



BRAIS MOURE

@mouredev



Git & GitHub

desde cero

2ª edición



“GUÍA DE ESTUDIO TEÓRICO-PRÁCTICA
PASO A PASO MÁS CURSO EN VÍDEO”



Git y GitHub desde cero

Guía de estudio teórico-práctica paso a paso más curso en vídeo

Brais Moure

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

Esta versión se publicó en 2024-04-18 ISBN
979-83-9120-047-5



Primera edición: abril de 2023 Segunda edición: abril de 2024

Todos los derechos reservados. No se permite la reproducción total o parcial de esta obra, ni su incorporación a un sistema informático ni su transmisión en cualquier forma o por cualquier medio, sea éste electrónico, mecánico, por fotocopia, por grabación u otros métodos, sin el permiso previo y por escrito del autor. La infracción de los derechos mencionados puede ser constitutiva de delito contra la propiedad intelectual (Art. 270 y siguientes del Código Penal). El copyright estimula la creatividad, defiende la diversidad en el ámbito de las ideas y el conocimiento, promueve la libre expresión y favorece una cultura viva. Gracias por comprar una edición autorizada de este

libro y por respetar las leyes del copyright al no reproducir, escanear ni distribuir ninguna parte de esta obra por ningún medio sin permiso.

© 2023 - 2024 Brais Moure Morais

A la comunidad: Por apoyar mi trabajo día a día y convertirlo en el mejor del mundo.

Índice general

Hola, mundo	1
README (Léeme)	2
Guía de estudio más curso en vídeo	2
Imágenes	3
Comparte	3
Feedback o errores	4
Segunda edición	5
Un año	5
Contenido actualizado	5
Secciones	8
Comandos	8
Conceptos	8
Curso	9
Introducción	10
GIT	13
Capítulo 1: Introducción	14
Conceptos	14
Curso	16
Capítulo 2: Historia	20
Conceptos	20

ÍNDICE GENERAL

Curso	22
Capítulo 3: Instalación \$git	24
Comandos	24
Conceptos	24
Curso	27
Capítulo 4: Comandos básicos de la terminal . . .	31
Comandos	31
Conceptos	31
Curso	33
Capítulo 5: Configuración \$git config	37
Comandos	37
Conceptos	37
Curso	38
Capítulo 6: Inicialización de un repositorio \$git	
init	42
Comandos	42
Conceptos	42
Curso	44
Capítulo 7: Ramas	47
Comandos	47
Conceptos	47
Curso	49
Capítulo 8: Guardado \$git add y \$git commit . . .	51
Comandos	51
Conceptos	51
Curso	52
Capítulo 9: Estado \$git log y \$git status	58
Comandos	58
Conceptos	58
Curso	60

ÍNDICE GENERAL

Capítulo 10: Operaciones con ramas \$git	
checkout y \$git reset	63
Comandos	63
Conceptos	63
Curso	65
Capítulo 11: Alias \$git alias	69
Comandos	69
Conceptos	69
Capítulo 12: Ignorar ficheros .gitignore	73
Comandos	73
Conceptos	73
Curso	76
Capítulo 13: Comparación de commits \$git diff	79
Comandos	79
Conceptos	79
Curso	81
Capítulo 14: Desplazamientos en una rama	84
Comandos	84
Conceptos	84
Curso	85
Capítulo 15: Reset y log de referencias \$git	
reset --hard y \$git reflog	89
Comandos	89
Conceptos	89
Curso	91
Capítulo 16: Etiquetas \$git tag	95
Comandos	95
Conceptos	95
Curso	97

ÍNDICE GENERAL

Capítulo 17: Creación de ramas \$git branch y \$git switch 100
Comandos 100
Conceptos 100
Curso 103

Capítulo 18: Combinación de ramas \$git merge . 107
Comandos 107
Conceptos 107

Capítulo 19: Conflictos 112
Comandos 112
Conceptos 112
Curso 116

Capítulo 20: Cambios temporales \$git stash . . . 122
Comandos 122
Conceptos 122
Curso 124

Capítulo 21: Reintegración de ramas 127
Comandos 127
Conceptos 127
Curso 129

Capítulo 22: Eliminación de ramas 131
Comandos 131
Conceptos 131
Curso 133

GITHUB 135

Capítulo 23: Introducción a GitHub 136
Conceptos 136
Curso 140

Capítulo 24: Primeros pasos 144

ÍNDICE GENERAL

Conceptos	144
Curso	144
Capítulo 25: Repositorio personal	147
Conceptos	147
Curso	151
Capítulo 26: Local y remoto	159
Conceptos	159
Curso	160
Capítulo 27: Autenticación SSH	163
Conceptos	163
Curso	165
Capítulo 28: Repositorio proyecto	174
Conceptos	174
Curso	175
Capítulo 29: Git en remoto \$git remote	177
Comandos	177
Conceptos	177
Curso	179
Capítulo 30: Subida de un proyecto	183
Comandos	183
Conceptos	183
Curso	184
Capítulo 31: Sincronización remota \$git fetch y \$git pull	187
Comandos	187
Conceptos	187
Curso	188
Capítulo 32: Clonación \$git clone	192
Comandos	192

ÍNDICE GENERAL

Conceptos	192
Curso	193
Capítulo 33: Subida de código \$git push	196
Comandos	196
Conceptos	196
Curso	198
Capítulo 34: Bifurcaciones	200
Conceptos	200
Curso	201
Capítulo 35: Flujo colaborativo	206
Conceptos	206
Curso	206
Capítulo 36: Pull requests	209
Conceptos	209
Curso	211
Lección 37: Ejercicio práctico	214
Conceptos	214
Curso	215
Capítulo 38: Conflictos en pull requests	217
Conceptos	217
Curso	219
Capítulo 39: Sincronización de bifurcaciones . . .	225
Comandos	225
Conceptos	225
Curso	227
Capítulo 40: Markdown	230
Conceptos	230
Curso	233

ÍNDICE GENERAL

GIT y GITHUB 236

Capítulo 41: Herramientas gráficas 237

 Conceptos 237

 Curso 239

Capítulo 42: Git y GitHub flow 247

 Comandos 247

 Conceptos 247

 Curso 250

Capítulo 43: Ejemplo GitFlow 258

 Conceptos 258

 Introducción 258

 Curso 260

Capítulo 44: Otros comandos \$git cherry-pick y \$git rebase 267

 Comandos 267

 Conceptos 267

 Curso 271

Capítulo 45: GitHub Pages y Actions 274

 Conceptos 274

 Curso 277

Otros comandos 280

 Introducción 280

 Listado 280

Buenas prácticas 283

 Introducción 283

 Git 283

 GitHub 285

 Conclusión 286

Próximos pasos 288

¡Muchas gracias! 289

Hola, mundo

¡Hola, mundo! Mi nombre es **Brais Moure**, autor del libro. Soy ingeniero de software desde 2010, y GitHub Star desde 2023.

En 2015 creé *MoureDev*, para dedicarme al desarrollo de software de forma *freelance* y especializarme en la creación de aplicaciones móviles. He publicado más de 150 apps, superado millones de descargas y colaborado con empresas de diferentes partes del mundo.

En 2018 comienzo a compartir contenido gratuito sobre programación en diferentes redes sociales, utilizando también el nombre de **@mouredev**. Hoy en día nuestra comunidad, sumando todos esos canales, ha superado el millón y medio.

Actualmente, combino mi trabajo como programador y divulgador.

Este es mi primer libro, creado con todo mi cariño desde Galicia para el mundo. Espero que te resulte muy útil.

Recuerda que puedes encontrar todo mi contenido en moure.dev¹.



brais moure @mouredev

¹<https://moure.dev>

README (Léeme)

Guía de estudio más curso en vídeo

Este libro está pensado para facilitar el aprendizaje de las herramientas **Git** y **GitHub** desde cero y de manera independiente, pero principalmente funcionará de una manera más efectiva si se aplica como recurso complementario al curso práctico gratuito y en vídeo (de 5 horas duración) que tengo publicado en [YouTube](https://mouredev.com/git-github). Si, 100% gratuito. Solo tienes que entrar en mouredev.com/git-github¹.



Entonces ¿Por qué un libro?

Personalmente, porque creo que la mejor manera de asegurar nuestro aprendizaje es combinando esta guía

¹<https://mouredev.com/git-github>

de fácil comprensión y el curso en vídeo. La guía servirá para seguir el curso paso a paso, explicar cada lección, extender sus conceptos y aprender muchas cosas nuevas. También encontrarás apartados para destacar y ampliar las ideas más importantes, y podrás consultar cualquier duda rápidamente.

Una vez aclarado esto, tú decides si leer este libro puede servirte de ayuda.

Apoyar esta publicación me sirve para seguir creando contenido gratuito sobre programación y desarrollo de software día a día. ¡Muchas gracias!

Imágenes

Las imágenes del libro se corresponden con capturas de pantalla del curso en vídeo (actualizadas en esta segunda edición), para ayudarte a obtener una referencia temporal. No son un elemento esencial para el seguimiento de la guía. Puedes consultarlas en máxima resolución, a color, y ordenadas por lección, accediendo a mouredev.com/imagenes-libro-git².

Comparte

¿Quieres que más gente conozca este recurso? Nómbrame como *@mouredev* en redes sociales y cuéntale a todo el mundo qué te ha parecido.

²<https://mouredev.com/imagenes-libro-git>

También puedes dejar una reseña con tu opinión en la plataforma en la que hayas adquirido el libro.

Feedback o errores

Si encuentras algún error, o quieres darme *feedback*, no dudes en escribirme a braismoure@mouredev.com.

Agradezco enormemente tu colaboración.

Segunda edición

Un año

En el momento en el que escribo esta sección ha transcurrido un año desde la publicación del libro. Sólo puedo decir una cosa: **GRACIAS**. Ni en mis mejores sueños imaginé que la acogida iba a ser tan grande.

El libro ha vendido miles de copias, cientos de personas me han escrito mensajes de agradecimiento, habéis aparecido en eventos con él para que os lo firme y, un año después, sigue ocupando las primeras posiciones en el ranking de ventas de su categoría. Lo repito: **GRACIAS**, de corazón.

Durante este año también sucedió algo que me hace muy feliz, y me gustaría compartir contigo: GitHub me reconoció como *GitHub Star*. Un premio internacional otorgado a los desarrolladores más influyentes por sus aportes a la comunidad, y tú eres parte de él. Puedes encontrar más información en stars.github.com/profiles/mouredev¹.

Contenido actualizado

Aquí tienes un resumen de los cambios introducidos en esta segunda edición:

¹<https://stars.github.com/profiles/mouredev>

- Todas las imágenes de los capítulos de la guía se han generado de nuevo para mejorar su legibilidad. Si algo se ha señalado en repetidas ocasiones sobre la primera edición, es que las imágenes (correspondientes a las capturas de pantalla del curso) en la versión impresa no poseían la calidad suficiente. Si bien es cierto, quiero apuntar un par de cosas:
 - Lamentablemente, este es un libro autoeditado. Esto quiere decir, entre muchas cosas, que no tengo control sobre el proceso de impresión llevado a cabo por Amazon. Al no contar con una editorial, de hacerlo de otra forma, me resultaría imposible costear y coordinar los envíos. Lo siento mucho.
 - Repetir que, igual que se dice en el apartado anterior, las imágenes son referencias al curso en vídeo y, por lo tanto, no son esenciales para comprender el contenido del libro. Aún así, puedes consultar el vídeo (mouredev.com/git-github²) y la versión digital de las imágenes (mouredev.com/imagenes-libro-git³) en cualquier momento.
- El comando `git checkout HEAD` se ha modificado por `git checkout HEAD -- .` en el capítulo 14.
- Cada vez que se nombra en el capítulo 19 el comando `git merge --mine`, también se hará referencia a `--ours`. También se especifica que para la resolución de conflictos debe añadirse el nombre del archivo.

²<https://mouredev.com/git-github>

³<https://mouredev.com/imagenes-libro-git>

- Se ha ampliado el apartado correspondiente a **GitHub Actions** en el capítulo 45. Ahora podrás conocer también cómo funciona esta potente funcionalidad de GitHub.
- Antes del último apartado del libro, llamado “*Buenas prácticas*”, se ha añadido uno nuevo con “*Otros comandos*”.
- Por último, se han corregido pequeños errores ortográficos. Ten en cuenta que la numeración de las páginas ha cambiado.

No son grandes cambios, pero espero que ayuden a mejorar la experiencia de lectura. Muchas gracias por contribuir a lograrlo.

Secciones

El libro estará dividido principalmente en una sección dedicada a Git y otra a GitHub. Llegando a combinar ambas partes hacia el final de este. También podrás encontrar un capítulo completo dedicado a realizar un ejercicio práctico y colaborativo entre todos los participantes del curso.

Por otra parte, la guía cuenta con 45 capítulos diferentes, divididos en tres apartados (comandos, conceptos y curso) que se repetirán en cada uno de ellos.

Vamos a detallar a continuación el objetivo de cada apartado.

Comandos

Sección opcional (no aparecerá en todos los capítulos) que resumirá los comandos de Git que se utilizarán por primera vez en un capítulo del vídeo. Es una manera de asociar rápidamente las instrucciones de línea de comandos relacionadas con cada uno de los temas que vamos a tratar para aprender a trabajar con Git.

Conceptos

Sección que servirá para introducir el capítulo y tratar de forma teórica cada uno de los conceptos que

aprenderemos en él. Cada concepto dispondrá de su propio apartado individual.

Curso

Sección que aplicará de forma práctica los conceptos tratados en el apartado previo. Explicando cómo hacer uso de ellos en un supuesto real.

Este apartado está directamente relacionado con el curso el vídeo, por lo que iniciará siempre con un texto como el que sigue a continuación:

Introducción: mouredev.com/git-github¹

Inicio: 00:00:00 | Duración: 00:03:15

En él podrás visualizar el título de la lección, un enlace directo a dicha clase, su inicio, y la duración total del fragmento de vídeo. Prueba a acceder a esta primera *URL* del curso <https://mouredev.com/git-github>.

Este último apartado, en algún momento, puede resultar redundante con respecto al de conceptos. Lo considero necesario para poder explicar cada una de las ideas y favorecer su aprendizaje.

¹<https://mouredev.com/git-github>

Introducción

Trabajar con nuestro código de forma segura es tan importante como aprender a programar, por eso, herramientas como **Git** y **GitHub** son esenciales en el mundo del desarrollo de software.

Registrar el histórico de trabajo de nuestro código, generar copias de seguridad, y trabajar en equipo de forma rápida y sin errores. Estas son las principales características de Git.

Con este libro, y a través de 45 capítulos, aprenderemos desde cero y paso a paso todo lo necesario para trabajar con Git, el sistema de control de versiones por excelencia, y GitHub, la plataforma en la nube de código colaborativo.

Cuando me planteé la temática de un nuevo curso sobre programación, me di cuenta de que Git está presente en todo el sector. Sinceramente, no existe una tecnología que se utilice tanto como Git en el mundo del desarrollo de software, sin importar el lenguaje de programación o el entorno en el que trabajemos. Sin duda, es un estándar que debemos de conocer.

¿No te lo crees? Vamos a revisar algún dato:

Comencemos con la encuesta desarrolladores de *StackOverflow* (insights.stackoverflow.com/survey¹). Es la más importante del sector, y seguramente la mejor para representar las tendencias actuales. Pues bien, cuando se pregunta sobre el sistema de control de versiones

¹ <https://insights.stackoverflow.com/survey>

más utilizado, aquí tienes los resultados: De toda la gente que ha respondido la encuesta, más del 93% utiliza Git. Si revisamos los datos a nivel profesional, casi un 97%. Si en esta misma encuesta buscamos la sección de plataformas de control de versiones, observaremos algo muy parecido. GitHub es con diferencia la más utilizada. Un 87% a nivel personal y un 55% a nivel profesional.

¿Quieres más motivos? Aquí tienes:

Estos son los resultados de la encuesta de la propia GitHub, su llamado, *Octoverse* (octoverse.github.com²). En el último año ha alcanzado cifras históricas, y más del 90% de las empresas mejor valoradas del mundo utilizan esta plataforma para alojar su código.

Creo que estos son motivos que nos dejan muy clara la importancia de Git y de GitHub.

Pues bien, lo que vas a leer a continuación es el resultado de su importancia. Una guía desde cero y para principiantes, basada en mi curso en vídeo y utilizando cada lección para introducir, ejemplificar y ampliar conocimientos.

A lo largo del curso encontrarás un ejercicio para poner en práctica todo lo aprendido, pero antes de comenzar

me gustaría hacer un repaso a los cuatro recursos que tienes a tu disposición para comenzar tu aprendizaje e intentar que sea más ameno y cercano.

- En primer lugar, su sitio web (github.com/mouredev/hello-git³). Allí podrás encontrar las 45 clases y un *link* que te llevará a cada parte concreta del curso, donde podrás ver en vídeo cada uno de los

²<https://octoverse.github.com>

³<https://github.com/mouredev/hello-git>

conceptos. También encontrarás en este lugar toda la información del curso y enlaces relevantes.

- Por otra parte, también tienes a tu disposición el servidor de *Discord* de la comunidad (discord.gg/mouredev⁴). En él encontrarás un canal que se llama *git-github*, donde preguntar, compartir y charlar con la comunidad sobre estas tecnologías.
- También tienes mi canal de *Twitch* (twitch.tv/mouredev⁵), donde realizo directos de lunes a viernes, y donde se llevó a cabo este curso en vídeo.

Aquí va un poco de filosofía GitHub (aunque te explicaré más adelante qué es esta plataforma): Si quieres apoyar este recurso, simplemente puedes hacer *star* en el sitio web que te he compartido. Es una manera muy rápida y valiosa de apoyar este contenido.



Ya, por último, y para practicar, tienes la plataforma de retos de programación de la comunidad (retosdeprogramacion.com⁶). Un lugar donde, por un lado, practicar programación, y, por otro lado, gran parte de los conceptos que aprenderemos en el curso, ya que la forma que tenemos de compartir cada una de las soluciones a los retos de código será utilizando Git y GitHub.

Hecho esta introducción... ¡Bienvenido/a a **Git y GitHub desde cero!**

⁴<https://discord.gg/mouredev>

⁵<https://twitch.tv/mouredev>

⁶<https://retosdeprogramacion.com>

GIT

Sección dedicada al estudio de Git



Capítulo 1: Introducción

Conceptos

Introducción

Git es un sistema de control de versiones distribuido que se ha convertido en una herramienta esencial para la mayoría de los desarrolladores de software. A menudo, se confunde con **GitHub**, pero son dos conceptos diferentes. GitHub es una plataforma en línea que se utiliza para alojar proyectos que se gestionan mediante Git. Este libro se centrará en esta primera parte en Git, y en cómo utilizarlo para controlar las versiones de nuestro código. Contará con una segunda parte centrada en GitHub y en cómo usarlo junto a Git.

Web oficial

La página web oficial de Git es git-scm.com¹. Aquí podemos encontrar toda la información que necesitamos para empezar a trabajar con Git. La página ofrece una guía detallada sobre cómo utilizar Git, y una lista completa de comandos que podemos usar. También tiene publicado *online* un libro gratuito llamado **Pro Git**, que está disponible en varios idiomas, incluyendo español.

¹<https://git-scm.com>

Git y GitHub

Git es una herramienta de control de versiones distribuido, que nos permite a los desarrolladores trabajar en un proyecto sin necesidad de estar conectados a un servidor central. Por otro lado, GitHub es una plataforma en línea que se utiliza para alojar proyectos que se gestionan mediante Git. Aunque GitHub depende de Git, no son lo mismo.

Sistemas de control de versiones

Un **sistema de control de versiones** es una herramienta que se utiliza para mantener un registro de los cambios que se hacen en un proyecto. Permite a los desarrolladores trabajar en un proyecto de manera colaborativa, manteniendo un historial de cambios y documentando cada uno de ellos. De esta forma, se puede seguir el progreso del proyecto y volver a versiones anteriores si es necesario, entre muchas otras funcionalidades.

Ramas

Una **rama** es un *nuevo camino* asociado al código que se está desarrollando en un proyecto. Las *ramas* permiten a los desarrolladores trabajar en diferentes partes del proyecto al mismo tiempo, sin interferir en el trabajo de los demás. Una vez que completamos una tarea en una *rama*, podemos fusionarla con otra *rama* del proyecto y combinar sus cambios.

Conclusión

Git es una herramienta muy potente que nos permite controlar las versiones de nuestro código de manera eficiente y colaborativa. La página web oficial de Git es una gran fuente de información para aprender a utilizar Git y para resolver cualquier duda que podamos tener. Además, GitHub nos ofrece una plataforma para alojar nuestros proyectos y colaborar con otros desarrolladores. En resumen, si nos interesa el sector del desarrollo de software, es imprescindible conocer y saber utilizar Git.

Curso

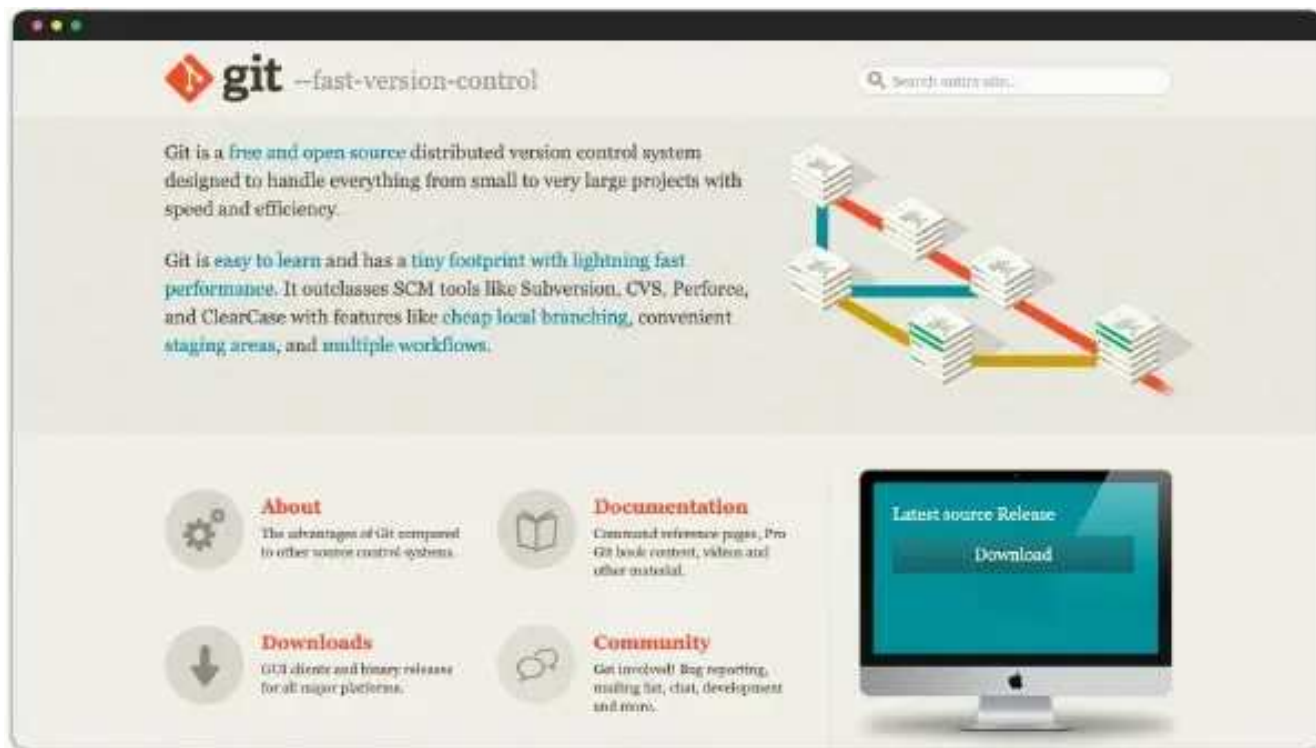
Lección 1: mouredev.com/git-github-01²

Inicio: 00:03:15 | Duración: 00:07:06

Comencemos hablando del primer sitio importante que debemos conocer: la página web oficial de Git. Así, iremos entendiendo poco a poco de qué trata esta herramienta. Dicha web será git-scm.com,³ y aquí podremos encontrar prácticamente todo sobre Git. Hay que dejar claro que existen dos conceptos: por un lado, Git, y por otro, GitHub. Por ahora, solo hablaremos de Git, sin confundirlo con GitHub. Más adelante, comenzaremos la sección dedicada a GitHub.

