ejecutar de nuevo el comando `git status` y obtendrás el siguiente resultado:



```
app — miguel.diazrubi@MacBook-Pro-de-MIGUEL — ~/Desktop/app — -zsh — 95x25
miguel.diazrubi at MacBook-Pro-de-MIGUEL in ~/Desktop/app on master x 22:56
[> git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

    new file:   example.json

miguel.diazrubi at MacBook-Pro-de-MIGUEL in ~/Desktop/app on master x 22:56
>
```

Ahora tu cambio aparece en una nueva sección *changes to be committed* y está listo para que hagamos un **commit**. En el siguiente capítulo verás en detalle en qué consiste este comando.

El comando `git add` no requiere que vayas añadiendo los ficheros y directorios de nuestro proyecto de uno en uno, también puedes utilizar lo siguiente para añadir todo los archivos del directorio donde estás ejecutando el comando:

`git add .`

Adicionalmente, puedes establecer el añadido de archivos según un patrón de su nombre o extensión:

```
git add *.properties
```

En el capítulo “*6. Historia y consulta de versiones*” verás con detalle cómo puedes deshacer los movimientos de avance que haces en todas estas áreas. Y es que es lo bueno de **Git**, ¡que todo tiene arreglo!

5. Confirmando cambios

En el capítulo anterior has dejado un cambio dentro del área de preparación, listo para ser confirmado y trasladado al repositorio.

Cada vez que hayas finalizado con la creación y/o modificación de archivos que conforman un cambio en el proyecto, puedes decirle a **Git** que quieres guardar una foto del proyecto en ese instante, o lo que comúnmente se denomina, un **commit**.

Recuerda que no tienes porque hacer un **commit** por cada cambio que hagas, sino que puedes esperar a tener todo un paquete de cambios en el **área de preparación** antes de hacer el **commit** que los engloba.

Para empezar, vas a hacer un cambio en tu código y vas a hacer tu primer **commit**. Para ello vas a editar el fichero `example.json` para introducir un sencillo texto *“Mi primer commit”*.



The screenshot shows a terminal window with a dark background and light-colored text. The title bar at the top reads "app — vi example.json — vi — vi example.json — 95x25". The main area of the terminal contains the following text:

```
Mi primer commit
~
```

There are approximately 20 blank lines above the text "Mi primer commit".

Ahora que ya tienes un cambio realizado sobre la versión original que creaste con el repositorio, estás preparado para hacer un **commit**.

¿Qué es un commit?

Un **commit** está compuesto por la siguiente información:

- **Identificador único del commit:** es un identificador único de tipo SHA1 que nos permite identificar unívocamente un *commit* realizado. Tiene un formato alfanumérico del tipo:

aa1b2fb696a831c89c53f787e03d863691d2b671

- **Autor:** y es que en cada *commit* que se realiza se almacena siempre el usuario que lo ha realizado. Se almacena tanto el nombre como el email del autor:

Miguel Díaz Rubio <info@migueliazrubio.com>

- **Descripción:** donde el desarrollador que hace el *commit* puede indicar un breve texto descriptivo indicando en qué consiste el *commit* realizado. Es común utilizar este texto para indicar el bug que corrige un *commit*, o una breve descripción de la funcionalidad o historia de usuario que implementa.
- **Fecha:** donde se almacena la fecha y hora completa en que ha sido realizado el *commit*.

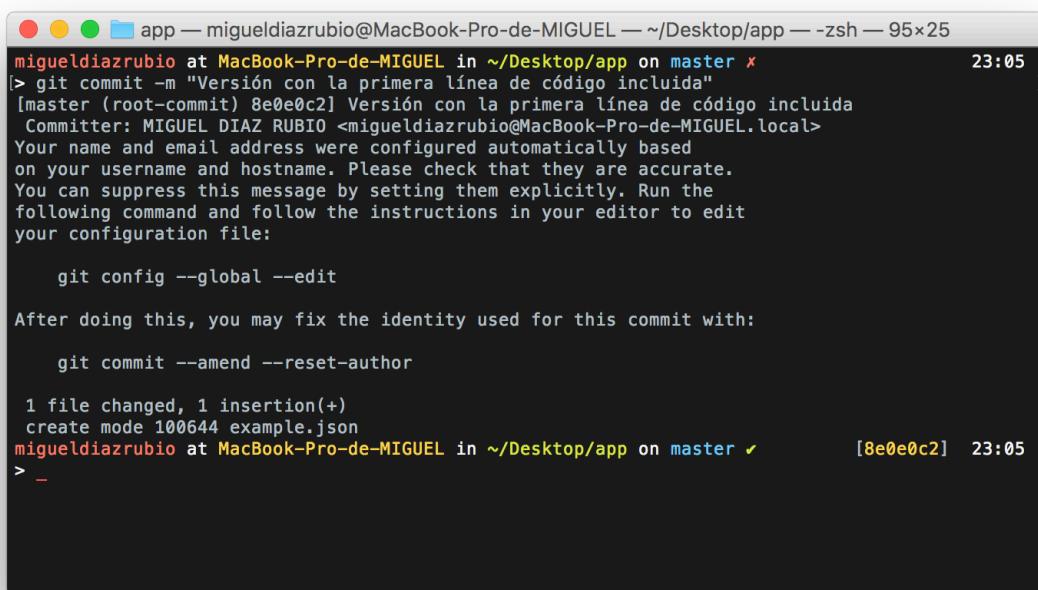
Es curioso conocer el funcionamiento que permite garantizar que el identificador del **commit** sea único en el mundo y en todos los repositorios locales o remotos que existen. Esto se consigue mediante un mecanismo de encriptación de tipo **SHA1**, que utiliza como datos de entrada, múltiples atributos del **commit** como son el autor y el mensaje indicado, entre otros.

Tu primer commit

Para crear un **commit** con el cambio que has realizado en el fichero `example.json`, debes utilizar el siguiente comando.

```
git commit -m "Versión con la primera línea de código incluida"
```

Esto te dará como resultado la creación del **commit**:



The screenshot shows a terminal window on a Mac OS X desktop. The title bar says "app — miguel diazrubio@MacBook-Pro-de-MIGUEL — ~/Desktop/app — -zsh — 95x25". The command entered is "git commit -m "Versión con la primera línea de código incluida"". The output shows the commit message, author information (MIGUEL DIAZ RUBIO <miguel diazrubio@MacBook-Pro-de-MIGUEL.local>), and a note about configuration. It also shows the command to edit global config and amend the commit. Finally, it shows the commit details: 1 file changed, 1 insertion(+), create mode 100644 example.json, and the commit hash [8e0e0c2] at 23:05.

```
miguel diazrubio at MacBook-Pro-de-MIGUEL in ~/Desktop/app on master x 23:05 ]> git commit -m "Versión con la primera línea de código incluida"
[master (root-commit) 8e0e0c2] Versión con la primera línea de código incluida
  Committer: MIGUEL DIAZ RUBIO <miguel diazrubio@MacBook-Pro-de-MIGUEL.local>
Your name and email address were configured automatically based
on your username and hostname. Please check that they are accurate.
You can suppress this message by setting them explicitly. Run the
following command and follow the instructions in your editor to edit
your configuration file:

  git config --global --edit

After doing this, you may fix the identity used for this commit with:

  git commit --amend --reset-author

1 file changed, 1 insertion(+)
create mode 100644 example.json
miguel diazrubio at MacBook-Pro-de-MIGUEL in ~/Desktop/app on master ✓ [8e0e0c2] 23:05
> _
```

Si te fijas en el texto que aparece, hay un párrafo donde se indica lo siguiente:

Committer: MIGUEL DIAZ RUBIO <miguel diazrubi@MacBook-Pro-de-MIGUEL.local>

Your name and email address were configured automatically based

on your username and hostname. Please check that they are accurate.

You can suppress this message by setting them explicitly. Run the following command and follow the instructions in your editor to edit

your configuration file:

```
git config --global --edit
```

After doing this, you may fix the identity used for this commit with:

```
git commit --amend --reset-author
```

Esto ha ocurrido, porque no tienes configurado en ningún sitio tus datos como autor, y por este motivo, **git ha creado unos por defecto** para poder hacer el **commit** en base a la máquina en la que te encuentras y tu nombre de usuario de **macOS**.

