

Aprendiendo Git



Miguel Ángel Durán

Aprendiendo Git

¡Domina y comprende Git de una vez por todas!

Miguel Angel Durán García

Este libro está a la venta en <http://leanpub.com/aprendiendo-git>

Esta versión se publicó en 03/01/2023



* * * * *

Éste es un libro de [Leanpub](#). Leanpub anima a los autores y publicadoras con el proceso de publicación. [Lean Publishing](#) es el acto de publicar un libro en progreso usando herramientas sencillas y muchas iteraciones para obtener retroalimentación del lector hasta conseguir el libro adecuado.

* * * * *

© 2021 - 2023 Miguel Angel Durán García

Tabla de contenido

Introducción

Prólogo

Sobre mí

Requisitos previos

Únete a la comunidad de Discord

Reportando erratas y sugerencias

Agradecimientos

Cambios entre versiones del libro

Un poco de teoría

¿Qué es un control de versiones y por qué es importante?

¿Qué es Git?

Los fundamentos de Git

Los tres estados en Git

¿Qué es una rama?

¿Qué representa la rama `master` o la rama `main`?

¿Git y GitHub son lo mismo? ¿Qué relación tienen?

Instalando y Configurando Git

¿Cómo saber si tengo Git instalado en mi sistema?

¿Cómo instalar Git?

¿Cómo configuro el nombre y correo usado en Git?

¿Cómo configurar el editor por defecto que abre Git?

¿Cómo puedo comprobar mi configuración de Git?

Git y la línea de comandos

Trabajando con Git de forma local

¿Cómo inicializar un nuevo proyecto Git?

¿Qué es el directorio de trabajo?

¿Cómo deshacer un archivo modificado?

¿Cómo añadimos archivos al área de staging?

¿Cómo puedo sacar un archivo o varios del área de staging?

[¿Qué es un commit?](#)

[¿Cómo puedo hacer un commit?](#)

[Haz commit y sáltate el área de staging](#)

[¿Qué es el HEAD?](#)

[¿Cómo puedo deshacer mis cambios?](#)

[¿Cómo puedo ignorar archivos?](#)

[¿Cómo consigo ignorar siempre los mismos archivos en todos mis repositorios?](#)

[¿Cómo le indico a Git que deje de hacer el seguimiento de un archivo \(o varios archivos\)?](#)

Ramas en Git

[Empezamos con las ramas](#)

[Trabajando con ramas](#)

[Fusionando ramas](#)

[Resolviendo conflictos](#)

[Hora de podar: Eliminando ramas](#)

Rebase

[¿Qué es el rebase?](#)

[No uses git rebase para esto](#)

[Los peligros del rebase](#)

Merge vs Rebase

Trabajando con Git de forma remota

[Creando un repositorio remoto en GitHub](#)

[Clonando un repositorio remoto ya creado previamente](#)

[¿Cómo enlazar un repositorio local con un repositorio remoto?](#)

[Traer los cambios del repositorio remoto a mi repositorio local](#)

[Escribiendo en el repositorio remoto](#)

[Trabajando con ramas en remoto](#)

Configurando la conexión SSH con GitHub

[Cómo generar una llave SSH](#)

[Usar una llave SSH](#)

[Añadir clave SSH a tu cuenta de GitHub](#)

[Probando, probando...](#)

Cómo contribuir a un proyecto de código abierto

¿Por qué debería contribuir al código abierto?

¿Cómo empiezo a contribuir a un proyecto de código abierto?

¿Cómo hago una Pull Request al proyecto original?

¿Cómo puedo sincronizar mi fork con el repositorio original?

¿Con qué puedo contribuir a un proyecto?

Flujos de trabajo y estrategias de ramas en Git

Git Flow

GitHub Flow

Trunk Based Development

Ship / Show / Ask

Conclusiones sobre las estrategias de flujos de trabajo en Git

Buenas prácticas al trabajar con Git

¿Cada cuánto debería hacer un commit?

¿Cómo puedo escribir un buen mensaje de commit?

¿Cómo puedo escribir un buen nombre de rama?

¿Debería alterar el historial de mi proyecto?

No hagas commit de código generado ni configuración particular

¿Qué debo tener en cuenta para hacer una buena Pull Request?

Cambia tu rama `master` a `main` o similares

Firma correctamente tus commit con GPG

¿Cómo debería revisar una Pull Request?

Hooks de Git

¿Qué es un hook?

¿Qué hooks hay disponibles?

¿Cómo puedo crear mi propio hook?

Alias en Git

¿Cómo crear tu propio alias en Git?

¿Puedo crear alias para comandos que ya existen?

¿Puedo crear un alias para un comando externo?

¿Cómo puedo listar todos los alias que he creado?

¿Cuáles son los mejores alias?

Stash, el almacén temporal de cambios

Guarda tus cambios en un stash

Aplicando los cambios del stash

Crea una rama a partir de un stash

Eliminando el almacén temporal

Trucos con Git

¿Cómo puedo aplicar los cambios de un commit a otro?

Cómo detectar qué commit es el que ha introducido un bug

¿Quién ha tocado este fichero? ¿Quién ha hecho cambios?

¿Cómo puedo saber quién añadió una línea por primera vez?

Recupera un archivo en concreto de otra rama o commit

Encuentra el primer commit de un repositorio

Descubre el máximo contribuidor de un repositorio

Recupera todos los commits para un usuario en específico

Clonar un repositorio sin descargar todo el histórico

Vuelve a la rama previa en la que estaba trabajando

Descargar los ficheros de un repositorio remoto sin tener que clonarlo

Aprovecha el auto-corrección de Git para que ejecute comandos parecidos

Domina el formato corto de `git status`

`--porcelain`, la opción para que nuestros scripts usen comandos de Git

Configuraciones a tener en cuenta

Errores comunes en Git y sus soluciones

Me dice que no es un repositorio de git

Hago `pull` y me dice que no es un repositorio

He escrito mal el último commit

He escrito mal la rama que he creado

He hecho un `git push` y me da error

He hecho commits a la rama principal que debían realizarse en otra rama

`xcrun: error: invalid active developer path`

GitHub CLI

Instalando `gh` en el sistema...

Realizando la configuración inicial de gh
Usando gh

Conclusiones del libro

Introducción

Prólogo

Hoy en día es imposible imaginar el desarrollo de software sin Git. [Según la encuesta de Stack Overflow de 2018](#), casi **el 90% de los desarrolladores usaban Git para manejar su código fuente**. Si te preguntas por qué no hay encuestas más recientes... es simplemente porque no preguntaron más. Su dominio empezaba a ser tan evidente que no dejaba margen a la curiosidad.

Sin embargo, cuando empecé la universidad, no existía Git. Recuerdo que lo primero que usé para tener versiones de mi código eran... ¡disquetes! Ah, suena tan arcaico y me hace sentir tan viejo.

Recuerdo confiar que el garabato que ponía en la etiqueta me sirviera para determinar en el futuro qué versión era. Muy ingenuo, si me preguntan ahora, porque no recuerdo haber sido capaz de rescatar nunca nada reseñable entre las decenas de disquetes que guardaba con exquisito celo.

Más tarde empecé a usar [Subversion](#) y aquello me pareció tan mágico que no me terminaba de fiar. ¿De verdad esto podía guardar todas las revisiones de mi código? ¿Y si yo quisiera cambiar una parte del código? ¿Cómo lo haría? ¿Y si otro compañero tocaba el fichero que yo estaba modificando? Sí, ser ingenuo es parte del camino del crecimiento de un desarrollador.

Recuerdo pocas cosas de la universidad. Pero no podría olvidarme de mi querido [TortoiseSVN](#), un cliente de *Subversion* que tenía una interfaz muy amigable e intuitiva. Y tenía una tortuga como logo. Una maravilla de la ingeniería para la época.

Subversion me parecía insuperable. **¿Qué nueva tecnología, digna de ciencia ficción, podría surgir para superar semejante proeza?** Pues ahí tuve otro aprendizaje: la supremacía de una herramienta, biblioteca o técnica puede parecer perenne pero, en realidad, puede ser fugaz.

Y es que apenas unos años después apareció Git, que a día de hoy es sin ninguna duda el sistema de control de versiones más usado en el mundo.

Desde su creación ha ido cautivando diligentemente al mundo del desarrollo hasta convertirse en el estándar de facto.

Siendo tan importante entender Git, su manejo y dominio, he decidido escribir este libro. Un libro que yo mismo hubiera estado encantado de leer cuando empecé. Un libro del que estoy seguro que podrás extraer algún aprendizaje y que te podrá servir de guía en el futuro.

Sobre mí

¡Hola, soy Miguel Ángel Durán! Cuento con 15 años de experiencia en el mundo del desarrollo. Actualmente trabajo como Enabler Frontend en *Adevinta Spain* y soy divulgador de contenido sobre tecnologías web. Estoy reconocido como **GitHub Star** y como **Google Developer Expert en Web Technologies**.

En [YouTube](#) tengo diferentes cursos totalmente gratuitos para aprender JavaScript, React, React Native, Next.js, Svelte y GraphQL, entre otros contenidos.

También en [Twitch](#) hago streamings sobre desarrollo con JavaScript en general. ¡Y nos lo pasamos genial! Te invito a pasarte. Puedes [ir al Calendario](#) para ver cuándo estoy en directo y qué programas tengo planeados.

Puedes encontrarme en Twitter como [@midudev](#) y en Instagram como [@midu.dev](#). Si te estás preguntando porqué en Instagram no soy también @midudev es porque hay otra persona con esa cuenta y sólo me la cedería a cambio de 1000\$...

Por último te dejo [mi página web](#) donde puedes encontrar artículos, cursos, tutoriales y mucho más.

Requisitos previos

El libro está escrito partiendo de la base que **no conoces nada sobre Git** pero da por sentados ciertos conocimientos mínimos de terminal. Por ejemplo, se da por hecho que eres capaz de moverte entre diferentes carpetas de tu sistema de ficheros, que eres capaz de crear y borrar ficheros y directorios, etc.

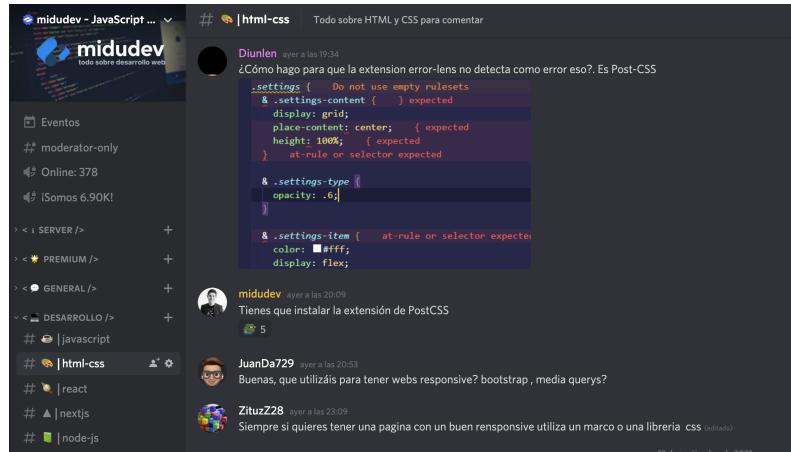
No es necesario que tengas conocimientos de programación en general ni de un lenguaje en particular, pero sí que deberías ser capaz de entender el concepto de código fuente. Igualmente, si estás aprendiendo Git seguramente es porque ya tienes una aplicación y quieres empezar a usarlo para controlar y versionar mejor los cambios que realizas.

Ten en cuenta que aunque lo aprendido en el libro sirve para diferentes sistemas operativos, se da por hecho que los comandos que ejecutamos en la terminal son en un sistema Unix. Esto quiere decir que **en Windows deberías estar usando la consola Bash de Ubuntu**.

Por último, indicarte que los comandos en la terminal vienen precedidos de un \$. Este símbolo **no es necesario que lo escribas en tu terminal** pero es una convención que te ayuda a saber que estás ejecutando un comando en la terminal (simula el *prompt* de la consola).

Únete a la comunidad de Discord

Hablando, compartiendo y ayudando a otras personas se aprende mucho. Por eso te invito a que te unas a [mi comunidad oficial de Discord](#). Allí podrás conocer a personas interesadas en el mundo de la programación que comparten recursos de aprendizaje y otras cosas interesantes.



En la comunidad de Discord encontrarás gente con tus mismos intereses dispuesta a ayudarte y compartir conocimiento. ¡No te la pierdas!

Reportando erratas y sugerencias

Creo que **los libros los termina de escribir quien los lee**. Por eso, he habilitado un repositorio de GitHub donde puedes abrir *issues* para avisar de errores que existan en el libro, ya sean ortográficos, técnicos o de maquetación. Intentaré solucionarlos lo antes posible y subir una versión actualizada con tus observaciones y correcciones.

Antes de reportar un problema, **te recomiendo que revises la edición a la que te refieres**. En la segunda página del libro encontrarás cuándo se publicó la versión. Por favor, **usa esa referencia a la hora de crear la issue** para que pueda revisar más fácilmente a qué versión te refieres.

Tienes [el repositorio disponible](#) para colaborar. **¡Cuento contigo para crear el mejor libro posible!** ¡Gracias!

The screenshot shows a GitHub repository named 'midudev / libro-aprendiendo-git-issues' with a public status. The 'Issues' tab is selected, showing 8 open issues and 6 closed issues. The open issues listed are:

- Error con los emoticonos? [#14](#) opened 4 hours ago by toadko
- Asistente de instalación de Windows [#13](#) opened 2 days ago by montycit
- Frase incompleta en "Clonando un repositorio remoto ya creado previamente" [#12](#) opened 3 days ago by pasagedev
- git checkout "restaurar todo el directorio" (pag 35) [#11](#) opened 5 days ago by KevinCamos

En la sección de Issues del repositorio encontrarás una forma de reportar las erratas y problemas que te encuentres en el libro. ¡Ya hay gente que lo ha hecho!

Agradecimientos

Escribir un libro, por pequeño que sea, requiere mucho tiempo y dedicación. Desde luego, mucho más de lo que en un principio pensé.

A esa versión mía del pasado, tan optimista, le dejo esta pequeña nota para explicarle que **hubiera sido completamente imposible sacar este conjunto de artículos sin la ayuda de muchas personas.**

En primer lugar a *Daniela Aguilera*, mi pareja de tantas cosas, por apoyarme en cada paso y recordarme siempre en qué tengo que enfocarme. Un apoyo constante.

También quiero agradecerle a *Daniel de la Cruz*, que ha prestado sus ojos, su experiencia y su tiempo para revisar que este libro sea la mejor versión posible.

Sin mis **mecenas de Patreon** tampoco hubiera sido posible aventurarme a juntar tantas letras. Les quiero agradecer especialmente a *Sergio David Serrano, Camila Maya, Juan Díaz, Facundo Capua, Imanol Decima, Ivan L'olivier, Marco Casula, Sergio Martínez y Vicente Albert* por su apoyo superlativo.

A mis **suscriptores de Twitch** y las personas que me siguen en las diferentes redes sociales. A ellas y todas esas personas que me apoyan... ¡Gracias!

Además, dejo aquí una lista de contribuidores que han aportado su grano de arena para hacer mejor el libro: *tictools, Markweell, DianaIT, antoniogiroz, FredDlt, mamjerez, pasagedev, rodrigo2604, davibern, imollm, montyclt, loadko, KevinCamos, enredadodev, AlexMA2, Mateoac12, carlosala, renansalazar, juansemprun, jonathino2590, JuaniSilva, MaximoFN, alejss, loadko, Renato6GS, palcaraz, FJFrau, moslisnas, twentyfivedots y emigimenezj*.

Cambios entre versiones del libro

Aquí tienes las últimas actualizaciones que ha recibido el libro.

1.7.0 (diciembre 2022)

- Añadida sección de rebase.
- Añadida sección de rebase vs. merge.
- Completada sección de trabajar con repositorios remotos.
- Añadidas algunas configuraciones globales interesantes para el CLI git.
- Arreglados más de 50 typos gracias al a comunidad.

1.6.0 (24 de abril de 2022)

- Completada la sección de *Contribuir a proyectos de código abierto*.
- Añadida nueva sección de *Errores comunes*.
- Avanzada la sección de *Trabajar en Git de forma remota*.
- 30 páginas más de contenido en general.
- Arreglados varios fallos de sintaxis y ortografía.

1.5.0 (7 de enero de 2022)

- Fusionando ramas
- Lidiando con conflictos al fusionar ramas
- Nueva buena práctica: Firma commits con GPG.
- Arreglados varios fallos de sintaxis y ortografía.
- Añadida imagen para git blame.

1.4.0 (8 de noviembre de 2021)

- Más contenido en el capítulo de *Ramas*
- Añadido un capítulo entero para hablar de *Stash*.
- Añadido en Trucos el formato `porcelain`.
- Hablar de `git status -s` en ¿Qué es el directorio de trabajo?
- Añadir en Trucos más sobre `git status -s`
- Añadido `git blame` en Trucos.

1.3.0 (20 de septiembre de 2021)

- Nueva estrategia *Ship/Show/Ask* en *Flujos de trabajo y estrategias de ramas en Git*
- Completada sección *Clonando un repositorio remoto ya creado previamente*
- Añadida *Haz commit y sáltate el área de staging* en *Trucos*
- Arreglados y añadidos algunos títulos que faltaban
- Todas las imágenes han sido optimizadas
- Añadido cómo saltarse los hooks de pre-commit y commit-msg
- Mejorado el truco de cómo volver a una rama anterior
- Primer contenido en la nueva sección de *Ramas*
- Algunos arreglos de ortografía

1.2.0 (12 de septiembre de 2021)

- Añadida *Trunk Based Development* en *Flujos de trabajo y estrategias de ramas en Git*
- Completar sección sobre *GitHub CLI*
- Añadir más imágenes
- Explicar qué es un commit
- Añadir buenas prácticas sobre cómo revisar una Pull Request

1.1.0 (6 de septiembre de 2021)

- Añadida sección de Estrategias de Flujo de Trabajo con Git
- Añadida sección para configurar SSH

1.0.0 (14 de agosto de 2021)

- ¡Primera versión de pre-lanzamiento!

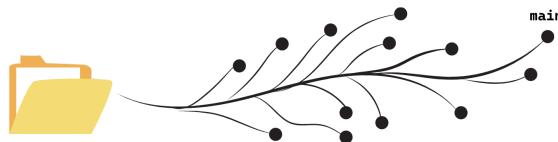
Un poco de teoría

¿Qué es un control de versiones y por qué es importante?

Un control de versiones es un sistema que registra cada cambio que se realiza en el código fuente de un proyecto. Esto te permite tener un histórico de todos los cambios producidos en sus ficheros, saber quién lo hizo y cuándo.

Podría entenderse como un sistema para crear copias de seguridad de nuestro código, y en gran parte así es, pero su uso también se extiende a cómo lidiamos con las actualizaciones de ese código y lo sincronizamos entre diferentes personas que trabajan y revisan el mismo proyecto.

Y es que, al hablar de un control de versiones, no hay que pensar en estos cambios como una simple línea del tiempo en el que cada cambio se registra. Lo mejor es pensar en un sistema de carreteras y autopistas en el que podemos crear ramas que generan bifurcaciones y que, de ser requerido, vuelven a fusionarse al ramal principal.



Normalmente el tronco del árbol se corresponde con la rama `master` o `main` y las ramas que se crean en ella serían las ramas de desarrollo o `branches`. Esto, sin embargo, es sólo una estrategia de las muchas que existen a la hora de trabajar con Git y similares.

La importancia de usar un sistema de control de versiones es que no sólo genera una copia de seguridad de los cambios. También permite que diferentes personas puedan colaborar en el desarrollo de una misma

aplicación, facilitando la sincronización y versionado de sus cambios, así como la revisión de estos por otras personas.

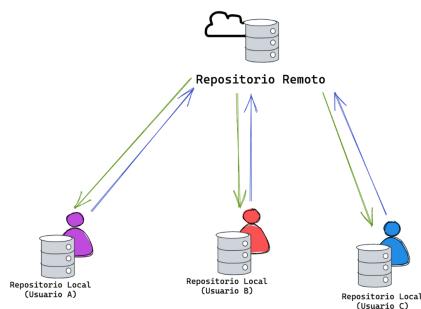
¿Qué es Git?

Git es un sistema distribuido de control de versiones, gratuito y de código abierto bajo licencia GPLv2. Fue [diseñado originalmente por Linus Torvalds](#), creador del *kernel* de Linux.



El logo de Git representa la posibilidad de crear bifurcaciones de tu código

Git fue concebido con la idea de ofrecer un gran rendimiento, ocupar menos espacio en disco y evitar la necesidad de un servidor central para que pudiera ser distribuido. De hecho, esto último es lo que hacía que fuese realmente novedoso respecto a otras alternativas del momento como Subversion, que tenían una solución más centralizada.



Git, al ser un sistema distribuido, aloja una copia completa del repositorio en cada máquina local que está trabajando en el código. Además, puedes tener uno o varios repositorios remotos para sincronizarlos

Con todas esas ventajas, Git se convirtió en un sistema de confianza a la hora de mantener versiones de aplicaciones con una gran base de código. De esta forma, en 2005 se empezó a desarrollar para ser utilizado con el núcleo de Linux, un reto nada desdeñable. De ahí hasta el día de hoy, Git ha seguido evolucionando y se ha establecido como el sistema de control de versiones más usado en el mundo de la programación y el desarrollo de software.

Los fundamentos de Git

El pilar de Git son los repositorios. Un repositorio es una carpeta en la que se almacenan las diferentes versiones de los ficheros de un proyecto y el histórico de los cambios que se han realizado en ellos.

Los repositorios pueden ser locales (los tenemos en nuestro ordenador) o **remotos** (ubicados en un servidor externo). Estos últimos son los que permitirán que otras personas puedan hacer cambios en el proyecto, que sean visibles y que puedan ser sincronizados por otros usuarios.

Cada repositorio tiene, al menos, una rama principal. Normalmente a esta rama se le llama `main` o `master`. Puedes imaginar que esta rama principal es un tronco de un árbol desde el que podemos crear ramas para generar bifurcaciones de nuestro código.

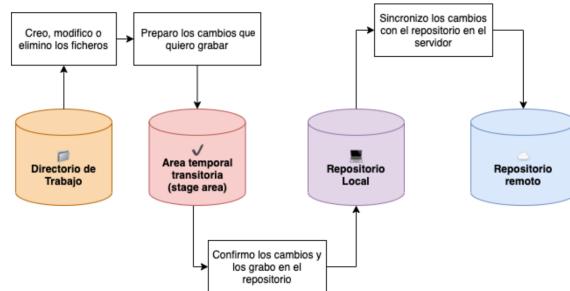
Al modificar los ficheros, los cambios se confirman y se guardan en una rama contenida en el repositorio, siendo esta rama el tronco principal o una de las bifurcaciones generadas. Si es una bifurcación es posible que esos cambios se integren en la rama principal (o en cualquier otra bifurcación) con un proceso llamado *merge*, que consiste en combinar los cambios de una bifurcación con otra rama.

Una vez se confirma ese proceso de combinación, podemos llevar los cambios al repositorio remoto. Esto se conoce como *push*. Puedes interpretarlo literalmente como la acción de empujar nuestros cambios locales al servidor. De manera análoga, la acción inversa (traer los cambios del repositorio remoto al local) se llama *pull*.

Los tres estados en Git

Al usar Git, los archivos de tu proyecto se pueden encontrar en uno de los siguientes estados:

- **modificado** (*modified*): El archivo contiene cambios pero todavía no han sido marcados para ser confirmados. Se encuentra en el directorio de trabajo.
- **preparado** (*staged*): Son los archivos que han sido modificados en el directorio de trabajo y se han marcado como preparados para ser confirmados en el repositorio local. Se encuentran en un área temporal transitoria. Esta acción recibe el nombre de *add*.
- **confirmado** (*committed*): El archivo se encuentra grabado en el repositorio local. Esta acción recibe el nombre de *commit*.



Aquí se puede ver el viaje que hacen los ficheros para pasar entre los diferentes estados y las acciones que lo provocan

Estos tres estados representan el ciclo de vida de los archivos en el repositorio.

Imagina un repositorio totalmente vacío. Lo primero que haríamos sería crear un fichero (*modificado*), entonces lo moveríamos al estado de *preparado* y, de ahí, lo grabaríamos en el repositorio y lo dejaríamos como *confirmado*.

Una vez confirmado todos los cambios en nuestro repositorio local hay que empujarlos (*push*) al repositorio remoto de forma que otra persona pueda

hacer *pull*. De esta forma podrá obtener esos cambios para sincronizarlos con su repositorio local y seguir trabajando de la misma forma.

Más adelante veremos qué comandos debemos utilizar para mover nuestros ficheros entre estos tres estados, además de la sincronización con repositorios remotos.

Ten en cuenta que en Git también puedes tener archivos ignorados que, pese a estar disponibles en el directorio de tu proyecto, en realidad son invisibles para Git. Estos ficheros quedan fuera del flujo de control y no son afectados por los estados de modificación, preparación y confirmación.

¿Qué es una rama?

Una rama es simplemente una versión de la colección de directorios y archivos del repositorio. Cada vez que se crea una nueva rama, se crea una copia de la colección de archivos actual.

A su vez, a partir de esta rama puedes crear más ramas. A veces puedes hacer que estas ramas y los cambios que hayas podido realizar en sus ficheros o directorios sean integrados en la rama principal. A esta acción, la de integrar una rama en otra, se le llama *merge* o fusionar.

¿Para qué sirven las ramas?

En un entorno de colaboración, donde diferentes personas están trabajando en un mismo código, se genera una evolución del código en paralelo. Mientras alguien está trabajando en añadir una nueva característica al proyecto, otra persona puede estar arreglando un *bug* y otra en añadir alguna documentación.

De esta forma, partiendo de un mismo código, se generan diferentes ramas. Esto sirve para aislar el trabajo de cada persona y que, una vez concluido, se pueda integrar en el tronco de nuestro repositorio que será, dicho de otro modo, la rama principal.

¿Qué representa la rama `master` o la rama `main`?

La rama `master` es la rama principal de un repositorio y normalmente se crea al iniciar un repositorio. El hecho que se llame `master` no es obligatorio y en realidad responde a una razón histórica. Simplemente siempre se le ha llamado así.

En la actualidad servicios como GitHub o GitLab recomiendan que la rama principal sea llamada `main` para [evitar connotaciones racistas](#). Ciertamente esta aproximación empieza a ser común en el uso de Git.

Sin embargo, otra gente puede llamar a esta rama de otra forma (como `trunk`) dependiendo de la estrategia que use a la hora de trabajar en equipo con sus repositorios. Al final, lo importante, es que **un repositorio necesita tener una rama principal independientemente de su nombre**.

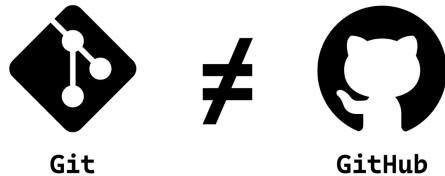
En la sección de *Buenas prácticas* del libro tienes una explicación más detallada de por qué ya no se llama `master` a la rama principal y cómo puedes migrar tu repositorio para que use `main` como nombre.

¿Git y GitHub son lo mismo? ¿Qué relación tienen?

Git es el sistema de control de versiones. Con ello podríamos comenzar a trabajar fácilmente en un proyecto de forma local. También, podríamos crear nuestro propio servidor de forma que cualquier persona pudiera desarrollar de forma remota.

Sin embargo, el mantenimiento que un servidor de este tipo requiere puede ser muy costoso y, en raras ocasiones, puede valer la pena.

Ahí es donde entra *GitHub*. Github es **un servicio de alojamiento en la nube de código fuente** basado en el sistema de control de versiones que Git ofrece para manejar repositorios. Añade además otras funcionalidades muy útiles para el desarrollo, como una UI muy amigable, GitHub Actions, Webhooks o Dependabot.



Git y GitHub están relacionados pero son dos cosas diferentes.

Hay que tener en cuenta que *GitHub* no es el único servicio de este estilo. **Existen otras alternativas como *GitLab* o *BitBucket*** que permiten también hospedar códigos fuentes en la nube versionados con Git.

Instalando y Configurando Git

¿Cómo saber si tengo Git instalado en mi sistema?

Normalmente `git` está instalado por defecto en muchos sistemas operativos, especialmente aquellos basados en `Unix`. Ejecuta en tu terminal el siguiente comando:

```
$ git --version  
git version 2.30.1 (Apple Git-130)
```

Si el resultado te indica la versión disponible en tu sistema, significa que funciona. Si, al contrario, te aparece un mensaje como `command not found: git` es que no tienes Git instalado en tu sistema o que está instalado, pero falta realizar alguna configuración. Por ejemplo, hacer que esté disponible en el PATH.

Todavía muchos sistemas y usuarios tiene instalada una versión antigua de `git`, basada en la versión 1. En el momento de escribir estas líneas, la última versión disponible es la 2.33.0. Si tienes una versión muy anterior, considera actualizarla.

¿Cómo instalar Git?

En **macOS** lo más sencillo sería instalar las Herramientas de Desarrollo de Xcode con el siguiente comando:

```
$ xcode-select --install
```

Alternativamente, puedes usar `brew` para instalarlo directamente:

```
$ brew install git
```

`brew`, o Homebrew, es un gestor de paquetes para macOS, que te permite instalar todo aquello que necesitas que Apple no instala de serie. Si no lo conocías y quieres probarlo, puedes ver cómo se instala en <https://brew.sh>.

Para sistemas **Linux**, puedes instalarlo desde el repositorio de software de tu distribución. Por ejemplo, para Ubuntu y Debian puedes instalarlo con el siguiente comando:

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

Para Fedora, sería así:

```
$ yum install git # Fedora 21 y anteriores  
$ dnf install git # Fedora >22
```

Para otras distribuciones, revisa la [página de descargas de Git](#).

En **Windows** es más sencillo descargar el instalador oficial y seguir los pasos del asistente: <https://git-scm.com/download/win>.

¿Cómo configuro el nombre y correo usado en Git?

Antes de comenzar a usar Git, es recomendable que hagas una mínima configuración.

Para hacer que tus commits se asocien a tu nombre y aparezca correctamente tu avatar en plataformas como *GitHub*, necesitas realizar la siguiente configuración:

Ejecuta en la terminal estos comandos cambiando el <tu nombre> y <tu email> por tus datos reales.

```
$ git config --global user.name "<tu nombre>"  
$ git config --global user.email "<tu email>"
```

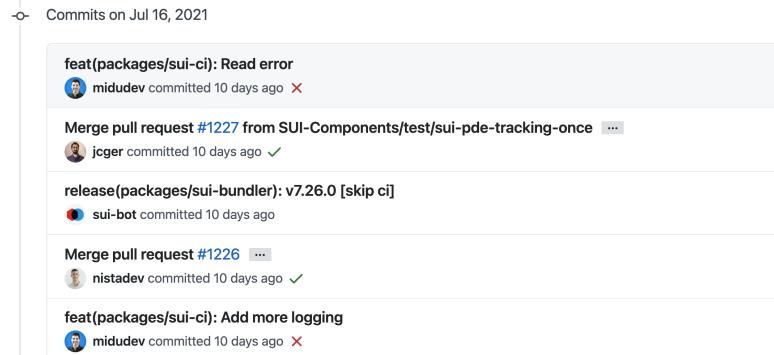
Por ejemplo:

```
$ git config --global user.name "Miguel Ángel Durán"  
$ git config --global user.email "miduga@gmail.com"
```

Si tienes que cambiar esta configuración para un repositorio en concreto, sólo tienes que obviar el parámetro `--global`. Esto te permitirá poder configurar tu nombre y correo para cada repositorio.

```
$ cd <tu directorio de repositorio>  
$ git config user.name "midudev"  
$ git config user.email "work@midu.dev"
```

-o- Commits on Jul 16, 2021



feat(packages/sui-ci): Read error
midudev committed 10 days ago ✘

Merge pull request #1227 from SUI-Components/test/sui-pde-tracking-once ...
jcger committed 10 days ago ✓

release(packages/sui-bundler): v7.26.0 [skip ci]
sui-bot committed 10 days ago

Merge pull request #1226 ...
nistadev committed 10 days ago ✓

feat(packages/sui-ci): Add more logging
midudev committed 10 days ago ✘

Al configurar correctamente tu correo, servicios como GitHub mostrarán el nombre y el avatar del usuario que ha realizado el commit

¿Cómo configurar el editor por defecto que abre Git?

Por defecto `git` intenta abrir el editor de texto `vim` para que puedas modificar los ficheros cuando encuentra conflictos o para darte la opción de escribir mensajes de commit más largos (en algunos sistemas operativos esto puede variar).

Sin embargo, aunque `vim` está ampliamente disponible en casi todos los sistemas operativos, se trata de un editor que tiene una pequeña curva de aprendizaje y no todo el mundo puede usarlo fácilmente.

Si lo prefieres, puedes configurar Git de la siguiente manera para que abra el editor de texto de tu elección:

```
$ git config --global core.editor "code"
```

En este caso, `code` (el comando ejecutable de Visual Studio Code) es el editor que queremos que Git utilice. Pero podéis usar otros editores como:

```
$ git config --global core.editor "atom" # Atom
$ git config --global core.editor "subl" # Sublime Text
$ git config --global core.editor "nano" # Nano
```

¿Cómo puedo comprobar mi configuración de Git?

Antes de editar tu configuración, también puede ser buena idea revisar qué configuración estás utilizando actualmente en git. Para mostrar todas las opciones puedes usar el siguiente comando:

```
$ git config --list

user.name=midudev
user.email=miduga@gmail.com
pull.rebase=false
core.repositoryformatversion=0
core.filemode=true
core.bare=false
core.logallrefupdates=true
core.ignorecase=true
pull.rebase=true
```

Si te fijas bien, podrás ver que la configuración para `pull.rebase` está repetida y tiene valores distintos. Esto puede ser normal, ya que puedes tener más de un archivo `gitconfig` en tu sistema (puedes tenerlo de forma local en el repositorio actual, de forma global o ser configuración del sistema). Lo importante que debes saber es que el último valor de la lista es el que prevalece.

Además, puedes usar los argumentos `--global`, `--local` y `--system` para leer sólo la configuración que te interese, pero ten en cuenta que sólo verás una parte de la configuración que realmente se está utilizando.

```
# para mostrar sólo la configuración global de git
$ git config --global --list

# para mostrar la configuración del repositorio local (si existe)
$ git config --local --list

# muestra la configuración del git del sistema (si existe)
$ git config --system --list
```

Alternativamente, puedes usar el parámetro `--show-scope` para saber de dónde proviene cada configuración:

```
$ git config --list --show-scope

global user.name=midudev
global user.email=miduga@gmail.com
global pull.rebase=false
local core.repositoryformatversion=0
local core.filemode=true
```

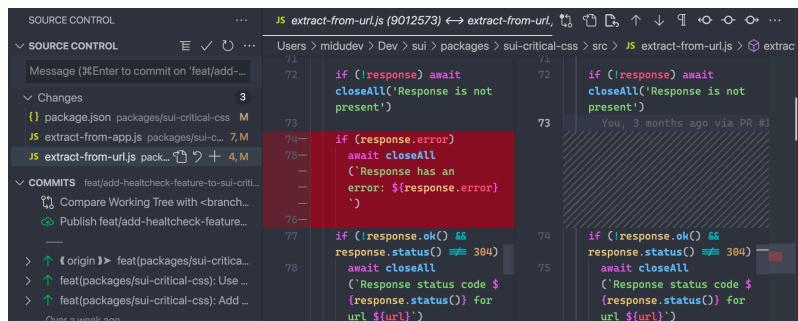
¿Te está fallando el comando? No te preocunes. Es posible que estés usando una versión antigua de Git, ya que la opción `--show-scope` fue introducida en la versión 2.26.0. Aunque puedes continuar sin problemas si obvias la opción... te recomiendo que te tomes un tiempo a actualizar tu versión de Git.

Por último, si quieres saber el valor de una configuración en particular, también puedes conseguirlo usando `git config <configuración>` de esta forma:

```
$ git config user.email
miduga@gmail.com
```

Git y la línea de comandos

Hay diferentes maneras de usar Git. Existen muchos programas visuales que te permiten utilizar git y administrar tus repositorios locales y remotos sin saber exactamente qué está haciendo el sistema de control por debajo... y son fabulosos, sin duda. Yo, sin ir más lejos, estoy encantado de usar la integración de *Git* con *Visual Studio Code*.



The screenshot shows the Visual Studio Code interface with the GitLens extension active. On the left, the Source Control sidebar displays a list of changes and commits. The main editor area shows a diff of a file named 'extract-from-url.js'. The GitLens extension has added annotations to the code, such as status indicators for lines (green for added, red for deleted) and hover cards with commit details like author, date, and PR links. The code itself is a snippet of JavaScript performing an await operation on a response object.

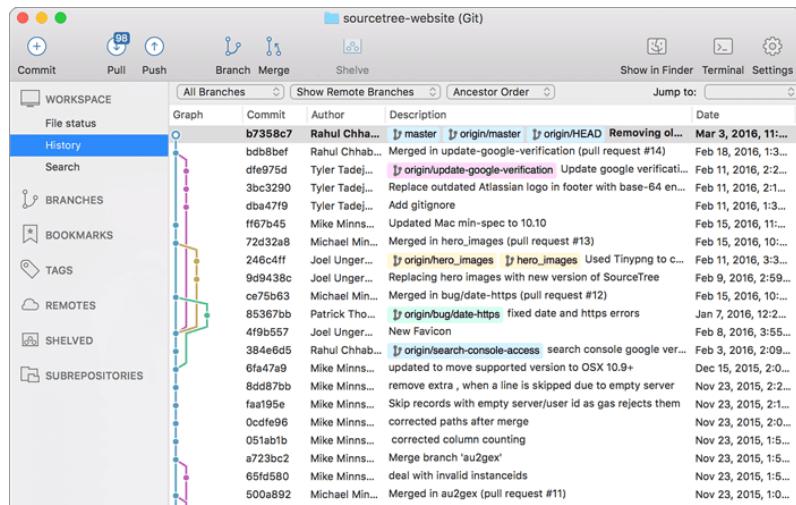
**La integración de Git con Visual Studio Code es fantástica.
Puedes usar la extensión GitLens para incluso llevárla al siguiente nivel.**

Sin embargo, en este libro vamos a estar usando siempre **la línea de comandos de Git**: `git`. Es indudablemente la interfaz más básica que se puede usar para trabajar con Git, pero también es la única que te ofrece las más complejas y completas funcionalidades de control. Además, sólo esta interfaz es completamente universal e independiente de la plataforma, del sistema operativo y de los editores de texto, también conocidos como IDE.

Pero hay otra razón que considero todavía más importante. Creo que saber cómo funcionan las cosas por debajo es fundamental para cualquier desarrollador, puesto que de esta forma entenderás lo que estás haciendo y serás capaz de enfrentarte a cualquier problema o reto que se te presente.

Verás que la línea de comandos de `git` se escribe siempre usando `git <verb>` (`git config`, `git add`, `git commit`, etc.) y que todos los verbos tienen una sintaxis específica. Si alguna vez tienes dudas sobre las posibilidades de un comando, puedes escribir `git <verb> --help` para ver

una descripción de lo que hace ese comando y una lista de todas las opciones disponibles.

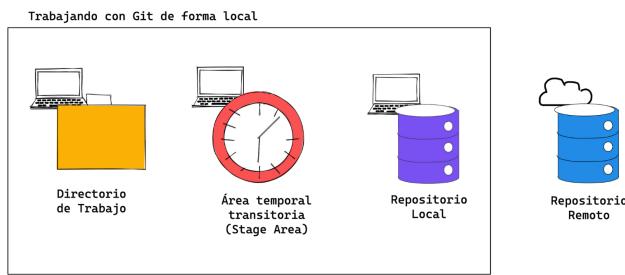


Sourcetree es una aplicación de escritorio gratuita para macOS y Windows que te ofrece manejar tu repositorio de Git

Trabajando con Git de forma local

Podemos usar Git en nuestro sistema, sin necesidad de enviar los cambios a la nube. Esto nos permitirá empezar a trabajar con Git, conocer algunos conceptos nuevos y, al menos, poder versionar los cambios que hacemos localmente.

En este capítulo vas a aprender a iniciar un proyecto desde cero en tu máquina y nos vamos a enfocar en las tres primeras etapas de un proyecto de Git:



El directorio de trabajo, el área temporal transitoria y el repositorio local están todo en nuestra máquina y son las primeras etapas en Git

¿Cómo inicializar un nuevo proyecto Git?

Si quieras crear un proyecto desde cero (dicho de otra forma, **crear un repositorio local**), puedes usar el comando `git init` e indicarle el nombre del proyecto. Esto creará una carpeta configurada y vacía con el nombre que le has indicado.

```
$ git init nuevo-proyecto  
$ cd nuevo-proyecto
```

Es posible que quieras iniciar un repositorio de una carpeta ya existente. Para ello, simplemente puedes usar el comando `git init` dentro de la raíz del directorio del proyecto.

```
$ cd <directorio del proyecto que ya existe>  
$ git init
```

Verás que en cualquiera de los dos casos te ha creado una rama principal por defecto (a día de hoy `master` aunque esto se está discutiendo y es posible que pronto pase a llamarse `main` por defecto). Esta rama es la rama principal de tu proyecto y es la que se usará para trabajar.

Si quieres que al crear un repositorio, la rama principal tenga otro nombre, puedes usar el parámetro `--initial-branch` para indicar el nombre de la rama que te gustaría usar. `git init nuevo-proyecto --initial-branch=main`. También puedes cambiar la configuración `init.defaultBranch` para cambiar el nombre de la rama principal que se usa por defecto.

A partir de aquí ya tienes tu repositorio inicializado. Eso sí, **sólo de forma local**.

Para revisar que todo ha ido bien puedes comprobar que existe el directorio `.git` dentro de nuestro proyecto. Para ello, en sistemas Unix, podéis usar `ls -al` o `dir` en Windows.

```
$ ls -a
```

```
.           assets      layouts      resources
LICENSE     content      package.json static
.git        README.md    public       vercel.json
```

En este directorio es donde git va a guardar toda la información de tu proyecto. Todos los archivos, histórico de cambios, bifurcaciones y más.

Importante: No borres nunca el directorio `.git` de tu proyecto. Si lo borras, no podrás acceder a los datos del repositorio y podrías perder todo lo que hayas hecho. Al menos, asegúrate siempre de haber sincronizado todos los cambios y ramas que te interesen con un repositorio remoto (`git push`) antes de borrarlo... si no quieres perder nada.

Otra forma sencilla de saber si el proyecto actual tiene un repositorio inicializado correctamente es ejecutar el comando `git status` en el directorio del proyecto.

```
# en un directorio que no contiene un repositorio:
git status

fatal: not a git repository (or any of the parent directories): .git

# en un directorio que contiene un repositorio:
git status

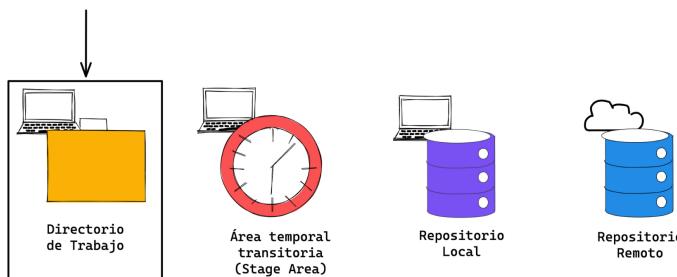
On branch main

No commits yet

nothing to commit (create/copy files and use "git add" to track)
```

¿Qué es el directorio de trabajo?

El directorio de trabajo es simplemente la carpeta de tu sistema de archivos donde tienes todos los ficheros de tu proyecto y en el que has iniciado tu repositorio. Esto es, el directorio que contiene los archivos que has creado o crearás y que han sido modificados.



El directorio de trabajo es la primera etapa en el ciclo de desarrollo de Git

Sitúate en el directorio en el que iniciaste previamente tu repositorio. Ejecuta `git status` para asegurarte que te encuentras en el directorio correcto.

Ahora, crea un archivo en tu directorio de trabajo. Para ello, puedes usar tu editor favorito o, simplemente, puedes utilizar el comando:

```
$ touch index.html
```

Con esto, has creado un archivo en el directorio de trabajo. Si vuelves a ejecutar el comando `git status` verás que Git te indicará que el archivo `index.html` no está siendo rastreado.

```
On branch main
```

```
No commits yet
```

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

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

Con esto, lo que hemos conseguido es dejar nuestro archivo `index.html` en el estado de **modificado**.

¿Te parece que `git status` es demasiado verboso? No te preocupes, puedes reducir la información que muestra con el comando `git status -s`. El parámetro `-s` es la forma corta de usar la opción `--short` y hace que la salida del comando se simplifique considerablemente.

```
$ git status -s  
M manuscript/changelog.md  
M manuscript/ramas.md  
M manuscript/ssh.md
```

¿Cómo deshacer un archivo modificado?

Acabas de modificar un fichero y te das cuenta que **quieres volver a la versión original que tenías en el directorio de trabajo**. Lo que tendrás que hacer dependerá del estado previo de ese archivo en el directorio de trabajo:

Si has modificado un archivo o directorio que ya había sido commiteado anteriormente...

Esto significa que has introducido una modificación sobre un archivo que ya había sido grabado en el repositorio previamente con `git commit` (que más adelante veremos cómo usar).

En ese caso puedes restaurar la versión previa de ese fichero usando el comando `git restore`:

```
# restaurar el archivo index.html
$ git restore index.html

# restaurar todo el directorio de trabajo
$ git restore .

# restaurar todos los archivos terminados en *.js
$ git restore '*.js'
```

¡Ten en cuenta que esto hará que pierdas los cambios que habías realizado en el archivo!

Si intentas ejecutar este comando en un archivo que no había sido grabado con un *commit*, te encontrarás un error como este:

```
# creamos un fichero new.html
$ touch new.html
# intentamos restaurar el fichero new.html
$ git restore new.html
error: pathspec 'new.html' did not match any file(s) known to git
```

Este error es normal, ya que `git restore` no puede, obviamente, restaurar un archivo que no existía antes. Para saber cómo puedes deshacerte de un archivo modificado, consulta la siguiente sección.

No te preocunes si todavía no sabes qué es un *commit*. Más adelante veremos qué significa y cómo hacerlo.

`git restore` es un comando relativamente nuevo, disponible a partir de la versión de Git 2.23. A día de hoy es la forma recomendable de realizar esta operación pero es posible que todavía no lo tengas disponible. Lo sabrás inmediatamente si ves este mensaje:

```
git: 'restore' is not a git command. See 'git --help'
```

No te preocunes si es así. Históricamente, para lograr esto mismo, se ha usado `git checkout` pero es un comando que hace demasiadas cosas y te puede generar confusión.

Igualmente, en el caso que no tengas disponible `git restore` te explico cómo puedes lograrlo con `git checkout`:

```
# restaurar el archivo styles.css
$ git checkout -- styles.css

# restaurar todos los archivos markdown
$ git checkout -- '*.md'

# restaurar todo el directorio de trabajo
$ git checkout .
```

Si el archivo o directorio no existía previamente...

Esto significa que queremos *limpiar* lo que hemos creado. Una opción sería simplemente borrar manualmente lo que hayas creado pero Git tiene un comando que puede ayudarte.

El comando es `git clean` y nos permite borrar todos los archivos que todavía no han sido rastreados por Git.