²<https://mouredev.com/git-github-01>

³<https://git-scm.com>,



En primer lugar, entendamos que Git es independiente de GitHub, aunque GitHub sí dependa de Git. Git es de código abierto, y todo su código, para que nos hagamos una idea, está en GitHub. Todo el código de Git es libre y se puede leer desde GitHub, que es una plataforma donde se aloja código fuente.

Git es un sistema de control de versiones distribuido muy importante. Existen diferentes sistemas de control de versiones, pero no todos son distribuidos. *¿Qué significa esto?* Que no depende de un único sitio. Si ese sitio se borra o falla, el código podría perderse, pero con Git, al ser distribuido, podemos tener una copia del código en cada equipo de las personas que trabajan en un proyecto. Si el servidor central falla, podríamos recuperar dicho trabajo al tener almacenado de manera *local* parte del código y el historial de cambios.

Bien, entendido el concepto de distribuido, ¿qué es un sistema de control de versiones? Si trabajas en programación, o incluso si estás empezando, quizás has visto bromas asociadas a proyectos con nombres como *proyecto-final-version-2* o *proyecto-final-version-2-final-*

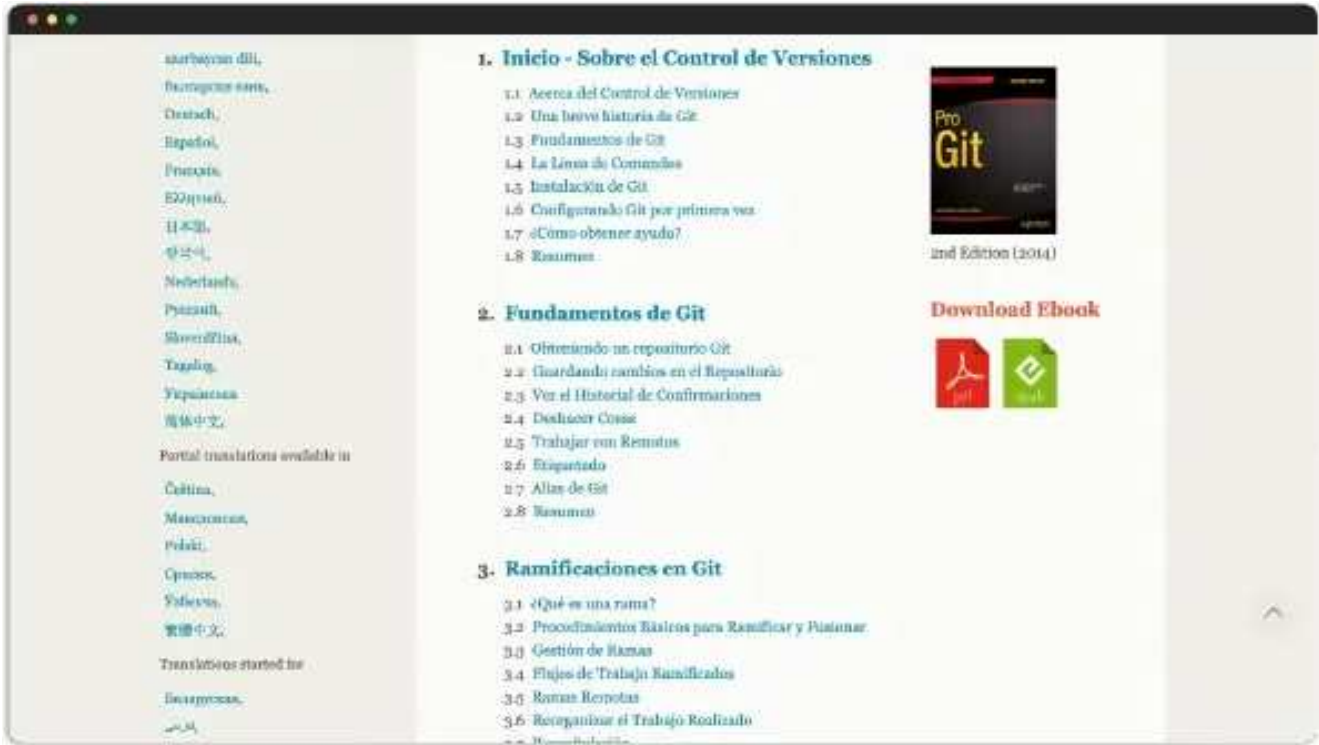
superfinal. Eso sucede cuando no trabajamos con un sistema de control de versiones, lo que nos puede llevar a perder información, cometer errores o borrar nuestros avances.

Un sistema de control de versiones nos permite llevar un registro de todo el historial de un proyecto, documentando y trazando cada uno de esos cambios. Podemos navegar por ese historial como si fueran los mensajes de WhatsApp, yendo hacia atrás o hacia adelante, borrando o saltando entre conversaciones. Iremos entendiendo todo esto poco a poco, no te preocupes.

También acabaremos comprendiendo los conceptos principales asociados a las *ramas*, y cómo nos moveremos entre ellas según nuestro proyecto evoluciona. Así que, resumiendo, tengamos siempre presente la web oficial, es un gran recurso.

Te contaré un secreto. La web de Git es uno de los mejores lugares para aprender Git. Contiene mucha documentación y un libro gratuito llamado *Pro Git*, en español. Traducido por la comunidad y validado por la gente de

Git. Se puede comprar, pero también está disponible gratis. Eso sí, se trata de un texto mucho más académico que el libro que te encuentras leyendo.



Cuando lleguemos a la parte de GitHub, también descubrirás dónde encontrar toda la documentación de GitHub.

Capítulo 2: Historia

Conceptos

Introducción

El control de versiones es fundamental en el mundo del desarrollo de software. Cuando varios desarrolladores participan en un mismo proyecto, es necesario coordinar sus esfuerzos y asegurarnos de que nos encontramos trabajando sobre una versión coherente del código. Además, es importante poseer un historial de cambios para poder volver atrás, en caso de cometer errores, o para recuperar una versión anterior que funcione correctamente.

Git es uno de los sistemas de control de versiones más populares y utilizados en la actualidad. En este libro, profundizaremos en su funcionamiento, configuración y uso, para sacarle el máximo partido a esta herramienta. Pero antes, un poco de historia.

El origen

Git fue creado por **Linus Torvalds**, el creador del *kernel* de *Linux*. En sus inicios, *Linus* utilizaba otro sistema de control de versiones llamado *BitKeeper*, para administrar el desarrollo de *Linux*. Sin embargo, en 2005, se produjo una disputa con la empresa propietaria de *BitKeeper*, que

llevó a que la comunidad de desarrollo de *Linux* perdiera el acceso a esta herramienta.

Ante esta situación, *Linus* decidió crear su propio sistema de control de versiones que pudiera cumplir con las necesidades del desarrollo de *Linux*. Así nació Git, que en un principio fue utilizado exclusivamente por la comunidad de desarrollo de *Linux*, pero que rápidamente se extendió a otros proyectos y empresas.

Las ventajas

Una de las principales ventajas de Git es que es un sistema distribuido. Esto significa que, cada desarrollador tiene una copia completa del *repositorio* en su máquina, lo que permite trabajar sin conexión a internet y facilita la colaboración en equipos remotos.

Otra ventaja de Git es que es muy eficiente en la gestión de modificaciones y *ramas*. Git permite crear *ramas* de forma muy sencilla, lo que facilita la incorporación de nuevas funcionalidades y la corrección de errores. Además, Git tiene herramientas para comparar versiones y *fusionar* esas *ramas*, lo que hace que el proceso de integración de cambios sea mucho más fácil y seguro.

Git también es muy flexible en cuanto a la forma de trabajar. Nos permite desarrollar empleando distintos flujos para gestionar nuestro código, desde los más sencillos hasta los más complejos. Esto hace que sea muy versátil y se adapte a las necesidades de cada proyecto.

Por último, Git es una herramienta *open source*, lo que significa que es gratuita y cuenta con el respaldo de una gran comunidad de desarrolladores y usuarios que

constantemente están mejorando y actualizando la herramienta.

Conclusión

Hemos dado contexto a Git, su origen y sus ventajas. Es importante entender que Git es una herramienta que evoluciona constantemente y que sigue siendo fundamental en el mundo del desarrollo de software.

En los próximos capítulos profundizaremos en el funcionamiento de Git, desde la instalación y configuración, hasta el uso de las distintas funcionalidades para gestionar el control de versiones de nuestros proyectos.

Curso

Lección 2: mouredev.com/git-github-02¹

Inicio: 00:10:21 | Duración: 00:04:14

Antes de introducirnos en la descarga y configuración

Git, es importante dar contexto sobre este sistema de control de versiones. Git es una herramienta que se usa principalmente en el sector del desarrollo de software, estando presente en él desde hace bastante tiempo. Apareció el 7 de abril de 2005, y cuenta con el respaldo de toda la comunidad open source.

Git sigue evolucionando constantemente, y no es una tecnología en absoluto estancada. De hecho, es muy posible que se haya lanzado una nueva versión

¹<https://mouredev.com/git-github-02>

recientemente (y seguramente no importa en qué momento estés leyendo este libro). Es parte fundamental en el día a día del mundo del desarrollo de software.

Ahora bien, es interesante que conozcamos quién es el creador de Git. Se trata de *Linus Torvalds*, una figura clave en el mundo del software, que, si me permites, tenemos la obligación de conocer. *Linus* también es el creador del *kernel* de *Linux*, y mientras trabajaba en él, se dio cuenta de que los sistemas de control de versiones existentes no cumplían sus expectativas y necesidades.



Por ello, y por algún otro motivo que comenté al principio del capítulo, decidió crear su propio sistema de control de versiones, y así nació Git. Hoy en día, prácticamente todo el mundo en el sector utiliza esta herramienta.

Git es un sistema de control de versiones con años de trayectoria, que sigue evolucionando y cuenta con el respaldo de la comunidad de desarrollo. *Linus Torvalds*, su creador, es una figura icónica en el mundo del software, y es importante conocer su contribución a través de Git y el *kernel* de *Linux*.

Capítulo 3: Instalación

Comandos

- 1 `git`
- 2 `git --version`
- 3 `git -v`
- 4 `git -h`

Conceptos

Introducción

Antes de empezar a utilizar Git, necesitamos configurarlo correctamente en nuestro sistema. A continuación, vamos a detallar cómo instalar Git en diferentes sistemas operativos.

Instalación en Windows

Si usamos **Windows**, la forma más fácil de instalar Git es a través de la página de descargas de Git para *Windows*:

git-scm.com/download/win¹

¹<https://git-scm.com/download/win>

Descargaremos el archivo de instalación y seguiremos las instrucciones. Durante la instalación, se instalará automáticamente una terminal compatible con Git, llamada **Git Bash**.

Instalación en macOS

Si usamos **macOS**, podemos instalar Git a través de *Homebrew*, un gestor de paquetes popular para equipos *Apple*:

git-scm.com/download/mac²

Para instalar Git desde *Homebrew*, abrimos la terminal y escribimos lo siguiente:

```
1 brew install git
```

Instalación en Linux/Unix

La mayoría de las distribuciones de **Linux** ya incluyen Git en sus repositorios:

3

git-scm.com/download/linux

Por ejemplo, para instalar Git en *Ubuntu* o *Debian*, abrimos la terminal y escribimos:

```
1 sudo apt-get install git
```

En caso de *Fedora*, abrimos la terminal y escribimos:

²<https://git-scm.com/download/mac>

³<https://git-scm.com/download/linux>

```
1 sudo dnf install git
```

Verificación de la instalación

Una vez que hayamos instalado Git, podemos verificar si funciona correctamente escribiendo el comando `git` en la terminal. Si visualizamos una lista de posibles comandos, significa que está instalado correctamente. También podemos comprobar la versión instalada de Git con `git --version` o `git -v`.

Si no sabemos cómo utilizar Git, o cualquier herramienta desde la terminal, probaremos a escribir el comando seguido de `-h`, para obtener ayuda. Por ejemplo, `git -h` nos mostrará una lista de comandos y opciones disponibles.

Uso básico

Una vez instalado Git correctamente, es hora de empezar a utilizarlo. Podremos hacerlo desde la terminal o distintas herramientas gráficas (*GUIs*) como *GitHub Desktop*, *GitKraken*, *Sourcetree* o *Fork*.

Además de las herramientas mencionadas anteriormente, también podemos integrar Git en nuestro flujo de trabajo diario utilizando editores de código o *IDEs*, como *Visual Studio Code* o *IntelliJ IDEA*, entre otros.

Estos editores e *IDEs* ofrecen extensiones y complementos para integrar Git directamente en el entorno de desarrollo. Con estas extensiones podemos llevar a cabo flujos de trabajo completos en Git.

Una vez conozcamos el uso de Git, dependerá de nosotros seleccionar las herramientas que mejor se adapten a nuestra manera de trabajar.

Conclusión

Git resulta muy simple de instalar. Siendo su uso igual entre sistemas operativos, independientemente de en el que nos encontremos.

Comenzaremos a usar Git desde la terminal, para así entender los fundamentos del sistema de control de versiones y generar unas bases más sólidas de conocimiento.

Curso

Lección 3: mouredev.com/git-github-03⁴

Inicio: 00:14:35 | Duración: 00:09:25

Vamos a abordar la configuración de Git y su instalación en distintos sistemas operativos. Como ya hemos comentado anteriormente, Git fue creado por *Linus Torvalds* y es ampliamente utilizado por empresas como *Google*, *Microsoft* o *Netflix*, entre otra infinidad de compañías.

Si usamos *Linux* o *macOS*, es probable que ya tengamos Git instalado por defecto. Sin embargo, puede que la versión no esté actualizada. No debemos preocuparnos, ya que, para llevar a cabo este curso, es muy posible que la versión que tengamos instalada ya sea suficiente.

⁴<https://mouredev.com/git-github-03>

Vamos a cubrir temas fundamentales que funcionan en casi cualquier versión de Git.

En la documentación oficial de Git (git-scm.com/downloads⁵), encontraremos instrucciones detalladas sobre cómo instalarlo en nuestro sistema operativo. Simplemente tenemos que dirigirnos a la sección de descargas y seleccionar nuestro sistema operativo.

En mi caso, y durante este curso, estaré usando *macOS*, pero también existen opciones para *Windows* y otros sistemas *Unix* como *Linux*.

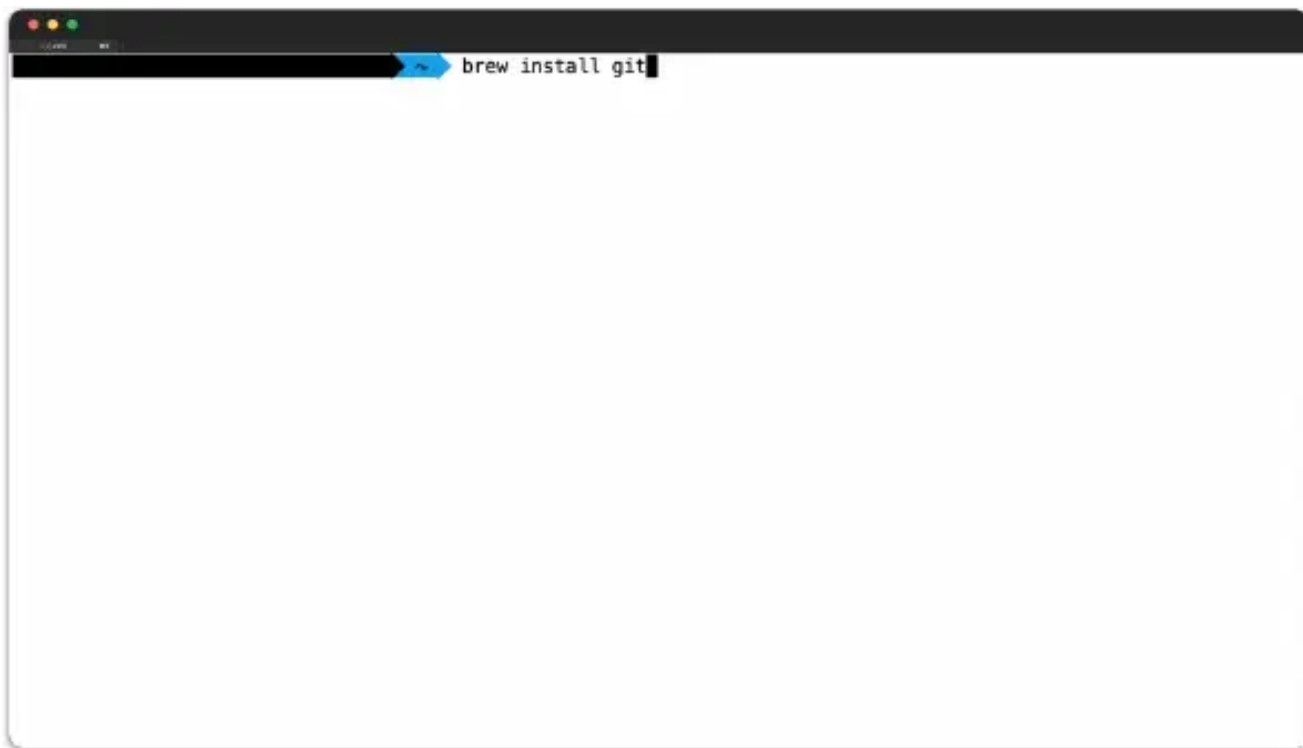
Una vez tengamos Git instalado, podremos trabajar con él desde la terminal del sistema, o mediante herramientas gráficas. Aunque es posible utilizar herramientas gráficas, inicialmente aprenderemos Git desde la terminal, ya que nos ayudará a comprender cómo funciona realmente.

Existen diferentes clientes gráficos disponibles para *Windows*, *macOS*, *Linux*, *Android* e *iOS*. Durante el curso, también descubriremos cómo utilizar Git desde editores de código o *IDEs*, *GitHub Desktop*, *GitKraken*, *Sourcetree* o *Fork*, entre otros, pero siente la libertad de explorar y elegir la herramienta que más te guste.

Hablemos de la instalación paso a paso. Si nunca hemos usado una terminal, también aprenderemos a utilizarla. Dependiendo del sistema operativo que estemos utilizando, necesitaremos una terminal compatible con Git, como *Git Bash* en *Windows*. Al instalar Git en *Windows*, también se instalará dicha terminal *Bash* automáticamente.

⁵<https://git-scm.com/downloads>

En *macOS*, podemos instalar Git usando *Homebrew*, un gestor de paquetes. Simplemente escribimos `brew install git` en la terminal y seguimos las instrucciones. Si ya lo tenemos instalado, la terminal nos lo indicará.



Una vez instalado Git, podemos verificar si funciona correctamente escribiendo el comando `git` en la consola. Si visualizamos una lista de comandos, significará que está instalado correctamente. También podemos comprobar la versión de Git utilizando los comandos `git --version` o `git -v`.

Si no sabemos cómo utilizar Git, o cualquier herramienta desde la terminal, podemos probar a escribir el comando seguido de `-h` para obtener ayuda. Por ejemplo, `git -h` nos mostrará una lista de comandos y opciones disponibles.

```

bisect    Use binary search to find the commit that introduced a bug
diff      Show changes between commits, commit and working tree, etc
grep      Print lines matching a pattern
log        Show commit logs
show      Show various types of objects
status    Show the working tree status

grow, mark and tweak your common history
branch    List, create, or delete branches
commit    Record changes to the repository
merge     Join two or more development histories together
rebase    Reapply commits on top of another base tip
reset     Reset current HEAD to the specified state
switch    Switch branches
tag        Create, list, delete or verify a tag object signed with GPG

collaborate (see also: git help workflows)
fetch     Download objects and refs from another repository
pull      Fetch from and integrate with another repository or a local branch
push      Update remote refs along with associated objects

'git help -a' and 'git help -g' list available subcommands and some
concept guides. See 'git help <command>' or 'git help <concept>'
to read about a specific subcommand or concept.
See 'git help git' for an overview of the system.
git --version
git version 2.44.0
git -v
git version 2.44.0
git

```

Recordemos que Git funciona de la misma manera en ~~Windows, Linux y macOS~~ *Windows, Linux y macOS*. Todos los comandos son iguales y se comportan de la misma forma en los distintos sistemas operativos.

Capítulo 4: Comandos básicos de la terminal

Comandos

- 1 `ls`
- 2 `cd <directorio>`
- 3 `cd ..`
- 4 `pwd`
- 5 `mkdir <nombre>`
- 6 `touch <nombre>`
- 7 `rm <nombre>`
- 8 `cp <nombre> <directorio>`
- 9 `mv <nombre> <directorio>`

Conceptos

Introducción

La **terminal**, **consola** o **línea de comandos**, es una herramienta muy potente que nos permite interactuar con nuestro sistema operativo de manera directa a través de instrucciones en formato texto. Aunque pueda parecer intimidante al principio, es importante comprender algunos conceptos básicos para poder utilizarla eficientemente.

En la terminal, como hemos comentado, todo se maneja mediante comandos de texto. Los comandos se escriben en una línea de texto y se ejecutan al presionar la tecla *Enter*. La terminal responde, de la misma manera, también con texto, ya sea una respuesta directa al comando o algún mensaje informativo o de error.

Comandos más importantes

Este es un listado con los comandos que más utilizamos habitualmente en *Bash*:

- **ls**: Muestra una lista con los archivos y carpetas del directorio actual.
- **cd**: Nos permite movernos por los diferentes directorios del sistema de archivos. Por ejemplo, `cd Desktop` nos lleva al directorio del Escritorio, y `cd ..` sube un nivel (retrocede) en el sistema de directorios.
- **pwd**: Muestra la ruta completa del directorio actual en el sistema de archivos.
- **mkdir**: Crea una nueva carpeta en el directorio actual. Por ejemplo, `mkdir "Hello Git"` crea una carpeta llamada *Hello Git*.
- **touch**: Crea un nuevo archivo vacío en el directorio actual. Por ejemplo, `touch hello_git.txt` crea un archivo llamado *hello_git.txt*.
- **rm**: Elimina un archivo o carpeta del directorio actual. Por ejemplo, `rm hello_git.txt` elimina el archivo *hello_git.txt*.
- **cp**: Copia un archivo de un lugar a otro. Por ejemplo, `cp hello_git.txt /Desktop/Hello \Git` copia el archivo *hello_git.txt* al directorio *Desktop/Hello Git*.

- **mv**: Mueve un archivo de un lugar a otro. Por ejemplo, `mv hello_git.txt /Desktop/Hello \Git` mueve el archivo *hello_git.txt* al directorio *Desktop/Hello Git*.

Estos son solo algunos ejemplos de los comandos que podemos utilizar en la terminal. Es importante señalar que, entre sistemas operativos y entornos de ejecución, se pueden utilizar comandos diferentes, por lo que es posible que debamos consultar su documentación para conocer los comandos específicos, y así poder llevar a cabo ciertas tareas.

Conclusión

En este capítulo hemos aprendido los conceptos básicos de la terminal y los comandos principales que podemos utilizar para interactuar con nuestro sistema de archivos.

Con una buena comprensión de los conceptos básicos y una práctica constante, podemos sacarle el máximo provecho a la terminal y mejorar significativamente nuestro flujo de trabajo como desarrolladores de software.