Configurando tus datos de autor

Si quieras **configurar tus propios datos de autor** a tu gusto, puedes hacerlo utilizando el comando **git config** e indicando las propiedades que quieras configurar. En mi caso:

```
git config --global user.name "Miguel Díaz Rubio"
```

```
git config --global user.email info@migueldiazrubio.com
```

Si quieras consultar la información de configuración que tienen ahora estas variables puedes hacerlo con los comandos:

```
git config user.name
```

```
git config user.email
```

También puedes consultar de golpe y porrazo todas las propiedades configuradas en git mediante el comando:

```
git config --list
```

Este comando muestra en mi caso lo siguiente:

```
credential.helper=osxkeychain
```

```
user.name=Miguel Díaz Rubio
```

```
user.email=info@migueldiazrubi.com
```

```
core.repositoryformatversion=0
```

```
core.filemode=true
```

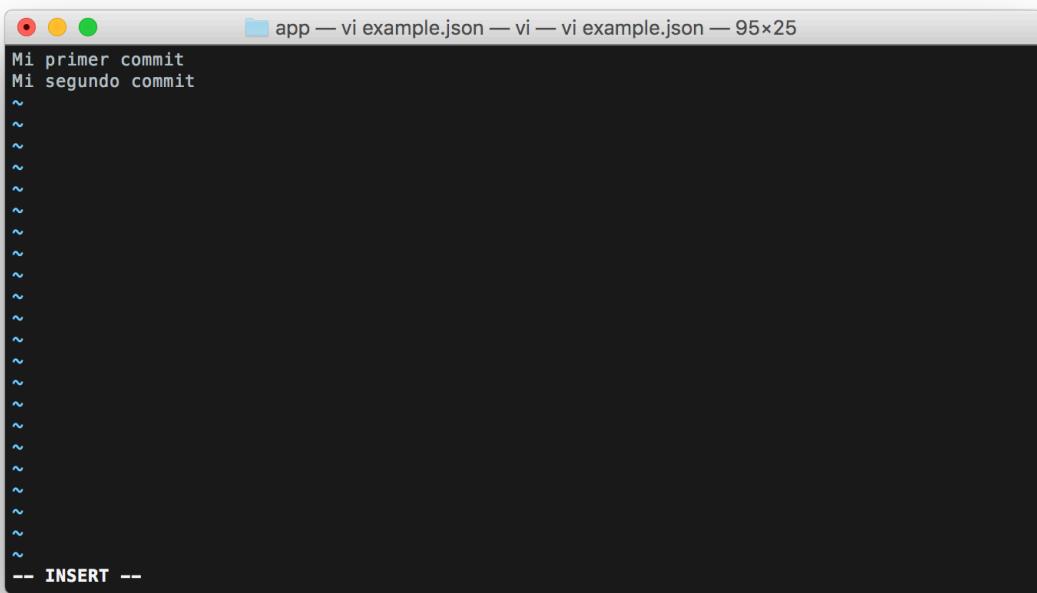
```
core.bare=false
```

```
core.logallrefupdates=true
```

```
core.ignorecase=true
```

```
core.precomposeunicode=true
```

Ahora que ya tienes configurados correctamente los datos de autor, vas a hacer un nuevo **commit**. Incluye en primer lugar dentro del fichero **example.json** una nueva línea:



A screenshot of a terminal window titled "app — vi example.json — vi — vi example.json — 95x25". The window shows the text "Mi primer commit" and "Mi segundo commit" followed by a series of approximately 30 tilde (~) characters. At the bottom of the screen, the text "-- INSERT --" is displayed.

A continuación, vas a añadir el cambio a la zona de preparación, y crearás tu segundo **commit**:

`git add example.json`

`git commit -m "Segunda versión del fichero con dos líneas"`

Si ahora ejecutas un `git status` verás que se muestra el siguiente mensaje:

`On branch master`

`nothing to commit, working tree clean`

Esto quiere decir que tu repositorio, área de preparación y área de trabajo tienen exactamente la misma versión de los ficheros que componen el repositorio.

Corregir un commit recién hecho

En ocasiones es posible que hagas un **commit** e inmediatamente te des cuenta de que has cometido algún tipo de error en el mismo. Quizás faltan un archivo por modificar e incluir en el mismo, o incluso el mensaje que hemos indicado para el **commit** es incorrecto.

Pues bien, estamos a salvo, puedes enmendar el error cometido en el último **commit** mediante el siguiente comando:

```
git commit --amend
```

Si haces un **commit** y ves que te has equivocado, basta con que añadas con **git add** los cambios que faltaran en el área de preparación y hagas un nuevo **commit**, pero con el sufijo **--amend**. Esto hará que los cambios se incluyan en el último **commit** realizado, sin tener que crear uno adicional para corregir el error. Si quieres puedes incluir un **-m XXXXXX** y modificar el mensaje del mismo.

```
git commit -m 'Commit erroneo'
```

```
git add fichero_pendiente
```

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

¿Cuándo hacer commit?

Una pregunta que te puede estar rondando la mente en estos momentos es, ¿cada cuándo debo hacer un **commit** en **Git**?

Pues bien, no hay una norma escrita pero sí puedo darte mi recomendación personal:

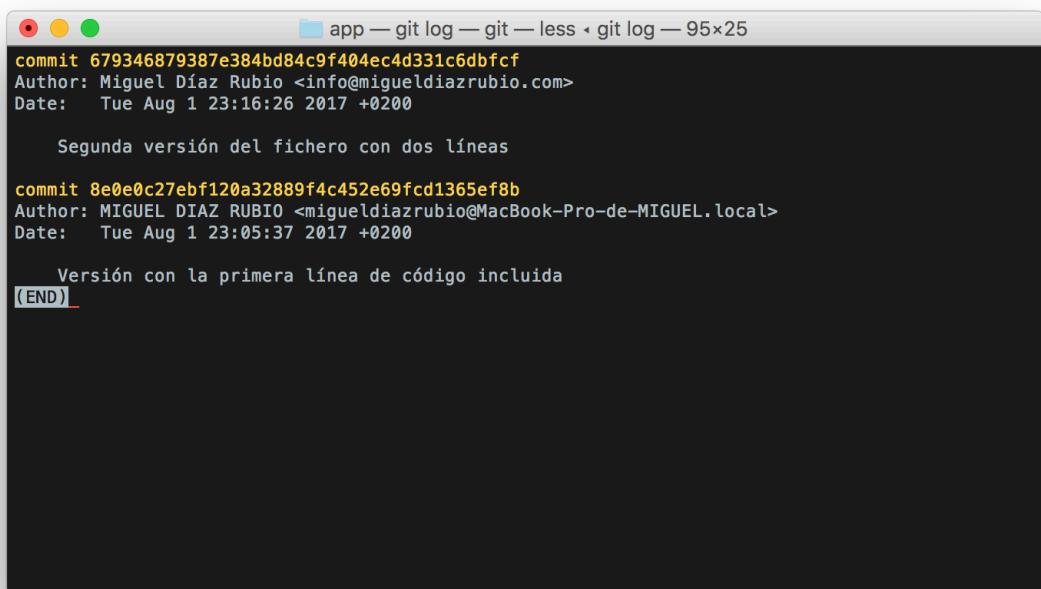
- Si la tarea de modificación del proyecto es una **tarea sencilla** o de poca envergadura, probablemente te baste con hacer un único **commit** cuando estés cómodo y confiado con la finalización de la tarea.
- Si por el contrario la **tarea es compleja** o te va a llevar varios días, intenta hacer un **commit** por cada bloque funcional o técnico que resuelvas de dicha tarea.

Algo que sí debes marcarte como una regla de oro, es que **nunca debes hacer un commit de una versión inestable de tu proyecto**. Todos tus **commit** deberían tener una versión funcional de la aplicación, con mayor

o menor cantidad de funcionalidad, y que si usas test (deberías) todos ellos estén *Passed*.

6. Consulta de versiones

Si en algún momento quieras repasar los **commits** que has realizado, o si por ejemplo te han pasado un repositorio y quieras entender exactamente cuál ha sido el ciclo de vida de **commits** realizados puedes hacerlo utilizando el comando **git log**:



```
app — git log — git — less • git log — 95x25
commit 679346879387e384bd84c9f404ec4d331c6dbfcf
Author: Miguel Díaz Rubio <info@migueliazrubio.com>
Date:   Tue Aug 1 23:16:26 2017 +0200

    Segunda versión del fichero con dos líneas

commit 8e0e0c27ebf120a32889f4c452e69fd1365ef8b
Author: MIGUEL DIAZ RUBIO <migueliazrubio@MacBook-Pro-de-MIGUEL.local>
Date:   Tue Aug 1 23:05:37 2017 +0200

    Versión con la primera línea de código incluida
(END)
```

Eso hará que se muestren todos los **commit** que se han hecho al repositorio, ordenados de más reciente a menos reciente (orden cronológico inverso), y para cada uno de ellos se muestra el identificador, el autor, la fecha y la descripción utilizada en el **commit**.

Por si te has quedado bloqueado en la pantalla de **log**, para salir de la misma basta con que pulses la tecla **q**.

El comando **git log** es enormemente potente y cuenta con muchos parámetros que te van a permitir adecuarlo a obtener la información exacta que necesitas.

Por ejemplo, si quieres que se muestren los ficheros modificados en cada **commit** puedes hacerlo con la opción **-p**:

```
git log -p
```

```
commit 679346879387e384bd84c9f404ec4d331c6dbfcf
```

```
Author: Miguel Díaz Rubio <info@migueliazrubio.com>
```

```
Date: Tue Aug 1 23:16:26 2017 +0200
```

```
Segunda versión del fichero con dos líneas
```

```
diff --git a/example.json b/example.json
```

```
index 46b1cdf..4e9152d 100644
```

```
--- a/example.json
```

```
+++ b/example.json
```

```
@@ -1 +1,2 @@
```

```
Mi primer commit
```

```
+Mi segundo commit
```

```
commit 8e0e0c27ebf120a32889f4c452e69fcd1365ef8b
```

```
Author: MIGUEL DIAZ RUBIO <migueldiazrubio@MacBook-Pro-de-MIGUEL.local>
```

```
Date: Tue Aug 1 23:05:37 2017 +0200
```

```
Versión con la primera línea de código incluida
```

```
diff --git a/example.json b/example.json
```

new file mode 100644

index 0000000..46b1cdf

--- /dev/null

Probablemente te resulte un poco críptica la lectura de los cambios en una consola de este tipo, y este es sin duda uno de los motivos que suelen llevar a los desarrolladores a utilizar entornos gráficos. En tu caso recuerda que aprenderás todos los conceptos y comandos necesarios desde terminal.

Esto puede parecer sufrir por sufrir, pero nada de eso. Lo que te va a permitir es conocer los comandos reales por si a lo largo de tu vida profesional te encuentras con situaciones donde no hay un entorno gráfico de **git**, o el que hay no lo conoces.

