

Control de versiones

Con git y GitHub

Introducción

Atrás quedaron las imágenes del solitario “súper programador”, encapuchado y en la oscuridad. Hoy en día, la programación es una actividad que se desarrolla con la participación de múltiples personas. Pero, ¿cómo coordinar el trabajo del equipo? ¿Cómo hacer que los aportes individuales no se pierdan? ¿Cómo podemos hacer para experimentar sin riesgos de romper aquello que estamos desarrollando?

El control de versiones es la piedra angular del trabajo en equipo, de forma que todo el equipo pueda trabajar sobre una única estructura de proyecto

Puesto de manera formal, un sistema de control de versiones es una combinación de tecnologías y prácticas para seguir y controlar los cambios realizados en los archivos de un proyecto, en particular en el código fuente y la documentación.

Esta es una de las partes esenciales del desarrollo de software moderno y su aplicación es valiosa tanto para equipos de trabajo como para usuarios individuales.

Git es una herramienta compleja y complicada, y en términos simples ofrece una forma de desarrollar en equipo sin perder rastro de la evolución de un programa. Aunque también sea de un alto valor para desarrolladores en solitario.

Dada la complejidad de la herramienta, no veremos muchas funciones interesantes, solo para que esto sea una introducción suave y guiada.

La razón por la cual el control de versiones es universal es porque ayuda virtualmente en todos los aspectos al dirigir un proyecto: comunicación entre los desarrolladores, manejo de los lanzamientos, administración de fallos, estabilidad entre el código, los esfuerzos de desarrollo experimental, atribución y trazabilidad en los cambios de los desarrolladores. El sistema de control de versiones permite al grupo coordinador central abarcar todas estas áreas.

El núcleo del sistema es la gestión de cambios: identificar cada cambio a los ficheros del proyecto, anotar cada cambio con información adicional (metadata) como la fecha y el autor de la modificación y disponer esta información para quien sea y como sea. Es un mecanismo de comunicación donde el cambio es la unidad básica de información.

En definitiva, el objetivo más importante de los sistemas de control de versiones es controlar los cambios en el desarrollo de un proyecto, permitiendo conocer no solo su estado actual, sino también los cambios a lo largo del tiempo que han llevado el proyecto a su estado actual.

Y visto desde otro punto de vista, podemos entender su funcionamiento como un 'deshacer' infinito el cual esta solo limitado a lo que esté guardado en el repositorio. Dentro del mismo, podemos crear copias completas del proyecto, con su propia historia sin multiplicar el espacio necesario y podemos retroceder y ver la evolución del proyecto, incluyendo estas copias completas.

Índice

Introducción.....	1
Conceptos de control de versiones.....	5
Glosario.....	5
El control de versiones.....	7
¿Cuál es el problema?.....	7
“Niveles” en el control de versiones.....	9
Nivel 0 – Duplicar manualmente archivos.....	10
Nivel 1 – Control de versiones locales.....	10
Nivel 2 – Control de versiones centralizado.....	10
Nivel 3 – Control de versiones distribuido.....	11
git.....	12
La historia del control de versiones.....	13
GIT Base.....	17
Instalación.....	17
Configuración inicial.....	17
Tu identidad.....	17
Editor de textos.....	18
Verificar la configuración.....	18
¿Y la contraseña?.....	19
Creación de una llave SSH.....	19
Cargar la llave publica en GitHub.....	20
Cambiano el método de conexión entre el repositorio local y el remoto en GitHub.....	22
Crear un repositorio.....	23
Usando GitHub.....	23
Línea de comandos – git init.....	28
Guardando y confirmando cambios en el repositorio.....	30
El ciclo de vida de un archivo.....	30
Como están las cosas – git status.....	31
Agregando al área de preparación archivos nuevos.....	32
Agregando archivos nuevos al repositorio.....	33
Mover archivos dentro del repositorios.....	33
Remover archivos del repositorio.....	33
Corrigiendo el último commit.....	34
Trabajando con otros repositorios ‘remote’.....	34
Etiquetado.....	35
GitHub.....	37
Pull Requests.....	37
Issues.....	37
Wiki y otras herramientas.....	37

Buenas prácticas.....	37
Usen mensajes de commit descriptivos.....	37
Hacer de cada commit una unidad atómica.....	38
Miren antes de guardar.....	38
Incorporen los cambios de los demás frecuentemente.....	39
Compartí tus cambios frecuentemente.....	39
No olvides la coordinación con los co-equippers.....	39
Recuerda que las herramientas están orientadas a líneas (de texto).....	39
No guardes archivos generados (o intermedios).....	40
No uses la fuerza.....	40
Soluciones a situaciones frecuentes.....	40
Quiero deshacer un cambio hecho sin commit.....	40
Bibliografía.....	41

Conceptos de control de versiones

Antes de continuar, veremos los conceptos fundacionales de estas herramientas con su nombre en inglés que son los que más verán.

Glosario

Para poder utilizar correctamente la herramienta, primero estableceremos un lenguaje común, que dentro del glosario utilizaremos el término en Inglés cuando no exista una alternativa razonable en Español. Y también para que les sea más fácil entender la documentación disponible y la herramienta en sí.

Así cuando use palabras como “mergear” o “pushear”, que aunque no sean palabras válidas, ofrecen un punto medio aceptable.

Branch	Un desdoblamiento del árbol de código fuente sobre el que se va a trabajar, conservando toda la historia y una conexión al punto en el que sucedió.
Trunk	Literalmente sería el “tronco” principal de código fuente pero conceptualmente es una rama más dentro del repositorio, aunque la más importante.
Merge	Operación de fusión de diferentes ramas.
Commit	Operación de confirmación de cambios en el sistema de control de versiones.
Changeset	Conjunto de cambios que hace un usuario y sobre los que se realiza una operación “Commit” de manera simultánea y que son identificados mediante un número único en el sistema de control de versiones.
Tag	Darle nombre a un <u>changeset</u> de forma de que sea fácil encontrarlo luego, o destacado por alguna razón, generalmente se lo utiliza para marcar lo usado en una entrega; versión 1.0 , por ejemplo.
Main (o master)	la rama principal en un repositorio.
Conflicto	Un cambio hecho por un usuario que es incompatible con los hechos por otros. Esto se produce generalmente al hacer cambios sobre un mismo archivo y mismas líneas y aunque el sistema es capaz de resolver algunos de estos cambios superpuestos, su resolución definitiva puede necesitar que verifiquemos que quede todo en orden.
Pull	Acción de ir a buscar cambios hacia otro repositorio (o rama)
Push	Acción de llevar cambios a otro repositorio; o ‘empujar’
Pull request	Esta acción propia de los sistemas sobre Git como GitHub o GitLab, notifica al propietario de un repositorio de contribuciones hechas para que “vaya a buscarlas” y se encargue de unir esa sugerencia de cambio en su repositorio.

SHA-1	Los hashes SHA-1 son lo que Git utiliza para generar identificadores, incluyendo los de los commits. Para calcularlos, Git no solamente utiliza los cambios que forman parte de un commit, sino también sus metadatos (tales como fecha, autor, mensaje), incluyendo los identificadores de todos los commits hechos para cambios anteriores. Esto hace que los ID de commits de Git sean virtualmente únicos. I.e., es ínfima la probabilidad de que un mismo ID se refiera a dos commits hechos de forma independiente, incluso si tuvieran los mismos cambios.
Origin	repositorio de donde hemos hecho checkout primero, se usa como la dirección por defecto.
Remote	indicación de otro repositorio
Staging area	Lugar en donde se prepara el próximo commit. (o área de preparación)

El control de versiones

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

Un control de versiones es un sistema que registra los cambios realizados en un archivo o conjunto de archivos a lo largo del tiempo, de modo que sea posible recuperar el estado de los archivos guardados a versiones específicas. Aunque en los ejemplos de este apunte usaremos archivos de código fuente, en realidad puedes hacer lo mismo con casi cualquier tipo de archivo que encuentres en una computadora.

Estos sistemas permiten regresar a versiones anteriores de tus archivos, regresar a una versión anterior del proyecto completo, comparar cambios a lo largo del tiempo, ver quién modificó por última vez algo que pueda estar causando problemas, ver quién introdujo un problema y cuándo, y mucho más. Y todos estos beneficios a un costo muy bajo, ¡solo es necesario registrar los cambios!.

¿Cuál es el problema?

Los equipos de software que no utilizan ninguna forma de control de versiones a menudo se encuentran con problemas como no saber qué cambios que se han hecho están disponibles para los usuarios o la creación de cambios incompatibles entre dos partes no relacionadas que tienen que desvincularse y revisarse exhaustivamente. Si eres un desarrollador que nunca ha utilizado el control de versiones, puede que hayas añadido versiones a tus archivos, quizás con sufijos como "**final**" o "**r2**", y que después hayas tenido que enfrentarte con una nueva versión final. Quizás has convertido en comentarios bloques de código, porque quieres desactivar una determinada función sin eliminar el código, con el miedo de que pueda utilizarse más adelante. El control de versiones es una forma de solucionar estos problemas.

Al final, cuando todo viene bien cuando trabajamos solos y en algo predecible, usar una herramienta adicional es solo trabajo extra...



Figura 1: Cuando está todo alineado, no hay ningún problema

Pero en el momento que el trabajo comienza a involucrar a otros, cada aporte por otro miembro del equipo en la tarea que sea, produce cambios en el documento que se dan en simultáneo con los cambios propios [y los del resto del equipo].

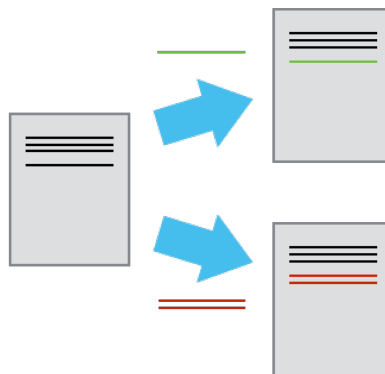
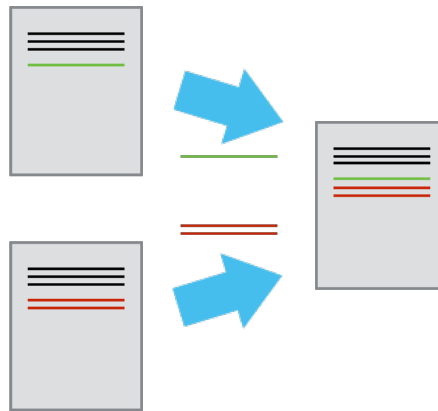


Figura 2: Pero cuando empezamos a trabajar en simultaneo con otra persona...

El archivo que le enviamos a nuestro colega lo transforma en dos. Cada documento evolucionará de manera separada hasta que sea necesario unificarlos para la versión “Final”.



*Figura 3: Las cosas se complican al unir
estos cambios simultáneos*

Al momento de unir los documento, no podemos simplemente descartar uno de los dos. Es necesario que veamos que cambio en uno y otro para reconstruir este documento mejorado.