Curso

Lección 4: mouredev.com/git-github-04¹

Inicio: 00:24:00 | Duración: 00:06:13

¹<https://mouredev.com/git-github-04>

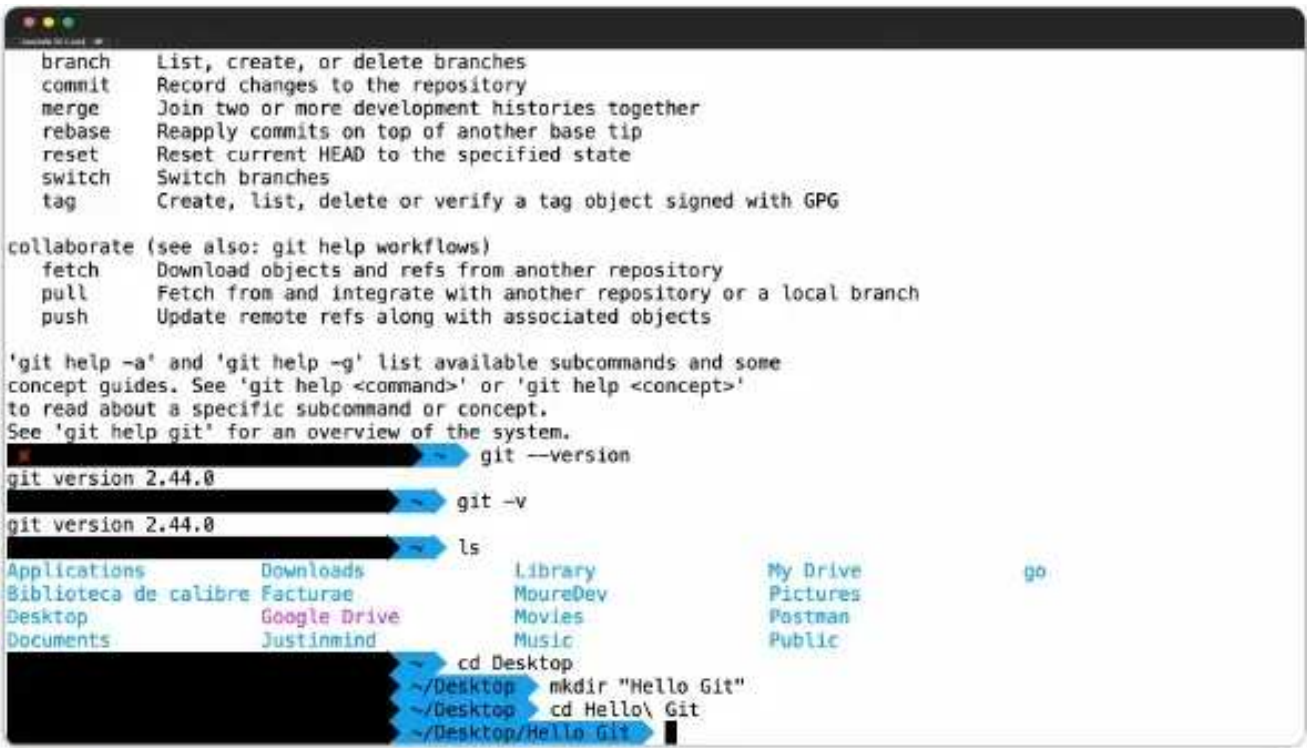
Es hora de comprender algunos conceptos básicos de la terminal, ya que quizás no sepamos cómo interactuar con ella. Para esto, dedicaremos esta clase a aprender al menos los comandos principales de la consola. Si esta ha sido la primera vez que hemos abierto una terminal, es el momento de aprender cómo usarla.

Git es un sistema de control de versiones que funciona especialmente bien con código. Está diseñado para manejar un gran volumen de archivos utilizados durante el proceso de desarrollo de software. Funciona mucho mejor con código que, por ejemplo, si lo que queremos hacer es una copia de seguridad de nuestro propio ordenador, o de grandes ficheros multimedia.

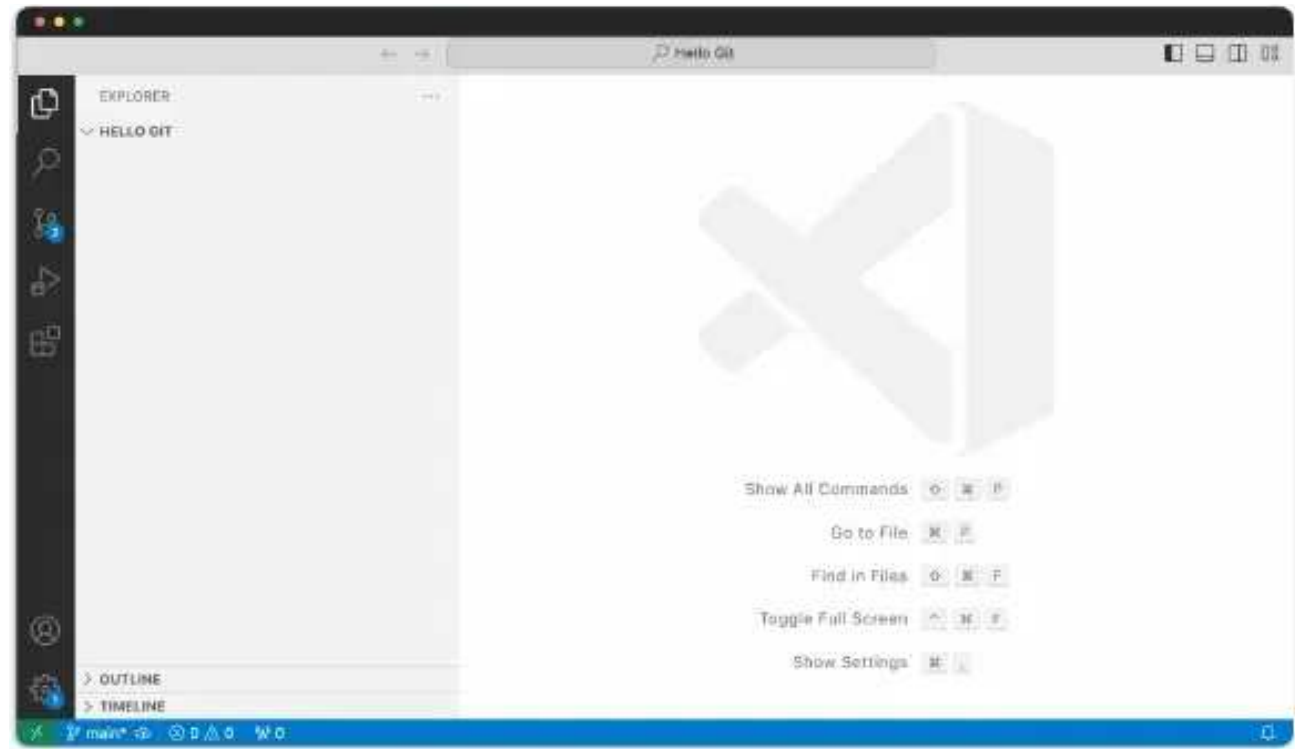
Vamos a hablar de los comandos principales que podemos manejar en una terminal. Por ejemplo, si escribimos `ls`, lo primero que veremos es un listado de todos los directorios que tenemos en ese punto concreto de nuestro sistema. Podemos movernos por los sistemas de archivos con el comando `cd`. Por ejemplo, si queremos ir al escritorio, escribiremos `cd Desktop`, pudiendo presionar la tecla de tabulación para autocompletar posibles destinos.

Si queremos conocer cuál es la ruta en la que nos encontramos, podemos escribir `pwd`.

Para comenzar a trabajar con Git, primero necesitamos crear una carpeta. El comando para crear una carpeta es `mkdir`. Imagínate que queremos crear un directorio llamado *Hello Git*, pues para ello lanzaremos el comando `mkdir "Hello Git"`. Para desplazarnos dentro de la carpeta, podemos usar el comando `cd Hello\ Git/`.



Aunque estemos usando la terminal, es también habitual trabajar dentro de un editor de código o del sistema de archivos del sistema. Por ejemplo, podemos abrir **Visual Studio Code**, el *IDE* que yo usaré durante el curso, con el comando `code .` (puede que tengas que configurar este acceso directo al editor). A partir de aquí, podemos empezar a crear archivos o directorios y trabajar en el proyecto de software que deseemos.



Recordemos que esto es solo una introducción a Git

y la terminal. Hay muchos más comandos y conceptos que aprender para sacarle el máximo provecho a estas herramientas.

Capítulo 5: Configuración

Comandos

- 1 `git config`
- 2 `git config --global user.name <nombre>`
- 3 `git config --global user.email <email>`

Conceptos

Introducción

Para comenzar a trabajar con Git necesitaremos realizar una pequeña configuración inicial. Una manera de identificar nuestras interacciones dentro del sistema.

Identificación

Una de las características clave de Git es que todas las acciones que se realizan en el sistema deben estar asociadas a un **autor**. Esto es importante, ya que nos permite a los desarrolladores rastrear quién hizo qué cambio en el código. Si un error o un problema surge en el proyecto, podremos revisar el historial de cambios y encontrar quién hizo el cambio y qué causó el problema.

Al trabajar con Git, cada usuario debe poseer su propio identificador, que se corresponderá con su **nombre** y dirección de **correo electrónico**. Estos identificadores se utilizan para etiquetar cada cambio realizado en el proyecto, lo que permite una fácil identificación de cada acción registrada en el sistema.

Configuración inicial

Antes de comenzar a trabajar con Git, es necesario realizar una configuración inicial que incluye la asignación de un nombre de usuario y una dirección de correo electrónico.

La configuración inicial de Git se realiza a través del comando `git config`. Si lo ejecutamos junto con el argumento (o *flag*) `--global`, estableceremos la configuración de Git a nivel global, lo que significará que dicha configuración se aplicará a todas las interacciones con Git desde nuestra sesión de usuario en el equipo.

La configuración de Git se realiza mediante dos propiedades: `user.name` y `user.email`. Es importante que ambos valores se configuren correctamente para que el sistema de control de versiones funcione según lo esperado.

Curso

Lección 5: mouredev.com/git-github-05¹

Inicio: 00:30:13 | Duración: 00:06:34

¹<https://mouredev.com/git-github-05>

Vamos a empezar trabajar en el proyecto de ejemplo (que hemos creado como *Hello Git* en la lección anterior), para ir entendiendo poco a poco qué es Git. A medida que avancemos en él, iremos explicando en paralelo los distintos conceptos de Git.

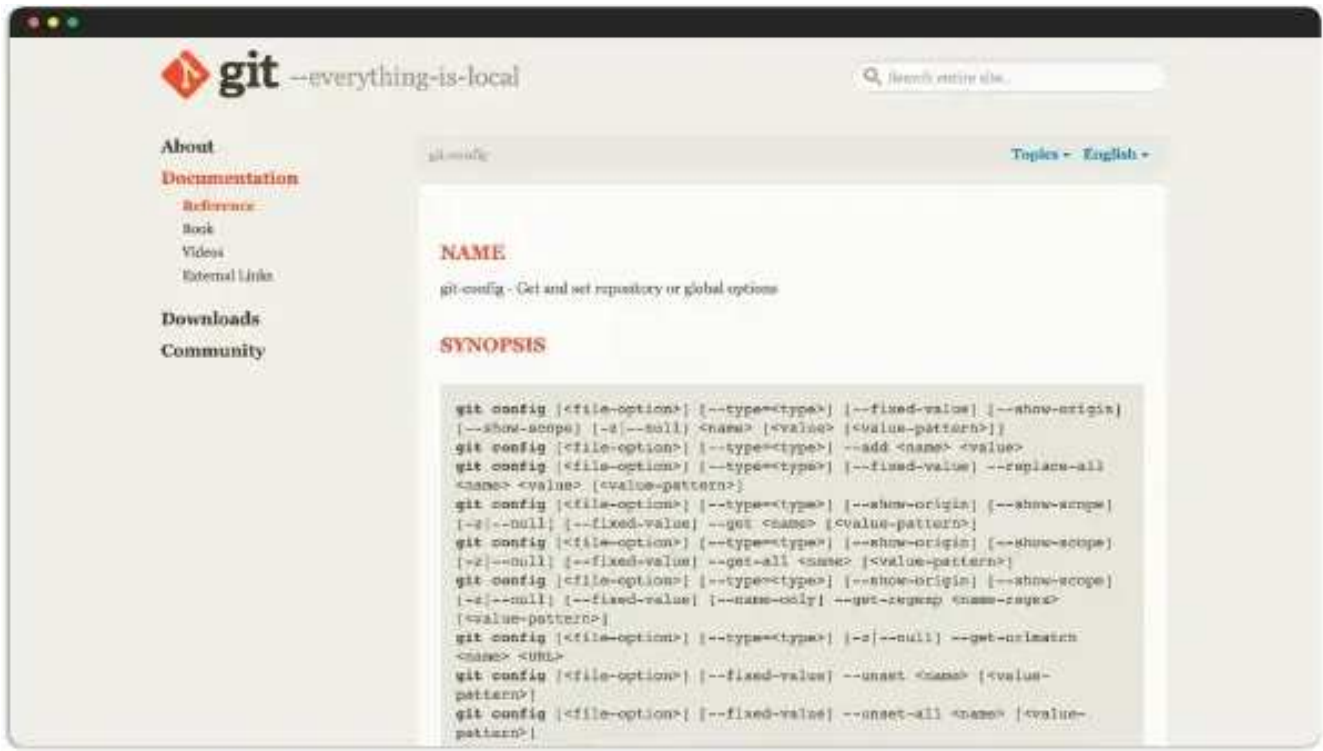
Siempre que trabajamos con Git, todo lo que hagamos tiene que estar asociado a alguien. ¿Por qué? Porque podríamos tener un proyecto en el que trabajamos solo nosotros, o un proyecto en el que trabajan más personas desarrollando código. Git nos ayudará a que tú puedas trabajar en tu ordenador con un código, yo en mi ordenador con otro código, y, llegado el caso, podamos unir ambos desarrollos sin problemas. Si durante ese proceso de combinación encontramos un error, podremos ver qué cambios ha hecho una persona u otra, qué modificaciones en el código han provocado un conflicto, volver a versiones anteriores, etc.

Como decíamos, Git nos obliga a tener asociado siempre un autor, un identificador a quien atribuirle todas las acciones que hagamos desde Git. Lo primero que vamos a hacer es configurar Git desde cero. Para ello, necesitamos tener un nombre de usuario y un email (los que tú quieras). Esto es esencial, un requisito obligatorio. El propio sistema nos obligará a definirlos en el momento que queramos hacer algo en Git por primera vez.

Para limpiar la consola, simplemente podemos escribir `clear`. La configuración de Git es muy amplia, si queremos profundizar, lo mejor es consultar la documentación oficial, donde encontraremos todo lo que se puede hacer con él:

git-scm.com/doc²

²<https://git-scm.com/doc>



Git es un sistema enorme, pero entender las bases no es algo que lleve muchísimo tiempo. Aunque existan un gran número de combinaciones de comandos, con más o menos una decena podemos llevar a cabo las acciones más comunes con las que trabajaremos día a día desde Git.

Para establecer la configuración de Git, tendremos que escribir el comando `git config`. A continuación, lo que tendremos que hacer es decidir a qué nivel vamos a establecer esta configuración. Lo habitual, y sobre todo cuando estamos empezando, es establecerla a nivel

global. Para ello, escribimos el parámetro `--global`. Esto significa que la configuración se aplicará a nivel global a la hora de trabajar con Git dentro de nuestro equipo (nuestro ordenador, no me refiero a un equipo de personas) y sesión de usuario. La configuración no suele ser algo concreto para un proyecto, al contrario, se suele usar de forma general para cualquier interacción que realicemos con Git, independientemente del proyecto. Si establecemos que la configuración es global, va a afectar a todo lo que se haga desde Git en la sesión de usuario de nuestra máquina.

Para configurar nuestro usuario escribimos `git config --global user.name`, y entre comillas ponemos el nombre de nuestro usuario. Una vez hecho esto, pulsamos *Enter*. Ahora, de la misma manera que hemos establecido el *name*, configuraremos la segunda y última propiedad obligatoria, el *email*. Establecemos la propiedad `git config --global user.email`. Lo ejecutamos, y este fichero de configuración se actualizará con nombre y email.

A screenshot of a terminal window with a dark background. The prompt is '~/.Desktop/Hello Git'. The first command entered is 'git config --global user.name "MoureDev"' and the second is 'git config --global user.email "braismoure@gmail.com"'. Both commands are highlighted in blue. The terminal shows the output of the first command as 'user.name' and the output of the second as 'user.email'.

```
~/.Desktop/Hello Git git config --global user.name "MoureDev"
~/.Desktop/Hello Git git config --global user.email "braismoure@gmail.com"
```

Esto es lo único que necesitamos para configurar inicialmente Git. Sin estas dos variables, sin el nombre y el email, no podremos ni comenzar a trabajar con el sistema de control de versiones.

Capítulo 6: Inicialización de un repositorio \$git init

Comandos

```
1 git init
```

Conceptos

Introducción

Configurado Git, es hora de preparar nuestro proyecto para empezar a trabajar con el sistema de control de versiones.

Inicialización

Antes de profundizar en las herramientas y características de Git, es importante entender los conceptos básicos del sistema de control de versiones. Git funciona mediante la creación de **fotografías** o **instantáneas** (llamadas **commits**) del estado de nuestro proyecto en diferentes momentos a lo largo del tiempo.

Cada `commit` representa un conjunto de cambios realizados en él.

Para trabajar con Git, primero debemos inicializar un **repositorio** en nuestro proyecto. Crearemos un nuevo *repositorio* lanzando el comando `git init` desde la consola, desde la carpeta raíz del proyecto. Al ejecutar este comando, Git creará una carpeta oculta llamada **.git**, que contendrá todas las referencias asociadas al sistema de control de versiones.

Una vez inicializado el *repositorio*, podemos comenzar a lanzar distintos comandos en ese directorio para ejecutar acciones propias del contexto de Git. Es un paso obligatorio para que el sistema reconozca ese directorio como un lugar en el que Git está operativo.

Repositorio

El término *repositorio* de Git hace referencia al lugar donde se almacena el historial de cambios realizados en un proyecto, así como las diferentes versiones del mismo. Se podría decir que es una base de datos que guarda la evolución de todo el proyecto a lo largo del tiempo, incluyendo el código fuente, la documentación y cualquier otro archivo que se encuentre en ese directorio. En un *repositorio* de Git, se registran las diferentes versiones de los archivos, y se realiza un seguimiento de los cambios que se han llevado a cabo. Cada versión de un archivo se almacena asociado a un `commit`, que contiene una *instantánea* de los cambios realizados en ese momento. De esta manera podemos consultar cómo ha evolucionado el proyecto a lo largo del tiempo.

Los *repositorios* de Git pueden ser **locales** (almacenados

en el equipo *local* del desarrollador) o **remotos** (almacenados en un servidor en la nube). Los *repositorios remotos* nos resultan útiles para colaborar con otros desarrolladores en un mismo proyecto, ya que permiten compartir los cambios realizados y fusionarlos en una única versión de este. También nos sirven para trabajar de forma segura y tener en todo momento un respaldo de nuestro proyecto en la nube.

Es posible que alguno de estos términos te resulte desconocido. No te preocupes, ya hablaremos al detalle de cada uno de ellos.

Curso

Lección 6: mouredev.com/git-github-06¹

Inicio: 00:36:47 | Duración: 00:05:36

Comencemos a aprender los conceptos principales con los que tenemos que trabajar en Git.

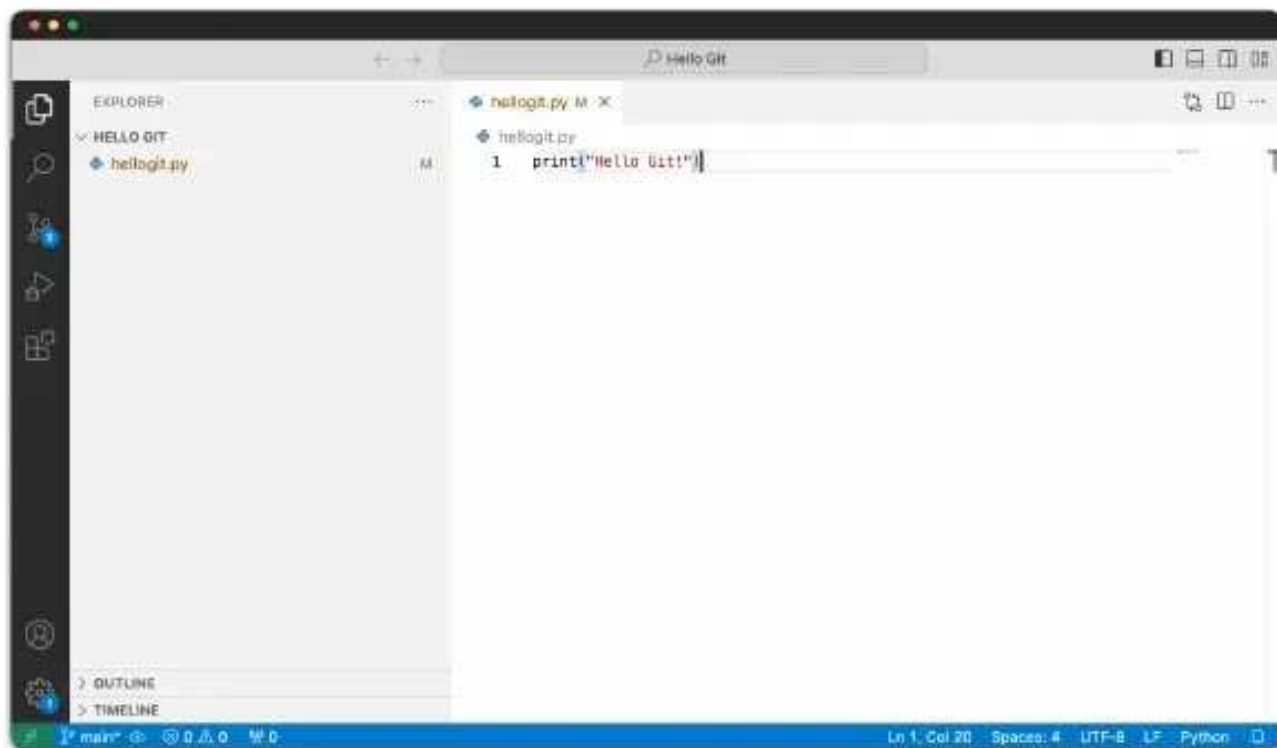
Nos vamos a nuestro editor de código y creamos un archivo. En este caso, vamos a crear uno en Python

llamado *hellogit.py*. No importa el tipo de archivo o su nombre. Mientras podamos escribir algo en él. Supongamos que se corresponde con el código de nuestro proyecto.

Dentro del archivo, escribimos algo simple, como un `print`, o lo que se nos ocurra. No importa lo que escribamos aquí, lo importante es entender el manejo de control de versiones. Supongamos que en algún momento editamos este archivo y agregamos otra línea

¹<https://mouredev.com/git-github-06>

de texto, otro print. A continuación, nos preguntamos: *¿qué tenía yo el otro día en este archivo? ¿cómo vuelvo atrás?*. Aquí es donde entra la importancia del control de versiones.



Sin un control de versiones, cada vez que trabajamos en un proyecto perdemos toda referencia a lo que había antes. De aquí que comencemos a guardar copias de un mismo proyecto para intentar respaldarlo. Ya nunca más será así.

Para trabajar con Git, nos vamos a la consola, y nos situamos en la carpeta raíz donde está nuestro archivo *hellogit.py*.

Para indicar que queremos trabajar con Git en este directorio, simplemente ejecutamos `git init` en la consola. Al hacer esto, se creará una carpeta oculta llamada *.git*. No necesitamos entender todo lo que hay dentro de esta carpeta, pero es donde Git guarda las referencias asociadas al sistema de control de versiones.



Ya está, desde este momento nuestro directorio (y todo lo que se encuentre en su interior) trabaja con un control de versiones, lo que significa que podemos empezar a emplear todas las características que Git nos proporciona. A medida que avancemos, iremos aprendiendo mucho más sobre ellas.

Cuando ejecutamos `git init`, la consola nos mostró un mensaje diciendo que se había creado una **rama** llamada **master** (de momento no le prestaremos atención a estos términos). Dependiendo de la terminal que estemos utilizando, incluso podría indicarnos el

nombre de la rama en la que nos encontramos. Este no es algo proporcionado por Git, sino que depende de cómo está configurada la terminal a la hora de mostrarnos rutas e información asociada a ellas. Si queremos personalizar nuestro terminal, podemos utilizar, entre otras muchas opciones, **oh-my-zsh**. Con esta utilidad podremos configurar nuestra consola y hacerla más agradable a la hora de trabajar con Git:

ohmyz.sh²

²<https://ohmyz.sh>

Capítulo 7: Ramas

Comandos

- 1 `git config --global init.defaultBranch main`
- 2 `git branch -m main`

Conceptos

Introducción

Las **ramas** (llamadas *branch*) en Git son una de las características más poderosas de esta herramienta de control de versiones. Cuando hablamos de *ramas* nos referimos simplemente a diferentes líneas de desarrollo separadas, en las cuales podemos trabajar de manera independiente, y sin afectar el trabajo que se está realizando en otras *ramas*. Esto nos permite probar distintas ideas, experimentar con diferentes enfoques y hacer cambios importantes sin afectar al código principal del proyecto.

En la mayoría de los proyectos, es común poseer varias *ramas* de trabajo al mismo tiempo. Por ejemplo, podemos usar *ramas* para el desarrollo de distintas funcionalidades, para corrección de errores, para probar nuestros conceptos, etc. Cada una de estas *ramas* puede tener su propio conjunto de cambios y compromisos,

lo que permite una mayor flexibilidad y control en el desarrollo del proyecto.

