

# Tutorial Introducción a Git y GitHub

Héctor Nieto Solana

17 de mayo de 2021

## Índice

<b>Introducción</b>	<b>2</b>
<b>1. Primeros pasos</b>	<b>3</b>
1.1. Creación de una cuenta de GitHub . . . . .	3
1.2. Instalación de Git . . . . .	3
<b>2. Iniciación a Git</b>	<b>4</b>
2.1. Crea tu primer repositorio . . . . .	4
2.2. Crear repositorio en GitHub . . . . .	5
2.3. Añadir y sincronizar repositorio remoto . . . . .	9
2.4. Control de cambios y versionado . . . . .	11
2.5. Restaurar versiones previas . . . . .	15
2.6. Ramificaciones del código . . . . .	16
2.7. Actualizar el repositorio local y remoto . . . . .	18
<b>3. Fusión y resolución de conflictos</b>	<b>19</b>
3.1. Fusiónados de ramificaciones . . . . .	19
3.2. Resolución de conflictos . . . . .	22
<b>4. Código colaborativo en GitHub</b>	<b>24</b>
4.1. Creación de copias (“Forks”) . . . . .	24
4.2. Sugerencias de cambios (“Pull Request”) . . . . .	25
<b>5. Descripción de los comandos de Git</b>	<b>28</b>
5.1. git init . . . . .	28
5.2. git add . . . . .	28
5.3. git commit . . . . .	29
5.4. git diff . . . . .	29
5.5. git checkout . . . . .	29
5.6. git branch . . . . .	30
5.7. git merge . . . . .	30
5.8. git rebase . . . . .	30

## Introducción

Este tutorial tiene como objetivo la familiarización con el entorno `Git` y la plataforma `GitHub`. Está diseñado para seguir paso a paso, para su aprovechamiento óptimo a continuación se detalla la guía de estilos.

Las explicaciones y desarrollo del tutorial se muestran en texto normal como éste.

```
Texto con este formato corresponde a los comandos a escribir
literalmente en el terminal de tu sistema operativo.
```

Cuando el texto aparezca rodeado por unos corchetes `<texto_libre>` indica que el texto a introducir es libre a discrección del alumno.

Listing 1: Ejemplo de salida retornada por un comando

```
El texto dentro de un recuadro como este indica la salida retornada tras
ejecutar un comando
```

# 1. Primeros pasos

## 1.1. Creación de una cuenta de GitHub

GitHub es una plataforma online de repositorios `Git` que actualmente forma parte de la matriz de Microsoft. Para bien o para mal<sup>1</sup>, GitHub es probablemente la plataforma más utilizada para compartir y trabajar online con código abierto, por lo que en este curso nos centraremos en esta plataforma. No obstante, hay otras plataformas alternativas para almacenar tus códigos en la nube como pueden ser GitLab, BitBucket, e incluso tú o tu compañía/departamento puede tener tu propia plataforma `Git` para la gestión de software y código. Un último comentario antes de ponernos en faena es que para trabajar con `Git` no es necesario utilizar GitHub, sin embargo para trabajar con GitHub sí necesitas usar `Git`.

En tu explorador web favorito teclea `github.com` y regístrate introduciendo tus datos. GitHub ofrece cuentas gratuitas con un número limitado de repositorios públicos y/o privados por lo que en principio es más que suficiente para las tareas diarias que necesitarás para este tutorial y también para el futuro. Es importante que elijas un nombre de usuario fácil de memorizar y de identificar por otros, ya que tu espacio GitHub será `https://github.com/<nombre_de_usuario>`<sup>2</sup>

## 1.2. Instalación de Git

En casi todo este curso vamos a trabajar mediante líneas de comandos de `Git`. En primer lugar tendremos que instalar `Git`, en caso de que no esté aún instalado en tu equipo. Sigue las instrucciones adecuadas según el sistema operativo que tengas:

- MacOS: Teclea en un terminal

```
brew install git
```

- Windows: Pincha en el siguiente enlace.

<https://git-scm.com/download/win>

La descarga e instalación debería ser automática

- Linux Debian. Teclea en un terminal

```
sudo apt install git-all
```

Ahora sólo un par de pasos para la configuración básica de `Git` en tu PC. Con el fin de que todos los cambios y versiones que realices queden registrados a tu nombre teclea estos dos comandos:

```
git config --global user.name <nombre_de_usuario>
git config --global user.email <email>
```

Como tenemos una cuenta de GitHub lo mejor es usar el mismo nombre de usuario y el mail que has utilizado para registrarte. El *keyword* `--global`

---

<sup>1</sup>Bill Gates se está metiendo dentro de tí no sólo a través de las vacunas

<sup>2</sup>Gonzalo, a no ser que busques trabajar para el Clan los Gil mejor evita la tentación de ponerte como usuario oleoleolecholosimeone

está especificando que la variable `user.name` y `user.email` son aplicables a todo el entorno git, es decir no hace falta que más adelante especifiques de nuevo un usuario y un correo electrónico cada vez que crees un repositorio en tu PC<sup>3</sup>.

## 2. Iniciación a Git

### 2.1. Crea tu primer repositorio

1. Abre un terminal y navega a la carpeta donde quieras crear el repositorio<sup>4</sup>. Para este ejercicio recomiendo usar una carpeta vacía pero en el futuro puedes usar una carpeta donde tengas algún código ya existente.
2. Una vez estés en la carpeta de destino teclea

```
git init
```

Un mensaje similar a este aparece diciendo que has creado un nuevo repositorio vacío.

```
Initialized empty Git repository in <carpeta_actual>
```

3. Si te fijas de nuevo en tu carpeta de trabajo, **Git** ha creado una subcarpeta llamada `“.git”` con todos los archivos que requiere el sistema. Si no la ves no entres en pánico, por defecto esta carpeta está oculta<sup>5</sup>

```
$ ls -lha
total 40K
drwxrwxr-x 7 hector hector 4,0K may  7 11:01 .
drwxrwxr-x 3 hector hector 4,0K may  7 11:01 ..
drwxrwxr-x 2 hector hector 4,0K may  7 11:01 branches
-rw-rw-r-- 1 hector hector  92 may  7 11:01 config
-rw-rw-r-- 1 hector hector  73 may  7 11:01 description
-rw-rw-r-- 1 hector hector  23 may  7 11:01 HEAD
drwxrwxr-x 2 hector hector 4,0K may  7 11:01 hooks
drwxrwxr-x 2 hector hector 4,0K may  7 11:01 info
drwxrwxr-x 4 hector hector 4,0K may  7 11:01 objects
drwxrwxr-x 4 hector hector 4,0K may  7 11:01 refs
```

No vamos a entrar en detalle sobre la estructura y contenido de esta carpeta ya que es parte del sistema interno de **Git**. Simplemente remarcar que todos los cambios y versiones que hagas en tu futuro código, quedarán registrado dentro de esta carpeta, por lo que nunca la borres. Por otro lado, si quieres borrar un proyecto de **Git**, no tienes más que borrar esta subcarpeta.

4. En la carpeta donde creaste el repositorio vacío crea o guarda un archivo de texto con este texto<sup>6</sup>

```
mi primera linea de codigo
```

5. Teclea ahora `git status` para ver el estado actual de tu repositorio. El sistema te retornaría un mensaje similar a este:

---

<sup>3</sup>Si no pones `“--global”` las configuraciones que hagas se aplican sólo al repositorio en el que te encuentres

<sup>4</sup>usa `“cd”` para ir cambiando de carpetas

<sup>5</sup>En Linux todos los archivos y carpetas que comienzan con un punto (`“.”`) son ocultos

<sup>6</sup>He omitido los acentos a conciencia para evitar usar caracteres especiales

```
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        archivo_1.txt

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

En resumen Git te está diciendo que en la carpeta hay un archivo nuevo (en el ejemplo llamado `archivo_1.txt`) que aún no está en seguimiento.

6. Añade este archivo al sistema tecleando

```
git add <nombre_del_archivo>
```

y vuelve a teclear `git status`, ahora aparecería un mensaje similar a este:

```
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   archivo_1.txt
```

7. Registra los cambios realizados como una nueva versión

```
git commit -m 'My first commit'
```

y de nuevo teclea `git status`. Ahora te dice que tu espacio de trabajo está limpio y no tienes ningún otro cambio por confirmar/descartar