Si ejecutas el comando tal cuál verás que te da un error:

```
$ git clean  
fatal: clean.requireForce defaults to true and neither -i, -n, nor -f  
given; refusing to clean
```

Vamos a ver lo que nos ofrecen las opciones:

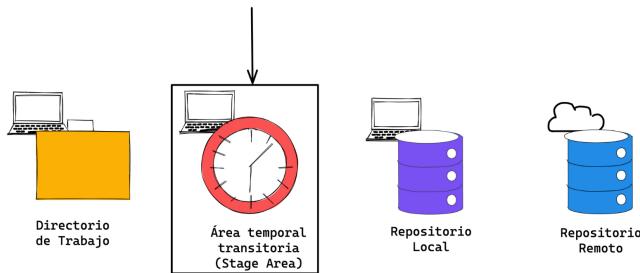
- `-n` para ejecutar un *dry-run*. Esto hará que se ejecute el comando y te diga cuál sería el resultado de borrar los archivos... ¡sin borrarlos!
- `-f` forzar el borrado de los archivos.
- `-d` para borrar también directorios que no existían antes.
- `-i` para que te pregunte antes de borrar cada archivo.

Un ejemplo de uso para un fichero que hemos creado en el directorio de trabajo y luego nos damos cuenta que queremos eliminar.

```
# creamos un fichero en el directorio de trabajo  
$ touch nuevo-archivo.js  
# ejecutamos git clean con el parámetro -n  
$ git clean -n  
Would remove nuevo-archivo.js  
# ahora que ya sabemos lo que borraría, lo hacemos  
$ git clean -f
```

¿Cómo añadimos archivos al área de staging?

El área de staging es un área temporal transitoria donde podemos mover los archivos **modificados** de nuestro directorio de trabajo al estado de **preparados**. Desde este estado, más adelante, los podremos pasar a **confirmados** y quedarán grabados en el repositorio.



Se llama también Área Temporal, ya que es un estado transitorio por el que pasan los archivos que, más adelante, vamos a querer grabar en el repositorio local

Para lograr el movimiento, tendremos que ejecutar el siguiente comando:

```
$ git add <archivos>
```

Esto añadirá los archivos indicados al área de staging. Si quieres añadir más de un fichero, puedes usar una lista de archivos separados por espacio o incluso puedes usar algunos patrones para indicar extensiones o directorios. Aquí tienes algunos ejemplos para que veas cómo lo podrías usar.

```
# el archivo que hemos creado anteriormente  
$ git add index.html  
  
# añade los archivos archivo1.js y archivo2.js  
$ git add archivo1.js archivo2.js
```

También puedes añadir todos los ficheros que terminen con una extensión en concreto:

```
# añade todos los archivos que terminen en .js  
$ git add *.js  
  
# añade todos los archivos que terminen en .html  
$ git add *.html
```

O directamente añadir todos los ficheros modificados:

```
$ git add --all  
$ git add -A  
  
# si estás en la raíz de directorio de trabajo  
$ git add .  
  
# todos los ficheros modificados de la carpeta "resources"  
$ git add resources/
```

Si intentas mover a este estado un archivo que no existe, recibirás un error como el siguiente:

```
$ git add archivo-que-no-existe.md  
fatal: pathspec 'archivo-que-no-existe.md' did not match any files
```

Recuerde que cada vez que añadas algo al *área de staging*, puedes usar `git status` para comprobar qué archivos están ahí y en qué estado se encuentra cada uno.

¿Cómo puedo sacar un archivo o varios del área de staging?

Es posible que hayas añadido algún archivo al área de staging pero en realidad no quieras grabarlo en el repositorio. Existe un comando que te permite volver del estado de **preparado** a **modificado** uno o varios ficheros.

Vamos a ver un ejemplo completo:

```
# añadimos el archivo index.html y
# usamos el status para ver que está en el área de staging
$ git add index.html
$ git status
On branch main

No commits yet

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

¡Oh no! Hemos añadido el fichero `index.html` pero en realidad no queremos que se grabe en el repositorio. Para sacar este fichero tienes que ejecutar el comando `git reset` de la siguiente forma:

```
$ git reset index.html
```

Ahora, si ejecutamos un `git status` veremos que el fichero `index.html` ha quedado en el estado de **modificado** y ha salido del área de staging.

```
On branch main

No commits yet

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

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

Como ves, para usar `git reset` hay que indicar el fichero.

```
git reset <archivo>
```

Por ejemplo:

```
git reset archivo2.js
```

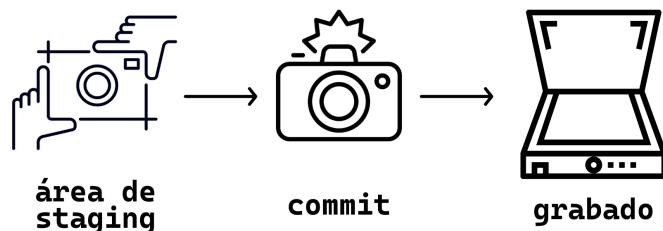
¿Qué es un commit?

Los commits sirven para registrar los cambios que se han producido en el repositorio. Es una de las piezas más importantes para entender cómo funciona Git.

Piensa en los commits como si fuesen fotografías. Cada fotografía muestra el estado de todos los archivos de tu repositorio en el momento en que se hizo y cada una va firmada con el *autor*, la *fecha*, *localización* y otra información útil.

Siguiendo con la analogía, si hicieses más fotos, al final tendrías un *álbum de fotos* que te contaría la evolución de un viaje o cualquier acontecimiento que estabas registrando con tu cámara.

Este álbum sería el historial de cambios del repositorio, que te permitiría entender qué hizo cada *commit* y en qué estado se encontraba el repositorio entonces.



El área de staging sería el encuadre, el commit la fotografía y, para guardarla, el repositorio escanearía la foto para replicarla

Es muy importante que entiendas que **cada commit es una fotografía de todo tu repositorio**, no sólo de las diferencias que hay entre los archivos que has modificado.

¿Cómo puedo hacer un commit?

Si quieras **guardar los cambios que tienes en el área de staging**, puedes hacer un commit con el siguiente comando:

```
$ git commit
```

Esto creará un nuevo commit en tu repositorio y añadirá una referencia al commit en la rama en la que estás trabajando. Pero antes de hacerlo, tienes que indicar el mensaje del commit. Para ello te abrirá el editor que hayas configurado previamente.

Si quieres añadir directamente un mensaje sin abrir el editor, puedes usar el parámetro `-m` o `--message`:

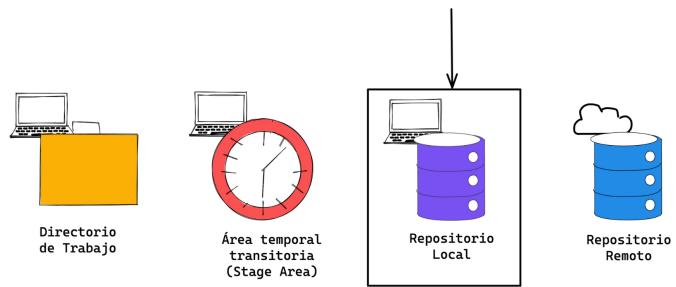
```
$ git commit -m "Add new search feature"
```

El mensaje especificado se usará como título del commit para describir los cambios realizados. Si quieres añadir más información a tu commit, más parecido a lo que podías hacer al abrir el editor del mensaje de commit sin parámetro, puedes volver a utilizar el parámetro `-m` tantas veces como quieras.

```
$ git commit -m "Add new search feature" -m "This new feature is awesome. Make the search to work faster and better!"
```

En el capítulo de *Buenas prácticas*, verás diferentes reglas que puedes seguir para escribir los mejores mensajes de commit posibles.

Ahora, es importante que entiendas que **estos cambios se van a grabar en tu repositorio local**. A partir de aquí, para deshacer estos cambios, tendrás que revertirlos grabando un nuevo commit en el historial de cambios del repositorio.



Una vez hemos confirmado los cambios, éstos se quedan grabados en el repositorio local

¡Recuerda! Hasta ahora estamos trabajando a nivel local y, por lo tanto, estos cambios sólo se quedan disponibles en nuestra máquina. Más adelante veremos cómo podemos enviar estos cambios a la nube para que otras personas puedan recuperarlos y trabajar sobre ellos.

Haz commit y sáltate el área de staging

El proceso natural en Git es que los archivos que has modificado, y quieres que sean grabados en el repositorio, deben estar en el área de staging, tal y como hemos visto antes.

Sin embargo, puedes saltarte directamente la etapa de staging y hacer un commit directamente con los ficheros que han sido modificados.

Para ello, puedes usar `git commit` pero con el parámetro `-a` o `--all`. De esta forma, Git preparará automáticamente todos los ficheros que hayas modificado y los añadirá al commit.

```
# vemos que tenemos un archivo modificado
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working direct\ory)

    modified:   CONTRIBUTING.md

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

# creamos un commit con los ficheros que hemos modificado
# y con un mensaje determinado
# ;sin usar git add previamente!
$ git commit -a -m 'Add new files without staging'

[master ad3385f] Add new files without staging
 1 file changed, 5 insertions(+), 0 deletions(-)

# también puedes agrupar los parámetros del comando
# en lugar de -a -m puedes usar -am
# de hecho es el que usaremos en el resto del libro
$ git commit -am 'Add other files without staging'

[master bb2235d] Add changes without staging
 1 file changed, 1 insertions(+), 0 deletions(-)
```

Ten en cuenta que **esto sólo funciona para archivos modificados**. Si es la primera vez que los has creado, vas a tener que añadirlos manualmente al área de staging con `git add` antes de poder usar `git commit`.

¿Qué es el HEAD?

HEAD es el puntero que referencia el punto actual del historial de cambios del repositorio en el que estás trabajando.

Normalmente será el último *commit* de la rama en la que te encuentres pero como también puedes moverte entre commits... es posible que HEAD no sea el último commit.

¿Y por qué se llama HEAD? Como cada cambio que se graba en el repositorio se identifica con un *hash*, del que hablaremos más adelante, sería muy complicado referirte a un commit concreto para situarte.

Imagina que puedes viajar en el tiempo y en el espacio. Pues el HEAD sería la referencia que siempre apunta al punto en el que estás, independientemente de dónde y cuándo te encuentres.

Puedes saber el valor de HEAD con el comando `git symbolic-ref HEAD`:

```
$ git symbolic-ref HEAD  
refs/heads/main
```

Otra forma de acceder a él sería mirando el contenido del archivo `.git/HEAD` aunque no es recomendable. Por otro lado, si lo que quieras no es tener la referencia de la rama pero sí el valor del HEAD en el que estás, puedes usar el comando `git rev-parse HEAD`.

```
$ git rev-parse HEAD  
5d689b826e172a7a5b78ce60c30b7d0e0891197c
```

Ahora te preguntarás... **¿por qué es importante saber qué significa HEAD?** Pues bien, más adelante verás que algunos comandos lo utilizan para indicar dónde estás trabajando o hacia dónde quieres moverte.

Entiende HEAD como un “estás aquí” de un mapa. Sólo puedes estar en un sólo lugar y ese lugar es el HEAD (en mayúsculas).

¿Cómo puedo deshacer mis cambios?

Lo mejor que tiene *Git* es que *casi* siempre que te equivocas puedes deshacer tu cambio y seguir siendo feliz. Digamos que te proporciona un montón de redes de seguridad para evitar que hagas cosas que... no deberías.

Una de esas veces es cuando hacemos un commit. **¿Qué pasa si nos hemos equivocado? ¿Cómo deshacemos el último commit? ¿Y si ya lo he publicado?**

Vamos a ver cómo podemos deshacer un commit que hemos hecho en nuestro repositorio local. **Más adelante, veremos cómo podemos conseguir lo mismo si ya hemos llevado nuestros cambios a la nube.**

Deshacer el último commit

A veces **queremos tirar para atrás el último commit** que hemos hecho porque hemos añadido más archivos de la cuenta, queremos hacer commit de otra cosa o, simplemente, porque ahora no tocaba.

Si todavía no has llevado tus cambios al repositorio remoto tienes dos formas de hacer esto. Ambas son válidas pero dependerá si quieres, o no, mantener los cambios del commit.

- Si **quieres mantener los cambios:**

```
$ git reset --soft HEAD~1
```

Con el comando `reset` hacemos que la rama actual retroceda a la revisión que le indicamos. En este caso le decimos `HEAD~1`. Esto significa que queremos volver a la versión inmediatamente anterior a la que estamos ahora.

`HEAD~1` es una referencia a la versión anterior a la que estamos ahora. Es equivalente a `HEAD^` pero en algunos sistemas el uso del carácter `^` puede ser problemático. Además, creo que `HEAD~1` es más fácil de leer.

El parámetro `--soft` es el que va a hacer que los cambios que habíamos hecho en el commit, en lugar de eliminarlos, nos los mantenga como cambios locales en nuestro repositorio.

De hecho, al ejecutar este comando, no se eliminarán los cambios del commit, sino que se mantendrán como cambios locales. Por lo que podrías volver a hacer exactamente el mismo commit que antes.

```
> git reset --soft HEAD~1
> git status
On branch main
Changes to be committed:
(use "git restore --staged <file>..." to unstage)
  modified: README.md
  modified: index.js
```

- Si **NO quieres mantener los cambios:**

```
$ git reset --hard HEAD~1
```

Es simplemente el mismo comando pero cambiamos `--soft` por `--hard`. Esto eliminará los cambios de los que habíamos hecho commit anteriormente. **¡Ojo! ¡Asegúrate que eso es lo que quieres antes de hacerlo!**

Si quieres arreglar el último commit

A veces no quieres tirar atrás el último commit que has hecho si no que simplemente quieres arreglarlo. Aquí hay dos opciones:

- **Sólo quieres arreglar el mensaje que has usado para el último commit:**

```
$ git commit --amend -m "Este es el mensaje correcto"
```

- **Quieres añadir más cambios al último commit:**

```
# Añade los archivos con modificaciones
# que quieras añadir al commit anterior
$ git add src/archivo-con-cambios.js
# Vuelve a hacer el commit con el parámetro amend
$ git commit --amend -m "Mensaje del commit"
```

Ya sea que sólo quieras cambiar el mensaje de commit o que además quieras añadir modificaciones en el último commit, lo importante es que **esto NO va a crear un nuevo commit si no que va a solucionar el anterior.**

Importante: El parámetro de `--amend` es muy útil pero sólo funciona con el último commit y siempre y cuando NO esté publicado en el repositorio remoto. Si ya has hecho `push` de ese commit, esto no va a funcionar. Deberías hacer un `git revert` en su lugar pero no te preocupes que lo explicaremos más adelante.

¿Cómo puedo ignorar archivos?

No todos los archivos que tienes en tu directorio de trabajo tendrán que ser controlados por Git para ser guardados en el repositorio. **A veces tienes ficheros o directorios enteros que no quieres grabar a ningún repositorio**, ya que son de configuración o no añaden valor al historial de cambios.

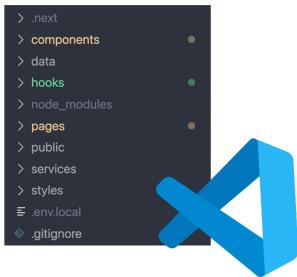
Para que Git los ignore, debes añadirlos a un archivo `.gitignore` en tu repositorio. Normalmente se sitúa en la raíz, pero puedes tener uno en cada directorio (además de tener uno global que afectará a todos los repositorios del sistema).

Vamos a ver un ejemplo de que podría ser el contenido de un archivo `.gitignore`:

```
node_modules # Ignorar carpeta de módulos
.env # Ignorar fichero con variables de entorno
.DS_Store # Ignorar fichero de sistema
build/ # Ignorar carpeta generada
```

¿Qué archivos y carpetas deberías colocar aquí? Aquí te dejo una lista:

- Archivos que tengan credenciales o llaves de API (no deberías subirlas al repositorio, simplemente injectarlas por variable de entorno)
- Carpetas de configuración de tu editor (`/.vscode`)
- Archivos de registro (*log files*)
- Archivos de sistema como `.DS_Store`
- Carpetas generadas con archivos estáticos o compilaciones como `/dist` o `/build`
- Dependencias que pueden ser descargadas (`/node_modules`)
- Coverage del testing (`/coverage`)



En Visual Studio Code, los archivos ignorados aparecen con un gris oscuro. Como ves, están disponibles en tu directorio de trabajo pero si haces cambios en ellos, Git los ignorará

Te recomiendo que le eches un vistazo a gitignore.io. Este proyecto genera un archivo .gitignore en base al sistema operativo, editor y proyecto que le indiques y te puede servir de referencia en el futuro.

¿Cómo consigo ignorar siempre los mismos archivos en todos mis repositorios?

Una forma sería copiar una y otra vez el archivo `.gitignore` en todos los repositorios de tu sistema para luego adaptarlo al proyecto al que va dirigido.

Sin embargo, si tienes muy claro que quieras ignorar archivos en todos tus repositorios puedes crear un archivo `.gitignore_global` que te va a ayudar a dejar de preocuparte siempre por los mismos archivos, especialmente aquellos que tienen que ver con tu IDE o sistema operativo.

Para ello, primero creamos el fichero `~/.gitignore_global` con el contenido que queremos que se añada a todos los repositorios. Yo tengo algo muy parecido a esto:

```
# Archivos y directorios de sistema
.DS_Store
Desktop.ini
Thumbs.db
.Spotlight-V100
.Trashes

# Variables de entorno
.env

# Directorios de instalación y caché
node_modules
.sass-cache

# Configuración de editor
### VisualStudioCode ###
.vscode/*
!.vscode/settings.json
!.vscode/tasks.json
!.vscode/launch.json
!.vscode/extensions.json
*.code-workspace
```

Ahora, vamos a actualizar la configuración de `core.excludesfile` para que lea este archivo de forma global en todos los repositorios locales.

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

Ten en cuenta que esto es útil para archivos y directorios de tu propio sistema que quieras ignorar. Pero siempre es mejor que estos archivos estén bien definidos en el `.gitignore` de cada repositorio, puesto que es la referencia única que todo el mundo se descargará del repositorio remoto.

Por si te interesa, la compañía GitHub tiene un repositorio con una colección de [archivos de `.gitignore`](#) para diferentes lenguajes y sistemas operativos. No está muy actualizada pero puede ayudarte a entender qué ficheros deberías incluir ahí.

¿Cómo le indico a Git que deje de hacer el seguimiento de un archivo (o varios archivos)?

Imagina que has hecho commit de un fichero pero resulta que, más tarde, te has dado cuenta que en realidad no quieres que este archivo sea parte de tu repositorio. ¿Ahora qué hacemos?

Ya hemos visto que deberíamos haberlo añadido al `.gitignore...` pero a veces nos damos cuenta tarde. ¡No pasa nada! Vamos a ver cómo podríamos arreglar esto.

Primero, añade en el `.gitignore` el archivo o directorio que quieras que no sea parte de tu repositorio tal y como hemos visto antes.

Cómo conseguirlo manualmente...

Ahora, en el directorio de trabajo, podríamos borrar los ficheros que no queremos que sean parte del repositorio y luego hacer commit de los cambios.

Por ejemplo, imagina que quieres eliminar el archivo `config.local.js` de tu repositorio. Para ello, primero debes borrarlo del directorio de trabajo:

```
$ rm config.local.js
$ git add config.local.js
$ git commit -m "Remove config.local.js as it's not part of the repository"
```

Una vez hecho esto, la próxima vez que creemos el fichero `config.local.js` en el directorio de trabajo, no podrá incluirse en el repositorio ya que lo tendremos ignorado.

Usando git...

Esto mismo podemos conseguirlo con un simple, y seguramente más acertado comando, usando `git`. Para ello, debemos usar el comando `rm` e indicarle qué archivos queremos borrar. Siguiendo con el ejemplo anterior:

```
$ git rm config.local.js  
$ git commit -m "Remove config.local.js to ignore it"
```

¿Por qué usar `git rm`? Hay varias razones:

1. Porque `rm` es un comando de sistema y no de `git`. Por lo que es posible que no esté disponible en todos los sistemas operativos.
2. Porque `git rm` es un sólo comando y simplifica la tarea de borrar archivos de un repositorio.
3. `git rm` va a evitar que se borre si el archivo tiene alguna modificación.
4. Puedes usar el parámetro `--dry-run` para ver qué archivos se borrarán sin realizar la operación.

Como ves... ¡todo ventajas!

Las dos formas eliminan el fichero de tu directorio de trabajo. Pero... ¡Ten en cuenta que esto no elimina el archivo de tu historial! Por lo que aunque el fichero no exista en tu sistema o en el directorio de trabajo, el archivo sigue en el historial de cambios del fichero de `Git` y alguien podría recuperarlo. Hay otras formas de conseguir sobrescribir el historial para que tampoco esté disponible ahí.

¿Y cómo consigo lo mismo pero manteniendo el fichero en mi directorio de trabajo?

En ocasiones quieras que un archivo deje de ser parte de tu repositorio **pero que no se borre del directorio de trabajo**. Por ejemplo, a veces puede ser interesante para archivos de configuración o que han sido generados, y que no quieras volver a generarlos de nuevo localmente.

Puedes lograrlo también con `git rm` pero añadiendo el parámetro `--cached`:

```
$ git rm --cached <nombre-de-archivo>
```

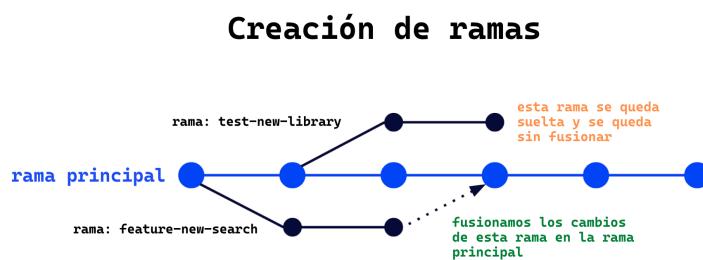
De esta forma, ese fichero se eliminará del repositorio, se añadirá al historial de cambios, pero no se borrará del directorio de trabajo.

Si necesitas borrar una carpeta y todos los ficheros que lo contiene, puedes utilizar el comando `git rm -r`. La `-r` indica que se borrará la carpeta y todos sus ficheros de forma recursiva.

Ramas en Git

Ya sabemos cómo trabajar con repositorios locales pero, lo cierto, es que hasta ahora hemos trabajado de forma lineal, sin ningún tipo de ramificación, en el que simplemente hemos ido haciendo cambios, grabándolos y luego subiéndolos a nuestro repositorio remoto.

Aunque esto ya tiene suficiente utilidad, Git ofrece mucho más que eso. **Ofrece la posibilidad de crear ramas.** Una rama es una versión del repositorio que se crea a partir de un commit.



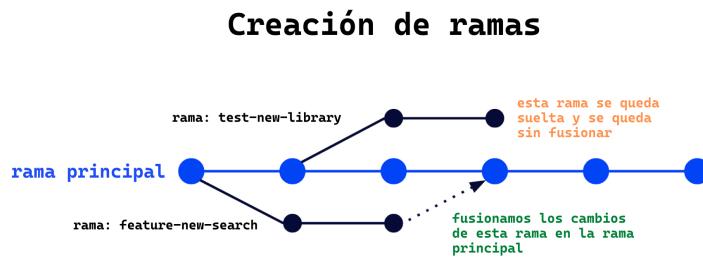
Un ejemplo sencillo son las ramas que se crean a partir de la rama principal. A estas ramas puedes hacerles commits y, más adelante, puedes decidir volver a fusionarla, o no, con la rama principal para que adopte los cambios.

El nombre de *rama* puede llevar a la confusión por la comparación con un árbol. Aunque es cierto que la creación de ramas se parece a las partes que nacen de un troco... en un árbol real éstas nunca vuelven a unirse al tronco. Esto es diferente a las **ramas de Git**, que tras divergir, se pueden volver a unir al tronco para que fusione sus cambios. En ese aspecto, la creación de ramas es más similar a una salida de una autopista que, más adelante, puede volver a llevarte a la misma vía original.

Empezamos con las ramas

La creación de ramas **nos permite el trabajo en paralelo** sobre una misma base de código o, dicho de otra forma, una rama representa una línea completamente independiente de desarrollo.

Al grabar cambios con *commits* en una rama, se genera una bifurcación en el historial de cambios del proyecto. Estos cambios pueden ser más adelante integrados en otra rama (normalmente la rama principal) o se puede eliminar la rama y dejar los cambios sin efecto.



En este ejemplo hemos creado dos ramas. La de arriba nos ha servido para probar alguna cosa y finalmente no la fusionamos a la rama principal. La de abajo se fusiona a la rama main tras grabar dos commits

Dominar el uso de ramas es esencial para trabajar con Git. Como verás más adelante existen algunas estrategias de trabajo que las usan constantemente. El trabajo en paralelo es clave y, por eso, la mayoría de personas y compañías las usan en su día a día.

Creando nuestra primera rama

El comando `git branch` nos permite crear, listar, eliminar y renombrar ramas. Al crear una rama, para movernos a ella, tendremos que usar otro comando: `git switch`.

```
# creamos la rama mi-primer-rama
$ git branch mi-primer-rama

# cambiamos a la rama mi-primer-rama
$ git switch mi-primer-rama
Switched to branch 'mi-primer-rama'
```

Si quieras hacer los dos pasos a la vez, puedes usar el comando `git switch -c mi-primer-rama`. Esto creará la rama y te llevará a ella con un sólo comando. Ten en cuenta que, si el nombre de la rama ya existe, recibirás un error:

```
# creamos la rama mi-primer-rama
$ git switch -c mi-primer-rama

# intentamos crear una rama con el mismo nombre
$ git switch -c mi-primer-rama
fatal: A branch named 'mi-primer-rama' already exists.
```

Para poder crear una rama debes, al menos, tener un commit en el repositorio. Si ejecutas `git init` y lo primero que intentas es crear una rama con `git branch` verás que recibes un error. Tiene sentido. ¿Cómo se podría crear una rama si no existe ningún tronco del que sostenerla?

git checkout, el comando que hacía demasiadas cosas

Si ya sabías algo de Git seguramente te estés preguntando... ¿Por qué no he usado `git checkout` para crear y cambiar entre ramas? Es tan sencillo como usar un simple comando:

```
# crea la rama mi-primer-rama
# y además cambia a ella
$ git checkout -b mi-primer-rama
Switched to a new branch 'mi-primer-rama'
```

Sin embargo, [desde agosto de 2019 el comando `git` incluye dos nuevos sub-comandos `git switch` y `git restore`](#). La idea era separar las responsabilidades de `git checkout` en varios comandos. Y, personalmente, considero que tiene bastante sentido.

Revisando la documentación de `git checkout`, podemos ver que `git checkout` *cambia entre ramas y restaura el directorio de trabajo*.

Podríamos entrar en un debate sobre si realmente esto sigue [la filosofía Unix](#) de hacer que un programa haga una cosa y lo haga bien... (y para no dejarte con la intriga, yo personalmente creo que no lo cumple).

Fue en 2005 que [un commit](#) hizo que `git checkout` pasase a ser un comando más complejo de lo que había sido históricamente. Antes, simplemente servía para manejar ramas.

Por compatibilidad, obviamente, `git checkout` no puede volver a su estado anterior, por lo que desde Git han decidido añadir los dos comandos `git switch` y `git restore`, como comentaba antes, para subsanar esto.

Ahora, `git checkout` no va a ir a ningún sitio. Si ya tienes comodidad trabajando con este comando, puedes seguir haciéndolo. Aún así, te recomiendo que pruebes a usar los nuevos para habituarte. Creo, sinceramente, que son más fáciles de entender y seguramente las personas a las que les enseñas a trabajar con Git te lo agradecerán.

Si nunca has oido hablar de `git checkout`, creo que está bien aprenderlo y entenderlo pero priorizaría a usar los nuevos comandos como mis principales.

Listando las ramas disponibles

Ahora que ya sabemos crear ramas es el momento de saber qué ramas tenemos disponibles en nuestro repositorio.

Por ahora estamos viendo las ramas disponibles a nivel local. Más adelante veremos que también podemos ver las ramas disponibles a nivel remoto que otras personas están trabajando en el mismo proyecto han dejado allí.

Para mostrar una lista de todas las ramas disponibles localmente, sólo tienes que ejecutar el comando `git branch`:

```
$ git branch  
  
feat/remove-husky-usage  
feat/remove-not-needed-deps  
feat/subscribe-to-use-case-without-decorators  
* feat/sui-bundler-webpack-5  
master
```

Verás que una de las ramas tiene, al principio, un *asterisco*. Esto significa que actualmente te encuentras en esa rama.

Normalmente las terminales actuales te indican en qué rama te encuentras sin necesidad de ejecutar un comando pero si necesitas averiguarlo rápidamente, puedes añadir un parámetro al comando para que te indique en qué rama estás actualmente.

```
$ git branch --show-current  
feat/upgrade-sui-lint-dependencies
```

Consigue la lista de las ramas más recientes

Si tienes muchas ramas en tus proyectos locales es posible que te cueste encontrar fácilmente las ramas más recientes.

Para mejorar esto, puedes usar el parámetro `--sort` y usar el valor `committerdate` para ordenar las ramas por fecha de creación.

```
$ git branch --sort=-committerdate  
  
* feat/sui-bundler-webpack-5  
  master  
  feat/migrate-to-webpack5-and-karma-6  
  feat/bump-karma-webpack-version  
  feat/upgrade-to-latest-stylelint  
  feat/upgrade-sui-lint-dependencies
```

Trabajando con ramas

Ahora que ya sabemos crear y mostrar las ramas disponibles, es el momento de trabajar con ellas. Vamos a ver paso a paso, con comandos y de forma ilustrada, el estado en el que se encuentra en cada momento.

Pongamos que nuestra rama principal es `master`:

```
# empezamos en la rama principal master
$ git branch --show-current
master

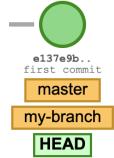
# grabamos un commit en la rama master
$ git commit -am "first commit"
```



Es nuestro primer commit en el repositorio. Fíjate que el puntero HEAD irá cambiando conforme nos vayamos “moviendo”

Ahora vamos a crear una rama llamada `my-branch`, donde vamos a trabajar en una nueva característica para nuestra aplicación:

```
# creamos nuestra primera rama y
# cambiamos a ella para empezar a trabajar
$ git switch -c my-branch
Switched to a new branch 'my-branch'
```



Como podemos ver, el puntero **HEAD** ahora mismo está en la rama **master** y **my-branch**, ya que ambas apuntan al mismo commit

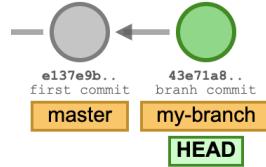
Vamos a empezar a trabajar en esta rama. Para ello, vamos a modificar el fichero `index.js` y vamos a grabar los cambios en esta rama.

```
# estamos en la rama my-branch
$ git branch --show-current
my-branch

# modificamos los ficheros necesarios
# desde nuestro editor favorito
$ code index.js

# si el fichero index.js es nuevo
# tendrás que añadirlo con git add index.js

# vamos a grabar el commit en la rama
$ git commit -am "branch commit"
[my-branch afaf06b] first branch commit
 1 file changed, 5 insertions(+), 0 deletions(-)
```



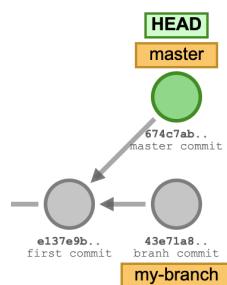
El commit se añade en la rama my-branch. El HEAD apunta a este commit y vemos como este commit tiene una flecha que enlaza con el de master, ya que es el que fue el origen

Ops! Nos hemos dado cuenta que en master hay un pequeño bug. Vamos a tener que cambiar a esa rama y hacer un commit para solucionarlo.

```
# cambiamos a la rama master
$ git switch master

# editamos el fichero en el editor...
$ code file-with-bug.js

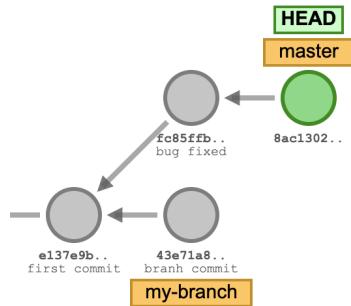
# grabamos el commit en la rama master
$ git commit -am "bug fixed"
```



Ahora vemos como las ramas empiezan a divergir. Ambas ramas comparten el commit inicial, pero la rama master ha ido por un sitio y la rama my-branch por otro

Vaya, pensábamos que ya lo teníamos arreglado. Pero hemos encontrado otro pequeño bug en la rama principal que debemos arreglar.

```
# hay que editar otro fichero para arreglarlo del todo  
$ code another-file-with-bug.js  
  
# grabamos el commit en la rama master  
$ git commit -am "another bug fixed"
```



El puntero HEAD apunta al último commit de la rama actual, que es master. Y vemos que este commit sólo se queda en la rama master

Con este pequeño ejemplo hemos visto como podemos usar las ramas para trabajar en un mismo proyecto en funcionalidades distintas e ir cambiando con `git switch` entre una rama y otra.

Pero... ¿qué gracia tiene hacer esto? ¿No llegan nunca las ramas a encontrarse? Claro que sí. Normalmente, creamos ramas para trabajar en una función o en una característica de nuestra aplicación para, luego, fusionarlas en otras ramas.

¿Quieres seguir trabajando en visualizar cómo quedaría el repositorio al trabajar con ramas? Pues prueba [Visualizing Git](#). Vas a poder ir añadiendo comandos de Git y podrás comprobar en tiempo real cómo va quedando el repositorio.

Fusionando ramas

Las bifurcaciones de código que hemos creado en forma de ramas tendrán **dos destinos**: acabar en el olvido para no terminar en ningún lado o ser fusionada en otra rama.

Cuando hablamos de fusión **nos referimos a que los cambios que hemos realizado en la rama se integran en otra rama**, de forma que el código que habíamos generado en la nueva rama se asimila en otra.

Aunque normalmente este tipo de fusión ocurre de una rama a la rama principal, debes tener en cuenta que en realidad podemos fusionar una rama con cualquier otra rama. Esto puede ser útil para asimilar cambios que alguien del equipo esté haciendo o un fix que todavía no se encuentre en la rama principal.

`git merge` el comando para fusionar ramas

Hemos estado trabajando antes en la rama `my-branch` y ya estamos preparados para llevar todos esos cambios a la rama principal. ¿Cómo podemos conseguirlo? Con el comando `git merge`.

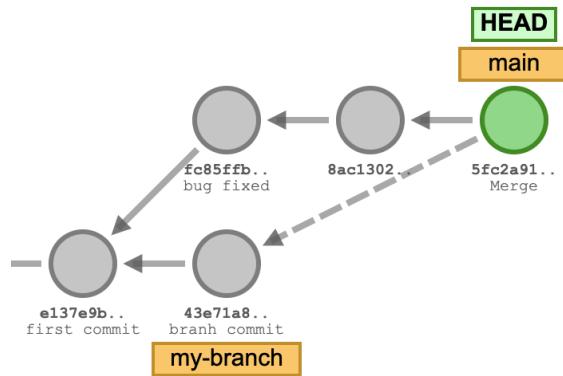
Este comando nos permite incorporar los cambios de una rama a la rama en la que nos encontramos en ese momento. Por ejemplo, si estamos actualmente en la rama `main` y hacemos un `git merge my-branch` haremos que la rama `main` incorpore y fuse los cambios que había en la rama `my-branch`.

Si tienes cambios sin guardar en tu rama actual, Git no te permitirá fusionar nada hasta que los guardes, hagas commit o los elimines. ¡Tenlo en cuenta!

```
# nos aseguramos que estamos en la rama destino  
$ git branch --show-current  
main
```

```
# vamos a incorporar en main los cambios de my-branch  
$ git merge my-branch
```

Si ahora ejecutas un `git log` verás que el último commit incluye la palabra *Merge* Este commit justamente incluye todos los cambios que se habían realizado en la rama `my-branch`. Git ha *replicado* esos cambios en la rama `main` y los ha fusionado.



Al ejecutar el comando `git merge`, se crea un nuevo commit que incluye todos los cambios de la rama de origen a la rama en la que nos encontramos ahora

Tipos de fusión

Git tiene varios tipos de fusión que se usarán dependiendo de la situación de las ramas o de los parámetros que le pasemos. Es importante conocerlas para saber qué está pasando cuando hacemos un `git merge`.

En el anterior caso hemos visto como Git ha creado un nuevo commit `Merge` ... que incluye todos los cambios de la rama de origen a la rama en la que nos encontramos ahora. Esto se conoce como *merge commit*. Esto es lo que ocurre por defecto.

Pero existen otras posibilidades:

Fast-forward

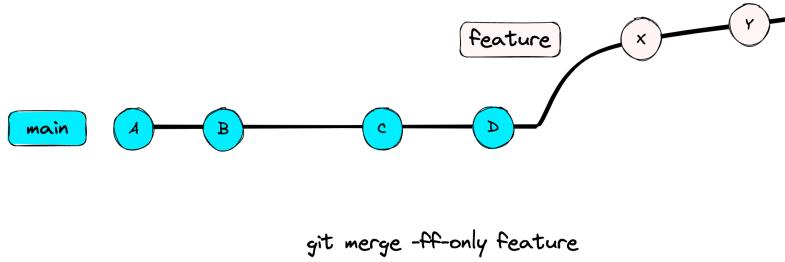
Cuando la rama de origen está por delante de la rama destino, Git simplemente mueve el puntero de la rama destino al último commit de la rama de origen. Esto se conoce como *fast-forward*.

Lo que ocurrirá es que la rama destino pasará a tener todos los commits que tenía la rama de origen y se dibujará como una línea recta sin bifurcaciones.

Para que esto ocurra:

- La rama de origen comparte el historial de commits de la rama destino.
- La rama de origen tiene por delante commits que no tiene la de destino.
- La rama destino no debe tener ningún commit que no esté en la rama de origen.

```
# vamos a la rama donde queremos fusionar cambios
$ git switch main
# traemos los cambios de feature-branch
# usando --ff-only para que solo
# haga fast-forward si es posible
$ git merge --ff-only feature-branch
```



Si la rama a fusionar está por delante de la rama destino y comparten el historial de commits, Git hará un fast-forward

Con ese comando, si el fast-forward no es posible, Git no lo hará y te mostrará un mensaje de error.

Si el `fast-forward` ha sido posible, verás que la rama destino se ha movido al último commit de la rama de origen y que no ha creado el commit de `Merge . . .`, ya que no ha sido necesario.

`git merge` usará este tipo de merge automáticamente siempre que sea posible.

Si quieres que Git siempre use el *merge* tipo *fast-forward*, puedes configurarlo con `git config --global merge.ff only`. Si no es posible, Git te mostrará un error.

No fast-forward

Este tipo de fusión se usa cuando la rama de origen no está por delante de la rama destino. Dicho de otra forma, cuando la rama que va a recibir los cambios tiene commits distintos a la rama que queremos fusionar.

En este caso, Git crea un nuevo commit que incluye todos los cambios de la rama de origen a la rama destino. Esto se conoce como *no fast-forward* o *true merge*.

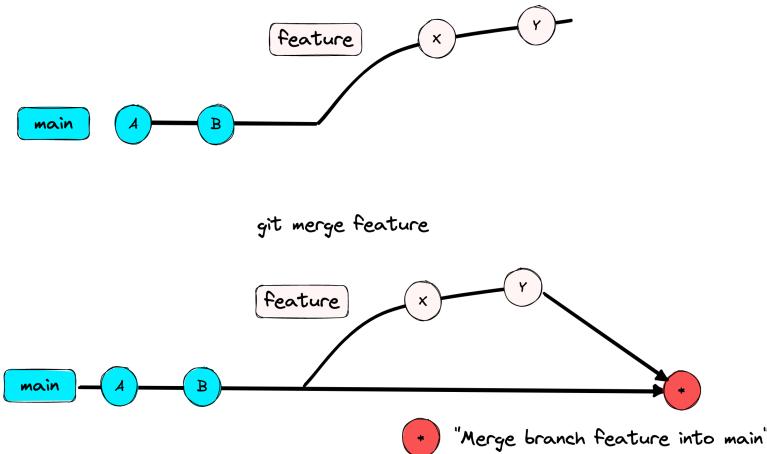
```
# vamos a la rama donde queremos fusionar cambios
$ git switch main

# creamos la rama feature
$ git switch -c feature

# hacemos un commit (después de hacer los cambios)
$ git commit -m "commit 1 feature"

# volvemos a main y hacemos un commit
$ git switch main
$ git commit -m "commit 1 main"

# hacemos el merge de feature sin fast-forward
$ git merge feature --no-ff
```



Si no hacemos un fast forward, Git nos crea un commit que reproducirá los cambios realizados en la rama feature desde que divergió de main.

Squash

Este tipo de fusión de cambios hace la fusión y junta todos los commits de la rama de origen en un solo commit que debe ser confirmado. Esto se conoce como *squash*.

```
# creamos la rama feature desde main
$ git switch -c feature

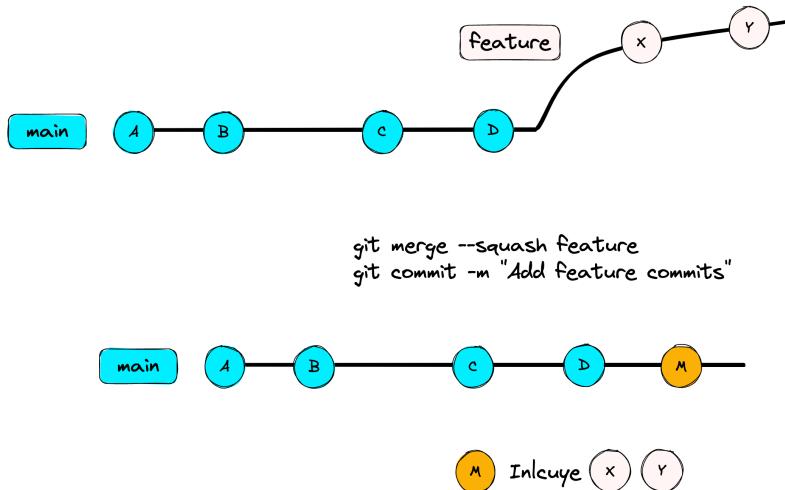
# hacemos tres commits (después de hacer los cambios)
$ git commit -m "commit 1"
$ git commit -m "commit 2"
$ git commit -m "commit 3"

# vamos a la rama donde queremos fusionar cambios
$ git switch main

# traemos los cambios de feature
# usando --squash para que haga squash
$ git merge feature --squash

$ git status
On branch main
Changes to be committed:
(use "git restore --staged <file>..." to unstage)
  new file: index.js
  new file: module.js
  new file: styles.css
```

```
# hacemos commit de los cambios  
$ git commit -m "Merge feature"
```



Si usamos `-squash`, Git agrupa los commits que vamos a fusionar y nos deja los cambios preparados para hacer commit

Como ves, al usar `git merge --squash` no nos crea un commit de Merge ... sino que nos deja los cambios preparados para hacer commit. Esto es porque Git no sabe qué mensaje de commit ponerle y, por tanto, deja que tú lo hagas.

Modificando el mensaje de Merge commit

Por defecto, al ejecutar el `git merge` y según el tipo de fusión, Git crea un commit automáticamente y lo grabado. Sin embargo, es posible que quieras evitar esto. Tienes dos opciones para que no lo haga y, así, modificar el mensaje del commit o hacer otras comprobaciones o cambios antes:

```
# Abre el editor antes de hacer el commit  
$ git merge --edit
```



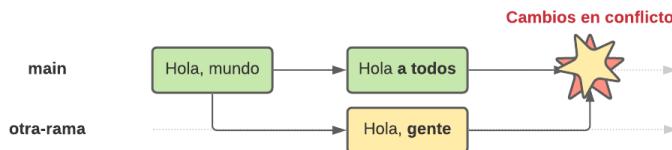
```
# Evita que haga commit automáticamente  
# y te deja los cambios preparados para hacer commit  
$ git merge --no-commit
```

Aunque puede ser útil en algunos casos muy concretos... lo normal es simplemente dejar que Git haga el commit automáticamente.

Resolviendo conflictos

Aunque Git hace un trabajo magnífico a la hora de fusionar ramas, **existen situaciones que pueden dar problemas**. ¿Qué pasa si al querer fusionar dos ramas, la de destino ha realizado cambios en las mismas líneas de un fichero que los que queremos fusionar? Tendríamos **conflictos**.

Un conflicto es una situación en la que **Git no es capaz de determinar qué cambio es el que tiene que prevalecer** una vez ocurra la fusión y, por lo tanto, requiere que el usuario lo resuelva.



Dada la naturaleza de sistema distribuido, es normal que a veces ocurrán conflictos al intentar fusionar dos ramas en Git. ¿Cómo iba a saber Git qué cambio es más importante que otro?

Que exista un conflicto no es malo. Puede ser hasta normal, especialmente cuando muchas personas están trabajando en un mismo proyecto y algunos ficheros son continuamente modificados. Si se convierte en algo demasiado frecuentemente es posible que se quiera evitar...

Creando nuestro primer conflicto

Vamos a ver cómo pueden surgir los conflictos... ¡generando uno! Para ello vamos a crear un repositorio desde cero con un fichero `index.html` con un contenido inicial:

```
$ mkdir git-conflict-test
$ cd git-conflict-test
$ git init .
Initialized empty Git repository in ~/git-conflict-test/.git/
# con este comando introducimos el contenido inicial al fichero index.h\
```

```
tml
$ echo "<p>This is our initial content</p>" > index.html
$ git add index.html
$ git commit -m "Add initial content"
[main (root-commit) 0d26ac1] Add initial content
 1 file changed, 1 insertion(+)
 create mode 100644 index.html
```

Ahora, vamos a crear una nueva rama llamada `changes` donde haremos algunos cambios que, después, entrarán en conflicto.

```
$ git switch -c changes
Switched to a new branch 'changes'

$ echo "<p>Completely different content</p>" > index.html
$ git commit -am "Edit index.html content to cause conflicts"
[changes e607927] Edit index.html content to cause conflicts
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Con esto hemos creado una nueva rama llamada `changes`, hemos cambiado a ella y hemos cambiado completamente el contenido del fichero `index.html`. Hasta aquí todo bien pero... ¿Qué pasa si antes de fusionar esta rama volvemos a la rama principal y hacemos otros cambios como el siguiente?

```
$ git switch main
Switched to branch 'main'
# Al usar >> lo que hacemos es añadir el contenido al final del fichero
$ echo "<p>New content to the file</p>" >> index.html
$ git commit -am "Append new content to the index file"
[main 4056b12] Append new content to the index file
 1 file changed, 1 insertion(+)
```

En este punto hemos evolucionado el mismo archivo `index.html` tanto en la rama principal como en la rama `changes`. ¿Qué va a pasar cuando intentemos fusionar los cambios de `changes` a `main`? Veamos.

```
# Nos aseguramos que estamos en la rama main
$ git branch --show-current
main

# Fusionamos los cambios de la rama changes en main
$ git merge changes
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
# ;Vaya, tenemos conflictos!
```

Es normal que hayan ocurrido conflictos. Al intentar traer a la rama principal los cambios de la rama `changes`, Git no ha sido capaz de determinar qué cambios son los que deben prevalecer.

En esta situación, si ejecutamos el comando `git status` podremos ver que el fichero `index.html` está en conflicto.

```
$ git status
On branch main
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Unmerged paths:
  (use "git add <file>..." to mark resolution)
    both modified: index.html

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

Como ves, aquí nos da dos opciones: podemos abortar completamente la fusión ejecutando el comando `git merge --abort` o podemos intentar solucionar el conflicto y continuar con la fusión con un `commit`.

Solucionando conflictos

Esta situación es posible que la encontremos frecuentemente en muchos proyectos. Por ello, **hay que evitar alarmarse**. Ahora que sabemos por qué ocurren los conflictos, entendemos que **no es que haya algo mal, es que simplemente debemos resolver cómo fusionar estos cambios**.