Git mantiene un seguimiento de todas las *ramas* en un repositorio, y nos permite cambiar de una *rama* a otra con facilidad.

Ventajas

Una de las ventajas de trabajar con *ramas* en Git es que podemos *fusionar* fácilmente los cambios de una *rama* en otra. Por ejemplo, si hemos desarrollado una característica en una *rama*, asociada a una nueva funcionalidad, podemos *fusionar* los cambios de esa *rama* en la *rama* principal, para integrarlos así dentro del proyecto base. Git proporciona herramientas para realizar esta fusión de manera sencilla, rápida y segura.

Es importante tener en cuenta que trabajar con *ramas* en Git puede ser un poco complicado al principio, especialmente si no estamos familiarizados con la terminología, o si no sabemos cómo manejar las *ramas* correctamente. Sin embargo, una vez que entendamos cómo funcionan las *ramas*, y cómo podemos utilizarlas para nuestro beneficio, seremos capaces de trabajar de manera mucho más eficiente y efectiva en nuestro proyecto.

Conclusión

Las *ramas* en Git nos permiten trabajar de manera independiente en diferentes líneas de evolución de un proyecto, lo que nos brinda una mayor flexibilidad y control en el desarrollo del mismo. Es importante

entender cómo funcionan las *ramas*, y cómo podemos utilizarlas en nuestro beneficio. Con práctica y experiencia podemos convertirnos en expertos en el manejo de las *ramas* en Git, aprovechando así al máximo la característica principal de esta herramienta de control de versiones.

Curso

Lección 7: mouredev.com/git-github-07¹

Inicio: 00:42:23 | Duración: 00:02:58

En esta lección vamos a hablar de las **ramas** en Git. Ya hemos creado un *repositorio* en la raíz de nuestro proyecto, y nos encontramos en la *rama* llamada **master**. Muy pronto entenderemos qué significa esto.

¿Qué es una *rama*? Podemos imaginarnos una *rama* de un árbol que se divide en otras *ramas*, y estas en otras. El código que creamos puede seguir diferentes flujos, teniendo cada *rama* un nombre y propósito. Por ejemplo, en nuestro caso, la *rama* en la que nos encontramos situados se llama *master*. Esta *rama* contiene nuestro código y proyecto actual, el directorio que hemos nombrado como *Hello Git*.

Como decíamos, nos encontramos en la *rama master*, que es como Git ha decidido llamar a esta primera *rama* del sistema de control de versiones. Sin embargo, hay otros nombres que se han introducido recientemente para referirse a la *rama* principal, como **main** o **trunk**, y ya no como *master*. El propio GitHub ya utiliza *main*

¹ <https://mouredev.com/git-github-07>

en lugar de *master*, aunque ya hablaremos de ello más adelante.

Si queremos cambiar el nombre de nuestra *rama* principal a *main*, podemos ejecutar el siguiente comando: `git config --global init.defaultBranch main` (un nuevo comando de configuración global de Git). Así, al crear nuevos repositorios, la *rama* principal se llamará *main* por defecto. Para cambiar el nombre de la *rama* actual de nuestro proyecto podemos usar `git branch -m main`. De esta manera, nuestra *rama master* pasará a llamarse *main*.



```
~/Desktop/Hello Git git init
hint: Using 'master' as the name for the initial branch. This default branch name
hint: is subject to change. To configure the initial branch name to use in all
hint: of your new repositories, which will suppress this warning, call:
hint:
hint:   git config --global init.defaultBranch <name>
hint:
hint: Names commonly chosen instead of 'master' are 'main', 'trunk' and
hint: 'development'. The just-created branch can be renamed via this command:
hint:
hint:   git branch -m <name>
Initialized empty Git repository in /Users/braismoure/Desktop/Hello Git/.git/
~/Desktop/Hello Git master git branch -m main
~/Desktop/Hello Git main
```

Recuerda que en la actualidad es común usar *main* en lugar de *master* para referirse a la *rama* principal de un proyecto.

Capítulo 8: Guardado. \$git add y \$git commit

Comandos

- 1 `git status`
- 2 `git add <archivo>`
- 3 `git add .`
- 4 `git commit -m "<mensaje>"`

Conceptos

Introducción

Hemos dado nuestros primeros pasos en Git. Ahora que ya conocemos algunos comandos básicos es importante que sigamos profundizando en más conceptos clave.

Como mencionamos en los capítulos anteriores, Git trabaja con *repositorios* y *ramas*. El *repositorio* es un espacio de almacenamiento que guarda el historial de cambios de nuestro proyecto. En el *repositorio* se encuentra toda la información de nuestro proyecto, incluyendo los distintos *puntos de guardado* que se hayan realizado. Una *rama*, por otro lado, es una línea de desarrollo independiente que parte de un *punto de*

guardado o `commit`. Las *ramas* nos permiten trabajar en diferentes funcionalidades de nuestro proyecto de manera independiente y segura, sin afectar a la *rama* principal o a otras *ramas* secundarias.

Commit

En cuanto al concepto de `commit`, debemos conocer que se refiere a la toma de una *fotografía* del estado actual de nuestro proyecto en un momento determinado. Cada vez que realizamos un `commit`, estamos guardando los cambios que hayamos realizado (y que nosotros seleccionemos) en nuestro proyecto en ese momento

específico. Los `commits` se almacenan en el *historial* de cambios del *repositorio* y se identifican por un **hash**, de identificador único.

El comando `git add` nos permite añadir archivos al área de **Stage**, que es una zona intermedia donde se preparan los cambios que queremos incluir en nuestro próximo `commit`. Es importante tener en cuenta que Git solo guarda los cambios que hayan sido incluidos en el área de *Stage* mediante `git add`. Por eso, es necesario ejecutar este comando cada vez que queramos añadir cambios a

un `commit`. Todo esto lo veremos en detenimiento en la sección destinada a este curso.

Una vez que tenemos los cambios preparados en el área de *Stage*, ejecutaremos el comando `git commit` para crear la *fotografía* y guardar los cambios en el *repositorio*. Debemos añadir un mensaje al `commit` mediante la opción `-m`. Este mensaje debe describir de manera clara y concisa los cambios que hayamos realizado en los ficheros que afectan a ese `commit`.

Curso

1

Lección 8: mouredev.com/git-github-08

Inicio: 00:45:21 | Duración: 00:08:06

Seguimos aprendiendo distintos comandos y cómo funciona el flujo principal de Git. En Git, es importante saber que trabajamos con un *repositorio* y una *rama*, donde el concepto clave es tomar *fotografías, instantáneas* o *puntos de guardado* de nuestro proyecto. Esto quiere decir, capturar y guardar el estado actual de nuestro proyecto para poder reflejarlo así en su historial.

Ya hemos finalizado los preparativos previos, por lo que podemos empezar a realizar guardados en Git. Para comprobar el estado de la *rama* actual, y de Git en nuestro proyecto, usamos el comando `git status`. Al ejecutarlo, nos muestra diferente información, y nos indica, por ejemplo, que en la *rama main* aún no hay *commits*. Más adelante, explicaremos cómo hacer un commit. Por ahora, vemos que tenemos el archivo *hellogit.py*, que hemos creado anteriormente, y donde hemos añadido un `print` (básicamente para que el fichero modifique su estado al cambiar el contenido de su interior).

¹ <https://mouredev.com/git-github-08>

```

~/Desktop/Hello Git git init
hint: Using 'master' as the name for the initial branch. This default branch name
hint: is subject to change. To configure the initial branch name to use in all
hint: of your new repositories, which will suppress this warning, call:
hint:
hint:   git config --global init.defaultBranch <name>
hint:
hint: Names commonly chosen instead of 'master' are 'main', 'trunk' and
hint: 'development'. The just-created branch can be renamed via this command:
hint:
hint:   git branch -m <name>
Initialized empty Git repository in /Users/braismoure/Desktop/Hello Git/.git/
~/Desktop/Hello Git p master git branch -m main
~/Desktop/Hello Git p main git status
On branch main
No commits yet
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    hellogit.py

nothing added to commit but untracked files present (use "git add" to track)
~/Desktop/Hello Git p main

```

En mi caso, también hay un archivo llamado *.DS_Store*, que es un archivo temporal y oculto creado por *macOS*. No nos interesa este archivo, así que lo dejamos al margen. Nos enfocaremos únicamente en el archivo de código de nuestro proyecto, llamado *hellogit.py*.

Mediante `git status`, Git nos señala que, aunque conoce estos archivos (*hellogit.py* y *.DS_Store*), no los tiene guardados de ninguna manera. Para guardarlos, debemos primero añadirlos utilizando `git add`. En nuestro caso, queremos añadir y versionar únicamente *hellogit.py*. Para hacerlo, ejecutamos la sentencia `git add hellogit.py`. A continuación, al ejecutar de nuevo `git status`, veremos que *hellogit.py* ya se encuentra en el área de *Stage*, una zona intermedia lista para ser guardada mediante un `commit`. Muy pronto conoceremos este nuevo concepto.

```
~/Desktop/Hello Git git init
hint: Using 'master' as the name for the initial branch. This default branch name
hint: is subject to change. To configure the initial branch name to use in all
hint: of your new repositories, which will suppress this warning, call:
hint:
hint:   git config --global init.defaultBranch <name>
hint:
hint: Names commonly chosen instead of 'master' are 'main', 'trunk' and
hint: 'development'. The just-created branch can be renamed via this command:
hint:
hint:   git branch -m <name>
Initialized empty Git repository in /Users/braismoure/Desktop/Hello Git/.git/
~/Desktop/Hello Git master git branch -m main
~/Desktop/Hello Git main git status

On branch main
No commits yet

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

nothing added to commit but untracked files present (use "git add" to track)
~/Desktop/Hello Git main git add hellogit.py
~/Desktop/Hello Git main *
```

Con `git add` ejecutado, y siendo conscientes de los ficheros de los que queremos tomar la primera *fotografía*, lo siguiente será confirmar dicha acción.

Recordemos los comandos: `git init` para iniciar el *repositorio*, `git status` para ver el estado de los archivos en la *rama*, y `git add` para añadirlos al área de preparación o *Stage*. Debemos saber que, si ejecutamos `git add .`, añadiremos todos los archivos pendientes de versionar, aunque en este ejemplo lo hemos hecho de uno en uno.

```
hint:
hint: git config --global init.defaultBranch <name>
hint:
hint: Names commonly chosen instead of 'master' are 'main', 'trunk' and
hint: 'development'. The just-created branch can be renamed via this command:
hint:
hint: git branch -m <name>
Initialized empty Git repository in /Users/braismoure/Desktop/Hello Git/.git/
~/Desktop/Hello Git master git branch -m main
~/Desktop/Hello Git main git status
On branch main
No commits yet
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    hellogit.py
nothing added to commit but untracked files present (use "git add" to track)
~/Desktop/Hello Git main git add hellogit.py
~/Desktop/Hello Git main + git status
On branch main
No commits yet
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   hellogit.py
~/Desktop/Hello Git main +
```

Ya tenemos listo el archivo *hellogit.py* para la *fotografía*. Lo siguiente es tomarla utilizando un `commit`. Sin más, para hacer una *fotografía* de lo que tenemos en el área de *Stage*, ejecutamos `git commit`. Podríamos simplemente presionar *Enter* para lanzarlo, pero esto abriría un editor de texto dentro de la terminal, donde deberíamos escribir un comentario asociado al `commit`. En lugar de eso, usaremos el comando `git commit -m`, seguido de un mensaje que describa de forma concisa qué se incluye en la *fotografía*, por ejemplo, “*Este es mi primer commit*”. De esta forma, especificando la propiedad `-m`, asociamos un mensaje al `commit` sin abrir un editor.

```

hint:  git branch -m <name>
Initialized empty Git repository in /Users/braismoure/Desktop/Hello Git/.git/
~/Desktop/Hello Git  master  git branch -m main
~/Desktop/Hello Git  main    git status

On branch main

No commits yet

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

nothing added to commit but untracked files present (use "git add" to track)
~/Desktop/Hello Git  main    git add hellogit.py
~/Desktop/Hello Git  main +  git status

On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   hellogit.py
~/Desktop/Hello Git  main +  git commit
Aborting commit due to empty commit message.
~/Desktop/Hello Git  main +  git commit -m "Este es mi primer commit"
[main (root-commit) cee84b4] Este es mi primer commit
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 hellogit.py
~/Desktop/Hello Git  main

```

Al ejecutar este comando, Git nos informa de que se ha creado un commit con un **hash** asociado. Este **hash** es muy importante en Git, ya que identifica de manera única cada *punto de guardado* en el sistema de control de versiones.

Si volvemos a ejecutar `git status`, veremos que ahora ya existe un commit, pero el archivo `.DS_Store` (propio de *macOS*) sigue sin estar incluido en el área de *Stage*, ya que no hemos realizado su `git add`, y por lo tanto no se ha reflejado en el commit.

De esta manera hemos realizado nuestra primera *fotografía* en Git.

Capítulo 9: Estado `$git` `log` y `$git status`

Comandos

- 1 `git log`
- 2 `git status`

Conceptos

Introducción

Como hemos visto, una de las principales características de Git es su capacidad para tomar *fotografías* de los cambios realizados en el código fuente. Estas fotografías se denominan *commits*, y representan una *instantánea* del estado del proyecto en un momento determinado.

Log

Para visualizar las *fotografías* realizadas en un *repositorio* de Git, se utiliza el comando `git log`. Al ejecutarlo, Git muestra una lista con todos los *commits* realizados en el *repositorio*, incluyendo el *hash* único que identifica a cada uno de ellos. Esta información es útil para rastrear

la evolución del proyecto y asegurarnos de que todas las *fotografías* se han almacenado según lo esperado.

Además de mostrar los *commits*, `git log` también comparte información sobre el autor de cada uno de ellos, incluyendo su nombre de usuario y la dirección de correo electrónico. Esta información es importante porque ayuda a identificar quién realizó cada cambio en el código fuente. Al configurar Git, recordemos que es obligatorio especificar un nombre de usuario y una dirección de correo electrónico para poder realizar *commits* en el *repositorio*.

Status

El comando `git status` es otro de los comandos más útiles en Git. Este comando muestra el estado actual del repositorio, incluyendo los archivos modificados, eliminados o agregados, así como también los archivos que se han añadido al área de *Stage*, junto con los que aún no han sido *seguidos* por Git.

Cuando se ejecuta `git status`, Git muestra una lista de los archivos modificados en el directorio de trabajo. Estos archivos pueden, o no, haber sido *seguidos* por Git. Si se han *seguido*, se mostrarán como *cambios listos para commit* en el área de *Stage*. Si no se han *seguido*, se mostrarán como *cambios no rastreados*. El término *seguir* hace referencia a si Git está teniendo en cuenta a ese archivo para realizar futuras acciones de guardado o eliminación.

Como ya hemos visto, para agregar los archivos modificados al área de *Stage*, se utiliza el comando `git add`, seguido del nombre del archivo modificado. Esto mueve el archivo al área de *Stage*, que es donde se

preparan los cambios para ser incluidos en el próximo `commit`.

También es importante destacar que `git status` muestra información adicional sobre el estado del repositorio, como la *rama* actual en la que nos encontramos trabajando, información sobre los *commits* (como el mensaje asociado), y si se ha fusionado con otra *rama*.

HEAD

El concepto de **HEAD** también es importante en Git. *HEAD* es un puntero que apunta al `commit` actual en el repositorio. En otras palabras, *HEAD* indica la posición actual en la línea de tiempo del proyecto. Cuando se realiza un nuevo `commit`, *HEAD* se mueve al nuevo `commit`, convirtiéndose en el `commit` más actual. Esto nos permite movernos fácilmente entre las diferentes versiones del proyecto y ver exactamente qué cambios se realizaron en cada `commit`.

Curso

Lección 9: mouredev.com/git-github-09¹

Inicio: 00:53:27 | Duración: 00:04:21

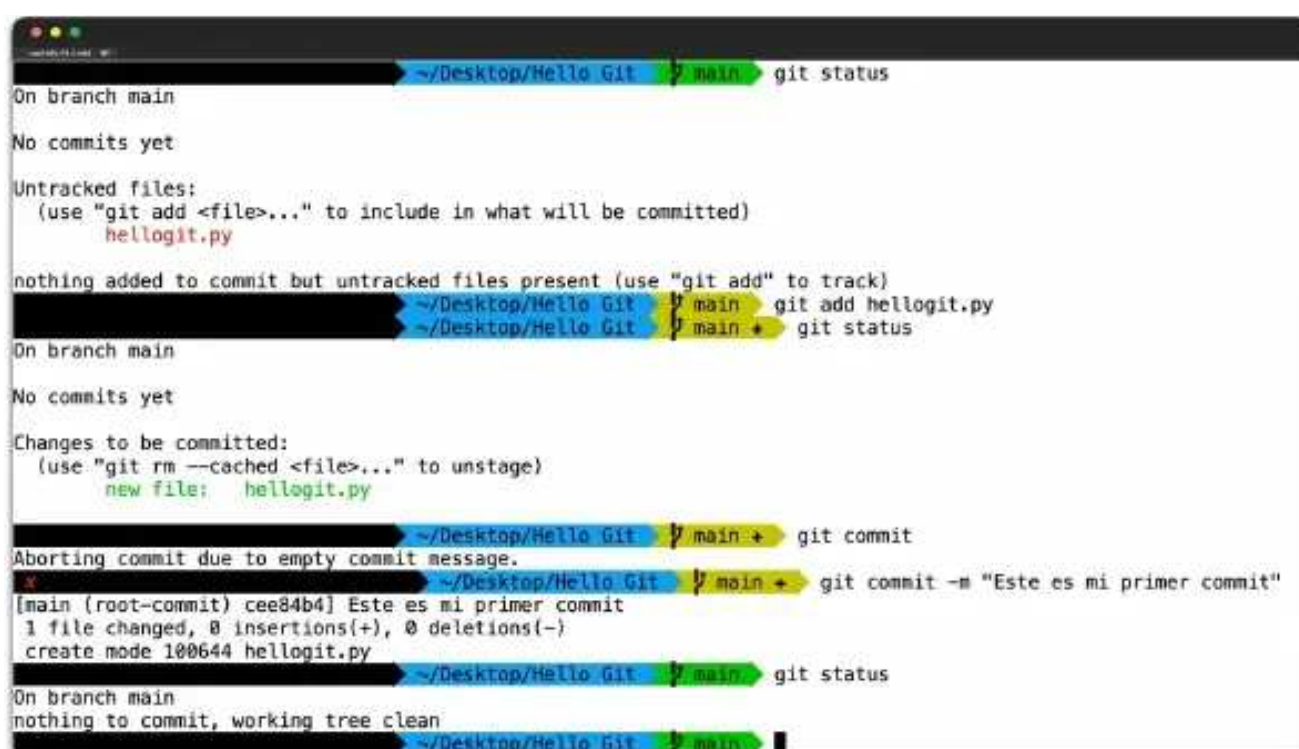
¿Quieres saber si se ha realizado un `commit` en Git? Para ello, vamos a usar `git log`.

Comprobaremos en la consola que ya tenemos un `commit`, junto a su *hash* único en la *rama main*. Más adelante, también hablaremos sobre el concepto de

¹<https://mouredev.com/git-github-09>

HEAD. El autor es importante, por eso hablamos de que era obligatorio a la hora de configurar Git. Si intentamos hacer un commit sin usuario o email, no nos dejará hacerlo. En nuestro caso, nuestro propio usuario, con cierto email, en una fecha y hora determinadas, y con un *hash* único asignado, generó un commit diciendo “*Este es mi primer commit*”. El que consideramos como nuestra primera *fotografía*.

Sigamos trabajando. Imaginemos que ahora creamos otro archivo llamado *hellogit2.py*, y le añadimos un print. Regresamos a la terminal y escribimos `git status`. Nos muestra que no solo está el archivo anterior, sino también uno nuevo llamado *hellogit2.py*, que podría añadirse al área de *Stage*. Comenzamos en un área *Local*, y tenemos la posibilidad de pasar archivos al área de *Stage*. ¿Cómo lo hacemos? Si queremos añadir *hellogit2.py* a un commit, tendremos que escribir `git add hellogit2.py`, y a continuación `git commit -m "Este es mi segundo commit"`. Finalizamos el commit y se añade ese nuevo archivo.



```
~/Desktop/Hello Git main git status
On branch main
No commits yet
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    hellogit.py

nothing added to commit but untracked files present (use "git add" to track)
~/Desktop/Hello Git main git add hellogit.py
~/Desktop/Hello Git main git status
On branch main
No commits yet
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   hellogit.py

~/Desktop/Hello Git main git commit
Aborting commit due to empty commit message.
~/Desktop/Hello Git main git commit -m "Este es mi primer commit"
[main (root-commit) cee84b4] Este es mi primer commit
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 hellogit.py
~/Desktop/Hello Git main git status
On branch main
nothing to commit, working tree clean
~/Desktop/Hello Git main
```

Al escribir `git log`, ahora aparecerán dos *commits*.

En uno, agregamos el primer archivo, y en el otro, el segundo. Cada commit ha generado un *hash* diferente. De esta manera tendremos dos *fotografías*, reflejando, además del archivo *hellogit.py*, también el archivo *hellogit2.py*.



```
commit: 43f281d603ab4ae38b7a67c7d4e26d281b59ef08 (HEAD -> main)
Author: MoureDev <braismoure@mouredev.com>
Date: Tue Mar 5 13:35:00 2024 +0100

Este es mi segundo commit

commit: cee84b47b96232900c796cd01de72fc89e7b0f7b
Author: MoureDev <braismoure@mouredev.com>
Date: Tue Mar 5 13:30:23 2024 +0100

Este es mi primer commit
(END)
```

Estas dos *fotografías* nos permiten movernos entre ambos estados del proyecto, y visualizar en cada caso lo que contienen. Ahora, imaginemos que editamos el primer archivo, cambiando el mensaje del print. Al hacer esto, hemos modificado el archivo. Si escribimos `git status`, nos dirá que *hellogit.py* ha cambiado con respecto a su última *fotografía*. El sistema de control de versiones lo detectará automáticamente. Visto esto, podemos plantearnos: ¿queremos guardar una *fotografía* de este momento? Si decidimos no guardarla, y continuar con el desarrollo, podemos hacerlo. Por ejemplo, cambiemos el contenido de *hellogit2.py* y guardemos los cambios. Al volver a ejecutar `git status` nos mostrará que hemos modificado tanto *hellogit.py* como *hellogit2.py*.

Capítulo 10: Operaciones con ramas \$git checkout y \$git reset

Comandos

- 1 `git checkout <archivo>`
- 2 `git reset`
- 3 `git log --graph`
- 4 `git log --pretty=oneline`
- 5 `git log --decorate`
- 6 `git log --graph --pretty=oneline --decorate`

Conceptos

Introducción

Vamos a explorar por una parte los nuevos comandos `git checkout` y `git reset`, que nos permitirán, entre otras cosas, regresar a estados anteriores de nuestros archivos que aún no han guardado sus cambios en Git, y, por otro lado, también aprenderemos a visualizar el historial de *commits* de diferentes maneras.

Checkout

Supongamos que acabamos de hacer algunos cambios en nuestro código y deseamos volver al estado anterior sin guardar las modificaciones. Para hacer esto, podemos usar el comando `git checkout`. Este comando nos permite situarnos en un punto específico del historial de *commits* o de un archivo.

Por ejemplo, si deseamos volver al estado previo de un archivo antes de modificarlo, podemos ejecutar el comando `git checkout <archivo>`.

Este comando nos llevará al estado previo de dicho archivo, correspondiente a la última *fotografía* tomada en la *rama* actual.

Reset

Si deseamos volver a la última *fotografía* completa tomada, podemos escribir `git reset`. Al lanzar este comando se nos informará de que se perderán los cambios en los archivos que no forman parte de un `commit`. Hecho esto, recuperaremos el contenido original del último punto de guardado de la *rama*.

Al ejecutar un `git reset`, Git nos mostrará una lista de archivos modificados que aún no se han guardado. Podemos elegir si deseamos hacer un *reset* de todos los archivos, o únicamente de algunos.

Visualizaciones

Si hacemos memoria, el comando `git log` nos permitía visualizar todo el historial de cambios que se han

realizado en un proyecto. Es muy útil para rastrear el progreso del proyecto y ver qué cambios se han llevado a cabo.