Un sistema de control de versiones en su concepto más básico, es una herramienta que realiza un seguimiento de los cambios por nosotros y nos ayuda a controlar que va cambiando, registrando todas las modificaciones en el tiempo, ayudándonos a integrar los cambios hechos por múltiples colaboradores en nuestros archivos conformando la siguiente versión del documento. Estos sistemas además registran información adicional útil a la gestión del proyecto como quien hizo el cambio, cuando y otra metadata útiles sobre los cambios.

El historial completo de commits para un proyecto en particular y sus metadatos forman un repositorio. Los repositorios pueden mantenerse sincronizados en diferentes computadoras, facilitando así la colaboración entre diferentes personas.

"Niveles" en el control de versiones

Es habitual ver en la literatura que la evolución de los sistemas de control de versiones anteriores son algo que fue desechado o dado por obsoleto. Esto no es cierto y es posible establecer una graduación que se construye sobre estos y en donde cada sucesivo nivel resuelve las falencias de los anteriores.

Nivel 0 – Duplicar manualmente archivos

Un método de control de versiones y frecuentemente utilizado por muchas personas, es copiar los archivos a otro directorio (quizás indicando la fecha y hora en que lo hicieron, si son ingeniosos). Este método es muy común porque es muy sencillo, pero también es tremendamente propenso a errores. Es fácil olvidar en qué directorio te encuentras y guardar accidentalmente en el archivo equivocado o sobrescribir archivos que no querías.

Este nivel es frecuentemente utilizado cuando tenemos que colaborar via correo electrónico. Es habitual encontrar documentos en el que cambiamos el nombre para indicar quien fue el que realizó la modificación

Nivel 1 – Control de versiones locales

Para afrontar este problema los programadores desarrollaron hace tiempo sistemas de control de versiones locales que contenían una simple base de datos, en la que se llevaba el registro de los cambios realizados a los archivos, generalmente de manera individual. Si bien estos sistemas funcionan para el trabajo individual, no son de mucha ayuda cuando hay más usuarios involucrados.

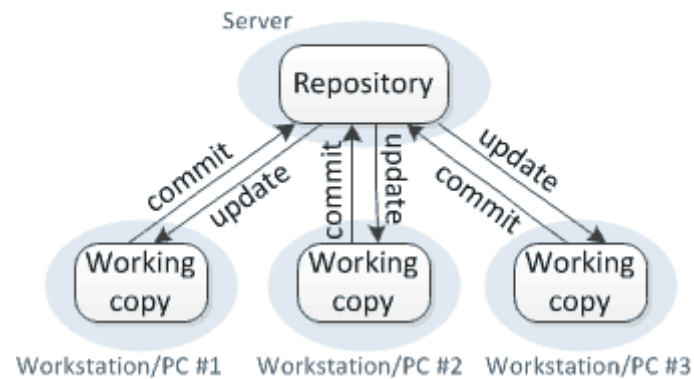
Nivel 2 – Control de versiones centralizado

Los sistemas de control de versiones centralizados, resuelven el problema de la colaboración haciendo que la información este disponible en un único lugar para todos los participantes del proyecto.

El trabajo en simultáneo trae un nuevo problema. ¿Qué pasa cuando dos usuarios modifican el mismo archivo? (<http://svnbook.red-bean.com/nightly/es/svn-ch-2-sect-2.html>)

Para resolver esta situación, los primeros sistemas de este nivel permiten que los usuarios bloqueen archivos individuales, de esta forma, hay solo un usuario haciendo modificaciones, eliminando el problema de la 'simultaneidad'. Con la acción de bloquear y desbloquear viene un nuevo problema, ¿que pasa cuando el usuario se olvida de desbloquear el archivo?

Centralized version control



Para resolver este problema, se crearon sistemas de control de versiones que intentan resolver estas situaciones con la acción de “merge” o fusión de cambios, haciendo que el ultimo usuario que introduce cambios simultáneos se encargue de resolver el conflicto. Afortunadamente, estos sistemas almacenan todos los cambios, incluso los producidos por cambios simultáneos.

Esta centralización trae consigo un problema interesante. Si el disco duro en el que se encuentra la base de datos central se corrompe, y no se han realizado copias de seguridad adecuadamente, se perderá toda la información del proyecto, con excepción de las copias de trabajo que las usuarios tengan en sus máquinas locales, pero estas no conservan la información histórica que está guardada en el repositorio central.

Aun a pesar de esto, la centralización permite un control muy detallado sobre los permisos de acceso y una administración simplificada.

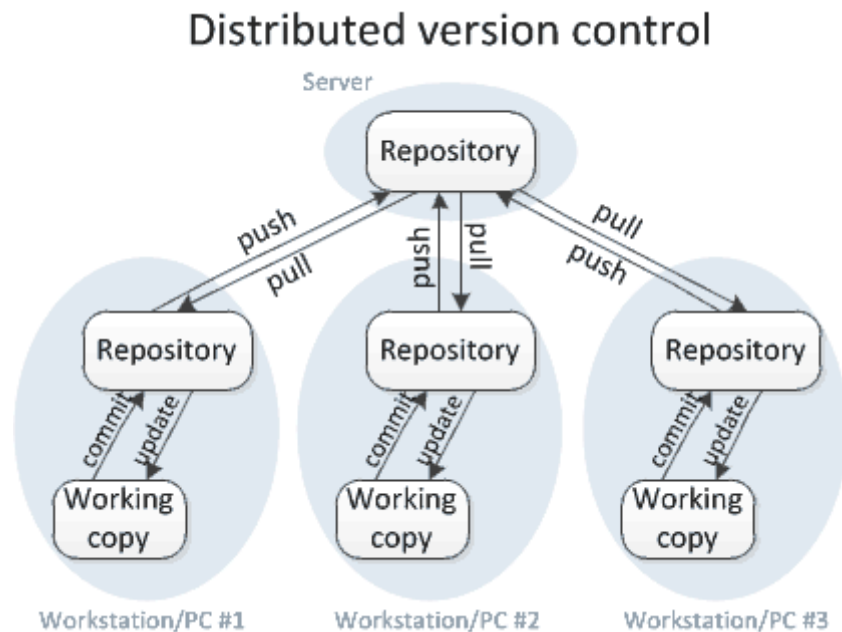
Nivel 3 – Control de versiones distribuido

Los sistemas de control de versiones distribuidos, no necesitan un servidor centralizado para alojar el repositorio. Aparte de eso, no son muy diferentes a los sistemas del nivel anterior.

Para funcionar de manera descentralizada, cada copia en donde trabajemos contiene una copia completa del historial, y cada repositorio puede compartir esta historia con otros repositorios.

Estos sistemas están focalizados en compartir cambios, cada cambio tiene una identificación única.

Los sistemas distribuidos no tienen una estructura impuesta. Es posible crear ubicaciones controladas muy similares a las de los sistemas centralizados, pero no estamos limitados y es posible que los participantes colaboren directamente antes de usar el repositorio principal.



El diagrama muestra como se usa habitualmente un sistema de este tipo, pero acá lo interesante es que no estamos limitados a esta estructura, y cada repositorio se puede comunicar con otro, la distinción de 'repositorio central' es solo una decisión de proyecto.

Al final del día, siempre es necesario contar con un repositorio principal al que podamos recurrir de forma de simplificar [y centralizar] la colaboración entre los miembros del equipo. [Y saber cual es la última versión del proyecto]

git

Git fue desarrollado inicialmente por [Linus Torvalds](#) y el equipo de desarrollo del [Kernel de Linux](#). Ellos estaban usando otro sistema de control de versiones distribuido propietario, Bitkeeper.

Aun siendo un gran avance contra lo que en aquel momento existía, Linus decidió desarrollar un sistema propio que se adaptara mejor a las necesidades del proyecto del Kernel de Linux.

Por ello, el mismo equipo de desarrollo del Kernel de Linux se tomó la tarea de construir desde cero un sistema de versionado de software, también distribuido, que aportase lo mejor de los sistemas existentes hasta el momento.

Una de las principales ventajas de Git es que casi todas las operaciones son dentro de la misma maquina, por lo que no hay que esperar la descarga de información para hacer uso de la historia contenida en el repositorio.

<https://www.atlassian.com/git/articles/10-years-of-git>

<https://git-scm.com/book/es/v2>

La historia del control de versiones

Traducido y adaptado de:

<https://www.flourish.org/2011/12/astonishments-ten-in-the-history-of-version-control/>

Francis Irving, desarrollador de TortoiseCVS [diciembre 2011]

En 10 asombrosos pasos

"Si realmente quieres ... una historia verdaderamente antigua, tienes que volver a los mazos delta en las tarjetas perforadas". [Jim Rootham]

Ver la historia de las herramientas es un tema importante, ya que el producto final [por ahora] parece increíblemente obvio. Y popular.

Sin embargo, se necesitaron décadas de innovación iterativa, de algunas de las mentes más inteligentes en el campo, para hacer algo tan aparentemente simple pero poderoso.

1. ¡El código fuente es texto en un archivo! [Década de 1960]

En retrospectiva, es obvio que el código fuente se almacena mejor simplemente escribiendo en documentos simples. Una breve lectura de la historia de ASCII da una idea de la complejidad de estar de acuerdo incluso con eso.

2. ¡Los seres humanos pueden realizar un seguimiento manual de las versiones del código! [Década de 1960]

Como todo, al principio no había software.

"En mi primer trabajo, teníamos un departamento de control de fuentes. Cuando tuvo su código listo para funcionar, llevó sus disquetes a las agradables damas de Source Control, ellas tomarían sus discos, actualizarían debidamente la biblioteca y crearían el producto listo para el cliente a partir de la fuente oficialmente reposicionada".
[Miles Duke]

3. ¡Puede mantener muchas versiones en un archivo! [1972, 1982]

Usando un elegante formato de archivo de tejido intercalado, SCCS dominó el control de versiones durante una década.

Llevó algunos años desarrollar un buen método para registrar los cambios de una versión de un archivo a la siguiente. "An Algorithm for Differential File Comparison" es un artículo relativamente tardío para leer sobre el tema [1976].

En 1982, el sucesor de SCCS, RCS (documento original que lo describe) usó estas diferencias al revés para vencer a SCCS, y asombró a este comentarista:

"Llegó RCS con sus deltas inversos, y pensé que eran las rodillas de la abeja"
[Anónimo]

4. ¡Cada uno puede tener su propia copia extraída! [mil novecientos ochenta y dos]

En ese momento, las personas tendían a iniciar sesión en un mainframe central y trabajar juntas a través de él. Con RCS, mediante enlaces simbólicos, se podía organizar de modo que cada persona trabajara con el mismo control de versiones, pero con su propia copia de trabajo.

"Habrà un archivo llamado RCS que es un enlace simbólico al repositorio principal de RCS que compartes con el resto de los miembros de tu grupo" [Información sobre el uso de RCS en Yale]

5. ¡Vaya! ¡Puede versionar varios archivos a la vez! [1986]

Sorprendentemente, hasta CVS, cada sistema de control de versiones era para archivos individuales separados. Sí, puede usar RCS con comodines para confirmar varios archivos o marcar ramas particulares. Pero en realidad no es parte del sistema.