Puedes consultar todas las opciones del comando git log en

la siguiente dirección

Cargando versiones antiguas

Es posible que durante el ciclo del desarrollo te encuentres ante situaciones donde quieras, o bien consultar cuál era el estado del código

en un momento temporal anterior, o bien descartar ciertos cambios que por error has confirmado en el repositorio mediante un **commit**.

En primer lugar, `git checkout` te permite ante situaciones donde hayas modificado cosas en el **área de trabajo**, pero aún no las hayas pasado al **área de preparación** (con `git add`), volver a la versión exacta del repositorio de dicho archivo, es decir, deshacer los cambios realizados. Para ello, simplemente tienes que hacer lo siguiente:

```
git checkout NOMBRE_DEL_ARCHIVO
```

Si quieres hacer esto mismo pero para todos los archivos, utilizarás:

```
git checkout -- .
```

Este comando únicamente deshace los cambios en el área de trabajo, ya que todos los cambios que hayan sido incluidos en el área de preparación no serán descartados.

Si quieres descartar todos los cambios de ambas áreas debes utilizar `git reset --hard`. Adicionalmente, si no sólo has hecho modificaciones a ficheros existentes, sino que además has añadido nuevos archivos o directorios, debes también ejecutar el siguiente comando:

```
git clean -df
```

Por otro lado, otro uso clásico del comando `git checkout` es para recuperar el código que había en un determinado **commit**. Para ello, basta con utilizar de nuevo el comando `git checkout` indicando el identificador del **commit** como parámetro, o simplemente los primeros caracteres del mismo (mínimo 4 caracteres, recomendado 6).

Vamos a probar a recuperar la versión de nuestro proyecto en el primer **commit** que hicimos, que en mi caso según `git log` tiene como identificador `8e0e0c27ebf120a32889f4c452e69fcd1365ef8b`:

```
git checkout 8e0e0c
```

Una vez ejecutado, veremos el siguiente mensaje, que se corresponde con el identificador del **commit** que has solicitado cargar, y el mensaje que utilizaste en su momento para realizarlo:

```
HEAD is now at 8e0e0c2... Versión con la primera línea de código incluida
```

Si ahora abres el fichero `example.json` verás que ahora sólo tiene una línea con el texto “*Mi primer commit*”.

Además, si ejecutas `git log` verás que únicamente se muestra el primer **commit**. Sin embargo, no has perdido los cambios, y puedes volver fácilmente al punto en que te encontrabas con el siguiente comando:

```
git checkout master
```

Quizás te estés preguntando, ¿qué es eso de **HEAD** que veíamos al hacer el *checkout* de un **commit** o esto de **master**?

Pues bien, en **Git** hay algo muy interesante y que permite hacer maravillas, que son las ramas o *branches*, y que será algo que verás más adelante en esta guía. En **Git** siempre tienes al menos una rama, y la que nos crea un repositorio por defecto se denomina **master**. Por otro lado, **HEAD** tienes que entenderlo como una especie de puntero, que siempre apunta a la rama en la que te encuentras en cada momento. Cuando has hecho el `git checkout 8e0e0c` tu **HEAD** se ha quedado apuntando al **commit** `8e0e0c27ebf120a32889f4c452e69fc1365ef8b`, y cuando has hecho el `git checkout master` lo que has hecho es volver a apuntar el **HEAD** a la rama **master**, que evidentemente continuaba teniendo los dos **commits** que habías hecho hasta ese momento.

Llegados a este punto, te recomiendo que añadas nuevos archivos, hagas cambios sobre los existentes y juegues un poco con las diferentes áreas y con las posibilidades que nos ofrece el comando `git checkout`.

Si ya estás preparado para continuar, ¡vamos adelante!

7. Trabajo en equipo

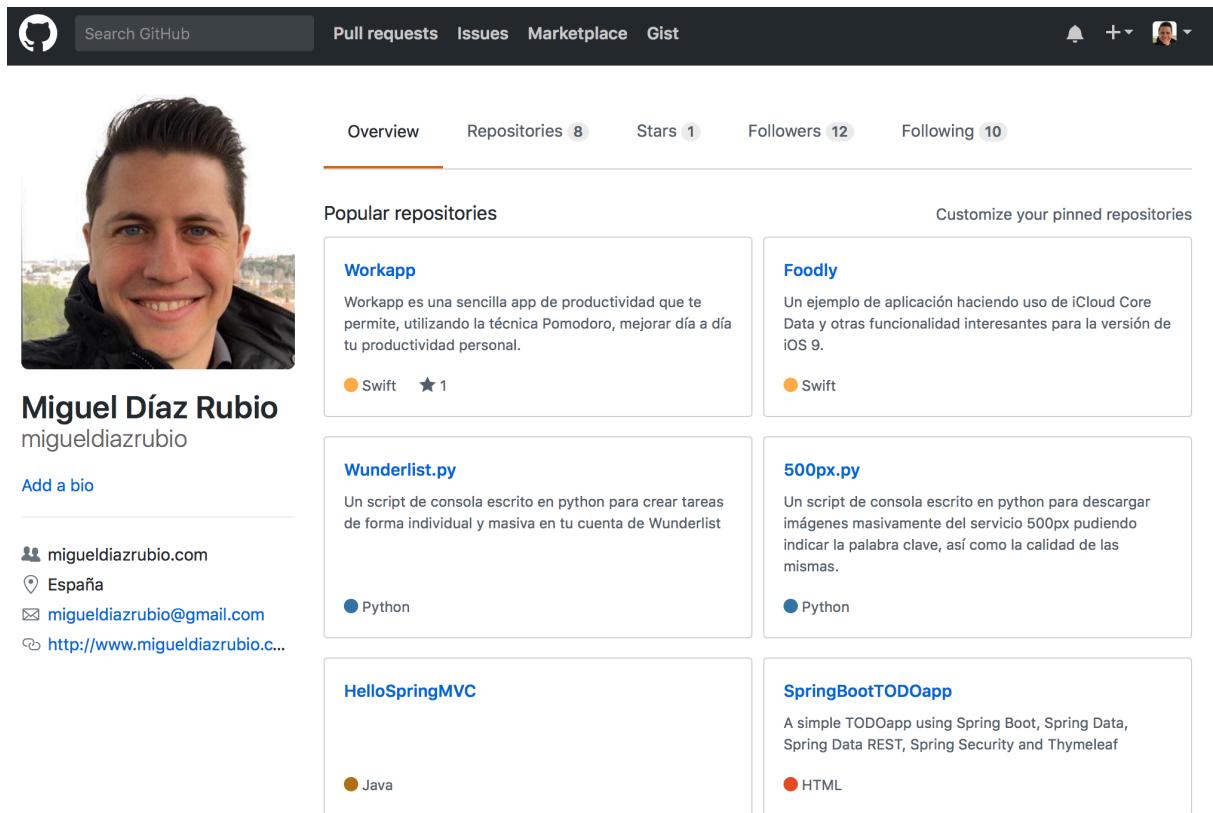
Llegados a este punto, ya tienes un conocimiento suficiente como para poder trabajar de forma individual con lo básico de **Git**. Sin embargo, te queda mucho camino por recorrer, pero vamos paso a paso.

El siguiente paso en el aprendizaje es sin duda el **trabajo en equipo**. Es decir, aprender a trabajar con un mismo repositorio múltiples personas de forma concurrente. Este sin duda es el escenario más común que te encontrarás si colaboras con algún proyecto **Open Source**, o bien si vas a trabajar en un proyecto en tu empresa.

En este escenario existe un repositorio remoto, denominado *remote*, que es clonado de forma local por cada uno de los desarrolladores. Cada cierto tiempo harán **commit** sobre su repositorio local, y cuando lo deseen, publicarán todos esos **commit** en el repositorio remoto, para que el resto de desarrolladores puedan descargarse dichas modificaciones y, que en definitiva, todos puedan trabajar de forma colaborativa en una única versión del proyecto.

Para simular este escenario y que puedas experimentar el que denomino *happy flow* del proceso (es decir, cuando todo sale bien), pero también puedes comprobar qué pasa cuando se producen conflictos, vas a publicar tu repositorio actual en la plataforma [Github](#), para que así puedas hacer cambios desde tu terminal y también desde [Github](#) como si de otro desarrollador se tratase. Quizás suene complejo, pero vas a ver que es muy fácil, y vas a poder interiorizar de forma rápida todos los conceptos necesarios.

Antes de nada, si no tienes una cuenta en [Github](#), por favor, créala en la página web de [github.com](#). Es gratis si tus repositorios son públicos. Mi cuenta es <https://github.com/migueldiazrubio> por si quieres pasarte a echar un vistazo a los proyectos que tengo publicados.

