Curso profesional de Git y GItHub

Contenido

[¿Qué es Git? 3](#__RefHeading___Toc277_1488830239)

[Características importantes de Git 3](#__RefHeading___Toc279_1488830239)

[Directorios de Git 4](#__RefHeading___Toc281_1488830239)

[¿Qué es GitHub? 5](#__RefHeading___Toc283_1488830239)

[Herramientas propias de GitHub 6](#__RefHeading___Toc285_1488830239)

[¿Qué es un sistema de control de versiones? 7](#__RefHeading___Toc287_1488830239)

[Comandos Git 8](#__RefHeading___Toc289_1488830239)

[Git init 8](#__RefHeading___Toc291_1488830239)

[Git status 8](#__RefHeading___Toc293_1488830239)

[Git add 8](#__RefHeading___Toc295_1488830239)

[Git rm –cached 8](#__RefHeading___Toc297_1488830239)

[Git Commit 8](#__RefHeading___Toc299_1488830239)

[Git Tag 8](#__RefHeading___Toc301_1488830239)

[Git diff 9](#__RefHeading___Toc303_1488830239)

[Git reset 9](#__RefHeading___Toc305_1488830239)

[Git reset –soft 9](#__RefHeading___Toc307_1488830239)

[Git reset –mixed 9](#__RefHeading___Toc309_1488830239)

[Git reset –hard 10](#__RefHeading___Toc311_1488830239)

[Las ramas en git 10](#__RefHeading___Toc313_1488830239)

[trabajando entre ramas 11](#__RefHeading___Toc324_2333484919)

# ¿Qué es Git?

Git es un sistema de control de versiones que originalmente fue diseñado para operar en un entorno Linux. Actualmente Git es multiplataforma, es decir, que ahora ya no solamente es compatible con Linux, sino también con MacOS y Windows.

Un sistema de control de versiones es el conjunto de herramientas que tenemos disponibles para poder gestionar de una forma eficiente las distintas versiones de nuestros proyectos.

## Características importantes de Git

* Git se diferencia de otros sistemas de control de versiones en la forma en la que modela sus datos. Usualmente otros sistemas almacenan la información en una lista de cambios en archivos, mientras Git lo hace como un conjunto de archivos.
* La integridad con la que cuenta es bastante seria. No existen cambios, corrupción en archivos o cualquier alteración sin que Git lo sepa. Esto funciona gracias a una verificación con la que cuenta mediante un checksum, que es básicamente una suma de comprobación que se hace previo al almacenamiento de información.
* Casi todo en Git es local. Es difícil que se necesiten recursos o información externos, basta con los recursos locales con los que cuenta.

Git cuenta con 3 estados en los que podemos localizar nuestros archivos:

* Committed. Es la parte en la que nuestra información está segura alojada en nuestras bases de datos.
* Modified. En esta parte hemos realizado cambios en nuestros archivos, pero aún no se ven reflejados en nuestra base de datos.
* Staged. En esta parte marcamos nuestros archivos modificados dejándolos listos para confirmarlos.

## Directorios de Git

Esta es quizá la parte más importante de Git, ya que es el lugar donde se almacenan los metadatos y las bases de datos para nuestros proyectos, y es justamente lo que se copia cuando clonamos de un ordenador a otro los archivos.

Dentro de estos directorios es posible que no solamente tengamos nuestra información útil, sino también ciertos archivos de caché innecesarios, para ello es posible ignorar directorios sin afectar lo necesario de nuestros proyectos.

Una buena práctica para tener todo en orden y eliminar lo innecesario es crear un archivo .gitignore donde podamos incluir esos directorios que no dan valor a nuestros proyectos y así mantener solamente lo necesario.

¿Habías utilizado o escuchado de Git antes de este curso?

Cuéntanos en el sistema de discusiones cómo lo conociste y qué otras características te animaron a preferirlo.

# ¿Qué es GitHub?

GItHub es un servicio de alojamiento que ofrece a los desarrolladores repositorios de software usando el sistema de control de versiones, Git.

Existen varios aspectos que hacen que GitHub sea una opción eficiente para el control y gestión de tus proyectos.

Estos son algunos de ellos:

* GitHub permite que alojemos proyectos en repositorios de forma gratuita.
* Los repositorios son públicos, sin embargo Github tiene una forma de pago que te permite alojar tus proyectos de forma privada.
* Permite que puedas compartir tus proyectos de una forma mucho más fácil.
* Te permite colaborar para mejorar los proyectos de otros y a otros mejorar o aportar a los tuyos.
* Ayuda reducir significativamente los errores humanos, a tener un mejor mantenimiento de distintos entornos y a detectar fallos de una forma más rápida y eficiente.
* Es la opción perfecta para poder trabajar en equipo en un mismo proyecto.
* Ofrece todas las ventajas del sistema de control de versiones, Git, pero también tiene otras herramientas que ayudan a tener un mejor control de nuestros proyectos.