En CVS era el valor predeterminado modificar todos los archivos de forma recursiva. De repente, el software se convirtió en un árbol recursivo de archivos de texto, en lugar de solo un directorio o un archivo individual.

Estaba mal implementado, ya que no era "atómico" [el sucesor de Subversion arregló esto en 2000], pero realmente eso no importa con el propósito de asombro.

6. ¡Dos personas pueden editar el mismo archivo al mismo tiempo, y fusiona lo que ambos hicieron! [1986]

A finales de la década de 1990 trabajé en Creature Labs. Estábamos cambiando de Visual SourceSafe [comercial, hecho por Microsoft] a CVS [código abierto, hecho por un grupo de hippies].

Francamente, no se podía creer que pudiera cumplir su principal promesa mágica: permitir que varias personas editaran el mismo archivo al mismo tiempo y poder fusionar sin problemas sus cambios sin romper nada.

El bloqueo exclusivo de SourceSafe fue un problema real cuando estábamos haciendo Creatures 3. Recuerdo una ocasión en particular en la que agregamos la recolección de basura, lo que significó editar la mayoría de los archivos de código, y el programador principal tuvo que verificar cada archivo exclusivamente durante el fin de semana mientras implementaba eso.

Este artículo de 1986 es un excelente registro histórico de esta magia, en la que Dick Grune sufre el mismo problema mientras su equipo codifica un compilador en Holanda, y así inventa CVS.

7. ¡El repositorio compartido puede estar en una máquina remota! [1994]

La mayor parte de este tiempo la gente usaba principalmente el control de versiones en una computadora. Algunas versiones de RCS, y, por lo tanto, CVS, tenían un mecanismo de intercambio de archivos remoto para permitirle tener un repositorio de código remoto en 1986.

“Si se usa una versión de RCS que puede acceder a archivos en una máquina remota, el repositorio y los usuarios pueden estar en diferentes máquinas” [Dick Grune]

Pero parece que fue solo en 1994 cuando se agregó un protocolo TCP / IP, que la idea realmente despegó.

“[CVS] no se volvió realmente omnipresente hasta después de que Jim Blandy y Karl Fogel [más tarde dos directores del proyecto Subversion] organizaron el lanzamiento de algunos parches desarrollados en Cygnus Software por Jim Kingdon y otros para hacer que el software cliente CVS se pudiera usar en el futuro final de una conexión TCP / IP desde el repositorio” [Eric Raymond]

8. ¡Alojamiento de control de versiones de código abierto gratuito! [1999]

Este no es un avance en la tecnología de control de fuente, pero fue asombroso, y los avances sociales en Internet pueden ser tan importantes como los técnicos:

La tendencia era que las versiones anteriores de OSS fueran difíciles de encontrar ... John T. Hall tenía la idea de que si los proyectos se desarrollaban en el sitio, las versiones antiguas estarían allí de forma predeterminada. Un servicio de plataforma de desarrollo era audaz, pero nadie más lo estaba haciendo, y pensamos "¿por qué no?" [Brian Biles]

Festejando como si no hubiera un mañana [para sus acciones], VA Linux presentó SourceForge al mundo. Esto fue genial para proyectos nuevos [como mi TortoiseCVS].

En aquel entonces era difícil y costoso conseguir un servidor en Internet, y no era fácil ni barato configurar el control de fuente y un rastreador de errores. Este nuevo servicio, a pesar de su falta de modelo de negocio, dio a luz numerosos proyectos que poco antes.

9. ¡Puede distribuirlo todo para que no haya un repositorio central! [2005]

Hubo una ola de sistemas de control de versiones a principios de los años noventa, lo que hizo que el control de versiones estuviera completamente distribuido.

Es decir, su máquina local tiene una copia completa del historial del código y puede bifurcarse y fusionarse fácilmente de igual a igual con cualquier otra copia del mismo. Por cierto, la misma característica hace que sea mucho más fácil ramificar y fusionar en general.

Dado eso, parece injusto que haya fechado este asombro en 2005. Eso es porque no estoy grabando la primera vez que alguien hizo algo asombroso, sino la primera vez que se produjo y se hizo popular. Abril de 2005 fue cuando se lanzaron Mercurial y Git.

El artículo "Los riesgos del control de versiones distribuidas" [finales de 2005] muestra lo radical que se veía este material novedoso.

10. Cuando pagas, eso también es una bifurcación, ¡y puedes hacerlo en público! [2008]

El éxito de GitHub se debe a varias razones [que merecen una publicación de blog completa, aunque ya he mencionado una de ellas antes].

En el contexto de esta publicación, el asombro fue que tal vez desee hacer públicos incluso sus pequeños trucos al código de otras personas. Antes de GitHub, solíamos mantenerlos en nuestra propia computadora.

Hoy en día, es tan fácil hacer una fork, o incluso editar código directamente en su navegador, que potencialmente cualquiera puede encontrar incluso las correcciones de errores menos pulidas inmediatamente.

GIT Base

Instalación

En pos de maximizar el aprendizaje en el uso de esta herramienta, recomendamos fuertemente que no utilicen herramientas gráficas para interactuar con git.

En Windows, visiten <https://gitforwindows.org/> para descargar el instalador.

En Linux, pueden usar el gestor de paquetes de su distribución para instalarlo.

En MacOS... no sabría decirles, por lo que si alguien quiere compartir los pasos de instalación los agregaremos.

Configuración inicial

La configuración inicial dependerá de la herramienta final que utilicen, aquí estarán los pasos necesarios para la línea de comandos, que es el mínimo común denominador.

Git posee una herramienta `git config` la cual permite cambiar todos los parámetros de funcionamiento de manera centralizada.

La herramienta obtiene los parámetros de configuración a dos niveles: (son tres, al agregar lo que es global a la computadora pero no trataremos aquí)

- I. El archivo `~/.gitconfig` o `~/.config/git/config` contiene los parámetros relacionados a el usuario actual, para modificar los parámetros alojados aquí es necesario agregar `--global` a `git config`.
- II. El segundo lugar de donde se puede almacenar configuración es en cada repositorio individual, dentro de la carpeta `.git/config`. Es posible forzar el guardado de configuración en esta ubicación utilizando el parámetro `--local`.

Tu identidad

El paso previo más importante aquí, es identificarnos para que nuestros colegas y servicios de alojamiento de repositorios puedan reconocer nuestros `commits` al momento de compartirlos en la web.

```
~/dev/git/casero$ git config --global user.name "Mi Nombre"
```

```
mrtin@maeve:~/dev/git/casero$ git config --global user.email  
correo@electronico.com
```

Y al usar la opción `--global`, nos aseguramos de hacer esta configuración una sola vez y aunque el nombre no tiene porqué ser el nombre de usuario, la dirección de correo electrónico tiene que existir y ser la que utilicemos en GitHub.

! Sin la configuración del correo electrónico hecha correctamente, GitHub no podrá conectar los commits con el usuario. Esto trae algunas complicaciones con sistemas que gestionan los repositorios de manera automática.

Editor de textos

En este paso, le indicamos a git que editor de textos utilizaremos cuando nos pida hacer alguna modificación de archivos; son varias operaciones y no tener esto configurado puede provocar problemas difíciles de detectar luego.

En GNU/Linux esto es mas sencillo, y podemos indicar casi directamente el editor:

```
mrtin@maeve:~/dev/git/casero$ git config --global core.editor emacs
```

Aunque parezca tentador, usen un editor simple para esta tarea.

En Windows una alternativa popular es [Notepad++](#) (un editor de textos que resalta la sintaxis y estructura de una amplia variedad de lenguajes de programación y es Software Libre)

Para configurarlo tenemos que utilizar

```
mrtin@maeve:~/dev/git/casero$ git config --global core.editor "'C:/Program  
Files/Notepad++/notepad++.exe' -multiInst -notabbar -nosession -noPlugin"
```

Esta instrucción abre el editor de forma simplificada para que sea lo más rápido posible.

Para más detalles sobre la configuración del editor vean la página de manual de [git config core.editor](#).

Verificar la configuración

Para verificar que la configuración es la correcta, podemos ejecutar

```
mrtin@maeve:~/dev/git/casero$ git config --list
user.email=profe@unrn.edu.ar
user.name=MrTin V
ilu
color.status=auto
#... etc etc
```

Si vemos algun parametro inesperado, podemos usar el comando

```
mrtin@maeve:~/dev/git/casero$ git config --list --show-origin
```

Nos permite saber de que archivo esta tomando la configuración para cada ítem así como la ubicación de cada uno.

```
file:/etc/gitconfig      filter.lfs.process=git-lfs filter-process
file:/etc/gitconfig      filter.lfs.required=true
file:/home/mrtin/.gitconfig  user.email=profe@unrn.edu.ar
file:/home/mrtin/.gitconfig  user.name=MrTin Vilu
```

¿Y la contraseña?

El ultimo paso en la configuración inicial, es uno que varias aplicaciones de escritorio resuelven de manera automática, pero es necesario hacer a mano si usamos alguna más básica o desde la linea de comandos directamente.

A pesar de que podemos utilizar GitHub con nuestro usuario y contraseña, esto esta actualmente marcado para ser removido [deprecado] y sera reemplazado definitivamente por una forma más segura de identificación y autenticación.

Creación de una llave SSH

Para leer como funciona esto, pueden iniciar su búsqueda por [WikiPedia](#)

En GNU/Linux la llave *muy probablemente* ya esté creada, y pueden pasar a la segunda parte.

Para verificar si la llave ya esta creada pueden usar

```
mrtin@maeve:~/dev/git/casero$ ls -al ~/.ssh
total 60
```

```
drwx----- 3 mrtin mrtin 4096 feb  8 19:14 .
drwxr-xr-x 72 mrtin mrtin 4096 may  8 20:30 ..
-rw----- 1 mrtin mrtin 2590 nov 29 2019 id_rsa
-rw-r--r-- 1 mrtin mrtin 565 nov 29 2019 id_rsa.pub
-rw----- 1 mrtin mrtin 20120 may  7 18:07 known_hosts
```

En gris, son los archivos que corresponden a la llave.

```
mrtin@maeve:~/dev/git/casero$ ssh-keygen -t rsa -b 4096 -C "correo@gmail.com"
Generating public/private rsa key pair.
# Nos pide la ruta en la que se va a guardar las credenciales, en este caso
# damos enter, para que las guarde en una ubicación estándar en $HOME
Enter file in which to save the key (/home/mrtin/.ssh/id_rsa):
# Es muy importante crear una contraseña para la llave, esta puede
# (y debería) ser diferente a la de GitHub. Tiene que ingresarla dos veces
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
# Y luego nos muestra el resultado de la operación
Your identification has been saved in /home/mrtin/.ssh/id_rsa
Your public key has been saved in /home/mrtin/.ssh/id_rsa.pub
The key fingerprint is:
SHA256:Elha6rZr0jv+QUD8KrMHbB+wh879gaEHtGxpWLw2/zw tu_correo@gmail.com
The key's randomart image is:
+---[RSA 4096]---+
| ..  o          |
| o.  *          |
| .+.  .         |
|+oo+.  .        |
|oBBo+  . S      |
|**+Bo.  .       |
|++o=.         |
|.o=o.=E        |
| o+B+.o.       |
+-----[SHA256]-----+
```

Luego de crear la llave, es necesario cargarla en el llavero, para lo cual primero nos aseguramos de que el programa de llavero este en funcionamiento con:

```
mrtin@maeve:~/dev/git/casero$ eval $(ssh-agent -s)
```

Y agregamos la llave que creamos.

```
mrtin@maeve:~/dev/git/casero$ ssh-add ~/.ssh/id_rsa
Enter passphrase for /home/mrtin/.ssh/id_rsa: <contraseña>
```

```
Identity added: id_rsa (tu_correo@gmail.com)
```

Con la llave creada y cargada en el llavero, pasamos al siguiente paso.