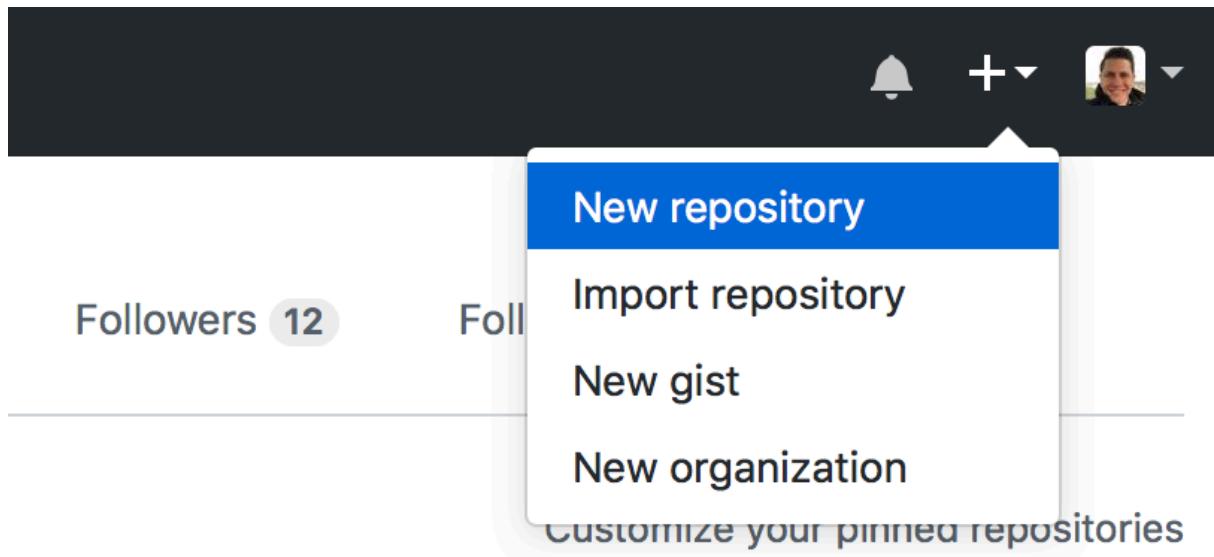


The screenshot shows a GitHub profile page for a user named Miguel Díaz Rubio. At the top, there's a navigation bar with links for Pull requests, Issues, Marketplace, and Gist. Below the navigation bar, there's a header with a profile picture, the user's name, and some statistics: Repositories (8), Stars (1), Followers (12), and Following (10). A red horizontal bar highlights the 'Overview' tab. Below this, there's a section titled 'Popular repositories' containing five repository cards:

- Workapp**: A Swift app for productivity using the Pomodoro technique. It has 1 star.
- Foodly**: An example application using iCloud Core Data and other interesting features for iOS 9. It's written in Swift.
- Wunderlist.py**: A Python script for creating tasks on Wunderlist. It's written in Python.
- 500px.py**: A Python script for downloading images from the 500px service. It's also written in Python.
- HelloSpringMVC**: A Java Spring MVC application. It's written in Java.
- SpringBootTODOapp**: A simple TODO application using Spring Boot, Spring Data, Spring Data REST, Spring Security, and Thymeleaf. It's written in HTML.

On the right side of the profile page, there's a link to 'Customize your pinned repositories'. On the left, there's a sidebar with the user's name, a 'Add a bio' button, and a list of social links: miguel Diaz Rubio's website, location (España), email (miguel Diaz Rubio@gmail.com), and a link to his website (<http://www.miguel Diaz Rubio.com>).

Una vez registrado, vas a crear un repositorio con el icono + de la parte superior derecha de la pantalla.



En mi caso, he creado el repositorio con el nombre **mysampleapp**.

Create a new repository

A repository contains all the files for your project, including the revision history.

Owner / Repository name 

Great repository names are short and memorable. Need inspiration? How about [turbo-octo-winner](#).

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're importing an existing repository.

Add .gitignore: **None** | Add a license: **None** | 

Create repository

*Es importante que NO marques la opción **Initialize this repository with a README** para evitar conflictos cuando subas a Github tu repositorio actual. Ya verás más adelante a qué me refiero con eso de conflictos.*

Una vez creado el repositorio dispones de una URL para el mismo, que puedes utilizar para publicar tu código. En mi caso en: <https://github.com/migueldiazrubio/mysampleapp.git>.

Ahora, vas a indicarle a tu repositorio local, que quieres que se publique en remoto dentro de tu repositorio de **Github** recién creado. Para ello utilizarás el comando `git remote add origin`:

```
git remote add origin https://github.com/migueliazrubio/  
mysampleapp.git
```

Una vez que tienes asociado tu repositorio local y remoto, vas a subir la versión del repositorio que tienes en local al servidor de **Github**. Para ello, utilizarás el comando `git push`:

```
git push origin master
```

Como ves, estamos indicando en el comando que quieres hacer un **push** con todos tus cambios al **origin** que acabas de configurar en el comando anterior, y refiriéndote a la rama **master**.

Lo primero que te solicita **Github** son las credenciales para publicar en dicho repositorio:

```
Username for 'https://github.com':
```

```
Password for 'https://migueliazrubio@github.com':
```

Si introduces tus credenciales de **Github**, y esperas un poco verás que se muestra lo siguiente en pantalla:

Counting objects: 6, done.

Delta compression using up to 4 threads.

Compressing objects: 100% (2/2), done.

Writing objects: 100% (6/6), 583 bytes | 0 bytes/s, done.

Total 6 (delta 0), reused 0 (delta 0)

To <https://github.com/migueldiazrubio/mysampleapp.git>

* [new branch] master -> master

Branch master set up to track remote branch master from origin.

Si ahora te diriges de nuevo a **Github**, a la página de tu repositorio verás que ya aparece tu fichero `example.json`.

migueldiazrubio / mysampleapp

Unwatch 1 Star 0 Fork 0

Code Issues 0 Pull requests 0 Projects 0 Wiki Settings Insights

mysampleapp Edit

Add topics

2 commits 1 branch 0 releases 0 contributors

Branch: master New pull request Create new file Upload files Find file Clone or download

Miguel Díaz Rubio Segunda versión del fichero con dos líneas Latest commit 6793468 2 days ago

example.json Segunda versión del fichero con dos líneas 2 days ago

Help people interested in this repository understand your project by adding a README. Add a README

A partir de este momento, cualquier persona puede descargarse tu código, y si está declarado como contribuidor del repositorio, incluso enviarte sus modificaciones.

A continuación, vas a hacer un **commit** desde **Github**, para ver como tendrías que hacer en tu equipo local para recuperar dicha información del repositorio remoto.

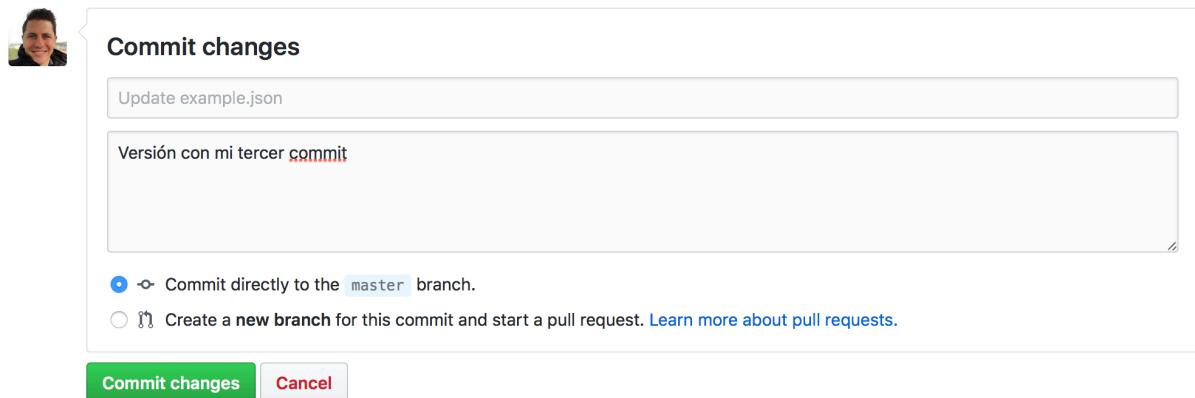
Para ello, pulsa sobre el archivo **example.json** y verás un pequeño icono de un lápiz desde donde poder editar el archivo *online*:

A screenshot of a GitHub file page for 'mysampleapp / example.json'. The page shows a commit history with one entry from 'Miguel Díaz Rubio' made '2 days ago'. Below the commit, there are statistics: '3 lines (2 sloc) | 35 Bytes' and a list of two commits. At the bottom right of the page, there is a prominent 'Edit this file' button with a pencil icon, which is highlighted with a red box.

Ahora al final del todo del fichero añade un nuevo texto "Mi tercer commit".

A screenshot of the GitHub file editor for 'example.json'. The editor shows the file content with three commits: 'Mi primer commit', 'Mi segundo commit', and 'Mi tercer commit'. The 'Mi tercer commit' is the newest addition at the bottom of the list. The editor interface includes buttons for 'Edit file', 'Preview changes', and code styling options like 'Spaces', '2', and 'No wrap'.

Para hacer un **commit** con este cambio, basta con que te dirijas a la parte inferior de la pantalla, donde puedes introducir una descripción del **commit**:



Una vez pulsado el botón de **commit**, y dado que estas tocando directamente en **Github**, ya tienes actualizado tu repositorio remoto con este tercer **commit**.

Ahora, verificarás la historia del repositorio desde **Github**, al igual que hacías anteriormente mediante el comando **git log**. Para ello, tienes un botón **History**.

A screenshot of the GitHub commit history page. At the top, it says '4 lines (3 sloc) | 52 Bytes'. Below that is a table with three rows, each representing a commit:

1	Mi primer commit
2	Mi segundo commit
3	Mi tercer commit

At the bottom right of the table are buttons for 'Raw', 'Blame', and 'History'. There are also icons for copy, edit, and delete.

Y verás claramente que tienes un total de tres **commit** en el repositorio.

History for [mysampleapp](#) / `example.json`

-o-	Commits on Aug 3, 2017
	 Update example.json <small>...</small> migueldiazrubio committed on GitHub 3 hours ago
-o-	Commits on Aug 1, 2017
	 Segunda versión del fichero con dos líneas Miguel Díaz Rubio committed 2 days ago
	 Versión con la primera línea de código incluida MIGUEL DIAZ RUBIO committed 2 days ago

Sin embargo, si haces en local un `git log` verás que únicamente figuran dos.