Este comando nos mostrará una lista de todos los *commits* que se han realizado en el proyecto, junto con información detallada sobre quién hizo los cambios, cuándo se hicieron, y qué archivos se modificaron.

- Si deseamos revisar el historial de *commits* de una manera más visual, podemos usar el comando `git log --graph`. Este comando nos mostrará una representación gráfica de las *ramas* (cómo se dividen, y cómo se relacionan entre sí) y los *commits* del proyecto.
- Si queremos ver el historial de *commits* de una manera más simplificada, podemos usar el comando `git log --pretty=oneline`. Este comando te mostrará una vista rápida de cada commit en una sola línea. Podremos consultar rápidamente el *hash* del commit y el mensaje de confirmación desde una vista compacta.
- También podemos utilizar el comando `git log --decorate` para consultar información adicional sobre los *commits*. Este comando nos permite visualizar rápidamente la línea de progreso de nuestra *rama* y sus *etiquetas* (un concepto que veremos más adelante) sin mostrar el *hash* completo.

Por supuesto, puedes combinar todas las propiedades nombradas: `git log --graph --pretty=oneline --decorate`

Curso

1

Lección 10: mouredev.com/git-github-10

Inicio: 00:57:48 | Duración: 00:05:14

Imaginemos que no queremos tener en cuenta ciertos cambios en nuestro código, y deseamos regresar a un estado anterior sin guardar lo que hemos estado haciendo en el proyecto. Para eso, podemos usar `git checkout`. Si escribimos `git` y presionamos la tecla de tabulación, veremos diferentes comandos que podemos utilizar. Recuerda que esto nos puede resultar muy útil.

Como comentábamos, vamos a explorar el comando `git checkout`. Un comando que nos permite situarnos en un punto específico de un commit o archivo.

Supongamos que queremos volver al estado previo del archivo *hellogit2.py*, antes de modificarlo. Si recordamos, no hicimos un commit de los cambios, así que vamos a ejecutar `git checkout hellogit2.py`. Al lanzarlo, se nos indicará que se ha actualizado. Ahora, el archivo está en su estado previo al cambio, como en la última *fotografía* realizada en la esa *rama*.

Si queremos volver a la última *fotografía* completa, podemos escribir `git reset`. Al lanzarlo, Git nos informará de que las modificaciones no guardadas en los archivos se perderán.

Si lo deseamos, podemos hacer un `git reset` de todo el proyecto, o simplemente usar `git checkout` y regresar al estado previo de un archivo concreto.

¹ <https://mouredev.com/git-github-10>



Recuperemos un concepto del que ya hemos hablado: la revisión del `log`. Vamos a extender su funcionalidad para conocerlo más en profundidad.

Hagamos un nuevo `commit`, modificando el contenido de `hellogit.py`. Como siempre, ejecutaremos `git add hellogit.py` para añadir los cambios, y `git commit -m "<mensaje>"` para confirmarlos. Con `git log` ahora descubriremos que ya se han realizado tres *commits*.

Podemos ver ese historial de *commits* de diferentes maneras. Por ejemplo, utilizando `git log --graph` para mostrar una representación gráfica de las *ramas*. Si deseamos verlo de manera más simplificada, podemos ejecutar `git log --graph --pretty=oneline`. Así tendremos una vista rápida de los tres *commits* en una única línea cada uno.


```
commit 2d2a582eb0b3b3a9df4095195b18e9b587678985 (HEAD -> main)
Author: MoureDev <braismoure@mouredev.com>
Date: Thu Mar 7 08:16:27 2024 +0100

    Se actualiza el texto del print

commit 43f281d683ab4ae38b7d67c7d4e26d281b59ef06
Author: MoureDev <braismoure@mouredev.com>
Date: Tue Mar 5 13:35:00 2024 +0100

    Este es mi segundo commit

commit cee84b47a96732908d796cd81de72fc89e7b017b
Author: MoureDev <braismoure@mouredev.com>
Date: Tue Mar 5 13:30:23 2024 +0100

    Este es mi primer commit
(END)
```

Además, es posible abreviar los *hashes* de los *commits* utilizando otro comando. ~~Problemas con git log~~ ^{graph --decorate --all --oneline}. Esto nos permite visualizar rápidamente la línea de progreso de nuestra *rama* sin mostrar el *hash* completo (suficiente para la mayoría de los casos en los que tengamos que utilizar ese *hash*).

En resumen, en esta lección hemos aprendido los comandos `git checkout` y `git reset` para regresar a estados anteriores de nuestros archivos. Por otra parte, también hemos ampliado nuestro conocimiento sobre cómo visualizar el *log* de *commits* de diferentes maneras.

Capítulo 11: Alias \$git

Comandos

- 1 `git config`
- 2 `git config --global alias.tree '<comando>'`
- 3 `git tree`

Conceptos

Introducción

Git es una herramienta esencial en el sector del desarrollo de software. Posiblemente pasaremos gran parte de nuestro tiempo interactuando con Git y sus comandos. Sin embargo, a veces puede resultar difícil recordar ciertos comandos de Git, junto a sus propiedades y combinaciones.

Afortunadamente, Git nos permite crear **Alias** para simplificar este proceso.

Alias

Para acceder a la configuración de Git utilizábamos el comando `git config` junto al modificador `--global`,

para que esta se aplique a todas las interacciones de nuestro usuario en Git.

Una vez que hemos accedido a la configuración de Git, podemos crear distintos *Alias* para los comandos que usamos con frecuencia, o que nos resulten especialmente complejos. Por ejemplo, podríamos crear un *Alias* llamado **test**, que tenga asociado un comando concreto de Git.

Para crear ese *Alias* con el nombre *test*, simplemente lanzamos el comando `git config --global alias.test '<comando>'`. Hecho esto, cada vez que necesitemos ejecutar ese comando, simplemente tendremos que escribir `git test` desde la terminal.

Conclusión

La creación de *Alias* es solo una de las formas con que podemos personalizar Git para adaptarlo a nuestras necesidades. A medida que nos familiaricemos con Git y progreseemos en su uso, podremos agregar más *Alias* para los comandos que usamos con frecuencia.

Por darte un ejemplo, también podemos personalizar

Git ajustando la configuración predeterminada. Tendremos la posibilidad de cambiar el editor de texto predeterminado a la hora de confirmar un `commit`, establecer límites para los mensajes asociados al `commit`, o cambiar la forma en que se muestran los mensajes de error. Entre muchas otras configuraciones.

Curso

Lección 11: mouredev.com/git-github-11¹

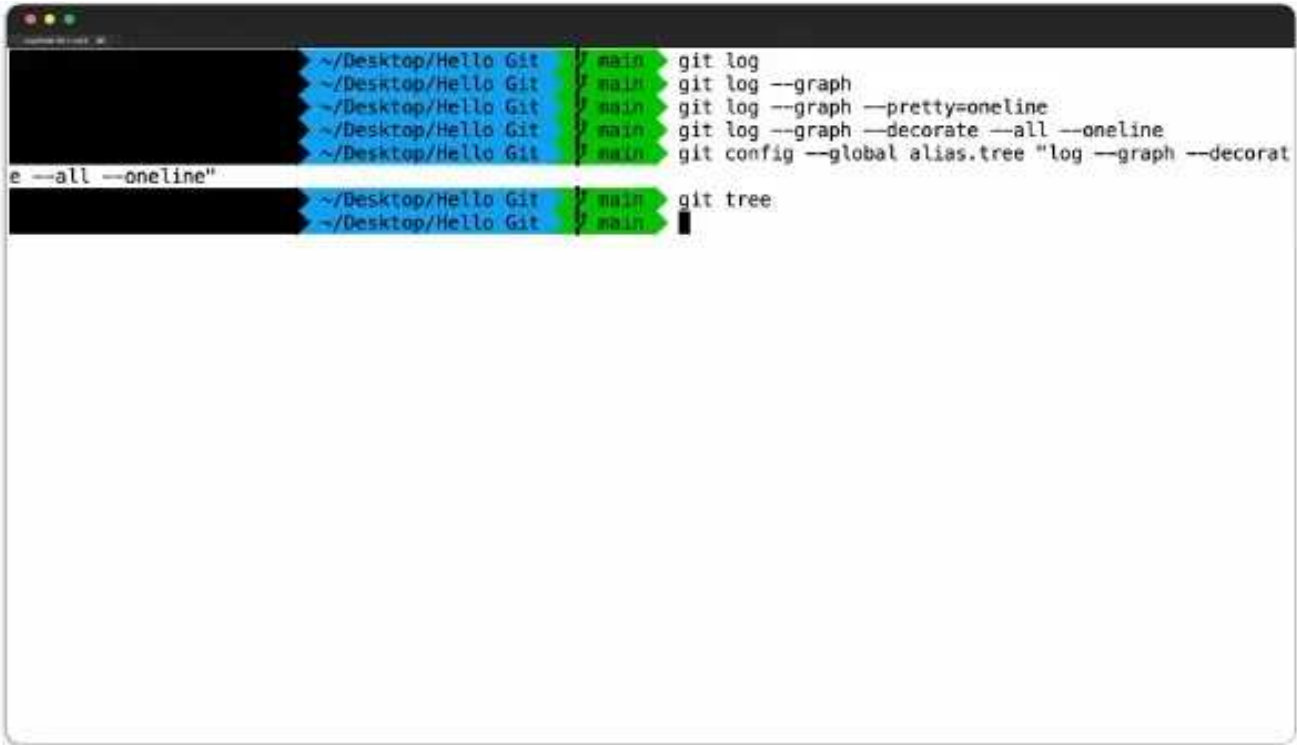
Inicio: 01:03:02 | Duración: 00:02:03

A veces, resulta difícil recordar ciertos comandos de Git, junto a sus propiedades y combinaciones. Por suerte, Git nos permite crear los llamados **Alias** dentro de la configuración de nuestro usuario. ¿Recuerdas cómo se accede a la configuración de Git? Utilizábamos el comando `git config`. Y la marcábamos como `--global` para que la configuración fuera específica del usuario de la sesión.

A continuación, vamos a crear un *Alias* relacionado con el capítulo anterior, y el comando complejo `git log --graph --decorate --all --oneline`. ¿*Qué nombre le pondremos?* Vamos a elegir **tree**, ya que nos hace pensar en la representación de árbol de nuestras *ramas*. Este nombre es totalmente personalizable. Después, entre comillas, especificamos el comando que queremos ejecutar y asociar. En este caso, será `git config --global alias.tree 'log --graph --decorate --all --oneline'`. Ejecutamos el comando y ya estaría creado

~~nuestro nuevo *Alias*.~~

¹<https://mouredev.com/git-github-11>



```
~/Desktop/Hello Git | main git log
~/Desktop/Hello Git | main git log --graph
~/Desktop/Hello Git | main git log --graph --pretty=oneline
~/Desktop/Hello Git | main git log --graph --decorate --all --oneline
~/Desktop/Hello Git | main git config --global alias.tree "log --graph --decorat
e --all --oneline"
~/Desktop/Hello Git | main git tree
```

¿Recuerdas del archivo de configuración donde guardamos nuestro usuario y correo electrónico? Si lo abrimos ahora veremos que también incluye el *Alias* que acabamos de crear. Este *Alias*, llamado *tree*, nos permitirá escribir simplemente `git tree` para lanzar su ejecución asociada, sin tener que recordar el comando complejo completo.

Conforme avancemos, podremos agregar más *Alias*. Únicamente tendremos que ir añadiéndolos a la configuración. Incluso podemos crear un *Alias* que ejecute varios comandos al mismo tiempo. La idea principal es que conozcamos la posibilidad de personalizar Git creando diferentes comandos más

Capítulo 12: Ignorar archivos `.gitignore`

Comandos

- 1 `touch .gitignore`
- 2 `git add .gitignore`

Conceptos

Introducción

A veces puede suceder que no queremos incluir ciertos archivos en un `commit`, ya sea porque son temporales, exponen información delicada, o simplemente no son relevantes para el proyecto.

En estos casos, es necesario saber cómo ignorar archivos en Git. En este capítulo, vamos a explicar cómo hacerlo utilizando el archivo llamado **`.gitignore`**.

El archivo `.gitignore`

El `.gitignore` es un archivo especial que Git utiliza para ignorar ciertos elementos, directorios o patrones en un proyecto. Este archivo se coloca en la raíz del proyecto y

se le pueden añadir reglas. Es importante destacar que, los archivos que se añadan a *.gitignore* no se eliminarán del sistema, sino que simplemente serán ignorados por Git a la hora de trabajar con ellos.

Creación

Para crear un archivo *.gitignore* podemos utilizar el comando `touch` desde la terminal o desde el propio sistema de archivos del sistema operativo. Es importante que el archivo lleve un punto al principio para que sea interpretado como oculto. También es esencial que se llame exactamente *.gitignore*. Una vez creado, se visualizará en la lista de archivos del proyecto.

Uso

Para ignorar archivos en Git, debemos añadir la ruta o el nombre del archivo como contenido del *.gitignore*. Por ejemplo, si queremos ignorar un *<nombre_archivo>* en todo el proyecto, debemos añadir la línea ***/<nombre_archivo>* dentro del *.gitignore*.

Esta línea indica a Git que debe ignorar archivos llamados *<nombre_archivo>*, situados en cualquier parte proyecto. Es importante destacar que la línea debe comenzar con dos asteriscos, que indican que la regla se aplicará en cualquier lugar del proyecto.

Una vez que se ha añadido la línea al archivo *.gitignore*, Git dejará de considerar el archivo *<nombre_archivo>* en el área de *Stage*, y no se incluirá en ningún *commit* futuro.

A continuación, vamos a nombrar las formas más habituales de ignorar archivos. Dentro del archivo

.gitignore se pueden utilizar diferentes mecanismos para especificar los archivos, carpetas o patrones que se quieren ignorar en el sistema de control de versiones.

Algunos de estos mecanismos son:

- Archivos por su nombre: Se puede escribir el nombre exacto del archivo que se quiere ignorar. Por ejemplo: `archivo_temporal.txt`.
- Carpeta completa: Se puede escribir el nombre de una carpeta completa que se quiere ignorar. Por ejemplo: `carpeta_temporal/`.
- Patrón: Se pueden utilizar patrones que coincidan con múltiples archivos o carpetas que se quieren ignorar. Algunos ejemplos de patrones comunes son:
 - `*.log`: Ignora todos los archivos con extensión `.log`.
 - `**/temp`: Ignora la carpeta *temp* en cualquier parte del proyecto.
- Especificar varias reglas: Se pueden utilizar múltiples reglas en el archivo *.gitignore*, separadas por líneas en blanco o por un salto de línea.

Y estos son solo unos pocos mecanismos.

Es importante destacar que Git también soporta algunos caracteres especiales en los patrones de los archivos que se quieren ignorar, como el asterisco (*) para representar cero o más caracteres, el signo de interrogación (?) para representar un solo carácter, el signo de admiración (!) para realizar una negación, o los corchetes ([]) para especificar un conjunto de caracteres. Estos mecanismos pueden ser muy útiles para definir patrones más específicos y detallados.

Comprobación

Para comprobar que un archivo está siendo ignorado por Git, podemos ejecutar el comando `git status`, aunque existen otras opciones. Si el archivo está siendo ignorado, no debería aparecer en la lista de archivos pendientes de añadir al área de *Stage*.

Es importante destacar que el archivo *.gitignore* sí que debe ser añadido al *repositorio* en Git. Para hacer esto, se debe utilizar el comando `git add .gitignore`, y a continuación realizar un `commit` con el mensaje correspondiente. Una vez que se ha añadido el archivo *.gitignore* al *repositorio*, ya no será necesario volver a hacerlo.

Curso

Lección 12: mouredev.com/git-github-12¹

Inicio: 01:05:05 | Duración: 00:03:59

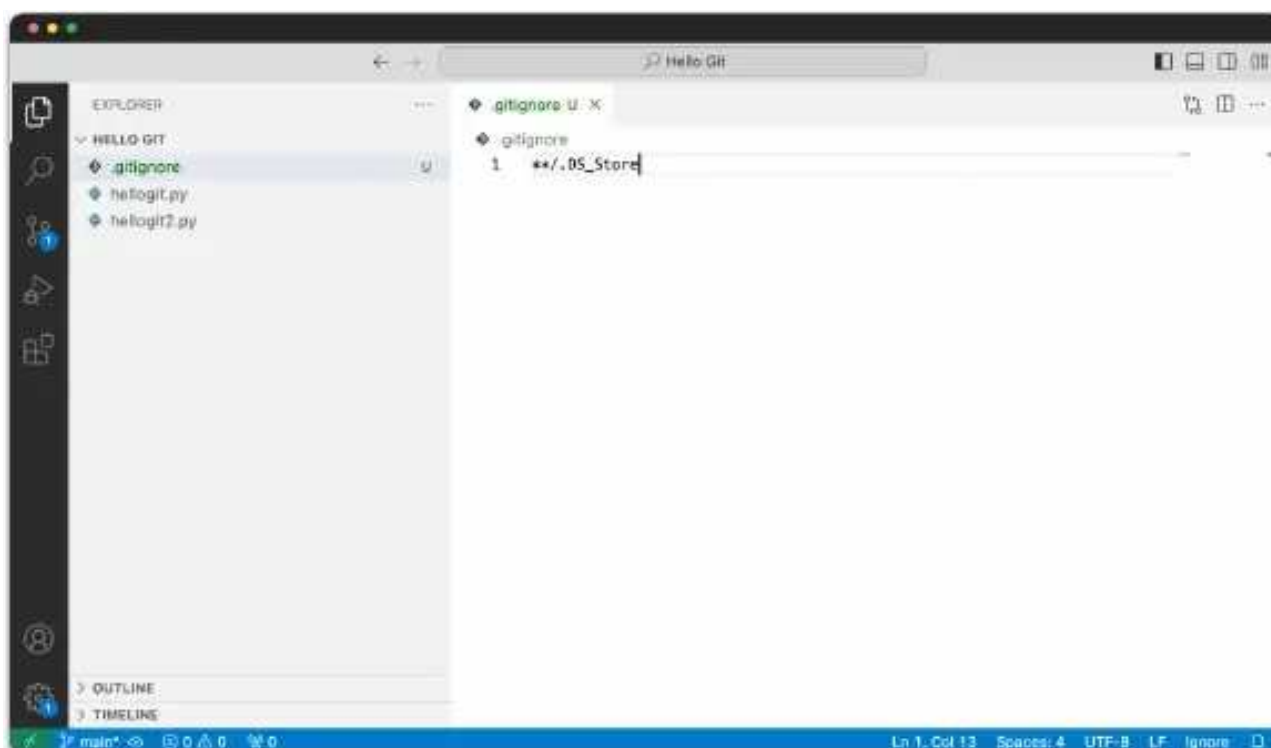
Hasta el momento, hemos realizado tres `commits`, en los que hemos modificado distintos ficheros de nuestro proyecto. Pero quizás nos hemos dejado algo atrás. Si revisamos el estado de nuestro *repositorio* con `git status`, nos daremos cuenta de que existe un archivo llamado *.DS_Store*. Nunca hemos incluido este archivo en los `commits`, porque es temporal y tiene referencias a la propia máquina y al sistema operativo *macOS*. Al subirlos cambios, no queremos incluirlo, y no podemos borrarlo, ya que se regenera constantemente. Aquí es donde entra el concepto de ignorar archivos. Si tenemos la certeza

¹<https://mouredev.com/git-github-12>

de que nunca vamos a querer hacer una *fotografía* de un archivo, significará que tampoco queremos que aparezca como *pendiente* cada vez que ejecutamos `git status`. Para conseguir esto, existe un archivo propio de Git llamado **.gitignore**.

Vamos a crear un nuevo archivo llamado *.gitignore*. Podemos hacerlo desde la terminal con el comando `touch`, o desde el sistema de archivos. Recordemos que tiene que llevar un punto al principio para transformarlo oculto, y, por supuesto, debe llamarse *.gitignore*. Una vez creado, lo visualizaremos en la lista de archivos.

Los archivos, rutas, o expresiones que añadamos al *.gitignore*, serán ignorados por Git. Creamos este archivo en la raíz del proyecto, y, para excluir los *.DS_Store*, agregamos la línea `**/.DS_Store`.



Así estamos indicando que cualquier archivo llamado *.DS_Store* no se tendrá en cuenta para añadir al área de *Stage*. No importa dónde esté ubicado el archivo *.DS_Store*, será ignorado.

Sin embargo, el archivo *.gitignore*, sí que debe ser

añadido al *repositorio*. Lanzamos `git add .gitignore`, y a continuación `git commit -m "<mensaje>"`. Ejecutamos `git status`, y validamos que ya no hay ningún fichero pendiente. Nuestra *rama* está limpia.



```
~/Desktop/Hello Git | main touch .gitignore
~/Desktop/Hello Git | main git add .gitignore
~/Desktop/Hello Git | main + git commit -m "Se añade el .gitignore"
[main 5fe044c] Se añade el .gitignore
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 .gitignore
~/Desktop/Hello Git | main git status
On branch main
nothing to commit, working tree clean
~/Desktop/Hello Git | main
```

Si tu proyecto no tiene un fichero temporal *.DS_Store*, puedes crear uno de ejemplo para poner en práctica lo aprendido ignorando ficheros por nombre.

Capítulo 13: Comparación de commits

\$git diff

Comandos

- 1 `git diff`
- 2 `git diff <hash_commit_a> <hash_commit_b>`
- 3 `git diff --name-only <hash_commit_a> <hash_commit_b>`

Conceptos

Introducción

En este capítulo vamos a profundizar en el concepto de comparación de commits y su contenido.

Diff

El comando `git diff` nos permite consultar los cambios exactos realizados en nuestro código antes de realizar un `commit` (entre otras cosas). Para entender cómo funciona, vamos a utilizar un ejemplo. Hasta el momento, hemos realizado diferentes *commits* en una *rama*, pero ahora

queremos llevar a cabo otras modificaciones en ciertos archivos.

Antes de hacer un `commit`, podemos utilizar el comando `git diff` para examinar los cambios exactos que hemos realizado.

En la consola, los cambios realizados aparecen con un signo menos (-) en la línea que ha sido eliminada, y un signo más (+) en la línea que ha sido añadida.

De esta manera, podemos llevar un control preciso de todos los cambios que hemos llevado a cabo en el proyecto.

Diff entre commits

Además de utilizar el comando `git diff`, para ver los cambios realizados en nuestro código antes de hacer un `commit`, también podemos emplear este comando para visualizar los cambios realizados entre dos *commits* específicos. Esto nos permite detectar todos los cambios efectuados en nuestro proyecto en un período de tiempo determinado.

Para utilizar el comando `git diff` entre dos *commits* específicos, debemos indicar los identificadores únicos de los *commits*, los llamados *hash*, que podemos consultar cuando hacemos un `git log`. Por ejemplo, si queremos ver los cambios realizados entre el `commit` `<hash_commit_a>` y el `commit` `<hash_commit_b>`, debemos escribir el comando `git diff <hash_commit_a> <hash_commit_b>` en la consola.

Este comando nos muestra los cambios realizados entre los dos *commits* especificados. En este caso, los cambios realizados también aparecerán con un signo menos (-) en

la línea que ha sido eliminada, y un signo más (+) en la línea que ha sido añadida.

También podemos utilizar el comando `git diff` con otros argumentos, como el parámetro `--name-only`, que nos muestra solo los nombres de los archivos que han sido modificados entre los dos *commits* especificados. Por ejemplo, si queremos ver los nombres de los archivos modificados entre los *commits* `<hash_commit_a>` y `<hash_commit_b>`, podemos lanzar el comando `git diff --name-only <hash_commit_a> <hash_commit_b>`.

Conclusión

El comando `git diff` es una herramienta muy útil que nos permite ver los cambios realizados en nuestro código, ya sea antes de hacer un *commit*, o entre dos *commits* específicos. Esto nos permite tener un mayor control sobre nuestro proyecto y asegurarnos de que estamos realizando los cambios correctos.

Curso

Lección 13: mouredev.com/git-github-13¹

Inicio: 01:09:04 | Duración: 00:02:50

Vamos a aprender un comando más antes de adentrarnos a fondo en el concepto de *rama*. Hasta el momento, hemos seguido avanzando y realizando diferentes *commits* en nuestra *rama main*.