Cargar la llave publica en GitHub

Primero tenemos que obtener el texto de la llave, para eso utilizamos el editor de texto que tengamos a mano (notepad, nano, notepad++, etc.) y abrimos el archivo `/home/mrtin/.ssh/id_rsa`. El programa `xclip`, permite enviar al portapapeles un archivo desde la consola, es necesario instalarlo para su uso en GNU/Linux.

```
mrtin@maeve:~/dev/git/casero$ cat ~/.ssh/id_rsa.pub
ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAQgQCtHVn7KalehLHWGNJPOqNviU7nIBYS1S1dsc4hc19RAHpTP
...lTzv43PVQDYn1Y0mNqIEnb9wZL77VppKUSrDmEk= tu_correo@gmail.com
mrtin@maeve:~/dev/git/casero$ xclip -i ~/.ssh/id_rsa.pub
```

La llave publica comienza por `ssh-rsa` y termina con la dirección de correo que le indicamos; `tu_correo@gmail.com`.

Con el texto de la llave en el portapapeles, vamos a <https://github.com/settings/keys> y damos a "new SSH key":

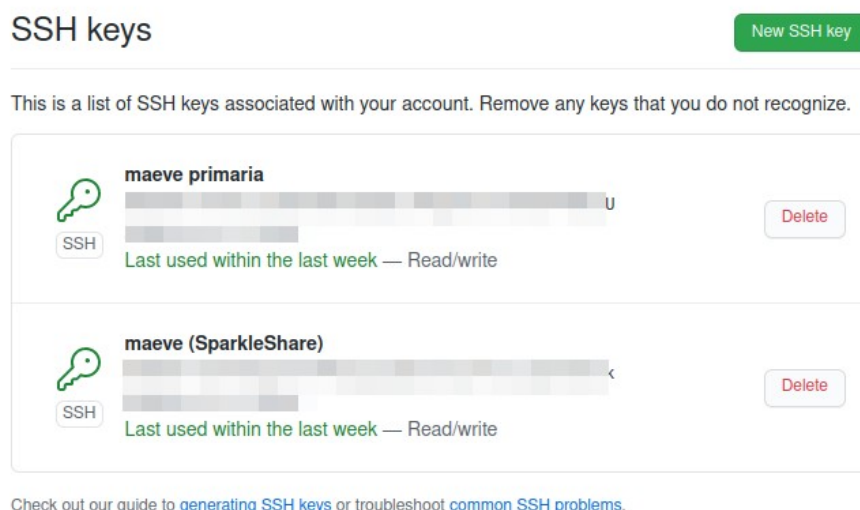
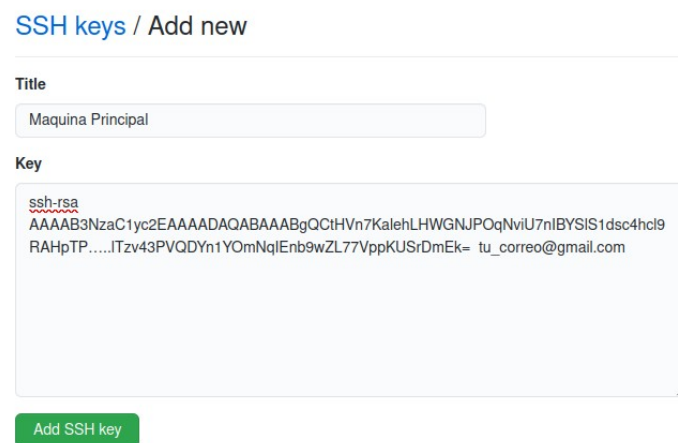


Figura 4: Listado de llaves agregadas a la cuenta.

Acá también pueden revocar llaves de notar actividad sospechosa o porque no están usando más esa computadora. Por la misma razón, es importante que cada computadora que utilicemos tenga una llave propia.

Aquí le damos un nombre para identificar la llave (y máquina a la que pertenece) y pegamos el portapapeles que contiene la llave.



The screenshot shows the 'SSH keys / Add new' page on GitHub. It has a form with two main sections: 'Title' and 'Key'. The 'Title' section has a text input field containing 'Maquina Principal'. The 'Key' section has a large text area containing an SSH key. The key is labeled 'ssh-rsa' and consists of a long alphanumeric string followed by the email 'tu_correo@gmail.com'. At the bottom of the form is a green button labeled 'Add SSH key'.

Figura 5: Agregado de una llave SSH

Y le damos “Add SSH key” y finalizamos el agregado de la llave.

Cambiando el método de conexión entre el repositorio local y el remoto en GitHub

Si por alguna razón tienen algún problema haciendo **push** de **commits** al repositorio (o están cansados de ingresar su usuario y contraseña con cada operación), verifiquen que al momento de hacer el **checkout** inicial, este haya sido con la dirección **SSH**:

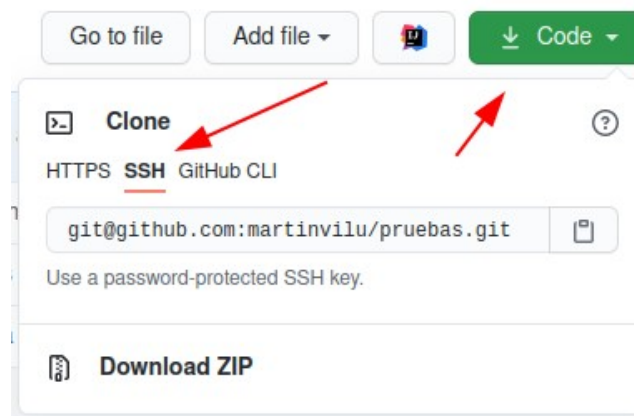


Figura 6: La forma recomendada para clonar un repositorio

Y no con la dirección HTTPS,

Con la instrucción `git remote`, podemos ver como está conectado con GitHub nuestro proyecto.

```
mrtin@maeve:~/dev/git/casero$ git remote -v
origin https://github.com/martinvilu/pruebas.git (fetch)
origin https://github.com/martinvilu/martinvilu.git (push)
```

Para cambiarlo, ejecutamos `git remote set-url`:

```
mrtin@maeve:~/dev/git/casero$ git remote set-url origin
git@github.com:martinvilu/pruebas.git
```

Y ahora podemos verificar el cambio ejecutando nuevamente `git remote -v`.

```
mrtin@maeve:~/dev/git/casero$ git remote -v
origin      git@github.com:martinvilu/pruebas.git (fetch)
origin      git@github.com:martinvilu/pruebas.git (push)
```

Descargando un repositorio creado – `git clone`

Antes que crear un repositorio y en especial si desean trabajar sobre GitHub (o cualquier otro servicio de los servicios online), pueden crear el repositorio en el servicio y descargarlo para trabajar. Esto es preferible para el 99,99% de los casos y simplifica un conjunto de configuraciones que son necesarias para que funcione todo correctamente.

Pero muy en especial, las cátedras poseen un sistema de creación de repositorios automáticos que hacen que toda la parte de inicialización de repositorios desde la consola no sea necesaria. Aunque es posible, la conexión requiere de múltiples pasos que no hacen de esto una tarea simple.

Empleando la dirección que está en la [Figura 6](#), podemos descargar el repositorio de forma que es utilizable por `git`. Si utilizan el botón que dice “Download ZIP” esto descargara solo los archivos del repositorio y no será posible trabajar con el mismo.

Este comando creará un directorio con el repositorio ya conectado con la dirección del que indicaron como `origin`, por lo que todas las operaciones de compartir cambios, actuarán sobre este de manera automática.

```
mrtin@maeve:~/dev/git/casero$ git clone https://github.com/martinvilu/pruebas
Cloning into 'pruebas'...
remote: Enumerating objects: 351, done.
remote: Counting objects: 100% (73/73), done.
remote: Compressing objects: 100% (36/36), done.
remote: Total 351 (delta 52), reused 43 (delta 37), pack-reused 278
Receiving objects: 100% (351/351), 75.60 KiB | 36.00 KiB/s, done.
Resolving deltas: 100% (146/146), done.
```

Pueden indicar luego de la dirección del repositorio, el nombre del directorio a crear:

```
$> git clone https://github.com/martinvilu/pruebas directorio-destino
```

Esto creará la carpeta `directorio-destino` con el repositorio dentro, listo para usar.

! Recuerden que si cambian el directorio, no utilizar espacios en los nombres de los directorios, para que les sea más fácil indicarlo por la línea de comandos. Además de que `git` entenderá el espacio como separador y no como parte del nombre.

Crear un repositorio

Existen dos formas de crear un repositorio; la primera involucra el uso de GitHub, siendo esto lo más simple, ya que les ahorra múltiples cuestiones de configuración de **origin** para compartir luego el repositorio.

Y la otra forma de creación de repositorios, es la ideal si utilizaran el mismo fuera de servicios como GitHub

Usando GitHub

Con una cuenta de usuario creada, en la página principal tenemos acceso al botón para crear un repositorio nuevo.

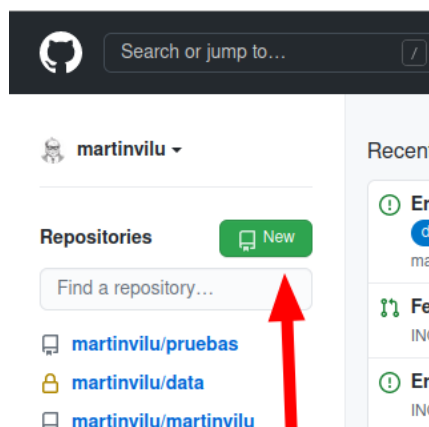


Figura 7: Primer paso en la creación de un repositorio

También podemos ir directamente a <https://github.com/new>

Aquí, lo más importante es indicarle un nombre al repositorio, todo lo demás es opcional, pero no está de más.

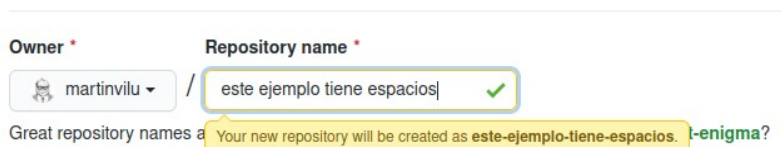
- ! Si el repositorio ya existe en su computadora, no agreguen nada de las opciones de esa página, aunque es posible conectar dos repositorios con historias “separadas” esto también es bastante complejo.