`commit 679346879387e384bd84c9f404ec4d331c6dbfcf`

Author: Miguel Díaz Rubio <info@migueldiazrubio.com>

Date: Tue Aug 1 23:16:26 2017 +0200

`Segunda versión del fichero con dos líneas`

`commit 8e0e0c27ebf120a32889f4c452e69fcd1365ef8b`

Author: MIGUEL DIAZ RUBIO <migueldiazrubio@MacBook-Pro-de-MIGUEL.local>

Date: Tue Aug 1 23:05:37 2017 +0200

Versión con la primera línea de código incluida

Esto es porque ambos repositorios **local** y **remoto** no están sincronizados. Para poder descargar los cambios de remoto, debes utilizar el comando opuesto al **push**, utilizado recientemente, y que se denomina **pull**.

```
git pull origin master
```

De nuevo estás utilizando los parámetros **origin** para indicar que quieres hacer del **pull** de tu repositorio remoto definido con el comando anterior `git add remote origin`, y a continuación estás indicando **master** para seleccionar la rama deseada.

Al ejecutar el comando verás algo como esto:

```
remote: Counting objects: 3, done.
```

```
remote: Compressing objects: 100% (2/2), done.
```

```
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
```

```
Unpacking objects: 100% (3/3), done.
```

```
From https://github.com/migueliazrubio/mysampleapp
```

```
* branch      master  -> FETCH_HEAD
```

```
6793468..38fb48e master -> origin/master
```

Updating 6793468..38fb48e

Fast-forward

```
example.json | 1 +
```

```
1 file changed, 1 insertion(+)
```

Y si ahora haces un `git log` verás que ya tienes los 3 **commit**, tal y como mostraba en **Github** el repositorio remoto.

```
commit 38fb48e15a271ec1632e675f025c703e6e0f02f7
```

```
Author: Miguel Díaz Rubio <miguel diaz rubio@gmail.com>
```

```
Date: Thu Aug 3 11:26:22 2017 +0200
```

```
Update example.json
```

```
Versión con mi tercer commit
```

```
commit 679346879387e384bd84c9f404ec4d331c6dbfcf
```

```
Author: Miguel Díaz Rubio <info@miguel diaz rubio.com>
```

```
Date: Tue Aug 1 23:16:26 2017 +0200
```

Segunda versión del fichero con dos líneas

1

commit 8e0e0c27ebf120a32889f4c452e69fc1365ef8b

Author: MIGUEL DIAZ RUBIO <migueliazrubi@MacBook-Pro-de-MIGUEL.local>

Date: Tue Aug 1 23:05:37 2017 +0200

Versión con la primera línea de código incluida

Múltiples repositorios remotos

Nada impide que tengas varios repositorios remotos asociados con el comando `git add remote`. Es posible que tu cliente cuente con un repositorio donde quiere que le vayas publicando los cambios, y que además, cuentas con un repositorio en Github o servicios similares para habilitar trabajar en equipo con tus compañeros.

Si en algún momento necesitas consultar los repositorios remotos que hay configurados en tu repositorio, puedes hacerlo con el comando:

`git remote -v`

8. Resolución de conflictos

En el capítulo anterior has visto cómo trabajar con repositorios remotos utilizando los comandos `git push` y `git pull`. Sin embargo, te has enfrentado en todo momento a un escenario muy sencillo, donde no hay conflictos en los cambios.

Cuando dos personas están trabajando en un mismo archivo, puede ocurrir, que se realicen `commit` por parte de ambas personas, y cuando vayas a hacer el `push` o `pull` no pueda ser automática la promoción o descarga de dicho `commit`.

Cuando esto ocurre, tiene que haber una intervención humana, para indicarle a `Git`, qué cambios de cada uno de los dos `commit` quieres conservar y cómo, para a continuación generar un tercer `commit` fruto de la fusión de los cambios de los dos anteriores.

De nuevo vas a simular este escenario, para que puedas entender cómo proceder en esos casos. Para ello, vas a añadir una cuarta línea en el fichero `example.json` mediante un `commit` desde `Github`, y añade también

una cuarta línea distinta mediante un **commit** en local. Esto generará la siguiente situación en tus archivos:

Versión local del archivo `example.json`

Mi primer commit

Mi segundo commit

Mi tercer commit

Cuarta línea desde local

Versión remota del archivo `example.json`

Mi primer commit

Mi segundo commit

Mi tercer commit

Cuarta línea en remoto

Como se puede ver, en ambos casos has modificado el mismo fichero, pero con contenido diferente. A continuación, haz **commit** de ambos cambios desde **Github** y local.

Tienes ante ti un claro conflicto, que tienes que gestionar adecuadamente. Es importante conocer que este tipo de conflictos siempre se resuelven en local y posteriormente se hace **push** al remoto.

En primer lugar, intenta hacer un **pull** de los contenidos del remoto a ver qué ocurre:

```
> git pull origin master
```

```
remote: Counting objects: 3, done.
```

```
remote: Compressing objects: 100% (2/2), done.
```

```
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
```

```
Unpacking objects: 100% (3/3), done.
```

```
From https://github.com/migueliazrubio/mysampleapp
```

```
* branch      master    -> FETCH_HEAD
```

```
38fb48e..0a4d902 master    -> origin/master
```

```
Auto-merging example.json
```

```
CONFLICT (content): Merge conflict in example.json
```

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

Como puedes observar, **Git** ha tratado de hacerlo lo mejor posible, pero en las tres últimas líneas se puede leer que el *auto-merging* no ha sido posible. Además te indica que ha dejado su intento de fusión en el fichero `example.json` en local. Abre el fichero para ver que deberes te ha dejado **Git**:

Mi primer commit

Mi segundo commit

Mi tercer commit

<<<<< HEAD

Cuarta línea desde local

=====

Cuarta línea en remoto

>>>>> 0a4d9027cac5948223617e12049b64f80233c6dd

Básicamente, lo que te está indicando es que en local tenías el texto “Cuarta línea desde local” y en remoto en esa misma línea ponía “Cuarta línea en remoto”. Además, te indica el identificador del **commit** remoto con el que se ha producido el problema. Para esto, **Git** utiliza una sintaxis especial utilizando los símbolos `<<<<<`, `=====`, y `>>>>>`.

Si quisieras averiguar quién ha publicado y para qué el **commit** remoto antes de proceder con la fusión manual de ambos cambios, puedes hacerlo mediante el comando:

```
git show 0a4d9027cac5948223617e12049b64f80233c6dd
```

Bueno, pues en nuestro caso, vamos a imaginar que la fusión consiste en dejar ambas dos líneas introducidas en local, pero modificando la que queda como quinta línea, con el texto “Quinta línea en remoto”. Para ello, aplica en el fichero dicha modificación y aprovecha para eliminar todos las líneas que introduce **Git** para indicarte los cambios. El fichero final debería quedar como el siguiente:

```
Mi primer commit
```

```
Mi segundo commit
```

```
Mi tercer commit
```

```
Cuarta línea desde local
```

```
Quinta línea en remoto
```

Si ahora haces un `git status` verás que se muestra el siguiente texto:

```
Your branch and 'origin/master' have diverged,
```

and have 1 and 1 different commits each, respectively.

Esto es porque **Git** es capaz de ver que en local y remoto, el último **commit** es distinto y por tanto hay una falta de sincronización.

Ahora tienes que generar un nuevo **commit** con la fusión que has hecho en tu archivo.

`git add *`

`git commit -m "Version fusionada de los cambios de la 4 y 5
línea."`

Por último, haz un **push** al remoto para que ambos repositorios queden alineados.

`git push origin master`

Si ahora haces un `git status` verás lo siguiente:

`On branch master`

`Your branch is up-to-date with 'origin/master'.`

`nothing to commit, working tree clean`

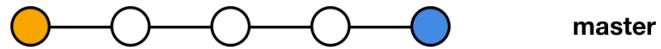
¡Bien! Ya tienes de nuevo tus repositorios perfectamente sincronizados en local y remoto, y has sido capaz de resolver un conflicto manualmente tú mismo.

En este apartado, has gestionado un conflicto sencillo y en un único fichero. Sin embargo en el ciclo de vida del desarrollo en equipos de un proyecto, pueden ocurrir conflictos de mucha mayor complejidad, y que deberás resolver con el mismo mecanismo. Es en estos casos cuando te facilita mucho la vida el uso de un entorno gráfico para la gestión de **Git**.

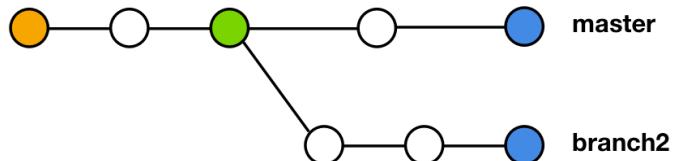
¡Pero aguanta, insensato, que en nada acabamos con los comandos y podrás instalarte el software que deseas y empezar a ver pantallas con colorines!

9. Ramas y fusiones

Según vas trabajando en un proyecto y creando más y más **commit**, vas construyendo un camino temporal, que se denomina **rama**.



Sin embargo, en cualquier punto o **commit**, puedes necesitar generar un camino alternativo o bifurcación, momento a partir del cual tendrás dos ramas para el mismo proyecto.



Normalmente, estas bifurcaciones o también llamados *forks* se producen cuando vas a abordar un gran evolutivo o funcionalidad sobre tu proyecto, y quieres hacerlo en un entorno aislado que no impacte al resto de

desarrolladores, que seguirán trabajando en la rama **master** mientras tanto.