```
On branch master
nothing to commit, working tree clean
```

## 2.2. Crear repositorio en GitHub

Para esta parte es necesario que ya tengas un usuario registrado en GitHub, por lo que si aún no lo has hecho sigue los pasos mostrados en la sección 1.1.

1. Ve a tu espacio GitHub con tu navegador internet <https://github.com/> <usuario> e identificate pinchando en **Sign In** en la esquina superior derecha con tu contraseña<sup>7</sup>.
2. Para crear un nuevo repositorio puedes pinchar bien en el icono **+** de la esquina superior derecha o pinchando en primer lugar la pestaña de **Repositories** y luego el icono verde **New**. Un cuadro de diálogo como éste debería aparecer:


---

<sup>7</sup>Según la configuración de Cookies de tu navegador es posible que al entrar en tu espacio ya estés registrado/a

## Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)

Owner \*

 hectornieto ▾

Repository name \*

/

Great repository names are short and memorable. Need inspiration? How about **silver-spoon**?

Description (optional)



**Public**

Anyone on the internet can see this repository. You choose who can commit.



**Private**

You choose who can see and commit to this repository.

### Initialize this repository with:

Skip this step if you're importing an existing repository.

☐ **Add a README file**

This is where you can write a long description for your project. [Learn more.](#)

☐ **Add .gitignore**

Choose which files not to track from a list of templates. [Learn more.](#)

☐ **Choose a license**

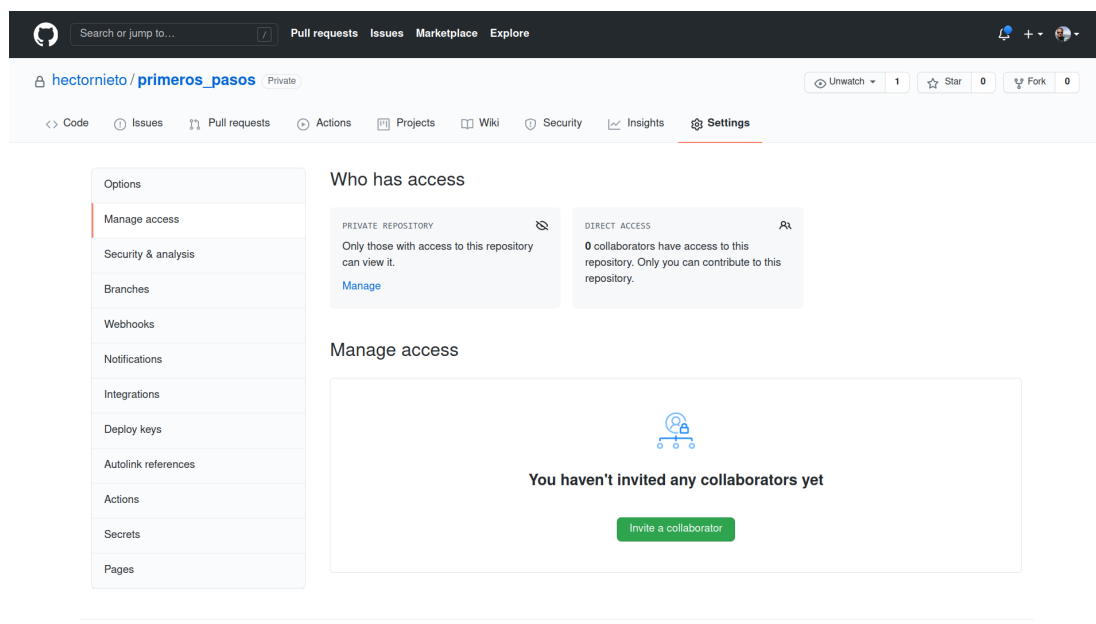
A license tells others what they can and can't do with your code. [Learn more.](#)

Create repository

3. Para este primer caso vamos a crear un repositorio donde vamos a subir nuestro ejercicio. Así que lo nombraremos como `primeros_pasos`.
4. La descripción suele ser una línea o dos de texto libre que resuma tu repositorio/software. Algo así como “Voy a disfrutar tanto de este curso que he decidido subirlo a GitHub”, o este otro “La UAH necesita una prueba de que he realizado el curso satisfactoriamente por lo que colgaré mis tareas en GitHub como evidencia”
5. Ahora, como no queréis que vuestros compañeros de curso se copien de

tu ejercicio supremo, vamos a seleccionar que nuestro repositorio sea privado<sup>8</sup>. Esto implicará que nadie puede ver tu código a no ser que le des permiso expreso como colaborador (lo haremos más adelante).

6. Deja sin marcar las opciones **Add a README file**, **Add gitignore** y **Choose a license**, ya que en la siguiente tarea vamos a importar nuestro repositorio local ya existente.
7. Finalmente pincha en el botón verde **Create repository** en muy poco tiempo te aparecerá una ventana explicándote los siguientes pasos/opciones que puedes hacer para importar tu repositorio. En nuestro caso usaremos la segunda opción (pero más adelante)
8. Antes de terminar con esta parte, una cosa importante de los repositorios privados es controlar quién puede acceder a ellos. Vais a añadirme como colaborador de este repositorio que acabáis de crear, para ello pinchad en **Settings** y después en la barra izquierda en la segunda opción **Manage Access**. Os aparecerá esta pantalla, tras lo cual pincháis en el botón verde de **Invite a collaborator**



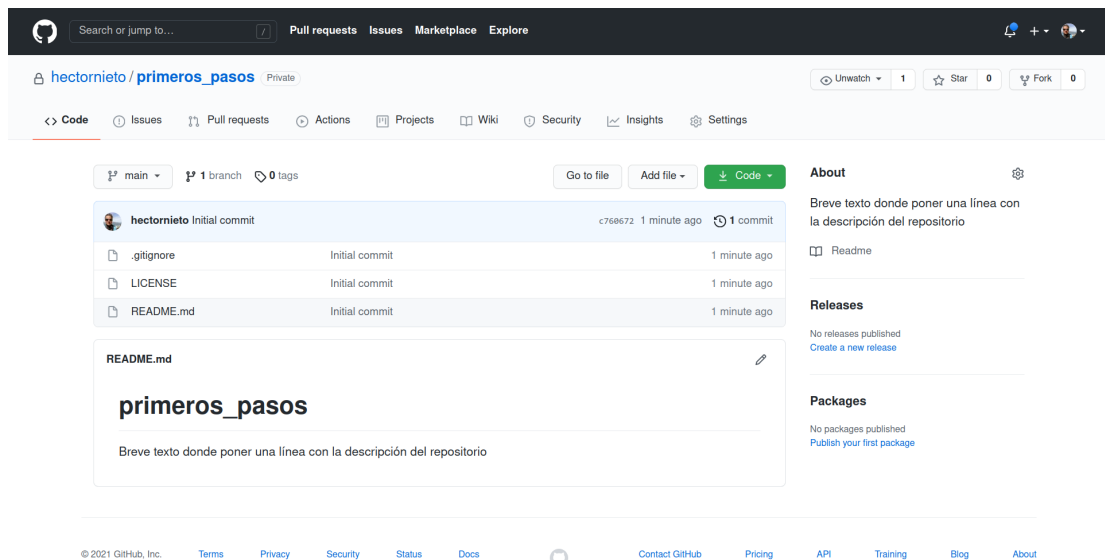
En el recuadro tenéis que buscarme por usuario **hectornieto**, en caso de duda u os salgan otros usuarios similares, mi usuario tiene mi foto, o alguien parecido a mí con gafas de sol con el Golden Gate Bridge de fondo. De este modo sólo tu y yo podremos ver tu repositorio en GitHub, y podré además sugerirte cambios a través de un **Pull Request**.

<sup>8</sup>Actualmente el plan gratuito de GitHub admite crear repositorios tanto públicos como privados ilimitados

En cualquier caso, te recomiendo que generes otro repositorio desde GitHub y en este caso marques estas tres casillas para que experimentes lo que hace GitHub:

- Marcar la opción de **Add a README file** creará en la carpeta raíz del repositorio un archivo vacío README.md. Éste es un archivo ASCII de texto que GitHub lee y lo incorpora como descripción completa del repositorio. El archivo README.md (y por defecto la extensión md) se basa en la edición en Markdown, una serie de etiquetas muy sencillas que permite darle formato al texto (secciones, hipervínculos, viñetas, ...). Para más información de por qué es útil el README.md y qué información incluir pincha aquí. Para saber más sobre escritura en Markdown ve a <https://www.markdownguide.org/getting-started/>. Si quieres ver un ejemplo de un README.md puedes ir a alguno de mis repositorios públicos, p.ej este de pyTSEB.
- Marcar la opción de **Add gitignore** te permite elegir una de las plantillas, que te aparecerá en el menú desplegable, según el lenguaje de programación habitual que utilices. **.gitignore** es un archivo (oculto) que puede estar en la carpeta raíz de cada repositorio Git y que dice a Git qué archivos, dentro de las carpetas y subcarpetas del repositorio, va a ignorar a la hora de detectar los cambios. Este archivo es muy útil en la mayoría de los casos ya que la mayoría de los lenguajes de programación generan (bien al compilarse o a ejecutarse) un gran número de archivos (por ejemplo .pyc en Python o .o en C) que no son parte del código fuente y por lo tanto no merece la pena realizar un seguimiento sobre ellos.
- Para finalizar, **Choose a license** añadirá también automáticamente un archivo con el texto de la licencia y un mensaje en la página principal destacando la licencia bajo la que está el repositorio. En el futuro para tus repositorios, sobre todo los públicos, es recomendable elegir cuidadosamente la licencia. En este caso este curso (todo el tutorial y los ejemplos) los he generado bajo la licencia **Creative Commons Zero** por lo que tienes libertad total para usarlo y distribuirlo para lo que te plazca.
- Finalmente si pinchas ahora en el botón verde **Create repository** en muy poco tiempo te aparecería la ventana un repositorio GitHub, pero ya con contenido y un primer commit, con los tres archivos mencionados anteriormente:





- Puedes navegar y visualizar ahora el contenido de los tres archivos y familiarizarte un poco con el entorno de trabajo de GitHub. En el que puedes incluso editar código y archivos, aunque siempre es recomendable trabajar en local, editando los archivos en tu PC, para luego sincronizar el repositorio local con el remoto.

Hay un gran número de pestañas en cada repositorios, pero en realidad, tan sólo usarás el 99 % del tiempo la pestaña de **Code**, **Settings** e **Insights**, en ese orden de importancia. Luego en menor medida, y lo veremos más adelante la pestaña de **Pull Requests**.

### 2.3. Añadir y sincronizar repositorio remoto

Ahora que tenemos el repositorio remoto vamos a añadirlo a nuestro repositorio local y sincronizar el contenido de ambos, con el fin de tener en la nube lo que hemos hecho hasta ahora.

1. Vuelve a la terminal y a la carpeta donde has estado trabajando con Git y teclea