Si está todo en orden con el nombre del repositorio:



Figura 8: Un repositorio listo para ser creado

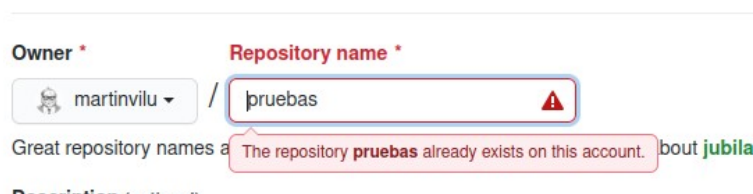
Podemos crear el repositorio con cualquier nombre, sin embargo, si el mismo contiene espacios u otros caracteres no aceptables en la dirección del mismo, GitHub nos mostrará una advertencia en amarillo y el nombre que será utilizado. Por ejemplo, “este repositorio tiene espacios” quedará como “**este-repositorio-tiene-espacios**”. Además de que los caracteres no admitidos son reemplazados por guiones “-”.



The screenshot shows the GitHub repository creation interface. The 'Owner' field is set to 'martinvilu'. The 'Repository name' field contains 'este ejemplo tiene espacios'. A green checkmark is visible next to the name. Below the name field, a yellow tooltip message states: 'Your new repository will be created as este-ejemplo-tiene-espacios. t-enigma?'. The text 'Great repository names a' is partially visible on the left.

Figura 9: Ejemplo de las substituciones por guiones en un nombre

Y no permitirá la creación del repositorio si el nombre ya está en uso. Tengan en cuenta que el nombre en nuestra computadora puede ser diferente a este (no deja de ser un directorio mas)



The screenshot shows the GitHub repository creation interface. The 'Owner' field is set to 'martinvilu'. The 'Repository name' field contains 'pruebas'. A red error message box is displayed below the name field, stating: 'The repository pruebas already exists on this account.' The text 'Great repository names a' is partially visible on the left, and 'bout jubila' is partially visible on the right.

Figura 10: El nombre del repositorio en GitHub

A continuación, una descripción del proyecto, es opcional y se puede modificar luego.



The screenshot shows the 'Description (optional)' field in the GitHub repository creation interface. The field is empty and has a light gray border.

Figura 11: La descripción del proyecto o el propósito del repositorio

La configuración de la privacidad del repositorio:

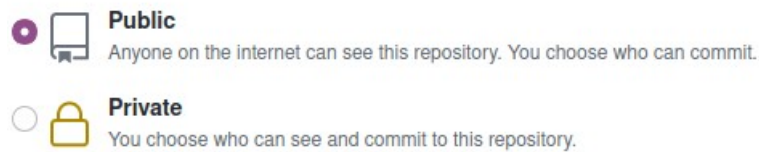


Figura 12: Visibilidad del repositorio

Un repositorio público es visible a todos y accesible desde nuestro perfil, sin embargo, esto no significa que cualquiera puede cargar cambios.

Un repositorio privado es solo visible a quienes explícitamente agreguemos al mismo. Además de poder dar diferentes niveles de acceso al mismo, para por ejemplo que esta no pueda cargar **commits** directamente.

! En el último paso, son algunos archivos comunes que podemos agregar al repositorio, podemos obviar este paso si el repositorio que estamos creando ya existe en nuestras computadoras. También podemos ignorar esta parte si tenemos pensado hacerlo por nuestra cuenta.

Initialize this repository with:

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

☐ **Add a README file**

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

☐ **Add .gitignore**

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

☐ **Choose a license**

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

Figura 13: Plantillas de archivos para agregar al repositorio

El archivo **README.md** es un archivo que será mostrado en la página principal del repositorio y suele ser utilizado para describir el proyecto, sus características y otra información importante.

Un archivo **.gitignore**, indica a **git** que archivos puede ignorar de manera segura. Esto hace que la herramienta no indique la presencia de archivos sin agregar al repositorio. Esto

aplica para todo lo que sea intermedio o subproducto de los archivos. Si lo podemos “regenerar” con la información del repositorio, ¿para que ocupar espacio?

Al seleccionarlo GitHub nos presenta una lista de plantillas referidas a diversos lenguajes de programación:

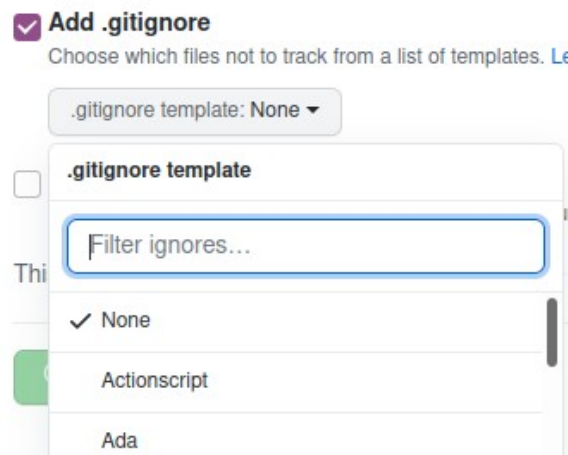


Figura 14: Archivos `.gitignore` base según el tipo de proyecto.

De la misma manera que con el archivo `.gitignore`, podemos elegir un archivo para indicar que se puede hacer y que no con lo que tengamos cargado en el mismo.



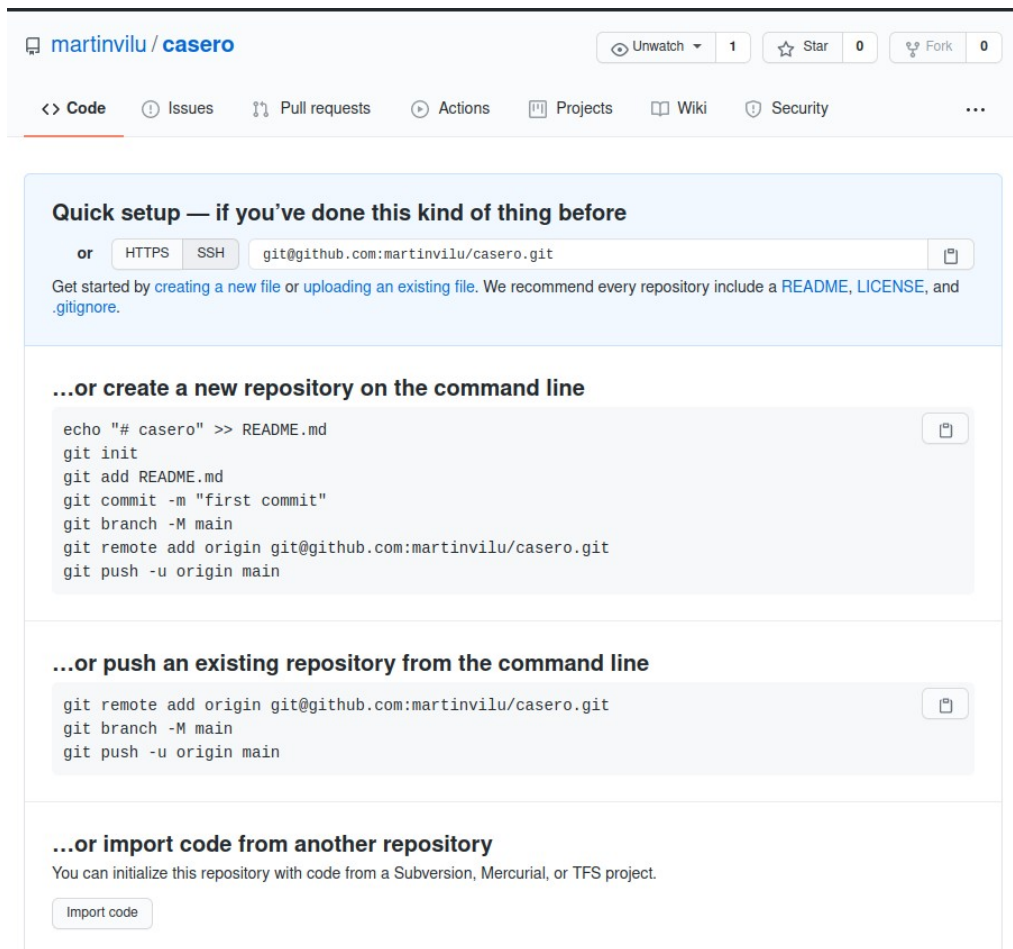
Figura 15: Licencias para agregar automáticamente al crear el repositorio.

Si todo está bien, podemos proseguir a crear el repositorio

Cuando está todo en orden, el botón de Crear Repositorio tendrá un color verde, mientras si hay algún inconveniente por resolver, estará un poco más gris.

Una observación, cuando no indiquen ninguna inicializan, GitHub les mostrara una ayuda rápida con los pasos a seguir:

Que es esencialmente, lo que está en la sección siguiente para poder cargar un repositorio ya existente o fresquito creado en nuestra computadora.



ProTip! Use the URL for this page when adding GitHub as a remote.

Figura 16: Captura de un repositorio recién creado

Crear el repositorio de esta manera es ideal para cuando recién empezamos, no hace falta configurar nada más, como veremos a continuación.

Línea de comandos – `git init`

Primero, nos tenemos que ubicar en la carpeta en la que queremos crear el repositorio.

```
mrtin@maeve:~/dev/git$ pwd
/home/mrtin/dev/git
mrtin@maeve:~/dev/git$ git init casero
Inicializado repositorio Git vacío en /home/mrtin/dev/git/casero/.git/
```

El repositorio en sí, se encuentra en la carpeta `.git` por lo que si borramos esa carpeta, borramos el lugar en donde está la historia, pero no los archivos que estén actualmente en la carpeta `casero`.

```
mrtin@maeve:~/dev/git$ cd casero
mrtin@maeve:~/dev/git/casero$ ls -la casero
total 12
drwxrwxr-x 3 mrtin mrtin 4096 may  8 21:51 .
drwxrwxr-x 9 mrtin mrtin 4096 may  8 21:51 ..
drwxrwxr-x 7 mrtin mrtin 4096 may  8 21:51 .git
```

Pero aparte del repositorio en sí, no hay nada más.