Para poder resolver el conflicto, hay que ver el contenido del archivo `index.html` y entender qué es lo que hay que hacer para que la fusión sea correcta.

```
$ git diff
++<<<<< HEAD
+<p>This is our initial content</p>
+<p>New content to the file</p>
+=====
+ <p>Completely different content</p>
++>>>>> changes
```

Como ves el contenido de nuestro archivo tiene unas anotaciones un tanto extrañas que nosotros no hemos incluido en ningún momento. Ha sido Git que, al hacer la fusión, ha intentado separar el conflicto en dos partes.

Por un lado tenemos el contenido del archivo que se encontraba en la rama principal y que está limitado por <<<<< HEAD y ===== y por otro lado tenemos el contenido de la rama changes que está limitado por >>>>> changes (Incoming Change).

Dicho de otra forma, arriba tenemos el contenido que ya existía en la rama de destino de la fusión (la rama main) y debajo de la línea divisoria ===== tenemos el contenido de la rama de origen (la rama changes), que queremos incorporar a la rama main.

Al resolver, deberemos decidir entre:

- Nos quedamos con los cambios de la rama main
- Nos quedamos con los cambios que vienen de la rama changes
- Modificamos los cambios para hacer una fusión personalizada

Por desgracia, a día de hoy, git no tiene un comando para resolver conflictos desde la línea de comandos. Para ello puedes usar una aplicación visual o editor de código como *Visual Studio Code* que, además, te dará una ayuda visual con enlaces directos para resolver y elegir el cambio con el que te quieras quedar.

```
You, seconds ago | 1 author (You) | Accept Current Change | Accept Incoming Change | Accept Both Changes | Compare Changes  
<<<<< HEAD (Current Change)  
<p>This is our initial content</p>  
<p>New content to the file</p> You, an hour ago • Append new content to the file  
=====  
<p>Completely different content</p>  
>>>>> changes (Incoming Change)
```

El editor Visual Studio Code ofrece una experiencia muy buena a la hora de lidiar con conflictos y te ofrece accesos directos para resolverlos

La otra opción es, manualmente, **modificar el fichero para quedarte con el cambio que quieras**. En nuestro caso vamos a conseguir que se mantengan tanto los cambios que teníamos en la rama principal como los que queremos

aplicar, para ello simplemente **eliminamos todas las anotaciones que ha hecho Git** y la dejamos así:

```
$ git diff

diff --cc index.html
index 9e684df,ab7339b..0000000
--- a/index.html
+++ b/index.html
@@@ -1,2 -1,1 +1,3 @@@
+<p>This is our initial content</p>
+<p>New content to the file</p>
+ <p>Completely different content</p>
```

Ahora ya podemos fusionar los cambios de la rama `changes` a la rama `main`, para ello vamos a ver el estado en el que está, luego añadimos a la fase de *staging* los cambios y hacemos commit.

```
$ git status
On branch main
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Unmerged paths:
  (use "git add <file>..." to mark resolution)
    both modified:   index.html

# añadimos el archivo al área temporal
$ git add index.html
# y hacemos commit
$ git commit -m "Merge changes branch to main"
[main 2c72238] Merge changes branch to main

# revisamos otra vez el estado para ver
# que ya no hay conflicto
$ git status
On branch main
nothing to commit, working tree clean
```

Hora de podar: Eliminando ramas

Después de fusionar una rama en otra rama (normalmente, la rama principal) es posible que quieras eliminarla para no dejarla suelta. Para ello puedes usar el comando `git branch` con el parámetro `--delete` o, de forma corta, `-d`.

```
# borramos la rama llamada "mi-primera-rama"
$ git branch --delete mi-primera-rama
Deleted branch mi-primera-rama (was 7c60765) .
```

Si la rama ya ha sido fusionada previamente, entonces todo habrá ido correctamente y nos habrá devuelto un mensaje similar al que tienes arriba... Sin embargo, si la rama no la habías fusionado (*merge*) previamente, entonces te devolverá un error:

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

Creo que el mensaje nos ha devuelto un *spoiler*, directamente. Y es que en el caso que quieras borrar una rama que no ha sido fusionada previamente, deberás usar el parámetro `-D`. Este parámetro le indica a Git que queremos borrar la rama sin importar si ha sido fusionada o no.

```
# borramos la rama llamada "mi-primera-rama"
$ git branch -D mi-primera-rama
Deleted branch mi-primera-rama (was 7c60765) .
```

¿Por qué hace esta comprobación Git? Es sencillo. Borrar una rama que no ha sido fusionada... puede hacerte perder muchas horas de trabajo. Asegúrate de que estás seguro de querer borrarla antes de hacerlo.

¿Cómo eliminar ramas de mi repositorio local que ya no se usan?

Normalmente la idea detrás de las ramas que creamos es que sean fusionadas con otra rama y, por lo tanto, desaparezcan. Cuando su fusión ocurre en un repositorio remoto... ¿qué pasa con los repositorios locales que habían creado esas ramas? Pues que quedan ahí, descolgadas del repositorio remoto.

Estas ramas se pueden, en la jerga de Git, *podar* (en inglés *prune*). Para ver qué ramas serían podadas de las que tenemos en local, y que ya no son necesarias porque en el repositorio remoto ya han sido fusionadas, ejecutamos:

```
# el comando mira qué ramas locales han dejado
# de estar enlazadas con el repositorio remoto origin
# el --dry-run hace que las muestre pero no las elimine
$ git remote prune origin --dry-run
```

```
Pruning origin
URL: git@github.com:midudev/midu.dev.git
* [would prune] origin/feat/add-lighthouse-ci
* [would prune] origin/imgbot
* [would prune] origin/newsletter
```

La opción `--dry-run` nos permite ver qué ramas se eliminarían, pero no las eliminamos. Es un concepto que se utiliza mucho en programación y en la línea de comandos para ver qué resultado tendría una acción sin realizarla. De esta forma, podemos revisar antes las consecuencias que tendría la ejecución.

Ahora que ya hemos visto que es lo que se eliminaría y tenemos claro que sea así, podemos ejecutar el comando `git remote prune` para eliminar todas las ramas que no sean necesarias:

```
# ejecutamos el mismo comando que antes
# pero sin la opción --dry-run
$ git remote prune origin
```

```
Pruning origin
URL: git@github.com:midudev/midu.dev.git
* [pruned] origin/feat/add-lighthouse-ci
* [pruned] origin/imgbot
* [pruned] origin/newsletter
```

Rebase

Ya hemos visto cómo fusionar cambios de una rama en otra. Para ello hemos usado el comando `git merge`.

Sin embargo existe otro comando que nos permite conseguir el mismo resultado a base de *reordenar* el historial de commits. Este comando es `git rebase`.

Rebase es, seguramente, uno de los comandos más complejos de Git (y más peligroso). Por ello, vamos a dedicarle un capítulo entero para explicarlo y que lo entiendas bien.

¿Qué es el rebase?

El rebase es una **operación que nos permite reescribir el historial de commits** de una rama.

Esto significa que podemos **añadir, mover, ordenar o eliminar commits directamente de un historial de commits**. Es como tener el superpoder de viajar en el tiempo y cambiar el pasado, presente o futuro de nuestra rama.

¿Y si ya tenemos `git merge`? ¿Para qué necesitamos `git rebase`? La principal razón es la de **mantener el historial de commits de tu rama** lo más limpio, ordenado y lineal posible.

¿Cómo usar `git rebase`?

Para usar `git rebase` debemos seguir los siguientes pasos:

1. **Seleccionar la rama a la que queremos añadir commits**: para ello debemos situarnos en la rama a la que queremos añadir commits. Por ejemplo, si queremos añadir commits a la rama `main`, debemos situarnos en ella:

```
$ git switch main
```

2. **Ejecutar `git rebase`**: una vez situados en la rama a la que queremos añadir commits, ejecutamos `git rebase` y le pasamos como parámetro la rama de la que queremos añadir commits:

```
$ git rebase feature
```

3. **Resolver los conflictos**: si hay conflictos, en este paso deberemos resolverlos.

Resolviendo conflictos con `git rebase`

Al hacer un `git rebase` es posible que tengas conflictos. Como estamos reescribiendo el historial de commits, es posible que dos commits diferentes modifiquen la misma línea de un mismo archivo y Git no sabe qué cambio aplicar.

Git te lo indicará con el siguiente mensaje impreso en la terminal:

```
error: could not apply af42187... something to add to patch A
```

Cuando hayas resuelto este problema, ejecuta “`git rebase --continue`”. Si prefieres saltar este parche, ejecuta “`git rebase --skip`” en su lugar.

Para volver a la rama original y detener el rebase, ejecuta “`git rebase --abort`”.

Could not apply fa39187f3c3dfd2ab5faa38ac01cf3de7ce2e841... Change fake file

Aquí, Git te está indicando qué commit está causando el conflicto (fa39187).

Tienes tres opciones:

1. Puedes ejecutar `git rebase --abort` para deshacer completamente el rebase. Git volverá al estado de tu rama tal como era antes de que se llamara a `git rebase` y podrás empezar de cero.
2. Con `git rebase --skip` puedes saltar completamente el commit. Esto significa que ninguno de los cambios introducidos por el commit problemático se incluirá. Ojo porque, si haces esto, estás perdiendo cambios.
3. Arreglar el conflicto. Básicamente, modificar los archivos para que el conflicto deje de existir. Una vez hecho esto, debemos ejecutar `git rebase --continue` para que Git continúe procesando el resto del rebase.

Rebase interactivo

El rebase interactivo es una herramienta que nos permite reescribir el historial de commits de una rama de forma interactiva.

Para ello debemos añadir el parámetro `-i` o `--interactive` cuando usemos el comando:

```
$ git rebase --interactive
```

Esto creará un archivo en el editor de texto que tengas configurado por defecto. En este archivo verás una lista de commits de la rama actual, ordenados cronológicamente, que podrás editar:

```
pick 2ee9f59 Fix linter errors
pick c29f6c8 Remove not needed countdown
pick ee4b85c add official color
pick 82ff110 add iframe twitch
pick 1a3e40d iframe centered
pick ca745cf change script
pick 0e3d6f8 fix eslint errors
pick bde26c5 fix: fix lint errors
pick 490df4c fix: iframe in mobile
pick cabb633 feat: show twitch iframe only if midudev is live on twitch
pick 96ce7b6 feat: add animated twitch iframe border

# Rebase 8d7813d..8686bad onto 8d7813d (11 commands)
```

En este archivo, cada línea representa un commit. En la primera columna, verás un comando que indica qué hacer con el commit. Los comandos más comunes son:

- p, pick (commit): usa el commit tal como está
- r, reword (commit): usa el commit pero cambia el mensaje
- e, edit (commit): usa el commit pero permite editarlo
- s, squash (commit): usa el commit pero lo fusiona con el anterior
- f, fixup (commit): usa el commit, lo fusiona con el anterior y descarta el mensaje
- x, exec (command): ejecuta el comando en el *shell*
- b, break: para aquí para continuar el rebase más tarde con ‘git rebase – continue’
- d, drop (commit): borra el commit
- l, label (label): etiqueta el HEAD actual con un nombre
- t, reset (label): resetea la etiqueta HEAD a otro nombre
- m, merge [-C (commit) | -c (commit)] (label) [# (oneline)]: crea un commit de *merge* usando el mensaje del commit original (o el *oneline*,

si no se especificó ningún commit original). Usa `-c` (commit) para reescribir el mensaje del commit.

Por defecto el comando es `pick` y, por tanto, se usa el commit tal como está. Si quieras cambiar el comando de un commit, simplemente cambia la palabra por el comando que quieras usar (puedes usar la letra o el comando completo).

Cambiar el mensaje de un commit

Si quieras cambiar el mensaje del commit `c29f6c8`, simplemente cambia la palabra `pick` por `reword`:

```
pick 2ee9f59 Fix linter errors
reword c29f6c8 Remove not needed countdown
...
```

Guarda el archivo y Git te abrirá un nuevo fichero para que indiques el nuevo mensaje del commit.

Reordenando commits

También puedes cambiar el orden de los commits. Por ejemplo, si quieres que el commit `ee4b85c` se haga antes que el commit `82ff110`, simplemente cambia el orden de las líneas:

```
pick 2ee9f59 Fix linter errors
pick c29f6c8 Remove not needed countdown
pick 82ff110 add iframe twitch # <-- arriba
pick ee4b85c add official color # <-- abajo
```

Guarda el archivo y Git reordenará los commits.

Borrar commits

Si quieres borrar un commit, simplemente cambia el comando `pick` por `drop`:

```
pick 2ee9f59 Fix linter errors
pick c29f6c8 Remove not needed countdown
drop ee4b85c add official color # <-- este commit se borrará
pick 82ff110 add iframe twitch
```

Guarda el archivo y Git borrará el commit.

Fusionar commits

Si quieres fusionar dos commits, simplemente cambia el comando `pick` por `squash`:

```
pick 2ee9f59 Fix linter errors
pick c29f6c8 Remove not needed countdown
squash ee4b85c add official color # <-- este commit se fusionará con el \
anterior
squash 82ff110 add iframe twitch # <-- este commit se fusionará con el \
anterior
```

Te abrirá un nuevo archivo para que indiques el mensaje del commit resultante de la fusión.

```
# This is a combination of 3 commits.
# This is the 1st commit message:
add iframe twitch

# This is the commit message #2:
add official color

# This is the commit message #3:
Remove not needed countdown

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

Guarda el archivo y Git fusionará los commits con el nombre de commit que hayas indicado.

Separar un commit en dos o más

Si quieres separar un commit en dos o más, simplemente cambia el comando `pick` por `edit`:

```
pick 2ee9f59 Fix linter errors
pick c29f6c8 Remove not needed countdown
edit ee4b85c add official color # <-- este commit se podrá editar
pick 82ff110 add iframe twitch
```

Ahora, podemos hacer un `git reset` para deshacer los cambios del commit `ee4b85c` y hacer un nuevo commit con los cambios que queramos usando `git add` y `git commit`:

```
$ git reset HEAD^
$ git add file1
$ git commit -m 'Update file1 formatting'
$ git add file2
$ git commit -m 'Add file2 feature'
$ git rebase --continue # continuamos con el rebase
```

Te recomiendo que antes de aplicar el rebase, pruebes a jugar con todos estos comandos en un repositorio de prueba para que veas cómo funcionan y qué problemas puedes encontrarte. Jugar con el historial siempre es peligroso, ya que el orden de los commits es muy importante para Git.

Abortar un rebase

Si por cualquier motivo quieres abortar un rebase, simplemente ejecuta `git rebase --abort`:

```
$ git rebase --abort
```

Esto te devolverá el historial a su estado original, eliminará la carpeta `.git/rebase-apply` y te dejará en el commit en el que estabas antes de empezar el rebase.

No uses `git rebase` para esto

A veces tenemos la tentación de usar `rebase` para *arreglar* el historial. ¿Se te ha escapado un token de una API? Vamos a reescribir el historial para borrarlo. ¿Te has equivocado en un commit y ya has hecho `push`? Vamos a reescribir el historial para arreglarlo...

Pues no, no lo hagas. En el caso del token de una API, lo mejor que puedes hacer, es generar o conseguir un nuevo token. Una vez que un token ha llegado a un repositorio remoto, especialmente si es público en GitHub, puede ser demasiado tarde ya que existen *bots* que lo detectan y lo reportan.

En el caso de un commit equivocado, depende un poco de la situación. Si ya has hecho `push` y es una rama en la que estás trabajando, nadie más trabaja en ella y, sobretodo, no es la rama principal, entonces puedes hacer `rebase`. En cualquier otro caso, lo mejor es que uses `git revert`.

Vamos, **no uses `git rebase` simplemente porque quieres dejar el historial bonito.**

Los peligros del rebase

Reescribir la historia suena peligroso. Y lo es. Asegúrate de entender qué estás haciendo antes de usar el comando `git rebase`.

1. **Perdida de trabajo:** uno de los mayores peligros de usar `git rebase` es la posibilidad de perder trabajo. Al reescribir la historia del repositorio, es posible eliminar commits que todavía son necesarios en tu rama actual. Si no tienes cuidado, es posible que acabes perdiendo trabajo importante sin darte cuenta.
2. **Conflictos silenciosos:** otro peligro de usar `git rebase` es la posibilidad de crear conflictos al integrar commits. Al reescribir la historia del repositorio, es posible que `git rebase` cree conflictos con commits que ya hayan sido integrados pero que no recibas avisos de que esto ha ocurrido.
3. **Una historia artificial:** al reescribir la historia de un repositorio, es posible que se pierda información sobre el desarrollo de la rama. ¿De dónde salieron los commits que tiene la rama? ¿En qué orden fueron escritos? Sí, la historia queda más lineal y limpia pero es artificial, ya que no refleja exactamente el trabajo del equipo.

Merge vs Rebase

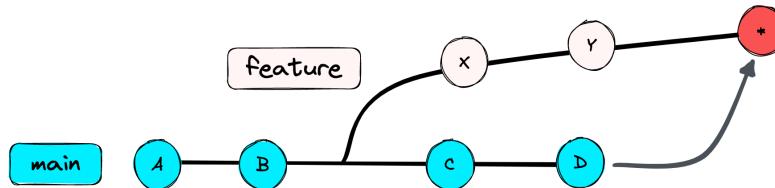
Merge y *Rebase* son dos técnicas comunes en Git que se utilizan para integrar cambios de una rama en otra. Ambas técnicas pueden **cumplir el mismo fin** pero tienen importantes diferencias que debes tener en cuenta al decidir cuál usar.

Las diferencias

Merge implica fusionar dos ramas en una sola, creando un nuevo punto de unión en el historial del repositorio. Esto significa que al hacer un *merge* se unen dos ramas.

Si el historial de commits está alineado, el *merge* no crea un nuevo commit y usa el *fast-forward* para avanzar la rama a la que se le están incorporando los cambios.

Si no es posible, se crea un nuevo commit de tipo *merge* para incorporar los cambios de la rama.

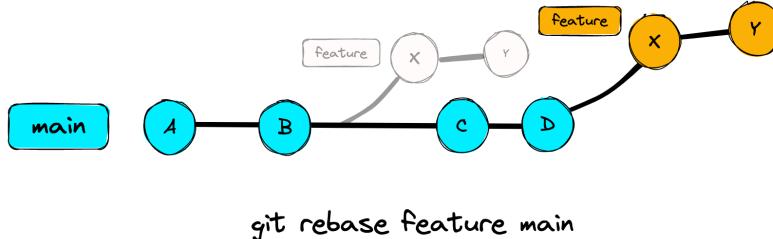


`git merge feature main`

Imagina una rama “feature” como el de la imagen. Queremos incorporar los cambios de “main” a “feature”. Al ejecutar “git merge feature main”, nos crearía un commit (el de asterisco) en la rama donde queremos incorporar los cambios de la otra.

Rebase implica reordenar los cambios, lo que significa que se replican los cambios de la rama en la que se está trabajando sobre la otra. Esto resulta en

una sola línea de *commits* en el historial, que muestra una secuencia lineal de cambios.



Imagina una rama “feature”. Al ejecutar “git rebase main feature” se moverían los commits de la rama “feature” a la rama “main”.

merge empuja el historial hacia adelante, mientras que *rebase* reordena el historial de commits sin necesidad de crear nuevos commits (simplemente aplica los que ya existen).

¿Qué ventajas y desventajas tienen?

Antes de elegir entre *merge* y *rebase*, debes tener en cuenta las ventajas y desventajas de cada uno.

Ventajas de *merge*

- Es más fácil de entender para los nuevos usuarios de Git.
- Aunque hace que el historial este más sucio de commits de *merge*, te da una visión real de lo que ha ido ocurriendo en la rama.
- No existe ningún peligro ya que no reescribe el historial de commits. Lo peor que puede pasar es que tengas que resolver conflictos.

Desventajas de *merge*

- Si usas *merge* con frecuencia de una rama muy activa, tu historial de commits se llenará constantemente de commits de *merge* que no aportan valor.
- Al generar commits de *merge*,añades carga al historial y en repositorios grandes puede ralentizar el proceso de clonado.

Ventajas de *rebase*

- El historial de commits queda más limpio y es más fácil de seguir la evolución del código ya que no se generan commits de `merge` innecesarios.

Desventajas de *rebase*

- Es más difícil de entender para los nuevos usuarios de Git.
- Reescribir el historial de commits es peligroso. Puedes causar problemas muy graves si no tienes cuidado.
- Pierdes información sobre cuando se han incorporado los cambios en la rama.

¿Cuándo usar cada uno?

Ahora que ya conoces las ventajas y desventajas de cada uno, es hora de elegir cuál usar. Y, para mí, la respuesta es simple: **usa *merge* para integrar cambios en una rama principal y usa *rebase* para integrar cambios en una rama de desarrollo.**

O, dicho de otro modo: nunca usaría *rebase* en una rama principal. Ni **tampoco en una rama pública donde más personas están trabajando.**

Otra regla, bastante acertada, es usar *rebase* sólo en código que todavía no has compartido en un repositorio remoto. Nunca, nunca, **nunca hagas rebase de algo que has compartido**, ya que eso reescribiría el historial de commits y causaría problemas para los demás.

Este punto está basado en mi experiencia personal. Otras personas consideran que lo correcto es usar siempre *rebase* en lugar de *merge*, ya que así el historial de commits queda más limpio. Sin embargo, yo prefiero usar *merge* para integrar cambios en una rama pública, ya que así puedo ver fácilmente cuándo se han incorporado los cambios en la rama y evito problemas de reescritura del historial de commits al trabajar en equipo. Además, considero que **un historial de commits limpio es menos importante que tener un historial de commits que refleje la realidad.**

Trabajando con Git de forma remota

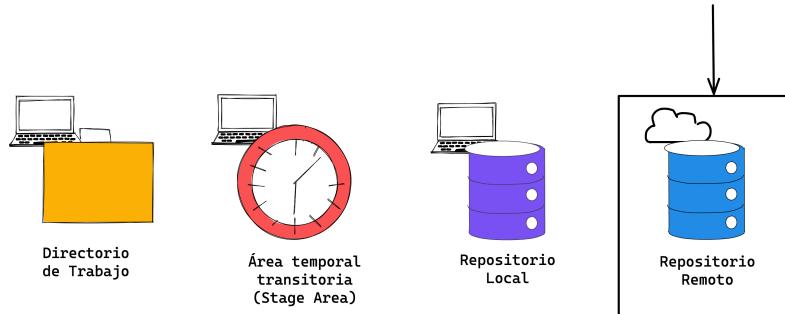
Hasta ahora hemos visto cómo podemos trabajar con repositorios locales. Esto quiere decir que los cambios se quedan en nuestro ordenador y **otras personas no pueden verlos**. Aunque esto puede tener cierta utilidad, esto no es lo que queremos.

Para que un equipo de personas pueda colaborar en un mismo código base, se necesita una forma de comunicar los cambios entre ellos. Y esto **lo logramos con los repositorios remotos**.

Los repositorios remotos son repositorios que están hospedados en un servidor y que servirá de punto de sincronización entre diferentes repositorios locales.

Se puede crear y levantar tu propio servidor de Git para lograr esto pero en este libro **vamos a enfocarnos a lograr esto usando GitHub**.

GitHub es el proveedor de repositorios en la nube para hospedar código fuente más utilizado en la actualidad, especialmente en el mercado occidental.



**Los repositorios remotos no están en nuestra máquina.
Están hospedados en un servidor externo pero podemos sincronizar nuestros cambios cuando queramos.**

Creando un repositorio remoto en GitHub

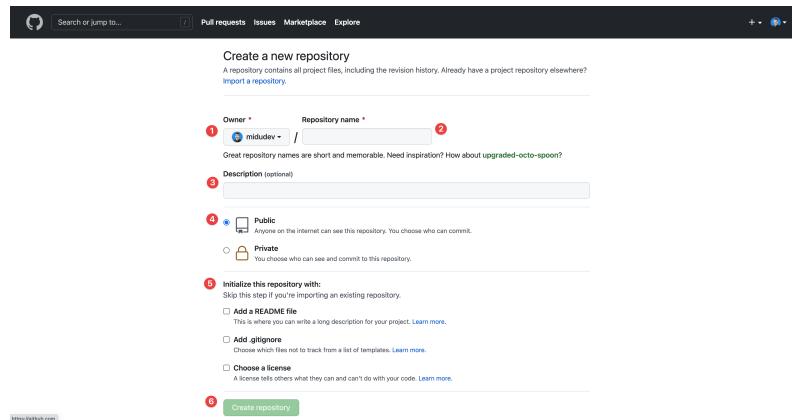
Como hemos dicho antes, vamos a usar *GitHub* para crear nuestro primer repositorio remoto. **Es completamente gratis** y nos permite crear un número ilimitado de repositorios, tanto públicos como privados.

Aunque *GitHub* es actualmente el servicio de hospedaje de repositorios remotos más utilizado, existen otras alternativas igual de válidas como [GitLab](#) o [BitBucket](#) (ambos compatibles con *Git*).

Para crear un nuevo repositorio remoto en GitHub accede a la página <https://repo.new/>.

Al acceder te pedirá tus credenciales para poder continuar. Si tienes una cuenta ya creada, podrás iniciar sesión con ella. De lo contrario, créate una cuenta para poder continuar.

Una vez que hayas creado tu cuenta, podrás crear un repositorio nuevo y te aparecerá esta pantalla:



En esta pantalla podremos crear nuestro repositorio remoto

1. **Es el dueño del repositorio.** Por defecto aparece tu cuenta de usuario pero también podemos crear organizaciones que sean dueñas de

repositorios.

2. **El nombre del repositorio.** Este nombre será el que aparezca en la URL del repositorio y el que usaremos para poder acceder a él más adelante. El nombre debe ser único entre los repositorios que ya tengamos pero no entre todos los que existen en GitHub.
3. **Una descripción corta** sobre qué trata el repositorio.
4. **El tipo de visibilidad del repositorio.** Puedes elegir entre público o privado. Ten en cuenta que si es público, cualquier persona podrá descubrir el repositorio, ver su código e incluso intentar contribuir. Si no quieres que nadie pueda verlo, ponlo en privado (puedes cambiarlo más adelante).
5. Puedes iniciar tu repositorio con un archivo llamado *README.md* que contenga una breve descripción del repositorio. También puedes añadir desde el inicio un archivo `.gitignore` para evitar hacer seguimiento de archivos que no quieras que se suban al repositorio y añadir una licencia para determinar qué uso puede hacerse del código.
6. Cuando ya lo tengas todo preparado, es hora de darle a *Create Repository*.

Por favor, no descuides seleccionar una licencia. Si no seleccionas ninguna significa que el código que subes, aunque visible, tiene todos los derechos reservados. Por lo tanto, nadie podría modificar ni redistribuir el código sin tu permiso explícito. Igual es lo que quieras, pero lo ideal es que tengas una licencia en tu repositorio para evitar un malentendido.

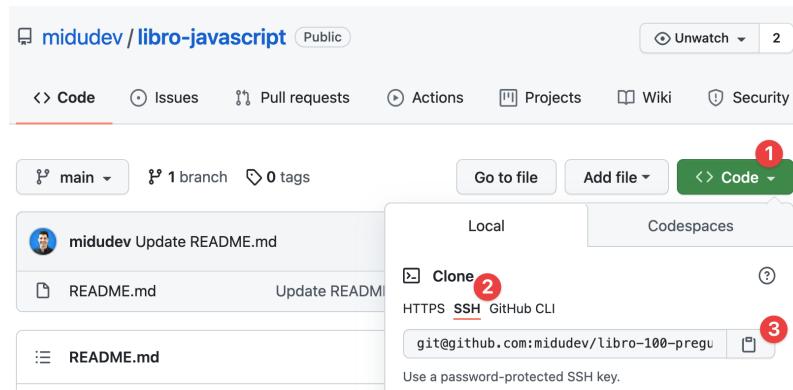
¿Te interesa saber cómo **crear un repositorio remoto desde la línea de comandos**? ¡Se puede! Revisa la sección sobre **GitHub CLI** en este libro, que te explico cómo.

Clonando un repositorio remoto ya creado previamente

Imagina que ya existe un repositorio remoto creado y quieres trabajar con él en tu ordenador. ¿Cómo puedes crear un repositorio local en tu ordenador con los ficheros y ramas del repositorio remoto para poder trabajar con él?

A esta acción se le llama *clonar*. Se llama así porque, literalmente, **lo que hacemos es una copia exacta del repositorio remoto en nuestra máquina**. Todo el historial de commits y todas las ramas son descargadas en nuestro ordenador. Recuerda que, como ya hemos comentado anteriormente, **Git es un sistema distribuido**, lo que significa que podemos tener copias completas de los repositorios en diferentes lugares y sincronizarlos entre ellos.

Para clonar un repositorio remoto necesitamos saber su dirección. Esta dirección puede ser su dirección *HTTPS* o *SSH*. Normalmente, **la forma preferida, debería ser usando SSH**.



Primero pulsas el botón Code. En el diálogo, selecciona SSH si no está ya activado. Después, pulsa el icono del lateral para que se copie la dirección en tu portapapeles

Ahora que tenemos la dirección, es el momento de usar el comando `git clone` para clonar el repositorio remoto en nuestra máquina.

```
# Para clonar un repositorio remoto con la dirección SSH
$ git clone git@github.com:midudev/libro-javascript.git
```

```
Cloning into 'libro-javascript'...
remote: Enumerating objects: 12, done.
remote: Counting objects: 100% (12/12), done.
remote: Compressing objects: 100% (7/7), done.
remote: Total 12 (delta 2), reused 3 (delta 0), pack-reused 0
Receiving objects: 100% (12/12), done.
Resolving deltas: 100% (2/2), done.

# Para clonar un repositorio remoto con la dirección HTTPS
$ git clone https://github.com/midudev/libro-javascript.git
```

Repasemos un poco el resultado de ejecutar esta línea:

1. Hemos ejecutado el comando `git clone` y le hemos indicado que queremos clonar el repositorio situado en `git@github.com:midudev/libro-javascript.git` (dirección SSH).
2. Git ha clonado el repositorio remoto en nuestro ordenador. Ha creado una carpeta `libro-javascript` en el lugar de la línea de comandos que nos encontrábamos.
3. En esa carpeta, se ha creado una copia del repositorio remoto que, ahora, es un repositorio local.
4. En el repositorio local ha dado nombre a una carpeta llamada `.git` que contiene toda la información sobre el repositorio (versiones, commits, ramas...)
5. Nos ha dejado un directorio de trabajo listo con la rama principal, listas para ser usada.

Por defecto al clonar un repositorio se crea una carpeta con el mismo nombre que el repositorio remoto. Si quieres que el nombre de la carpeta sea diferente puedes indicarlo:

```
# clonamos el repositorio midudev/blog
# y lo hacemos en el directorio blog-midudev
git clone git@github.com:midudev/blog.git blog-midudev
```

¿No tienes configurada la llave SSH para tu ordenador? ¡No te preocupes! Tienes un capítulo completo en este libro llamado *Configurando la conexión SSH con GitHub* que te puede guiar paso a paso para que lo hagas y entiendas qué es y cómo funciona la autenticación de SSH.

Aunque a la hora de explicar la clonación de repositorios nos hemos enfocado en repositorios remotos, debes saber que también puedes usar `git clone` para clonar un repositorio local a otro destino. **No es muy usado** pero a veces queremos trabajar con el mismo repositorio en dos directorios de trabajo y hacerlo puede ayudarte.

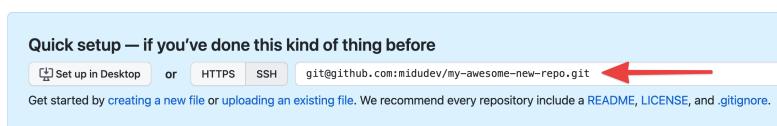
¿Cómo enlazar un repositorio local con un repositorio remoto?

A veces empezamos un desarrollo con un repositorio local y luego, más adelante, queremos subir nuestro código al repositorio remoto. Para ello, tenemos que hacer una sincronización entre los dos repositorios.

Para conectar nuestro repositorio local con el repositorio remoto debemos usar el comando `git remote add` y le pasariamos dos parámetros. El primero sería el *alias del repositorio remoto* y el segundo sería la *dirección del mismo repositorio remoto*. Sería algo así: `git remote add <alias> <dirección>`.

Como *alias* podemos usar cualquier nombre que queramos, pero por defecto usamos `origin` para indicar que el repositorio remoto que estamos sincronizando es el remoto principal.

Sobre la **dirección del repositorio** remoto... Si has seguido los pasos anteriores para crear un repositorio remoto, habrás llegado a una pantalla similar a esta:



Cada vez más, GitHub está utilizando las direcciones SSH por defecto. Por ello, es buena idea que siempre intentes utilizar estas

De esta forma, para añadir un repositorio remoto a nuestro repositorio local, tenemos que ejecutar:

```
# añadimos como origin la dirección ssh del repositorio remoto
$ git remote add origin git@github.com:midudev/my-awesome-new-repo.git
```

Ten en cuenta que un repositorio local, en realidad, **puede tener enlazados tantos repositorios remotos como queramos**. Normalmente sólo será uno, como en este caso, pero más adelante veremos que es útil tener también otros repositorios remotos en algunos casos.

Para saber qué repositorios remotos tiene enlazados nuestro repositorio local, podemos ejecutar el comando `git remote`:

```
# con -v conseguimos una salida más detallada
git remote -v

origin  git@github.com:midudev/my-awesome-new-repo.git (fetch)
origin  git@github.com:midudev/my-awesome-new-repo.git (push)
```

Traer los cambios del repositorio remoto a mi repositorio local

Cuando trabajamos en equipo, es muy común que haya cambios en el repositorio remoto que no tenemos en nuestro repositorio local. Para traer esos cambios a nuestro repositorio local, tenemos que ejecutar el comando `git pull` y le pasaremos dos parámetros:

- El primer parámetro sería el *alias* del repositorio remoto.
- El segundo sería el nombre de la rama sobre la que queremos traer los cambios.

Por ejemplo, para traer los cambios del repositorio remoto *origin* y la rama *main*, deberíamos ejecutar lo siguiente:

```
# traemos los cambios del repositorio remoto
# con alias origin y de la rama main
$ git pull origin main

From github.com:midudev/miduconf-website
 * branch           main      -> FETCH_HEAD
Auto-merging src/components/Principal.astro
Merge made by the 'ort' strategy.
 src/components/Principal.astro          |  27 +++++-
 .../logos/PlainConcepts.astro          |  61 ++++++-----
 src/global.css                          |  20 +////
 3 files changed, 92 insertions(+), 16 deletions(-)
```

¿Qué pasa si hay conflictos?

A veces, cuando hacemos un `git pull` nos encontramos con que hay conflictos entre los cambios que tenemos en nuestro repositorio local y los cambios que hay en el repositorio remoto. Esto puede ocurrir por varias razones:

- Hemos hecho cambios en el mismo archivo que alguien ha modificado, en las mismas líneas.
- Hemos eliminado un archivo que alguien ha modificado.

- Hemos creado un archivo y alguien ha creado otro con el mismo nombre.

Sea como sea, en este punto, **Git no sabe qué hacer** y nos deja elegir qué cambios queremos conservar. Para ello, debemos resolver los conflictos manualmente y luego hacer un `git add` y un `git commit` para que los cambios queden reflejados en nuestro repositorio local.

En la sección de *Ramas* tienes una sección completa donde te explico cómo resolver conflictos.

La estrategia de merge por defecto

Desde la versión 2.27.0 de `git` cuando hacemos un `git pull` te muestra una advertencia si no tienes configurada qué estrategia debe seguir el comando a la hora de traer los cambios del repositorio remoto.

Esto es porque antes de esta versión, **la estrategia de merge por defecto era hacer un merge** si no se podía hacer un *fast-forward*. A partir de ahora, debemos indicar nosotros qué estrategia queremos que siga `git` para traer los cambios del repositorio remoto.

```
$ git pull origin main

From github.com:midudev/miduconf-website
 * branch            main      -> FETCH_HEAD
hint: You have divergent branches and need to specify how to reconcile \
them.
hint: You can do so by running one of the following commands sometime b\
efore
hint: your next pull:
hint:
hint:   git config pull.rebase false    # merge
hint:   git config pull.rebase true     # rebase
hint:   git config pull.ff only        # fast-forward only
hint:
hint: You can replace "git config" with "git config --global" to set a \
default
hint: preference for all repositories. You can also pass --rebase, --no\
-rebase,
hint: or --ff-only on the command line to override the configured defau\
lt per
```

```
hint: invocation.  
fatal: Need to specify how to reconcile divergent branches.
```

Como ves, tenemos tres opciones:

- `git config pull.rebase false` para hacer un *merge* cuando no se pueda hacer un *fast-forward* de la rama remota en nuestro repositorio local.
- `git config pull.rebase true` para hacer un *rebase* cuando no se pueda hacer un *fast-forward* de la rama remota en nuestro repositorio local.
- `git config pull.ff only` para que sólo acepte un *fast-forward* al traer los cambios, de lo contrario nos dará un error.

Cualquiera de las tres opciones es válida. Si no quieres complicarte, puedes usar la primera, ya que es la que se comporta como antes de la versión 2.27.0 de `git`. Si quieres que tu historial de *commits* sea más limpio, puedes usar la segunda opción. Y si quieres tener más control sobre lo que está pasando, puedes usar la tercera opción.

Recuerda que puedes adoptar una estrategia de merge o rebase para todos tus repositorios o para cada uno de ellos de forma independiente usando la opción `--global` o no. Esto dependerá de tus preferencias.

Escribiendo en el repositorio remoto

`git push` es un comando que permite enviar los cambios de un repositorio local a un repositorio remoto. Para ello, tenemos que ejecutar el comando `git push` y le pasariamos dos parámetros:

- El primer parámetro sería el *alias* del repositorio remoto.
- El segundo sería el nombre de la rama sobre la que queremos enviar los cambios.

Por ejemplo, para enviar nuestros *commits* al repositorio remoto *origin* y la rama *main*, deberíamos ejecutar lo siguiente:

```
# subimos los cambios del repositorio local  
# al repositorio remoto con alias origin y a la rama main  
$ git push origin main
```

Esto hará que nuestros cambios locales se reflejen en el repositorio remoto y otras personas podrán verlos y sincronizar sus repositorios locales para trabajar con ellos.

Más adelante veremos cómo podemos enviar una rama distinta a la principal. Por ahora no nos preocuparemos por ello.

¡No me deja hacer push! Me dice que ha sido rechazado

Como hemos visto en Git tenemos por un lado repositorios locales y remotos. Cuando hacemos `push`, estamos intentando llevar nuestros cambios locales a un repositorio remoto para grabarlos pero a veces **puede arrojarnos un error en el que dice que ha rechazado la actualización**.

Normalmente esto ocurre porque nuestro repositorio local no tiene cambios que han ocurrido en el repositorio remoto. Estos cambios pueden ser nuevos

commits o que el historial de cambios de la rama han cambiado (se ha sobrescrito el historial).

El error en cuestión se vería así:

```
$ git push origin main

To https://github.com/midudev/my-repo.git
! [rejected] main -> main (non-fast-forward)
error: failed to push some refs to 'https://github.com/midudev/my-repo.\n
git'
hint: Updates were rejected because the tip of your current branch is b\
ehind
hint: its remote counterpart. Integrate the remote changes (e.g.
hint: 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for detail\
ls.
```

Aunque podrías usar el parámetro `-f` para forzar el *push* y saltar este error, **es mejor no hacerlo si no sabes por qué estás teniendo este problema**.

En realidad el propio error ya nos da la explicación de por qué no se ha podido enviar el cambio: tu repositorio local no tiene cambios que han ocurrido en el repositorio remoto.

Además, nos da una posible solución: tenemos que integrar los cambios del repositorio remoto en nuestro repositorio local con `git pull` antes de poder enviar los cambios.

¿Por qué pasa esto? Básicamente el historial que tienes en tu repositorio local es, actualmente, incompatible con el remoto. Lo podríamos ver en este ejemplo:

```
A --- B --- C --- D (repositorio remoto)
A --- B --- E           (tu repositorio local)
```

Tu repositorio local no tiene los *commits* `C` y `D` y, por lo tanto, al intentar enviar `E`... ¿cómo lo podría hacer? Cuando el *commit* modifica ficheros que ya han sido modificados en *commits* que te faltan, Git dejaría el repositorio

remoto con conflictos porque es incapaz de saber cómo tiene que solucionar esas divergencias.

Piensa que conflictos sólo pueden existir en repositorios locales, así que tiene que sincronizar los cambios localmente, arreglar los conflictos que existan y entonces subirlos al remoto, donde ya no existirá ningún conflicto.

Trabajando con ramas en remoto

Llevar cambios locales al repositorio remoto está muy bien pero no vamos a querer hacerlo siempre directamente contra la rama principal. Lo que vimos en la sección de *Ramas* nos va a servir también aquí pero, esta vez, para trabajar en remoto.

Creando una rama remota

Para crear una rama remota, sólo tenemos que crear la rama en nuestro repositorio local y luego enviarla al repositorio remoto.

```
# Creamos una rama en el repositorio local
$ git switch -c website
Switched to a new branch 'website'

# Enviar la rama a nuestro repositorio remoto
# para ello usamos git push:
# git push <alias-repositorio> <rama>
$ git push origin website

Total 0 (delta 0), reused 0 (delta 0), pack-reused 0
remote:
remote: Create a pull request for 'website' on GitHub by visiting:
remote:     https://github.com/midudev/manual-git/pull/new/website
remote:
To github.com:midudev/manual-git.git
 * [new branch]      website -> website

# Ten en cuenta que si intentas enviar una rama
# que no existe en el repositorio local
# tendrás un error:
$ git push origin rama-que-no-existe

error: src refspec rama-que-no-existe does not match any
error: failed to push some refs
```

Como ves tenemos un mensaje sobre crear una Pull Request. Lo ignoramos por ahora, pero más adelante explicamos esto en detalle.

Ya tenemos nuestra rama en el repositorio remoto. Ahora, podemos empezar a crear commits en nuestro repositorio local y enviarlos al repositorio

remoto a la rama que hemos creado.

```
# Revisamos que estamos de forma local en la rama
$ git branch --show-current
# Creamos tres archivos vacíos
# Y vamos a hacer commit de forma separada
# de cada uno de ellos
$ touch index.html styles.css app.js
$ git add index.html
$ git commit -m "Add index.html"
$ git add styles.css
$ git commit -m "Add styles"
$ git add app.js
$ git commit -m "Add JavaScript"
# Enviamos los cambios a la rama remota
$ git push origin website

Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 10 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 276 bytes | 276.00 KiB/s, done.
Total 3 (delta 1), reused 1 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To github.com:midudev/manual-git.git
  39af206..2bd2e0b  website -> website
```

Integrando cambios en otra rama

Una vez que hemos terminado de trabajar en nuestra rama, vamos a querer integrar los cambios en la rama principal. Tenemos diferentes formas de hacerlo pero vamos a ver las dos más habituales: haciendo el *merge* en local y enviar los cambios al repositorio remoto o enviando una *pull request*.

Integrando los cambios localmente y enviándolos

Para integrar los cambios de una rama en otra, tenemos que hacer un *merge* de la rama que queremos integrar en la rama principal. En nuestro caso, la rama `website` en la rama `main`.

```
# Volvemos a la rama principal
$ git switch main
Switched to branch 'main'

# Integrar los cambios de la rama website
# en la rama principal main
$ git merge website
```

```
Updating 39af206..2bd2e0b
Fast-forward
 index.html | 0
 styles.css  | 0
 app.js      | 0
 3 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 index.html
 create mode 100644 styles.css
 create mode 100644 app.js
```

```
# Enviar los cambios a la rama principal
$ git push origin main
```

Nota: Si intentas hacer un *merge* de una rama que no está actualizada con la rama principal, Git te dará un error. Para solucionarlo, tendrás que actualizar la rama que quieras integrar con la rama principal.

Existe otro comando para integrar cambios de una rama en otra: `git rebase`. En este caso, no se crea un commit de *merge* sino que se crean los commits de la rama que queremos integrar en la rama principal. Tienes un capítulo completo dedicado a *Rebase* más adelante.

Integrando los cambios con una Pull Request

Otra forma de integrar los cambios de una rama en otra es enviando una *Pull Request* desde el repositorio remoto. Esto es muy útil cuando trabajamos en equipo y queremos que otra persona revise los cambios que hemos hecho antes de integrarlos en la rama principal.

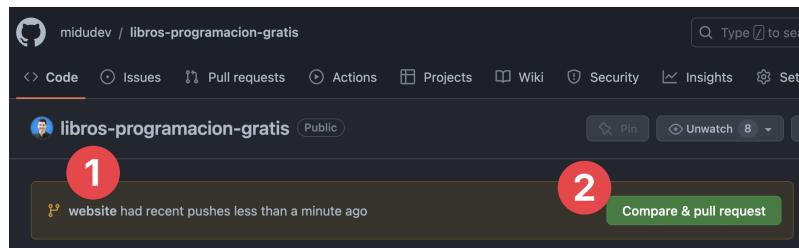
A las *Pull Request* se las abrevia en ocasiones como *PR*. Es posible que encuentres esa notación también en el libro.

Para lograrlo debes haber enviado desde tu repositorio local la rama que quieras integrar en la rama principal. En nuestro caso, la rama `website`.