## Herramientas propias de GitHub

* Wiki. Esta herramienta ayuda en el mantenimiento necesario para las distintas versiones que tenga la página.
* Sistema de seguimiento de problemas. Esta herramienta es la que permite que otras personas, ya sean de tu equipo o no, puedan hacer mejoras, sugerencias y optimizaciones en nuestros proyectos. Así como también nos permite a nosotros hacerlo en los proyectos de otros.
* Herramienta de revisión de código. Esta herramienta nos permite poder hacer pequeñas anotaciones respecto a ciertas partes de un proyecto y con ello tener la colaboración de otros, creando una discusión de las mejoras y optimizaciones.
* Visor de ramas o branch. Nos permite conocer el progreso que llevamos en nuestros proyectos, mostrándonos cuántos updates hemos realizado a partir de nuestro master (primera versión del proyecto), así como también nos permite mirar los cambios que hemos realizado.

En este curso vas a poder aprender mucho más de cómo funciona GitHub y cómo usarlo.

Recuerda que todas tus dudas y/o aportes puedes dejarlos en el sistema de discusiones que encontrarás en cada clase.

# ¿Qué es un sistema de control de versiones?

Un sistema que registra los cambios realizados sobre un archivo o conjunto de archivos a lo largo del tiempo. Este tipo de sistemas nos permiten volver en el tiempo y salvar nuestro trabajo.

La idea con este curso es ser capaces de crear un proyecto del cual tendremos siempre las versiones que modificamos a lo largo del desarrollo.

Los tipos de sistemas de control son:

* **Local Computer:** Solo vive en nuestro computador.
* **Centralizado:** No depende únicamente de un computador en el que se trabaja, sino que depende del súper servidor en donde se almacena la información. El servidor provee las copias a sus hijos, pero solo guarda los cambios en un solo lugar.
* **Sistema de control distribuido:** Cada uno de los que participan en el proyecto, tienen copia del proyecto que se realiza, por eso no dependemos de un solo computador que almacene toda la información.

**NOTA:** Git es un Sistema de Control de Versiones Distribuido.

# Comandos Git

## Git init

Con git init, iniciamos un nuevo directorio de git, es como decir la carpeta contenedora de mi repositorio. Para borrar el repositorio borramos la carpeta .git

## Git status

Nos muestra el status de nuestro repositorio.

## Git add

Con este comando añadimos todos los archivos que creemos en nuestro repositorio.

## Git rm –cached

Recibe un parámetro que se refiere al nombre del archivo que deseamos eliminar de nuestro repositorio.

## Git Commit

Con este comando confirmamos o comprometemos nuestros archivos al repositorio, es decir, con este comando agregamos nuestros archivos del directorio de trabajo al repositorio.

## Git Tag

Con este comando podremos versionar nuestro proyecto

Creando un tag

***Git tag –a <número de versión> -m ‘comentarios de la versión’***

Borrando un tag

***Git tag –d <número del tag>***

Para renombrar un tag

***Git tag –f –a <número de versión> -m ‘comentarios’***

Luego eliminamos el tag al que realizamos el renombramiento

## Git diff

Con este comando podemos ver las diferencias entre los Commit que hemos realizado, algo así como una comparación entre las versiones, esto nos servirá al momento de consultar los cambios que hemos realizado en nuestro proyecto.

Comparando Commit por los tag

***Git diff <versión 1> <versión 2>***

También podemos comparar los Commit con el número de SHA

## Git reset

Con el git reset podemos volver a una versión anterior de nuestro proyecto, existen varios comando que mostraremos a continuación.

Debemos tomar en cuenta que reset recibe como parámetro el SHA de nuestro Commit desde el cual queremos quitar los cambios, es decir, debemos colocar el SHA que queremos que se mantenga en nuestro master

## Git reset –soft

Este comando nos va a quitar los cambios de mi Commit seleccionado y dejara todo listo para hacer un git add a nuestro proyecto, cabe destacar que no elimina los archivos de nuestro directorio

***Git reset –soft <número SHA del Commit que queremos mantener>***

## Git reset –mixed

Este comando a diferencia del git –soft nos elimina todos los cambios que hemos añadido al repositorio, es decir, los add que realizamos en el Commit actual no estarán disponibles, debemos volver a agregarlos.

***Git reset –mixed <número SHA del Commit que queremos mantener>***

## Git reset –hard

Con este comando eliminaremos todo, tanto del staging como del working directory, no nos quedara nada de lo que habríamos realizado después del Commit especificado.

Aun podemos recuperar la información si tenemos el SHA del Commit posterior.