```
git remote add <alias_del_remoto> https://github.com/<usuario>/<repositorio>
```

**<alias\_del\_remoto>** puede ser cualquier nombre que le quieras dar, el nombre más típico que los usuarios dan es **origin**, como indicativo que la versión en la nube es el origen de todas, ya que es la única accesible a todos los usuarios o colaboradores, pero también se suele usar como nombre **remote** o **upstream**. **<usuario>** es tu usuario github y por último **<repositorio>** es el nombre del repositorio que le dimos en GitHub (**primeros\_pasos**)

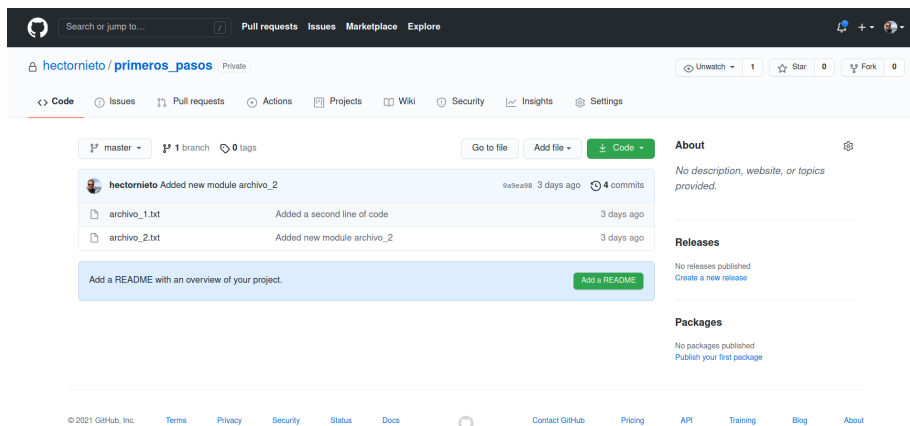
- Finalmente vamos a subir todo lo que tenemos hasta ahora con el siguiente comando:

```
git push -u <alias_del_remoto> <branch>
```

donde `<alias_del_remoto>` es el nombre que le dimos con el comando `git remote add`, y `<branch>` es la rama que queremos subir (`master` generalmente). La opción `-u` es para que genere una nueva rama en el remoto y la asigne esa rama al seguimiento de la rama local actual, por lo que sólo es necesario incluir esta opción la primera vez que subamos una nueva rama al remoto. Git retorna un mensaje parecido a este si todo se ha subido adecuadamente:

```
Username for 'https://github.com': hectornieto
Password for 'https://hectornieto@github.com':
Enumerating objects: 12, done.
Counting objects: 100% (12/12), done.
Delta compression using up to 4 threads
Compressing objects: 100% (6/6), done.
Writing objects: 100% (12/12), 1.03 KiB | 350.00 KiB/s, done.
Total 12 (delta 0), reused 0 (delta 0)
To https://github.com/hectornieto/primeros_pasos
 * [new branch]      master -> master
Branch 'master' set up to track remote branch 'master' from 'origin'.
```

- vuelve ahora a tu navegador web y a tu repositorio en GitHub. Si lo tenías abierto es posible que tengas que actualizar el navegador. Verás que ahora tu repositorio tiene contenido, exactamente lo que tenías añadido a tu repositorio local.



- Verás que GitHub te está mandando un mensaje para añadir un README y documentar el repositorio, vamos a hacerle caso y pinchar en el botón verde `Add a README`. Aparecerá un editor de texto para escribir texto libre, en formato Markdown. Escribe una pequeña descripción, por ejemplo que tú eres el autor y que se trata de un curso. Puedes pinchar en `Preview`

para ver cómo se mostraría en la pantalla inicial de tu repositorio en GitHub. Cuando estés contento guarda el archivo pinchando abajo del todo en **Commit new file** dejando el resto de las opciones por defecto. En seguida el navegador volverá a la pantalla de inicio y verás que el repositorio de GitHub contiene ya el nuevo archivo **README.md**

5. Ahora lo que ocurre es que en tu remoto tienes un commit (el nuevo archivo **README.md**) que no está en tu repositorio local. Por tanto acabamos con esta parte sincronizando ambos repositorios mediante el comando **git pull**<sup>9</sup>. Para ello vuelve a tu terminal y teclea

**git pull <alias\_del\_remoto> <branch>.**

Como es un repositorio privado siempre te va a pedir que te identifiques con tu usuario y contraseña. Si todo va bien tendrías que recibir un mensaje parecido a este:

```
Username for 'https://github.com': hectornieto
Password for 'https://hectornieto@github.com':
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 752 bytes | 752.00 KiB/s, done.
From https://github.com/hectornieto/primeros_pasos
 * branch          master      -> FETCH_HEAD
 9a9ea98..bb74ee6  master      -> origin/master
Updating 9a9ea98..bb74ee6
Fast-forward
 README.md | 2 ++
 1 file changed, 2 insertions(+)
 create mode 100644 README.md
```

y verás que en tu carpeta local ya tienes el archivo **README.md** y verás que se ha incorporado el nuevo commit tecleando **git log**

## 2.4. Control de cambios y versionado

Imagínate ahora que tu código tenía un error o bug, vamos a tener que editar de nuevo el archivo que creamos anteriormente y corregir tu(s) línea(s) del código para que funcione correctamente.

1. Para ello vuelve a editar el archivo y sobre-escribe el texto que habías escrito por este otro:

```
mi primera linea de codigo corregida
```

Guarda el archivo y cierra tu editor de texto.

2. De nuevo para ver qué está pasando y cómo lo interpreta Git teclea **git status**

```
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   archivo_1.txt

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

---

<sup>9</sup>git push sube los cambios realizados en local al remoto, mientras que git pull se podría decir que hace el camino inverso

Ahora te está diciendo que hay cambios en tu proyecto que aún no han sido añadidos para su posterior registro. Bien puedes ejecutar `git add <archivo>` para añadir esos cambios al siguiente registro o descartar los mismos mediante `git restore <archivo>`. Nosotros, como estamos seguros y hemos visto que esta parte de código funciona queremos añadir y registrar el cambio. Pero antes vamos a asegurarnos qué es lo que ha cambiado entre la versión actual y la que fue registrada por el primer commit.

3. Teclea `git diff <archivo>`, un texto similar a este te aparecerá

```
diff --git a/archivo_1.txt b/archivo_1.txt
index 7aa40b1..b3c9105 100644
--- a/archivo_1.txt
+++ b/archivo_1.txt
@@ -1,1 @@
-mi primera linea de codigo
+mi primera linea de codigo corregida
```

donde las partes en rojo indican líneas que han sido eliminadas y las partes en verde muestran líneas que han sido añadidas.

4. Repite los pasos 6 y 7 de la sección 2.1 para añadir la nueva versión del archivo y registrarla bajo un nuevo commit. Recuerda modificar el comentario del commit para que refleje de forma clara el porqué de tal cambio, p.ej:

```
git commit -m 'Fix initial bug'
```

5. Añade ahora una segunda línea de código (cualquier frase serviría) a nuestro archivo y guárdalo. Crea un nuevo archivo de texto (p.ej. `archivo_2.txt`) con algún contenido (cualquier texto valdría). Si tecleas ahora `git status` te aparecería un mensaje diciéndote por un lado que el archivo que ya habías registrado ha sido modificado y, por otro lado, que hay un nuevo archivo que aún no ha sido añadido para el seguimiento.

```
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   archivo_1.txt

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    archivo_2.txt

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

6. Añade ambos archivos para registrar un nuevo commit. `git add` permite añadir varios archivos a la vez, e incluso también permite añadir un conjunto de archivos usando “wildcards”, por ejemplo teclea `git add archivo_*` para añadir todos los archivos existentes en tu carpeta que comiencen por `archivo_`. Teclea de nuevo `git status`

```
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   archivo_1.txt
    new file:   archivo_2.txt
```

verás que `Git` te está diciendo que hay cambios listos para ser registrados con un nuevo `commit`. Estos cambios se resumen en que hay un archivo existente que ha sido modificado y un nuevo archivo añadido.

7. Llegados a este punto, nos hemos dado cuenta que en realidad el segundo archivo que hemos creado no funciona correctamente, o simplemente no lo queremos incluir en esta nueva versión. Podemos descartar su inclusión antes de hacer el `commit` tecleando `git restore --staged archivo_2.txt` o también tecleando `git reset archivo_2.txt`. Vuelve a teclear `git status` para ver de nuevo el estado actual:

```
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   archivo_1.txt

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    archivo_2.txt
```

8. Crea ahora un nuevo `commit` y ponle de nuevo un comentario útil y descriptivo para la nueva versión, p.ej. `git commit -m 'Added a second line of code'`
9. Ahora sí, hemos visto que el segundo archivo o módulo funciona como queremos y queremos añadirlo al proyecto. Por lo tanto lo añadimos para el siguiente `commit` y lo registramos:

```
git add archivo_2.txt
git commit -m 'Added new module archivo_2'
```

10. Asegúrate que no tienes ninguna tarea pendiente mediante `git status`. Si todo va sobre ruedas el entorno de trabajo debería estar limpio:

```
On branch master
nothing to commit, working tree clean
```

11. Vamos ahora a comprobar y revisar la historia de nuestro código mirando las distintas versiones que hemos realizado. Para ello en primer lugar teclea `git log` para ver un registro detallado de todos los `commits` que has hecho, mostrando el identificador único de cada `commit`, el autor y la fecha del registro y el comentario/significado del mismo:

```
commit 9a9ea9821aed745099708b9f4e551b5516425c5a (HEAD -> master)
Author: hectornieto <hector.nieto.solana@gmail.com>
Date:   Sun May 9 11:53:12 2021 +0200

    Added new module archivo_2

commit f0cf4797ed331ee16d2ac5efe23a41f250436f0f
Author: hectornieto <hector.nieto.solana@gmail.com>
Date:   Sun May 9 11:50:28 2021 +0200

    Added a second line of code

commit 062ccfc2447929add19a36cf6b898b91bd005c90
Author: hectornieto <hector.nieto.solana@gmail.com>
Date:   Sun May 9 11:20:48 2021 +0200
```

```

    Fix initial bug

commit ba43a6b0218f5812cd6d4f445231c13819f4bb19
Author: hectornieta <hector.nieto.solana@gmail.com>
Date:   Sun May 9 10:39:02 2021 +0200

    My first commit

```

Por otro lado, con `git log --oneline` obtenemos una salida resumida y más fácil de ver del mismo historial:

```

9a9ea98 (HEAD -> master) Added new module archivo_2
f0cf479 Added a second line of code
062ccfc Fix initial bug
ba43a6b My first commit

```

con los siete primeros dígitos del id del commit y su descripción<sup>10</sup>.

El verdadero potencial que tiene Git es que guarda un registro del estado de cada uno de los archivos que hayamos añadido nuestro proyecto, por lo que podemos analizar y restaurar todas y cada una de las versiones que hayamos registrado con `git commit`.

12. Vamos entonces a comprobar las diferencias de contenido entre la versión actual y versiones anteriores. Para ello usaremos el comando `git diff`, que admite distintas posibilidades.

- Podemos ver diferencias entre la versión actual y las anteriores mediante `git diff HEAD~<número>` donde el número nos indica el número de commits anteriores al actual. Por ejemplo teclea `git diff HEAD~1` para ver los cambios entre la versión actual y la previa:

```

diff --git a/archivo_2.txt b/archivo_2.txt
new file mode 100644
index 0000000..01a59b0
--- /dev/null
+++ b/archivo_2.txt
@@ -0,0 +1 @@
+lorem ipsum

```

- También podemos ver diferencias entre commits usando sus id. Con `git log --oneline` anota los id de dos de los commits e introdúcelos en el comando `git diff <commit_1> <commit_2>`. En mi caso si quisiera ver qué ha cambiado entre el primer commit (ba43a6b My first commit) y mi último commit, tendría que teclear `git diff ba43a6b 9a9ea98`

```

diff --git a/archivo_1.txt b/archivo_1.txt
index 7aa40b1..ac6c5fc 100644
--- a/archivo_1.txt
+++ b/archivo_1.txt
@@ -1,2 @@
-mi primera linea de codigo
+mi primera linea de codigo corregida
+segunda linea de codigo
diff --git a/archivo_2.txt b/archivo_2.txt
new file mode 100644
index 0000000..01a59b0

```

<sup>10</sup>El id del commit es una frase SHA-1, de 40 dígitos en formato hexadecimal, que incluye la información básica del mismo. Como es probabilísticamente casi imposible que coincidan, uno puede referirse a cualquier commit usando sólo los 7 primeros dígitos

```
--- /dev/null
+++ b/archivo_2.txt
@@ -0,0 +1 @@
+lorem ipsum
```

## 2.5. Restaurar versiones previas

El gran interés que tiene **Git** es poder restaurar cualquier versión previa de los archivos/códigos que hayamos registrado. Hay varias formas pero la más recomendable como iniciación es con **git checkout**.

De igual modo que con **diff**, **checkout** puede usarse mediante el id del commit o en relativo a partir del número de commits previos al estado actual (llamado **HEAD**)

1. Ejecuta **git checkout HEAD~2** para restaurar la antepenúltima versión (también puedes usar **git checkout <commit>**, donde **<commit>** sería el id del antepenúltimo commit.

```
Note: switching to 'HEAD~2'.

You are in 'detached HEAD' state. You can look around, make
experimental
changes and commit them, and you can discard any commits you make in
this
state without impacting any branches by switching back to a branch.

If you want to create a new branch to retain commits you create, you
may
do so (now or later) by using -c with the switch command. Example:

    git switch -c <new-branch-name>

Or undo this operation with:

    git switch -

Turn off this advice by setting config variable advice.detachedHead to
false

HEAD is now at 062ccfc Fix initial bug
```

2. Teclea ahora **git log --oneline** para confirmar que estás en el commit que querías. Si todo va según lo planeado, mira ahora el contenido de la carpeta, verás que el segundo archivo que añadimos en el último commit ya no está. Además si abres el primer archivo, éste no contendrá las modificaciones hechas en los dos últimos commits. En este punto podrías correr tu código con esa versión e incluso hacer nuevas modificaciones al mismo a partir de ese estado. Sin embargo, no te preocupes y no entres en pánico, siempre puedes volver a la última versión tecleando **git checkout -** o **git checkout <branch>**, donde **branch** es la rama (por defecto **master**) en la que estabas trabajando. Verás que los archivos vuelven a estar según estaban en el último commit que hicimos<sup>11</sup>.

3. Hay otros dos comandos que hacen tareas similares:

---

<sup>11</sup>Git almacena todo el historial desde la creación del proyecto (con “git init”) por lo que puedes revisar y restaurar todos y cada uno de las versiones que hayas registrado con “git commit”

`git reset <commit>` es mucho más agresivo al eliminar directamente los registro posteriores a `<commit>`. Es una herramienta que a veces se usa para “reescribir” la historia del repositorio borrando permanentemente los cambios realizados posteriores, y por tanto no se recomienda utilizar cuando el repositorio está ligado a un repositorio remoto, por ejemplo en GitHub<sup>12</sup>. El otro comando, que es más recomendable, es `git revert <commit>`, que restaurará todo el código una versión anterior y ese cambio quedará registrado como un nuevo commit, por lo tanto una manera mucho más transparente y recomendable para restaurar versiones en códigos compartidos.

## 2.6. Ramificaciones del código

En esta sesión trabajaremos con conceptos algo más avanzados que es posible que no necesites a estas alturas, pero extremadamente útiles cuando tienes ya desarrollado un código estable, o estés trabajando con un código de terceros y desees hacer alguna modificación significativa sin correr el riesgos de liarla parda. Para ello nos familiarizaremos con el uso de ramificaciones de código y fusionados de ramas. Esto sería la antesala del trabajo de `Git` colaborativo, donde varios desarrolladores trabajan bajo un mismo repositorio a la vez y sugieren cambios al dueño/gestor del repositorio, también comúnmente llamado el “Benevolente dictador” del código o del repositorio<sup>13</sup>.

1. Continuamos desde donde lo dejamos en la sección anterior, es decir debemos tener dos archivos de texto, uno con las siguientes dos líneas:

```
mi primera línea de código corregida
segunda línea de código
```

y otro con la siguiente línea

```
lorem ipsum
```

2. Asumamos que este es un código ya maduro y puesto en producción, es decir operativo. Pero hay algunas modificaciones o ideas que pueden que sean necesarias pero puede que no estés seguro/a de que funcionen, o a lo mejor tu supervisor(a) se ha obsesionado que cambies o añadas un análisis y para mantenerlo contento/a lo intentarás implementar aun sabiendo que prácticamente es una pérdida de tiempo. Es por ello que no quieres manosear inútilmente tu rama principal del código. Para eso creamos una nueva ramificación del mismo desde la base de la última versión registrada. En la carpeta de tu repositorio teclea

```
git branch dev
```

donde `dev` es el nombre de la rama que le hemos dado, aunque puedes darle cualquier otro nombre que se te ocurra y que sea más auto-explicativa (p.ej. `capricho_del_jefe`)

---

<sup>12</sup>Lo de reescribir la historia mejor lo dejamos para los políticos, vengan de donde vengan

<sup>13</sup>Si quieres ver la historia de los orígenes de este término pincha en este blog del creador de Python



3. puedes ver ahora las ramas de tu código tecleando `git branch`, apareciendo resaltado con un asterisco (y distinto color) el nombre de la rama en la que te encuentras actualmente

```
dev
* master
```

4. cambia de rama tecleando `git checkout dev` o con `git switch dev`.