```
# Enviamos los cambios locales de la rama 'website'
# al repositorio remoto 'origin'
# a la rama 'website'
$ git push origin website
```

En GitHub se le llama *Pull Request* a la petición de integrar cambios de una rama en otra. Este nombre puede cambiar en otros hostings de código. Por ejemplo, en GitLab se le llama *Merge Request*.

Para enviar una *Pull Request* desde GitHub, tenemos que ir a la página de nuestro repositorio y seleccionar la rama que queremos integrar en la rama principal.

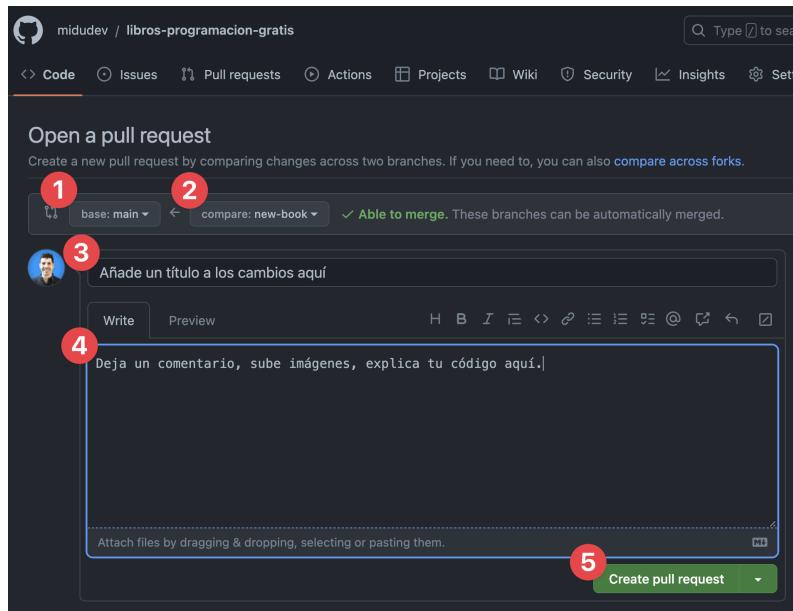


Al entrar al repositorio remoto en GitHub, nos aparecerá un banner que nos indica que una rama ha recibido cambios. El punto 1 nos indica la rama que ha sido actualizada recientemente, el punto 2 es el botón que podemos pulsar para iniciar el proceso de “Pull Request”

Si pulsamos el botón de *Compare & pull request*, nos aparecerá una página donde podemos seleccionar la rama principal con la que queremos integrar los cambios.

Una vez que hemos seleccionado la rama, nos aparecerá una página con los cambios que se van a integrar en la rama principal. Si estamos de acuerdo con los cambios, podemos hacer clic en el botón *Create pull request*.

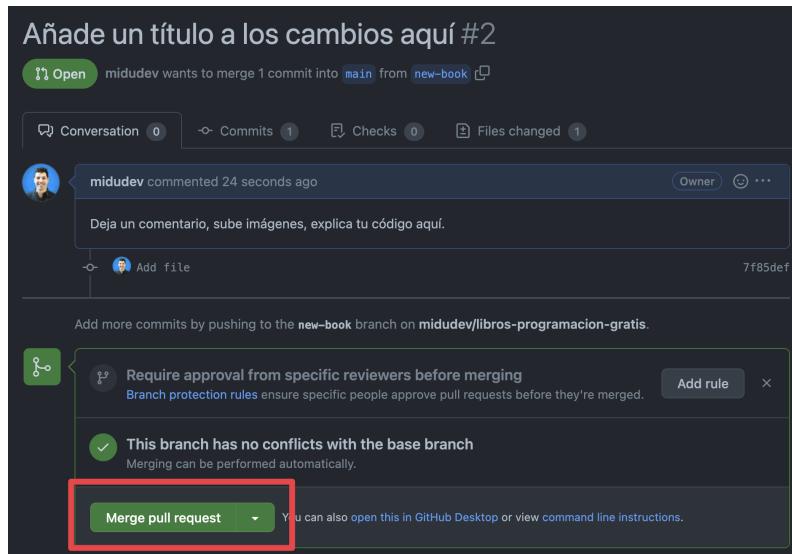
Nos abrirá una nueva página donde podemos añadir un título y una descripción a la *pull request*. Una vez que hemos terminado, podemos hacer clic en el botón *Create pull request*.



El punto 1 indica el destino de la petición de cambios (donde queremos añadir los cambios de la rama). El punto 2 es la rama que queremos fusionar. Recuerda dejar un buen título y descripción en los puntos 3 y 4. Y finalmente, pulsamos 5 para hacer la PR.

Una vez que hemos enviado la *pull request*, podemos esperar a que otra persona revise los cambios. En ese punto se puede iniciar una conversación con comentarios, peticiones de cambios, bloquear la *pull request* o aceptarla.

Si se acepta, finalmente, se debe pulsar el botón de *Merge pull request* para integrar los cambios en la rama de destino.

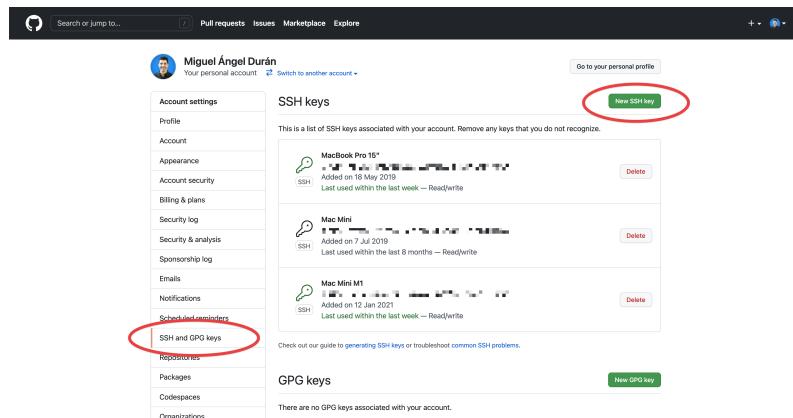


Así queda nuestra Pull Request. Si finalmente queremos integrar los cambios en la rama de destino, en este caso la rama 'main', entonces le daremos al botón que señalamos en la imagen

Configurando la conexión SSH con GitHub

SSH es un protocolo de comunicación segura que permite a los usuarios de una red conectar a un servidor remoto. Esto nos permite trabajar con repositorios remotos sin necesidad de usar usuario y contraseña cada vez que hagamos una acción.

GitHub ofrece una forma de configurar la conexión SSH. Para ello, debes iniciar sesión en tu cuenta de GitHub y acceder a la página de configuración de tu cuenta en <https://github.com/settings/keys>.



En la pantalla de configuración debemos ir a la sección de “SSH and GPG keys” y luego hacer click en “New SSH Key”

En la siguiente pantalla nos pedirá un título, que será idealmente el dispositivo o servidor que debe tener acceso a nuestra cuenta, y una llave SSH. Vamos a dejar esa ventana abierta y ya volveremos más adelante.

¡Venga! Vamos a lograr esa llave SSH.

Si usas GitHub Desktop, la aplicación de escritorio oficial de GitHub, puedes obviar toda la configuración de SSH. ¿Entonces por qué lo explicamos? Por dos razones: es interesante

que sepas usar Git en el entorno original en el que está pensado, que es la terminal, y porque es importante conocer cómo funcionan las cosas por detrás para no pensar que es magia.

Cómo generar una llave SSH

Antes de generar una llave SSH, podemos revisar que no tengamos ya una generada previamente. Para ello podemos usar `ls` para ver todas las llaves SSH que tenemos en nuestra máquina.

```
# listamos las llaves SSH que ya tenemos
$ ls -al ~/.ssh
```

La localización `~/.ssh` suele ser la carpeta donde se generan las llaves SSH pero puede ser diferente dependiendo del sistema operativo que estemos usando. Si no encuentras las llaves SSH o no tienes claro dónde están, no te preocupes, las generaremos.

Si no te aparece ningún fichero con extensión `.pub`... ¡no pasa nada!

Ahora te enseño a generar una. Si te salen los ficheros, puedes saltar al paso “*Usar una llave SSH*”.

Para generar una llave SSH debes usar el comando `ssh-keygen`. Te explico cómo usarlo:

- Usaremos el algoritmo `RSA`, un algoritmo antiguo basado en la factorización de números primos, que es el que usa la mayoría de las llaves SSH.
- También le indicamos el tamaño de la llave que queremos generar. En este caso decimos que sea de 4096 bits (que es lo recomendado para este algoritmo).
- Y, finalmente, indicamos con el parámetro `-C` un comentario que describa el uso de la llave.

Esto se traduce en la siguiente ejecución:

```
$ ssh-keygen -t rsa -b 4096 -C "tu.email@gmail.com"
> Generating public/private rsa key pair.
```

Después te preguntará **dónde queremos guardar el fichero donde se guardará la llave**. En este caso, lo guardaremos en el directorio que nos

propone que, aunque puede variar en cada sistema operativo, suele ser el lugar correcto.

```
> Enter a file in which to save the key (/Users/you/.ssh/id_rsa): [Pres\ns enter]
```

Tras esto, te preguntará por un *passphrase*. Esto no es más que tienes la posibilidad de añadir una frase de contraseña a la llave que generamos. Aunque la puedes dejar vacía, es recomendable que le pongas alguna contraseña a tu llave.

```
> Enter passphrase (empty for no passphrase): [Type a passphrase]
> Enter same passphrase again: [Type passphrase again]
```

Después de esto, ya habrá generado la llave SSH y, para asegurarnos, podemos volver a ejecutar el comando que vimos anteriormente para listar las llaves SSH disponibles:

```
$ ls -al ~/.ssh
total 32
drwx----- 6 midudev staff 192 17 oct 16:12 .
drwxr-xr-x+ 77 midudev staff 2464 17 oct 16:13 ..
-rw-r--r--@ 1 midudev staff 159 30 jul 21:06 config
-rw----- 1 midudev staff 411 12 ene 2021 id_ed25519
-rw-r--r-- 1 midudev staff 98 12 ene 2021 id_ed25519.pub
-rw-r--r-- 1 midudev staff 3772 17 mar 2021 known_hosts
```

Usar una llave SSH

Ya tenemos la llave SSH. Ahora vamos a levantar el *agente ssh*, un pequeño programa que queda latente en nuestra máquina y que se encarga de usar la llave SSH adecuada para las conexiones que realizamos.



Cuando hablamos de llaves SSH es literalmente eso, una llave. Ahora estamos generando en nuestra máquina la llave pero más tarde le diremos a GitHub que esta misma llave debe reconocerla como válida

Para ponerlo en marcha debemos ejecutar el siguiente comando:

```
$ eval "$(ssh-agent -s)"
```

Ahora tenemos que añadir la llave SSH que hemos generado previamente. Para ello, debemos ejecutar el comando `ssh-add`, de forma que le daremos al agente la ruta del fichero con la llave SSH que puede usar.

```
# En macOS se usa el parámetro -K  
$ ssh-add -K ~/.ssh/id_rsa
```

```
# En Windows y Linux no se necesita  
$ ssh-add ~/.ssh/id_rsa
```

¡Ya podemos conectarnos a GitHub! Vamos a ello.

Añadir clave SSH a tu cuenta de GitHub

Para que GitHub sepa que la llave SSH que hemos creado es válida... ¡se lo tenemos que decir de alguna forma! ¿Recuerdas la ventana que habíamos dejado abierta al inicio de la explicación? Bueno, si no, no te preocupes. Ve a esta URL: <https://github.com/settings/keys>

Y allí pulsa el botón `New SSH Key`. En la nueva pantalla introduce como título el dispositivo en el que has creado la llave SSH. Por ejemplo, yo pondría “*Mac Mini M1*”. De esta forma siempre sabré que la llave que aparece es la de ese dispositivo en concreto.

Ahora nos falta añadir la *Key*, que será la llave que hemos creado previamente. ¿Cómo la recuperamos? Para ello, podemos ejecutar uno de los siguientes comandos para copiar la llave SSH en el portapapeles:

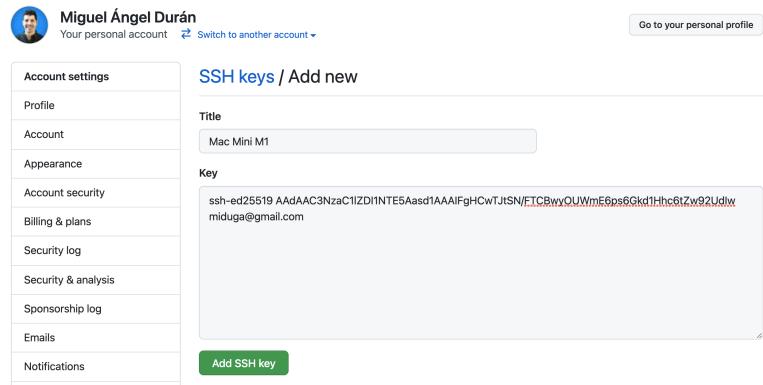
```
# en MacOS  
pbcopy < ~/.ssh/id_rsa.pub
```

```
# en Windows  
clip < ~/.ssh/id_rsa.pub
```

```
# en Linux  
xclip -sel clip < ~/.ssh/id_rsa.pub
```

Ten en cuenta que `~/.ssh/id_rsa.pub` es la ruta por defecto. Si has usado otra o, por alguna razón, tu sistema operativo guarda las llaves SSH en otro sitio, seguramente tendrás que adaptar el comando. Normalmente será un fichero `.pub`.

El contenido del portapapeles lo pegamos tal cuál en la caja de texto de las preferencias de GitHub y hacemos clic en `Add SSH Key`.



A screenshot of the GitHub 'SSH keys / Add new' page. At the top, it shows the user's profile picture and name, 'Miguel Ángel Durán'. Below that is a sidebar with 'Account settings' and various account management links. The main area is titled 'SSH keys / Add new' and contains fields for 'Title' (set to 'Mac Mini M1') and 'Key' (containing a long string of characters). A green 'Add SSH key' button is at the bottom.

No te preocupes que la llave que aparece no es válida... Ten en cuenta que este tipo de llaves tiene que ser privada y no debes compartirla de ninguna manera.

Probando, probando...

Ya lo tenemos todo listo. ¡Vamos a probar que todo funciona correctamente! Para ello, intentamos conectaros por SSH a GitHub.

```
$ ssh -T git@github.com
> La autenticidad del host 'github.com (DIRECCIÓN IP)' no se puede establecer.
> La clave de huella digital RSA es SHA256:nThbg6kXUpJWG17E1IGOCspRomTx\dcARLviKw6E5SY8.
> ¿Estás seguro de que quieras continuar conectado (sí/no)?
```

Pulsamos que sí y, si todo ha ido bien y hemos configurado correctamente la llave SSH, veremos que nos aparece el siguiente mensaje en la terminal.

```
Hi midudev! You've successfully authenticated, but GitHub does not provide shell access.
```

Cómo contribuir a un proyecto de código abierto

El mundo del software, tal y como lo conocemos hoy en día, no sería el mismo si no existiese el código abierto. De alguna forma, cuando visitas con tu navegador un sitio web, estás usando código abierto. Ya sea en tu sistema operativo, el navegador o la propia página web. **¡Hasta yo para escribir este libro que lees he usado en algún punto algo de código abierto!**

Que cientos de miles de personas del mundo del desarrollo puedan acceder a millones y millones de líneas de código, leerlas, compartirlas y mejorarlas, hace que la evolución del software esté a otra velocidad.

El propio Git, como hemos dicho anteriormente, es de código abierto. Un proyecto que, de otra forma, hubiera acabado siendo muy diferente a lo que conocemos hoy en día.

¿Por qué debería contribuir al código abierto?

Podría intentar convencerte que **tienes una deuda con el código abierto** ya que, de alguna forma, te has beneficiado de él en algún momento de tu vida. Pero voy a evitar jugar esa carta...

En lugar de eso, **quiero convencerte de que contribuir de alguna forma al código abierto va a ayudarte a mejorar tu carrera profesional** por diferentes motivos.

Te hace subir de nivel

Contribuir al código abierto es una forma de mejorar como profesional en tu sector. No sólo vas a necesitar conocer diferentes tecnologías para hacerlo, sino que además tu código va a ser revisado por otros contribuidores.

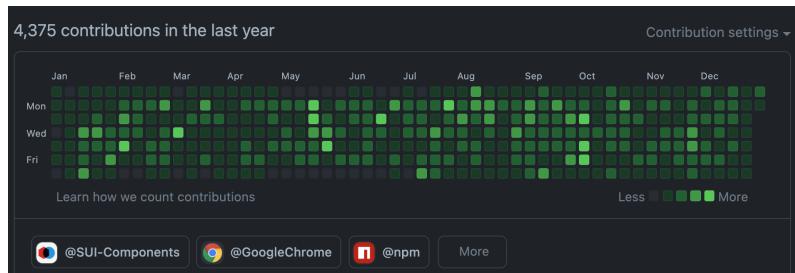
Esto **hará que recibas opiniones sobre tu código** y otras personas podrán ver qué es lo que está mal y qué puedes hacer para mejorarlo.

Además podrás ver soluciones alternativas y la gente compartirá su experiencia contigo además de forzarte a seguir unas buenas prácticas con tu solución.

Ganas reconocimiento

Hoy en día, muchas personas están buscando una forma de trabajar en el desarrollo de software. **¿Cómo puedes demostrar que eres capaz de hacerlo?**

Si ya cuentas con años de experiencia, es sencillo. Simplemente explicas en qué lugares has trabajado, atesoras unas cuantas referencias y, quizás, tengas suficiente.

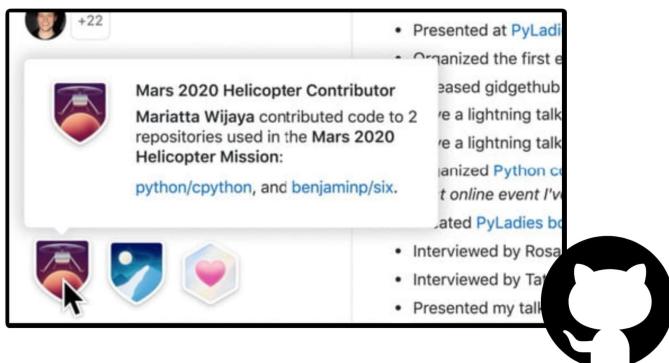


Tus contribuciones aparecerán en GitHub. No hay que obsesionarse con contribuir todos los días pero sí que ofrece algo de gamificación ver que vas rellenando cuadraditos verdes.

Pero si no es el caso, una forma de ganar experiencia con desarrollos que tienen impacto en muchas personas es contribuir al código abierto. Si lo haces en proyectos que son reconocidos, las empresas y posibles futuros clientes lo tendrán en cuenta como algo a valorar positivamente.

Impacto en la comunidad

Los proyectos de código abierto dan servicio a millones de personas y ayudan a cientos de miles de proyectos cada día. La propia NASA escribió un artículo en julio de 2021 donde explicaba [los proyectos de código abierto que habían ayudado al helicóptero Ingenuity a aterrizar en Marte](#).



GitHub otorgó una insignia a las personas que habían contribuido a proyectos de código abierto que fueron usados por la NASA en la misión Ingenuity

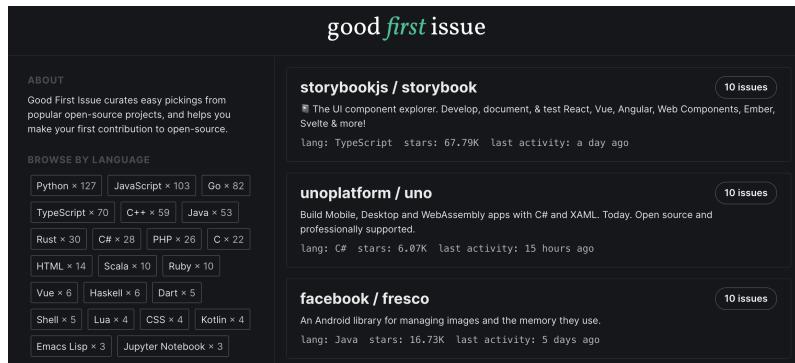
Al final, tampoco hace falta que tu proyecto llegue lejos del sistema solar. Por ejemplo, una biblioteca de UI que usen diferentes aplicaciones web, una herramienta para comprobar el código, una utilidad de testing... y tu trabajo será usado por miles de personas. ¿Cómo suena eso?

¿Cómo empiezo a contribuir a un proyecto de código abierto?

En primer lugar, busca un proyecto de código abierto que te interese. Si no lo encuentras, puedes crear uno... aunque el impacto que puedes tener es muy pequeño al principio y vas a echar en falta el *feedback* del resto de personas.

Si no lo tienes claro, te recomiendo que visites la página de [Good First Issue](#). Esta página te permite buscar proyectos de código abierto que requieren ayuda y que han etiquetado algunas tareas como fáciles para gente que quiera contribuir por primera vez en el código abierto.

Puedes filtrar por diferentes lenguajes como *JavaScript*, *Go* o *Python*.



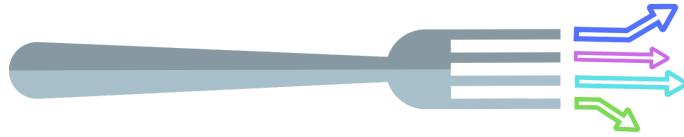
Good First Issue te ofrece errores sencillos de arreglar en centenares de repositorios reconocidos

Una vez tengas claro el proyecto al que vas a contribuir, es el momento de crear un *fork* del mismo.

¿Qué es un fork?

Un fork es una copia de un proyecto que ha tomado como base un código fuente que ya existía. La copia, entonces, puede recibir modificaciones que no han sido aprobadas en el código original y puede ser mantenido por

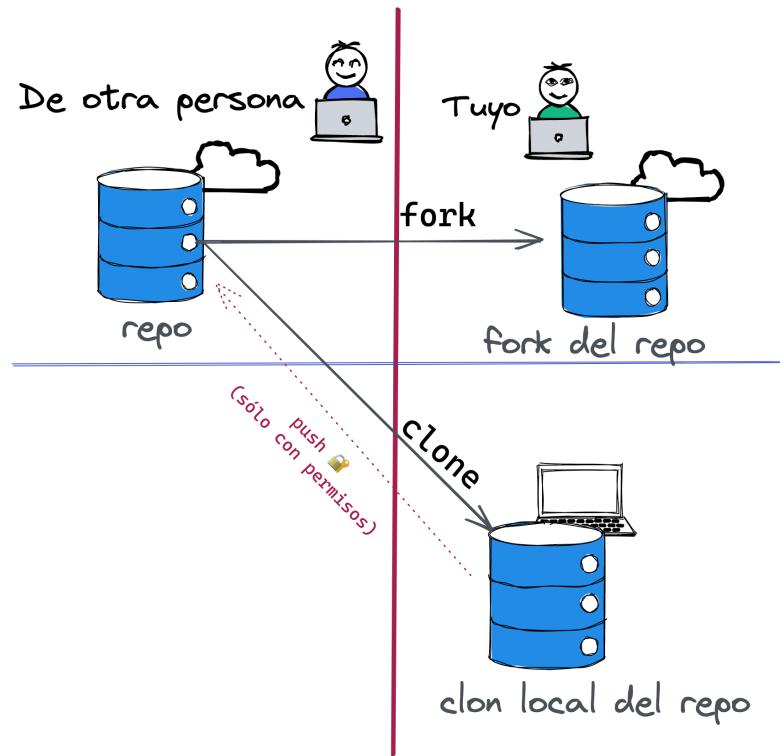
personas distintas, de forma que su desarrollo puede tomar una dirección distinta.



La palabra no tiene nada que ver con tenedor pero, por si te sirve para recordarlo, puedes pensar en cada terminación del tenedor que va a un lado diferente...

Si te preguntas por qué se usa la palabra *fork* tienes que saber que, en inglés, la traducción no sólo significa *tenedor*. En realidad, la palabra se usa para referirse también a bifurcaciones, dividir en ramas e ir por partes separadas. Y se conoce ese uso de la palabra desde el siglo XIV. De esta forma, en el mundo del desarrollo, se empezó a usar para referirse a la operación de crear procesos que eran copias de si mismo, algo muy típico en Unix. Más adelante empezó a usarse también para referirse a divergir código en sistemas como Git.

Cuando hablamos de Git, un *fork* es, literalmente, copiar un repositorio y evolucionar el código en ese repositorio sin afectar o tener en cuenta el original.



Muchas veces se puede confundir un **fork** con un **clone** pero no son lo mismo. El **fork** crea un repositorio en la nube dónde tú tienes todos los permisos. El **clone** hace una copia local de un repositorio donde tú puedes tener, o no, los permisos

La diferencia entre *clonar* un proyecto y hacer un *fork* es que al hacer *clone* simplemente estamos creando una copia local del proyecto original, manteniendo el repositorio remoto intacto. Con un *fork* estamos creando una copia remota del proyecto, con nuevos permisos y, por lo tanto, una nueva dirección.

¿Por qué haríamos algo así? Pues bien, puede ser por diferentes motivos:

- **El proyecto original de código abierto ha sido abandonado y no tenemos privilegios para enviar nuevo código.** De esta forma, cuando creamos un *fork*, podemos generar una copia del código original y publicar nuevas versiones usando un nuevo nombre.
- **El proyecto original no está de acuerdo con cambios que queremos aplicar.** Así, al crear un *fork*, podemos hacer los cambios que

consideremos sin importar ni impactar al original. Un ejemplo de esto fue el *fork* que se realizó de Node.js en su día para crear io.js (hoy en día deprecado).

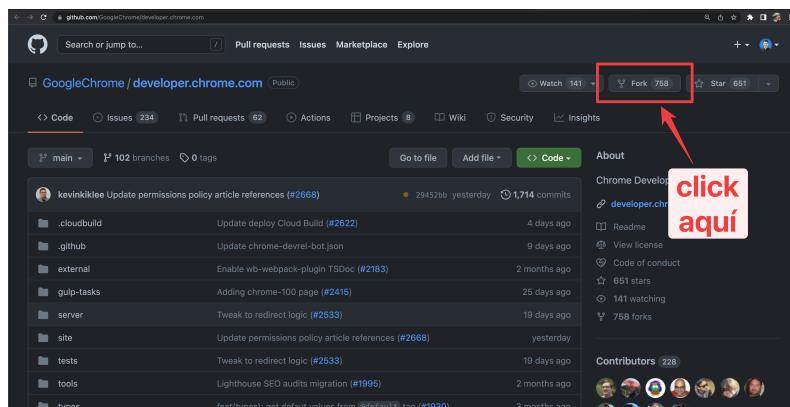
- **No tenemos permisos para enviar código al proyecto original.** Al crear un fork, podemos crear una Pull Request que nos permita enviar cambios al proyecto original. Esto sería evolucionando la copia y, después, enviando una petición de cambios al original con el nuevo código.

Justamente, por esta última razón, es que vamos a necesitar crear un fork para poder contribuir al proyecto.

Creando nuestro primer fork

¡Una cosa! En el libro nos enfocamos en GitHub, que es el servicio de alojamiento de repositorios más grande del mundo. Sin embargo esta operación puede ser ligeramente diferente en otros servicios como BitBucket y GitLab. Sería imposible cubrirlos todos pero lo aprendido aquí será muy similar a cómo se haría en otros servicios.

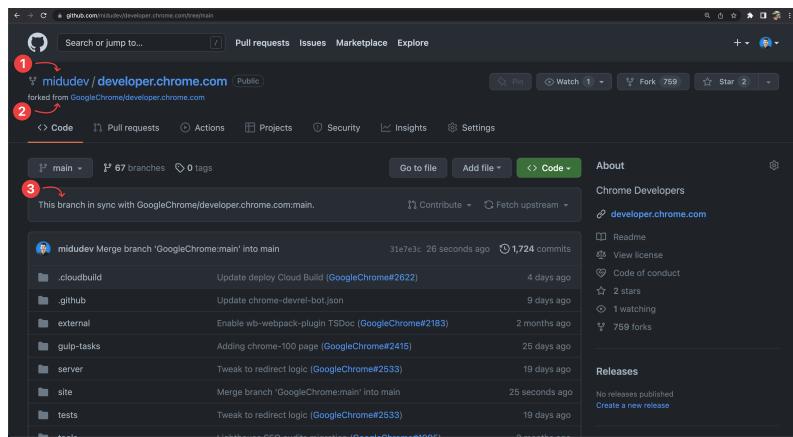
Para crear un fork debemos acceder a la página del repositorio original. En este caso, vamos a acceder al [repositorio dónde se encuentra la documentación y el blog de Google Chrome.](#)



Al entrar al repositorio, veremos que tenemos un botón que nos invita a hacer un Fork del repositorio

Como verás, en la parte superior derecha, existe una botonera dónde podemos suscribirnos a los cambios del repositorio (*Watch*), crear un fork del mismo (*Fork*) o guardar el repositorio en favoritos (*Star*). Al lado de cada uno, el número indica el número de personas que han realizado esa misma acción.

Hacer clic en *Fork* se iniciará un proceso de unos pocos segundos. Una vez finalizado, se abrirá una nueva página donde podremos ver el repositorio que hemos creado.



Aunque el repositorio se parece mucho al original, existen algunas diferencias importantes que nos indican que el fork ha sido un éxito y que nos encontramos ante un repositorio diferente

Si revisas la imagen, verás que hemos indicado diferentes puntos importantes. Vamos a explicarlos.

1. **Ahora el repositorio te pertenece.** Por eso ha cambiado la organización anterior (*GoogleChrome*) por tu nombre de usuario (*midudev*). Esto no es sólo un cambio visual. Significa que con estos de permisos de administración puedes hacer lo que quieras con este repositorio.
2. Un mensaje pequeño nos indica que este repositorio es un fork de otro. En este caso de *GoogleChrome/developer.chrome.com*. De esta forma existe una relación entre los repositorios y cualquier persona puede acceder al original con el enlace.

3. Un mensaje nos indica que la rama en la que nos encontramos está sincronizada con la misma rama del repositorio original. En este caso la rama *main*. También nos podría decir si hemos hecho commits en nuestro repositorio que no están en la rama del repositorio original o, al revés, que el repositorio original tiene commits que nuestro *fork* no tiene para esa rama.

¿Quieres hacer un fork desde la terminal? Más adelante en el libro, en la sección GitHub CLI, hablamos sobre el uso del comando `gh` de GitHub. Este comando te puede simplificar mucho la vida si no quieres tener que visitar la página de GitHub para realizar ciertas acciones. En este caso para hacer un fork de un repositorio podrías simplemente ejecutar `gh repo fork https://github.com/<repo-org>/<repo-name>.git --clone`.

¿Cómo hago una Pull Request al proyecto original?

Como hemos dicho antes, al hacer un *fork* podemos tener diferentes objetivos. **El más común es poder contribuir al repositorio original** y esto lo haríamos a través de una *Pull Request*.

Una *Pull Request*, o de forma abreviada **PR**, es una petición de cambios que se envía al repositorio original. Estos cambios que queremos llevar a cabo los tenemos que agrupar en commits, en una rama, en nuestro *fork*. Con esos cambios podremos crear nuestra petición.

Como bien dice el nombre, es una petición. Las peticiones se pueden aceptar o denegar. Si la petición es aceptada, el cambio se aplica al repositorio original. Si no, el cambio se descarta. A veces es posible que en

En la sección *Buenas prácticas* del libro vas a encontrar una serie de consejos que te ayudarán a crear una Pull Request para tener más posibilidades de que sea aceptada y tus cambios fusionados.

```
# primero nos clonamos el fork
# fíjate que estoy usando el repo de mi organización midudev
$ git clone git@github.com:midudev/developer.chrome.com.git

# ahora podemos acceder a la copia local
$ cd developer.chrome.com

# para empezar a trabajar, creamos una rama nueva
$ git switch -c add-spanish-translations
Switched to a new branch 'add-spanish-translations'

# hacemos los cambios que queramos en el editor
# en este caso voy a abrir el proyecto con VS Code
# y voy a añadir unas traducciones a la documentación
$ code .

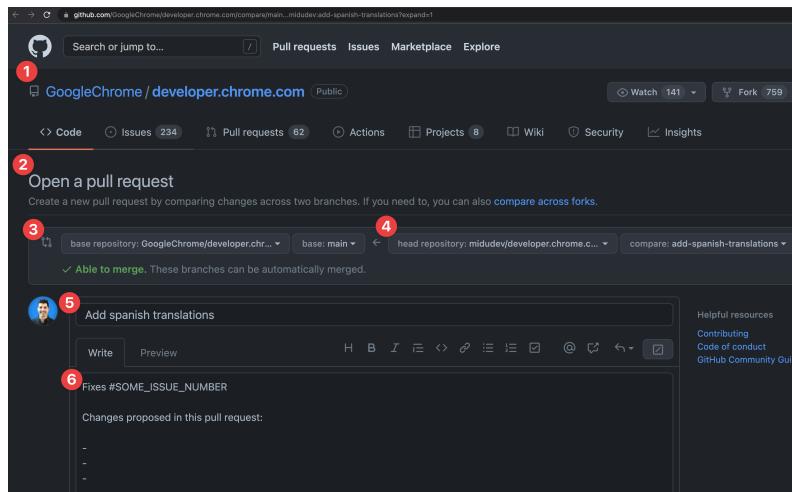
# los añadimos a la fase de preparación
$ git add .
# hacemos un commit
$ git commit -m "Add spanish translations for Chrome 101"
[add-spanish-translations d2b217d8] Add spanish translations for Chrome\
101
1 file changed, 1 insertion(+), 1 deletion(-)
```

```
# hacemos push de la rama al repositorio remoto
$ git push origin add-spanish-translations

Enumerating objects: 15, done.
Counting objects: 100% (15/15), done.
Delta compression using up to 10 threads
Compressing objects: 100% (8/8), done.
Writing objects: 100% (8/8), 692 bytes | 692.00 KiB/s, done.
Total 8 (delta 6), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (6/6), completed with 6 local objects.
remote:
remote: Create a pull request for 'add-spanish-translations' on GitHub \
by visiting:
remote:     https://github.com/midudev/developer.chrome.com/pull/new/a\
dd-spanish-translations
remote:
To github.com:midudev/developer.chrome.com.git
 * [new branch]      add-spanish-translations -> add-spanish-translat\
ions
```

Al hacer push a nuestro repositorio remoto, nos indica que podemos crear una *Pull Request* visitando la [URL que nos ha indicado en la terminal](#).

Al visitarla nos vamos encontraremos una pantalla como esta:



Al crear una petición de cambios de un fork, GitHub ya nos ofrece hacerlo

Vamos a repasar los puntos clave de la imagen:

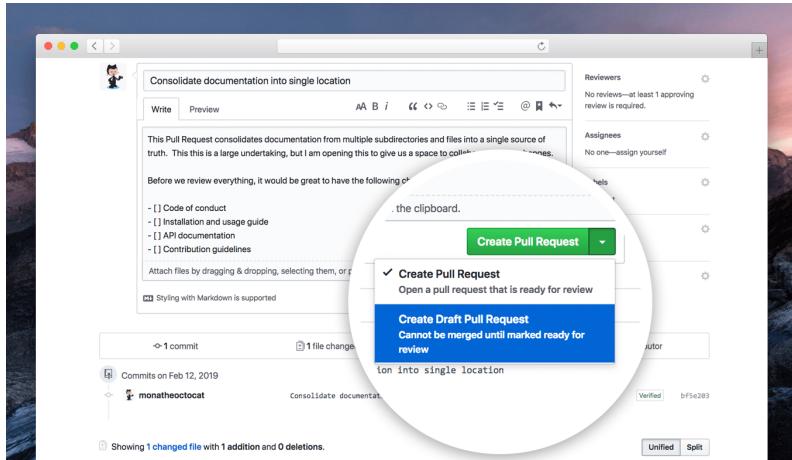
1. Arriba nos indica **sobre qué repositorio estamos operando**. Si te fijas bien, aparece *GoogleChrome/developer.chrome.com*. Esto significa que estaríamos haciendo una petición de cambios al repositorio original y no al fork. El fork es *midudev/developer.chrome.com*.
2. Esta es **la acción que queremos realizar**. Crear una *Pull Request* comparando entre dos ramas de dos repositorios diferentes.
3. Aquí podemos elegir **el repositorio remoto al que le queremos hacer la petición de cambios y a qué rama**. Por defecto nos ha seleccionado el repositorio original pero podríamos hacer la petición a cualquier fork (incluso el nuestro mismo) y a cualquier rama disponible.
4. Estos son **los cambios que queremos hacer**. Aquí estamos indicando que los cambios que hemos hecho en nuestro fork en la rama `add-spanish-translations` son los que queremos llevar al repositorio original.
5. **El título de la petición de cambios**. Aparecerá en la lista de Pull Requests del repositorio original.
6. **La descripción**. En este caso ya aparecía algo escrito. ¿Por qué? Porque el repositorio original tiene configurada una plantilla por defecto. Esto puede variar dependiendo de cada repositorio y es útil para asegurarte que todas las peticiones siguen una estructura.

Una vez que hayas completado la información tendrás que pulsar el botón verde *Create Pull Request*, se sitúa debajo del cuadro de texto para la descripción.

Marcar tu Pull Request como Borrador (Draft)

Es interesante que conozcas también la posibilidad de crear borradores de PRs. De esta forma podrás crear una petición de cambios que está todavía con trabajo en progreso. Que se trata de un borrador (*draft*).

Es útil porque de esta forma das visibilidad a los cambios que quieres realizar, puedes empezar a recibir comentarios pero dejas bien claro que todavía no está terminada (y, por lo tanto, no es posible fusionarla incluso aunque personas con permisos del repositorio original quieran).



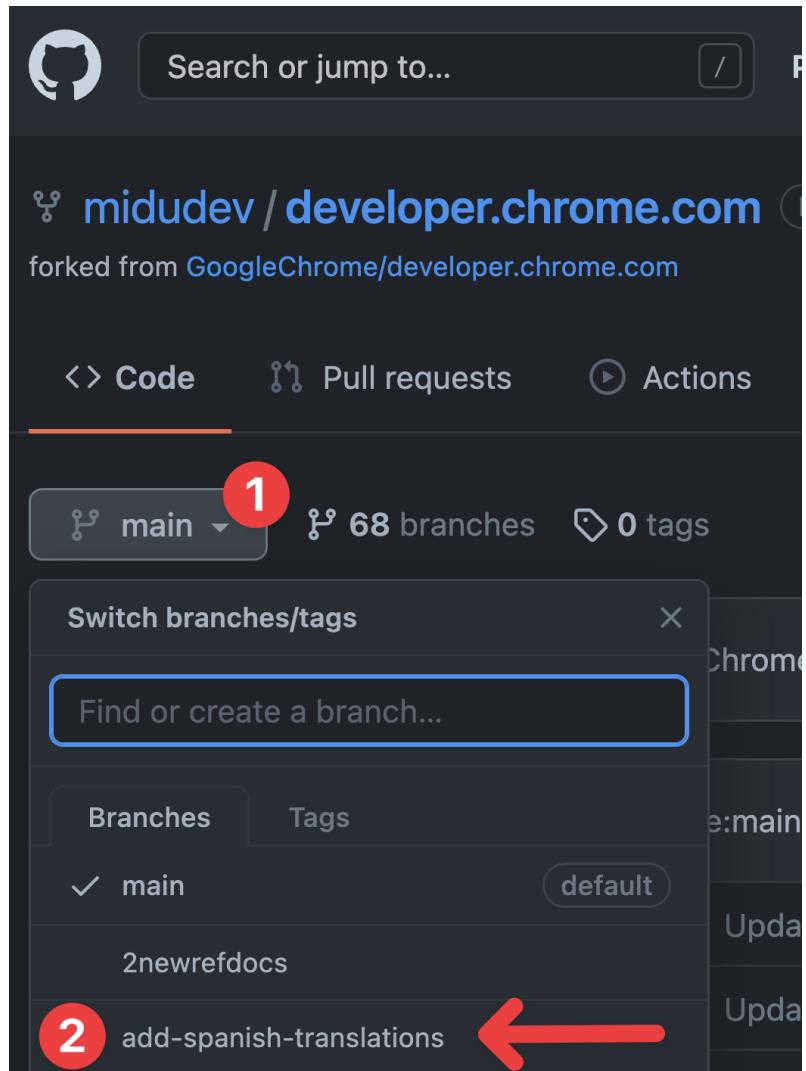
Puls a la flecha al lado del botón al crear la Pull Request y te aparecerá una opción para marcar la petición como borrador (draft)

Hasta que no se marque la Pull Request como lista no será posible fusionarla.

Otra forma de hacer la Pull Request

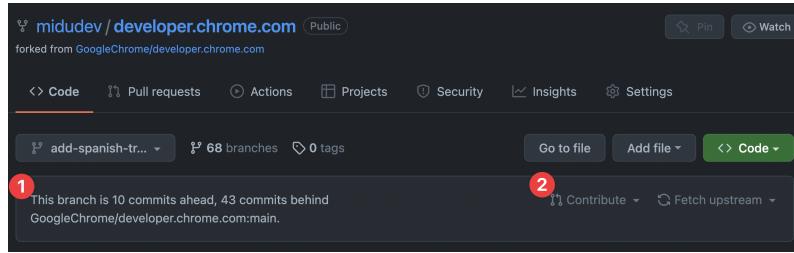
¿Has perdido la URL y ahora no sabes cómo hacer la Pull Request? No te preocunes, GitHub tiene una manera para que siempre puedas crearla fácilmente.

Para ello sólo tienes que [ir a tu fork en GitHub](#). Allí seleccionas la rama en la que has hecho los cambios con los que quieras crear la *PR*.



En el listado de ramas, selecciona la que quieras enviar como Pull Request al repositorio original.

Una vez las seleccionas, la vista del repositorio cambiará para reflejar los archivos que tienen en la rama.



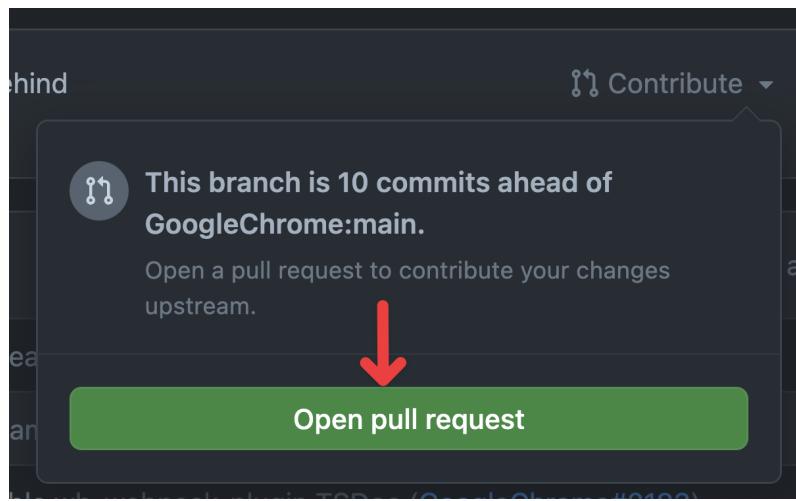
Al entrar en una rama en nuestro repositorio remoto, tendremos información sobre su sincronización respecto al repositorio original y un botón para iniciar la Pull Request

1. Aquí podrás ver cómo la rama de nuestro fork tiene el historial de *commits* sincronizado respecto al repositorio original. En este caso nos dice que hemos hecho 10 commits que están por delante del *HEAD* del repositorio original pero que hay 43 commits que han hecho en la rama *main* del repositorio original y nosotros no los tenemos.

Esos 10 commits son justamente los que vamos a querer llevar en nuestra Pull Request. **Es totalmente normal que existan estas divergencias.**

Respecto a los 43 que no tenemos nosotros... ¿qué hacemos? **Depende del contenido de estos commits si es algo que nos afecta o no.** Si, por ejemplo, son commit que han modificado los mismos archivos que nosotros, vamos a tener que hacer un *merge* para poder converger. Si no tocan los mismos ficheros, podremos continuar sin problemas.

2. Este es el botón al que tenemos que darle para que nos aparezca de nuevo la ventana para crear nuestra Pull Request. Al hacerlo, nos aparecerá el siguiente diálogo:



Nos vuelve a recordar que tenemos commits que están por delante y nos invita a crear nuestra petición de cambios

Si nuestra rama no tuviese commits que fuesen por delante del HEAD del repositorio original, no tendríamos que hacer nada. No podríamos hacer una Pull Request porque no tenemos ningún cambio que llevar.

¿Cómo puedo sincronizar mi fork con el repositorio original?

Como hemos visto antes, nuestro fork se había quedado por detrás del repositorio original. **¿Cómo hacemos para que nuestro fork no se quede obsoleto al poco tiempo?** ¿Hay que borrarlo y volverlo a crear? No, no es necesario. Lo que tenemos que hacer es actualizarlo y ya está.

Usando la terminal

Ya sabes que me gusta enseñarte a usar y entender los comandos de Git en la terminal. Y **sincronizar un fork no iba a ser una excepción**.

Además, ahora verás por qué es interesante poder tener más de un repositorio remoto enlazado a un mismo repositorio local.

Cuando ejecutamos el comando `git remote` en la terminal, nos aparece una lista de repositorios remotos que tenemos enlazados a nuestro repositorio local.

Normalmente nos aparece sólo el repositorio remoto *origin* que es el que estamos usando. En nuestro caso nos el *midudev/developer.chrome.com* que es el que estamos usando.

```
$ git remote --verbose  
origin  git@github.com:midudev/developer.chrome.com.git (fetch)  
origin  git@github.com:midudev/developer.chrome.com.git (push)
```

Como vemos, nuestro repositorio local está enlazado a nuestro fork a través del alias *origin* pero lo que queremos es justamente traer los cambios del repositorio original... Así que vamos a tener que añadirlo entre los repositorios remotos.

Normalmente cuando se añade el repositorio remoto original a un fork se le llama *upstream*. Es una convención que se usa en muchos repositorios de desarrollo pero puedes llamarle como prefieras.

```
# añadimos el repositorio remoto original
# con el alias upstream
git remote add upstream git@github.com:GoogleChrome/developer.chrome.co\
m.git
```

Ahora podemos ver que el repositorio remoto *upstream* está enlazado a nuestro fork.

```
$ git remote --verbose

origin  git@github.com:midudev/developer.chrome.com.git (fetch)
origin  git@github.com:midudev/developer.chrome.com.git (push)
upstream      git@github.com:GoogleChrome/developer.chrome.com.git (fetch)
upstream      git@github.com:GoogleChrome/developer.chrome.com.git (push)
```

origin -> nuestro fork

upstream -> el repositorio original

¡Listo! Ya estamos preparados para sincronizar nuestro fork con el repositorio original. Para eso, sólo necesitamos usar `git pull`. Le indicamos que queremos descargar e integrar los cambios de la rama `main` del repositorio original al repositorio local de nuestro fork.

```
$ git pull upstream main

From github.com:GoogleChrome/developer.chrome.com
 * branch            main      -> FETCH_HEAD
   b7ff3d9e..29452bbf  main      -> upstream/main

Removing site/en/docs/devtools/css/print-preview/index.md
Removing site/en/100/100.11tydata.js
Removing site/_data/banner.yml
Merge made by the 'recursive' strategy.

.cloudbuild/deploy.yaml          | 16 +-+
.eleventy.js                      | 29 +-+
.github/chrome-devrel-bot.json    | 49 ++-
.redirects.yaml                   |  7 +
site/_data/authorsData.json       | 29 +-+
site/_data/banner.yml              |  6 -
site/_data/docs/devtools/toc.yml  |  7 +-+
```

Ahora estos cambios están en nuestro repositorio local. Para llevarlos a nuestro fork, podemos usar `git push`.