¹ <https://mouredev.com/git-github-13>

Es todo este tiempo siempre nos hemos encontrado en el extremo de la *rama* actual, pero recordemos que poseemos referencias a los *commits*, llamados *hash*, y, por lo tanto, podemos consultar qué hay en cada punto. Ahora, realizaremos un nuevo cambio en uno de nuestros archivos, *hellogit.py*, modificando el texto de su `print`. Imaginemos que hemos estado programando y no nos acordamos de todo lo que hemos modificado, así que no estamos seguros de si queremos hacer un `commit` de esos cambios. Como queremos saber qué hemos cambiado, con respecto al último `commit`, podremos usar el comando `git diff`.

A terminal window with a dark background and light text. The output of the command 'git diff a/hellogit.py b/hellogit.py' is displayed. It shows the index hash 'b851255..4c0990a 100644'. Below this, it shows the diff for 'a/hellogit.py' and 'b/hellogit.py'. The diff indicates a deletion of a line (marked with '-') and an addition of a new line (marked with '+'). The deleted line is '-print("New Hello Git!")' and the added line is '+print("New Hello Git with changes!")'. The diff ends with '(END)'.

```
diff --git a/hellogit.py b/hellogit.py
index b851255..4c0990a 100644
--- a/hellogit.py
+++ b/hellogit.py
@@ -1,1 @@
-print("New Hello Git!")
+print("New Hello Git with changes!")
(END)
```

Al ejecutarlo, observamos que nos señala que en el archivo *hellogit.py* algo ha desaparecido y algo nuevo ha aparecido. En concreto, una línea ha sido eliminada, y otra línea ha sido añadida. Esto hace referencia a que el contenido del `print` ha sido modificado. De esta manera, y sin hacer un `commit`, podemos examinar los cambios realizados en el código.

Esta funcionalidad es muy importante para entender el valor que Git aporta a nuestro trabajo. Nos permite tener

un control absoluto de todo lo que hacemos con nuestro código, incluso antes de realizar un `commit`. El comando `git diff` nos muestra con un signo menos (-) lo que se ha eliminado, y con un signo más (+) lo que se ha añadido. Gracias a esto podemos llevar un control más preciso de nuestros cambios.

Capítulo 14: Desplazamientos en una rama

Comandos

- 1 `git checkout <hash>`
- 2 `git checkout HEAD -- .`

Conceptos

Introducción

Una de las características más importantes de Git es su capacidad para realizar el seguimiento de cambios en el código, y permitirnos desplazarnos a través de diferentes versiones del mismo. En este sentido, los comandos `git log` y `git checkout` son muy útiles, ya que nos permiten visualizar todo el historial de modificaciones de nuestro proyecto, incluyendo quién hizo cada cambio, cuándo se realizó, y cuál fue el mensaje asociado, así como movernos al estado de cada commit.

Desplazamiento

Aunque el comando `git log` nos permite visualizar todo el historial de cambios, a veces puede resultar necesario desplazarnos a un `commit` específico. Para ello, podemos utilizar el comando `git checkout`, seguido del *hash* del `commit` al que queremos desplazarnos.

Al hacer esto, Git nos advertirá de que podríamos perder cambios que no hemos guardado, ya que nos estamos desplazando a una versión anterior del proyecto. Si estamos seguros de que queremos hacerlo, podemos confirmar, y Git actualizará el contenido de nuestro proyecto por el del `commit` correspondiente.

Es importante tener en cuenta que al desplazarnos a un `commit` anterior, los archivos que han sido modificados o eliminados en versiones posteriores, pueden no aparecer en nuestro proyecto actual. Esto es normal, ya que nos estamos moviendo a una versión anterior del proyecto que no incluye dichos cambios. Sin embargo, si en algún momento queremos volver al estado *fotografiado* del proyecto, podemos utilizar el comando `git checkout HEAD -- ..`. Esto no afectará a los archivos nuevos (no *rastreados*) ni al *área de preparación*.

Recordemos que otra forma de movernos al final de la *rama* es utilizando el *hash* abreviado del último `commit`.

Visualizar el árbol de commits

Para tener una mejor comprensión de cómo se relacionan los diferentes *commits* en nuestro proyecto, utilicemos los diferentes comandos de visualización del *log*.

Curso

1

Lección 14: mouredev.com/git-github-14

Inicio: 01:11:54 | Duración: 00:07:37

Recordemos el comando `git log`. Nos enseñaba todo lo que había estado pasando en nuestro proyecto, como cuando creamos un primer archivo, y, según pasaba el tiempo, realizábamos varios *commits* en base a sus modificaciones. Pero, ¿y si queremos desplazar nuestro código al estado de un commit específico? Por supuesto, podemos hacerlo. Ya hemos visto cómo usar

`git checkout` para devolver el contenido de un archivo al último commit realizado.

Si escribimos el *hash* del primer commit después de `git checkout`, algo así como `git checkout 0000000`, siendo `0000000` el *hash* correspondiente, Git nos avisará de que podríamos perder los cambios que no hemos guardado antes de llevar a cabo el desplazamiento. Al hacerlo, nuestro editor indica que algunos archivos han sido eliminados, pero en realidad, nos hemos movido a un estado anterior del proyecto.

¹<https://mouredev.com/git-github-14>

```
~/Desktop/Hello Git | main ● git checkout hellogit.py
Updated 1 path from the index
~/Desktop/Hello Git | main ● git checkout cee84b47a96232900d796cd01de72fc89e7b0f7b
Note: switching to 'cee84b47a96232900d796cd01de72fc89e7b0f7b'.

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

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

  git switch -c <new-branch-name>

Or undo this operation with:

  git switch -

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

HEAD is now at cee84b4 Este es mi primer commit
~/Desktop/Hello Git | - cee84b4 |
```

Para regresar al estado actual de nuestro proyecto, podemos usar `git checkout HEAD`. Al hacerlo, nos desplazamos al extremo final de la *rama* actual. Podemos revisar en el `git log` que hemos cambiado nuestra posición en el historial de *commits*. Otra manera de movernos al final de la *rama*, es simplemente haciendo `git checkout` seguido del *hash* abreviado del último commit.

Con `git log`, podemos comprobar cómo nuestro proyecto puede identificar los *commits*, y movernos entre ellos usando solo la parte final del *hash*. Al ejecutar

`git tree` (como antes), observamos que el puntero dentro de la *rama main* está en el último commit, al igual que *HEAD*. Nuestros últimos archivos han vuelto a aparecer en nuestro proyecto *local*.



Recordemos que Git es un sistema de control de versiones distribuido. Tenemos una copia de todas las fotografías de la rama actual en nuestra máquina, pero Git las almacena de tal manera que no las vemos. Los archivos que parecían eliminados, en realidad, estaban almacenados, volviendo a aparecer cuando nos movimos al commit correspondiente.

Al desplazarnos por los *commits*, los archivos desaparecen y vuelven a aparecer dependiendo de su estado en ese momento. Esto es lo que significa trabajar con un control de versiones, y desplazarnos

siempre refiriéndonos al concepto de rama, no a una fotografía de la rama actual, como lo veremos a lo largo de este capítulo.

Capítulo 15: Reset y log de referencias \$git

reset --hard y \$git reflog

Comandos

- 1 `git reset --hard`
- 2 `git reset --hard <hash>`
- 3 `git reflog`

Conceptos

Introducción

Continuando con nuestro aprendizaje de Git, en este capítulo vamos a explorar dos nuevos comandos: `git reset --hard` y `git reflog`. Estos comandos nos permitirán manejar, aún mejor, nuestro historial de *commits*, y corregir errores en caso de ser necesario.

Reset —hard

El comando `git reset --hard` es una variante más radical del comando `git reset`. Mientras que `git reset` nos permitía retroceder en el tiempo hasta un punto específico en nuestro historial de commits, con `git reset --hard` podremos eliminar todo lo que se haya hecho después del punto de retorno que le indiquemos, incluyendo los cambios no confirmados en el área de trabajo, y los *commits* adicionales que se hayan realizado.

Es importante tener en cuenta que el comando `git reset --hard` es una operación peligrosa, ya que borra permanentemente cualquier cambio posterior al punto de reseteo.

Reflog

¿Qué sucede si nos equivocamos al realizar un `git reset --hard` y queremos recuperar los cambios perdidos? Aquí es donde aparecerá para rescatarnos el comando `git reflog`. Este comando nos muestra el historial completo de todas las acciones realizadas en nuestro *repositorio*, incluidos los *commits* que creíamos haber eliminado con

el comando `git reset --hard`.

Podemos utilizar esta lista para buscar el *hash* del commit al que queremos volver y recuperar los cambios perdidos.

Para recuperar esos cambios, simplemente buscamos en el listado el *hash* del commit al que queremos volver y ejecutamos de nuevo `git reset --hard` con ese identificador. Esto nos llevará de vuelta al punto en el que nos encontrábamos antes de ejecutar el `git reset --hard`.

Conclusiones

Es muy importante tener en cuenta que estos comandos son operaciones peligrosas que pueden tener consecuencias graves si se usan incorrectamente. Por lo tanto, es fundamental tener cuidado y asegurarse de que estamos en el punto correcto de nuestro historial antes de ejecutar cualquiera de ellos.

En general, estos comandos pueden ser muy útiles en situaciones en las que necesitamos ajustar nuestra línea de tiempo de *commits* o corregir errores en nuestro código. Con la práctica y el uso constante, podemos aprovechar al máximo las funcionalidades de Git para mejorar nuestro flujo de trabajo y colaboración en proyectos de software.

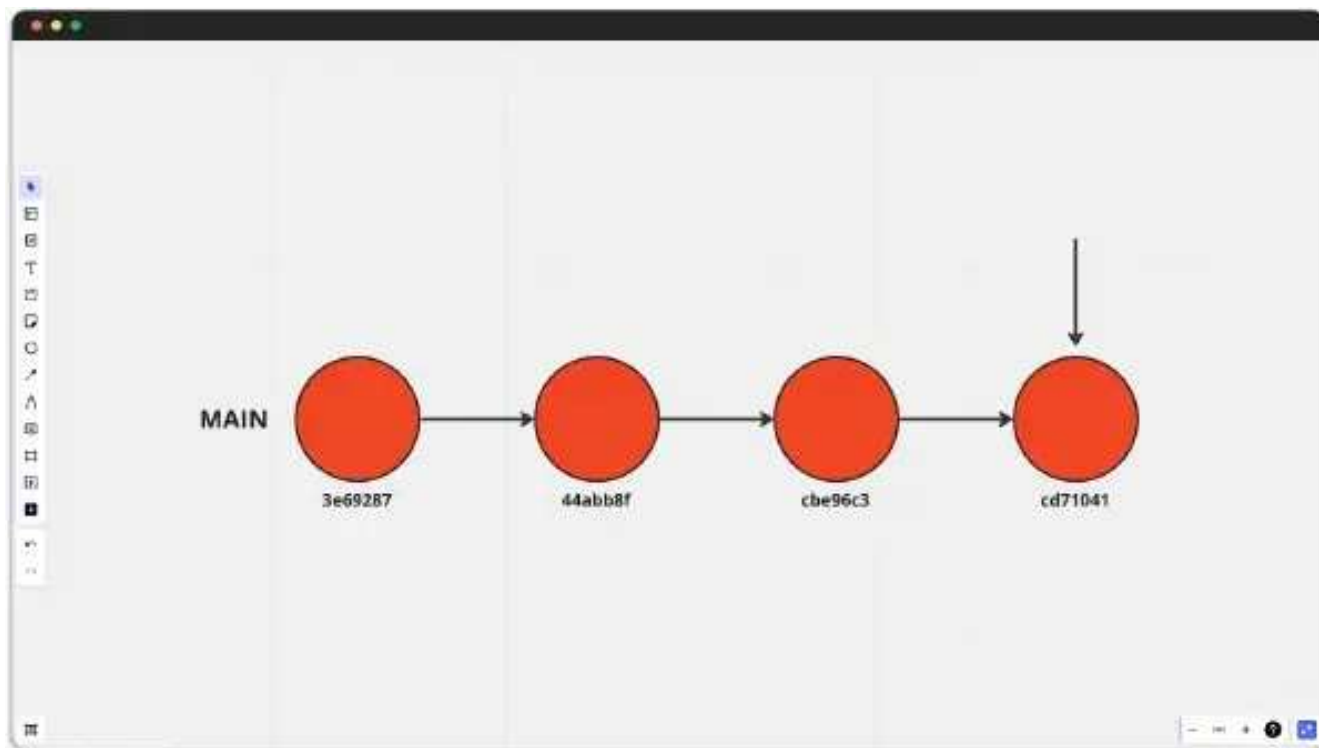
Curso

Lección 15: mouredev.com/git-github-15¹

Inicio: 01:19:31 | Duración: 00:08:06

Sigamos aprendiendo nuevos comandos de Git. En concreto, un par más. En primer lugar, recordemos lo que hacía `git reset`. Este comando nos permitía desechar cambios o movernos a un punto específico en nuestro *repositorio*, sin considerar todo lo que se hizo después.

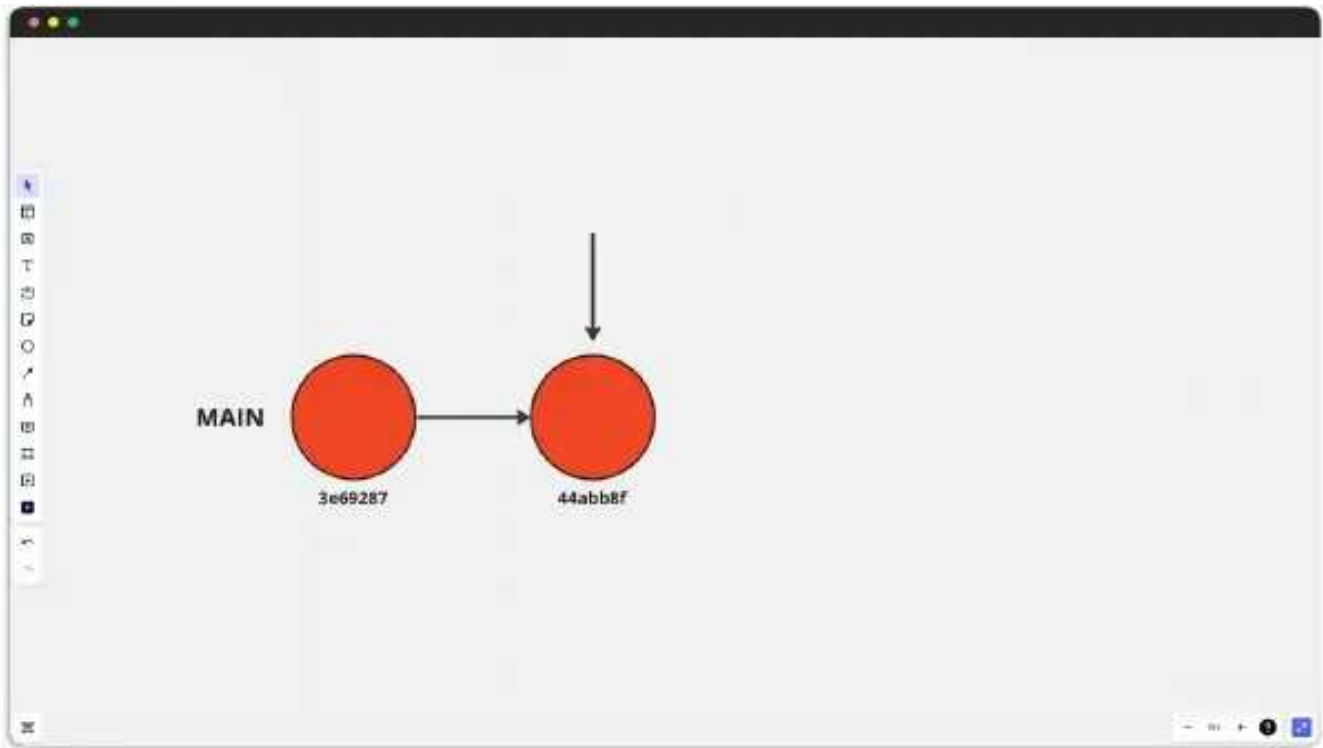
¹ <https://mouredev.com/git-github-15>



Existe una variante más destructiva del `git reset`, que añade el argumento (o *flag*) `--hard`: el `git reset --hard`. ¿Qué significa? Hemos hecho cuatro *commits* en nuestra *rama main*, cada uno con un identificador que nos permite movernos entre ellos. Hasta ahora, ya comprobamos que con `git reset` podíamos resetear cambios o posicionarnos en un punto concreto de la línea del tiempo de nuestro *repositorio*.

Supongamos que nos damos cuenta de que los dos últimos *commits* son errores, y queremos eliminarlos. En este caso, podemos usar su variante `git reset`

`--hard`. Para hacer esto escribimos `git reset --hard` seguido de la hash del commit al que queremos regresar. Al ejecutar este comando, la cabeza (*HEAD*) de nuestra *rama* se moverá al commit seleccionado, y todos los cambios posteriores desaparecerán, como si nunca hubieran existido.



Pero, ¿qué pasa si nos equivocamos al hacer un `git reset --hard` y queremos recuperar los cambios perdidos? Aquí es donde aparece un nuevo comando, el llamado `git reflog` (log de referencias). Este comando nos muestra el historial completo, repito, completo, de interacciones en nuestro *repositorio*, incluidos los *commits* que pensábamos que habíamos eliminado. Para recuperar los cambios perdidos, simplemente buscamos el *hash* del commit al que queremos regresar y ejecutamos de nuevo `git reset --hard` con ese identificador.

```
4d23cf3 (HEAD, main) HEAD@{0}: checkout: moving from cee84b47a96232900d796cd01de72fc89e7b0f7b to 4d23cf3
cee84b4 HEAD@{1}: checkout: moving from main to cee84b47a96232900d796cd01de72fc89e7b0f7b
4d23cf3 (HEAD, main) HEAD@{2}: commit: Se modifica el print
5fe844c (tag: clase_1) HEAD@{3}: commit: Se añade el .gitignore
2d2a502 HEAD@{4}: commit: Se actualiza el texto del print
437281d HEAD@{5}: commit: Este es mi segundo commit
cee84b4 HEAD@{6}: commit (initial): Este es mi primer commit
(END)
```

De esta manera, utilizando `git reset --hard` y `git reflog`, podemos movernos hacia adelante y hacia atrás en nuestro historial de `commits`. Con estos comandos, podemos corregir errores y ajustar nuestra línea del tiempo según sea necesario.

Capítulo 16: Etiquetas

\$git tag

Comandos

- 1 `git tag`
- 2 `git tag <nombre_tag>`
- 3 `git tag <nombre_tag> <hash_commit>`
- 4 `git show <nombre_tag>`
- 5 `git checkout <nombre_tag>`
- 6 `git tag -d <nombre_tag>`

Conceptos

Introducción

En desarrollo de software, es esencial mantener un historial detallado de los cambios realizados en nuestro código. Git y GitHub son herramientas ampliamente utilizadas para gestionar versiones de código y colaborar en proyectos de manera eficiente. Una de las características más útiles de Git es su capacidad de etiquetar puntos específicos en el historial de cambios de nuestro *repositorio* utilizando *tags*.

Tag o etiqueta

Un **tag** en Git es una referencia a un punto específico en el historial de cambios de nuestro repositorio. Al crear un tag, podemos darle un nombre para identificarlo fácilmente en el futuro. Los *tags* pueden ser utilizados para marcar versiones de una aplicación, o cualquier otro punto importante en el historial de cambios.

Creación

Crear un tag en Git es muy sencillo. Simplemente escribimos `git tag` seguido del nombre que queremos darle a la etiqueta. Es recomendable utilizar nombres descriptivos y fáciles de entender, preferiblemente en minúsculas y con guiones bajos.

También podemos asignar un tag a un commit concreto utilizando `git tag` seguido del nombre y su *hash*.

Visualización

Para ver una lista de todos los *tags* que hemos creado, podemos utilizar el comando `git tag` sin argumentos. Esto nos mostrará una lista de todos los tags en orden alfabético.

Si queremos ver más detalles sobre un tag específico, podemos utilizar el comando `git show` seguido del nombre del tag. Esto nos mostrará información detallada sobre el commit asociado a ese tag.

Desplazamiento

Los *tags* pueden ser utilizados para movernos rápidamente entre diferentes *commits* en nuestro historial de cambios. Para hacer esto, podemos utilizar el comando `git checkout` seguido del nombre del tag.

Eliminación

Por último, si queremos eliminar un tag que ya no necesitamos, podemos utilizar el comando `git tag -d` seguido del nombre del tag.

Es importante tener en cuenta que eliminar un tag no afecta el historial de cambios de nuestro *repositorio*, solo elimina la referencia a la etiqueta en ese punto específico.

Conclusión

Los *tags* son una herramienta muy útil en Git que nos permiten etiquetar puntos importantes en nuestro historial de cambios. Con los *tags* nos podemos desplazar rápidamente entre diferentes *commits*, y mantener un registro detallado de versiones de nuestra aplicación. Es importante utilizar nombres descriptivos y fáciles de entender para nuestros *tags*, y recordar que eliminar un tag no afecta el historial de cambios del *repositorio*.

Curso

Lección 16: mouredev.com/git-github-16¹

¹<https://mouredev.com/git-github-16>

Inicio: 01:27:37 | Duración: 00:09:59

Vamos a hablar sobre el comando `git tag`. Un tag, o etiqueta, nos permite hacer referencia a un commit específico. Los *tags* nos resultan muy útiles cuando queremos marcar puntos importantes en nuestro *repositorio*, como versiones o puestas en producción de la aplicación.

Podríamos usar un tag para marcar todo lo que hemos subido hasta un punto específico y significativo, como por ejemplo una supuesta versión *1.0* de nuestra aplicación. Para crear un tag, simplemente escribimos en la terminal `git tag` seguido del nombre del tag. Es una buena práctica utilizar minúsculas y guiones bajos en los nombres de los tags, por ejemplo: `git tag v1.0`.



Una vez creado el tag, si ejecutamos `git log`, observaremos que se ha añadido el tag a nuestro commit correspondiente. Podemos agregar tantos *tags* como queramos.

```

commit: 5fe044ce96e5ff7214008e54caad2e35059f5407 (HEAD -> main, tag: clase_1)
Author: MoureDev <braismoure@mouredev.com>
Date: Thu Mar 7 08:49:14 2024 +0100

    Se añade el .gitignore

commit: 2d2a582eb0b303a9df4095195b18e9b587678965
Author: MoureDev <braismoure@mouredev.com>
Date: Thu Mar 7 08:16:27 2024 +0100

    Se actualiza el texto del print

commit: 43f281d6683ab4ae38b7a67c7d4e26d281b59ef06
Author: MoureDev <braismoure@mouredev.com>
Date: Tue Mar 5 13:35:00 2024 +0100

    Este es mi segundo commit

commit: cee84b47a96232900d798cd01de72fc89e7b077b
Author: MoureDev <mouredev@gmail.com>
Date: Tue Mar 5 13:30:23 2024 +0100

    Este es mi primer commit
(END)

```

Para ver una lista de todos los *tags* que hemos creado, podemos usar el comando `git tag` sin argumentos. Para movernos entre diferentes *commits* usando *tags*, podemos emplear `git checkout` seguido del nombre del tag, por ejemplo: `git checkout v1.0`.

Los *tags* también nos permiten movernos rápidamente a puntos específicos de nuestro *repositorio* sin tener que buscar el *hash* del commit. Esto es especialmente útil cuando queremos solucionar errores en versiones anteriores, o trabajar sobre el código fuente de una versión específica.

Si nos hemos desplazado, otra manera de volver al final de nuestra *rama* actual es utilizando `git checkout` seguido del nombre de la *rama*, por ejemplo: `git checkout main`.

Hemos visto cómo utilizar `git tag`, y cómo movernos entre diferentes *commits* usando *tags*. Por supuesto, los *tags* también se puede eliminar usando el argumento `-d`, por ejemplo, ejecutando `git tag -d v1.0`.

Capítulo 17: Creación de ramas `$git branch` y `$git switch`

Comandos

- 1 `git branch`
- 2 `git branch <nombre_rama>`
- 3 `git switch <nombre_rama>`
- 4 `git checkout -b <nombre_rama>`
- 5 `git switch -c <nombre_rama>`

Conceptos

Introducción

Uno de los conceptos fundamentales de Git son las *ramas*, que permiten a los equipos trabajar en diferentes flujos de desarrollo de manera independiente y colaborativa. En esta lección, exploraremos más a fondo el concepto de *ramas* en Git, aprendiendo a gestionarlas con los comandos `git branch` y `git switch`.

Utilidad

Las *ramas* en Git permiten a los equipos trabajar en diferentes flujos de trabajo de manera independiente sin afectar la *rama* principal.