Cuando el desarrollo que estás haciendo en tu rama creada ha finalizado y sabes que funciona correctamente, es el momento en el que debes fusionar tu rama con la rama principal o **master**.

En **Git** es tan sencillo trabajar con ramas, que en muchas ocasiones, no se recurre a ramas sólo para grandes funcionalidades a implementar (como en otros sistemas como SVN), sino para casi cualquier nuevo cambio a realizar. Esto ofrece una libertad absoluta para trabajar de forma independiente en correcciones de nuestro producto, y nuevos desarrollos para el mismo. De hecho, se suele utilizar como nomenclatura para dichas ramas el nombre de la funcionalidad a implementar.

Esto, que a priori puede parecer algo muy complejo, realmente ya lo has estado haciendo parcialmente cuando estabas trabajando en local y remoto con el mismo repositorio. El funcionamiento en este caso va a ser muy similar.

Las ramas, al igual que los **commit**, puedes dejarlas en tu repositorio local, o publicarlas en el remoto para que otros puedan colaborar con sus aportaciones.

Un escenario posible sería tener un proyecto donde tienes la rama **master**, donde están trabajando 2 desarrolladores en corregir incidencias que detectan los clientes en producción. En paralelo, tienes una rama **version-2.0** donde están trabajando otros 2 desarrolladores en nuevas funcionalidades que ha solicitado tu *Product Owner*.

Es importante saber que cuando se genera una rama, los **commit** que se hagan a partir de entonces, ya pertenecerán a dicha rama.

Evidentemente, cuando te enfrentas a proyectos donde se trabaja con ramas, el numero de conflictos que se van a producir puede crecer, y es importante gestionar correctamente qué ramas se crean y para modificar qué funcionalidad de tu proyecto, con el objetivo de evitar al máximo los conflictos que se produzcan. Por ejemplo, intentar que no haya 3 ramas que vayan a modificar, por ejemplo, el proceso de autenticación de tu proyecto, ya que se producirán un número enorme de conflictos. Sería preferible en ese caso, hacer esos tres cambios sobre la misma rama.

De todos modos, hay mucha escritura en internet al respecto de lo que se denomina **Git Workflow**, y puedes aplicar cualquiera de los métodos existentes para gestionar tu código, como hacerlo de manera personalizada para encajar mejor en tu problemática y necesidades.

Al final de este capítulo te daré unas pinceladas de las que podrían ser las buenas prácticas en este sentido.

¡No nos enrollemos más y vamos a crearnos una rama en nuestro repositorio local!

Cuando se trabaja con ramas en **Git**, se utilizan esencialmente tres comandos (uno de ellos ya lo hemos utilizado):

- **git branch**: para crear nuevas ramas de nuestro proyecto
- **git checkout**: para poder actualizar tu área de trabajo con una rama deseada
- **git merge**: que te ayudará a fusionar ramas, cuando el ciclo de vida de alguna de ellas ha finalizado y debe confluir en la rama principal (u otra).

Imagina que quieres crear una nueva rama para la creación de un nuevo archivo en tu repositorio. En primer lugar, vas a generar una rama “version-2.0”:

```
git branch "version-2.0"
```

Una vez creada, puedes utilizar en cualquier momento el comando **git branch** sin parámetros, para verificar las ramas que tiene el repositorio en el que te encuentras.

`git branch`

Si quieres consultar las ramas remotas puedes hacerlo mediante:

`git branch -r`

Y por último, si quieres consultar las ramas tanto locales como remotas:

`git branch -a`

Todos estos comandos muestran la lista de ramas disponibles según lo indicado:

`* master`

`version-2.0`

Además ves que se muestra un asterisco indicando la rama a la que está apuntando ahora mismo tu `HEAD` (¿te acuerdas de que lo vimos al principio?).

Si quieres apuntar tu `HEAD` a la rama recién creada, puedes hacerlo mediante el comando `git checkout`:

`git checkout version-2.0`

Esto te mostrará como resultado el siguiente mensaje:

```
Switched to branch 'version-2.0'
```

Ahora, vuelve a ejecutar `git branch`:

```
master
```

```
* version-2.0
```

El asterisco ahora está en la rama adecuada.

Ahora crea un nuevo archivo `branch.json` en tu proyecto, y crea con él tu primer **commit** en esta nueva rama.

```
git commit -m "Primer commit en la rama version-2.0"
```

¡Enhorabuena! ¡Has creado una rama y creado tu primer **commit** en la misma!

A continuación, y para evitar que puedas perder el trabajo que estás haciendo en esta rama, vas a publicar a **Github** la nueva rama con este **commit**:

```
git push origin version-2.0
```

En este caso, como puedes observar, el último parámetro del comando ya no es **master**, ya que la rama que estás publicando es otra.

Si te diriges a **Github**, podrás comprobar que ahora tienes dos ramas también en el repositorio remoto.

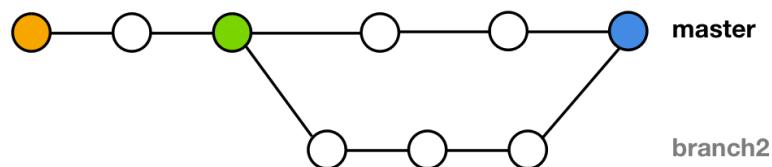
Your recently pushed branches:

version-2.0 (1 minute ago)

Compare & pull request

Fusionando cambios de una rama sobre otra

Imagina que ya has acabado con todos los cambios que querías hacer en tu rama, y puedes llevar los cambios de esta rama **version-2.0**, a la rama **master**.



Para fusionar los cambios o **commit** de una rama sobre otra, utilizarás el comando **git merge**, situándote previamente mediante **git checkout** en la rama destino de la fusión (en tu caso, master).

Por tanto primero cambia a la rama **master**:

```
git checkout master
```

Y a continuación procede con la fusión de la rama **version-2.0**:

```
git merge version-2.0
```

El resultado que verás en consola será algo como lo siguiente:

```
Updating eee6d01..9202eac
```

```
Fast-forward
```

```
branch.json | 0
```

```
1 file changed, 0 insertions(+), 0 deletions(-)
```

```
create mode 100644 branch.json
```

Evidentemente, de nuevo te has enfrentado a un escenario muy simple y sin conflictos, ¡pero qué digo! ¡si tu ya sabéis resolver conflictos!

Si en este momento ejecutas un `git status` verás que te indica que tu rama local tiene un **commit** más que la rama remota, y te sugiere que pubiques los cambios con `git push`.

```
On branch master
```

Your branch is ahead of 'origin/master' by 1 commit.

(use "git push" to publish your local commits)

nothing to commit, working tree clean

Pues bien, vamos a hacerle caso y publicar a **Github** la rama **master** con los cambios ya fusionados de la rama **version-2.0**.

`git push origin master`

Si ahora haces un `git log` podrás comprobar cómo tu rama **master** tiene el **commit** que hiciste en la rama **version-2.0**.

`commit 9202eac83df0774d17ef32dfebbd5c98aa7012f2`

`Author: Miguel Díaz Rubio <info@migueliazrubio.com>`

`Date: Thu Aug 3 16:16:39 2017 +0200`

Primer commit en la rama version-2.0

Conocer la procedencia de los commit de la historia

Como habrás observado, es posible que a lo largo del ciclo de vida de tu proyecto, algunos **commit** de los que conforman la historia del mismo, no hayan sido generados en la rama principal, sino que para ello se hayan generado ramas distintas, que luego han sido fusionadas.

Para conocer con este nivel de detalle la procedencia de cada uno de los **commit** tienes un comando que te va a ayudar mucho:

`git reflog`

El resultado de este comando, es un completo detalle de cada uno de los pasos que se han dado en el repositorio y la procedencia de los mismos:

```
9202eac HEAD@{0}: checkout: moving from version-2.0 to  
master
```

```
9202eac HEAD@{1}: checkout: moving from master to  
version-2.0
```

```
9202eac HEAD@{2}: merge version-2.0: Fast-forward
```

eee6d01 HEAD@{3}: checkout: moving from version-2.0 to master

9202eac HEAD@{4}: checkout: moving from master to version-2.0

eee6d01 HEAD@{5}: checkout: moving from version-2.0 to master

9202eac HEAD@{6}: commit: Primer commit en la rama version-2.0

eee6d01 HEAD@{7}: checkout: moving from master to version-2.0

eee6d01 HEAD@{8}: commit (merge): Version fusionada de los cambios de la 4 y 5 línea.

6f39304 HEAD@{9}: commit: Cuarta línea en local

38fb48e HEAD@{10}: pull origin master: Fast-forward

6793468 HEAD@{11}: checkout: moving from 8e0e0c27ebf120a32889f4c452e69fcd1365ef8b to master

8e0e0c2 HEAD@{12}: checkout: moving from master to 8e0e

6793468 HEAD@{13}: reset: moving to HEAD

6793468 HEAD@{14}: reset: moving to HEAD

6793468 HEAD@{15}: reset: moving to HEAD

6793468 HEAD@{16}: checkout: moving from master to master

6793468 HEAD@{17}: commit: Segunda versión del fichero con
dos líneas

8e0e0c2 HEAD@{18}: commit (initial): Versión con la primera
línea de código incluida

Eliminando ramas de nuestro repositorio

Cuando una rama ha sido fusionada en la rama principal, deja de ser útil para el repositorio, y si lo deseas, puedes borrarla.

Para ello puedes utilizar el comando `git branch -d`:

`git branch -d version-2.0`

En el caso de que la rama contenga cambios que aún no hayan sido fusionados a la rama principal, este comando devolverá un error del tipo:

error: The branch 'version-2.0' is not an ancestor of your current HEAD.