```
$ git push origin main
```

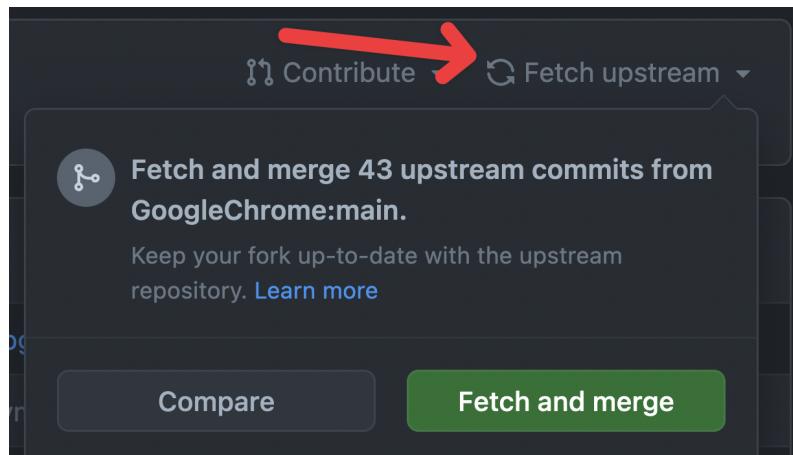
```
Enumerating objects: 10, done.  
Counting objects: 100% (10/10), done.  
Delta compression using up to 10 threads  
Compressing objects: 100% (4/4), done.  
Writing objects: 100% (4/4), 470 bytes | 470.00 KiB/s, done.  
Total 4 (delta 3), reused 0 (delta 0), pack-reused 0  
remote: Resolving deltas: 100% (3/3), completed with 3 local objects.  
To github.com:midudev/developer.chrome.com.git  
d2b217d8..f6f10183 main -> main
```

Ten en cuenta que esto tendrás que hacerlo también en otras ramas en las que estuvieses trabajando. Si no lo haces, las ramas que hayas creado ya en tu fork se quedarán también por detrás del historial de commits de la rama `main` del repositorio original.

Usando la UI de GitHub

“Demasiado texto” quizás has pensado al ver la sección anterior. En realidad, una vez que lo entiendes, es bastante mecánico pero es posible que te parezca más fácil, y objetivamente lo es, **darle a un botón directamente en la UI de GitHub**.

Por ello, puedes ir a la página de tu fork, seleccionar la rama que quieres sincronizar y simplemente darle al botón *Fetch upstream*. Se abrirá un diálogo que te permitirá descargar y fusionar los cambios de la rama principal del repositorio original.



Tan fácil como darle a un botón para actualizar la rama de tu fork

Verás que si tu rama está totalmente sincronizada, al abrir el diálogo te lo dirá y el botón estará desactivado.

¿Entonces por qué te he explicado todo el rollo anterior? Bien. Lo primero es porque **esto es el camino feliz**. Un camino en el que no te encuentras conflictos y todo funciona perfectamente. Pero si tienes conflictos al hacer la fusión, vas a tener que lidiar con ellos y eso es mejor hacerlo usando los comandos de la terminal y un editor.

Por otro lado, de esta forma entiendes qué mecanismos hay detrás de ese botón mágico de GitHub.

¿Con qué puedo contribuir a un proyecto?

Ahora que ya sabes cómo hacer y sincronizar forks y crear peticiones de cambios... Es posible que estés pensando: "*Oh, qué bonito es todo esto, pero mi nivel no es lo suficientemente bueno para poder hacerlo.*"

No te culpo. Yo lo he pensado docenas de veces. **A todos nos pasa.** Da vértigo participar en proyectos de código abierto, especialmente aquellos proyectos que son muy importantes para la comunidad o de personas que admirás.

Algunas ideas para que te animes a contribuir a proyectos de código abierto, sin necesidad de ser experto en programación o teniendo que crear soluciones de muchas líneas de código, podrían ser:

- **Solucionar errores ortográficos** que existan en el README, documentación o comentarios. Son peticiones de cambio pequeñas, sin prioridad alta, que no comportan mucho esfuerzo ni entender muy bien todo el código.
- Muchas veces, especialmente proyectos grandes, buscan **traductores para la documentación**. Esa es una muy buena oportunidad ya que, al mismo tiempo que contribuimos, vamos entendiendo mejor el proyecto.
- **Añadir tests** es una forma excelente de contribuir. Consigues muchas cosas: haces que el código sea más mantenible, de mejor calidad y es una contribución que es muy difícil de rechazar. Además, es una forma perfecta para entender el código.
- Al final puedes incluso contribuir a un proyecto de código abierto sin hacer *Pull Request*. Puedes ir a la sección de *Issues* y ayudar a cribar los problemas que la gente va presentando al repositorio. Muchas veces son peticiones de ayuda o necesitan una buena demostración para que las personas que mantienen el proyecto puedan entender mejor el problema.

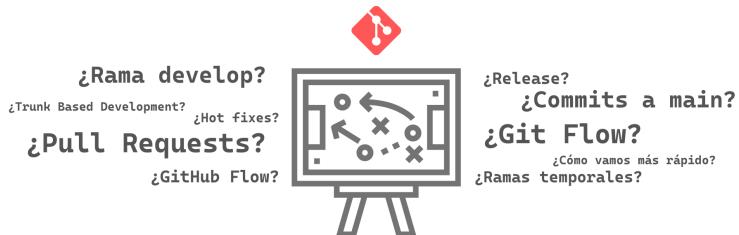
Todos hemos empezado por algún sitio. Anímate.

Flujos de trabajo y estrategias de ramas en Git

Hasta ahora hemos visto cómo se puede utilizar Git para manejar el código fuente de un proyecto. Sin embargo todavía no hemos hablado de las estrategias que se pueden seguir a la hora de trabajar en equipo.

Todas las estrategias se basan esencialmente en la forma en la que vas a tratar de crear (o no) ramas y fusionarlas a la rama principal. Y no, no hay una estrategia perfecta. Cada equipo y proyecto es un mundo, y esas peculiaridades son las que pueden marcar la diferencia para elegir entre una estrategia u otra.

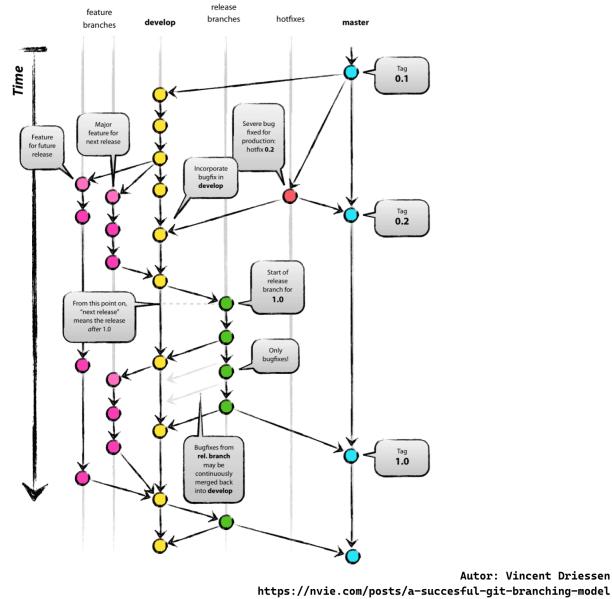
En este capítulo vamos a revisar las tres estrategias más famosas que se pueden seguir a la hora de trabajar en equipo: **Git Flow**, **GitHub Flow** y **Trunk Based Development**. También hablamos de otra estrategia, bastante más moderna, llamada **Ship / Show / Ask**. Al final del capítulo te compartiré mi opinión al respecto y a qué creo que deberías aspirar.



Usar una buena estrategia en un proyecto puede de determinar la velocidad, o incluso el éxito, de los desarrollos del equipo

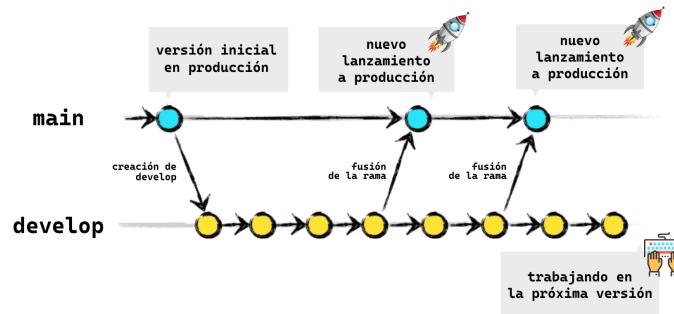
Git Flow

Una de las estrategias más famosas a la hora de trabajar en equipo es **Git Flow**. Fue ideada por el desarrollador *Vincent Driessen* en el año 2010 y presentada en el artículo ["A successful Git branching model"](#).



Una vez que la rama `develop` esté lista y estable, sus cambios serán incorporados a la rama `main` para crear una nueva versión de producción. Más adelante veremos cómo lo logramos.

Al final, en **Git Flow** tienes que entender que cada commit a la rama principal `main` se refiere a una versión de producción.



Al incorporar los cambios de la rama `develop` a `main` se genera un nuevo lanzamiento de versión a producción

Las ramas de apoyo

Además de las ramas `main` y `develop` existen las ramas de apoyo. Hay tres tipos de ramas de apoyo y, a diferencia de las principales, son ramas temporales que serán eliminadas una vez que sean fusionadas. Cada rama tiene una misión en concreto:

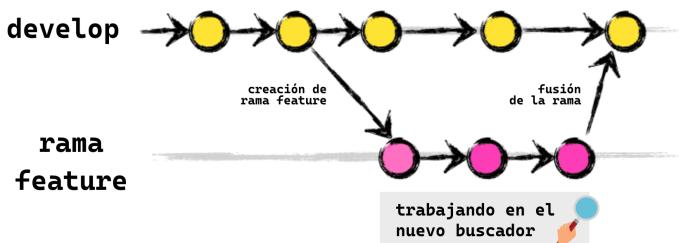
- **Feature:** Cuando trabajas en una nueva característica para el proyecto.
- **Release:** Cuando preparas el lanzamiento de una nueva versión.
- **Hotfix:** Para trabajar en cambios imprevistos como parches para arreglar un bug o un problema en producción.

Ramas Feature

Lo natural en cualquier proyecto es trabajar en nuevas características para iterarlo y hacerlo cada vez mejor y más completo. Estas nuevas

funcionalidades, aunque no se sabe exactamente cuándo se lanzarán a producción, están planificadas y, normalmente, documentadas.

Para trabajar en este tipo de desarrollo en **Git Flow** existen las ramas feature. Estas ramas se crean a partir de la rama `develop` y, una vez finalizan, son fusionadas de nuevo en `develop` y eliminadas.



Creamos ramas feature desde la rama develop y, más tarde, volverán a ser integradas en esta rama

Cómo crear y fusionar una rama feature

Como hemos dicho anteriormente, debemos crear la rama `feature` a partir de `develop`. Para ello usamos el siguiente comando:

```
# creamos una rama llamada feature-new-search desde develop
# también puedes usar el comando:
# $ git checkout -b feature-new-search develop
$ git switch -c feature-new-search develop
Switched to a new branch "feature-new-search"
```

Aquí trabajaremos en la feature, añadiendo los commits que sean necesarios a la rama, hasta que estemos satisfechos con la nueva característica.

Entonces será el momento de fusionar ese trabajo a la rama `develop` para que esté disponible en el próximo lanzamiento.

```
# vamos a la rama develop  
# también puedes usar el comando:  
# $ git checkout develop  
$ git switch develop  
Switched to branch 'develop'
```

```

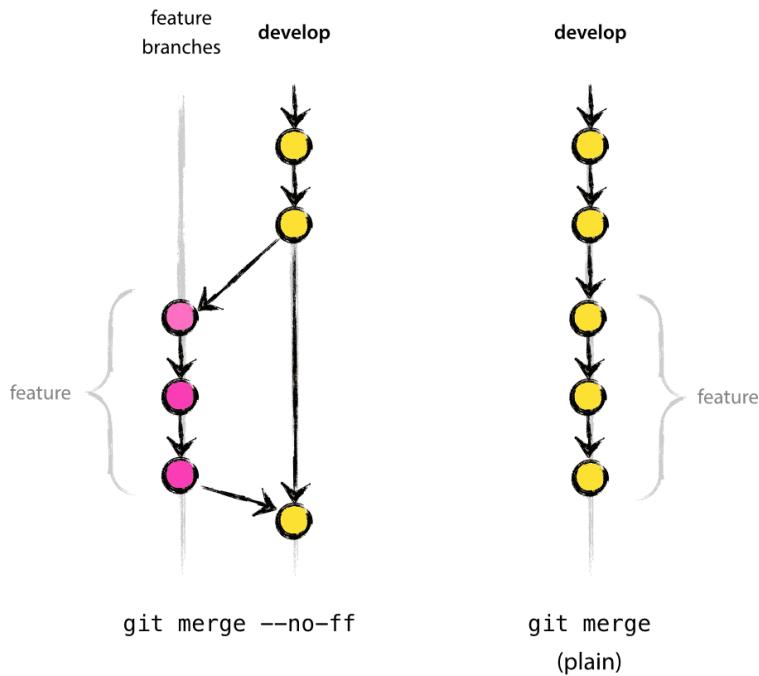
# fusionamos nuestra feature sin fast-forward
$ git merge --no-ff feature-new-search
Updating dc3a92a..12c6443
(Summary of changes)

# eliminamos la rama que ya no necesitamos
$ git branch -d feature-new-search
Deleted branch feature-new-search (was 12c6443).

# empujamos los cambios al repositorio remoto
$ git push origin develop

```

El parámetro `--no-ff` es opcional. Sirve para crear siempre un commit al fusionar la rama, incluso cuando se podría realizar un *fast-forward*. Esto se hace para dejar en el historial de commits un commit que contenga todos los cambios de la rama que se está fusionando. De otra forma sería imposible seguir el flujo de trabajo de la rama `develop` a la hora de saber qué cambios se han añadido para una rama en concreto. Aunque el historial queda “más sucio”, **la información es más importante que la limpieza**.



Aquí puedes ver la diferencia entre ejecutar `git merge` con o sin el argumento `-no-ff`

Resumen de ramas feature:

- Se crean desde: `develop`
- Se fusionan en `develop`
- Convención de nombre: `feature-*`

Ramas Release

Las ramas `release` sirven para preparar un nuevo lanzamiento de nuestro código a producción. Para ello, una vez que la rama `develop` ha sido validada y tiene los cambios que queremos lanzar, creamos a partir de ella una rama `release`. Se fusionan de nuevo en `develop`, `main` y, después, eliminadas.

Aunque no es recomendable, en las ramas `release` se puede seguir trabajando para añadir algún pequeño cambio de última hora o algún parche a un error que haya sido detectado justo antes del lanzamiento. También se puede, y seguramente es mejor opción, usar un `hotfix` como veremos más adelante.

Lo que sí que vamos a evitar es trabajar en cualquier nueva característica que podría en su lugar ser creada en una rama `feature`. La idea es que la rama `release` refleja la versión final del código que va a llegar a producción y será la nueva rama `main` del proyecto.

Cómo crear y fusionar una rama release

Las ramas `release` se crean, como hemos comentado, a partir de la rama `develop`. Para ello usamos el siguiente comando:

```
# creamos la rama release con la versión 1.2.0 desde develop
$ git checkout -b release-1.2.0 develop
Switched to a new branch "release-1.2.0"

# esto debería actualizar los archivos necesarios
# para el cambio de versión
# o también podemos hacerlo en CI
$ ./bump-version.sh 1.2.0
Files modified successfully, version bumped to 1.2.0.

# añadimos un commit con los cambios generados
$ git commit -am "Bumped version number to 1.2.0"
[release-1.2 74d9424] Bumped version number to 1.2
1 files changed, 1 insertions(+), 1 deletions(-)
```

En este punto podemos realizar algunos cambios aunque es poco recomendable... Aunque la rama `release` puede vivir durante un tiempo, lo mejor es que sea usada lo antes posible.

Para finalizar la rama de `release` tenemos que fusionarla con `develop` y `main`. Para ello hacemos:

```
# cambiamos a la rama main
# (también puedes usar git checkout main)
$ git switch main
# fusionamos la rama release con la rama main
$ git merge --no-ff release-1.2.0
Merge made by recursive.
(Summary of changes)
# creamos un tag para etiquetar la nueva versión
$ git tag -a 1.2.0

# ahora nos toca también hacer lo propio con develop
$ git switch develop
Switched to branch 'develop'
$ git merge --no-ff release-1.2.0
Merge made by recursive.
(Summary of changes)
```

Es posible que encuentras algún conflicto al volver a la rama `develop`, ya que las personas del equipo han podido seguir trabajando sobre ella. No pasa nada, resuelve los conflictos y haz un commit a `develop` para solucionarlos.

Una vez hemos terminado con la rama de `release`, procedemos a eliminarla.

```
$ git branch -d release-1.2.0
Deleted branch release-1.2.0 (was ff452fe).
```

Resumen de ramas `release`:

- *Se crean desde: `develop`
- *Se fusionan en `develop` y `master`
- *Convención de nombre: `release-*`

Ramas Hotfix

Las ramas de apoyo `hotfix` son ramas temporales que se crean para trabajar en cambios imprevistos. Una vez que se hayan validado y aprobados, se incorporan a la rama `main` para crear una nueva versión de producción.

Puedes traducir `hotfix` como *parche*, *pañó caliente* o *tirita*. Soluciones temporales a problemas o cambios imprevistos. Por eso, **al crear una rama `hotfix` se debe hacer desde la rama `main`** ya que no se podría crear una solución desde la rama `develop` ya que contiene cambios que pueden ser inestables todavía.

Lo que sí que tiene sentido es que, más adelante, fusionaremos el mismo `hotfix` a `develop` para que el problema no vuelva a surgir y todo el equipo tenga solucionado el problema.

Cómo crear una rama hotfix

```
# Creamos una rama hotfix desde la rama principal main
$ git switch -c hotfix-2.5.1 main
Switched to a new branch "hotfix-2.5.1"

# Hacer bump de la versión es opcional y depende
# del sistema de CI y CD que uses
$ ./bump-version.sh 2.5.1
Files modified successfully, version bumped to 2.5.1.

$ git commit -am "Bump version to 2.5.1"
[hotfix-2.5.1 32claff] Bump version to 2.5.1
1 files changed, 1 insertions(+), 1 deletions(-)

# Hacemos el commit con el arreglo del problema
$ git commit -m "Fix bug causing app not working properly"
[hotfix-2.5.1 44c2aff] Fix bug causing app not working properly
3 files changed, 24 insertions(+), 16 deletions(-)

# Fusionamos la rama hotfix con la rama main
$ git switch main
Switched to a new branch "main"

$ git merge --no-ff hotfix-2.5.1
Merge made by recursive.

# También fusionamos el hotfix a la rama develop
$ git switch develop
Switched to branch 'develop'

$ git merge --no-ff hotfix-2.5.1
Merge made by recursive.

# Ya podemos eliminar la rama
```

```
$ git branch -d hotfix-2.5.1
Deleted branch hotfix-2.5.1 (was 44c2aff) .
```

¡Ojo! A veces es posible que ya tengas preparada una rama de *release*. Si es el caso y la falta de parche en `develop` no está bloqueando a ningún equipo, en lugar de fusionar el parche a la rama `develop` lo haremos directamente en la misma rama de *release* que haya creada, ya que esa misma rama será integrada en `develop`.

Resumen de ramas hotfix:

- *Se crean desde: `main`
- *Se fusionan en `develop` (o `release`) y `main`
- *Convención de nombre: `hotfix-<version>`

Algunas notas sobre Git Flow

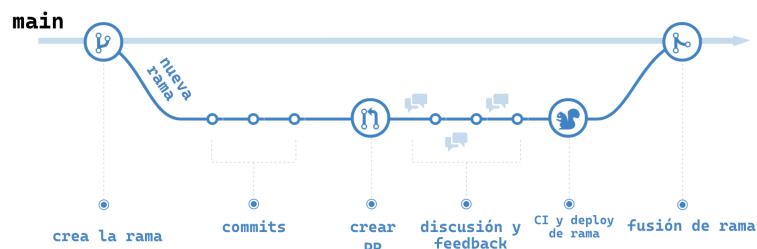
Git Flow es, y sigue siendo, una estrategia muy popular. No hay ninguna duda que hace 10 años aportaba orden y control a la hora de trabajar en equipo y, aún hoy, sigue contando con algunas ventajas importantes. Es especialmente interesante para productos fuertemente versionados o para equipos que necesitan una estrategia de trabajo muy estable y con reglas muy claras e identificadas.

Sin embargo, el propio *Vincent Driesen* (el creador de Git Flow) añadió en 2020 una nota respecto a Git Flow: el desarrollo del software ha evolucionado y, seguramente, **no siempre Git Flow es la mejor opción**. Por ejemplo, en la creación de web apps, **Git Flow añade complejidad innecesaria** ya que no es necesario mantener múltiples versiones y normalmente no se hace un rollback a través de Git.

Al final hay que tener en cuenta que cada proyecto y cada equipo tiene su propia estrategia de trabajo y, por lo tanto, **Git Flow no es una solución universal**.

GitHub Flow

GitHub Flow es una estrategia creada por la propia *GitHub* y pensada especialmente para equipos y proyectos que hacen despliegues de forma regular. Se basa en la creación de **Pull Requests** que serán discutidas para que se integren en la rama principal que siempre está actualizada con los cambios más recientes y preparada para ser desplegada.



**GitHub Flow es una alternativa más simple de Git Flow.
Tiene menos liturgias, es más fácil de entender y favorece
los despliegues continuos de tu proyecto**

Esta estrategia es muy utilizada especialmente en proyectos de código abierto ya que es una estrategia que elimina liturgias y simplifica la contribución de personas ajenas a la organización.

¿Qué es un Pull Request? Una Pull Request, o abreviado **PR**, es una petición de cambios que se envía a través de una rama de GitHub. Lo que se pide es que los cambios que presenta la rama sean incorporados a otra rama (que normalmente es la rama principal pero no necesariamente tiene que ser siempre así).

GitHub Flow tiene dos tipos de ramas:

- **main (o master):** La rama principal que contiene los cambios que se despliegan regularmente.

- Cualquier otra rama que quiere ser integrada en la rama principal.

Para trabajar de esta forma, primero creamos una rama desde la rama principal `main`:

```
# creamos una rama desde la rama principal main
$ git switch -c feature-new-cool-thing main
```

Luego añadimos todos los commits que consideremos importantes para la rama `feature-new-cool-thing`:

```
# añadimos un par de commits a nuestra rama
$ git commit -am "Add new cool thing"
[feature-new-cool-thing fb8f8f9] Add new cool thing

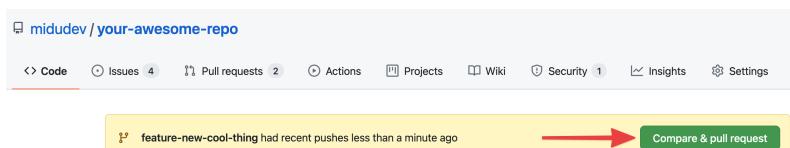
1 file changed, 1 insertion(+), 1 deletion(-)

$ git commit -am "Add cool thing to the README"
[feature-new-cool-thing fb8f8f9] Add cool thing to the README

1 file changed, 1 insertion(+), 1 deletion(-)

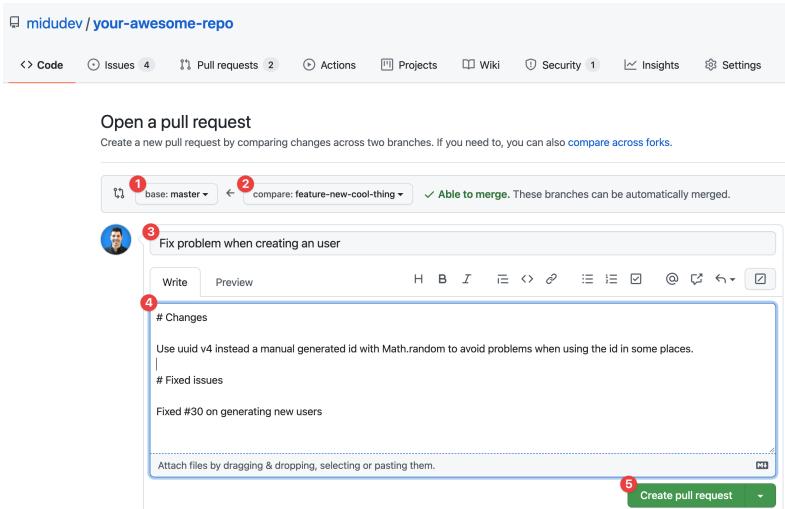
# sincronizamos esta rama al repositorio remoto con alias origin
$ git push origin feature-new-cool-thing
Total 0 (delta 0), reused 0 (delta 0), pack-reused 0
remote:
remote: Create a pull request for 'feature-new-cool-thing' on GitHub by \
visiting:
remote:     https://github.com/midudev/your-repo/pull/new/feature-new- \
cool-thing
remote:
To github.com:midudev/your-repo.git
 * [new branch]      feature-new-cool-thing -> feature-new-cool-thing
```

Ahora podemos acceder a GitHub y crear una nueva petición de cambios para integrar la rama `feature-new-cool-thing` a la rama principal:



Si entramos al repositorio, podemos ver que GitHub ahora muestra una notificación que nos invita a realizar una Pull Request de esta rama.

Si hacemos clic en el botón “*Compare & pull request*” nos abrirá la pantalla para crear una Pull Request, donde debemos seguir los pasos indicados.



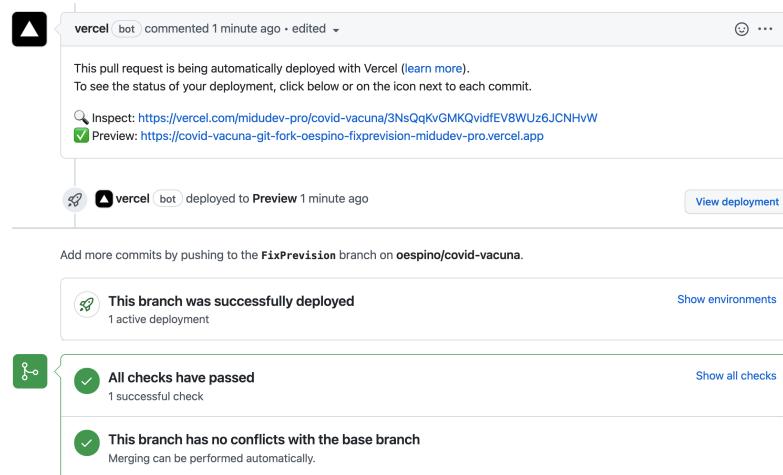
El paso 1 es la rama de destino, en este caso la rama principal “master”. El paso 2 es la rama que queremos fusionar. La 3 y la 4 nos deja dar más contexto sobre los cambios que queremos añadir. Finalmente el botón creará la Pull Request

Si tienes acceso de colaboración al repositorio, podrás crear directamente una Pull Request como hemos visto. Si no, deberás crear un fork como hemos visto en capítulos anteriores, para poder realizar una Pull Request desde tu fork.

Después de todo esto se iniciará una discusión por parte de la organización, las personas que colaboran en el proyecto y la comunidad respecto a los cambios que quieras incorporar. En ocasiones es posible que pidan (o exijan) que se añadan más cambios a la rama `feature-new-cool-thing` antes de que se pueda integrar a la rama principal.

Revisa la sección de **Buenas prácticas** para ver cómo puedes crear mejores PRs que sean más fáciles de ser aceptadas.

También es posible que exista algún proceso de revisión automatizada. A eso se le llama Continuous Integration (CI) y puede contener una serie de pruebas para verificar que el código funciona, es correcto y sigue los estándares de la organización. En ocasiones, incluso, se despliegan los cambios de la rama a una dirección para que todas las personas implicadas puedan ver los cambios funcionando.



En este repositorio, por ejemplo, se despliega automáticamente cada Pull Request con el servicio de Vercel y se ofrece en un comentario una dirección para visitar la web. ¡Muy útil!

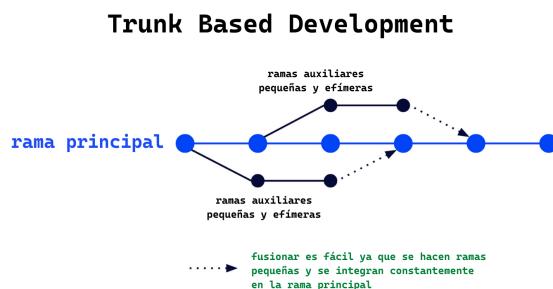
Una vez la revisión haya sido un éxito y las comprobaciones han pasado, alguien con suficientes permisos podrá aceptar integrar los cambios en la rama principal. A veces incluso el botón de “Fusionar” se activa después de pasar algunas comprobaciones y puedes hacerlo tú mismo.

Una cosa más. Aunque hemos visto todos los pasos como si estuviéramos usando GitHub como servidor donde hospedar el repositorio remoto, **esta estrategia es compatible con GitLab, Bitbucket y otras alternativas.**

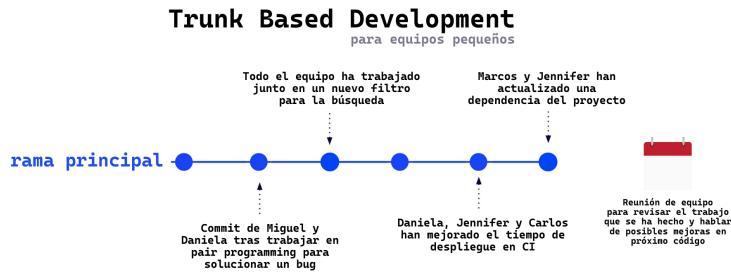
Trunk Based Development

El Trunk Based Development es una estrategia que se basa en que el mayor tiempo de desarrollo se concentra en una sola rama llamada *trunk* (tronco) que normalmente corresponderá con `main` (o `master`). Esto quiere decir que se evita la creación de ramas auxiliares y, sólo en algunos casos que se requieran, se crean con un tiempo de vida muy limitado (máximo un par de días).

Por raro que parezca, **esta estrategia es la más antigua de todas las que hemos visto**. Considera que, antiguamente, los equipos de desarrollo no contaban con las posibilidades de controles de versiones modernos y distribuidos, que facilitaban la creación de ramas que se mantenían actualizadas y que se pudieran integrar en otras ramas.



En esta estrategia se prioriza hacer commits directamente a la rama principal. En el caso de necesitar ramas, se hacen Pull Request pequeñas y que duren poco tiempo para ser integradas lo antes posible



En el caso de los equipos pequeños, la idea es que los commits a la rama principal sean constantes gracias a programar en pares o en grupo

Así que esta estrategia la podríamos resumir en: **haz commit a main y hazlo lo más frecuentemente posible con pequeños cambios.** Y crea PRs pequeñas y rápidas sólo cuando sea necesario.

¿Qué puede salir mal? Pues menos de lo que esperas si cuentas con un buen sistema de *Integración y Despliegue Continuo (CI/CD)*. De hecho, aunque puedes pensar que esta estrategia sólo tendría sentido para un proyecto muy pequeño, en realidad suele ocurrir casi lo contrario. Grandes empresas, como *Google* y *Facebook*, usan esta estrategia en un monorepositorio de miles de líneas de código.

Un monorepositorio no deja de ser un repositorio pero que contiene más de un proyecto o paquete publicable dentro. Tiene bastantes ventajas respecto a experiencia de desarrollo pero es un reto a la hora de crear una integración rápida.

¿Y cómo es posible que puedan hacerlo? Porque **satisfacen todas las consideraciones a tener en cuenta para poder usar Trunk Based Development en tu organización** con garantías:

1. Necesitas contar con un sistema de *Integración Continua (CI)* que permita verificar que el código funciona (*tests* útiles y con buena cobertura), es correcto y sigue los estándares de la organización (*lint*). De esta forma se automatiza gran parte del trabajo manual.

2. El equipo trabaja usando técnicas como la programación a pares (*pair programming*) o la programación en grupo (*mob programming*), que **permiten que los miembros de un equipo trabajen en conjunto** con otros miembros (del mismo u otro equipo) para resolver un problema.



Existe la falsa creencia que hacer Pair Programming hace que el equipo vaya más lento. Se ha demostrado que estas prácticas hacen que la aplicación tenga menos errores, el conocimiento se comparta más fácilmente y tareas complicadas sean más factibles

3. **Se deben hacer commits constantemente**, para que los cambios sean más fáciles de integrar. Cambios pequeños y frecuentes.
4. Existen redes de seguridad automatizadas que **deshacen un pase a producción en el caso que algo haya salido mal**. Por ejemplo, un sistema que analiza las métricas de conversiones de una tienda online que, si detecta que hay una desviación demasiado fuerte y negativa, devuelve la aplicación a una versión anterior (*rollback automatizado*).
5. Cada día es lo mismo para un desarrollador. O, dicho de otro modo, no hay “*días de despliegue*” o “*días de no-despliegue*” (como en el caso de los viernes). No hay *code freeze* que valga. No hay distracciones, la rama `trunk` siempre está disponible y siempre se puede desplegar.

En definitiva, **esta estrategia requiere que la organización sea responsable y madura** para evitar conflictos entre personas y errores en la aplicación.

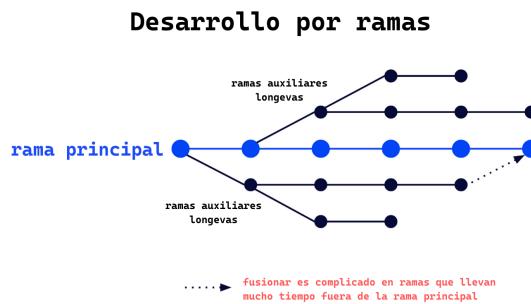
“Las ramas crean distancia entre desarrolladores y nosotros no queremos eso” (Frank Compagner de Guerrilla Games)

Beneficios de Trunk Based Development

Vamos a revisar algunos beneficios que trae seguir la estrategia de *Trunk Based Development* en tu organización o equipo:

Integración continua y menos fricción

La rama principal recibe constantemente un flujo de cambios pequeños. En lugar de estar trabajando en un pequeño silo durante largos períodos de tiempo y formarte una burbuja donde “*todo funciona*”, esto **hace que constantemente tu trabajo local esté sincronizado con el remoto** y, además, tu trabajo esté **constantemente siendo validado por los tests automatizados** y cobertura de código de la integración continua.



En las estrategias que se basan en ramas, la fusión de las ramas que llevan mucho en el proyecto suele traer problemas ya que se han alejado demasiado tiempo de la principal.

Menos trabajo manual

En lugar de crear muchas ramas y con gran cantidad de código, que luego deben ser revisadas por otros colegas del equipo, la estrategia de *Trunk Based Development* **confía en la automatización continua y en la colaboración entre el equipo de desarrollo** mientras se trabaja en esas nuevas características.

En el caso que una Pull Request sea necesaria, se debe hacer corta y concisa, de forma que revisarla de forma manual sea algo rápido y sencillo. Incluso útil para, simplemente, recibir feedback.

Despliegue a producción continuo

En lugar de tener días o momentos especiales para hacer despliegue a producción, **la rama principal llega constantemente al usuario final siempre que el sistema de integración continua haya pasado todas sus revisiones correctamente.**

La idea es mantener siempre la rama principal *en verde*. Esto quiere decir que la rama principal siempre tiene que tener todos los tests pasando, cumpliendo una cobertura de código y con todas las revisiones correctas.

La idea es hacer múltiples despliegues diarios a producción que incorporan cambios pequeños, incluso cambios que todavía no son visibles porque se encuentran detrás de un *Feature Toggle*. De esta forma, esas nuevas funcionalidades se pueden activar bajo demanda, para un pequeño grupo de usuarios, y desplegarlo al resto de usuarios de forma progresiva y controlada si no hay ningún problema.

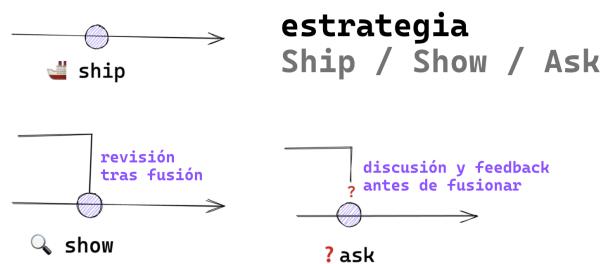
Para realizar *Feature Toggles* y *Progressive Delivery Experimentation* tienes herramientas como [Optimizely](#), [Split](#) o [Launch Darkly](#). También puedes hacerlo manualmente, aunque, dependiendo de lo que busques y las analíticas que quieras controlar, vas a tener que montar tu propia plataforma y esto puede ser costoso de crear y mantener.

Ship / Show / Ask

Ship / Show / Ask es una estrategia de ramas que combina la idea de crear Pull Request con la habilidad de seguir publicando cambios rápidamente. Fue [presentada por Rousan Wilsenach en el blog del mítico Martin Fowler](#).

Los cambios que creamos en el repositorio se categorizan en tres:

- *Ship*: Se fusiona en la rama principal sin revisión
- *Show*: Abre una petición de cambios para que sean revisados por CI pero se fusiona inmediatamente
- *Ask*: Abre una PR para discutir los cambios antes de fusionarlos



Las tres posibles categorías que pueden tener tus cambios
son los que dan nombre a la estrategia.

Ship

Ship significa que vamos a hacer un cambio **directamente a la rama principal**. No esperamos revisiones de código, ni integración. Vamos **directos a producción** (aunque antes del despliegue sí se harán los tests o checks pertinentes para evitar errores)



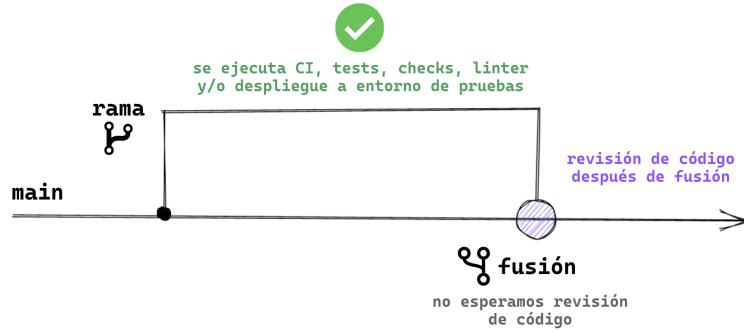
El commit va directamente a producción, sin pasar por Pull Request

Pensado para:

- He añadido una nueva funcionalidad con un patrón establecido.
- He arreglado de forma sencilla un bug por un error.
- Actualizaciones de documentación.
- Mejora de código por *feedback* del equipo o la comunidad.
- Se añaden nuevos *tests* para evitar errores.

Show

En este caso sí usamos *Pull Request* pero no esperamos revisiones manuales del código. Es decir, **esperamos que los tests automatizados**, pruebas de cobertura y validación de código sean exitosos **pero no que otra persona revise el código**.



En Show creamos Pull Request para validar que las pruebas automáticas pasan pero no esperamos opiniones de otras personas del equipo

Esto no quiere decir que no ocurran conversaciones sobre el código. La diferencia es que ocurrirán después de hacer la fusión de los cambios.

La idea es que el trabajo fluya hacia adelante, con el menor número de bloqueos, pero que sigan existiendo espacios dónde se pueda hablar y discutir sobre cómo mejorar las prácticas de desarrollo y el código que se crea.

El equipo es notificado que se creó una Pull Request, la revisan posteriormente y después se hacen las observaciones que sean necesarias. Lo interesante es que hace que esa PR queda fácilmente identifiable y separada.

Las *Pull Request* muchas veces se usan como una forma de **forzar la conversación dentro del equipo y compartir conocimiento**. A veces, puede ser buena idea. Pero **nunca deben ser una sustitución** a buenas dinámicas de trabajo en equipo y usar programación a pares o en grupo.

Pensado para:

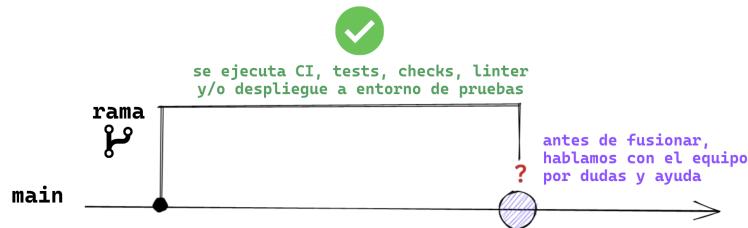
- Hacer arreglos necesarios para bugs y dejar constancia para que se aprenda.

- Crear pequeñas mejoras de código o *refactors*.
- Añadir nuevas funcionalidades siguiendo estructuras ya acordadas.
- Funcionalidades con pruebas automáticas.

Ask

Esta categoría es similar a *Show* pero aquí **sí esperamos al feedback de nuestro equipo** ante de fusionar la rama. Lo hacemos porque existe algo de incertidumbre: bien porque la solución es complicada, no sabemos implementarla, existen dudas...

La idea es que la rama dure el mínimo tiempo posible para no bloquear el trabajo de otros miembros del equipo



**Con Ask sí esperamos que el equipo revise la Pull Request.
No esperamos aprobación, esperamos conversación para
eliminar la incertidumbre.**

Una cosa importante a destacar es que decidir usar la categoría de *Ask* no quiere decir que esperemos la aprobación de nuestros colegas. Estamos abriendo una vía de conversación y debate antes de fusionar la rama ya que existe algún bloqueo o duda, pero es posible que no estemos buscando una revisión general de los cambios (si es así, deberíamos indicarlo).

Pensado para:

- Cuando es un trabajo muy grande y se necesita ayuda.

- Hay dudas sobre cómo hacerlo funcionar o la calidad del código.
- Existe incertidumbre sobre lo que estamos haciendo.
- Estamos esperando a que algo ocurra para poder fusionar la rama.

Las reglas de Ship / Show / Ask

Obviamente, poder llegar a seguir algunas de las categorías requiere algunas reglas.

1. Tenemos un buen sistema de *CI/CD*, fiable y rápido, que hace que la rama principal siempre sea desplegable y que evite que lleguen errores no deseados a producción.
2. Confiamos en el equipo y existen buenas prácticas de desarrollo. Pair programming, mob programming, seniority... y, sobretodo, existe responsabilidad. **La persona se responsabiliza de decidir la categoría de su cambio.** Un gran poder, poder hacer *merge* de tus propias Pull Request, conlleva una gran responsabilidad (no romper producción).
3. Las revisiones de código no son requerimientos para que las PRs sean fusionadas.
4. Las ramas son lo más pequeñas posibles, tienen un tiempo de vida corto y siempre salen directamente desde la rama principal.
5. El equipo ha sabido lidiar con el ego individual, las personas confían en el resto del equipo y las pruebas automáticas pasan. El equipo entiende que la rama principal puede contener código sin terminar detrás de *Feature Flags* u otros mecanismos similares.

Últimas palabras sobre Ship / Show / Ask

Creo que la idea detrás de esta estrategia, en realidad, es la de **responsabilizar a cada persona con su trabajo, empoderar al equipo, darle autonomía y tratar a la gente que lo integra por igual.**

También está que el desarrollo sea **más rápido** y la entrega sea más continua, pero para poder lograrlo se va a necesitar mucha confianza. Algo que, seguramente, **no todos los equipos estén preparados de inicio.**

En cualquier caso creo que *Ship / Show / Ask* podría ser una mezcla de estrategias que ya hemos visto anteriormente... simplemente deja la puerta abierta a que cada persona decida por sí misma cuál es la mejor categoría para cada cambio y le pone nombre.

Conclusiones sobre las estrategias de flujos de trabajo en Git

En este capítulo hemos visto algunas de las estrategias más famosas para trabajar con ramas y en equipo con Git. Son opciones muy interesantes pero, por favor, **no adoptes ninguna sin análisis previo y, sobretodo, ten en cuenta que las estrategias no son mantras inalterables.**

La estrategia que sigamos tiene que responder a la **entrega de valor continua, la experiencia de desarrollo y la calidad del software** que entregamos.

En mi empresa seguimos GitHub Flow. Nos ha ayudado a realizar despliegues a producción de una forma mucho más continua y ha simplificado tanto la estrategia que hay poco que explicar.

En nuestro caso tiene todo el sentido ya que trabajamos con aplicaciones web, que no requieren de un versionado estricto y, en caso de problema, retroceder a una versión anterior es muy fácil (cambiamos el contenedor de Docker por uno anterior y listo).

En el caso del desarrollo de otro tipo de software, como un videojuego, puede tener más sentido usar una estrategia de versionado más estricto como **Git Flow**. No quiere decir que no funcione otra o que tu equipo deba seguir esa... de hecho, [los creadores de Killzone y Horizon Zero Dawn usan Trunk Based Development.](#)

Lo que quiero decirte con esto es que **no hay una bala de plata**. Cada equipo, cada desarrollo, cada proyecto... son circunstancias diferentes que, seguro, hay que lidiar de forma distinta.

¿A qué deberíamos aspirar?

Si me preguntas a qué deberíamos aspirar como profesionales que trabajan en un proyecto de software, en equipo y ejercen la ingeniería... Creo que **la**

aspiración es hacer commits en la rama principal continuamente evitando la creación de ramas auxiliares.

Esto sería aproximarse al máximo lo que hacemos en *Trunk Based Development* o lo que propone *Ship/Show/Ask*. Con las adaptaciones, según el equipo y el proyecto, que se requieran.

Obviamente, este tipo de estrategia requiere una buena disciplina, *seniority*, responsabilidad individual y de equipo, un buen sistema de *CI/CD* tanto en fiabilidad como velocidad y, por supuesto, un **buen trabajo en equipo**.

No es algo que se puede hacer de un día para otro pero creo que se pueden ir adoptando pequeñas mejoras en el equipo para poder aspirar a ello.

Primero, **limitar la vida de las ramas que se crean**. Evitar ramas que duren más de dos o tres días.

Segundo, **construir un mejor sistema de CI/CD**. ¿Tienes tests automatizados? ¿Qué cobertura tienes? ¿No lo sabes? Empieza por ahí. Construye las bases de una mejor estrategia a través de sacar datos objetivos del punto en el que te encuentras. ¿Cuanto tiempo tardas en conseguir que un commit que se crea llegue a producción? Su viaje debe ser lo más corto posible. Descubre en qué puntos se bloquea. ¿Es por revisión manual? ¿Hay conflictos?

Tercero. Una vez tengas mejoras en integración continua, **abre la puerta a hacer commits a la rama principal cuando trabajes en equipo**. Empezad a hacer alguna pequeña tarea en grupos de tres o más personas o, como mínimo, a pares.

Y, así, **pequeñas mejoras incrementales** en el equipo, en tu repositorio y, claro, la organización. Hasta llegar a una nueva estrategia más eficiente para tus desarrollos.

No intentes cambiar de estrategia de trabajo en equipo de golpe. Adopta el método *kaizen*. Esta filosofía japonesa defiende que cambios **pequeños continuados dan mejores**

resultados y tiene más probabilidades de éxito que un único cambio grande.

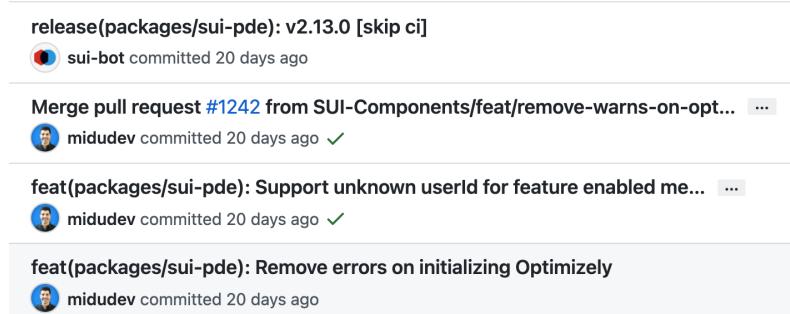
Buenas prácticas al trabajar con Git

¿Cada cuánto debería hacer un commit?

A menudo. Es mejor hacer commits pequeños, agrupando pequeñas mejoras o acciones, que un commit con todo lo que se quiere hacer.

Como decía Julio César: *divide y vencerás*. Divide la tarea en trozos pequeños. Cada trozo es un commit. Avanza en tu tarea y plasma esos cambios con frecuencia.

Esto, además de ayudarte con la productividad, también puede ser útil para tener siempre una copia de seguridad de tu trabajo o, también, para que cualquier otra persona pueda retomar tu trabajo desde el punto que lo dejaste.



La lista de commits cuenta la historia del proyecto. Leerlo debería ayudar a cualquier persona a entender la evolución del proyecto.

También es útil que sincronices regularmente la rama de trabajo con la rama en la que pretendes fusionar tus cambios mientras estás trabajando en ella. De esta forma evitarás conflictos en el futuro y si los hay serán lo suficientemente pequeños para que puedas lidiar con ellos.

Hacer commit a menudo no significa que debas hacer commits sin sentido. Graba tus progresos en iteraciones pequeñas pero que tengan un significado y que, si puede ser, no deje tu aplicación o proyecto sin funcionar.

¿Cómo puedo escribir un buen mensaje de commit?

Escribir buenos mensajes de commit es importante para que el histórico de tu proyecto sea legible, fácilmente escaneable y, obviamente, comprensible por cualquier persona que participe en el proyecto.

Por ello voy a darte **6 reglas para escribir un buen mensaje de commit:**

1. Usa el verbo imperativo (`Add`, `Change`, `Fix`, `Remove`)

Aunque el mensaje puede sonar un poco brusco, el verbo en forma imperativa es una buena manera de expresar la acción que se realiza en el commit. Por ejemplo, `Add` significa que se añade un nuevo archivo, `Change` significa que se modifica un archivo existente y `Fix` significa que se arregla un bug.

	COMMENT	DATE
○	CREATED MAIN LOOP & TIMING CONTROL	14 HOURS AGO
○	ENABLED CONFIG FILE PARSING	9 HOURS AGO
○	MISC BUGFIXES	5 HOURS AGO
○	CODE ADDITIONS/EDITS	4 HOURS AGO
○	MORE CODE	4 HOURS AGO
○	HERE HAVE CODE	4 HOURS AGO
○	AAAAAAA	3 HOURS AGO
○	ADKFJSLKDFJSOKLFJ	3 HOURS AGO
○	MY HANDS ARE TYPING WORDS	2 HOURS AGO
○	HAAAAAAAAANDS	2 HOURS AGO

AS A PROJECT DRAGS ON, MY GIT COMMIT MESSAGES GET LESS AND LESS INFORMATIVE.

Ejemplo de cómo los mensajes se van degenerando...

Sé que muchas veces estamos tentados a escribirlo en pasado “*Added...*”, “*Fixed...*” o “*Removed...*” pero cada commit hay que entenderlo como una instrucción para cambiar el estado del proyecto. Dicho de otro modo, el verbo imperativo nos permite saber en qué estado queremos que el proyecto se encuentre en el momento de añadir el commit.

Sólo hay que ver también los mensajes de commit que el propio Git nos añade. Al hacer merge de una rama, por ejemplo, usa `Merge branch`.

Lo mejor es que el mensaje del commit complete esta frase: “*Si aplico este commit, entonces este commit...*”

- ...add a new search feature
- ...fix a problem with the topbar
- ...change the default system color
- ...remove a random notification

2. No uses punto final ni puntos suspensivos en tus mensajes

Usar puntuación, más allá de las comas, es innecesario a la hora de crear un buen mensaje de commit. Cada carácter cuenta a la hora de describir un cambio, así que no lo desperdigues con puntos innecesarios.