# Las ramas en git

Las ramas en git son de mucha importancia porque nos permiten trabajar en nuestro proyecto de forma ordenada, y definir los privilegios que cada colaborador tendrá en nuestro proyecto.

Crear ramas <nombre de la nueva rama>

**git branch <nombre de la rama>**

para crear una rama y moverse a ella de una vez hacemos lo siguiente.

**Git checkout -b <nombre de la rama>**

listar ramas.

**Git branch -l**

para eliminar las ramas hacemos.

**Git branch -d <nombre de la rama>** con esto borramos y para forzar el borrado reemplazamos la -d por una -D

para renombrar las ramas ya creadas.

**Git branch -m <nombre actual> <nombre nuevo>**

## trabajando entre ramas

Una vez tenemos las ramas creadas en nuestro proyectos y deseamos luego de hacer el commit en mi nueva rama, mezclar con el master de mi proyecto hacemos lo siguiente.

Nos ubicamos en la rama en la cual queremos que se mezclen nuestros cambios, en nuestro caso **master**

**Git merge <nombre de la rama que queremos mezclar>**

existiran momentos en los que debas atender cambios en otra rama mas sin embargo no tienes todos los datos en tu rama actual como para hacer un commit, que hacer en ese caso en git existen los **stash** que no son mas que guardados temporales que podemos hacer en estos casos en los que ecesitamos salir de nuestra rama sin haber terminado los cambios. Para los stash tenemos los siguientes comandos.

Git stash – con esto hacemos un guardado del estado, ya con esto podemos movernos y trabajar en la rama que deseamos seguir

para recuperar mis stash hacemos lo siguiente.

**Git stash list** nos listara los stash que tengamos en esta rama

los stash se nos guardaran bajo la siguiente sintaxis stash@{0}

podemos borrar stash que es una buena practica una vez que hacemos el commit

**git stash drop stash@{numero del stash}**

para aplicar el stash que hicimos basta con usar la siguiente linea

**git stash apply –**aplicara el ultimo stash que hicimos, si tenemos muchos stash y queremos aplicar uno en especifico basta con agregar al comando anterior un **stash@{numero de stash}**

presentamos aquí una lista de comando stash que pueden ser de ayuda

**git stash** : Guarda el trabajo actual de manera temporal. (Archivos modificados o eliminados)

**git stash -u** : Crea un stash con todos los archivos. (Añadiendo los creados Untracked)

**git stash list** : Permite visualizar todos los stash existentes.

**git stash clear** : Elimina todos los stash existentes.

**git stash show stash@{num\_stash}** : Muestra los archivos cambiados en ese stash.

**git stash show -p stash@{num\_stash}** : Muestra los archivos cambiados en ese stash en detalle.

**git stash drop stash@{num\_stash}** : Elimina un stash específico.

**git stash apply** : Aplica el stash más reciente. El que tiene num\_stash=0.

**git stash apply stash@{num\_stash}** : Aplica los cambios de un stash específico.

**git stash pop** : Aplica el cambio más reciente y elimina el stash.

**git stash pop stash@{num\_stash}** : Aplica los cambios de un stash específico y elimina el stash.

# Que es cherry pick

Si estás trabajando en una rama, pero de repente notas que hiciste un cambio en la rama que no debías, para esto podemos usar cherry pick. Este comando nos puede salvar la vida, ya que nos permite sacar cambios específicos de una rama y mezclarlos en otra.

Para cambiar el commit de la rama y agregarla en la correcta hacemos lo siguiente

copiamos el sha de nuestro commit nos vamos a la rama master y creamos la rama en la que queremos que este nuestro commit seleccionado, nos dirigmos a ella y tecleamos

**git cherry-pick <sha del commit>**

y ahora si podemos mesclar esta rama con el commit deseado, con nuestra master del proyecto

# GitHub

En git podemos clonar o hacer un fork de otro proyecto sobre el cual querramos tener una imagen. Con git clone **<direccion url o ssh del repositorio de github>**

## Git remote

Con git podemos agregar un repositorio de git hub en nuestra pc, con el siguiente comando.

**Git remote add <nombre del repositorio> <ruta ssh o http de nuestro repositorio en github>**

Ahora bien si necesitamos saber si nuestro repositorio local ya tiene agregado un repositorio de github ejecutamos el siguiente comando.

**Git remote -v**

y nos traera un msj con la descripcion de el repositorio de github que añadimos.

Si resulta que colocaste un repositorio que no era el que querias tambien puedes removerlo con el siguiente comando.

**Git remote remove <nombre del repositorio>**

para usar git y github es necesario que sepamos como agregar los cambios de nuestro repositorio a github, si no no tendria ciencia todo esto.

Vamos a compartir nuestro código con el repositorio remoto. Antes de poder enviar cambios tengo que tener los mismos cambios en el repositorio remoto que en el repositorio local.