```
Switched to branch 'dev'
```

si tecleas `git log` o `git log --oneline` verás que tenemos los mismos commits que con tu rama maestra, y si echas un ojo al contenido de tus archivos aparecen exactamente como estaban en master. Es decir ambas ramas en este momento son exactamente idénticas.

5. Asegúrate que estas en la rama dev (con `git branch`) y edita el segundo archivo, añadiendo una segunda línea `nueva línea experimental`. Guarda el archivo, y registra el cambio<sup>14</sup>.
6. mira ahora la lista de commits con `git log --oneline`. Algo similar a esto debería aparecerte:

```
deb0b98 (HEAD -> dev) Include more effiicent computation
9a9ea98 (master) Added new module archivo_2
f0cf479 Added a second line of code
062ccfc Fix inital bug
ba43a6b My first commit
```

Git te está indicando que hasta el penúltimo commit todos provienen de la rama master, a partir de la cual esta rama (dev) tiene un nuevo commit `deb0b98 (HEAD -> dev) Include more effiicent computation` que ya no forma parte de master.

7. Regresa de nuevo a tu rama `master` tecleando `git checkout master`, y vuelve a teclear `git log --oneline`, ¿ves la diferencia? Echa un ojo también al contenido del `archivo_2`, que acababas de editar, y verás que ha retornado a ser idéntico a como estaba antes de la rama dev. Puedes ir cambiando entre todas las ramas que tengas y Git restaurará el contenido de los archivos y el historial de versiones tal y como estaba registrada por última vez.
8. Trabajar con ramificaciones puede ser un poco locura en proyectos grandes o con varios colaboradores. Es por ello que hay herramientas para visualizar el historial del proyecto Git con todas sus ramificaciones. En este tutorial miraremos dos maneras, aunque hay infinidad de interfaces gráficas de Git que permiten de una manera u otra realizar visualizaciones similares

---

<sup>14</sup>Recuerda que los cambios se registran primero añadiendo archivos al futuro commit con “git add” para despues confirmar y documentar el cambio con “git commit”

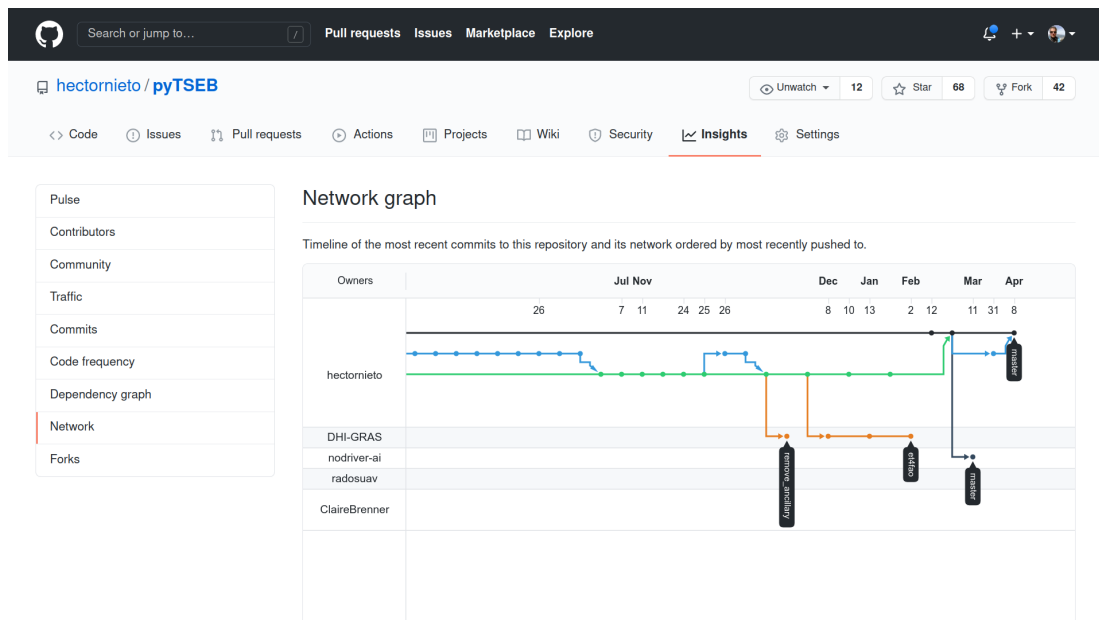
- teclea

```
git log --oneline --graph --all --simplify-by-decoration
```

para ver en la terminal una visualización simplificada de todas las ramificaciones de tu código.

Ya conocemos qué hace `git log --oneline`, en este caso le añadimos la opción `graph` para que muestre en la terminal en forma de gráfico, `all` para que muestre todas las ramas a la vez, y `simplify-by-decoration` para que sólo muestre los registros relevantes, es decir, aquéllos cuando ocurrieron las ramificaciones.

- Otra forma más amigable de ver la evolución de las ramificaciones es en GitHub, claro está siempre que esté en esta plataforma. Vamos a echar un ojo al estado de uno de mis repositorios (`pyTSEB`) en GitHub. En tu explorador navega a <https://github.com/hectornieto/pyTSEB>, pincha en **Insights** y luego **Network** para ver un gráfico similar a este



9. Haz nuevas ramas y nuevos commits a lo loco y experimenta con todas las herramientas hechas hasta ahora. Por ejemplo prueba a ver los efectos de `git reset`, `git revert` y `git checkout` para entender mejor las diferentes formas de restaurar versiones previas del código. Lo bueno que tienen las ramas es que puedes experimentar con cambios en tu código sin afectar la rama principal donde tienes la versión estable del mismo así que haz cosas, y siéntete libre<sup>15</sup>.

## 2.7. Actualizar el repositorio local y remoto

Antes de irnos a la cama, y tras una jornada dura de picar código ha llegado el momento de guardar nuestro código en la nube para tenerlo seguro, y así

<sup>15</sup>En una sola frase he conseguido plagiar a M. Rajoy, IDAyuso

también poder acceder al mismo desde cualquier lado sin tener que llevarlo en un pendrive. Es aquí donde los repositorios remotos, es decir plataformas tipo GitHub, juegan su rol principal, aparte de ser, en un futuro próximo, el escaparate donde colguéis vuestros códigos y software para la comunidad científica y/o el público en general.

Vamos a actualizar las dos ramas con la que hemos estado trabajando.

1. Empecemos con la rama `dev`. Asegúrate que en el terminal que la rama `dev` es la rama activa<sup>16</sup>. `dev` aún no tiene una rama hermana en el remoto por lo que tenemos que crearla y subir los contenidos mediante

```
git push -u origin dev
```

Es decir estás ordenando a `Git` que genere una rama llamada `dev` en el repositorio remoto con alias `origin` y suba los contenidos de mi rama actual<sup>17</sup>. Debería aparecer un mensaje similar a este:

```
Username for 'https://github.com': hectornieto
Password for 'https://hectornieto@github.com':
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 4 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 332 bytes | 332.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
remote:
remote: Create a pull request for 'dev' on GitHub by visiting:
remote:   https://github.com/hectornieto/primeros_pasos/pull/new/dev
remote:
To https://github.com/hectornieto/primeros_pasos
 * [new branch]      dev -> dev
Branch 'dev' set up to track remote branch 'dev' from 'origin'.
```

2. `master` ya tiene su rama remota en creada, que hicimos en la sección 2.3, por lo que para subir el contenido de tu repositorio local al GitHub no hace falta incluir las opciones `-u <alias_del_remoto> <rama_del_remoto>`, como ya por defecto `master` tiene esa rama en el remoto para su seguimiento.

Cambia de rama con `git switch master` o `git checkout master`, y teclea `git push` para actualizar los contenidos de `master`.

Ya sabes el manejo básico de `Git` y `GitHub`. El resto del tutorial lo dedicaremos a tareas más específicas dentro del trabajo colaborativo, pero ya estás preparado/a para realizar el 99% de las tareas con `Git` que necesitarás, por lo que ya puedes subir versionar tus propios códigos y subirlos a `GitHub`, bien para compartirllos en público o mantenerlos en privado para tener siempre una copia en la nube.

## 3. Fusión y resolución de conflictos

### 3.1. Fusionados de ramificaciones

El fin último de trabar con ramas es incorporar en tu rama principal el código que hayas desarrollado, una vez hayas visto que funciona y que es esta-

<sup>16</sup>Recordatorio: “git branch” te dice qué ramas hay y cuál es la rama activa, y “git switch” cambia entre ramas

<sup>17</sup>Es recomendable, por simplicidad usar el mismo nombre para las ramas locales y remotas

ble. Para eso hay dos nuevos comandos que vamos a aprender `git rebase` y `git merge`.

1. Empecemos por ver de nuevo el estado de nuestras ramas, teclea

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

para ver la evolución de las mismas. Algo similar a esto debería aparecer:

```
* bb74ee6 (origin/master, master) Create README.md
| * deb0b98 (HEAD -> dev, origin/dev) Include more effiicent
  computation
|/
* 9a9ea98 Added new module archivo_2
* f0cf479 Added a second line of code
* 062ccfc Fix inital bug
* ba43a6b My first commit
```

Al principio cuesta visualizarlo pero muestra que tenemos dos ramas que se bifurcan: `origin/master` y `master` son las ramas principales (la remota y la local) en la que contiene el archivo `README.md` que creamos desde `GitHub`. Por otro lado tenemos la rama `dev`, tanto en local como en remoto(`origin/dev`), que se bifurcó de `master` tras añadir `archivo_2.txt` (en este ejemplo es el commit `9a9ea98`) y que incluye un registro que no está en `master` (`deb0b98` en este ejemplo).

Cuando estamos trabajando con ramas, y sobre todo cuando queremos incorporar los cambios realizados en una rama en la rama principal, es conveniente tener actualizada la rama que estamos desarrollando con las últimas actualizaciones de la rama principal<sup>18</sup>. Esto se hace con el comando `git rebase master`.

2. Activa la rama `dev` mediante `git switch dev` o `git checkout dev`
3. Antes de hacer la actualización, y para que veas mejor lo que va a pasar haz una nueva rama a partir de `dev` con: `git branch dev_old`
4. teclea de nuevo `git log --oneline --all --graph` para ver que la nueva rama que has creado se encuentra en el mismo estado que `dev`

```
* bb74ee6 (origin/master, master) Create README.md
| * deb0b98 (HEAD -> dev, origin/dev, dev_old) Include more effiicent
  computation
|/
* 9a9ea98 Added new module archivo_2
* f0cf479 Added a second line of code
* 062ccfc Fix inital bug
* ba43a6b My first commit
```

5. Ahora actualiza `dev` al último registro de la rama `master` mediante

```
git rebase master
```

---

<sup>18</sup>Imagínate que tienes tu software ya en producción (rama principal, llamémosle version 1.0) pero ya estás trabajando con la versión 2.0 en otra rama. Mientras trabajas en la nueva versión es posible que otros usuarios o tú mismo te des cuenta de la existencia de algunos bugs y/o pequeñas mejoras que hacer en la versión 1.0, por lo que añades nuevos commits a la rama principal, teniendo así la version 1.1. Llegados a este punto te interesa que la rama sobre la que estás desarrollando la versión 2.0 incorpore también esos cambios

Que viene a indicar resitúa mi rama actual al último commit de la rama master. Vuelve a ejecutar

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

para ver la situación de cada una de las ramas tras el rebase. También mira el contenido de tu repositorio para cada una de las ramas y las diferencias entre ellas.

Ahora que ya tenemos re-actualizada nuestra rama en desarrollo con la última versión de la rama principal, podemos seguir trabajando mejorando nuestro código o incorporar definitivamente los cambios a la rama **master**. Para esto último se utiliza el comando **git merge**. **git merge <branch>** indica al sistema que coja la rama **<branch>** y la fusione en la rama actual

1. Queremos fusionar la rama **dev** en la rama **master**, por lo que primero de todo tenemos que activar la rama **master** **git switch master**
2. Ya estando en la rama que queremos actualizar mediante el fusionado, teclea **git merge dev**

Como anteriormente habíamos hecho un rebase de dev a master, el fusionado se tendría que hacer sin problemas y un mensaje similar a este debería aparecer

```
Updating bb74ee6..cdd9770
Fast-forward
 archivo_2.txt | 2 ++
 1 file changed, 2 insertions(+)
```

3. Ahora **master** tiene todos los cambios y actualizaciones hechas en **dev**, por lo que ambas ramas son idénticas y, de hecho podríamos eliminar **dev**. Tecllea **git branch -D dev** para borrar definitivamente **dev** de tu repositorio local, aunque seguirá existiendo en tu repositorio remoto. Para eliminar ramas del remoto, por ejemplo de GitHub, puedes hacerlo directamente desde la web de GitHub o mediante el comando **git push <alias\_del\_remoto> --delete <rama\_de**

Si te fijas de nuevo en el mensaje que recibiste al ejecutar **git merge dev**, La actualización/fusionado se ha realizado mediante la estrategia **Fast Forward**, que significa que simplemente git puede añadir todos los registros nuevos al final del último registro. Esto es debido a que previamente al fusionado hicimos una actualización de la rama **dev** para que incluyese todos los commits de **master** posteriores al momento en que ambas ramas se separaron. Esto a veces no siempre es posible y, sobre todo al actualizar/sincronizar los repositorios remotos, que han podido sufrir cambios desde otro ordenador o por otro usuarios.

Afortunadamente **Git** es suficientemente inteligente para decidir qué estrategia de fusionado es la mejor para cada caso (según los registros en común y las divergencias entre ramas), por lo que no te tienes que preocupar mucho de esto<sup>19</sup>. Sin embargo, puede darse el caso, y más frecuente de lo que te imaginas sobre todo en las fases de iniciación de **Git**, que haya casos en los que el sistema no pueda hacer la fusión de forma automática, y entonces es cuando tenemos nosotros que resolver el conflicto.

<sup>19</sup>Sobre todo para el nivel de programación al que aspiramos

## 3.2. Resolución de conflictos

Un conflicto es debido a que al hacer ejecutar una actualización de ramas, bien sea por **merge** o **rebase**, hay líneas del código que han sido modificadas en las dos ramas y discrepan en su contenido. Cuando esto ocurre, que puede ser bastante probable si no hacemos fusionados y actualizaciones con el remoto frecuentes, **Git** avisa que no ha podido completar el fusionado por la existencia de conflictos y nos pide que manualmente los solucionemos.

Vamos a generar voluntariamente un conflicto y ver lo que pasa y los pasos a seguir para resolverlo:

1. Partimos de la situación anterior, en la que tenemos una rama **master** con los cambios de **dev** ya incorporados.
2. Crea una nueva rama y llámala por ejemplo **futuro\_conflicto**: `branch futuro_conflicto`
3. con la rama **master** activada edita uno de los archivos de texto (por ejemplo **archivo\_1.txt**) y añade una línea con este texto `nueva linea de codigo de la rama master`
4. Registra el cambio en un nuevo commit, p.ej.

```
git add archivo_1.txt
git commit -m 'master conflicting commit'
```

5. cambia ahora a la rama **futuro\_conflicto** y edita el mismo archivo<sup>20</sup> con esta línea:

```
nueva linea de codigo en futuro_conflicto
```

6. registra el cambio en un nuevo commit, p.ej.

```
git add archivo_1.txt
git commit -m 'futuro_conflicto conflicting commit'
```

7. Echa un ojo ahora a las diferencias entre las dos ramas

```
git diff futuro_conflicto master
```

```
diff --git a/archivo_1.txt b/archivo_1.txt
index 7230ec3..16094fb 100644
--- a/archivo_1.txt
+++ b/archivo_1.txt
@@ -1,3 +1,3 @@
 mi primera linea de codigo corregida
 segunda linea de codigo
-nueva linea de codigo en futuro_conflicto
+nueva linea de codigo de la rama master
```

Verás que sólo difieren en una línea.

8. Ahora vamos a intentar fusionar la rama **futuro\_conflicto** en **master**. Activa la rama **master** y teclea

```
git merge futuro_conflicto
```

Recibirás un mensaje parecido a este:

---

<sup>20</sup>verás que al estar un commit por detrás de **master** la línea que recientemente editaste no aparece

```
Auto-merging archivo_1.txt
CONFLICT (content): Merge conflict in archivo_1.txt
Automatic merge failed; fix conflicts and then commit the result.
```

Que viene a decir que **Git** no puede por sí solo fusionar las ramas por la existencia de conflictos en el `archivo_1.txt`. Nos pide que arreglemos el conflicto y que registremos el cambio.

9. Edita de nuevo el archivo en conflicto, verás que presenta un texto similar a este:

```
mi primera linea de codigo corregida
segunda linea de codigo
<<<<<< HEAD
nueva linea de codigo de la rama master
=====
nueva linea de codigo en futuro_conflicto
>>>>>> futuro_conflicto
```

**Git** te está indicando dónde y en qué modo está el conflicto. Tenemos que resolver todo el texto que está entre `<<<<<<` y `>>>>>>`, de modo que lo que hay entre `<<<<<< HEAD` y `=====` es la parte conflictiva que pertenece al último commit (**HEAD**) de la rama actual (**master**), y lo que está entre `=====` y es la parte conflictiva que hay en la rama **futuro\_conflicto**.

10. Edita manualmente el texto de modo que tu código final se ajuste a lo que realmente querías, por ejemplo podemos pensar que la parte de **master** es la buena, o viceversa, o incluso que ninguna de las dos líneas es la correcta y reescribir esa línea.
11. Una vez termines y estés satisfecho con la resolución del conflicto tu texto no debería contener ningún carácter del tipo `<<<<<<`, `>>>>>>` o `=====`. Por ejemplo:

```
mi primera linea de codigo corregida
segunda linea de codigo
nueva linea de codigo con conflicto resuelto
```

Podemos guardar el archivo y cerrarlo.

12. Ya tenemos el archivo como nos gustaría que quedase tras la fusión, para continuar con la misma tendemos que volver a añadirlo al espacio de trabajo: `git add archivo_1.txt`
13. Opcionalmente podemos registrar el conflicto con un nuevo commit
14. Esto lo repetiríamos para cualquier otro conflicto que tengamos, una vez todos estén resueltos y añadidos al espacio de trabajo podemos continuar con la fusión mediante