If you are sure you want to delete it, run 'git branch -D version-2.0'.

Si aún así quieras borrarla, tal y como indica el mensaje de error, podemos utilizar el modificador `-D` para borrarla igualmente.

Finalmente, si lo que quieras es borrar una rama remota, puedes hacerlo con el comando `git push` de la siguiente forma:

`git push origin --delete version-2.0`

Buenas prácticas con ramas

Últimamente se ha puesto de moda una forma de organizar las ramas en Git que favorece la distribución sin riesgo de las diferentes tipologías de cambios que se suelen aplicar en el día a día en un proyecto de gran envergadura.

Esta buena práctica propone crear la siguiente estructura de ramas en Git:

- **master:** es la rama de producción de nuestro proyecto, sobre la cual nunca se deben acometer cambios directamente por ningún motivo, y que en todo momento debe ser completamente estable.
- **develop:** es la rama dedicada al desarrollo e integración de nuevas funcionalidades de nuestro proyecto. Cuando una nueva versión del proyecto y sus nuevas funcionalidades sean completamente estables, se fusionarán todos los cambios con la rama *master*.
- **features:** por cada nueva funcionalidad, se creará una rama de tipo *feature*, teniendo como origen la rama *develop*, que una vez estable será fusionada de nuevo con la rama *develop*.
- **hotfix:** por cada corrección o incidencia a resolver en nuestro proyecto, crearemos una rama de tipo *hotfix*, teniendo como origen la rama *master*, que una vez estable será fusionada de nuevo con la rama *master*.

Las ramas de tipo **features** y **hotfix**, normalmente utilizan dichas palabras como prefijo para la nomenclatura de las mismas, por ejemplo, “`hotfix_inc12100_login_error`”.

Como ves, una forma muy organizada de trabajar en Git, y que hará que tengas perfectamente controlada cada nueva línea de código o cambio a introducir en tu proyecto.

10. Etiquetas o *tags*

En **Git**, como en otros muchos sistemas de control de versiones, puedes establecer etiquetas para identificar la situación del proyecto en un momento dado. Normalmente estas etiquetas están relacionadas con los lanzamientos o *releases* del proyecto, es decir, en general para marcar momentos importantes en la vida del proyecto.

Es importante saber que, estas etiquetas no se publican por defecto en repositorios remotos al hacer un `git push`. Si quieres, por el contrario, que sí se publiquen, tienes que solicitarlo:

```
git push --tags
```

Las etiquetas realmente no son modificaciones del repositorio y, por tanto, no es posible utilizar comandos como `git checkout` para cargar en tu área de trabajo el código que había cuando hiciste una determinada etiqueta.

Sin embargo, es posible lograr el mismo efecto con un pequeño truco, y es que puedes utilizar el comando `git checkout` para generar una rama que contenga el código que había en dicha etiqueta:

```
git checkout -b version2 v2.0.0
```

En este caso **version2** es la rama que vas a crear, y **v2.0.0** es la etiqueta de la que partirá.

Como ves la etiqueta tiene una nomenclatura muy peculiar, y es que normalmente se trata de seguir la siguiente norma. La versión del proyecto suele estar compuesto por tres dígitos:

- 1º: para indicar cambios mayores o de gran envergadura
- 2º: para indicar cambios menores
- 3º: para indicar pequeños ajustes

Si haces una corrección de una o varias incidencias, aumentarías el tercer dígito de la versión. Si introduces una o varias nuevas funcionalidades de poco impacto, aumentarías el segundo dígito. Si estás lanzando una versión donde has rediseñado la aplicación o introducido grandes funcionalidades, aumentarías el primer dígito. Esto no es algo obligatorio, pero ayuda al usuario a entender la trascendencia de la versión que estás lanzando.

Genera una etiqueta de tu versión actual de la rama `master` del repositorio para que veas lo sencillo que es:

```
git tag v1.0.0 -m "Primera versión de la aplicación"
```

¡Así de sencillo!

Si en algún momento quieres conocer las diferentes etiquetas que se han generado para la rama de un repositorio, puedes hacerlo con el comando:

```
git tag -l
```

11. Pull request

Normalmente, para poder publicar cambios en un repositorio remoto, alguien tiene que haberte designado como colaborador de dicho repositorio. En **Github** esto es algo muy sencillo de hacer desde la barra superior del proyecto.



El problema viene cuando queremos proponer un cambio al propietario de un repositorio, y no somos colaboradores del mismo. Podríamos solicitarle acceso, pero normalmente un colaborador no es alguien que hace una aportación puntual, sino alguien que colabora activamente en el proyecto en cuestión.

A la gente de **Github** se le ocurrió una fenomenal idea, que consiste en que cualquiera puede proponer al dueño de un repositorio cambios en el mismo, todos ellos empaquetados dentro de una especie de rama que conforma lo que se denomina una **pull request**.

El proceso para hacer una **pull request** es muy sencillo y se puede resumir en los siguientes pasos:

1. En primer lugar, debes tener tu propia copia en **Github** del proyecto. Esto se consigue a través de otra interesante funcionalidad similar a `git clone` que se llama **fork**. Navega a la web del repositorio sobre el que queremos hacer la *pull request* y pulsa en el botón superior derecho **fork**. Esto creará una copia exacta de la situación actual del proyecto, dentro de tu cuenta en **Github**.



2. Haz una copia local de dicho proyecto mediante un `git clone`.
3. Crea una rama para aplicar los cambios que deseas proponer mediante `git branch myfork` y a continuación cambia al **HEAD** a dicha rama con `git checkout myfork`.
4. Aplica todas las modificaciones que quieras realizar sobre dicha rama.
5. Publica la rama en **Github** mediante `git push origin myfork`.
6. Desde la web del proyecto, accede a la sección **branches** y pulsa sobre la opción **New pull request** de la rama correspondiente.



7. Accederás a una pantalla donde podrás escribir una breve descripción con la explicación de la propuesta de modificación que estás realizando en el proyecto. Una vez que hayas cumplimentado toda la información y pulses la opción **Create pull request**, al autor le llegará una notificación en su cuenta de **Github**, desde la cual podrá aceptar o denegar tu propuesta. En caso de que los cambios que has realizado sean integrables de forma automática (sin conflictos), si el autor acepta tu cambio, este será incorporado directamente a la rama **master** del proyecto original.

Este es un proceso que no tiene complejidad alguna y que nos facilita poder colaborar con cualquier proyecto de código abierto en **Github**.

¡No seas vergonzoso y anímate a colaborar en otros proyectos!

12. Trucos y consejos

En este apartado me gustaría contarte algunos **trucos y consejos** que he recopilado sobre el uso de **Git**, que van a hacer que tu productividad se dispare o bien que puedas personalizar a tu gusto la experiencia de uso del sistema.

Gitignore

Este puede que sea el consejo más básico de todos los que siguen, pero no por ello deja de ser interesante. Normalmente, en un proyecto de desarrollo de software, hay ficheros que no tiene sentido que se publiquen en el repositorio, ya que no aportan nada al resto de usuarios. Suelen ser ficheros, por ejemplo, de tipo temporal que genera el entorno de desarrollo integrado (IDE) que estemos utilizando.

Pues bien, tenemos una forma muy sencilla de indicar a Git qué archivos debe obviar en la gestión de su flujo de trabajo.

Para ello, debes crear en el raíz de tu proyecto un fichero con el nombre `.gitignore`. En su interior puedes especificar un patrones para la exclusión de uno o varios archivos. En cada línea del archivo puedes incluir un patrón diferente, y todos ellos se aplicarán automáticamente para que Git deje de tratar los ficheros que cumplan dichos patrones.

`.gitignore`

`**/logs`

`*.data`

`mytest.properties`

Si te fijas, en la primera línea se utilizan dos asteriscos, lo cual quiere indicar que se excluyan todos los ficheros ubicados dentro de un directorio `logs`, independientemente de la ruta donde se encuentre. Es decir, que aplicará al directorio `/logs`, pero también al `/actions/logs`.

En la segunda línea estas excluyendo todos los ficheros con extensión `.data` del raíz de tu proyecto.

En la última línea simplemente estás excluyendo un fichero concreto `mytest.properties`.

Todos estos ficheros y directorios no se mostrarán cuando hagas un `git status` por ejemplo, ni tampoco formarán parte, por tanto, de ningún `commit` que hagas.

Normalmente, según la tecnología o lenguaje con el que estés trabajando, encontrarás fácilmente en internet el fichero `.gitignore` que mejor encaja contigo. Te recomiendo echar un vistazo a [GitIgnore.io](https://www.gitignore.io) en la web <https://www.gitignore.io>, que justamente te permite generar automáticamente el fichero en base a las tecnologías que le indicas.

Alias en Git

En Git tienes la opción de configurar alias para tus comandos, es decir, pequeños atajos para evitar escribir grandes cantidades de texto.

Si quieres que cuando hagas un `git cm` se haga un `git commit` basta con que ejecutes el siguiente comando para configurar el alias a nivel Git:

```
git config --global alias.cm 'git commit'
```

Algunos alias que puedes crear y que pueden serte realmente de utilidad son:

```
git config --global alias.unstage 'reset HEAD --'
```

```
git config --global alias.last 'log -1 HEAD'
```

El primero de ellos te crea un comando `git unstage` que lo que va a hacer es vaciar tu área de preparación, quitando todos aquellos elementos que hayas añadido con `git add`.

El segundo alias te crea un comando `git last` que muestra la información del último **commit** realizado.