Un detalle importante que es también la diferencia más importante con respecto a crear el repositorio primero en GitHub, es el hecho de que de esta manera, el repositorio no está conectado con ningún otro. Pero a diferencia de lo visto en [Cambiando el método de conexión entre el repositorio local y el remoto en GitHub](#), ahora no hay ninguna conexión.

```
mrtin@maeve:~/dev/git/casero$ git remote -v
# Vacío
```

Para lo que tenemos que agregar un nuevo 'origin'

```
mrtin@maeve:~/dev/git/casero$ git remote add origin
git@github.com:martinvilu/casero.git
```

GitHub usa `main` en lugar de `master` como rama principal, para eso tenemos que renombrar la rama de la siguiente manera (para que funcione el repositorio local debe tener por lo menos un `commit` hecho)

```
mrtin@maeve:~/dev/git/casero$ git branch -M main
```

Si lo anterior falla, podemos agregar un README.md desde la consola, antes de repetir el cambio de nombre y el `push` al repositorio en GitHub

```
mrtin@maeve:~/dev/git/casero$ echo "# Casero y Markdown desde la consola!" >>
  README.md
mrtin@maeve:~/dev/git/casero$ git add README.md
mrtin@maeve:~/dev/git/casero$ git commit -m "Hola repo Casero!"
[master (commit-raíz) 50137d0] Hola repo casero, primer commit
```

```
1 file changed, 1 insertion(+)  
create mode 100644 README.md
```

Esto para luego hacer el **push** hacia el repositorio en GitHub, en donde es necesario indicar que el **push** configure el repositorio “**upstream**” (**-u** o **--set-upstream**)

```
mrtin@maeve:~/dev/git/casero$ git push --set-upstream origin main  
Enumerando objetos: 3, listo.  
Contando objetos: 100% (3/3), listo.  
Escribiendo objetos: 100% (3/3), 267 bytes | 267.00 KiB/s, listo.  
Total 3 (delta 0), reusado 0 (delta 0), pack-reusado 0  
To github.com:martinvilu/casero.git  
* [new branch]      main -> main  
Rama 'main' configurada para hacer seguimiento a la rama remota 'main' de  
'origin'.
```

Para más información, <https://git-scm.com/docs/git-init>.

Podemos aprovechar estos pasos para hacer que nuestro repositorio se conecte con otro servicio como GitHub, como GitLab, BitBucket o cualquier otro. Para mantener todo simple, esto no será cubierto aquí.

Guardando y confirmando cambios en el repositorio

Con el repositorio listo, ya podemos empezar a trabajar, agregando, modificando, moviendo y borrando archivos.

El ciclo de vida de un archivo

En un repositorio, cada archivo puede tener dos estados, rastreado (ya dentro del repositorio) y sin rastrear, aquellos archivos que no fueron agregados (y en algunos casos no es necesario hacerlo) o Tracked Files y Untracked Files.

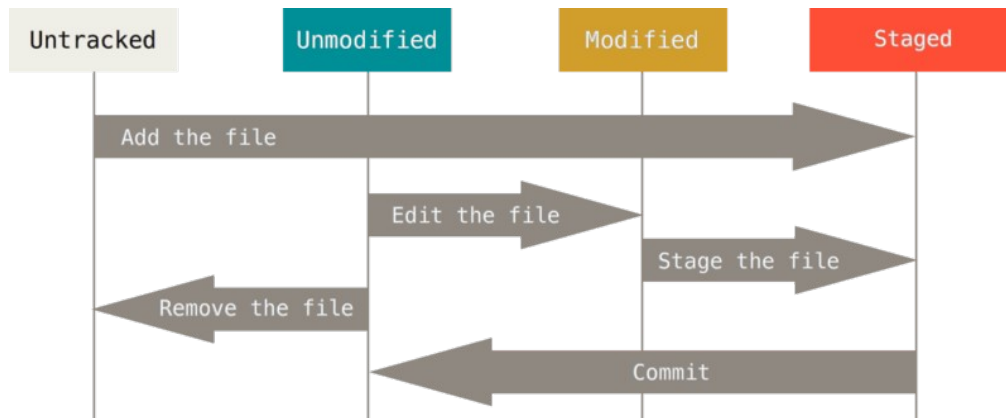


Figura 17: El ciclo de trabajo de un archivo en git

El ciclo de vida completo de un archivo comienza con los archivos sin rastrear (**untracked**), los cuales deben ser agregados al área de preparación (**staging area**) del repositorio para que pasen a ser ‘rastreados’ (**tracked**), para confirmar este cambio, es necesario confirmar el cambio (**commit**). Cuando esto sucede, el archivo pasa a estar sin modificaciones (**unmodified**).

Desde el estado “sin modificaciones”, el archivo puede ser removido para volver a ser “sin rastrear”, o con alguna modificación, pasar a estarlo. Un archivo modificado debe ser puesto en el “área de preparación” para poder ser “confirmado” y que vuelva a “Sin Modificar”

Como están las cosas – **git status**

El comando **git status** nos indicará cuál es el estado del repositorio; cambios pendientes de confirmación, modificaciones fuera del área de preparación y commits sin compartir.

```
mrtin@maeve:~/dev/git/casero$ git status
En la rama main
Tu rama está adelantada a 'origin/main' por 1 commit.
(usa "git push" para publicar tus commits locales)

Cambios a ser confirmados:
(usa "git restore --staged <archivo>..." para sacar del área de stage)
renombrado:    manual.md -> carpeta/manual.md
```

En este ejemplo, status nos indica que hay commits sin cargar en origin (GitHub) y en la segunda parte, están los cambios pendientes sin commit. [“Está adelantado”].

Asimismo, el repositorio tiene un cambio sin confirmar, un cambio de nombre.

Acá es en donde el archivo `.gitignore` que GitHub nos sugiere agregar es importante, un archivo ignorado no será indicado dentro de `git status` para realizar acciones sobre el mismo.

Agregando al área de preparación archivos nuevos

Para poder grabar cambios en un `commit`, es necesario agregarlos al Staging Area ya que de otra forma veremos un mensaje de que hay cambios pero como no están preparados para ser guardados, no hay nada para confirmar.

Tengan en cuenta que `README.md` ya estaba en el repositorio.

```
# Introducimos un cambio
mrtin@maeve:~/dev/git/casero$ echo "otra linea" >> README.md
# Y lo intentamos guardar
mrtin@maeve:~/dev/git/casero$ git commit -m "Agregado: Una nueva linea"
En la rama main
Tu rama está actualizada con 'origin/main'.

Cambios no rastreados para el commit:
  (usa "git add <archivo>..." para actualizar lo que será confirmado)
  (usa "git restore <archivo>..." para descartar los cambios en el directorio
de trabajo)
    modificado:      README.md

sin cambios agregados al commit (usa "git add" y/o "git commit -a")
```

Por lo tanto, entre la modificación y antes del `commit`, es necesario agregar el archivo para que entre en el índice de archivos a guardar.

```
mrtin@maeve:~/dev/git/casero$ git add README.md
mrtin@maeve:~/dev/git/casero$ git commit -m "Agregado: Una nueva linea"
```

El comando `git add` tiene un atajo para agregar todos los archivos para hacer el `commit` (`-a/--all`), esto no es recomendable. Es preferible agregar los cambios individualmente, de forma que mantengamos el control sobre lo que agregamos al `commit` y que el mensaje en el mismo sea representativo del cambio hecho.

Agregando archivos nuevos al repositorio

Si el archivo no estaba agregado como en el ejemplo anterior es necesario agregarlo con `git add` también de forma que esté en el área de preparación previo a confirmarlo.

```
mrtin@maeve:~/dev/git/casero$ ls
README.md
mrtin@maeve:~/dev/git/casero$ echo "# Manual 1" >> manual.md
mrtin@maeve:~/dev/git/casero$ git add manual.md
mrtin@maeve:~/dev/git/casero$ git commit -m "Agregado: Pagina inicial del
manual 1"
```

Mover archivos dentro del repositorio

Al momento de ser necesario mover archivos dentro de un repositorio, es necesario hacerlo con las herramientas de `git`, de forma de que la acción no pierda la historia del archivo. Mover un archivo de manera tradicional es visto por el repositorio como que el mismo fue eliminado y un archivo nuevo ingresó.

```
mrtin@maeve:~/dev/git/casero$ git mv -v manual.md carpeta/
Renombrando manual.md a carpeta/manual.md
```

La opción `-v/--verbose` hace que la instrucción no sea silenciosa y nos indique que fue lo que hizo, de otra forma solo mostrará algo si tuvo problemas.

Remover archivos del repositorio

El comando `git rm` borra el archivo para que este no este en el repositorio y de la computadora.

```
mrtin@maeve:~/dev/git/casero/carpeta$ git rm manual.md
rm 'carpeta/manual.md'
mrtin@maeve:~/dev/git/casero$ git status
En la rama main
Tu rama está adelantada a 'origin/main' por 2 commits.
(usa "git push" para publicar tus commits locales)

Cambios a ser confirmados:
(usa "git restore --staged <archivo>..." para sacar del área de stage)
borrado:      carpeta/manual.md
```

Recuerden que remover es un cambio que queda en el área de preparación hasta que sea confirmado.

Corrigiendo el último commit

Antes de que el último commit sea compartido, es posible corregirlo, utilizando `git commit --amend`. Con esta instrucción podemos volver el último commit al staging area y hacer los ajustes necesarios.

Esta instrucción lo que hace es borrar el último commit y reemplazarlo por uno nuevo, esta es la razón por la cual solo es posible hacerlo si el commit no fue compartido.

```
mrtin@maeve:~/dev/git/casero$ git commit --amend
```

Si nos olvidamos de agregar un archivo, también podemos modificar el último commit de la siguiente manera.

```
mrtin@maeve:~/dev/git/casero$ git commit -m 'Commit inicial'
mrtin@maeve:~/dev/git/casero$ git add archivo_olvidado
mrtin@maeve:~/dev/git/casero$ git commit --amend
```

Trabajando con otros repositorios 'remote'

Hasta ahora, hemos visto como trabajar con un único repositorio, `origin`, que hasta ahora hemos configurado para que trabaje con nuestra cuenta de GitHub.

Siendo que git es distribuido, podemos trabajar con cualquier cantidad de repositorios remotos, siendo la única limitación, que estos sean en definitiva, el mismo repositorio.

Esto permite colaborar directamente con el resto del equipo, sin que los demás tengan una cuenta en GitHub, ¡cada miembro del equipo puede alojar su repositorio en servicios separados! No es algo particularmente recomendable, ya que solo sube la complejidad en la coordinación del proyecto. A pesar de ser distribuido, lo mejor que se puede hacer, es contar con un único repositorio central. La centralización tiene sus ventajas y es la razón por la cual los sistemas de control de versiones de este tipo no han desaparecido completamente.

Git aporta un nivel de flexibilidad a la hora de trabajar muy grande, tal vez demasiada. Esta se puede aprovechar para acomodar cualquier forma de trabajo sin que la herramienta se interponga, aunque para eso, la herramienta en si es bastante compleja.

Para esto, es necesario usar el comando `git remote add [nombre] [url]`;

```
mrtin@maeve:~/dev/git/casero$ git remote
origin
mrtin@maeve:~/dev/git/casero$ git remote add unrn https://github.com/INGCOM-UNRN/casero
mrtin@maeve:~/dev/git/casero$ git remote -v
origin    git@github.com:martinvilu/casero.git (fetch)
origin    git@github.com:martinvilu/casero.git (push)
unrn      git@github.com:INGCOM-UNRN/casero.git (fetch)
unrn      git@github.com:INGCOM-UNRN/casero.git (push)
```

Usar la opción `-v` acá muestra la dirección asociada al nombre del remote.

Ahora, podemos traer las modificaciones de este repositorio usando `git fetch`

```
mrtin@maeve:~/dev/git/casero$ git fetch unrn
remote: Counting objects: 43, done.
remote: Compressing objects: 100% (36/36), done.
remote: Total 43 (delta 10), reused 31 (delta 5)
Unpacking objects: 100% (43/43), done.
From  git@github.com:INGCOM-UNRN/casero.git
```

También podemos compartir cambios con ese remote

```
mrtin@maeve:~/dev/git/casero$ git push origin master
```

Sin embargo, esto solo funcionará si el repositorio al que estamos haciendo push tenemos permiso para hacer cambios o es de nuestra propiedad.