```
git merge --continue
```

Si no has registrado commits durante esta fase, al ejecutar este último comando **Git** te dará la opción de editar el texto del commit, puedes dejar ese texto por defecto o modificarlo.

¡Enhorabuena! ya ha solucionado tu primer conflicto durante una fusión. Si bien los pasos a realizar durante un conflicto siempre son los mismos: editar los archivos conflictivos, añadirlos al espacio de trabajo y continuar

el fusionado mediante el keyword (`--continue`), no hay ninguna regla sobre cómo resolver los mismos<sup>21</sup>, y cada caso que se te presente será distinto y tendrás que apelar al sentido común y al objetivo que tenga la(s) línea(s) de código conflictiva(s).

## 4. Código colaborativo en GitHub

En esta última tarea vais a subir vuestros progresos a mi repositorio del curso (<https://github.com/hectornieto/cursoGit>), con el fin de poder certificar a la UAH vuestro aprovechamiento del curso.

A la hora de trabajar en un equipo o incluso sugerir cambios en el repositorio de un tercero no es recomendable hacerlo directamente en el repositorio original. El proceso típico es hacer una copia (**Fork**) en tu cuenta GitHub, hacer los cambios necesarios y hacer un **Pull Request** desde tu repositorio GitHub hacia el repositorio original.

### 4.1. Creación de copias (“Forks”)

1. Primero vamos a descargar y sincronizar el repositorio del curso a una carpeta de tu PC. Teclea

```
git clone https://github.com/hectornieto/cursoGit
```

 Un mensaje parecido a éste aparecerá

```
Cloning into 'cursoGit'...
remote: Enumerating objects: 72, done.
remote: Counting objects: 100% (72/72), done.
remote: Compressing objects: 100% (49/49), done.
remote: Total 72 (delta 29), reused 60 (delta 20), pack-reused 0
Unpacking objects: 100% (72/72), 13.05 MiB | 9.12 MiB/s, done.
```

y tendrás un repositorio local del curso en la carpeta `cursoGit`. Si tienes tiempo y ganas, puedes ver todos los commits que se han hecho para preparar el curso, desde que se inició el repositorio.

2. Teclea `git remote -v` para confirmar que el repositorio ya está ligado al repositorio remoto con alias `origin`.

```
origin https://github.com/hectornieto/cursoGit (fetch)
origin https://github.com/hectornieto/cursoGit (push)
```

3. Abre tu navegador y ve al repositorio `cursoGit` de GitHub: <https://github.com/hectornieto/cursoGit>. Asegúrate de ingresar en GitHub con tu usuario y contraseña, y pincha en el botón **Fork** de la parte superior derecha.
4. Ahora ve a tu cuenta GitHub y verás que tienes en tu GitHub una copia del repositorio del curso. Si miras a través de las distintas ramas del repositorio verás en primer lugar que he creado una rama por cada uno de vosotros, y al pinchar en cada una de ellas aparecerá un mensajito que dice **This branch is even with hectornieto:<branch>**. Normal, porque de momento acabamos de hacer la copia.

---

<sup>21</sup>si la hubiera Git ya habría hecho el fusionado automáticamente



5. Vuelve al terminal de tu PC y añade también la copia remota de tu GitHub a tu repositorio local con:

```
git remote add <alias_del_remoto> https://github.com/<usuario>/cursoGit
```

Donde `<alias_del_remoto>` podría ser cualquier nombre excepto `origin` ya que este alias ya está asignado al repositorio de mi cuenta. Un nombre común en estos casos suele ser `upstream`.

6. Confirma que tu repositorio local está ahora conectado a los dos repositorios remotos, el original al que más adelante haremos el `Pull Request` y su copia en tu propio espacio GitHub, mediante el comando `git remote -v`

```
origin  https://github.com/hectornieto/cursoGit (fetch)
origin  https://github.com/hectornieto/cursoGit (push)
upstream https://github.com/complutig/cursoGit (fetch)
upstream https://github.com/complutig/cursoGit (push)
```

La idea de tener dos remotos es usar el remoto original para tener siempre actualizado tus repositorios, tanto el local como tu copia remota. Para ello se recomienda confirmar el estado de sincronización con `git status` y en caso de que haya nuevos commits en el repositorio original traerlos a tus copias con `git pull origin <branch>`. Luego subir los cambios que hagas desde tu repositorio local a tu repositorio remoto, con el fin último de hacer el `Pull Request` desde tu remoto al remoto original.

## 4.2. Sugerencias de cambios (“Pull Request”)

Ya tenemos nuestra copia local y remoto. Ahora lo que vamos a hacer para terminar el curso es añadir todos los archivos que has creado en la práctica anterior al repositorio del curso. Para ello vas a trabajar con la rama asociada a tu nombre de usuario GitHub.

1. Para asegurarnos que tenemos los últimos registros del repositorio original teclea `git fetch --all`
2. Como hay una rama remota con tu nombre de usuario GitHub teclea `git checkout <usuario>` para que el sistema construya esa rama en tu repositorio local.
3. Crea una carpeta vacía llamada `ejercicio_<usuario>` donde sustituye `<usuario>` por tu nombre de usuario GitHub.
4. Copia todo el repositorio de `primeros_pasos` a esta carpeta
5. Añade esa carpeta y todo su contenido al sistema Git y registra el cambio con un commit

```
git add ejercicio_<usuario>
git commit -m <mensaje>
```

6. Sube los cambios a tu repositorio remoto con:

```
git push <alias_del_remoto> <branch>
```

donde `<alias_del_remoto>` es el alias de tu remoto (p.ej. `upstream`) y `<branch>` es el nombre de la rama que estás subiendo (que en este caso es el nombre de tu usuario GitHub). SI todo va bien un mensaje similar a este debería aparecer