```
git commit -m "Add new search feature." # Mal. Con punto final.  
git commit -m "Fix a problem with the topbar..." # Mal. Puntos suspensi\\  
vos.  
git commit -m "Change the default system color" # Bien.
```

¿Por qué? **El primer mensaje de commit es el título del commit.** Y según las reglas de puntuación para títulos, tanto en castellano como en inglés, estos no llevan puntuación final. Sobre los puntos suspensivos... ¡Nuestros commits no deberían tener ningún suspense! Deben ser una instrucción clara y concisa.

Si te fijas, los commits que genera GitHub no tienen puntos suspensivos ni punto final en ningún caso.



Los mensajes que crea GitHub al fusionar una rama no usa puntuación al final. Para seguir esta regla, debemos tratar el primer mensaje del commit como el título o asunto.

3. Usa como máximo 50 caracteres para tu mensaje de commit

Sé corto y conciso. Si tienes mucho que explicar es probable que tu commit contenga demasiados cambios. ¿Puedes separarlo en diferentes commits? Pues entonces hazlo.

Haz que el mensaje sea claro, conciso y directo para que realmente refleje los cambios que contiene.

```
# MAL. Muy largo.  
git commit -m "Add new search feature and change typography to improve \  
performance"  
# BIEN. Corto, comprensible.  
git commit -m "Add new search feature"  
# BIEN. Al grano pero con detalle.  
git commit -m "Change typography to improve performance"
```

4. Añade todo el contexto que sea necesario en el cuerpo del mensaje de commit

A veces necesitas proveer de **más contexto a tu commit**. Para ello, en lugar de saturar el sumario del commit, añade información que sea necesaria en el cuerpo del mensaje.

Puedes lograrlo usando `git commit -m "Add summary of commit" -m "This is a message to add more context."` pero en estos casos lo mejor es que uses directamente `git commit` de esta forma:

```
git commit
```

Esto te permitirá añadir fácilmente un mensaje de commit con saltos de línea usando el editor.

Como hemos comentado antes, el primer mensaje de commit es el título del mismo. Pero el resto de mensajes pertenece al cuerpo y, por lo tanto, sí debes usar todas las reglas de puntuación que tendría un texto normal.

5. Usa un prefijo para tus commits para hacerlos más semánticos

Cuando un proyecto crece, es necesario que existan ciertas reglas para que el historial sea legible. Para ello, puedes añadir un prefijo para darle más significado a los commits que realizas. A esto se le llama *commits semánticos* y se haría de la siguiente manera:

```
tipo-de-commit(scope) : descripción>
```

Por ejemplo:

```
feat(search) : add new search feature
^--^-----^ ^-----^
| | |
| |     L--> # Descripción del cambio
| |
|     L--> # Contexto del cambio
|
-----> # Tipo del cambio
```

En *mono-repositorios multi-paquete*, puedes añadir también la información del paquete que es afectado por el commit. Se le conoce como `scope` y sería de la siguiente forma:

```
feat(backend) : add filter for cars
fix(web) : remove wrong color
```

Sobre el tipo de cambio, lo mínimo es diferenciar entre `fix` y `feat` pero existen diferentes convenciones. Una de ellas, bastante famosa, es la [convención que sigue el framework Angular](#).

Estos serían los prefijos:

- **feat**: para una nueva característica para el usuario.
- **fix**: para un bug que afecta al usuario.
- **perf**: para cambios que mejoran el rendimiento del sitio.
- **build**: para cambios en el sistema de build, tareas de despliegue o instalación.
- **ci**: para cambios en la integración continua.
- **docs**: para cambios en la documentación.
- **refactor**: para refactorización del código como cambios de nombre de variables o funciones.
- **style**: para cambios de formato, tabulaciones, espacios o puntos y coma, etc; no afectan al usuario.
- **test**: para tests o refactorización de uno ya existente.

Otra ventaja muy importante de utilizar commits semánticos es que podrás leer el historial para publicar nuevas versiones de un paquete, desplegar nuevas versiones de una aplicación o generar un CHANGELOG con todos los cambios.

6. Considera usar utilidades para hacer commit

Puedes usar `husky` para ejecutar scripts o comandos antes de realizar diferentes acciones sobre el repositorio, gracias a los hooks de git. Por ejemplo, puedes ejecutar los tests antes de subir los cambios al repositorio remoto.

```
# instalamos las dependencias
npm install husky --save-dev
npx husky install

# iniciamos husky en nuestro repositorio
npm set-script prepare "husky install"
npm run prepare

# creamos un hook para que pase los tests antes de hacer push
npx husky add .husky/pre-push "npm test"
git add .husky/pre-push

git commit -m "Keep calm and commit"
# los tests se ejecutarán antes de realizar el push
git push -u origin master
```

Con `commitlint` puedes asegurarte de que los commits sean semánticos, legibles y sigan una convención que elijas.

Para instalar `commitlint`:

```
# Install commitlint cli and conventional config
npm install --save-dev @commitlint/{config-conventional,cli}

# For Windows:
npm install --save-dev @commitlint/config-conventional @commitlint/cli

# Añadir hook para revisar el mensaje de commit
npx husky add .husky/commit-msg 'npx --no-install commitlint --edit "$1\"'
```

Puedes usar sistemas como `conventional-changelog` para leer el `CHANGELOG` y generar nuevas versiones o publicar paquetes. También con `commitizen` puedes usar una línea de comandos que te haga elegir el tipo de commit y así no tener que depender de realizar esto manualmente en el propio mensaje.

¿Cómo puedo escribir un buen nombre de rama?

Al crear una rama en Git existen multitud de posibles convenciones. Voy a darte algunos consejos que seguro te ayudarán a la hora de elegir un buen nombre para tu rama. Sin embargo, me gustaría que también tengas en cuenta que **dependiendo de la metodología de trabajo de tu organización, empresa o equipo, estos consejos pueden ser más o menos válidos**.

Estos consejos están basados en mi experiencia después de muchos años trabajando en diferentes empresas y equipos de desarrollo.

1. Sé consistente al nombrar tus ramas

Sea como sea que al final decidas crear los nombres de las ramas, deberías usar siempre el mismo patrón. Sé consistente y documenta si hace falta las decisiones tomadas, de forma que todo el equipo de trabajo pueda entender las reglas que hay que seguir.

Por ejemplo, la primera decisión sería hacer que los nombres de las ramas sean todo en minúscula y que las palabras sean separadas por un guión -. Sea esto así o de otra forma si prefieres, lo importante es que todas las ramas sigan el patrón decidido.

2. Usa el nombre de la acción que se realiza en la rama

De la misma forma que en los commits semánticos indicábamos qué tipo de acción realiza el commit, también en las ramas se puede hacer algo similar. No hace falta ser tan específico como el commit pero sí puede ayudarte a saber de qué trata la rama rápidamente.

- **bug**: Cambios de código para arreglar un bug conocido.
- **feature**: Desarrollo de una nueva característica.
- **experiment**: Experimentos que nunca serán fusionados.

- **hotfix**: Cambio rápido de un error crítico.

Con esto en mente, algunos ejemplos de nombre de ramas serían:

```
bug/avoid-creating-lead-twice  
feature/add-new-user-form  
experiment/try-new-ui-design  
hotfix/fix-typo-in-name
```

3. Usa los IDs de JIRA o el sistema de tickets que uses

Aunque la convención anterior es una buena forma de identificar de qué tipo de rama se trata, es muy importante que el nombre de la rama sea único. Y lo cierto es que usando palabras no es difícil encontrarte dos veces la rama `hotfix/fix-typo..`

Además, parece que el nombre de la rama a veces no da el suficiente contexto para saber realmente en qué trabaja o qué soluciona. Para ello, una buena idea es adjuntar al principio del nombre de la rama la ID del ticket o de la issue que esté asociada. Eso, obviamente, si estás usando algún sistema para gestionar tu proyecto (que seguramente debería ser así, por rudimentario que sea el sistema).

De esta forma, tus ramas podrían ser así:

```
989-hotfix/fix-typo-in-name  
1110-feature/add-new-user-form  
1240-experiment/try-new-ui-design  
1255-hotfix/fix-typo-in-name
```

Ahora es mucho más fácil buscar más contexto sobre estas ramas, pese a que no quede claro con su propio nombre. Además, esto hace que buscar ramas sobre soluciones anteriores sea mucho más sencillo. De esta forma puedes ejecutar `git checkout` o `git switch` seguido del inicio del número de ticket o issue para ver si ya existe una rama creada y, si lo está, ver qué cambios se han hecho en ella.

```
$ git switch 12[pulsa-tab]  
1240-experiment/try-new-ui-design  
1255-hotfix/fix-typo-in-name
```

¿Debería alterar el historial de mi proyecto?

No. Excepto si tienes muy buenas razones. **Y tienen que ser muy buenas.**

La única buena razón para hacer esto es que has publicado una contraseña, una llave de una API o información sensible que no debería estar en el historial. Incluso en ese caso, si es posible, **es mejor idea reiniciar la contraseña o la llave.** ¿Por qué? Porque esa información ya es vulnerable al haber sido expuesta y borrarla del historial no garantiza nada.

Si eso no fuese posible, entonces sí puedes hacer un `git rebase` para cambiar el historial de tu repositorio. Pero ten en cuenta que esto no es normalmente una buena idea y es mejor evitarlo. Piensa que al cambiar el histórico, todos las personas con acceso al repositorio remoto deberán sincronizar de nuevo correctamente sus repositorios locales.

Si, por ejemplo, hay un error en el código o has subido un commit que está mal, entonces siempre es mejor, si es posible, revertir el commit con `git revert` y dejar reflejado en el historial que se hizo ese cambio.

De hecho, dejar en el historial este tipo de errores es una buena idea. Los errores y fallos son una parte importante de la evolución de un proyecto y es importante que sean reflejados en el historial. **Para no repetirlos en el futuro lo mejor no es ocultarlos, es más bien todo lo contrario.**

Un sitio donde puede tener sentido hacer `rebase` es en las ramas. Cuando estás desarrollando en una rama, puedes hacer `rebase` con la rama principal. De esta forma recribes el historial de la rama de desarrollo y puedes evitar los commits de `Merge` que pueden añadir ruido al historial.

No hagas commit de código generado ni configuración particular

Un error muy común que se puede ver en repositorios es el de guardar archivos que han sido generados automáticamente por algún proceso o el de guardar configuración de un usuario o editor en concreto. Por ejemplo, imagina que tienes una web y que antes de desplegarla tienes que generar una carpeta `build` con los archivos estáticos minificados y listos para subir. Pues bien, la carpeta `build` debería ser ignorada por Git.

Esto se hace por diversos motivos:

- Añaden ruido innecesario al historial de cambios y se puede hacer molesto a la hora de revisar código.
- A la larga hará que nuestro repositorio se vuelva más pesado por culpa de estos ficheros que cambian constantemente y pueden ser bastante complejos.

Así que **no lo hagas**. Usa correctamente tu archivo `.gitignore` para evitar controlar estos ficheros en tu repositorio.

Respecto a configuraciones particulares, como la configuración del editor, existen normalmente mejores soluciones a Git. Ten en cuenta que esta configuración puede entrar en conflicto con la configuración de otra persona que también esté trabajando en ese mismo proyecto.

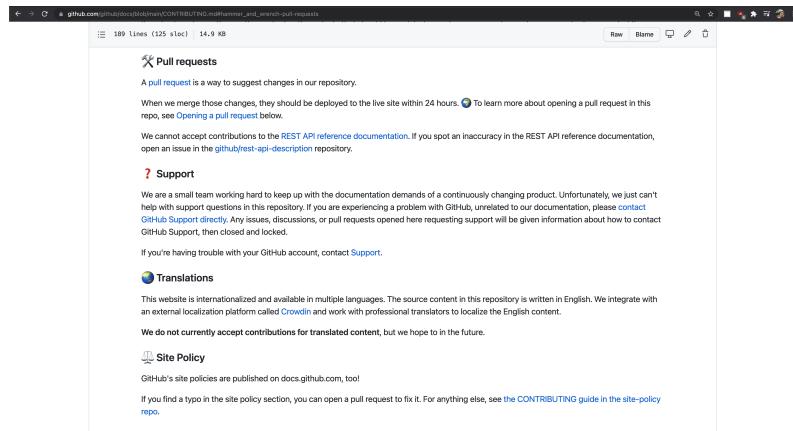
¿Qué debo tener en cuenta para hacer una buena Pull Request?

Crear una buena Pull Request es muy importante para conseguir que tus contribuciones lleguen a ser fusionadas en la rama principal del proyecto. Y no hay nada más frustrante que crear código para que luego no se haga nada con él... ¡Así que revisa estas reglas para que tus PRs lleguen a buen puerto!

1. Léete el archivo CONTRIBUTING.md o sigue el estilo del repositorio

Antes de querer contribuir a un proyecto, **busca el archivo CONTRIBUTING.md que se suele encontrar en la raíz del repositorio.** No todos los proyectos cuentan con uno pero, si lo tiene, es vital que lo leas, ya que normalmente explica todo lo que debes tener en cuenta antes de abrir una *Pull Request*.

Si no lo encuentras, no te preocupes. Estudia el repositorio. ¿Qué estilo de commits utilizan? ¿Qué nombre de ramas? ¿Usan un estilo en concreto a la hora de escribir el código? Sigue al máximo la forma de trabajar de los demás.



La propia GitHub tiene muchos repositorios públicos con un fichero Contributing. Revísalos porque a veces no aceptan ciertos tipos de contribuciones.

2. Respeta el estilo del repositorio

A veces un repositorio de código sigue un estilo particular. Por ejemplo, algunos de los repositorios de JavaScript pueden no usar puntos y coma. Otros sí. Y, en ocasiones, tienen un linter configurado para revisar que el código siga ese estilo.

Independientemente de que tenga linter configurado o no, **es importante que el código que subes siga el estilo del repositorio**. Por ejemplo, no te dediques a cambiar en tu código las comillas simples por las dobles, sólo por el simple hecho que te gustan más.

Si lo haces, **vas a distraer a las personas que revisan tu código** y pueden no aceptar tus cambios por traer cambios que no vienen a cuento. Así que evítalo a toda costa.

Si todavía crees que es imperativo el cambio de estilo... lo mejor es que primero abras una *issue* en el repositorio y expongas las razones por las que crees que es necesario.

3. Enfoca tu código en una sola cosa

¿Quieres arreglar un *bug* de una aplicación? ¿Quieres añadir una nueva funcionalidad? ¿Quieres añadir un nuevo test? ¿Hay que mover un directorio? Pues separa cada una en una Pull Request diferente.

Es mucho más fácil revisar y aceptar una Pull Request que hace una sola cosa a revisar una Pull Request que aprovecha a hacer muchas cosas.

Por ejemplo: Imagina que quieras arreglar un bug y que, hacerlo, es una sola línea de código. Pero te das cuenta que no te gusta la localización del fichero y, además de arreglar el problema, decides mover el fichero a otro lugar.

Esto hará que en la Pull Request tu cambio, que parecía sencillo, se convierta en una gran tarea y que la diferencia de código entre la versión actual y la que tu quieras fusionar ahora sean todas las líneas del fichero.

Es mejor, en ese caso, que hagas dos Pull Requests. Una para arreglar el bug y otra para mover el fichero.



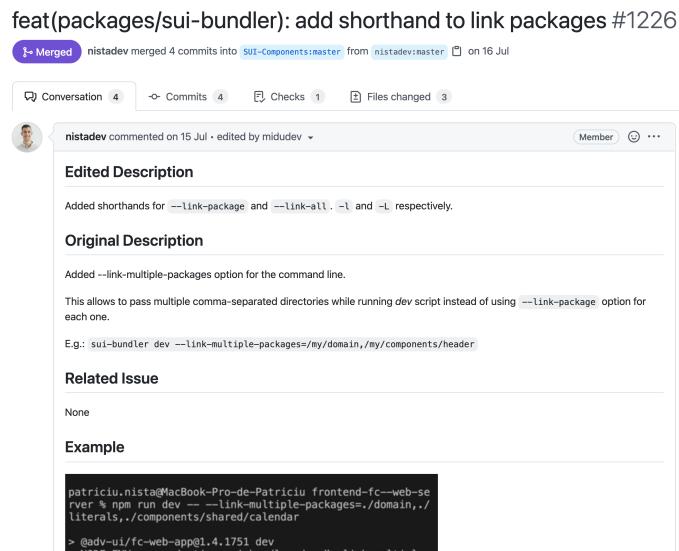
Es evidente que cuantas más líneas de código tenga que revisar una persona... más nerviosa se va a poner. Menos es más.

Lo mejor que puedes hacer al crear una Pull Request es empatizar con las personas que van a revisarla. ¿Cómo te sentirías tú ante una PR de mil líneas de código? ¿Cómo te sentirías tú ante una PR de una línea de código?

Esto no quiere decir que hagas una Pull Request para cada commit. Una Pull Request puede, y seguramente debe, tener más de un commit que explique cada pequeño cambio que se ha realizado para lograr la finalidad de la Pull Request.

4. Explica tu Pull Request

Mucha gente dice que el código habla por si solo y, aunque es genial que tu código sea lo suficientemente explicativo para que la gente sepa lo que hace de un vistazo, es muy buena idea que tu Pull Request sea acompañada de una buena explicación de qué has hecho y quéquieres conseguir, además de cualquier otra información que pueda ayudar a la gente a tomar una decisión con tu código.



Añadir una descripción a tu Pull Request descriptiva e incluso actualizarlo si es necesario, puede ayudar a ser mejor evaluada

Y si una imagen vale más que mil palabras. ¿Qué puede valer un GIF o un vídeo mostrando la funcionalidad? Existen muchas aplicaciones y herramientas que nos permiten grabar la pantalla. Yo en mi caso uso en macOS la aplicación [CleanShot](#).

Cambia tu rama master a main o similares

Desde el 1 de octubre de 2020, [GitHub cambió el nombre de la rama principal](#) por defecto de los repositorios. Su nombre pasó de ser `master` a `main`. Más tarde, el 10 de marzo de 2021, [GitLab anunció lo mismo](#).

Este cambio viene por las **connotaciones racistas** que tienen los términos `master` y `slave`. En el año 2020 surgieron muchas protestas por el uso de terminología anticuada o con percepción discriminatoria en muchos proyectos tecnológicos (otro ejemplo es el uso de `blacklist` para algo negativo y `whitelist` para algo positivo).

En junio de 2020 la *Software Freedom Conservancy* publicó una [declaración donde resumía las razones por las que el término `master` era ofensivo](#).



El nombre de `master` como rama principal va a quedar completamente obsoleto y, poco a poco, veremos menos repositorios usándolo

Pese a que tú, personalmente, no te sientas ofendido por este término lo cierto es que **algunas personas sí sienten como un agravio este uso y, simplemente, por empatía se puede entender que es una buena idea cambiar la rama principal por defecto**.

Si en realidad no estás de acuerdo con este cambio y las razones que hay detrás... igualmente te recomiendo adoptar el cambio. Al final hay que entender que es el nuevo nombre de rama principal que se va a usar en todos los proyectos y servicios, por lo que **cuanto antes lo asimiles, mejor**.

De hecho, **ten en cuenta que desde la versión 2.28.0 de git, después de iniciar un repositorio local, ya te advierte que usar master como rama principal va a cambiar en próximas versiones.**

```
hint: Using 'master' as the name for the initial branch.  
hint: This default branch name is subject to change.  
hint: To configure the initial branch name to use  
hint: in all of your new repositories, which will suppress  
hint: this warning, call:  
hint:  
hint:   git config --global init.defaultBranch <name>  
hint:  
hint: Names commonly chosen instead of 'master' are  
hint: 'main', 'trunk' and 'development'. The just-created  
hint: branch can be renamed via this command:  
hint:  
hint:   git branch -m <name>
```

Mi recomendación es que ejecutes `git config --global init.defaultBranch main` en tu terminal, de forma que por defecto use la rama `main` en todos tus nuevos repositorios nuevos. Así encajará con **GitHub**, que es uno de los servicios más usados para hospedar repositorios, y otros servicios que, seguro, también van a adoptar este cambio por convergencia.

Ten en cuenta que **la rama principal, como hemos explicado en el libro, se puede llamar con el nombre que prefieras**. Si ha sido `master` ha sido, realmente, porque históricamente ha sido así pero nada impide que ya existiesen repos con nombres para su rama principal como `develop` o `trunk`. Al final parece que la preferida va a ser `main` pero puedes usar tu creatividad si lo prefieres.

Ahora... ¿Deberías cambiar tus ramas `master` a `main`? Si estás creando un repositorio nuevo es obvio que es buena idea crearla ya con `main` como rama principal. Pero... ¿qué pasa con los repositorios que ya tenemos?

Cómo migrar la rama principal de `master` a `main`

Primero, tenemos que mover la rama `master` a `main`. Para ello usamos el comando `git branch`. El parámetro `--move` es el que indica que es un

movimiento de rama o renombre. De esta forma todo el historial de commits de `master` estará disponible también en la nueva rama.

```
$ git branch --move master main
```

Ahora, **tenemos que actualizar el repositorio remoto** (si es que lo tenemos). Recuerda que, si no, el cambio se quedaría sólo en tu máquina. También debemos usar el parámetro `--set-upstream` para indicar que también será la rama remota por defecto. De esta forma, cualquier `git pull` que hagamos más tarde, se hará sobre la rama `main` remota.

```
# actualizamos el repositorio remoto origin  
# y además hacemos que la rama main sea la rama remota por defecto  
$ git push --set-upstream origin main
```

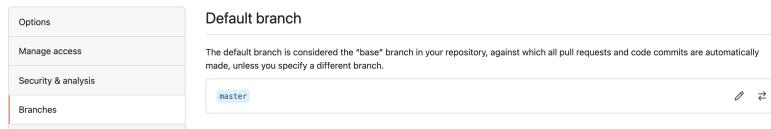
```
Total 0 (delta 0), reused 0 (delta 0), pack-reused 0  
remote:  
remote: Create a pull request for 'main' on GitHub by visiting:  
remote:     https://github.com/midudev/react-slidy/pull/new/main  
remote:  
To github.com:midudev/react-slidy.git  
 * [new branch]      main -> main  
Branch 'main' set up to track remote branch 'main' from 'origin'.
```

Ahora tenemos que asegurarnos que nuestro puntero `HEAD` apunta también a esta nueva rama remota `main`.

```
# movemos el puntero HEAD a la rama main remota de origin  
$ git remote set-head origin main  
# para asegurarnos que funciona podemos ejecutar  
$ git branch -r  
origin/HEAD -> origin/main # esto es lo que queremos
```

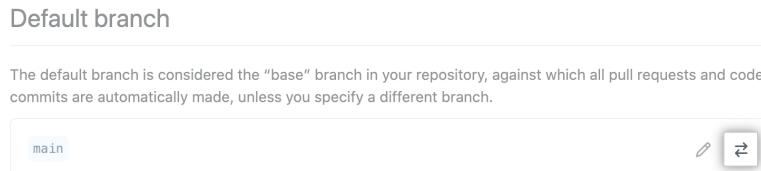
Ahora tenemos que cambiar la rama por defecto en el servicio de hospedaje de repositorios que usemos. Vamos a ver cómo hacerlo en **GitHub** pero en otros productos sería muy similar.

En el caso de **GitHub**, tenemos que ir a nuestro repositorio en GitHub y `Settings -> Branches`.

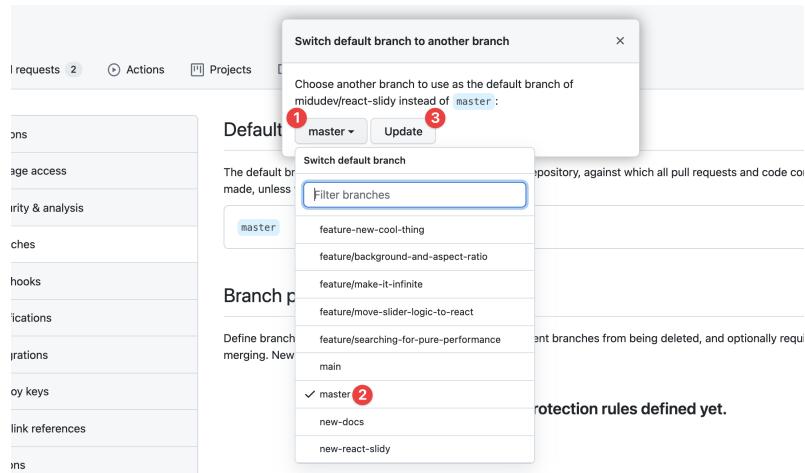


Cambia la rama por defecto de master a main

Debajo del título *Default Branch*, encontrarás la rama principal (que debería ser `master` o la que sea que estuvieses usando hasta ahora). Encontrarás dos iconos a su derecha. Un lápiz y dos flechas. Pulsa las dos flechas, haz clic en el desplegable y selecciona la nueva rama `main`. Finalmente, pulsa `Update`.



Pulsa las dos flechas para que se abra una ventana flotante donde seguiremos con el cambio de rama por defecto



Renombrando la rama de master a main

Te aparecerá un aviso **indicando que cambiar tu rama por defecto puede tener consecuencias no esperadas** que pueden afectar a nuevas PRs y

clonaciones. Pulsamos el botón que dice que lo entendemos y que queremos actualizar la rama igualmente... ¡y ya lo habremos conseguido!

¿Qué consecuencias inesperadas tiene esto? Pues bastantes. Para empezar si usas *Travis*, *CircleCI* o *GitHub Actions*, deberás asegurarte que ahora escuchan cada push a la nueva rama `main` en lugar de la que tenía antes. Otro ejemplo puede ser los servicios de *Vercel* o *Netlify* que, normalmente, hacen despliegues automáticos dependiendo de la rama. Revisa sus configuraciones para asegurarte que usan la rama correcta para los despliegues a producción. También, como veremos más adelante, otras personas tendrán que hacer cambios a su entorno local.

Sin embargo, todavía nos queda una cosa importante. **Eliminar nuestra rama `master`.** De lo contrario, todavía estará por ahí y las personas que trabajan en este repositorio podrían confundirse, intentar usarla y actualizarla.

Para ello vamos a actualizar el repositorio remoto `origin` y eliminar la rama `master` de ahí.

```
$ git push origin --delete master
To github.com:mividudev/react-slidy.git
 - [deleted]          master
```

Ahora mismo tú lo tienes todo solucionado. ¡Felicidades! Pero... ¿Qué pasa con las personas que colaboran contigo? Tienen dos opciones: eliminar su repositorio local y volver a clonarlo (la solución que siempre funciona) o podemos ejecutar una serie de comandos para asegurarnos que usamos la rama correcta.

```
# vamos a la rama master
$ git checkout master

# recuperamos todas las ramas disponibles del repo remoto
$ git fetch --all
Fetching origin

# actualizamos el puntero HEAD
# para que apunte a la rama principal
# del repositorio remoto
$ git remote set-head origin --auto
origin/HEAD set to main

# cambiamos nuestra rama local
```

```
# para que use la nueva rama remota main
$ git branch --set-upstream-to origin/main
Branch 'main' set up to track remote branch 'main' from 'origin'.
```

```
# (opcional) renombramos nuestra rama local master
$ git branch -m master main
```

Firma correctamente tus commit con GPG

Al inicio del libro, en la sección de *Instalando y Configurando Git*, hemos visto cómo podíamos configurar Git para que *firme* los commits con nuestra autoría. Para ello, tenemos que configurar el nombre y correo electrónico.

Esto es importante ya que va a **ayudar al equipo a saber quién realizó ciertos cambios** y pedirle apoyo en el caso que algo no quede claro. Hay que verlo como una forma de poder conseguir más contexto y **no de control de quién ha cometido un fallo**.

Lo haríamos con dos comandos:

```
$ git config --global user.name "<tu nombre>"  
$ git config --global user.email "<tu email>"
```

Por ejemplo:

```
$ git config --global user.name "Miguel Ángel Durán"  
$ git config --global user.email "miduga@gmail.com"
```

Sin embargo debes saber algo. **Este tipo de firma no es segura.** Como te habrás dado cuenta, cualquier persona podría usar los datos de otra persona y hacerse pasar por ella...

Para mejorar la seguridad de tus commits, y que no quede duda de la autoría de ellos, puedes firmarlos con una llave GPG y, de esta forma, mostrar que ha sido verificado.

Generando llaves GPG

GPG es un software de cifrado híbrido que combina criptografía convencional de claves simétricas para la rapidez con criptografía de claves públicas para el fácil compartimiento de claves seguras.

Primero, vamos a ver si tenemos instalada la implementación de GPG en nuestro sistema. Normalmente se usa *GnuPG* por lo que podemos ver si lo tenemos instalado ejecutando el siguiente comando en la terminal.

```
$ gpg --version
gpg (GnuPG) 2.3.3
libgcrypt 1.9.4
Copyright (C) 2021 Free Software Foundation, Inc.
License GNU GPL-3.0-or-later <https://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.

Home: /Users/midudev/.gnupg
Algoritmos disponibles:
Clave pública: RSA, ELG, DSA, ECDH, ECDSA, EDDSA
Cifrado: IDEA, 3DES, CAST5, BLOWFISH, AES, AES192, AES256, TWOFISH,
          CAMELLIA128, CAMELLIA192, CAMELLIA256
AEAD: EAX, OCB
Resumen: SHA1, RIPEMD160, SHA256, SHA384, SHA512, SHA224
Compresión: Sin comprimir, ZIP, ZLIB, BZIP2
```

Si te aparece que no encuentra el comando, deberás instalarlo.

Si estás en Windows, te recomiendo que uses la máquina virtual de Ubuntu para conseguirlo y sigas los pasos para Debian, que sería ejecutar el comando `sudo apt-get install gnupg`.

En Linux, simplemente, usa el gestor de paquetes adecuado a tu sistema.

Si estás en macOS lo mejor es usar Homebrew, como hemos hecho anteriormente con otros paquetes, y ejecutar el comando `brew install gnupg`.



GnuPG permite cifrar y firmar tus datos y comunicaciones, incluye un sistema versátil de gestión de claves, así como módulos de acceso para toda clase de directorios de claves públicas.

Ahora ya podemos generar la llave GPG. Para ello, vamos a ejecutar el comando `gpg --full-generate-key`

Los pasos que debemos seguir serán:

1. Elegir `RSA (sign only)` como el tipo de clave.
2. Elegir `4096` como el tamaño de la clave.
3. Sin expiración (o puedes elegir la fecha que prefieras, pero tendrás que renovarla).
4. Tu nombre real o nombre de usuario de GitHub.
5. El correo electrónico que usas actualmente para firmar el commit.
(puedes ejecutar `git config --global user.email` para encontrarlo)
6. Añade un comentario como `Git Sign Commits` para saber de qué va la llave.

Si quieres, **puedes añadir una contraseña para tu llave.** Puede ser útil por si alguien obtiene acceso a tu ordenador y no le sirva simplemente copiándola. Aunque si alguien tiene acceso a tu ordenador, seguramente este sea el menor de tus problemas.

Una vez generada, podemos recuperar la información de la llave ejecutando:

```
> gpg --list-secret-keys --keyid-format SHORT
```

```
/Users/midudev/.gnupg/pubring.kbx
```

```
sec    rsa4096/A2XXXXXXX 2022-01-02 [SC] [caduca: 2024-01-02]
      7F1B507XXXXXXXXXXXXXXXXXXXXXX
uid          [  absoluta ] Miguel Ángel Durán <miduga@gmail.com>
```

Después del `rsa4096`, tenemos `A2XXXXXXX` que sería la ID de nuestra llave y, en este caso, es lo que nos interesa. En este caso, ten en cuenta, que tu ID será diferente y, por lo tanto, debes seguir los pasos usando tu llave y no la mía.

Ahora podemos indicarle a Git que queremos usar esa llave para firmar nuestros commits. Para ello, ejecutamos el comando `git config --global user.signingkey A2XXXXXXX`.

También necesitamos asegurarnos que el `gpg-agent` está ejecutándose en nuestro sistema y además tenemos que informarle del shell actual que estamos usando para ello ejecutamos estos dos comandos:

```
# Reiniciamos el agente de gpg
gpgconf --kill gpg-agent
gpgconf --launch gpg-agent

# variable de entorno que le informa del shell actual
export GPG_TTY=$(tty)
```

El segundo comando, la idea, es lo que lo añadas en tu fichero `~/.zshrc` o `~/.bashrc` para que se inicie siempre que levantes una nueva sesión de tu línea de comandos.

Firmando commits con llave GPG

Ahora ya podemos firmar nuestros commits. Para ello, cuando hagamos un commit, debemos añadir el parámetro `-s` y utilizará la llave que hemos configurado previamente.

```
$ git commit -m "Add signed commit" -s
[main aafac37] Add signed commit
 1 file changed, 127 insertions(+)
```

Si añadiste una contraseña a tu llave GPG, es normal que te pida una contraseña en este punto. De esta forma se asegurará que eres tú quien firma el commit...

Obviamente no vas a querer añadir en cada *commit* el parámetro `-s`. Si quieres firmar todos tus commits a partir de ahora, lo mejor es que lo indiques en la configuración de Git.

```
# firma automáticamente todos los commits
$ git config --global commit.gpgsign true
```

Es posible que en cada commit te pida la contraseña. En algunos sistemas puede guardarse en el llavero del sistema pero... si no lo hace, puede ser un poco molesto. Según el sistema hay, igualmente, formas de saltarse pero escapa de lo que se pretende cubrir en el libro.

Configurando GitHub

Ahora ya sólo nos queda enlazar nuestra llave local con los repositorios remotos. Vamos a ver cómo hacerlo en GitHub, que es el proveedor más usado, pero los pasos deben ser muy similares con otros como BitBucket o GitLab.

Primero, tenemos que recuperar el valor de la llave GPG que hemos configurado previamente.

```
# Recuperamos la llave GPG para usarla en GitHub
# Recuerda que A2XXXXX es mi llave pero la tuya será diferente
$ gpg --armor --export A2XXXXXX
-----BEGIN PGP PUBLIC KEY BLOCK-----

mQINBGHROEYBEACY16xEQ02EJwH1yODpEaxLeLR8EdMgY8RAbrinLROkQbqnkhEH
...
DtcqaBqqquiolNPUb9Z1kubLb89Q84=
=I123T+
-----END PGP PUBLIC KEY BLOCK-----

# O mejor...
# Para copiar directamente en el portapapeles en macOS
$ gpg --armor --export A2XXXXXX | pbcopy
```

Ahora dirígete a <https://github.com/settings/gpg/new> y pega en la caja de texto la llave GPG que acabas de copiar. Después, haz click en el botón `Add GPG key`.

Y, con esto, ya podrías empujar tus commits a GitHub y ver en el repositorio que el historial de commits muestra aquellos que han sido firmados con llave GPG con una nueva etiqueta llamada *Verified*.

The screenshot shows a GitHub repository named 'Commit signature / manual-git' with a private status. The repository has 7 watchers, 0 forks, and 4 stars. The main branch is 'main'. A tooltip indicates that a commit was signed with a verified GPG key. The commit details show a user named 'midudev' committing on Jan 2, 2022. The commit message is 'New commit'. The commit is signed by 'midudev' (Miguel Ángel Durán) with a GPG key ID: EC4D6ABCD2DBCF23. The commit is shown as 'Verified'.

En GitHub nos mostrará información sobre si un commit ha sido firmado o no con llaves GPG y, de esta forma, asegurarnos que la autoría es real y no simulada

¿Cómo debería revisar una Pull Request?

Cuando recibimos una nueva Pull Request, debemos revisarla y aceptarla, rechazarla o... ignorarla. Bueno, esto último no lo hagas. De hecho ese es el primer consejo.

Voy a compartirte una serie de consejos y buenas prácticas a seguir a la hora de revisar una petición de cambios en un repositorio (*Pull Request*). Estos consejos pueden servirte tanto para trabajar en una empresa como en un proyecto de código abierto.

Valora el tiempo de la persona y empatiza

Alguien ha dedicado tiempo de su vida en crear una petición de código en tu proyecto o en el proyecto de la empresa. ¿No te gusta la Pull Request? ¿No estás de acuerdo? ¿No la piensas aceptar? No pasa nada, ya veremos cómo podemos lidiar con ello. Pero aún así **debemos valorar el tiempo que ha dedicado, seguramente, con su mejor intención**.

La empatía es, seguramente, una de las mayores virtudes con las que se puede contar en el mundo de la programación. Ponernos en el lugar de la persona y de cómo nos gustaría a nosotros que nos tratasen.

Intenta dedicarle tiempo de calidad a la revisión de la petición de cambios. Considera que, pese a no estar al 100% de acuerdo con los cambios, igual puedes fusionarla si satisface casi todas las condiciones y el resto podrías adaptarlas tú mismo más tarde.

Recuerda que el código no es de piedra, por lo que siempre puedes hacer cambios y mejoras sobre cualquier PR que recibas.

Proporciona siempre feedback positivo

Cuando reaccionamos a una Pull Request hay que tener en cuenta que hay una persona detrás. Está bien que expresemos nuestras opiniones y que pidamos

cambios, pero **debemos saber hacerlo para que la persona pueda recibir el mensaje y, además, mejorar con ello.**

Si cuando damos feedback, este feedback no ayuda a la otra persona a mejorar el código o su nivel, **entonces no es feedback, es otra cosa.** Y esas otras cosas pueden disparar las emociones de las personas de forma innecesaria.

Compara las dos siguientes formas de dar el mismo *feedback*:

Esto está mal. ¡Usa el spread operator en lugar de poner tanto código!

¡Muchas gracias por colaborar en el proyecto! Esto soluciona al problema. ¡Es genial! Te sugiero que uses el spread operator en la línea 14. Creo que esto te ahorrará tres líneas y, además, hará que el código sea más legible. ¿Qué te parece?

Como puedes ver **ambos mensajes valoran lo mismo pero lo hacen de formas muy distintas** (y seguramente con resultados muy diferentes).

En el segundo mensaje sólo estamos valorando cosas positivas mientras le damos un feedback constructivo y, además, le decimos por qué se lo damos.

Recibir feedback es un regalo. También debemos saber recibir comentarios para mejorar como profesionales. Al final, debemos entender que si una persona nos pide cambios también lo hace con la mejor intención.

Concreción y claridad

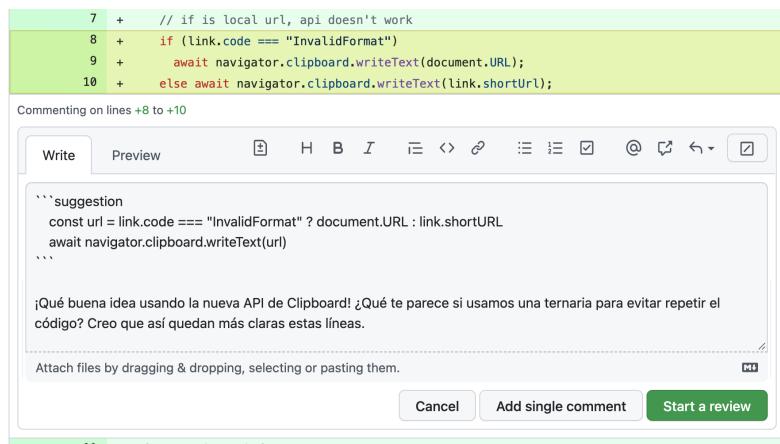
Cuando comentamos una Pull Request debemos aportar **claridad, brevedad y concreción**. Dicho de otra forma, evitamos las generalidades y vamos al grano. De esta forma **evitamos los malos entendidos**, reducimos el alcance de la discusión y nos aseguramos de que la persona pueda entender nuestra opinión.

Compara las dos siguientes formas de dar el mismo *feedback*:

No me gustan los nombres de las variables. ¿Las puedes cambiar?

Creo que podríamos mejorar algunos nombres de variables. Por ejemplo, en lugar de la variable `sent` podríamos llamarle `isSent` para que quede claro que es un tipo de dato lógico. Lo mismo para `called`. Podríamos usar `hasCalled`. ¿Cómo lo ves?

También puedes utilizar las sugerencias de código, de forma que propongas una solución directa a la petición que tienes. De esta forma ofrecemos una solución alternativa a lo que ha hecho y no le bloqueamos mientras espera a hacer sus cambios para tener de nuevo nuestra opinión. Reducimos el *feedback loop*.



En GitHub puedes proporcionar sugerencias de código en tu comentario. La persona que haya creado la PR podrá aceptar los cambios sugeridos directamente.

Entiende el contexto

No es lo mismo revisar una PR que contiene una nueva característica que puede romper la compatibilidad de una API a una petición de cambios que, pese a no tener un código perfecto, está arreglando un bug que bloquea producción o a cientos de usuarios.

Entiende que el contexto es importante a la hora de valorar estas peticiones. Es posible que a veces tengamos que poner paños calientes o

parches a nuestro código y que, pese a no ser el más bonito, sí que cumpla su cometido.

Por ejemplo, si ves que hay algo que no te convence en el código, puedes añadir un comentario en la línea de código que ves mejora pero dejarlo para más adelante. Compara estas dos formas de dar feedback:

Bufffffff, este cambio es inmantenible.

Entiendo que este cambio es importante para desbloquear los pasos a producción. Fusionemos la rama y después trabajamos juntos para mejorar la solución, de forma que sea más mantenible. ¿Qué te parece?

El código está en constante evolución y mejora. **Hay que tirar de pragmatismo en ocasiones.**

Añade un archivo de CONTRIBUTING.md

Crea un archivo `CONTRIBUTING.md` en la raíz de tu repositorio para que puedas documentar los pasos que una persona debe realizar para poder aportar cambios a tu proyecto.

También puedes explicar el proceso de revisión de las *Pull Requests*, un código de conducta, los estándares de código y cualquier otra cosa que pueda ayudar a la gente a entender cómo contribuir al proyecto.

Hooks de Git

Conforme nuestro repositorio va creciendo vamos a necesitar tener más control sobre qué código se sube y cómo. Por supuesto, para facilitar la vida del equipo de desarrollo, vamos a necesitar automatizarlo.

Ahí es donde entran los hooks de Git, una forma de automatizar ciertas tareas que deben ejecutarse en ciertos momentos del proceso de subida de código.



¿Qué es un hook?

Un hook, o punto de enganche, es la posibilidad de ejecutar una acción o script cada vez que ocurre un evento determinado de Git.

Estos scripts se colocan dentro de la carpeta `.git/hooks` con el nombre del evento que escuchará para ser ejecutado. En esa carpeta, por defecto, ya existen unos pocos scripts de ejemplo:

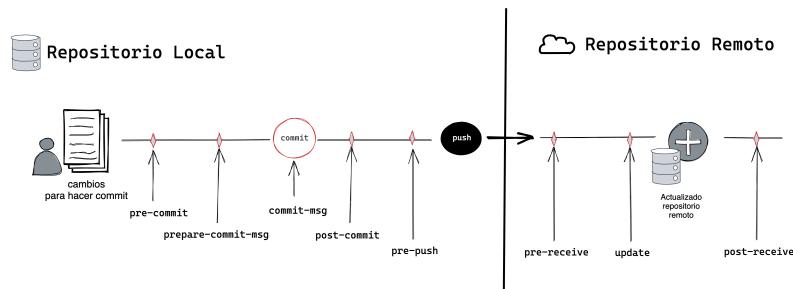
<code>applypatch-msg.sample</code>	<code>pre-push.sample</code>
<code>commit-msg.sample</code>	<code>pre-rebase.sample</code>
<code>post-update.sample</code>	<code>prepare-commit-msg.sample</code>
<code>pre-applypatch.sample</code>	<code>update.sample</code>
<code>pre-commit.sample</code>	

Este listado contiene los hooks más usados pero no son los únicos.

Entra a cualquier repositorio que tengas inicializado con git y luego lista los hooks que hay en la carpeta `.git/hooks`. ¿Qué hay dentro? Revisa el contenido de los ficheros.

¿Qué hooks hay disponibles?

Existen puntos de enganche para eventos en el lado del cliente y en el lado del servidor. En este manual vamos a revisar los más útiles de ambos lados aunque, en realidad, los que normalmente vas a usar son los disponibles en el cliente.



En este diagrama puedes ver todos los hooks del lado del cliente y del servidor

Hooks del lado del cliente (repositorio local)

Los hooks del lado del cliente sólo afectan al repositorio local que los contiene. Esto significa que puedes tener el mismo repositorio clonado de forma local varias veces y, sin embargo, ejecutar diferentes hooks.

Esto es muy importante que lo tengas en cuenta: **los hooks son sólo parte del repositorio que los contiene y no se guardan ni se sincronizan con otras réplicas** del repositorio automáticamente. Más adelante veremos cómo podemos hacer que los hooks se instalen cada vez que un usuario quiera trabajar en un repositorio.

Los hooks más importantes son:

- pre-commit
- prepare-commit-msg
- commit-msg
- post-commit

- `pre-push`

Otros hooks interesantes a destacar:

- `post-checkout`
- `post-merge`

`pre-commit`

Se ejecuta inmediatamente después del comando `git commit` pero antes de que Git te pida un mensaje para el commit y que genere el objeto de commit. Desde el script puedes salir con un código diferente a 0 para abortar el commit.

¿Para qué lo puedes usar?: Puede ser un buen sitio para ejecutar el linter sobre los archivos que han sido modificados. También podrías comprobar si se está intentando hacer un commit de demasiados archivos o líneas de código cambiadas.

`prepare-commit-msg`

Se ejecuta cuando Git va a crear el mensaje que usará en el commit. Igual que `pre-commit`, también puedes abortar el commit si es necesario.

¿Para qué lo puedes usar?: Puedes usarlo para modificar el mensaje del commit o añadir cualquier información extra. Ten en cuenta que, después de preparado, el usuario todavía podría modificar el mensaje por lo que no es un buen sitio para comprobar que el mensaje de commit sigue un patrón estándar.

`commit-msg`

Se ejecuta una vez que se ha preparado el mensaje de commit y el usuario ha hecho todas las modificaciones sobre el mismo. Desde este script todavía puedes abortar que el commit se realice si es necesario.

¿Para qué lo puedes usar?: Es el sitio perfecto para hacer todas las comprobaciones pertinentes al mensaje. Por ejemplo, si usas una convención de mensajes de commit como `conventional-commit`, aquí puedes usar una herramienta como `commitlint` para asegurarte que el mensaje sigue la convención y abortar el commit si no lo hace.

¿Sabías que al hacer un commit puedes saltar la ejecución de los commits `pre-commit` y `commit-msg`? Para ello, puedes usar el comando `git commit --no-verify`. Aunque no es recomendable, puedes usarlo si quieres que no se ejecuten los hooks.

post-commit

Una vez que el commit ha sido grabado, se ejecuta este hook que te permite ser notificado que la operación se ha realizado con éxito. Esto quiere decir que desde este script ya no puedes evitar que el commit se realice y no importa el código de salida del script.

¿Para qué lo puedes usar?: Su uso principal es la de notificar. Podrías usarlo para enviar un mensaje a Slack a tu equipo cada vez que haces un commit en tu repositorio local, para que vean que estás trabajando en algo. Un poco raro... pero puede ser útil.

pre-push

Este hook se ejecuta tras llamar al comando `git push` y ocurre antes de que se haya transferido ningún objeto al repositorio remoto. Desde este script puedes evitar que el push se realice si devuelves un código de error.

¿Para qué lo puedes usar?: Para ejecutar una batería de tests de forma que evites que llegue a un repositorio remoto cambios que no pasan las pruebas.

post-checkout y post-merge

Estos dos hooks son ejecutados tras realizar un `git checkout` y `git merge` respectivamente.

¿Para qué lo puedes usar?: A diferencia del `post-commit` que sólo sirve para posibles notificaciones, estos dos hooks pueden tener otras utilidades. La más interesante es la de limpiar el directorio de trabajo, tras realizar un `checkout`, o el de limpiar las ramas que ya no se usan tras realizar un `merge`.

Hooks del lado del servidor (repositorio remoto)

En el lado del servidor, en el *repositorio remoto*, tienes tres puntos de enganche:

- `pre-receive`
- `update`
- `post-receive`

Es muy probable que nunca necesites realizar ningún tipo de script que se ejecute en uno de estos puntos de enganche. Sin embargo, es interesante conocerlos ya que páginas como GitHub o GitLab los usan intensivamente a la hora de construir.

`pre-receive`

Este hook se ejecuta una vez antes de que las referencias sean actualizadas en el repositorio remoto. Al estar en el lado del servidor, es capaz de rechazar cualquier cambio si es necesario.

¿Para qué lo puedes usar?: Si fuese posible, puedes tener en el lado del servidor un script para comprobar que los commits que se quieren guardar están bien formados. O que el usuario que intenta grabar los commits tiene los permisos necesarios para hacerlo. También puedes evitar commits que tengan conflictos o intentos de `rebase`.

`update`

Muy similar a `pre-receive`, pero este hook se ejecuta por cada referencia que quiere ser actualizada. Si, por ejemplo, en un sólo `push` intentas enviar 3 ramas distintas... entonces este hook será ejecutado 3 veces.

¿Para qué lo puedes usar?: Similar a los usos de `pre-receive`, con la diferencia que puedes evitar de una forma granular cada actualización (de forma que algunas referencias sí puedan ser grabadas y otras no).

post-receive

Tras grabar los cambios en el repositorio remoto, este hook se ejecuta con la información de cada cambio. Ya no puedes evitar que los cambios sean grabados y sólo sirve para notificar.

¿Para qué lo puedes usar?: En este caso para notificaciones. Por ejemplo, enviar un correo a todos los usuarios del repositorio que se han grabado nuevos cambios en el repositorio remoto. O reflejar en una UI las nuevas referencias, ramas o commits disponibles.

¿Cómo puedo crear mi propio hook?

Para crear un propio hook sólo tienes que crear un archivo `nombre-del-hook` en la carpeta `.git/hooks` y en él poner el código que quieras que se ejecute. Puedes usar todo tipo de intérpretes de lenguaje de programación como `bash`, `node`, `python`, `perl`, etc.

Verificar archivos con un *linter* antes de hacer commit

Vamos a crear un hook que verifica los archivos con un *linter* antes de hacer un commit. Para ello, vamos a crear un script que se ejecute antes de hacer un commit.

Primero creamos el fichero `pre-commit` en la carpeta `.git/hooks`. En un sistema Unix, puedes ejecutar el siguiente comando.