Etiquetado

Git tiene la posibilidad de marcar, o más específicamente, agregar una etiqueta a puntos específicos del historial como importantes. Esta funcionalidad se usa normalmente para resaltar versiones que fueron lanzadas [v1.0, por ejemplo].

```
mrtin@maeve:~/dev/git/casero$ git tag -a v1.4 -m 'Mi versión 1.4'
mrtin@maeve:~/dev/git/casero$ git tag
v0.1
v1.3
v1.4
```

En este ejemplo, se utilizan las opciones `-a/--annotate` para guardar una etiqueta completa, la cual guarda más información e incluso le podemos agregar un mensaje que es agregado con `-m`.

Podemos ver el detalle de la etiqueta usando `git show` indicando la etiqueta:

```
mrtin@maeve:~/dev/git/casero$ git show v1.4
tag v1.4
Tagger: Martín Vilugron <mrvilugron@unrn.edu.ar>
Date:   Sat May 8 20:19:12 2021 -0300

Mi versión 1.4

commit ca82a6dfff817ec66f44342007202690a93763949
Author: Martín Vilugron <mrvilugron@unrn.edu.ar>
Date:   Mon May 03 21:52:11 2021 -0300

    Arreglos otrograficos en el README.md
```

El comando muestra la información del etiquetador, la fecha en la que el commit fue etiquetado y el mensaje de la etiqueta, antes de mostrar la información del commit.

Las etiquetas por defecto, no son compartidas al momento de hacer un push, por lo que es necesario enviarlas explícitamente a un repositorio remoto.

```
mrtin@maeve:~/dev/git/casero$ git push origin v1.4
Counting objects: 14, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (12/12), done.
Writing objects: 100% (14/14), 2.05 KiB | 0 bytes/s, done.
Total 14 (delta 3), reused 0 (delta 0)
To git@github.com:martinvilu/casero.git
* [new tag]          v1.4 -> v1.4
```

También podemos enviar a todas las etiquetas juntas usando la siguiente instrucción.

```
mrtin@maeve:~/dev/git/casero$ git push origin --tags
Counting objects: 1, done.
Writing objects: 100% (1/1), 160 bytes | 0 bytes/s, done.
Total 1 (delta 0), reused 0 (delta 0)
To git@github.com:martinvilu/casero.git
* [new tag]          v1.3 -> v1.3
* [new tag]          v1.4 -> v1.4
```

De forma de que cuando alguien clone y traiga de tu repositorio, también tendrán las etiquetas.

Existe otro tipo de etiqueta llamado 'ligera' que no trataremos aquí y pueden ver más información en <https://git-scm.com/book/es/v2/Fundamentos-de-Git-Etiquetado>

GitHub

GitHub es una plataforma de colaboración construida sobre Git que permite a un equipo trabajar sobre el mismo código con varias herramientas adicionales que facilitan la tarea de desarrollo de software. Y a pesar de que no es la única herramienta de este tipo disponible, ofrece facilidades para la enseñanza que no están disponibles. Por lo que los invitamos a chusmear [GitLab](#) que es de código abierto y [Bitbucket](#) de Atlassian. Así y todo, no [son las únicas](#), pero estas permiten su utilización sin costo.

Esta parte tratará el uso que le daremos en clase, piensen que la herramienta es de una complejidad suficiente como para ameritar su propio apunte.

GitHub Classroom

Esta es la razón por la cual utilizamos GitHub en lugar de GitLab. La herramienta automatiza la creación de repositorios privados a partir de una plantilla para cada alumno de la cátedra, simplificando la gestión en cátedras numerosas.

Como el repositorio se crea en GitHub, no inicialicen uno en su computadora. Clonenlo para empezar a trabajar, esto es mucho más simple y fácil.

! Aunque es técnicamente posible enlazar un repositorio inicializado en nuestra computadora con el creado en GitHub, los pasos necesarios para esta tarea no son triviales y pueden provocar que la infraestructura para la corrección deje de funcionar.

La utilización de esta herramienta viene con un [Pull Request](#) que permite que los profesores agreguen comentarios directamente sobre cada línea de su programa y en todos los archivos que componen la entrega.

GitHub Actions

Esta herramienta forma parte de lo que se conoce como DevOps, o Development Operations, la cual consiste en la automatización de todas las tareas posibles en el desarrollo de software.

Esta herramienta permite ejecutar los programas de soporte del proyecto al momento que compartimos cambios a GitHub.

Las posibilidades de esta herramienta, en líneas generales es:

- I. Ejecutar los tests, podemos verificar que nuestro programa funciona como esperamos en el momento. Esto forma parte de lo que se conoce como análisis dinámico, ya que requiere que el programa sea ejecutado.
- II. El análisis estático, que no depende del funcionamiento del programa, revisan el código ante problemas comunes y conocidos. Estas herramientas son utilizadas también para garantizar que el código escrito siga alguna convención establecida.
- III. Generar los archivos de instalación, lo que se conoce como 'continuous release' o entrega continua.
- IV. Instalar, o 'desplegar' el programa de manera automática.
- V. Cualquier otra cosa a la que le podamos crear un script.

La utilización de esta herramienta requiere su configuración y armado del script, este es un tema abierto a contribuciones. De momento, utilicemos lo que los profesores tengan configurado en los repositorios de las actividades.

! Muy importante.
Tengan presente que todas estas tareas consumen recursos. Por lo que
■ no abusen de las facilidades haciendo `git push` frecuentes. Lo único que logran es que todos quedemos sin las herramientas de verificación automática, ya que estos recursos se descuentan de la cátedra en general.

Como prácticamente todo en su carrera de desarrollador de software, para mas información, consulten el manual de la herramienta <https://docs.github.com/es/actions>

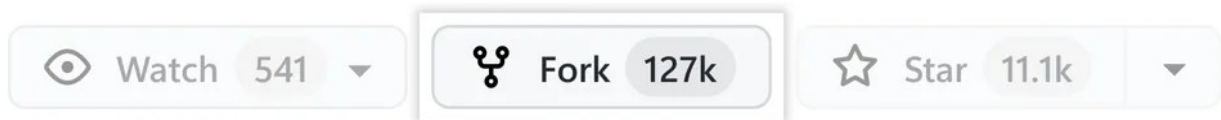
Forks

Cuando necesitamos trabajar con el repositorio de alguien más, aparte de subir commits directamente a este, la forma más fluida es la de hacer una copia para programar en forma separada. La razón por la cual es más fluida, tiene que ver con el concepto siguiente, el de [Pull Request](#).

Cuando hacemos un fork de un repositorio, obtenemos una instancia de todo el repositorio con todo su historial, en donde podemos hacer lo que queramos sin afectar la versión original.

Esto es utilizado cuando deseamos contribuir a un repositorio que es público, pero no podemos hacer cambios directamente. Luego de hacer el fork y hacer los cambios, podemos proponerlos al repositorio original para que sean incorporados.

Para ello es necesario hacer click en el 'tenedor'



Podemos elegir copiar solo una rama de desarrollo, pero lo normal es no hacer ningún cambio en el nombre o descripción.

La utilización de esta facilidad, retiene la conexión con el repositorio original, y nos permite luego enviar a revisión los cambios para que sean agregados al repositorio inicial.

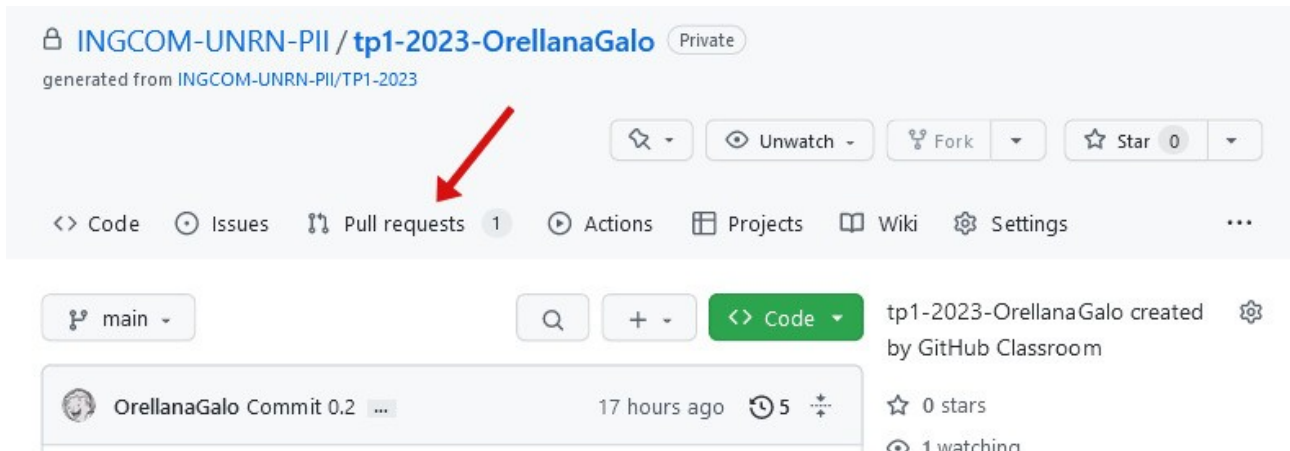
En el [manual de GitHub](#), está la forma de conectar nuestro fork con el repositorio original, para mantenernos actualizados de los cambios de existir. Pensando en proyectos más grandes, que los cambios son prácticamente diarios o incluso más rápidos.

Pull Requests, solicitudes de cambios

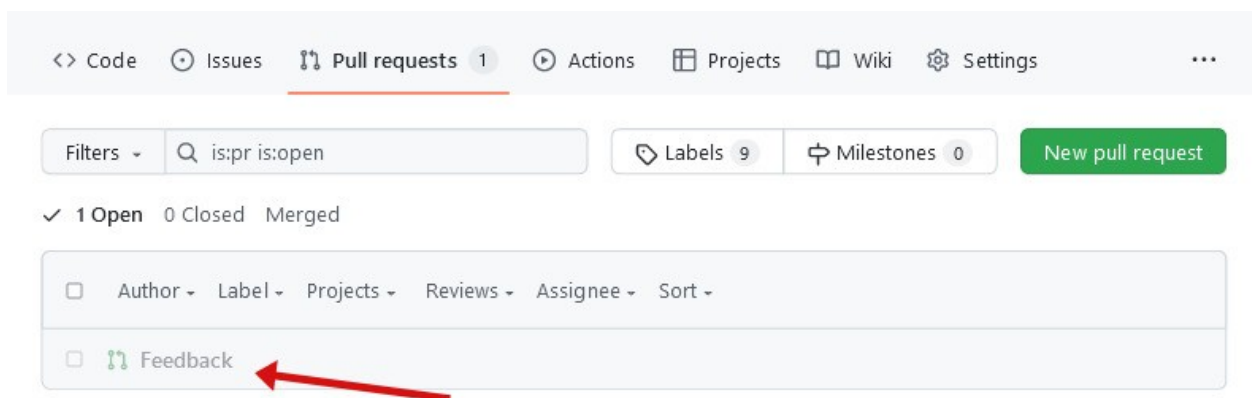
A lo largo de las materias que hagan uso de GitHub, le daremos gran uso a los Pull Requests como la forma de corregir y hacer el seguimiento a los Trabajos Prácticos.