```
Username for 'https://github.com': hectornieto
Password for 'https://hectornieto@github.com':
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Delta compression using up to 4 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (2/2), 237 bytes | 237.00 KiB/s, done.
Total 2 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To https://github.com/complutig/cursoGit
5b0e2c6..f98d500 test -> test
```

7. Ahora que has subido los cambios a tu repositorio remoto puedes ir a tu cuenta GitHub, navegar al repositorio `cursoGit` y activar la rama con tu nombre. Debería aparecer una pantalla con un mensaje similar a este:

test had recent pushes 5 minutes ago [Compare & pull request](#)

test 3 branches 0 tags [Go to file](#) [Add file](#) [Code](#)

This branch is 1 commit ahead of hectornieto:test. [Contribute](#) [Fetch upstream](#)

Commit	Message	Time
f98d500	Rename folder	10 minutes ago
	Include latex figures folder	yesterday
	Add supplementary readings folder	7 days ago
	Additional latex ignore files	yesterday
	Initial commit	7 days ago
	Fix GitHub sign-up URL	2 days ago
	Add list of commands to discuss	yesterday
	Add preliminary questionnaire file	2 days ago
	Update Course program	2 days ago
	Add Tutorial LaTeX file	7 days ago

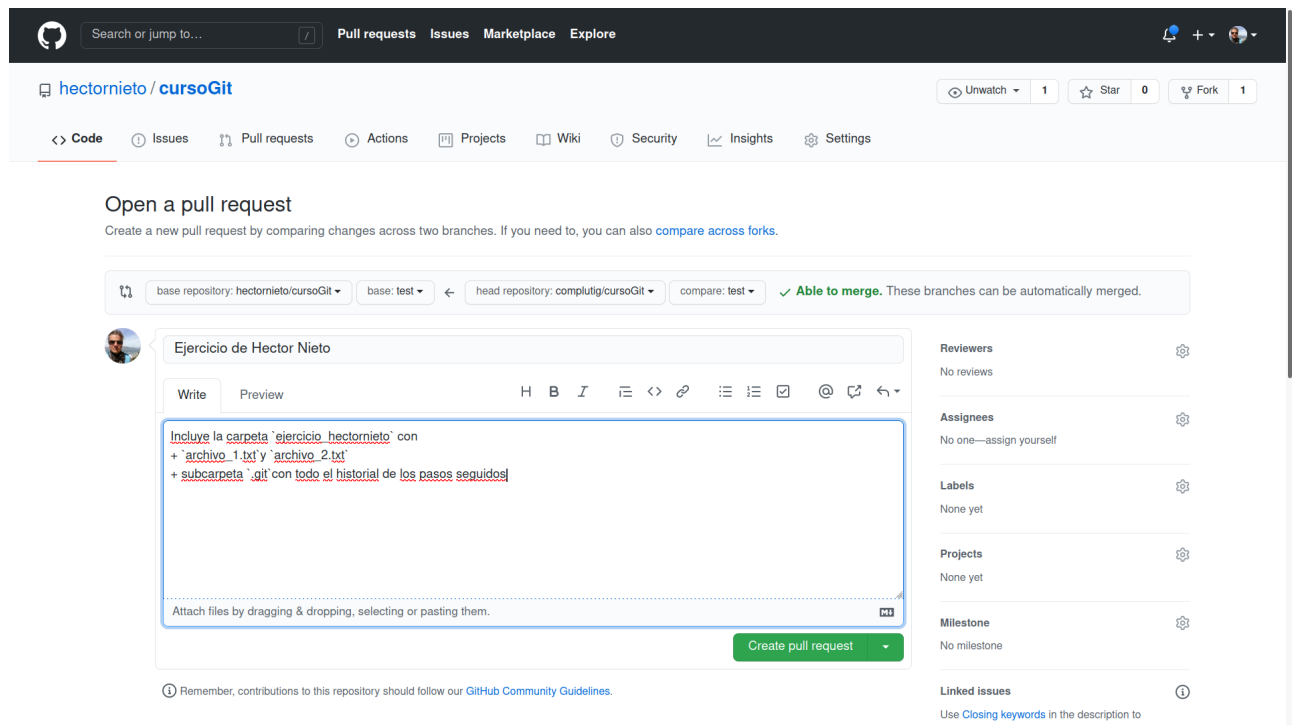
README.md

## Curso Git

### Introducción

Como ves ya GitHub detecta que ha habido cambios entre la rama que tienes en tu repositorio remoto y la misma rama del repositorio original y te ofrece la posibilidad de comparar ambas ramas y hacer un pull request

8. Pincha en el icono verde de **Compare & Pull Request**. Aparecerá una pantalla como esta



En la que podrías modificar el repositorio base al que iría destinado el **Pull Request** y las ramas de origen y destino del mismo. En principio conviene revisar esto y en su caso modificar de acuerdo a lo que uno realmente quiera hacer. En el caso que nos atañe simplemente añade el nombre del **Pull Request**, que generalmente tiene que resumir en una línea el por qué de la sugerencia del cambio (p.ej. **Ejercicio de <tu\_nombre>**) y luego se suele añadir más detalles de los cambios en el cuadro de texto libre.

9. Una vez completes el nombre y la descripción pincha en **Create pull request** para finalizar con el proceso.

¡Enhorabuena! Has terminado con éxito este tutorial. Ahora te recomiendo que continúes practicando y empieces a crear repositorios con tu código, como ves **Git** es muy seguro y no deberías temer equivocarte porque siempre puedes volver a un estado previo.

Una vez que haces un **Pull Request** a un repositorio de **GitHub**, su dueño recibirá un e-mail con la notificación que tiene pendiente aprobar, comentar o rechazar la sugerencia. Si lo aprueba, el repositorio original se actualiza automáticamente con los cambios que sugeriste.

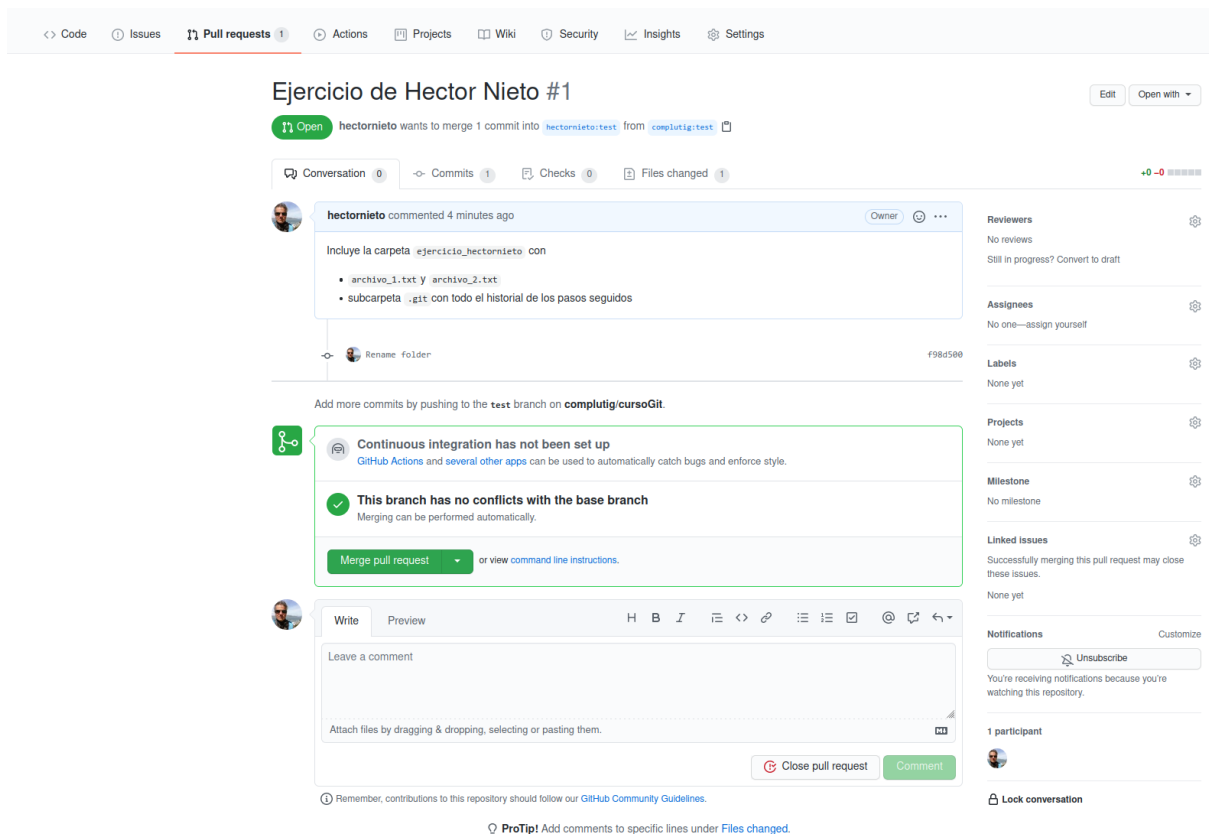


Figura 1: Ejemplo de Pull Request pendiente por aprobar por el dueño del repositorio

## 5. Descripción de los comandos de Git

### 5.1. git init

Inicializa un repositorio git dentro de la carpeta actual. Al ejecutar `git init` Git genera automáticamente una subcarpeta `.git` con todos los archivos del sistema de git. Puedes borrar un repositorio git simplemente borrando esta carpeta.

### 5.2. git add

Añade archivos o carpetas para ser registrados con `git comit`. `git add` admite añadir de una sólo vez una serie de arhivos mediante su inclusión separados por espacios `git add <archivo_1> <archivo_2> <...>`. `git add` también admite wildcards para añadir todos los archvos presentes en una carpeta que cumplan con un patrón en su nombre. Por ejemplo `git add *.py` añadiría todos los archivos dentro de la carpeta con la extensión `.py`.

### 5.3. git commit

Confirma los cambios realizados tras `git add` en una nueva versión (o commit). La forma más frecuente de utilizar este comando es con

```
git commit -m '<descripción_del_commit>'
```

donde es recomendable que `<descripción_del_commit>` sea un mensaje que identifique de manera concisa y clara los cambios realizados tras el último commit<sup>22</sup>.

El otro modo típico de utilizar este comando es con `git commit --amend`, que permite re-editar o actualizar el último commit realizado.

### 5.4. git diff

`git diff` es el comando utilizado para visualizar diferencias entre versiones y archivos<sup>23</sup>. Si tecleas `git diff` a secas en el terminal el sistema devuelve todas las modificaciones y diferencias que haya entre los archivos en su estado actual y la última versión registrada con `commit`.

Por otro lado, tecleando `git diff <archivo>` Git permite visualizar las diferencias para un archivo en cuestión entre su versión actual y la última versión registrada con `commit`.

`git diff <commit_1> <commit_2>` por otro lado te permite ver las diferencias entre dos versiones registradas

También puedes usar

```
git diff <commit_1> <commit_2> -- <archivo>
```

para ver las diferencias de un archivo entre dos versiones, o incluso

```
git diff <branch_1> <branch_2> -- <archivo> .
```

para ver las diferencias de ese archivo entre dos ramas distintas.

Finalmente también puedes comparar el estado de tus archivos entre tu repositorio local y remoto:

```
git diff <alias_del_remoto>/<branch_1> <branch_2> -- <archivo> .
```

Como ves las posibilidades y combinaciones para evaluar distintas versiones del código a lo largo del tiempo y en distintas ramas es infinita.

### 5.5. git checkout

Este comando es muy versátil y permite realizar distintos tipos de operaciones según el contexto en el que se aplique

- `git checkout <commit>` o `git checkout HEAD~<número>` permite restaurar una versión previa del repositorio
- `git checkout <branch>` permite ir de una rama a otra.
- `git checkout <archivo>` permite restaurar un archivo a su estado original, definido por el último commit.

<sup>22</sup>Una buena práctica en programación y en Git es realizar commits lo más específicos posible, mejor muchos y cortos que un gran commit con muchos cambios dentro de los archivos

<sup>23</sup>Algo similar a la herramienta de control de cambios de Word

## 5.6. git branch

Se trata de operaciones para trabajar con ramificaciones. Estos son las operaciones y variantes del comando más típicas:

- `git branch` proporciona un listado de ramificaciones existentes y señala la ramificación actual.
- `git branch <branch>` crea una nueva rama con nombre `<branch>` como réplica de la ramificación en la que actualmente te encuentres.
- `git branch -D <branch>` borra la rama de nombre `<branch>`. Para poder borrar una ramificación tienes que hacerlo desde otra rama distinta.

## 5.7. git merge

## 5.8. git rebase