Creación

Para crear una nueva *rama* en Git, utilizamos el comando `git branch`, seguido del nombre de la nueva *rama*.

Desplazamiento

Para desplazarnos a una *rama* diferente, utilizamos el comando `git switch`, seguido del nombre de la *rama* a la que deseamos movernos.

También podemos crear una *rama* y desplazarnos directamente a ella. Para ello utilizamos `git checkout -b` o `git switch -c`, seguido del nombre de la nueva *rama*.

Recuerda que al cambiar de *rama*, nuestros archivos y directorios se actualizarán según el estado de la *rama* a la que nos estamos desplazando. Por lo tanto, debemos asegurarnos de guardar y hacer `commit` de nuestros cambios antes de cambiar de *rama* para evitar perder cualquier trabajo.

Diferencia entre switch y checkout

Hemos utilizado `switch` y `checkout` para desplazarnos entre *ramas*, por eso es muy habitual plantearse cuál

es la diferencia entre ambos comandos. La diferencia principal entre `git switch` y `git checkout` es que `git switch` está diseñado específicamente para cambiar entre *ramas*, mientras que *git checkout* tiene varias funciones, incluyendo la capacidad de cambiar entre *ramas*, *hash*, *tags* y *commits*.

En otras palabras, `git switch` es una instrucción más especializada enfocada exclusivamente en las *ramas*, mientras que `git checkout` contempla una utilización más general.

Además, `git switch` tiene una sintaxis más clara y fácil de entender que `git checkout`, lo que la hace más fácil de utilizar, y reduce la posibilidad de cometer errores.

Por lo tanto, siguiendo las propias recomendaciones del equipo de Git, si deseamos cambiar entre *ramas*, es recomendable utilizar *git switch* en lugar de *git checkout*.

Desarrollo

Una vez que estamos en una nueva *rama*, podemos trabajar en ella como lo haríamos en otra cualquiera. Por ejemplo, podemos crear nuevos archivos, modificar los existentes, ejecutar más comandos de Git, etc.

Conclusión

Las *ramas* en Git son una herramienta fundamental para permitir a los equipos trabajar en diferentes flujos de manera independiente y colaborativa. Los comandos `git branch` y `git switch` son esenciales para gestionar estas *ramas*.

Además, las *ramas* también pueden ser utilizadas para experimentar sin afectar la *rama* principal. Por ejemplo, podemos crear una nueva *rama* para probar una nueva funcionalidad, o hacer cambios radicales en el código fuente sin afectar a la estabilidad del proyecto.

Sin embargo, tengamos en cuenta que, tener demasiadas *ramas*, puede hacer que la gestión del proyecto sea más compleja. Por lo tanto, es recomendable mantener un número limitado de *ramas* y eliminar las que ya no son necesarias.

El uso de *ramas* en Git es una práctica esencial para el desarrollo de proyectos colaborativos de manera eficiente y organizada. Ya, por último, nombrar que las *ramas* también se puede fusionar, integrando los cambios y funcionalidades de ambas en una única, todo esto sin afectar la estabilidad del proyecto. Hablaremos más del concepto de fusión en los capítulos siguientes.

Curso

Lección 17: mouredev.com/git-github-17¹

Inicio: 01:37:36 | Duración: 00:11:30

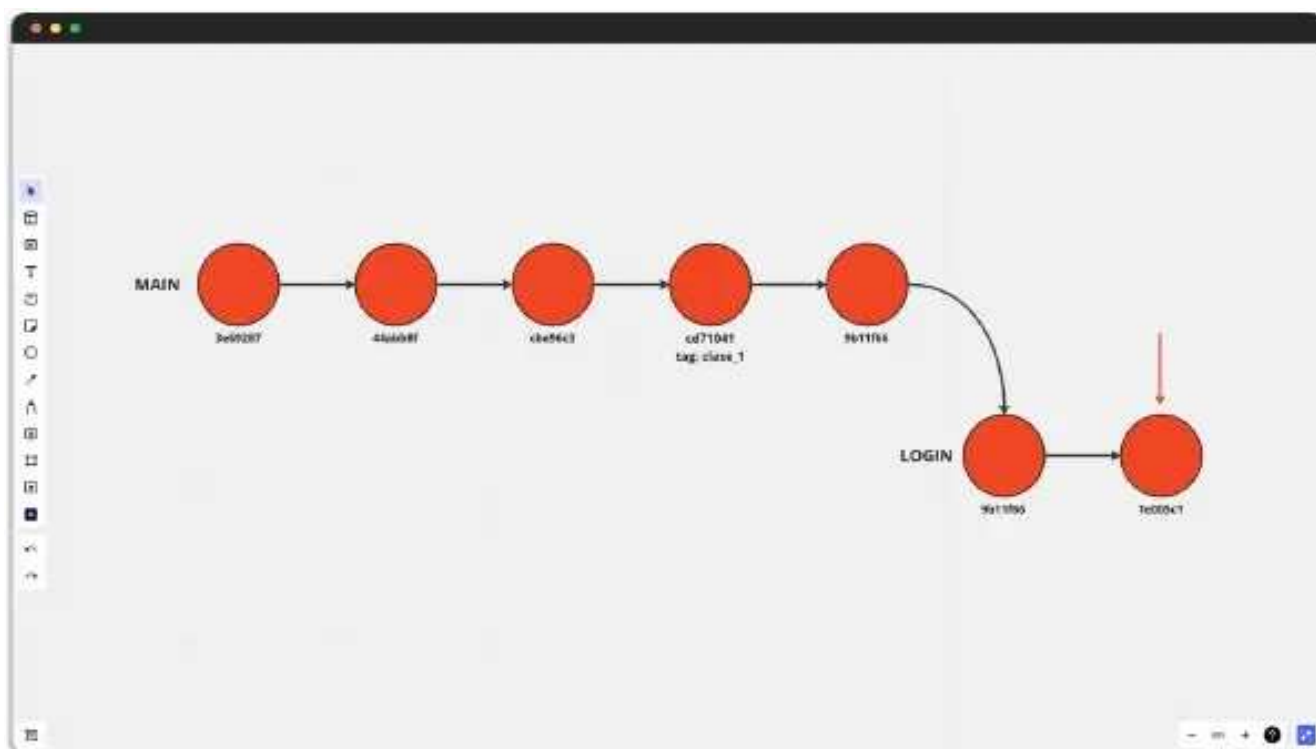
En esta lección, vamos a explorar el concepto principal de Git: las *ramas*. Hasta ahora, hemos trabajado en una sola línea del tiempo, la llamada *rama* principal o *main*. Git permite crear ramificaciones, y, para gestionarlasy, usaremos el comando `git branch`.

¿Por qué usar *ramas*? Imaginemos que queremos trabajar en una nueva funcionalidad, por ejemplo, cómo agregar

¹<https://mouredev.com/git-github-17>

un inicio de sesión a nuestra aplicación ficticia, claro está, sin afectar la *rama* principal. Para ello, crearemos una *rama* llamada *login*, para así trabajar en ella de forma independiente.

¿Cómo creamos ramas? Únicamente ejecutamos el comando `git branch` seguido del nombre de la *rama*, en este caso, `git branch login`. Esta ejecución creará la *rama* *login*, pero, a pesar de haber creado la nueva *rama*, aún nos encontramos situados en la *rama* *main*.



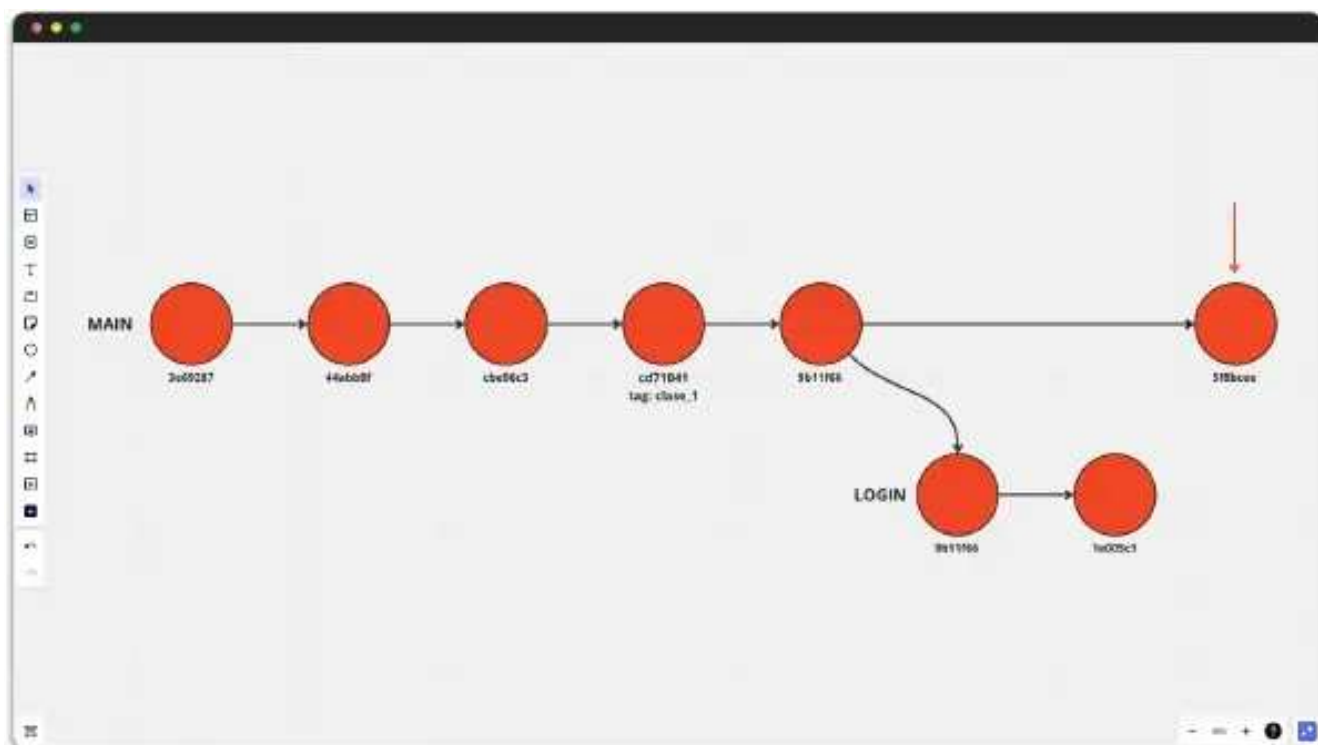
Para desplazarnos a la nueva *rama*, utilizaremos un nuevo comando, `git switch`, como siempre, seguido del nombre de la *rama* a en la que nos queremos situar, en este caso, `git switch login`. Ya nos encontramos en la *rama* *login*, pudiendo así comenzar a trabajar en nuestra nueva funcionalidad sin interferir con el contenido y evolución de la *rama* principal. Por ejemplo, vamos a crear un nuevo archivo llamado *login.py*, y a añadirle los cambios que deseemos, como un `print` de prueba. A continuación, ejecutamos nuestra combinación habitual de `git add` y `git commit` para guardar estos cambios en la *rama* *login*.

```

~/Desktop/Hello Git main git branch login
~/Desktop/Hello Git main git tree
~/Desktop/Hello Git main git switch login
Switched to branch 'login'
~/Desktop/Hello Git login

```

Para regresar a la *rama main*, ejecutaremos `git switch main`. Ya en la *rama main*, observaremos que no aparecen los cambios realizados en la *rama login*. Así, cada *rama* funciona de manera independiente, permitiéndonos trabajar en diferentes funcionalidades sin afectarse entre sí.



Las *ramas* se pueden dividir aún más, lo que permite a los equipos colaborar autónomamente y de maneras infinitas. Cuando las funcionalidades estén listas, podrán

integrarse en la *rama* principal (o en la *rama* que se desee).

En resumen, las *ramas* en Git permiten trabajar en diferentes flujos de trabajo de manera independiente y colaborativa. Los comandos `git branch` (junto con sus variaciones) y `git switch` son fundamentales para gestionar estas *ramas*.

Capítulo 18: Combinación de ramas con `git merge`

Comandos

```
1 git merge <nombre_rama>
```

Conceptos

Introducción

Como ya hemos dicho, una de las características más importantes de Git es su capacidad para gestionar flujos de trabajo en base a *ramas*.

Las *ramas* son copias independientes del código base, lo que nos permite trabajar en diferentes características o funcionalidades sin afectar al código principal. Sin embargo, en algún momento será necesario integrar el trabajo realizado en una *rama* en otra, para mantener así la coherencia y la compatibilidad entre las diferentes versiones, y continuar evolucionando el proyecto.

Merge

Git nos ofrece el comando `git merge` para *fusionar* los cambios realizados en una *rama* con otra. Este comando

toma los cambios realizados en una *rama* y los aplica a otra, creando un nuevo commit que combina ambos historiales.

Para realizar un merge, es necesario estar situados en la *rama* de destino y ejecutar el comando `git merge`, seguido del nombre de la *rama* que se desea *fusionar*. Es importante tener en cuenta que antes de realizar un merge, se deben resolver los *conflictos* que puedan surgir si hay cambios en ambas *ramas*, cambios que afecten a un mismo archivo. El concepto de *conflicto* lo veremos detallado en el siguiente capítulo.

Una vez que se realiza un merge, es posible comprobar que los cambios se han aplicado correctamente en la *rama* de destino utilizando los comandos `git status` o `git log`. Es importante recordar que, al realizar un merge, se crea un nuevo commit que combina el historial de ambas *ramas*, lo que significa que la *rama* de destino avanzará en el tiempo más allá del último commit realizado en la *rama* que se fusionó.

Conclusión

El comando `git merge` es esencial para mantener la coherencia y la compatibilidad entre diferentes versiones del código en un proyecto colaborativo (o individual). A través de la *fusión* de *ramas*, es posible integrar el trabajo realizado en una versión del código. Siempre es importante asegurarnos de resolver cualquier *conflicto* antes de realizar un merge, y comprobar que los cambios se hayan aplicado correctamente en la *rama* de destino.

Curso

Lección 18: mouredev.com/git-github-18¹

Inicio: 01:49:06 | Duración: 00:05:23

Imaginemos que otro equipo de desarrollo ha seguido trabajando en la *rama main*, llegando un momento en el que queremos saber qué ha estado haciendo dicho equipo. Puede que hayan pasado horas, o incluso días desde que comenzaron a trabajar, por lo que nos interesa asegurarnos de que, lo que han implementado en la *rama main*, sigue siendo compatible con lo que tenemos en nuestra *rama login* (que creamos en la lección anterior). Básicamente, necesitamos mantener ambos flujos actualizados.

Lo que buscamos es aplicar los cambios de la *rama main* en nuestra *rama login*. Así comprobaremos que ambos desarrollos no han provocado ningún efecto colateral al ejecutar sus funcionalidades de manera conjunta. Para ello existe el comando `git merge`.

Podemos definir `merge` como la acción de *fusionar* o *combinar* cambios entre *ramas*. Si nos encontramos en la *rama login*, y queremos añadir los cambios de la *rama main*, únicamente ejecutaremos `git merge main`.

¹ <https://mouredev.com/git-github-18>

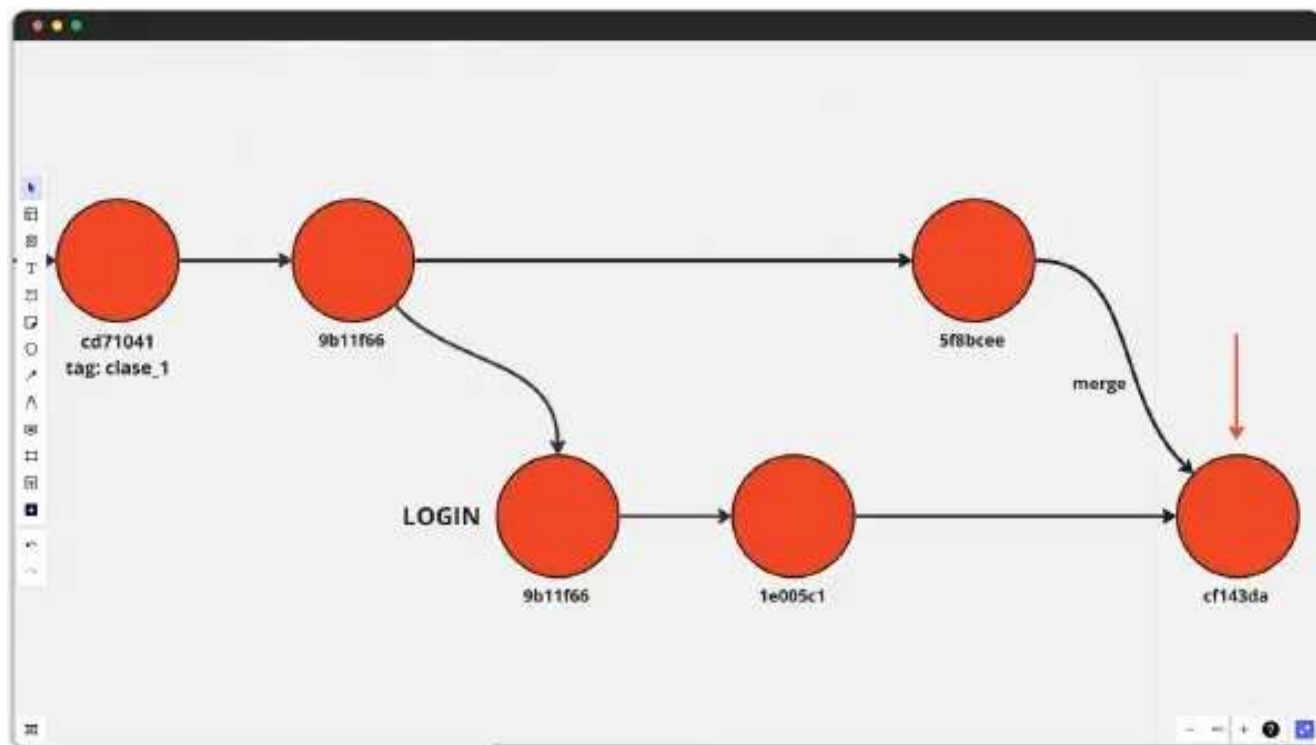
```

~/Desktop/Hello Git  main git branch login
~/Desktop/Hello Git  main git tree
~/Desktop/Hello Git  main git switch login
Switched to branch 'login'
~/Desktop/Hello Git  login git tree
~/Desktop/Hello Git  login git log
~/Desktop/Hello Git  login git status
On branch login
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    login.py

nothing added to commit but untracked files present (use "git add" to track)
~/Desktop/Hello Git  login git add .
~/Desktop/Hello Git  login git commit -m "Login"
[login d997940] Login
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 login.py
~/Desktop/Hello Git  login git status
On branch login
nothing to commit, working tree clean
~/Desktop/Hello Git  login git switch main
Switched to branch 'main'
~/Desktop/Hello Git  main git add .
~/Desktop/Hello Git  main git commit -m "Git 3 v2"
[main 50c9d78] Git 3 v2
1 file changed, 1 insertion(+)
~/Desktop/Hello Git  main git switch login
Switched to branch 'login'
~/Desktop/Hello Git  login git merge main

```

Al hacer esto, observaremos que la *rama login* ha podido actualizar su contenido, siempre que existan modificaciones en *main* posteriores a la creación de *login*. Esos cambios de *main*, ahora también los tenemos en *login*. Para comprobarlo, podemos usar `git log`, observando un commit que nos dice *merge branch main into login*, lo que nos indica que hemos combinado el contenido de la *rama main* dentro de *login*, confirmándonos que dicho proceso de *fusión* ha funcionado correctamente.



Hecho esto, la *rama login* ha avanzado en el tiempo más allá del último commit que se hizo en *main*. Cada cambio en el historial implica un nuevo commit, por lo que este proceso de combinación ha generado un nuevo identificador *hash*. Lo que acabamos de hacer es el proceso básico asociado a un merge entre *ramas*, logrando así mantener la coherencia entre ambas.

Capítulo 19: Conflictos

Comandos

- 1 `git diff`
- 2 `git merge --theirs <archivo>`
- 3 `git merge --mine (o --ours) <archivo>`

Conceptos

Introducción

Recordemos una vez más que una de las funcionalidades principales de Git es permitir que varios equipos trabajen en un mismo proyecto, coordinando sus cambios y asegurándose de que el resultado final sea coherente y funcional. Todo esto, como vimos, gracias a las *ramas* y los merge.

Sin embargo, el trabajo con *ramas* puede presentar algunos desafíos, especialmente cuando los cambios de dos o más *ramas* entran en *conflicto*, y dicho *conflicto* debe ser resuelto para poder continuar trabajando. En este capítulo, vamos a explorar en detalle qué es un *conflicto* en Git, cómo se produce, y cómo podemos solucionarlo.

Definición

Un **conflicto** en Git ocurre cuando dos *ramas* han modificado el mismo archivo y Git no puede decidir qué cambio es el correcto. Esto puede ocurrir, por ejemplo, cuando dos equipos están trabajando en diferentes funcionalidades de un proyecto, pero ambos necesitan modificar un mismo archivo para lograr sus objetivos.

El *conflicto* se producirá cuando Git intenta *fusionar* las diferentes *ramas* y detecta que existen actualizaciones en el mismo archivo, y, en concreto, en la misma línea. En lugar de elegir un cambio por encima del otro, Git nos indica que existe un *conflicto*, y nos pide que lo resolvamos manualmente.

Solución

Resolver un *conflicto* en Git implica revisar los cambios que han hecho las diferentes *ramas*, y decidir cuál es el correcto. Para ello, Git nos muestra una vista de los cambios en confrontación, indicando las diferentes partes que han sido modificadas en cada *rama*.

Para solucionar el *conflicto*, debemos editar manualmente el archivo en problemas, y elegir qué cambios queremos mantener, o incluso combinar. Esto implica revisar cuidadosamente los cambios y decidir cuáles son necesarios para nuestro proyecto, siempre teniendo en cuenta si esa decisión puede afectar negativamente a la ejecución en la otra *rama*. Una vez que hayamos tomado una decisión, debemos guardar el archivo y realizar un commit para confirmar los cambios.

Existen varias maneras de solucionar un *conflicto* en Git, pero una de las más comunes es utilizando el comando

`git merge`. Este comando *fusiona* dos *ramas*, y si existe un *conflicto*, nos permite revisar los cambios y aportar una solución manualmente. También podemos utilizar el comando `git diff` para comparar los cambios entre dos *ramas* y detectar posibles problemas.

Proceso

Si hay *conflictos*, Git nos informará de que ha habido un problema al realizar la combinación de las *ramas*, y mostrará los archivos que se han visto implicados.

- Tenemos que abrir el archivo en *conflicto* y resolver los problemas manualmente. Git nos mostrará los conflictos en cada archivo con el siguiente formato.

```
1 <<<<<< HEAD
2 <codigo_rama_local>
3 =====
4 <codigo_rama_a_fusionar>
5 >>>>>> <nombre_rama_a_combinar>
```

- Editaremos manualmente el archivo para dejar únicamente la versión correcta, o combinaremos fragmentos de ambos.
- Después de solucionar los conflictos, añadiremos los cambios a Git con el comando `git add <archivo>`. Este comando marca el *conflicto* del archivo como solucionado y listo para ser confirmado.
- Resueltos los problemas, pasaremos a la fase confirmación utilizando `git commit -m "<mensaje>"`.

Por otra parte, también existe un mecanismo para seleccionar directamente las modificaciones de nuestra *rama*, o las de la que queremos combinar. Todo esto sin tener que editar y corregir el fichero en *conflicto* de forma manual. Para ello utilizaremos `git merge --theirs <archivo>` y `git merge --mine (o --ours) <archivo>`, opciones que nos permiten especificar qué versión de un archivo en *conflicto* se debe conservar durante el proceso de combinación.

- `git merge --theirs <archivo>`: Esta opción conserva los cambios de la *rama* que se está fusionando (rama a combinar), y descarta los cambios de la *rama local* (rama actual). Es decir, toma *su versión* (theirs) en lugar de la *nuestra* (mine o ours).
- `git merge --mine (o --ours) <archivo>`: Esta opción conserva los cambios de la *rama local* (rama actual), y descarta los cambios de la *rama* que se está fusionando (rama a combinar). Es decir, toma *nuestra versión* (mine o ours) en lugar de *su versión* (theirs).

Es importante tener en cuenta que estas opciones deben ser usadas con precaución, ya que pueden descartar cambios importantes en algunos casos.