Para traer los cambios del repositorios remoto vamos a usar el comando git fetch. Este comando recibe dos parámetros 1. desde donde y hacia donde van.

Luego de traer los cambios, debemos mezclarlos con la rama que estamos usando en local. Para esto usamos git merge.

Para traer los datos de mi repositorio de github hacemos lo siguiente

**git fetch <nombre del repositorio> <rama de nuestro repositorio github>**

luego hacemos un merge para mezclar los datos de nuestro github con nuestro repositorio local.

**Git merge <nombre del repositorio/rama>**

si nos da algun error agragamos esto.

**Git merge <> --allow-unrelated-historie**

luego nos pedira que hagamos un commit para guardar los cambios de la mezcla

nos podemos ahorrar tanto comando usando un

**Git remote pull <nombre del repositorio github> <rama>**

obviamente tambien debemos saber como enviar los cambios que hacemos en nuestro repositorio local a github esto lo hacemos con un.

**Git push <nombre del repositorio remoto> <rama>**

para agregar los tags de nuestro repositorio local hacemos

**git push <nombre de repositorio remoto> <rama> --tags**

y por su puesto tambien podemos enviar nuestras ramas a github para tener una copia exacta de todo y lo hacemos con.

**Git push <nombre del repositorio remoto> <rama>**

# Exploremos que nos ofrece github

* Unwatch (Nos permite que nos lleguen notificaciones si hay cambios en el repositorio)
* Not watching (Se te notifica cuando participas o mencionas)
* Ignoring (Nunca ser notificado)
* Star (Asi como en facebook se vive a base de likes, aqui se vive a base de estrellas, cuanto mas estrellas mas valorado es ese proyecto, y a su misma vez tiene muchos forks)
* Fork (Indica la cantidad de gente que hizo copia de su repositorio para poderles enviar cambios)
* Podemos ver las estadísticas de nuestro proyecto en Insights

Tambien tenemos los siguientes apartados:

**contributors** (Nos permite ver el avance de nuestro proyecto)

**traffic** (Nos muestra el trafico de nuestro proyecto)

**punch card** (Muestra los dias que a estado trabajando)

**network** (Vemos el flujo de ramas)

**members** (Nos muestra la gente que le a hecho fork a nuestro repositorio)

**dependents** (Nos muestra si nuestro repositorio depende de otro, es decir de un super repositorio)

‘‘LeonidasEsteban’’ Para proyectos complejos se puede hacer un “dependecia” de repositorios o sub repositorios. Es como un proyecto dentro de otro pero es muy raro usarlos y de preferencia yo solo recomiendo un solo reposotorio’.

**Settings**:

**collaborators** (Aqui se añade a mas gente para que colabore en este proyecto)

**branches** (Podemos cambiar la rama principal, y proteger ramas)

CONSEJO: Protege siempre la rama master (Una vez protegida ningun cambio va a ir directamente a master tenemos que pasar por cierto proceso para que un codigo vaya a master) (Este proceso lo vemos dentro del curso)

**pull request** (Es una forma de tomar una rama alterna por ejemplo la rama RD y no mezclarla directamente con master. Si no enviar una solicitud de mezcla desde la rama RD a la master, asi alguien mas puede ver esa solicitud, algun colaborador, etc…Y se haga un code review)

**Webhooks** (Es la forma de poderle agregar git code a otros servicios para que hagan un auto deploy)

**Integrations & servicies** (Incluir otros servicios como amazon, y que pueden hacer uso de nuestro repositorio)

**Deploy keys** (Son las llaves publicas ssh de tu servidor para poder hacer git pull desde el servidor, y tener tu código en producción.)

**Pull requests** (Vemos una lista de pull requests)

**Issues** (Son la forma en que manejamos nuestros problemas, bugs en github. Por ejemplo: hay un bug, este no va a ser reportado si no que lo que vamos a hacer es crear un issue comentando donde hay un problema, se hace de esta manera, ocurre en tal sistema operativo, etc…)

**Proyects** (Nos aparece nuestros proyectos)

Al crear un “issue” o un “pull request”, github te permite incluir etiquetas en los mismos. Entre las etiquetas que nos pueden interesar a los que estamos comenzando a usar la plataforma se encuentran “help wanted” y “good first issue”. Github anima a los desarrolladores a que marquen los “issue” y los “pull request” que son más aptos para los que inician en este mundo con estas etiquetas y adicionalmente se pide a la comunidad que den un apoyo especial para estos issue/pull request. ¿Quiéres comenzar a colaborar en un proyecto real en Github pero aún te sientes inseguro con tus conocimientos? Haz una búsqueda de estás etiquetas y encontrarás miles de oportunidades para hacer tu primer aporte.