```
$ touch .git/hooks/pre-commit
```

Además, necesitamos hacer que el script sea ejecutable. Lo podemos conseguir con este comando:

```
$ chmod +x .git/hooks/pre-commit
```

Ahora, con tu editor favorito, lo primero que vamos a hacer es indicar que el script que queremos ejecutar debe usar el shell disponible en tu sistema.

```
#!/bin/sh
```

Ahora vamos a escribir nuestro pequeño script. Es posible que no conozcas este lenguaje, ya que es un script de Shell, pero es muy sencillo y te voy a dejar unos pocos comentarios para que no te pierdas.

```
# creamos una variable para guardar el número de salida del proceso
linter_exit_code=1
# ejecutamos un comando para recuperar sólo los archivos .js y .jsx
# que han sido modificados en los commits para revisarlos
staged_js_files=$(git diff --cached --diff-filter=d --name-only | grep \
-E '\.(js|jsx)$')
```

```

# ejecutamos nuestro linter contra los archivos .js y .jsx modificados
# le pasamos el parámetro --fix para intentar arreglarlos si es posible
# y ponemos --quiet, para evitar el output en la consola
./node_modules/.bin/standard $staged_js_files --quiet --fix

# con esta línea, recuperamos el código de salida del
# último proceso ejecutado
linter_exit_code=$?

# volvemos a añadir los archivos modificados a la lista de commits
# por si el --fix ha hecho efecto, así nos evitamos tener que
# volver a hacer commit
git add -f $staged_js_files

# si el código de salida es distinto de cero, es que hubo algún error
if [ $linter_exit_code -ne 0 ]
then
    echo "[error] Linter errors have occurred"
    exit 1
else
    echo "[ok] Linter did not find any errors"
    exit 0
fi

```

Ahora que ya lo tenemos, podemos usar colores en la salida de la consola usando unas secuencias de ANSI definida. Vamos a crear estas variables para usarlas en los mensajes:

```

# variable para hacer el texto de color rojo
RED="\033[1;31m"
# variable para hacer el texto de color verde
GREEN="\033[1;32m"
# variable para quitar el color del texto
NC="\033[0m"

```

Estas variables, ahora se pueden interpolar en el string para hacer que los mensajes sean de un solo color. Es importante al finalizar el mensaje limpiar el color de la cadena de texto.

```

if [ $linter_exit_code -ne 0 ]
then
    echo "${RED} [error] Linter errors have occurred ${NC}"
    exit 1
else
    echo "${GREEN} [ok] Linter did not find any errors ${NC}"
    exit 0
fi

```

Y al final quedaría de esta forma:

```
RED="\033[1;31m"
GREEN="\033[1;32m"
NC="\033[0m"

linter_exit_code=1
staged_js_files=$(git diff --cached --diff-filter=d --name-only | grep \
-E '\.(js|jsx)$')

./node_modules/.bin/standard $staged_js_files --quiet --fix
linter_exit_code=$?
git add -f $staged_js_files

if [ $linter_exit_code -ne 0 ]
then
    echo "${RED} [error] Linter errors have occurred ${NC}"
    exit 1
else
    echo "${GREEN} [ok] Linter did not find any errors ${NC}"
    exit 0
fi
```

Limpiar ramas locales tras merge

Script compatible con git version 2.28.0 o superior.

Vamos a crear un hook que nos permita limpiar las ramas locales y remotas tras hacer un merge. Para ello, vamos a crear un script que se ejecute después de hacer un merge. He añadido pequeños comentarios para que no te pierdas.

```
#!/bin/bash
exec < /dev/tty

# Recuperamos el nombre de la rama actual
branch_name=$(git branch | grep "*" | sed "s/* //")

# Recuperamos el nombre de la rama recién fusionada
reflog_message=$(git reflog -1)
merged_branch_name=$(echo $reflog_message | cut -d" " -f 4 | sed "s/://\
")

main_branch=$(git config --get init.defaultBranch)

# Si la rama recién fusionada es la principal, salimos
if [[ $merged_branch_name = $main_branch ]]; then
    exit 0
fi

# Escribimos en la consola información
echo " "
echo "Fusionada \"$merged_branch_name\" en \"$branch_name\". "
```

```
# Preguntamos al usuario la acción
read -p "¿Quieres borrar la rama \"\$merged_branch_name\"? (y/N) " answer

# Revisamos que la respuesta sea "y"
if [[ "$answer" == "y" ]]; then
    # Borramos la rama local
    echo "Borrando la rama local: \"\$merged_branch_name\""
    git branch -d $merged_branch_name

    # Borramos la rama remota
    echo "Borrando la rama remota"
    git push origin --delete $merged_branch_name
    exit 1
else
    echo "No he podido borrar la rama \"\$merged_branch_name\""
fi
```

Ahora, para lograr que se ejecute este pequeño script tras cada merge, vamos a crear un hook en la carpeta `.git/hooks` y le vamos a dar permisos de ejecución.

```
chmod +x .git/hooks/post-merge
```

Anímate a modificar el script para dejarlo a tu gusto y que se adapte a tus necesidades.

Alias en Git

Hay muchos comandos en Git y, junto con los parámetros, demasiadas combinaciones. Los alias permiten definir una serie de comandos que pueden ser usados en lugar de los nombres completos.

Así que ya sea porque quieres ahorrarte unas pocas teclas, mejorar tu experiencia de desarrollo o, simplemente, porque no quieres recordar todos los parámetros que necesitas para realizar cierta acción, los alias son una buena opción.

Vamos a revisar qué son, cómo crearlos, cómo usarlos y veremos algunos ejemplos útiles.

git co → git commit

git st → git status

Los alias pueden ser sencillos atajos para comandos muy usados a complejas combinaciones de comandos con parámetros...

¿Cómo crear tu propio alias en Git?

Git te permite crear tus propios alias fácilmente para comandos que usas habitualmente en tu proyecto con este sistema de control de versiones.

Para crear tu propio comando debes usar el comando `git config` y ponerlo de la siguiente manera:

```
$ git config --global alias.[nombre-del-alias] "comando a ejecutar"
```

También puedes añadir alias directamente editando el archivo de configuración de git. Para ello, ejecuta `git config --list --show-origin` y edita el fichero según el ámbito donde quieras que se pueda usar tu alias (normalmente `global`).

Pongamos un ejemplo real de *alias*. Imaginemos que queremos simplificar la sintaxis de `git log`. Vamos a aprovechar que `git log --oneline` muestra los commits en una sola línea. Pero lo usamos tanto que no queremos recordarlo. Pues vamos a crear un alias.

```
$ git config --global alias.ll "log --oneline"
```

Ahora, podemos utilizar este nuevo alias de la siguiente forma:

```
$ git ll  
31239c5 (HEAD -> main) Fix commit  
578345d Probando nuevo log
```

Vamos a ver otro ejemplo, en este caso para recuperar el último commit de un repositorio. Para ello vamos a crear un alias llamado `last`:

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

Y podemos usarlo de la siguiente manera:

```
$ git last  
commit ee9caa237c4ebd95f88a9106f4fd891569f22692 (HEAD -> master, origin\
```

```
/master, origin/HEAD)
Author: midudev <miduga@gmail.com>
Date:   Mon Jul 12 13:17:25 2021 +0200

    Fix styles

assets/styles/global.css | 7 +++++-
1 file changed, 3 insertions(+), 4 deletions(-)
```

El uso de alias puede facilitarte mucho la vida a la hora de trabajar con git.
¡Yo tengo al menos una docena de ellos!

Ten en cuenta que no puedes sobrescribir comandos de git. Por ejemplo, no puedes crear un alias con el nombre de `commit` ya que éste ya existe. En el caso de que un alias tenga el nombre de un comando de git ya definido, se ignorará el alias (no recibirás un error al crearlo).

¿Puedo crear alias para comandos que ya existen?

Sí. Además de crear alias que sean compuestos con parámetros, también puedes simplemente renombrar los comandos que ya tiene git, de forma que sean más fáciles de usar.

```
$ git config --global alias.br branch  
$ git config --global alias.ch checkout  
$ git config --global alias.co commit  
$ git config --global alias.st status
```

Ahora, podrías usar la forma corta de los comandos:

```
$ git br # git branch  
$ git ch # git checkout  
$ git co # git commit  
$ git st # git status
```

Además, **ten en cuenta que al usar un alias puedes usar parámetros**. Por ejemplo, si quieres añadir un mensaje a un *commit* usando el alias `git co` lo puedes hacer de la siguiente manera:

```
$ git co -m "Add new feature"
```

¿Sabías que si escribes mal el comando `comit` (con una m), Git te recordará cuál es el comando más similar? Esto es genial, pero incluso puedes crearte un alias con estos errores típicos para que no te preocupes en el futuro: `git config --global alias.comit commit`.

¿Puedo crear un alias para un comando externo?

Sí. Aunque lo normal es crear un alias que sea un comando de git, en ocasiones puede ser interesante crear un alias para un comando que no sea exactamente de la línea de comandos de git pero que esté lo suficientemente relacionado como para que sea útil.

Por ejemplo, es posible que tengas un cliente visual de git y lo quieras abrir con `git gui`. Puedes hacerlo de la siguiente manera:

```
$ git config --global alias.gui '!sourcetree'
```

Fíjate que el comando `sourcetree` viene precedido de una exclamación `!`. Esto significa que el comando será ejecutado como un comando de terminal (*shell*). Además, ten en cuenta que el comando `sourcetree` se ejecutará en el directorio raíz del repositorio, sin importar en qué carpeta del directorio de trabajo te encuentres.

¿Cómo puedo listar todos los alias que he creado?

Ya conoces `git config`. Con `git config --list` puedes recuperar todas las configuraciones que has creado y ver, entre ellas, los alias que has creado.

Sin embargo, si tu configuración global es muy grande, puedes tener problemas para leerla y distinguir los alias entre tantas otras cosas. Así que puedes usar el parámetro `--get-regexp` para filtrar la configuración y enfocarte en lo que te interesa.

```
# lista toda la configuración que contenga la palabra alias
$ git config --get-regexp alias
# más correcto, recupera las configuraciones que comienzan por alias
$ git config --get-regexp ^alias\.
```

¿Sabes lo mejor? Puedes crear un alias para mostrar la lista de alias que has creado. ¡Inception! Lo podríamos crear y usar de la siguiente manera:

```
$ git config --global alias.alias "config --get-regexp ^alias\."
# ahora podrás ejecutar el nuevo comando
$ git alias

alias.log log --oneline
alias.alias config --get-regexp ^alias\.
alias.last log -1 HEAD --stat
alias.bc switch -c
```

¿Cuáles son los mejores alias?

No creo que haya mejores o peores alias. Todo depende de la necesidad de cada persona y el uso que le dé a Git. Aunque podemos estar de acuerdo en una cosa: los mejores alias son los que van a hacer tu vida más fácil y va a simplificar tu uso de Git.

Para que te hagas una idea, en mi caso, no utilizo mucho los alias de `git`, ya que uso Oh My Zsh y el [plugin de git](#). Trae una decenas de alias para los comandos más comunes y para otros flujos de trabajo. Por ejemplo, uso `gst` para `git status` y `gco` para `git checkout`, entre otros.

Igualmente, para que no te vayas con las manos vacías te comparto algunos que he usado en algún momento o que sé que han sido útiles en algún momento para mis colegas de trabajo.

`git graph`

Te muestra de forma gráfica los cambios de un repositorio.

```
# creamos el alias para tener una vista gráfica de los cambios
$ git config --global alias.graph "log --all --graph --decorate --oneline"

# usamos el nuevo alias
$ git graph

* f4358d6 (HEAD -> master) Add transparent border
* 06dc9de (origin/master, origin/HEAD) Fix styles
* a2c4852 Fix diviendo typo
* 75d4dc3 Improve contrast ratio
* 6971958 Use chromium-aws-lambda to make it work at Vercel
* 1f237b4 Merge branch 'master' of github.com:midudev/midu.dev
| \
| * b60b708 Merge pull request #64 from midudev/imgbot
| |
| | * 9fabcf2 (origin/imgbot) [ImgBot] Optimize images
* | | 0068778 Add more logging to lambda
| / /
* | 848d316 Add some tweaks for reset icon
* | 1e51c7c Fix wrong behaviour on mobile
* | ale963f Remove svg and improve article pagination
* | 7f3f273 Add preload for tracking script
| /
```

git undo

Te permite fácilmente deshacer los commits que hayas realizado, y que todavía no hayas guardado en el repositorio.

```
# creamos el alias para deshacer un commit
$ git config --global alias.undo "reset --soft HEAD^"

# deshacemos el último commit
$ git undo

Unstaged changes after reset:
M app.js
```

git rank

Para mostrar una lista de los contribuidores con más *merges*.

```
# creamos el alias "git rank"
$ git config --global alias.rank "shortlog -sn --no-merges"

# usamos el alias para mostrar el ranking
$ git rank

44 midudev
11 d4nidev
9 codingwithdani
4 jonathandelgado
```

git fresh

¿Quieres empezar de nuevo? Si tienes claro que quieres descartar todos los cambios de archivos que tienes en tu directorio de trabajo y que no has grabado todavía... pero nunca te acuerdas de cómo usar `git reset` y `git clean` entonces este alias es para ti.

```
$ git config --global alias.fresh "\!git reset --hard HEAD && git clean\-
-f -d"
```

Ten en cuenta que esto deshará todos los cambios que no has guardado (*commit*) y, también, los ficheros que hayas creado y que no eran parte del repositorio antes.

git bc

Si creas constantemente nuevas ramas en tu proyecto y quieres cambiar a ellas de forma rápida, puedes crearte un alias llamado `git bc` de *branch change*.

```
# alias para crear una rama y cambiar a ella
$ git config --global alias.bc "switch -c"
$ git bc cool-branch-name
Switched to a new branch 'cool-branch-name'

# para versiones de git anteriores sin soporte a git switch
$ git config --global alias.bc "checkout -b"
```

git br

Para mostrar una lista de ramas que tienes en tu repositorio en una lista con colores e información útil:

```
$ git config --global alias.branches "branch \
--format='%(color:yellow)%(refname:short)%(color:reset) - \
%(contents:subject) %(color:green) (%(committerdate:relative)) \
[%(authorname)]' --sort=-committerdate"
```

Lo mejor de este alias es, además, revisar lo que estamos haciendo paso a paso.

Primero, ejecutamos el comando `git branch`.

Con el parámetro `--format`, estamos dando formato a la salida. Podemos usar algunas evaluaciones de variables especiales. Por ejemplo, `%(refname:short)` muestra el nombre de la rama.

`%(color:yellow)` nos permite cambiar el color de la salida. Para evitar que todo quede amarillo, evaluamos `%(color:reset)` para volver al color original.

Los demás campos creo que se explican por si solos sabiendo todo esto, pero puedes jugar con eliminarlos para determinar qué es cada uno.

Finalmente usamos el parámetro `--sort` para **ordenar las ramas por fecha de su último commit recibido.**

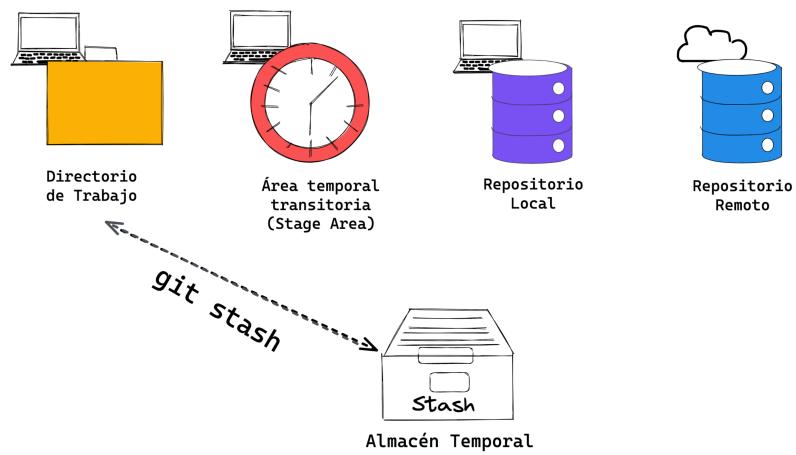
Stash, el almacén temporal de cambios

A la hora de trabajar con Git no es raro que tengas que cambiar activamente entre ramas. Imagina que estás en una rama, desarrollando una funcionalidad, y de repente te das cuenta que hay un error en la rama principal. ¿Qué puedes hacer?

Por un lado, **podrías crear un `commit` en tu rama para grabar los cambios** y, así, volver más tarde. Normalmente es la solución ideal, especialmente si quieres que ese cambio sea accesible por otras personas.

Sin embargo, si tu trabajo está a medias, es posible que no quieras hacer `commit` pero tampoco quieras perder los cambios. **Otra opción sería clonar el repositorio** en otra ubicación y trabajar en esa copia, para tener dos copias y así que no se pierda ningún cambio.

Pero... ¿Qué pasa si no quieres grabar los cambios y tampoco quieres clonar el repositorio otra vez... pero no quieres perder los cambios? Para ello existe el *stash*, un almacén temporal de cambios. El almacén tiene forma de pila, de forma que se van apilando los cambios que tienes en tu directorio de trabajo en un momento dado para que puedas recuperarlos más tarde.



Git Stash es un almacén temporal para cambios que no queremos grabar pero tampoco queremos perder

Guarda tus cambios en un stash

En primer lugar tienes que tener en cuenta que los cambios que puedes almacenar en la pila provisional son los cambios que has hecho en tu directorio de trabajo o que se encuentran ya en el área de *staging*. Dicho de otra forma, **cualquier cambio que no hayas todavía confirmado**.

Para asegurarte de qué cambios serán los que almacenes en el *stash*, puedes usar primero el comando `git status` y ver qué cambios están pendientes de grabar con un *commit*.

Imaginemos que tenemos un repositorio, con algún commit previo, donde hemos modificado dos ficheros. Uno lo hemos añadido ya al área de *staging* y otro lo hemos dejado modificado en el área de trabajo:

```
$ git status
On branch main
Changes to be committed:
  modified:   component.js

Changes not staged for commit:
  modified:   index.js
```

Ahora quiero cambiar de rama y trabajar en otra cosa pero no quiero perder estos cambios. Vamos a almacenarlos en el almacén temporal de cambios (*stash*)

```
$ git stash
Saved working directory and index state WIP on main: ff56eac commit
```

`git stash` es la forma corta de `git stash push`. La diferencia es que `git stash` no acepta todos los argumentos que puedes usar con `push`.

¡Ya has creado tu primer almacén temporal de cambios! Si vuelvas a ejecutar `git status` verás que ahora no tienes cambios en tu área de trabajo:

```
$ git status
On branch main
nothing to commit, working tree clean
```

¿Han desaparecido? ¡No! Están en una pila provisional, pero no en el área de trabajo. **Esta pila es local** y puedes revisarlo en la carpeta `.git/refs/stash`. Esto significa dos cosas:

- No se transfieren al repositorio remoto.
- Si borras tu repositorio local, perderás la pila de cambios temporales.

Para asegurarte que está a buen recaudo, puedes ejecutar el comando `git stash list` para ver los almacenes temporales disponibles que has creado:

```
$ git stash list  
stash@{0}: WIP on main: ff56eac commit
```

No confundas el área de almacenaje temporal (*stash*) con el área temporal transitoria (*staging*). El primero es, simplemente, un cajón dónde apilar cambios que puede ser que queramos recuperar más adelante y el segundo es una fase anterior a grabar los cambios en el repositorio.

¡Cuidado con los archivos nuevos creados!

Por defecto, `git stash` sólo almacena los ficheros modificados. Si estás creando un fichero por primera vez, que nunca ha sido grabado con un *commit* al repositorio, no lo almacenará por defecto y podrías perderlo si no se lo indicas...

```
# creamos un nuevo fichero new-file en el repositorio  
$ touch new-file.html  
  
# vemos que el nuevo fichero aparece sin seguimiento  
$ git status  
On branch main  
Untracked files:  
  new-file.html  
  
# al crear la entrada en el stash, nos dice que no  
# hay nada que grabar  
$ git stash  
No local changes to save
```

Como ves, no se ha grabado nada en el almacén temporal porque el fichero era nuevo. **¡Ten mucho cuidado!** A veces, si mezclas entre ficheros modificados y ficheros creados, podría indicarte que el *stash* ha funcionado correctamente... pero lo cierto es que **sólo habrá almacenado los ficheros modificados y no los creados.** ¡Fíjate!

```
# creamos un nuevo fichero new-file en el repositorio
$ touch new-file.html

# modificamos en el editor el fichero index.js
$ nano index.js

# revisamos el estado de los ficheros
$ git status
On branch main
Changes not staged for commit
    modified: index.js
Untracked files:
    new-file.html

# movemos los cambios locales a un stash
$ git stash
Saved working directory and index state WIP on main:
afaf06b first branch commit

# al mirar el status vemos que el fichero new-file.html
# está todavía como sin seguimiento
$ git status
On branch main
Untracked files:
    new-file.html

# y luego veríamos que el stash no tiene
# el fichero new-file.html
# ;Si lo borramos, lo perderíamos!
```

Para solucionar esto debes usar la opción `--include-untracked` o, de forma más corta, `-u`. De esta forma le indicamos a `git stash` que también queremos guardar en la pila provisional los archivos de los que todavía no hacemos un seguimiento.

```
# guarda en el almacenaje temporal también los recién creados
$ git stash -u
Saved working directory and index state WIP on main:
5002d47 our new homepage
HEAD is now at 5002d47 our new homepage

$ git status
On branch main
nothing to commit, working tree clean
```

Puedes incluso incluir archivos ignorados por Git en el almacén temporal. Para ello, usa la opción `--all` o, de forma más corta, `-a`.

Ten en cuenta que sólo puedes hacer `git stash` en un repositorio que tenga, como mínimo, un *commit* ya grabado. De lo contrario, te dará un error.

Comentando stashes con descripción

Como has podido ver en el ejemplo anterior cuando hemos visto la lista de *stashes* disponibles, los nombres no son excesivamente útiles. Esto es porque por defecto, `git stash` guarda las modificaciones locales y le genera un nombre como "WIP on <rama>: <hash> <mensaje de commit>".

Si tuvieramos tres cambios en la pila temporal de una misma rama, lo veríamos así:

```
$ git stash list
stash@{0}: WIP on main: 5002d47 our new homepage
stash@{1}: WIP on main: 5002d47 our new homepage
stash@{2}: WIP on main: 5002d47 our new homepage
```

No es muy útil, ¿verdad? Es difícil saber qué cambios han sido grabados en cada stash. ¡Vamos a solucionarlo!

Si quieras añadir un mensaje más descriptivo para saber qué es lo que has hecho, puedes añadir la opción `-m` o `--message` de forma que puedas añadir una descripción personalizada.

```
$ git stash -m "Trying new images types"
Saved working directory and index state On main:
Trying new images types

$ git stash list
stash@{0}: On my-branch: Trying new images types
```

`git stash -m <descripción>` es la forma corta de usar el comando `git stash push --message <descripción>`.

Existe otro comando en `git stash` que te permite añadir un comentario. Es `git stash save`. Sin embargo, esta opción está obsoleta y no se recomienda usarla. En su lugar, usa `git stash push` para añadir un comentario a un stash.

Aplicando los cambios del stash

Si tienes cambios en el almacén temporal (*stash*) y quieres recuperarlos tienes dos opciones para hacerlo. La primera opción los recuperará pero los eliminará de la pila mientras que la segunda forma los aplicará y los mantendrá igualmente en el *stash*.

Aplicar y eliminar cambios del stash

Como hemos dicho, el almacén temporal es como una pila donde se van acumulando los cambios. Por eso, es muy conveniente tener un sub comando `pop` que aplicará los últimos cambios almacenados. Una vez aplicados los sacará del almacén temporal.

```
$ git status
On branch main
nothing to commit, working tree clean

$ git stash list
stash@{0}: On main: Changes on index.html file
stash@{1}: On main: Trying new images types

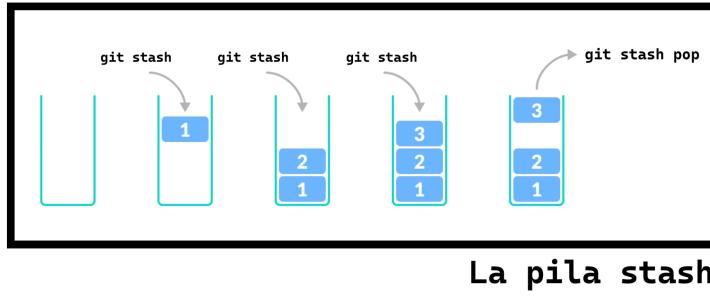
# le decimos que queremos aplicar los últimos
# cambios que tenemos en el almacén temporal
$ git stash pop
On branch main

Changes not staged for commit:
  modified:   index.html

Dropped refs/stash@{0} (32b3aa1d185dfe6d57b3c3cc3b32cbf3e380cc6a)

# vemos los cambios que quedan en el almacén
$ git stash list
stash@{0}: On main: Trying new images types

# se ha eliminado el índice 0 y el que estaba
# en el índice 1, ahora pasa a ser el 0
```



Los stashes se guardan en forma de pila por lo que si haces un pop, entonces recuperarás directamente el último guardado y lo extraerá de ahí

Si tienes archivos modificados en tu directorio de trabajo al aplicar un stash y estos son los mismos que se verían afectados por el stash, entonces **podrías recibir un problema indicando que tus cambios locales tienen que ser grabados, eliminados o guardados en otro stash antes de seguir**. Los cambios no aplicados no se perderán de la pila.

Aplicar un stash sin eliminarlo

Mientras que `pop` aplica y elimina un stash de la pila temporal, `apply` lo aplica sin eliminarlo. Es bastante útil cuando quieras aplicar un mismo cambio a diferentes ramas o, simplemente, por si quieres aplicar un cambio y conservarlo en el stash.

```
# para aplicar el último cambio que hemos guardado
$ git stash apply

# para aplicar el cambio guardado en un índice
$ git stash apply stash@{0}

# también se puede usar directamente el número
$ git stash apply 2
```

Crea una rama a partir de un stash

A veces los cambios que has hecho en el almacén temporal son demasiado golosos para dejarlos ahí. Por suerte, `git stash` te ofrece un comando para comenzar una nueva rama a partir de un stash.

Para ello, debes usar el comando `git stash branch` o `git stash branch <nombre de la rama>` e indicarle el índice del stash que quieras usar. El índice lo puedes ver al usar el comando `git stash list` envuelto entre llaves `stash@{n}` donde `n` es el índice del stash.

```
$ git stash list
stash@{0}: On my-branch: Trying new images types
stash@{1}: On my-branch: Optimize styles

# le indicamos a git que queremos crear una rama a partir
# del stash que está en el índice 1
$ git stash branch optimize-css 1

Switched to a new branch 'optimize-css'
On branch css
Changes not staged for commit:
  modified:  styles.css

Dropped refs/stash@{1} (aca9b94292a7d37977095916cd9d212c09e137b6)
```

¡Ten en cuenta que al hacer esta operación **el stash se perderá de la pila temporal!**
Por desgracia no existe ningún argumento que puedas pasar para evitar esto.

Crear una rama de un stash es útil cuando los cambios que almacenaste en el stash podrían dar conflictos al aplicarlos en la rama. Dicho de otra forma, que los archivos del directorio de trabajo difieren tanto que recuperar los cambios del stash podría ser un lío.

Esto es porque la rama que va a crear lo hará con el `HEAD` apuntando al `commit` cuando se creó el stash. Así no tendrás conflictos al crear la rama.

Eliminando el almacén temporal

Si quieras eliminar el almacén temporal, puedes usar el comando `git stash drop` e indicar la posición del stash que quieras eliminar.

```
# revisamos todos los stashes disponibles
$ git stash list
stash@{0}: On my-branch: Trying new images types
stash@{1}: On my-branch: Optimize styles

# eliminamos el stash en la posición 1
$ git stash drop 1
Dropped refs/stash@{1} (32b3aa1d185dfe6d57b3c3cc3b32cbf3e380cc6a)
```

Si quieres que el `drop` no muestre ningún tipo de información, puedes pasar el argumento `-quiet` o `-q`.

Si tienes claro que lo que quieres es limpiar la pila entera de *stashes*, puedes usar el comando `git stash clear`.

```
# eliminar todos los stashes
$ git stash clear
```

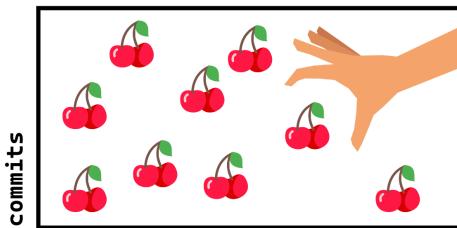
¡Ten mucho cuidado! Limpiar la pila del almacén temporal no requiere ningún tipo de confirmación. Al ejecutar el comando `git stash clear`, todos los stashes serán eliminados... sin preguntas ni segundas oportunidades.

Trucos con Git

¿Cómo puedo aplicar los cambios de un commit a otro?

Al trabajar con tantas ramas diferentes, a veces puede ser útil aplicar los cambios de un commit que hicimos en el pasado o que está disponible en otra rama. No es raro que un colega haya creado un *hotfix* en una rama y que queramos aplicarlo también en la nuestra o que un commit de hace unas semanas lo queramos volver aplicar tal cuál.

Para hacer esto, podemos utilizar el comando `git cherry-pick` que nos permite aplicar los cambios de uno o varios commits de cualquier rama a la rama actual sin necesidad de hacer un merge.



Literalmente, cuando hablamos de Cherry Pick, estamos hablando de selección de cerezas. Donde cada cereza es un commit.

Para poder usarlo, necesitamos conocer el SHA1 del commit (o commits) que queremos aplicar `git cherry-pick <commit-sha>`:

```
# aplicamos el commit 4ccb6d3
git cherry-pick 4ccb6d3

# aplicamos el commit 4ccb6d3 y el commit 8f9f8b8
git cherry-pick 4ccb6d3 8f9f8b8
```

También podemos usar el parámetro `-e` para editar el mensaje de commit original (por defecto, usa el mensaje del commit que estamos seleccionando)

```
git cherry-pick 4ccb6d3 8f9f8b8 -e
```

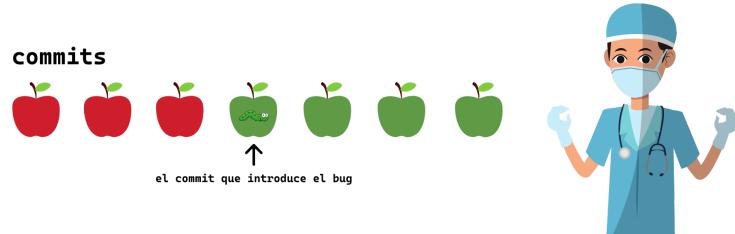
Esto te abrirá el editor de texto que tengas configurado por defecto y podrás editar el mensaje del commit.

Para poder hacer `git cherry-pick` tu directorio de trabajo debe estar limpio de archivos modificados.

Cómo detectar qué commit es el que ha introducido un bug

Imagina que estás en un proyecto y, de repente, hay un error en producción pero, desde el último pase, se han hecho cientos de commits. **¿Cómo sabes qué commit es el culpable?** Hacerlo a mano es una tarea difícil...

Para ello es bueno conocer `git bisect`. *Bisect* significa partir por la mitad y es justamente lo que va a hacer este comando: ir dividiendo toda la pila de commits en dos partes, una parte de la pila contendrá el error y otra parte no.



Con `git bisect` vamos dividiendo la lista de commits en dos para encontrar el commit que fue culpable de añadir un bug en nuestra app

Aunque algo así lo podríamos hacer de forma manual con `git checkout` esta herramienta es mucho más eficiente. Vamos a ver cómo lo deberíamos usar.

Primero, lo iniciamos:

```
$ git bisect start
```

Ahora tenemos que marcar el error. Para ello, vamos a ir a la rama de producción y ejecutamos `git bisect bad`. Si ya sabemos el commit del error, podemos pasar directamente el número de commit, pero en nuestro ejemplo vamos a usar `HEAD` para que el commit que estamos viendo sea el que está en producción.

Ya hemos indicado cuál es el commit que sabemos que tiene el error, ahora nos toca indicar un commit que sabemos que funciona correctamente. Podemos ir bastante atrás (por ejemplo, una semana atrás donde no ocurría el problema) y hacemos checkout a ese commit.

```
$ git checkout 587d364d # 587d364d es un commit de hace una semana  
$ git bisect good # indicamos que este commit sí funcionaba
```

También lo puedes hacer en un solo comando con `git bisect good 587d364d`.

Una vez hecho esto, `git bisect` cambiará el `HEAD` por un commit entre los sospechosos y nos indicará el número de pasos que deberíamos hacer para encontrar el error y el número de commits por revisar.

```
Bisecting: 16 revisions left to test after this (roughly 4 steps)  
[92e11f055a73eb61ca5c17657d84f8340b9bcc57] Update youtube count
```

En este punto deberemos probar el código para ver si el commit que probamos contiene el error. Si contiene el error ejecutamos `git bisect bad` y, si no lo contiene, ejecutamos `git bisect good`.

En cualquiera de los dos casos, `git bisect` volverá a cambiar el `HEAD` por otro commit para que volvamos a hacer la prueba y repetiremos los pasos hasta que definitivamente nos indique en qué commit se introdujo el problema.

```
$ git bisect good  
a6ee8be012ecd323a0cf0ba5be0c8e85bcad5d11 is the first bad commit  
commit a6ee8be012ecd323a0cf0ba5be0c8e85bcad5d11  
Merge: a053f1f 94fc4d4  
Author: Miguel Ángel Durán <miduga@gmail.com>  
Date:   Sat Jul 3 16:42:27 2021 +0200
```

Merge pull request #63 from midudev/new-search-less-deps

New search improving perf

```
assets/js/scripts.js      | 135 ++++++-----  
assets/styles/global.css |  5 +-  
layouts/_default/baseof.html |  1 -  
layouts/partials/logo.html |  77 ++++++-----  
4 files changed, 104 insertions(+), 114 deletions(-)
```

Una vez que tengamos el commit, debemos ejecutar `git bisect reset` para restablecer el `HEAD` correcto y finalizar el proceso.

Ahora que sabes esto... no uses este comando para señalar a un colega de trabajo. Todos cometemos errores. Y somos un equipo. :)

¿Quién ha tocado este fichero? ¿Quién ha hecho cambios?

Con `git log` somos capaces de ver el historial de cambios en general pero a veces queremos saber qué cambios han habido en un fichero en específico. Para ello tenemos que usar el comando `git blame`.

`git blame` necesita como parámetro el fichero que queremos inspeccionar para conocer el historial de cambios que ha recibido y la autoría de estas alteraciones.

```
# ¿qué cambios han habido sobre el fichero src/main.js?  
$ git blame src/main.js  
  
c3fe8972 src/main.js (midudev ...  
c603b8aa src/main.js (Jorge del Casar ...  
789bd97d src/main.js (midudev ...  
c3fe8972 src/main.js (midudev ...  
7392da07 src/main.js (Juan Vasquez ...
```

En la primera columna encontramos el número de cambio (c3fe8972), en la segunda el fichero que se alteró y en la tercera columna el nombre del autor del cambio (midudev). Después encontramos la fecha en la que se realizó el cambio y, por último, la línea que el autor alteró.

Por motivos de formato del libro no he incluido toda la salida del comando pero en los puntos suspensivos (...) está la información de la cuarta y quinta columna (fecha y cambio realizado).



Aunque este comando, como dice el nombre, te permite buscar al culpable... no lo uses para culpar a nadie. Al final, lo que llega a producción es responsabilidad de todo el equipo.

git blame es un comando muy potente que te permite enfocarte en un archivo y, además, filtrar a través de algunas opciones la forma de inspeccionar el historial de cambios del fichero. ¡Incluso puedes hacerlo a nivel de líneas! Te dejo algunas opciones para que las pruebes:

```
# Inspecciona sólo los cambios entre la línea 5 y 10
$ git blame -L 5,10 src/main.js

# ignora los cambios que son espacios en blanco
$ git blame -w package.json

# muestra la dirección de correo electrónico en lugar
# del nombre de usuario
$ git blame -e README.md

# detecta líneas que se han movido o copiado
# en el mismo fichero para mantener
# el autor original de esas líneas
$ git blame -M src/main.js

# detecta líneas que se han movido o copiado
# desde otros archivos y mantiene el
# autor original de las líneas
$ git blame -C src/main.js
```

¿Cómo puedo saber quién añadió una línea por primera vez?

Si intentas usar `git blame` para saber quién fue la primera persona que añadió una línea a un fichero... puedes encontrarte que puede ser algo engoroso, ya que `git blame` está pensado para que encuentres los cambios recientes.

Para eso es mejor usar `git log` ya que gracias a la opción de búsqueda `-S` puedes buscar líneas de código fácilmente y conocer la autoría de esos cambios. Además, es muy fácil ordenar los resultados de forma que conozcas quién fue la primera persona en añadir los cambios gracias a la opción `--reverse`.

```
# buscamos quién añadió por primera vez
# el método createEditor
$ git log -S "createEditor" --reverse

commit 5711c5f07e2fb95c60a08527b521dfb53efdf2be
Author: midudev <miduga@gmail.com>
Date:   Tue Aug 17 21:57:59 2021 +0200

    Add editor file to handle monaco editor creation
```

Recupera un archivo en concreto de otra rama o commit

Al hacer un `git cherry-pick` estamos aplicando el commit entero. De forma que si en el commit estábamos modificando cinco ficheros, entonces tendremos esos cinco ficheros modificados.

Muchas veces queremos limitar esto y recuperar simplemente un fichero desde otra rama. Para hacer esto, podemos usar el comando `git checkout <rama> -- <fichero>`:

```
git checkout old-feature-branch -- package.json
```

Además de usar ramas también podemos proporcionar el *SHA* de un commit en específico, de forma que recuperaremos el fichero de ese commit:

```
git checkout 4ccb6d3 -- package.json
```

Encuentra el primer commit de un repositorio

En ocasiones, ya sea por curiosidad o por necesidad, a veces quieres recuperar el primer commit de un repositorio. Para ello, puedes usar el comando `git log` y ponerlo de la siguiente manera:

```
git log main HEAD~ --oneline | tail -1 | awk '{ print $1 }'
```

Ten en cuenta que, dependiendo del repositorio, es posible que la rama principal no sea `main` y sea `master` o `develop`.

A veces, sin embargo, esto puede no funcionar ya que el primer commit no se hizo en la rama que ahora es la principal o existen demasiados forks que no puede encontrarse de esta forma.

```
$ git rev-list --max-parents=0 HEAD
```

Descubre el máximo contribuidor de un repositorio

Para conseguir la lista de contribuidores ordenada por número de contribuciones puedes usar el comando `shortlog`. Este comando resume la salida de `git log`, agrupando los commits por autor y mostrando el número de contribuciones de cada autor.

Usando `-s` conseguimos eliminar la descripción de los commits y muestra sólo el conteo de contribuciones. Con `-n` conseguiremos ordenar la salida por número de contribuciones por autor.

```
$ git shortlog -s -n

55  midudev
 3  Luis Badiali
 2  Aarón García Hervás
 1  Ismael Ramon
 1  Kiko Beats
 1  Dani de la Cruz
```

Si simplemente quieres saber el mayor contribuidor, puedes pasarle la salida al comando `head` para quedarte con el primero.

```
$ git shortlog -s -n | head -1

55  midudev
```

El número de contribuciones a un repositorio no es una medida de valor. En el caso de este ejemplo, puede ayudarte a saber quién es el contribuidor más activo... pero eso no significa que sean las contribuciones más valiosas. ¡Tenlo en cuenta!

Recupera todos los commits para un usuario en específico

A veces puede ser útil saber qué commits han sido hechos por un usuario específico. Con `git log` puedes filtrar usando el argumento `--author` y pasando como valor una expresión regular para buscar el usuario que te interese.

```
$ git log --author=<pattern>
$ git log --author=^Juan
$ git log --author=Miguel
```

También puedes usar el correo electrónico como un patrón para filtrar:

```
# filtra por el usuario con este correo electrónico
$ git log --author=miguelangel.duran@adevinta.com

# filtra todos los usuarios cuyo email termina con adevinta.com
$ git log --author=adevinta.com$
```

Clonar un repositorio sin descargar todo el histórico

Si intentas clonar un proyecto muy grande de un repositorio remoto es posible que encuentres que **el proceso tarda bastante tiempo o que el espacio que ocupa en disco es muy grande**.

Por ejemplo, vamos a **clonar el proyecto de React** desde su repositorio remoto alojado en GitHub:

```
$ git clone git@github.com:facebook/react.git
Cloning into 'react'...
remote: Enumerating objects: 193700, done.
remote: Counting objects: 100% (16/16), done.
remote: Compressing objects: 100% (15/15), done.
remote: Total 193700 (delta 3), reused 4 (delta 1), pack-reused 193684
Receiving objects: 100% (193700/193700), 163.38 MiB | 2.59 MiB/s, done.
Resolving deltas: 100% (136680/136680), done.
```

El resultado ha sido de **casi un minuto de descarga y un espacio de 163MB de disco duro**.

Esto es normal. Tienes que considerar que al clonar el repositorio y por la naturaleza distribuida de Git, **estás recuperando todos los commits y ramas que se han hecho hasta el momento**.

Pueden existir proyectos incluso más grandes. Imagina cuánto tiempo puede tomar clonar el núcleo de Linux, por ejemplo.

Para estos casos, en los que clonar el repositorio puede tomar demasiado tiempo y/o espacio, puedes usar el comando `--depth` para **evitar descargar todo el historial completo** y especificar el número de revisiones que necesitas.

Por ejemplo, puedes considerar usar `--depth=1` si solo te interesa recuperar el último commit grabado en el repositorio.

```
$ git clone --depth=1 <repo>
```

Vamos a probar otra vez con el repositorio de React:

```
git clone git@github.com:facebook/react.git --depth=1

Cloning into 'react'...
remote: Enumerating objects: 2412, done.
remote: Counting objects: 100% (2412/2412), done.
remote: Compressing objects: 100% (2099/2099), done.
remote: Total 2412 (delta 460), reused 892 (delta 207), pack-reused 0
Receiving objects: 100% (2412/2412), 5.37 MiB | 5.07 MiB/s, done.
Resolving deltas: 100% (460/460), done.
```

Bajar sólo la última revisión (--depth=1): *3 segundos y 5.37MB de disco*.

Bajar todo el historial: *59 segundos y 163.38MB de disco*.

La diferencia es abismal y puede ser clave para optimizar el proceso de clonado, especialmente en procesos donde la velocidad (como un proceso de integración continua) puede ser importante.

Si por cualquier caso necesitas recuperar todo el histórico de un repositorio clonado con --depth puedes usar el siguiente comando:

```
$ git pull --unshallow
```

Vuelve a la rama previa en la que estabas trabajando

Cuando trabajamos con ramas es normal ir saltando constantemente de una a otra y es complicado recordar de memoria el nombre de cada una de ellas. A veces simplemente quieres volver a la rama anterior.

Por suerte, existe una forma muy sencilla de volver a una rama anterior gracias a una sintaxis especial que soporta `git checkout` y `git switch`.