Son dos sencillos ejemplos que te permiten mejorar tu productividad usando **Git**.

Para borrar un alias que hayas creado puedes utilizar el modificador `--unset` como en el siguiente ejemplo:

```
git config --global --unset alias.last
```

Cherry Pick

Imagina que estas trabajando en una rama y has hecho muchos **commit** sobre la misma. De repente, te das cuenta de que te vendría muy bien poder trasladar concretamente uno de esos **commit** a la rama **master**, pero no quieres hacer un **merge** aún de ambas ramas. Pues estás de

enhorabuena, tenemos un comando que nos permite hacer este pequeño milagro, `git cherry-pick`.

Su uso es muy sencillo, solo tienes que indicar el identificador del **commit** que deseas migrar de rama y ¡voilá!:

```
git cherry-pick 3cdd67fed7ce2265cb366787455da6fc34c022f5
```

Como te estarás imaginando, este comando podría generar conflictos, que como ya has visto anteriormente, puedes solucionar manualmente en cada uno de los ficheros con conflictos (recuerda buscar los caracteres `<<<<< HEAD`).

Stash

Este es uno de los trucos que más me gustan, y es que resuelve un problema que, al menos a mi, se me produce en muchas ocasiones. Te encuentras trabajando en una rama que has creado para aplicar ciertos cambios a tu proyecto. En mitad del desarrollo de esos cambios, quieres saltar a otra rama para verificar un dato. Sin embargo, cuando ejecutas el `git checkout branch` se muestra un error diciendo que tienes cambios sin **commit** y que los perderás.

Pues bien, en este caso, el comando `git stash` es tu gran amigo, ya que te permite guardar todas esas modificaciones no confirmadas en una especie de pequeño backup:

`git stash`

Hecho esto, puedes cambiar de rama para revisar lo que deseas, y tras volver a la rama en la que te encontrabas trabajando, puedes ejecutar el siguiente comando para recuperar todas tus modificaciones no confirmadas:

`git stash pop`

Hooks

Git trae consigo una potente herramienta que te va a permitir establecer controles o incluir tu propia lógica dentro de ciertos eventos del flujo de trabajo de Git.

Existen a día de hoy ciertos puntos a los cuales te puedes enganchar (*hook*) para aplicar la lógica que deseas. Por defecto, dentro del directorio `.git/hooks` viene un script para cada uno de estos puntos:

- `applypatch-msg.sample`

- pre-push.sample
- commit-msg.sample
- pre-rebase.sample
- post-update.sample
- prepare-commit-msg.sample
- pre-applypatch.sample
- update.sample
- pre-commit.sample

Si quieres aplicar alguno de ellos, basta con eliminar la extensión `.sample` y empezarán a funcionar automáticamente.

Estos scripts están programados normalmente en **shell** y **PERL**, pero puedes utilizar cualquier lenguaje, siempre y cuando pueda ser un ejecutable. En la primera línea de cada script veréis una línea donde, dependiendo del lenguaje de programación utilizado, deberás indicar la ruta al intérprete que corresponda. Por ejemplo, el siguiente corresponde a un script **Python**:

```
#!/usr/bin/env python
```

Con este mecanismo podemos hacer, por ejemplo, que no se pueda hacer un **commit** si el mensaje no cuenta con ciertos elementos clave como el número de incidencia que resuelve.

Tenéis muchos ejemplos disponibles en la web de **Atlassian** dedicada a **Git**, en la siguiente dirección:

<https://www.atlassian.com/git/tutorials/git-hooks>

13. Editores visuales

Como te decía al principio de esta guía, creo firmemente que el aprendizaje de **Git** debe comenzar por los comandos y su uso desde la línea de comandos. Es un pequeño esfuerzo, pero te aseguro que dará su resultado en el futuro.

Ahora que ya cuentas con un montón de comandos en tu arsenal de **Git**, es momento de conocer algunas herramientas que sin duda te pueden ayudar a trabajar más cómodamente con **Git**: son los editores visuales.

Voy a presentarte algunas de las que considero las mejores opciones disponibles en el mercado en estos momentos.

- **Tower** (*PAGO*). Disponible en <https://www.git-tower.com/mac/>



- **SourceTree (GRATIS).** Disponible en <https://www.sourcetreeapp.com>

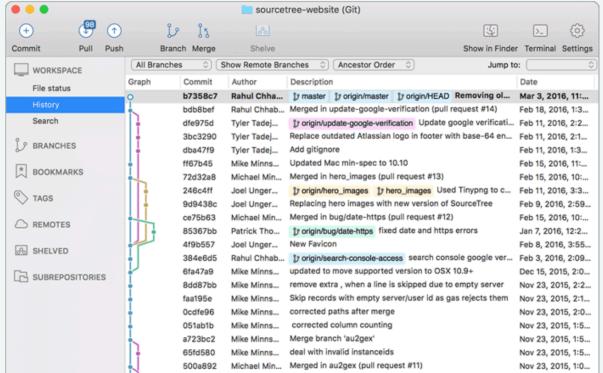
 **SourceTree**

[Download free](#)

Simplicity and power in a beautiful Git GUI

[Download for Mac OS X](#)

Also available for Windows



Date	Author	Description
Mar 3, 2016, 11...	Rahul Chhab...	[r] master [r] origin/master [r] origin/HEAD Removing old...
Feb 11, 2016, 13...	brianm...	[r] master [r] origin/master [r] origin/HEAD ...
Feb 11, 2016, 2:2...	dte975d1	[r] origin/patch-google-verification (pull request #14)
Feb 11, 2016, 2:1...	3bc3290	[r] origin/patch-google-verification. Update google verification...
Feb 11, 2016, 1:1...	dbae7f0	Replace outdated Atlassian logo in footer with base-64 en...
Feb 15, 2016, 11...	f6f7b45	Updated Mac min...spec to 10.10
Feb 15, 2016, 10...	72d32a8	Michael Min... Merged in hero_images (pull request #13)
Feb 11, 2016, 3:3...	246c0ff1	Joel Unger... [r] origin/hero_images... [r] hero_images Used TinyPNG to c...
Feb 9, 2016, 2:59...	9d9439c	Replacing hero images with new version of SourceTree
Feb 15, 2016, 10...	ce756d3	Merged in bug/date-https (pull request #12)
Jan 7, 2016, 12:2...	85367bb	[r] origin/bug/date-https fixed date and https errors
Feb 8, 2016, 3:55...	4f9b557	Patrick Tho... New Favicon
Feb 3, 2016, 2:09...	384e46d5	Rahul Chhab... [r] origin/search-console-access search console google ver...
Dec 15, 2015, 2:20...	6fa47a9	Mike Minns... updated to move supported version to OSX 10.9+
Nov 23, 2015, 2:2...	8dd87bb	remove extra , when a line is skipped due to empty server
Nov 23, 2015, 2:1...	faa195e	Mike Minns... Skip records with empty server/user id as gas rejects them
Nov 23, 2015, 2:0...	0cdfe96	Mike Minns... corrected paths after merge
Nov 23, 2015, 1:5...	051ab1b	corrected column counting
Nov 23, 2015, 1:5...	a723bc2	Mike Minns... Merge branch 'au2gex'
Nov 23, 2015, 1:5...	65fd580	Mike Minns... deal with invalid instances
Nov 23, 2015, 1:0...	500a892	Michael Min... Merged in au2gex (pull request #11)

- **GitHub for Desktop (GRATIS).** Disponible en <https://desktop.github.com>

The screenshot shows a GitHub commit interface. At the top, there are tabs for 'Current repository' (set to 'desktop'), 'Current Branch' (set to 'progress-reporting'), and 'Publish branch' (with a 'Publish this branch to GitHub' button). Below these are tabs for 'Changes' (selected) and 'History'. Under 'Changes', it says '1 changed file' and lists 'app/src/ui/app.tsx'. The main area shows a diff of the file content:

```
@@ -956,6 +956,8 @@ export class App extends React.Component<IAppProps, IAppState> {
 956   956
 957   const state = selection.state
 958   const remoteName = state.remote ? state.remote.name : null
 959 +  const progress = state.pushProgress || state.pullProgress
 960 +
 961   961   return <PushPullButton
 962     dispatcher={this.props.dispatcher}
 963     repository={selection.repository}
 964   965   @@@ -963,7 +965,7 @@ export class App extends React.Component<IAppProps, IAppState> {
 965     965     remoteName={remoteName}
 966     966     lastFetched={state.lastFetched}
 967     967     networkActionInProgress={state.pushPullInProgress}
 968 +  968     progress={state.pushProgress}
 969   969   >
 970   970 }
 971
```

Below the diff, there is a sidebar with a 'Show progress in toolbar' button, a 'Description' input field, and a 'Commit to progress-reporting' button.

Adicionalmente, muchos de los nuevos entornos integrados de desarrollo, suelen incluir *out-of-the-box*, herramientas gráficas para la gestión de Git.

14. Despedida

Si estás leyendo estas líneas, tengo que darte personalmente la **enhorabuena por el esfuerzo** que has hecho. Has llegado al final de esta guía, y si todo ha ido bien, deberías tener a estas alturas unos conocimientos solventes, que te van a permitir adentrarte con todas las garantías en el control de versiones con **Git**.

Si hay algo de esta guía que creas que podemos mejorar, no dudes en indicárnoslo en cualquiera de nuestros medios de contacto (están en el capítulo 1. *Bienvenida*, en el apartado **El Autor**).

¡Estamos atentos a todo vuestro *feedback*!

Gracias por confiar en nosotros para adentrarte en el maravillo mundo de **Git**.

¡Hasta pronto!