Por lo general, en caso de duda, la mejor práctica es resolver los *conflictos* manualmente, revisando cada archivo en *conflicto*, y decidiendo qué versión conservar. De esta manera, nos aseguramos de que los cambios más importantes sean incluidos en la *rama* final.

Conclusión

Para evitar *conflictos* innecesarios, es recomendable que los equipos de desarrollo se coordinen y trabajen en diferentes partes del proyecto, minimizando así la cantidad de archivos que se modifican al mismo tiempo. Si un *conflicto* aparece, es crucial que los equipos mantengan una comunicación activa para resolverlo de manera efectiva, asegurando el avance del proyecto sin problemas.

Curso

Lección 19: mouredev.com/git-github-19¹

Inicio: 01:54:29 | Duración: 00:09:13

Llegamos a una parte donde se puede complicar el proceso. Los merge que funcionan correctamente, no presentan ningún problema, pero, *¿qué pasa con los que no funcionan?* Vamos a analizar uno de los grandes dolores de cabeza cuando comenzamos a trabajar con Git y sus *ramas*, hablamos del concepto de **conflicto**.

Haremos algo. En un fichero *hellogit3.py*, ya existente en la *rama main*, y estando situado en la *rama login*, decidimos efectuar algún cambio en él, realizando un commit del mismo.

Como equipo asignado a *login*, hemos modificado el archivo *hellogit3.py*, que no debería ser competencia de nuestro desarrollo. Nos desplazamos a la *rama main* con `git switch`, y, por supuesto, observamos que ha

¹ <https://mouredev.com/git-github-19>

desaparecido el fichero *login.py*, ya que la *rama main* no tiene conocimiento de lo que se está haciendo en la *rama login*. Recordemos que anteriormente *fusionamos* (merge) el contenido de *main* en *login*. Por ello, *login* tiene los cambios de *main*, pero *main* no los de *login*.

Imaginemos que el equipo de *main* ha tenido que editar también a posteriori el fichero *hellogit3.py*. Ese equipo sí que estaba trabajando habitualmente en este archivo. Supongamos que era una funcionalidad que estaban implementando. Hacemos `commit` de ese cambio en *main*.

Ahoramismo, esto ha derivado en que tenemos un nuevo `commit` en las *ramas main* y *login*.

Desplacémonos de nuevo a la *rama login*.

Recuerda que usamos `git switch` como buena práctica para cambiar de *rama*. El comando `git checkout` nos servía para más cosas.

Vamos a analizar el siguiente punto. Queremos hacer exactamente lo mismo que antes. Hemos continuado trabajando en el *login*. De nuevo, queremos comprobar cómo está la *rama main* y combinar sus cambios en la *rama login*. Hacemos `git merge main`, acción que no ha funcionado, ya que ha aparecido un *conflicto* en el archivo *hellogit3.py*.

A terminal window with a dark background. The prompt is ~/Desktop/Hello Git login. The command git merge main has been executed. The output shows 'Auto-merging hellogit3.py' followed by a conflict message: 'CONFLICT (content): Merge conflict in hellogit3.py' and 'Automatic merge failed; fix conflicts and then commit the result.' The prompt is now ~/Desktop/Hello Git login ++>M< with a cursor at the end.

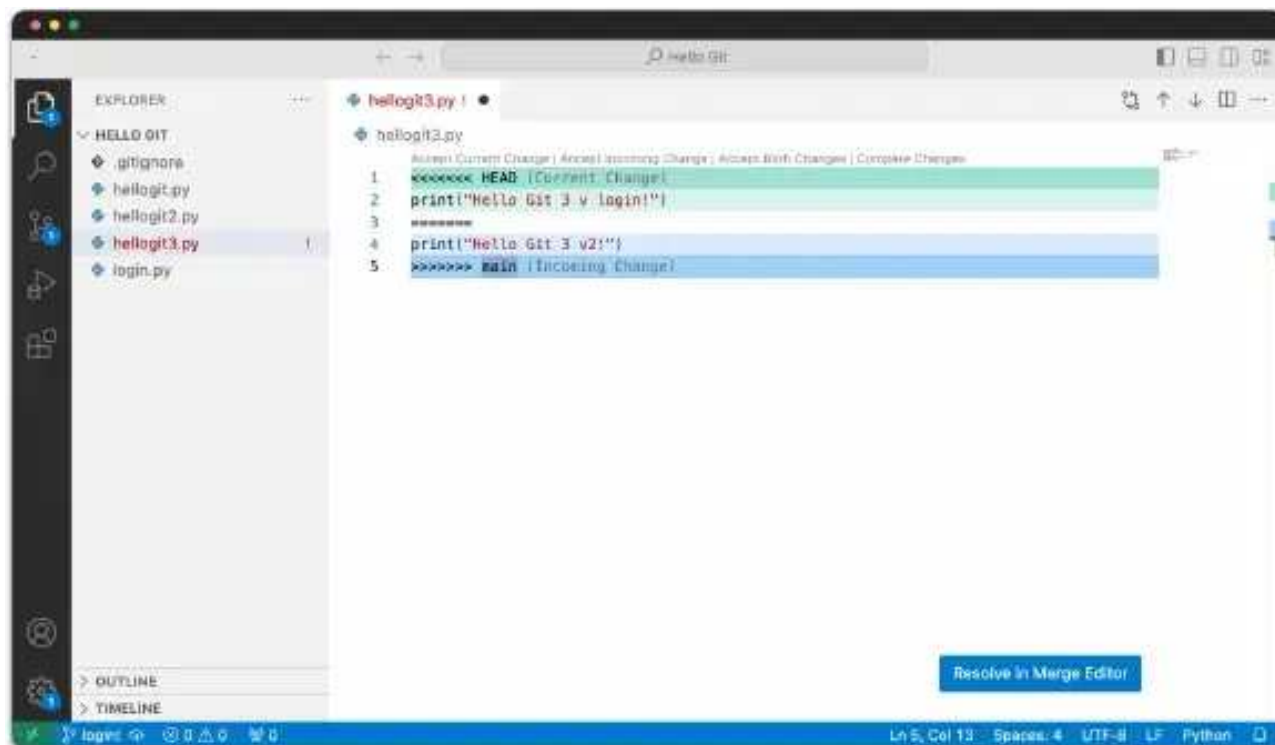
```
~/Desktop/Hello Git login git merge main
Auto-merging hellogit3.py
CONFLICT (content): Merge conflict in hellogit3.py
Automatic merge failed; fix conflicts and then commit the result.
~/Desktop/Hello Git login ++>M< █
```

¿Por qué? Porque tanto la gente que está en la *rama main*, como el equipo de la *rama login*, han tocado el mismo archivo en la misma línea de código.

Aquí siempre nos asalta la pregunta: *¿Qué haces modificando mi código? ¿Por qué?*

Git es un sistema muy listo, y no le gusta complicarse la vida, por lo nos presenta la siguiente situación: *Aquí existe un conflicto. Como sistema de control de versiones no tengo ni idea de si el código que debe permanecer es el de main, o el de login. Por lo tanto, no te voy a permitir hacer un merge mientras no llegues a un acuerdo.*

Si se modificaran líneas diferentes del código en el mismo archivo, el merge podría funcionar en la mayor parte de los casos. Pero en nuestro ejemplo se ha modificado la misma línea de código.



¿En qué se traduce todo esto? En que debemos editar el archivo `hellogit3.py`. Una vez abierta el editor nos ha resaltado las dos posibilidades. Nos ha dicho: *aquí hay conflictos, y mientras existan no puedes continuar con el proceso de merge.*

Dicho de otra forma, no podemos hacer un commit de este código. Debemos aclararnos y solucionarlo. ¿Con qué nos queremos quedar? ¿Con lo que hemos hecho nosotros en *login*, o con lo que se ha hecho por parte del otro equipo en la *rama main*?

Supongamos que, como equipo trabajando en la *rama login*, nos damos cuenta de lo siguiente: *este es un archivo del otro equipo. No teníamos que haberlo modificado.*

Finalmente, nos hemos dado cuenta de que lo hemos hecho mal (obviamente, las posibilidades en estos casos son infinitas). Nos vamos a quedar con lo que había desarrollado el equipo que trabajaba directamente desde *main*.

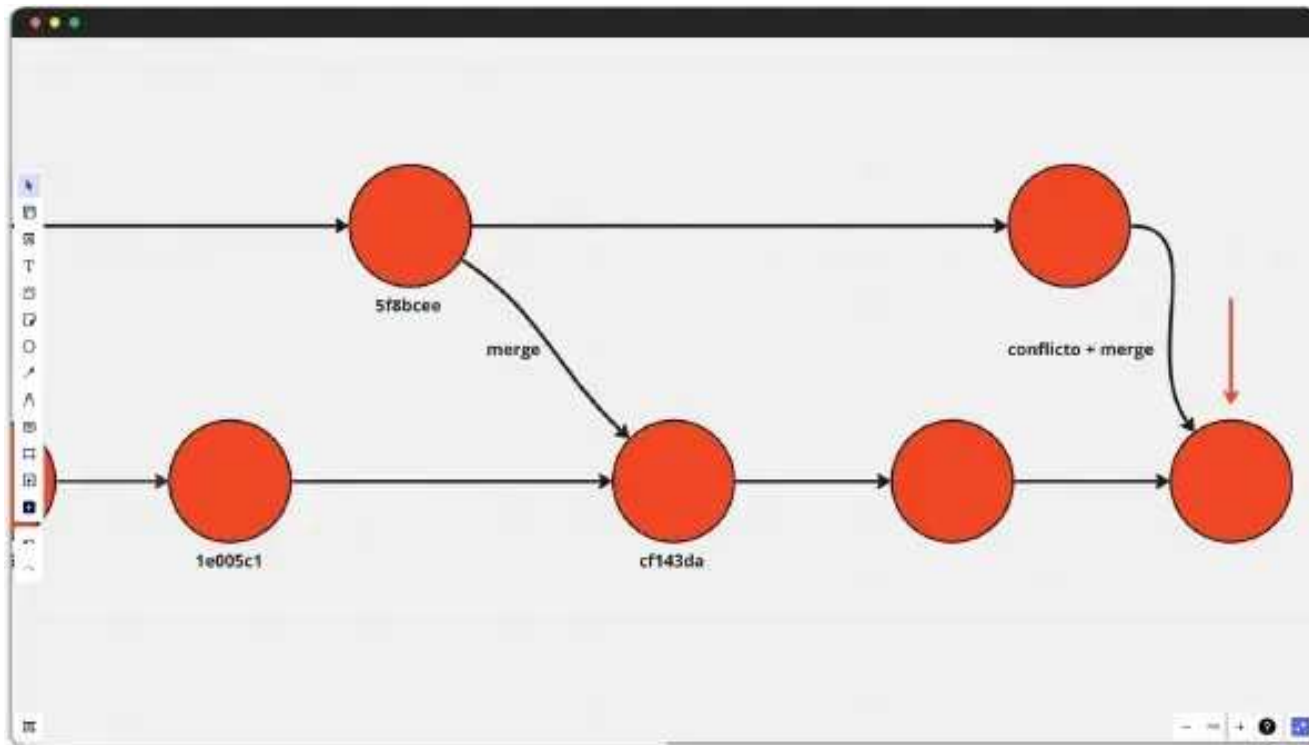
¿Cómo acabaremos resolviendo este conflicto? Podemos hacerlo directamente utilizando el comando `git`

`checkout --theirs`, ya que hemos decidido indicarle que nos queremos quedar con los cambios de la otra *rama*, no los nuestros. En caso contrario, utilizaríamos el parámetro `--mine` (o `--ours`), en vez de `--theirs`. Git es muy extenso, por lo que podríamos hacerlo de más maneras. Incluso de manera manual, retocando el código del archivo en caso de tener que combinar el código de ambos equipos (ya que ambos podrían haber modificado el archivo de manera totalmente lícita y necesaria para sus intereses).

Lo más importante de este proceso es que entendamos los fundamentos. Qué es lo que está pasando, y qué pasos debemos seguir para alcanzar una solución. Estos pasos, a veces son muy simples y rápidos, y otras veces deberemos dedicarle todo el tiempo que sea necesario. Aún así, Git es un sistema pensado para que existan los menores *conflictos* posibles. En este ejemplo hemos forzado el *conflicto* para poder explorar el proceso de corrección.

Habitualmente, si las tareas de los equipos están bien organizadas, no deberían existir demasiados *conflictos*, y, en el caso de aparecer alguno, tendrían que resolverse sin muchas complicaciones.

Solucionada la colisión en el fichero, *¿qué es lo siguiente que tenemos que hacer para acabar solucionando el conflicto?* Pues bien, vamos a intentar realizar un `commit` del archivo, con su correspondiente mensaje. Al lanzarlo, observaremos que no funciona. Git nos dice: *¿por qué quieres hacer un commit?* Revisemos con `git status`, descubriendo que, al modificar de nuevo el archivo para solucionar el *conflicto*, debemos realizar primeramente un `git add` de este. Lo añadimos. Ahora sí que podremos realizar el `git commit`.



Vamos a revisar el proceso. Ya no se nos indica ningún problema. Al ejecutar de nuevo `git status`, parece que todo está correcto. Si hacemos un `git log`, ya aparecerá la corrección del *conflicto*. Por fin nos hemos puesto de acuerdo. Volvemos a estar totalmente actualizados.

Hemos aprendido a enfrentarnos a *conflictos* en Git cuando trabajamos con *ramas*, y cómo solucionarlos de manera efectiva. Lo importante es entender qué cambios se han hecho en cada *rama* y coordinarnos entre equipos para evitar *conflictos* innecesarios. Cuando se presentan *conflictos*, es crucial saber cómo abordarlos, resolverlos,

y hacer un `commit` con la solución, para que así el trabajo en equipo pueda continuar, evitando futuros problemas. Git es una herramienta muy potente a la hora de gestionar colaboraciones, siempre y cuando se utilice de manera adecuada y siguiendo buenas prácticas.

Capítulo 20: Cambios temporales `$git stash`

Comandos

- 1 `git stash`
- 2 `git stash pop`
- 3 `git stash apply`
- 4 `git stash list`
- 5 `git stash drop <stash>`
- 6 `git stash clear`

Conceptos

Introducción

Continuando con el contexto de trabajo sobre *ramas*, en ocasiones puede ocurrir que necesitemos desplazarnos entre ellas mientras estamos trabajando en una tarea específica, pero aún no queramos hacer commit de los cambios realizados. Para estos casos, Git nos ofrece el comando `stash`, que nos permite guardar temporalmente las modificaciones en una *rama* sin tener que hacer commit.

Stash

Es un comando de Git que nos permite guardar temporalmente los cambios que hemos realizado en un archivo, o conjunto de archivos, sin tener que hacer commit. Cuando utilizamos stash, Git guarda una instantánea de los archivos modificados y los almacena en una pila, para que podamos trabajar en otra *rama* sin perder nuestro progreso. Los cambios guardados con stash se pueden aplicar posteriormente en la misma *rama*, o en otra diferente.

Utilización

Para utilizar stash, debemos seguir los siguientes pasos:

- Realizamos cambios en la *rama* actual. Para guardar temporalmente nuestros cambios sin hacer commit, debemos ejecutar `git stash`.
- Una vez nos hemos desplazado a la nueva *rama*, podemos realizar los cambios necesarios en los archivos correspondientes. Ya de vuelta a la *rama* en la que nos encontrábamos trabajando, podemos aplicar los cambios guardados previamente con stash, y continuar trabajando en ellos. Los recuperamos usando `git stash pop`. Este comando aplicará los cambios guardados con stash y los eliminará de la pila.
- Si preferimos recuperar, y mantener los cambios en la pila, podemos utilizar el comando `git stash apply`.

Es importante tener en cuenta que, si hemos guardado varios cambios con stash, debemos aplicarlos en el

orden inverso al que los hemos guardado. Es decir, el último conjunto de cambios guardado será el primero en ser aplicado.

Gestión

En ocasiones, puede ser útil visualizar qué conjunto de cambios hemos guardado en la pila de `stash`. Para ello, podemos utilizar el comando `git stash list`. Cada conjunto de cambios estará identificado por un nombre y un mensaje descriptivo.

Si decidimos que ya no necesitamos los cambios guardados en un conjunto de `stash`, podemos eliminarlo de la pila con el comando `git stash drop <stash>`. Este comando eliminará el conjunto de cambios asociados al nombre del `stash`.

Utilizando `git stash clear` limpiaremos la pila completa de `stash`. Simplemente para que lo tengamos en cuenta, podríamos llegar a recuperar `stash` previamente eliminados.

Conclusión

En resumen, `stash` es un mecanismo muy útil de Git que nos sirve para guardar temporalmente los cambios realizados en una *rama* sin tener que hacer `commit`. Esto nos permite desplazarnos entre *ramas*, o realizar otras tareas temporales sin perder nuestro progreso en la *rama* actual. Con `stash`, podemos trabajar de manera más eficiente y ordenada en nuestro proyecto, y recuperar nuestros cambios guardados cuando lo consideremos oportuno.

Curso

1

Lección 20: mouredev.com/git-github-20

Inicio: 02:03:42 | Duración: 00:06:29

Para esta lección, imaginemos que nos encontramos trabajando en la *rama login*, haciendo cambios en el archivo *login.py*. De repente, nos piden que arreglemos un error urgente en la *rama main*. Por supuesto, no queremos perder el trabajo de la *rama login*, ya que todavía no están finalizados como para hacer un commit con ellos. Para resolver esta situación tenemos el comando `stash`.

`git stash` nos permite guardar de forma temporal nuestros cambios sin hacer commit, así podremos cambiar de *rama* sin perder nuestro avance. Para guardar nuestros cambios en un stash simplemente ejecutamos `git stash`.



```
~/Desktop/Hello Git login git switch main
error: Your local changes to the following files would be overwritten by checkout:
  login.py
Please commit your changes or stash them before you switch branches.
Aborting
~/Desktop/Hello Git login git stash
```

¹<https://mouredev.com/git-github-20>

Una vez hecho esto, ya podemos cambiar a la *rama main* sin problemas con `git switch main`. Realizamos las correcciones necesarias en la *rama main*, y, cuando terminamos, volvemos a la *rama login* utilizando `git switch login`.

Para recuperar los cambios que guardamos en el stash utilizamos el comando `git stash pop`. Esto aplicará los cambios del stash y los eliminará de la lista. Si preferimos mantenerlos en la lista, podemos utilizar `git stash apply`.



```
~/Desktop/Hello Git login ● git checkout main
error: Your local changes to the following files would be overwritten by checkout:
login.py
Please commit your changes or stash them before you switch branches.
Aborting
~/Desktop/Hello Git login ● git stash
Saved working directory and index state WIP on login: 81d4b80 Corrección conflicto
~/Desktop/Hello Git login ● git stash list
~/Desktop/Hello Git login ● git switch main
Switched to branch 'main'
~/Desktop/Hello Git main ● git switch login
Switched to branch 'login'
~/Desktop/Hello Git login ● git switch main
Switched to branch 'main'
~/Desktop/Hello Git main ● git stash list
~/Desktop/Hello Git main ● git switch login
Switched to branch 'login'
~/Desktop/Hello Git login ● git stash list
~/Desktop/Hello Git login ● git stash pop
On branch login
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:   login.py

no changes added to commit (use "git add" and/or "git commit -a")
Dropped refs/stash@{0} (89bfd1782c1df4e0db26e3c97ec747aa71f25011)
~/Desktop/Hello Git login ●
```

En cualquier momento podremos revisar la lista de nuestros stash con el comando `git stash list`.

Si al final decidimos que no queremos utilizar los cambios que guardamos en un stash, será posible eliminarlos uno por uno con el comando `git stash drop <stash>`. También podemos eliminar todos utilizando `git stash clear`.

Capítulo 21 Reintegración de ramas

Comandos

- 1 `git diff <nombre_rama_a_reintegrar>`
- 2 `git merge <nombre_rama_a_reintegrar>`

Conceptos

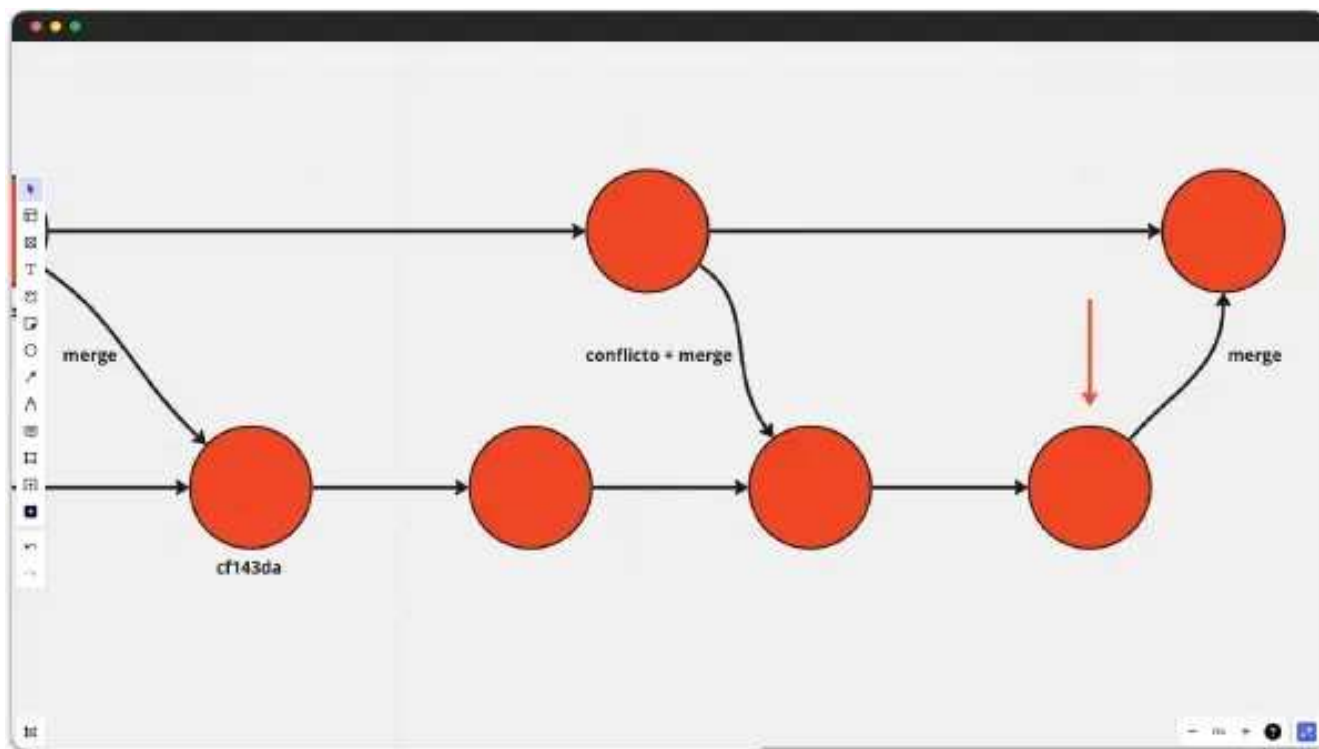
Introducción

El proceso descrito en este capítulo es fundamental para un correcto manejo de *ramas* en Git. Al trabajar en proyectos que evolucionan constantemente, es común que se utilicen diferentes *ramas* para desarrollar funcionalidades, correcciones de errores y mejoras.

Reintegración

Supongamos que hemos trabajado en una funcionalidad concreta dentro de una *rama*, llegando el momento de integrar este código con el de otra *rama* del proyecto, para así poder hacer uso de este desarrollo. Primero, necesitamos cambiar a la *rama* donde queremos añadir el nuevo código.

Una vez que ya estamos situados en dicha *rama*, usaremos el comando `git diff <nombre_rama_a_reintegrar>` para comparar los cambios entre ambas *ramas*. Si hay diferencias, podemos usar el comando `git merge <nombre_rama_a_reintegrar>` para agregar los cambios de la *rama* en la que hemos estado trabajando dentro de la *rama* en la que los queremos reintegrar. Este es el proceso de **merge**, reintegración o *fusión* en Git del que ya hemos hablado anteriormente, pero que en este caso aplica al proceso de evolución seguro de nuestro proyecto.



Después de completar el merge podemos usar el comando `git status` para asegurarnos de que todo esté correctamente integrado.

Conclusión

Emplear *ramas* en Git nos permite organizar nuestro trabajo y colaborar de manera efectiva. Al aprender a trabajar con diferentes *ramas* y *fusionar* cambios,