```
# miramos en qué rama estamos ahora
$ git branch --show-current
main

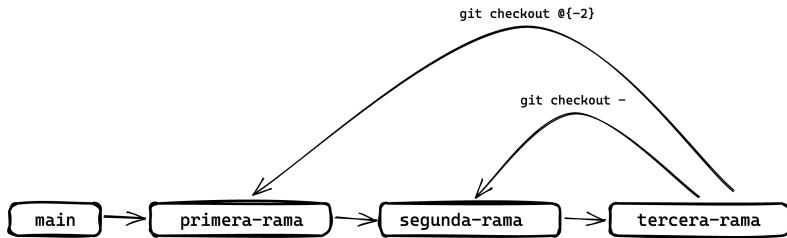
# cambiamos a la rama EPRSLC-16599-discard-cv
$ git switch EPRSLC-16599-discard-cv

Switched to branch 'EPRSLC-16599-discard-cv'
Your branch is up to date with 'origin/EPRSLC-16599-discard-cv'

# ;quiero volver a la rama anterior!
$ git switch -
Switched to branch 'main'
Your branch is up to date with 'origin/main'.

# si no tienes compatibilidad con git switch, usa:
$ git checkout -
Switched to branch 'EPRSLC-16599-discard-cv'
Your branch is up to date with 'origin/EPRSLC-16599-discard-cv'
```

Eso hará que volvamos a justamente la rama anterior a la que estábamos trabajando. Existe otra sintaxis un poco más potente que es `@{-N}` donde `N` es el número de revisiones a saltar, así que para volver justamente a la anterior, deberíamos hacer `@{-1}` y para volver dos atrás `@{-2}`.



De esta forma vas a poder cambiar entre ramas de forma muy sencilla sin necesidad de recordar nombres

Veamos un ejemplo práctico. En esta ocasión vamos a usar `git checkout`, pero recuerda que también puedes usar `git switch`:

```

$ git checkout main
Switched to branch 'main'

$ git checkout primera-rama
Switched to a new branch 'primera-rama'

$ git checkout segunda-rama
Switched to a new branch 'segunda-rama'

$ git checkout tercera-rama
Switched to a new branch 'tercer-rama'

$ git checkout @{-2} # volvemos dos atrás
Switched to branch 'primera-rama'

```

Descargar los ficheros de un repositorio remoto sin tener que clonarlo

Cuando clonamos un proyecto de un repositorio remoto de Git, también estamos recuperando el directorio `.git` con todo el histórico y, por lo tanto, en nuestro sistema se comportará como un proyecto de Git.

A veces sólo nos interesa descargar los ficheros para poder ejecutar o leer algo en nuestro sistema, sin necesidad de usarlo como repositorio de git.

Para ello, puedes usar el truco de clonar el repositorio sin descargar todo el histórico y luego borrar el directorio `.git`:

```
$ git clone --depth=1 git@github.com:midudev/midudev.git
$ cd midudev
$ rm -rf !$/git
```

Otra opción es usar la herramienta `degit` de Rich Harris (creador de *Rollup* y *Svelte*). Para ello tendrás que tener instalado *Node.js* y *npm* instalados en tu ordenador.

Usando `degit` le indicamos la dirección del repositorio remoto (ya sea su forma con `SSH` o `HTTPS`) y el directorio en el que queremos que se descargue el proyecto:

```
$ npx degit git@github.com:midudev/midudev.git midudev
> cloned midudev/midudev#HEAD to midudev

$ cd midudev
$ ls
README.md instructions package package.json src
```

`npx` es un alias para *Node Package Manager* (`npm`) que te permite instalar paquetes de forma global en una carpeta temporal y ejecutar el binario al vuelo.

Aprovecha el auto-corrector de Git para que ejecute comandos parecidos

Seguramente te ha pasado alguna vez... Has escrito `git comit` (con una `m` en lugar de dos) y Git te ha comentado que lo has escrito mal pero que se parece a un comando válido llamado `commit`:

```
$ git comit  
git: 'comit' is not a git command. See 'git --help'.  
  
The most similar command is  
  commit
```

Esto es muy útil, ya que te avisa que lo has escrito mal y te permite corregir el error en la posterior ejecución. Sin embargo, existe una forma todavía más interesante de conseguir esto.

Puedes hacer que Git ejecute automáticamente la primera sugerencia de forma que no tengas que volver a escribir nada. Para ello, tienes que configurar la opción `help.autocorrect` e indicar el tiempo de espera.

```
# ejecuta el comando sugerido después de 2 segundos  
$ git config --global help.autocorrect 20
```

No, no hay un error en el comando. Para esperar 2 segundos, hay que pasarle 20. Eso es porque el número que representa es la décima parte de un segundo.

Una vez configurado, verás que cuando ejecutes un comando mal, si encuentra una sugerencia, te aparecerá este aviso:

```
$ git comit  
  
WARNING: You called a Git command named 'comit', which does not exist.  
Continuing in 2.0 seconds, assuming that you meant 'commit'.  
# después de dos segundos...  
On branch main  
Your branch is up to date with 'origin/main'.
```

Puedes configurar el tiempo de espera con lo que encuentres más cómodo para ti. **Igual 2 segundos es demasiado rápido.** Si dejas el suficiente tiempo, ten en cuenta que podrías evitar la ejecución pulsando *Ctrl+C*.

Ten en cuenta que esto también funcionará con tus alias de comandos. De forma que si tienes un alias que se llama `st` y escribes `ts`, es posible que Git también termine ejecutando el alias correcto.

Domina el formato corto de `git status`

Ya hemos visto que `git status` nos permite ver el estado actual de nuestro directorio de trabajo pero, a veces, es ideal conocer las opciones disponibles para filtrar mejor lo que queremos ver o, simplemente, tener una salida más limpia.

Uno muy útil es la opción `--short` o, su forma corta, `-s` **para ver la salida lo más limpia posible** ya que, por defecto, se usa la opción `--long` que es mucho más verbosa.

```
# muestra el estado actual del directorio de trabajo
# con el menor output posible
$ git status -s

M manuscript/changelog.md
M manuscript/ramas.md
M manuscript/ssh.md
M manuscript/trabajando-con-git-de-forma-local.md
```

Como ves, el formato de salida es mucho más escueto. A la izquierda hay un código de dos letras que indican el estado del fichero. **Normalmente**, el primer espacio es el estado actual del índice y el segundo espacio es el estado actual del directorio de trabajo.

Aunque la salida es más escueta, puede ser difícil de leer si no tienes claro qué significa cada letra. Por eso te puede ser útil esta leyenda:

=	no modificado (espacio vacío)
M	modificado
A	añadido
D	borrado
R	renombrado
C	copiado
U	actualizado pero no fusionado
?	sin rastrear
!	ignorado

Con esta leyenda ahora verás que en el `git status -s` que hemos hecho antes tenemos cuatro archivos modificados que se encuentran actualmente en el directorio de trabajo.

Si añadimos al área de preparación el fichero *changelog.md*, veremos un sutil pero importante cambio en el `git status -s`:

```
$ git add manuscript/changelog.md
$ git status -s

M manuscript/changelog.md
M manuscript/ramas.md
M manuscript/ssh.md
M manuscript/trabajando-con-git-de-forma-local.md
```

Si te fijas bien, ahora el archivo *changelog.md* tiene la banderita de M (modificado) en la primera columna (la columna del índice) y en la segunda columna no tiene ningún valor. Esto se diferencia con el resto de ficheros que siguen modificados en el área de trabajo (segunda columna).

Prueba diferentes combinaciones de comandos y ejecuta `git status -s` para ver qué letras o símbolos te aparecen en las columnas. Revisa la leyenda que te he proporcionado antes e intenta comprender qué pasa con ficheros renombrados, ficheros nuevos, ficheros eliminados...

--porcelain, la opción para que nuestros scripts usen comandos de Git

Muchos comandos de Git ofrecen una opción llamada `--porcelain`. Esta opción nos permite que los comandos de Git sean más fáciles de leer y analizar por programas y scripts, ya que su salida es más limpia y estable entre versiones y configuraciones.

La palabra *Porcelain* viene del material con el mismo nombre: porcelana. La porcelana es el material del que están hechos los baños, normalmente, y tienen un acabado mucho más ideal para el usuario final. Por debajo, en realidad, están las tuberías. Esa sería la analogía por la que se usa ese nombre.

Casi todos los comandos de Git se consideran que son *de porcelana*. Básicamente, todos los que hemos visto en el libro. Pero... ¿sabías que existen otros comandos internos pero públicos como `git count-objects` que son los considerados *plumbing*?

Usando `git status` como ejemplo, podríamos ejecutar lo siguiente:

```
# una salida similar a `git status -s` pero
# sin colores e independiente de la config del usuario
$ git status --porcelain

M manuscript/changelog.md
M manuscript/ramas.md
M manuscript/ssh.md
M manuscript/trabajando-con-git-de-forma-local.md
M manuscript/trucos.md
```

Como puedes ver, en este caso, `git status --porcelain` da una salida similar a `git status -s` pero, si lo ejecutas en tu terminal, verás que en el caso de usar `--porcelain` no tiene ningún tipo de color.

Como la salida es estable y previsible, podríamos crear un pequeño script que nos dijese el número de cambios que tenemos en nuestro directorio de trabajo, de la siguiente forma:

```
# con wc -l contamos el número de líneas  
$ git status --porcelain | wc -l  
5
```

Hay otros comandos que también proporcionan la opción de `--porcelain` como `git commit`. En este caso la salida te indicará el `git status -s` resultante que tendrías después de ejecutar la operación **pero no realiza el commit**. Es una forma de asegurarte que puedes grabar los cambios sin problemas y revisar cómo quedaría el estado de tu directorio de trabajo antes de hacerlo.

Configuraciones a tener en cuenta

¡Importante! Antes de activar estas configuraciones, asegúrate de que entiendes lo que hacen y cómo afectan a tu flujo de trabajo.

Elimina automáticamente referencias remotas (por ejemplo, ramas que tienes en local y ya no existen) que ya no existen en el servidor cuando ejecutas `git fetch`:

```
git config --global fetch.prune true
```

Simplifica la salida de `git status` de forma que solo muestre los ficheros que han cambiado:

```
git config --global status.short true
```

```
# antes  
$ git status
```

```
On branch main  
Your branch is ahead of 'origin/main' by 13 commits.  
(use "git push" to publish your local commits)
```

```
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:   manuscript/ramas.md  
modified:   manuscript/trucos.md
```

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

```
# después  
$ git status
```

```
M manuscript/ramas.md  
M manuscript/trucos.md
```

Añade un color extra para diferenciar las líneas que sólo han sido movidas. Por defecto al hacer un `git diff` tenemos dos colores que nos indican las líneas añadidas o eliminadas. Este color extra nos indica las líneas que han sido movidas:

```
git config --global diff.colorMoved zebra
```

Si quieres sólo hacer push de tu rama actual y simplificar los parámetros que tienes que pasar al hacer `git push`, puedes usar:

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

Errores comunes en Git y sus soluciones

En esta sección, he recopilado algunos errores comunes que pueden surgir en el uso de Git, ya sea por problemas a la hora de trabajar con las ramas o algún tipo de configuración incorrecta.

También, he incluido algunas posibles soluciones para cada uno de ellos, junto a una pequeña explicación para que la próxima vez que te pase no te vuelvas a preguntar por qué no funciona.

Me dice que no es un repositorio de git

Te has leído el libro y estás deseando usar Git. Ejecutas tu primer comando y... ¡error! ¿Qué está pasando?

```
$ git status  
fatal: not a git repository (or any of the parent directories)
```

El problema indica que **estás intentando ejecutar Git en un directorio que no es un repositorio de Git**. Normalmente esto ocurre por dos razones:

- 1. No has ejecutado `git init` en el directorio.** Esto es necesario si, por ejemplo, es un directorio local y no uno que has clonado de un repositorio remoto.
- 2. No estás en el directorio correcto.** Puedes usar los comandos de `git` dentro de las subcarpetas de un directorio que sí haya sido iniciado como repositorio pero no puedes, por ejemplo, hacerlo un directorio superior. Asegúrate que estás en el correcto.

Hago pull y me dice que no es un repositorio

Es posible que alguna vez intentes traerte los cambios de un repositorio remoto y te diga que no es un repositorio de Git. Veamos este ejemplo:

```
$ git pull origi main  
fatal: 'origi' does not appear to be a git repository  
fatal: Could not read from remote repository.
```

Please make sure you have the correct access rights
and the repository exists.

Ese es el error. Es evidente que he escrito mal `origi` pero **no es tanto por escribirlo mal o bien**, si no por asegurarnos que estamos haciendo pull de un repositorio remoto que tengamos configurado. En realidad, que se llame `origin` es circunstancial, ya que podríamos tener nombres de repositorios remotos distintos.

```
# git pull <repositorio-remoto> <rama>  
# miramos qué repositorios remotos tenemos configurados  
$ git remote -v  
repo  git@github.com:midudev/manual-git.git (fetch)  
repo  git@github.com:midudev/manual-git.git (push)  
  
# traemos cambios del repositorio remoto  
$ git pull repo main  
  
# esto fallará porque no tenemos ese repositorio  
$ git pull origin main
```

También puedes hacer `pull` de un repositorio que no tienes configurado como remoto, pero entonces **debes indicar la dirección completa**:

```
$ git pull git@github.com:midudev/manual-git.git main
```

¿Por qué se llama casi siempre `origin`? Bien, es porque es el nombre que da `git` automáticamente al repositorio remoto cuando ejecutas un `git clone`. Es como en su día con `master` para la rama principal, pero esto es algo que puede cambiar. Por eso es importante entender el concepto y no pensar que el concepto es el nombre.

He escrito mal el último commit

No pasa nada. Si has cometido un error en el mensaje del commit anterior y todavía no has hecho `push` puedes solucionarlo así:

```
$ git commit --amend -m "Este es el mensaje correcto"
```

Pero ya he hecho `push`. ¿Qué hago ahora?

Si ya has hecho `push`, sólo te queda hacer un `push --force`. **Te aconsejo que no lo hagas porque el riesgo supera con creces el beneficio.** El historial de commits no es importante que esté impoluto. No pasa nada. Es mejor volver a crear un commit con el mensaje correcto y después hacer `push` con normalidad. Si todavía sigues con ganas de hacerlo... sigue leyendo.

Si sabes que ningún compañero ha trabajado con el commit incorrecto, entonces puedes hacer:

```
# Arreglamos el último commit con el mensaje correcto
$ git commit --amend -m "Este es el mensaje correcto"
# Forzamos el push y sobrescribimos el historial de commits
$ git push --force origin nombre-de-la-rama
```

Muy importante: Antes de hacer el `push` asegúrate que la rama a la que vas a hacer `push` es la correcta y que tienes todos los cambios sincronizados. De lo contrario, van a desaparecer cambios del historial.

Si no es el último commit el que quieres cambiar

¡Cuántos problemas! Si ya tu commit está hundido en el historial. ¡Déjalo tranquilo! Es que no compensa NADA teniendo en cuenta la de problemas que te puedes encontrar, especialmente si estás trabajando en código compartido.

Si todavía sigues con ganas de hacerlo... tendrás que utilizar `rebase`. Te recomiendo que uses el `rebase` interactivo, de forma que podrás reescribir el historial de commits y quedarte con sólo aquellos que te interesen. Para ello, ejecuta:

```
# Reescribimos el historial de commits  
# Queremos reescribir los últimos 5 commits  
$ git rebase -i HEAD~5
```

Muy importante: Recuerda que `rebase` reescribe el historial, moviendo commits. Por lo tanto, **no debes hacer `rebase` si hay commits que ya han sido compartidos**. Si no estás seguro, no lo hagas, ya que podrías dejarlo peor.

He escrito mal la rama que he creado

Con las prisas te puedes equivocar al crear el nombre de una rama. No pasa nada. Existe una forma muy sencilla de renombrarla sin necesidad de borrar y crear:

```
# creamos nuestra rama mal
$ git switch -c rama-nueba

# ¡AH! Mis ojos. Es con V!
# Renombramos el nombre de la rama
$ git branch -m rama-nueba rama-nueva
```

Si ya habías hecho push

Si ya habías hecho `push` de tu rama, no pasa nada. Asegúrate que en local tienes todos los cambios, ejecuta los comandos que hemos comentado antes y ahora ejecuta.

```
# Eliminamos la rama que está mal
$ git push origin --delete rama-nueba
# Enviamos la rama con el nombre correcto
$ git push origin rama-nueva
```

He hecho un git push y me da error

Es muy común que al intentar hacer un push de tus cambios te de un error. No pasa nada. Normalmente lo que pasa es que tu repositorio local no está sincronizado con el repositorio remoto. Por eso, **no puedes enviar tus cambios ya que Git no sería capaz de saber cómo tiene que integrarlos.**

Este sería el error:

```
# Despues de hacer unos commits, decidimos enviar
# los cambios al repositorio remoto
$ git push origin main

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

Como dice el error, **la actualización ha sido rechazada ya que el punto de la rama actual está detrás del punto remoto.** Es necesario integrar los cambios remotos (con `git pull ...`) antes de volver a hacer un `git push`.

He hecho commits a la rama principal que debían realizarse en otra rama

A todos nos ha pasado alguna vez. Te has dejado llevar por el ansia y te has puesto a hacer commits en la rama principal como si no hubiese un mañana. Normalmente no has hecho `push` (si es así ya entraríamos en otro tema).

Imagina que has hecho 3 commits en la rama principal pero debían ir a una rama. Existen diferentes maneras de hacerlo, pero la más limpia y sencilla sería esta:

```
# Nos aseguramos que estamos en la rama principal
$ git branch
* main

# Creamos la rama que debíamos haber
# creado desde el principio
$ git branch rama-con-cambios

# Así nos crea una rama con todos los commits actuales
# que teníamos en main

# SIN CAMBIAR DE RAMA (seguimos en main)
# Movemos la rama principal 3 commits atrás
$ git reset --keep HEAD~3

# Ahora podemos cambiar a la nueva rama
$ git switch rama-con-cambios
```

¿Y si la rama ya existía?

Si los commits que has hecho en master debían haber ido a una rama que ya existía, puedes usar `git merge` para integrar los commits de la rama principal a la rama.

```
# Vamos a la rama que queremos que tenga los commits
$ git switch rama-con-cambios

# Hacemos merge de la rama principal
$ git merge main
This will add the additional commits to the existing branch.

# Volvemos a la rama principal
$ git switch main
```

Movemos la rama principal 3 commits atrás
\$ git reset --keep HEAD~3

xcrun: error: invalid active developer path

Este es un error exclusivo de macOS pero muy frecuente.

A veces, tras actualizar macOS a una nueva versión de su sistema operativo, o por actualizar otras dependencias, es posible que al intentar utilizar algún comando de Git, la terminal te devuelva un error un poco críptico que dice así:

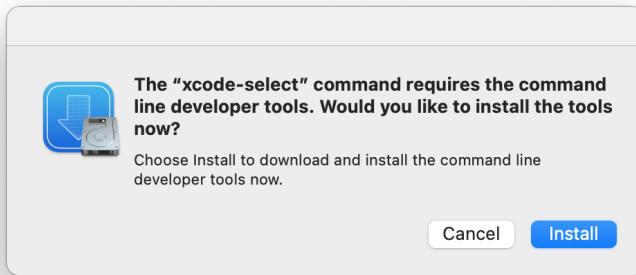
```
$ git status
xcrun: error: invalid active developer path (/Library/Developer/CommandLineTools),
missing xcrun at: /Library/Developer/CommandLineTools/usr/bin/xcrun
```

En realidad este error es que **deja de detectar que has aceptado los términos de uso de estas herramientas** y deja de poder utilizarlas. Otra razón puede ser que las que tenías instaladas se han eliminado por el paso de actualización y tienes que reinstalarlas.

Sea como sea, para solucionarlo, lo más sencillo es que las vuelvas a instalar. De esta forma se volverán a registrar en el path correcto. Para ello, ejecuta este comando:

```
$ xcode-select --install
```

Te debería aparecer una ventana preguntando si quieres instalar las herramientas de desarrollo que `xcode-select` necesita para poder ejecutarse. Le decimos que sí.



Ventana con el mensaje que va a instalar las herramientas de desarrollo

Tras esto, comenzará una descarga y después una instalación que puede durar unos minutos. Una vez finalizado, prueba de nuevo el comando de git en tu terminal y debería funcionar correctamente.

Soluciones si sigue sin funcionarte

En este punto, es raro que todavía no te funcione pero aquí te dejo algunas soluciones más que pueden ayudarte.

1. Reinicia el ordenador para asegurarte que se han registrado bien los cambios.
2. Si sigue sin funcionar, ejecuta en la terminal `sudo xcode-select --reset` para resetear toda la configuración.
3. Si todavía no funciona, prueba a descargar e instalar manualmente las *Command Line Tools for Xcode* desde la página de [Apple Developers](#).

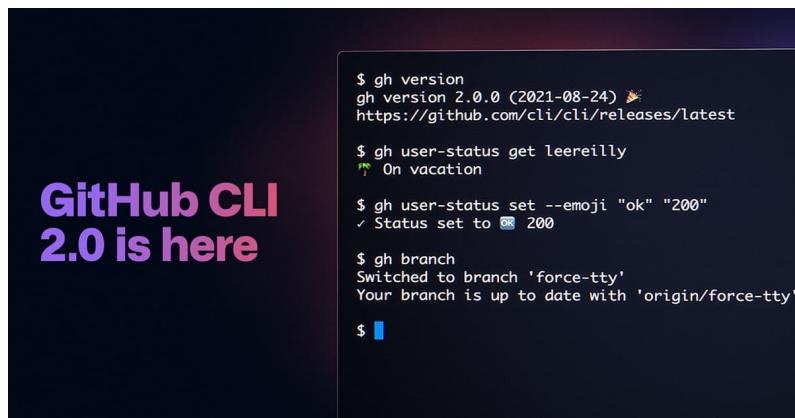
GitHub CLI

Este capítulo ha sido escrito basándose en la versión 2.21.1 de gh

Hoy en día **GitHub** es el mayor de todos los repositorios de Git en el mundo. En 2018, [Microsoft adquirió GitHub por 7.500 millones de dólares](#) y, desde entonces, no ha dejado de crecer.

Eso hace que **el uso de Git esté muy ligado a GitHub** y, por eso, mucha gente lo confunde o cree que es lo mismo. Pero ya hemos visto que no es así y que Git es una tecnología independiente a GitHub, que es donde se guardan los repositorios remotos de Git.

Dicho esto, este capítulo viene a explicarte la línea de comandos de `gh` para GitHub. Esta línea de comandos te permite interactuar con el repositorio remoto de GitHub.



The screenshot shows a terminal window with a dark background. On the left, there is a large watermark-like graphic with the text "GitHub CLI 2.0 is here" in white. On the right, the terminal displays several command-line interactions:

```
$ gh version
gh version 2.0.0 (2021-08-24) ✨
https://github.com/cli/cli/releases/latest

$ gh user-status get leereilly
⚡️ On vacation

$ gh user-status set --emoji "ok" "200"
✓ Status set to 🚧 200

$ gh branch
Switched to branch 'force-tty'
Your branch is up to date with 'origin/force-tty'

$ █
```

GitHub CLI 2.0 apareció en agosto del 2021 y entre sus novedades destacan el soporte a GitHub Actions y la posibilidad de usar extensiones

Te ayudará a trabajar con tu repositorio Git que está hospedado en GitHub. Por ejemplo, podrás crear *Pull Requests*, *Issues* y muchas cosas más desde tu terminal, sin necesidad de ir a la web de GitHub.

Es importante que no confundas la línea de comandos de Git `git` con la línea de comandos de GitHub `gh`.

Instalando gh en el sistema...

A diferencia de `git`, `gh` no está instalado por defecto en muchos sistemas por lo que es muy probable que tengas que instalarlo para poder utilizarlo. Te dejo aquí las instrucciones de los distintos sistemas operativos:

En macOS

Puedes instalar `gh` en tu sistema operativo usando [Homebrew](#) ejecutando el siguiente comando:

```
$ brew install gh
```

Si lo prefieres también puedes usar [MacPorts](#) con el siguiente comando:

```
$ sudo port install gh
```

En Linux

para Debian y Ubuntu puedes instalar `gh` con el gestor de paquetes `apt`:

```
$ curl -fsSL https://cli.github.com/packages/githubcli-archive-keyring.\
gpg | sudo gpg --dearmor -o /usr/share/keyrings/githubcli-archive-keyri\
ng.gpg

$ echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/key\
rings/githubcli-archive-keyring.gpg] https://cli.github.com/packages st\
able main" | sudo tee /etc/apt/sources.list.d/github-cli.list > /dev/nul\
l

$ sudo apt update
$ sudo apt install gh
```

Para **Fedora**, **CentOS** y **Red Hat**, puedes instalar `gh` usando el siguiente comando:

```
$ sudo dnf config-manager --add-repo https://cli.github.com/packages/rp\
m/gh-cli.repo
$ sudo dnf install gh
```

En Windows

Lo más sencillo es que descargues el instalador oficial de GitHub desde [aquí](#). Descarga el fichero `gh_1.x.0_windows_amd64.msi`, ejecútalo y sigue los pasos.

Realizando la configuración inicial de gh

Antes poder empezar a usar el comando `gh` tenemos que autenticarnos en GitHub. Para ello, ejecuta el comando `gh auth login`. Verás que te preguntará por tu tipo de cuenta (lo más normal es que uses la de GitHub.com, ya que la de Enterprise Server es para empresas).

```
$ gh auth login

? What account do you want to log into? [Use arrows to move, type to filter]
> GitHub.com
    GitHub Enterprise Server
```

Después de seleccionar el tipo de cuenta, te preguntará cuál es tu protocolo favorito para trabajar en Git. Elegiremos SSH.

```
? What is your preferred protocol for Git operations? [Use arrows to move, type to filter]
> HTTPS
    SSH

? Upload your SSH public key to your GitHub account? [Use arrows to move, type to filter]
> /Users/midudev/.ssh/id_ed25519.pub
    Skip

? How would you like to authenticate GitHub CLI? [Use arrows to move, type to filter]
> Login with a web browser
    Paste an authentication token

! First copy your one-time code: 1A6D-BECA
- Press Enter to open github.com in your browser...
```

```
✓ Authentication complete. Press Enter to continue...

- gh config set -h github.com git_protocol ssh
✓ Configured git protocol
✓ Uploaded the SSH key to your GitHub account: /Users/midudev/.ssh/id_ed25519.pub
✓ Logged in as midudev
```

Usando `gh`

El comando `gh` tiene dos niveles de comandos. El primer nivel sería el contexto sobre el que queremos trabajar. Por ejemplo, si queremos trabajar a nivel de repositorios, usaremos `gh repo`. Entonces, dentro de ese nivel, tendremos una serie de comandos que podremos ejecutar.

Hay muchos contextos disponibles pero nosotros nos vamos a enfocar en los siguientes:

- `repo`: Para crear, clonar, hacer fork y ver repositorios.
- `issue`: Para crear, ver, editar y cerrar issues.
- `pr`: Para crear, ver, editar y cerrar pull requests.
- `gist`: Para crear, ver y editar gists.

Otros disponibles son: *actions, alias, api, auth, browse, completion, config, extension, help, release, run, secret, ssh-key y workflow*

`gh` funciona por defecto sobre el repositorio en el que estamos localmente y **que tenga un repositorio remoto configurado que sea de GitHub**. Si intentas usarlo en un proyecto que es un repositorio de Git válido pero no está enlazado con un repositorio de GitHub remoto te dará el siguiente error:

```
$ gh issue list
no git remotes found
```

Si intentas ejecutar el comando en un directorio que no es un repositorio, te dirá lo siguiente:

```
$ gh issue list
fatal: not a git repository (or any of the parent directories): .git
/usr/bin/git: exit status 128
```

Si te aparece el error: “No default remote repository has been set for this directory”, es porque no tienes un repositorio remoto por defecto. Para solucionarlo, ejecuta `gh repo set-default` dentro de tu directorio de trabajo con un repositorio de GitHub.

`pr`: Administrando Pull Requests desde la línea de comandos

Las *Pull Request* (o petición de cambios) son **una forma de comunicar que quieres hacer llegar unos cambios a un repositorio de GitHub**. Estos cambios pueden ser una mejora en el código, una nueva funcionalidad o la solución a un error en el código.

Con `gh` podemos realizar todas las operaciones posibles sobre las *Pull Request* de un repositorio. Fusionarlas, revisarlas, cerrarlas, abrirlas de nuevo, crearlas, modificarlas... ¡y mucho más! Todo desde tu terminal.

A las Pull Request también se les conoce de forma abreviada como PR, por lo que es posible que a veces veas que en el libro me refiero a ellas de esa forma. Además, como verás, se usa `pr` para el contexto de las Pull Request en el comando `gh`.

¿Cómo puedo revisar una Pull Request de forma local?

¿Alguien ha creado una Pull Request a tu repositorio de GitHub y te gustaría revisarla fácilmente en tu máquina local? Gracias a `gh` es muy fácil conseguirlo. Sólo tienes que ejecutar el comando:

```
$ gh pr checkout <número-de-la-pull-request>

remote: Enumerating objects: 22, done.
remote: Counting objects: 100% (22/22), done.
remote: Compressing objects: 100% (8/8), done.
remote: Total 22 (delta 13), reused 22 (delta 13), pack-reused 0
Unpacking objects: 100% (22/22), 6.02 KiB | 342.00 KiB/s, done.
From github.com:midudev/midu.dev
 * [new ref]          refs/pull/62/head -> responsive-header

Switched to branch 'responsive-header'
```

Este comando hará por detrás **todos los pasos necesarios para sincronizar tu repositorio**: Descargar la rama que corresponda con el número que le has pasado y cambiar tu directorio de trabajo por el contenido de la rama que quieras revisar.

De esta forma, **podrás probar que el código de esta Pull Request es correcto desde tu máquina local**. Muy útil para asegurarte que el código hace lo que se espera.

Si ejecutas el comando pasándole un número de *PR* que no existe, recibirás el siguiente error:

```
$ gh pr checkout 1234
GraphQL error: Could not resolve to a PullRequest with the number of 12\34.
```

Además de usar el número de la PR como parámetro, **también puedes usar la URL de GitHub de la Pull Request e incluso el nombre de la rama que la contiene**. Por ejemplo:

```
# miramos la PR de la rama stylessh:main
$ gh pr checkout stylessh:main

# miramos la PR de esa URL
$ gh pr checkout https://github.com/midudev/codi.link/pull/17
```

issue: Administrando Issues desde la línea de comandos

Las *issues* (o problemas) son **una forma de comunicar y reportar errores o peticiones de mejora en el código de un repositorio**. Así que, a la hora de contribuir a proyectos alojados en GitHub, no sólo se logra con código, sino también con la creación y gestión de *issues*.

Creando una issue desde la línea de comandos

Aunque puedes crear *issues* si vas a la sección *Issues* del repositorio en GitHub, a veces no quieres salir de la terminal para lograrlo. Con `gh` es muy sencillo y te ofrece un montón de opciones para que puedas asignarle el problema a la persona adecuada o lo etiquetes muy fácilmente. Sólo tienes que ejecutar el comando:

```
$ gh issue create
Creating issue in midudev/midu.dev
```

? Title

Una vez añades el título, pulsas *Enter*. Entonces te preguntará si quieres añadir una descripción. Si no quieres añadir una descripción, pulsa *Enter* y seguirás con el siguiente paso.

Aquí puedes añadir metadatos a la *issue* como etiquetas, el proyecto, asignar a una persona, etc. Incluso te da la opción de continuar la creación de la issue directamente en GitHub, abriéndote una nueva ventana en tu navegador. Si prefieres hacerlo por la línea de comandos eventualmente podrás hacer *Submit* y crear la issue.

Ten en cuenta que el título de la issue no puede estar vacío ni ser de más de 100 caracteres. De lo contrario, te fallará la creación de la *issue*.

Si no quieres que te pregunte por el título ni la descripción, puedes pasarlo directamente como argumentos y la issue se creará directamente sin ningún asistente y te devolverá la URL con la *issue* creada.

```
# puedes crear una issue con título y descripción
# así no te aparece el asistente y se crea directamente
$ gh issue create --title "Missing dependency" --body "The dep react is\
missing"

Creating issue in midudev/midu.dev

https://github.com/midudev/midu.dev/issues/73
```

Siquieres puedes crear la *issue* sin título ni descripción pero pasarle otras opciones. De esta forma, sí te preguntará por el título y la descripción de forma interactiva (si no lo pasas como argumentos) pero ya tendrás algunos metadatos añadidos.

```
# también puedes crear una issue con etiquetas
# ojo: las etiquetas deben existir en el proyecto
# o te fallará la creación
$ gh issue create --label "performance,help wanted"

# puedes crearlas de forma separada
$ gh issue create --label bug --label "good first issue"
```

```
# puedes asignar la issue a más de una persona
$ gh issue create --assignee dapelu, jelowing

# incluso a ti mismo
$ gh issue create --assignee @me

# y también asignarlo a un proyecto por nombre
$ gh issue create --project "New Version"
```

Si lo único que quieras es crear una issue desde el navegador pero poder abrir la ventana desde la terminal, también puedes ejecutar `gh issue create --web`, que te abrirá la ventana de GitHub con la issue en blanco.

Editar issues desde la línea de comandos

¿Te has equivocado en algo al crear la *issue*? ¿Quieres añadir un cambio? No te preocunes. Una vez que tienes la *ID* de la issue o su dirección en **GitHub**, vas a poder modificar tantas veces como quieras el error reportado y, lo mejor, sin salir de tu terminal.

```
# Editamos el título y el cuerpo de la issue
$ gh issue edit 42 --title "Editor no funciona" --body "Falla al iniciar"
r :("

# Podemos añadir etiquetas
$ gh issue edit 42 --add-label "bug,help wanted"

# O quitar etiquetas
$ gh issue edit 42 --remove-label "core"

# Cambiar la asignación de la issue
# En este caso me la asigno a mí mismo y quito a @camimaya
$ gh issue edit 13 --add-assignee @me --remove-assignee camimaya

# También podemos modificar el proyecto asignado
$ gh issue edit 23 --add-project "Next" --remove-project old,v1

# El hito al que pertenece...
$ gh issue edit 23 --milestone "Version 1"

# Podemos añadir el comentario de la issue
# desde un archivo que ya existe
$ gh issue edit 23 --body-file body.txt
```

Como ves, usando el comando `edit` de las *issues* no puedes cambiar su estado de abierto a cerrado. Eso es porque, como ya verás más adelante, **existe un comando específico para eso.**

Listando y visualizando *issues*

Si tienes un proyecto de código abierto, seguramente te van a llegar muchas *issues*. Aprender a manejarlas va a ser una de los retos más importantes a los que te vas a enfrentar. Por suerte, con `gh`, hacerlo es muy fácil si dominas todas las posibilidades que te ofrece.

Lo más básico es, simplemente, mostrar una lista de las que hay con:

```
# listar las issues del repositorio
$ gh issue list

Showing 3 of 3 open issues in SUI-Components/sui-components

#1613 Demo PhotoUploader    issue with documentation    about 1 month ago
#1612 Demo Modal           issue with documentation    about 1 month ago
#1596 Clean V7 values      enhancement                 about 2 months ago
```

Por defecto te muestra un máximo de 30 issues. Puedes limitarlo usando el parámetro `-L`.

Esto es lo más básico pero, a partir de aquí, **tienes muchas opciones para filtrar, ordenar, etc.** Te muestro una lista de ejemplo pero no es una lista completa ya que las combinaciones son demasiadas.

```
# filtrar issues por etiquetas
$ gh issue list -l "performance" -l "good-first-issue"

# muestra las issues creadas por el usuario vekpol
$ gh issue list --author vekpol

# muestra las issues asignadas a mí
$ gh issue list -a @me

# las issues que ya han sido cerradas asignadas a mí
$ gh issue list --state closed -a @me
```

```
# usa una consulta para buscar, filtrar y ordenar las issues
$ gh issue list --search "error no:assignee sort:created-asc"

# abre una nueva ventana con la lista de issues en la web
$ gh issue list --web

# también puedes ver las issues de webs externas
# muestra las issues de React que sean buenas para principiantes
$ gh issue list -R facebook/react -l "good first issue"
```

¿Quieres tener la salida del comando en formato json? Prueba a ejecutar este comando
gh issue list -R midudev/codi.link --json id,title. Ahora que ves la salida,
prueba a añadir otros campos como body, state o labels con otros repositorios, para ver
la salida.

Un resumen rápido de las *issues* que nos interesan

Otra forma de tener un vistazo rápido a las *issues* que más nos pueden interesar de un repositorio es usando el comando `gh issue status`. La salida nos hará un buen resumen de las issues que tenemos abiertas, asignadas y en las que nos mencionan.

```
$ gh issue status

Relevant issues in midudev/midu.dev

Issues assigned to you
#59 Improve performance    about 1 day ago
#74 Mobile version fails   about 42 minutes ago

Issues mentioning you
#35 Wrong icon placement   about 3 days ago

Issues opened by you
#75 Slow installation      about 2 minutes ago
```

Leyendo una *issue* desde la terminal

También puedes acceder a toda la información de una *issue* sin necesidad de abrirla en el navegador. Para ello, ejecuta el comando `gh issue view` y te mostrará la información de la *issue* que le indiques (¡Incluso sus comentarios!). Para ello puedes usar la id de la *issue* o su URL en **GitHub**.

```
# leer la issue usando la id
$ gh issue view 35

Guardar parámetros en localStorage #35
Open • Sebass83 opened about 1 day ago • 0 comments
Labels: enhancement

Se pueden guardar los parámetros de la URL en el localStorage para tener un auto guardado. Incluso se podría desactivar desde dentro de las opciones.

View this issue on GitHub: https://github.com/midudev/codi.link/issues/35

javigaralva • 7d • Edited • Newest comment

Me parece muy buena idea! (#2)
```

Ahora, desde la terminal, consigue la lista de problemas reportados en el repositorio facebook/react usando el comando `gh issue list` y, después, ejecuta el comando `gh issue view` con la id de la issue que quieras leer. ¿Recuerdas cómo usar el parámetro `-R` (o `--repo`) para seleccionar otro repositorio?

Cerrar y eliminar issues de nuestro repositorio con `gh`

A la hora de cerrar las *issues* desde la terminal, puedes usar tanto la **ID** de la *issue* como su URL en GitHub.

```
# usando la id de la issue
$ gh issue close 74
✓ Closed issue #74 (Chrome not working)

# usando la URL completa de la issue
$ gh issue close https://github.com/midudev/midu.dev/issues/75
✓ Closed issue #75 (Slow initialization of app)
```

Una vez que la cierras... ¡es posible que te hayas equivocado! Así que la vas a querer **volver a abrir** y esto también lo puedes hacer desde la terminal fácilmente.

```
# reabrimos la issue 75 usando su id
$ gh issue reopen 75
✓ Reopened issue #75 (Slow initialization of app)
```

Si quieras eliminar definitivamente la *issue* de tu repositorio, puedes usar el comando `gh issue delete`. No te preocunes al hacerlo, porque el comando viene con un sistema de seguridad en el que te volverá a preguntar el número de la *issue* que quieras eliminar.

```
# eliminamos la issue 79 usando la URL
$ gh issue delete https://github.com/midudev/midu.dev/issues/79
You're going to delete issue #75.
? This action cannot be reversed. To confirm, type the issue number:
75
✓ Deleted issue #75 (Slow initialization of app).
```

repo: Administrando un Repositorio desde la línea de comandos

Los repositorios son una parte fundamental de GitHub, por lo que es importante que puedas administrarlos desde tu terminal. Gracias a `gh` puedes crear, clonar, editar, eliminar y listar repositorios además de crear bifurcaciones (forks).

Crear un *repository* con `gh`

Muchas veces uso el acceso directo <https://repo.new> para crear un nuevo repositorio de forma rápida y sencilla. Pero aquí hemos venido a hablar de `gh` y, como no podía ser de otra forma, podemos crear un repositorio desde la terminal y con un montón de posibilidades y opciones.

Si quieres crearlo y que te pregunte paso por paso todas las opciones, sólo tienes que ejecutar el comando `gh repo create` y te pedirá que le indiques el nombre del repositorio, su descripción, su visibilidad y otras opciones.

Si ejecutas este comando en tu terminal desde un directorio de trabajo que es un repositorio local inicializado con `git`, este nuevo repositorio remoto creado será añadido como `origin`.

```
# crea un repositorio local con el nombre de gallery-slider
$ git init gallery-slider
# entramos en el directorio de trabajo del repo
$ cd gallery-slider
# creamos el repositorio remoto que se añadirá como `origin`
$ gh repo create
```

```
# nos hará algunas preguntas sobre el repo a crear y al final...
✓ Created repository midudev/gallery-slider on GitHub
✓ Added remote https://github.com/midudev/gallery-slider.git
```

Puedes pasarle el nombre que prefieras al repositorio, pero es importante que lo pongas en minúsculas y sin espacios. Así puedes hacer que el repositorio nuevo no tenga el mismo nombre que la carpeta local.

```
# en la carpeta ~/gallery-slider hacemos
$ gh repo create gallery-slider-wc
✓ Created repository midudev/gallery-slider on GitHub
✓ Added remote https://github.com/midudev/gallery-slider.git

# Puedes usar también otra organización a la que tengas acceso
$ gh repo create my-org/gallery
✓ Created repository my-org/gallery on GitHub
✓ Added remote https://github.com/my-org/gallery.git
```

Si el directorio en el que te encuentras no es un repositorio local, también puedes inicializarlo con `gh repo create` después de crear el repositorio en GitHub. Simplemente te avisará el comando y tendrás que confirmarlo.

```
# ejecutamos desde un directorio que no es un repositorio
$ gh repo create new-project
# nos hará unas cuantas preguntas...
? Visibility Public
? Would you like to add a .gitignore? No
? Would you like to add a license? No
? This will create the "new-project" repository on GitHub.
? Continue? Yes
✓ Created repository midudev/new-project on GitHub
# en este punto nos avisa que creará una carpeta nueva
# llamada 'new-project' con el repo local y apuntando
# al repositorio remoto que acabamos de crear
? Create a local project directory for "midudev/new-project"? Yes
```

Clonando un repositorio con `gh`

Si quieres clonar un repositorio, puedes hacerlo con `gh repo clone`. Este comando te permite clonar un repositorio de GitHub y, además, te permite elegir el nombre del directorio donde se clonará el repositorio.

```
# clonamos el repositorio midudev/midu.dev
$ gh repo clone midudev/midu.dev

# clonamos el repositorio midudev/midu.dev en la carpeta blog
$ gh repo clone midudev/midu.dev blog
```

```
Cloning into 'blog'...
remote: Enumerating objects: 8044, done.
remote: Counting objects: 100% (157/157), done.
remote: Compressing objects: 100% (111/111), done.
remote: Total 8044 (delta 75), reused 101 (delta 29), pack-reused 7887
Receiving objects: 100% (8044/8044), 36.73 MiB | 12.50 MiB/s, done.
Resolving deltas: 100% (4293/4293), done.
```

Como ves, **no hace falta que le indiques la URL completa del repositorio**. Puedes usar el nombre del usuario y el nombre del repositorio directamente.

gist: Administrando gists desde la línea de comandos

Los *gist* de GitHub son repositorios de código pero pensados para ser pequeños y de uso único. Por ejemplo, una pequeña aplicación que te ayude a hacer una copia de seguridad de tu trabajo o un ejemplo de utilidad que te ayude a comprender un tema.

De hecho existen cientos de miles de estos pequeños extractos de código. Puedes [ir a su página para ver todos los que hay disponibles](#) y buscar alguno que pueda interesarte.

Si no quieres tener que abrir al navegador para disfrutar de estos *gists*, no te preocunes. Con `gh` no sólo vamos a poder crear estos trozos de código, si no que también **vamos a poder crearlos y editarlos desde la terminal**.

Creando un *gist* desde la terminal

```
# publica el fichero 'index.js' como un gist público
$ gh gist create --public index.js

# crea un gist privado con una descripción
$ gh gist create analytics.js -d "Ejemplo de cómo usar Analytics en tu \
sitio con JS"

# crea un gist que contenga diferentes ficheros
$ gh gist create index.js index.html styles.css

# crea un gist desde el standard input
$ gh gist create -

# crea un gist desde el output de otro comando
$ cat cool.txt | gh gist create
```

Más comandos para administrar *gists*

Una vez hemos creado nuestro *gist*, vamos a querer administrarlo de alguna forma. Ya sea verlo, editarlo o borrarlo definitivamente. Veamos una lista de los comandos disponibles para poder hacerlo:

```
# muestra el contenido del gist
$ gh gist show <id>

# edita el gist
$ gh gist edit <id>

# borra el gist
$ gh gist delete <id>

# borra el gist y todas sus versiones
$ gh gist delete --force <id>

# clona el gist a tu máquina local
$ gh gist clone <id>
```

Conclusiones del libro

¡Gracias por leer el libro! Espero que hayas aprendido algo nuevo. Mi objetivo era alejar los fantasmas y las magias negras que, parecen, tiene Git por detrás. Muchas veces, en programación, hacemos cosas por pura mecánica, sin entender qué hace por detrás y por qué.

A mi me gusta saber el por qué de las cosas. En libros, vídeos o tutoriales que veo siempre te dicen: *haz esto*. Vale, pero, **¿Por qué?** Sin entender el por qué... realmente perdemos parte de la información. Así que he intentado trasladar esa creencia al texto del libro.

También seguramente habrás notado que he querido adoptar los comandos y configuraciones más nuevos de Git. Por ejemplo, usar `git switch` y `git restore` en lugar de apoyarme sólo en `git checkout`. He recibido muchos comentarios sobre esto.

Lo primero `checkout` ha sido una anomalía histórica que **no sigue la filosofía UNIX** (que un comando haga una sola cosa y lo haga bien) ni el espíritu general de Git. No es que esté mal, claro, y va a seguir funcionando durante muchos años. Pero, especialmente la gente que empieza, se puede aprovechar de la claridad de los nuevos comandos.

Lo segundo es que quiero que el libro no te sirva sólo para el presente. También para el futuro. Y por eso he intentado acercarte al máximo posible a cómo puedes trabajar ya. Lo mismo pasa con todas las explicaciones sobre la rama principal con el nombre de `main`.

Muchas veces los tutoriales en Internet se quedan congelados con creencias antiguas, comandos clásicos y formas de hacer que han perdurado años simplemente por ser los primeros resultados de búsqueda en *Google*. Y eso **no los hacen ni peor ni mejor**, simplemente han quedado en una burbuja dónde las nuevas iteraciones no han sido capaces de hacerse un hueco.

También he querido deslizar mi opinión sobre estrategias, buenas prácticas y formas de trabajar. Están basadas en mi experiencia después de trabajar años con Git en proyectos reales. Pero **no siempre he pensado igual**. Con el tiempo he podido cambiar de opinión varias veces. Quién sabe si me pasa lo mismo en el futuro. No me da miedo cambiar de opinión ya que considero que es parte del aprendizaje. Así que, por supuesto, **te animo a que tengas el máximo espíritu crítico y tomes tus propias decisiones**.

En temas de Git un pilar muy importante, en mi opinión, además de las buenas prácticas y las estrategias, es la productividad. En el desarrollo de software vas a estar constantemente trabajando con Git así que cualquier cosa que puedas ahorrarte, siempre ayuda. Por eso le he dedicado también tanto tiempo a los *aliases*, *hooks* y la línea de comandos de *GitHub*. Básicamente, **me hubiera encantado que alguien me los hubiera explicado** a mí para no haber perdido tanto tiempo en el pasado.

Al final, **he querido darte el mejor libro para aprender Git en español basado en experiencias reales y actualizado a la última**. No sé si lo habré conseguido... Pero qué bien que me he quedado al terminarlo. Ahora, a por el siguiente. **Gracias por leerme**.