Conceptualmente, un Pull Request se utiliza para recibir cambios hechos en forks de otros usuarios, y estos pueden ser solicitados por el aportante. En Github, existe todo un conjunto de facilidades para hacer comentarios y sugerencias en este aporte que son de suma utilidad para la corrección de ejercicios y que veremos a continuación.

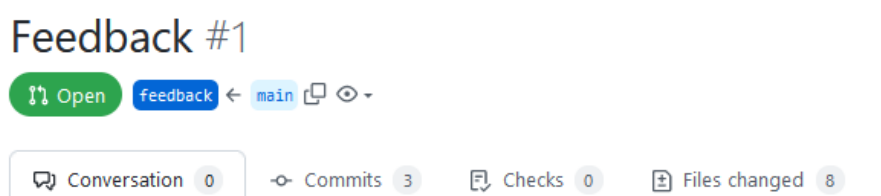
Actualmente, el detalle sobre esta herramienta, se encuentra en un apunte separado, aquí solo esta el uso base.



Ahi tendran la lista de Pull requests, en donde la que es creada automáticamente por GitHub Classroom se llama 'Feedback'.



Un pull request contiene información sobre la conversación, los commits hechos, chequeos automáticos (si existiesen) y la lista de archivos que han sido modificados.

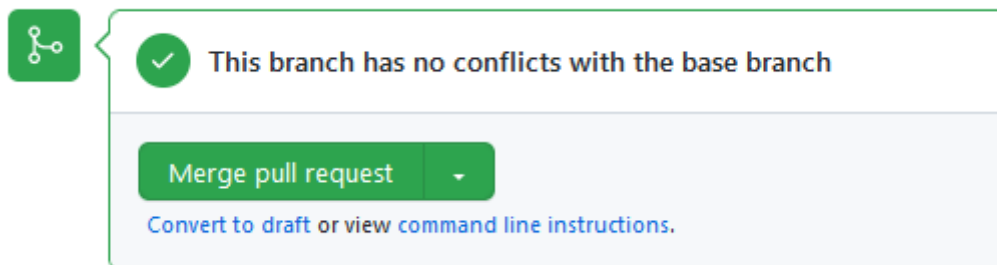


En la ficha Conversation, en donde al principio, el comentario de `github-classroom bot` que copio traducido a continuación:

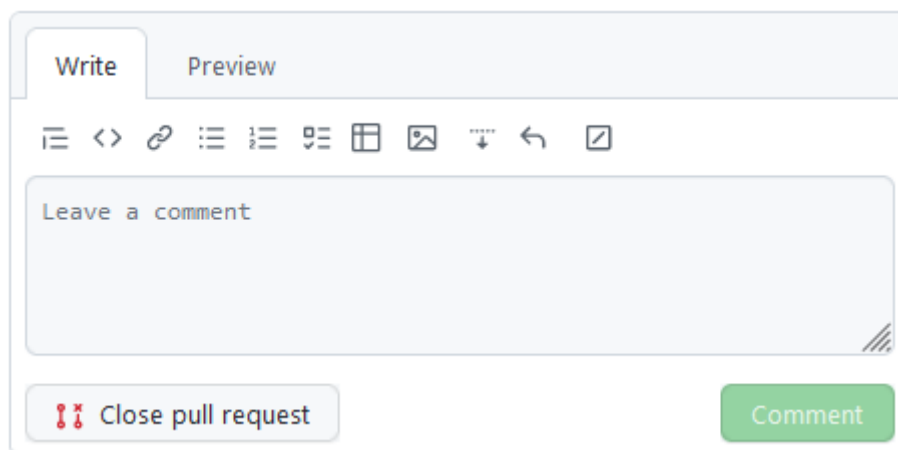
¡👋! GitHub Classroom creó este pull request como un lugar para que tu profesor deje comentarios sobre tu trabajo. Se actualizará automáticamente. No cierres ni fusiones este pull request, a menos que tu profesor te lo indique.
En este pull request, tu profesor puede dejar comentarios y opiniones sobre tu código. Haz clic en el botón Suscribirse para ser notificado si eso ocurre.
Haz clic en la pestaña Files changed o Commits para ver todos los cambios realizados en main desde que comenzó la tarea. Tu profesor también puede ver esto.

Aquí verán un resumen de la actividad de corrección hecha por el docente y sus commits. En la misma también pueden contestar a los comentarios hechos viendo en que línea fue hecho.

A no ser de que el profe se los haya pedido, no fusionen el pull request;



Esto no permite que se agreguen mas comentarios. Y puntualmente, no hace falta.



Lo mismo que el botón acá de "close pull request" si similar al anterior, pero este deja sin fusionar los cambios. Es una forma de cancelar que no se usa con GitHub Classroom.

Y al final, pueden hacer un comentario sobre la entrega en general.

Buenas prácticas

Estas buenas prácticas, no solo son recomendaciones, serán observadas en las correcciones, por lo que presten atención. Y aunque no cubren todo lo referido a la vida dentro del control de versiones, son un buen punto de entrada para no adquirir vicios en el uso de la herramienta que luego son muy difíciles de remover. Que, por otro lado, pueden provocar roces sin sentido con los compañeros de trabajo, aparte de hacerles la vida más fácil a ustedes mismos.

Usen mensajes de commit descriptivos

Solo toma un momento para escribir un buen mensaje de commit; no tiene porqué ser demasiado extenso. Como regla general, piensen en alguien más que tenga que revisar los cambios del proyecto.

Indicar el propósito del cambio es de lo mejor que pueden hacer, evitan a el potencial lector de los cambios tener que entrar en el cambio.

En GitHub también es posible utilizar referenciar directamente commits e issues indicando el número del mismo en los mensajes de commit.

Más información:

<https://docs.github.com/es/github/managing-your-work-on-github/linking-a-pull-request-to-an-issue>

Hacer de cada commit una unidad atómica

Cada commit debería tener un único propósito y debe implementar el mismo de manera completa. Esto hace más fácil escribir el mensaje de commit, relacionando el cambio referido a modificaciones, a alguna nueva parte del proyecto, esto hace que sea posible *deshacer* el cambio de ser necesario sin tener que buscar otros commits que tuvieron el mismo propósito.



Es frecuente pensar que es muy laborioso hacer esto, pero a la larga, ¡es de gran ayuda!

Miren antes de guardar

Eviten los commits indiscriminados, antes de hacer un commit, usen status para ver que lo que se está guardando tiene sentido con el mensaje (y forma parte del consejo “atómico”).

```
# Muestra todos los cambios y el estado de la copia de trabajo
$> git status
# Muestra los cambios, ayudando a redactar el mensaje de commit
$> git diff
# Y un commit de los archivos puntuales indicados
$> git commit archivo1 archivo2 -m "Mensaje de commit descriptivo sobre archivo1 y
archivo2"
```

También es posible hacer commit sobre archivos puntuales

Incorporen los cambios de los demás frecuentemente

Incorporar (fetch/pull) los cambios de los demás frecuentemente, baja el esfuerzo y reduce la probabilidad de que existan cambios que se superpongan. A pesar de que toda la información está disponible y al final, queda todo guardado, unir un cambio de manera manual por una modificación superpuesta requiere de trabajo adicional que puede ser evitado.

Compartí tus cambios frecuentemente

Es muy fácil perdernos en un largo día de trabajo y de la misma manera que tenemos que estar conectados con los repositorios de los demás para saber que están haciendo, también tenemos que pasar nuestros cambios en los repositorios para que los demás hagan lo mismo con nuestros cambios.

No olvides la coordinación con los co-equippers

Por más de que Git y GitHub sean excelentes herramientas de colaboración, no nos olvidemos de coordinar los esfuerzos con los demás. Al hacerlo, podemos evitar cambios superpuestos.

Parte de esta coordinación puede significar asignar a una persona una parte del programa para que se haga cargo de unir modificaciones. Git ofrece una enorme flexibilidad para acomodar todo tipo de formas de trabajo, pero antes de modificar algo compartido o que sea responsabilidad de otro, es mejor mandar un chat, email o llamada.

Recuerda que las herramientas están orientadas a líneas (de texto)

No es una buena idea cambiar la alineación y espaciado del texto, esto provoca cambios que no son más difíciles de procesar, ya que provocan cambios que fácilmente pueden superponerse con otros.

Por otro lado, no escribas líneas excesivamente largas; como regla general, mantenga cada línea en 80 caracteres. Cuantos más caracteres haya en una línea, mayor será la probabilidad de que varias ediciones caigan en la misma línea y, por lo tanto, entren en conflicto. Además, cuantos más caracteres, más difícil será determinar los cambios exactos al ver el historial de control de versiones.

! ■ Cuantos más caracteres haya en una línea, mayor será la probabilidad de que varias ediciones caigan en la misma línea y, por lo tanto, entren en conflicto.

Como otro beneficio para los autores del documento, las líneas de 80 caracteres también son más fáciles de leer al ver / editar el archivo de origen en la consola.

No guardes archivos generados (o intermedios)

Estos archivos, aparte de consumir espacio, son más archivos para mantener en el repositorio. Si estos se crean a partir de la información contenida en el repositorio, mejor guardar las instrucciones como para generarlo que guardarlo directamente.

En particular porque este tipo de archivos son más susceptibles a producir colisiones que es necesario unir, usen el `.gitignore` en la raíz del repositorio para simplemente olvidarse de este tema. Pueden consultar más sobre este tema en el manual oficial de git <https://git-scm.com/docs/gitignore>.

No uses la fuerza

En algunos casos, las herramientas se rehúsan a hacer lo que les pedimos, pero admiten una opción `-f/--force`. Esto es una mala idea que a la larga termina generando más trabajo para el equipo, con algo que pudo ser resuelto antes.

Soluciones a situaciones frecuentes

Se aceptan sugerencias de recetas de solución para problemas.

Quiero deshacer un cambio hecho sin commit.

Por accidente modificamos un archivo que esta dentro del repositorio y necesitamos volver a como estaba guardado.

```
$> git checkout archivo
```

Bibliografía

Atlassian. [s. f.]. *Aprende a usar git con bitbucket cloud / atlassian git tutorial*. Atlassian. Recuperado 9 de mayo de 2021, de <https://www.atlassian.com/es/git/tutorials/learn-git-with-bitbucket-cloud>

El control de versiones con git. [s. f.]. Recuperado 9 de mayo de 2021, de

<https://swcarpentry.github.io/git-novice-es/>

Git-Book. [s. f.]. Recuperado 9 de mayo de 2021, de <https://git-scm.com/book/es/v2>

Version control concepts and best practices. [s. f.]. Recuperado 9 de mayo de 2021, de

<https://homes.cs.washington.edu/~mernst/advice/version-control.html>

<https://zbib.org/925f540cdd6146279b20f750c15d1f25>