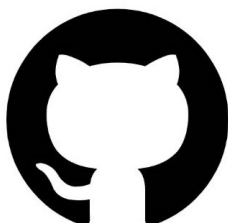


CURSO DE PROGRAMACIÓN FULL STACK

TUTORIAL: GUÍA INICIAL DE USO DE GIT CON GITHUB





Objetivos de la Guía

En esta guía aprenderemos a:

- Instalar git
- Vincular el access token
- Configurar GIT en el ordenador
- Utilizar los comandos básicos
 - Git Add
 - Git Commit
 - Git Push
 - Git Pull
 - Git Status
- Guardar los ejercicios realizados en las guías siguientes en tu repositorio

¿QUÉ ES EL CONTROL DE VERSIONES?

Los **sistemas de control de versiones** son programas que tienen como objetivo controlar los cambios en el desarrollo de cualquier tipo de *software*, permitiendo conocer el estado actual de un proyecto, los cambios que se le han realizado a cualquiera de sus piezas, las personas que intervinieron en ellos, etc. El *software* de control de versiones realiza un seguimiento de todas las modificaciones en el código en un tipo especial de base de datos. Si se comete un error, los desarrolladores pueden ir atrás en el tiempo y comparar las versiones anteriores del código para ayudar a resolver el error al tiempo que se minimizan las interrupciones para todos los miembros del equipo.

Los desarrolladores que trabajan en equipos están escribiendo continuamente nuevo código fuente y cambiando el que ya existe. El código de un proyecto, una aplicación o un componente de *software* normalmente se organiza en una estructura de carpetas o "árbol de archivos". Un desarrollador del equipo podría estar trabajando en una nueva función mientras otro desarrollador soluciona un error no relacionado cambiando código. Cada desarrollador podría hacer sus cambios en varias partes del árbol de archivos.

El control de versiones ayuda a los equipos a resolver estos tipos de problemas, al realizar un seguimiento de todos los cambios individuales de cada colaborador y ayudar a evitar que el trabajo concurrente entre en conflicto.

En definitiva, tener un control de los cambios en los códigos de nuestra aplicación es una variable crucial para el éxito de nuestro desarrollo. **Git** es un sistema de control de versiones de código abierto, diseñado para manejar grandes y pequeños proyectos con rapidez y eficiencia. La pretensión de este tutorial es abordar el uso básico de Git proporcionando ejemplos prácticos útiles para comenzar a administrar repositorios remotos con plataformas como **Bitbucket o GitHub**.

¿QUÉ ES GIT?

Git es la mejor opción para la mayoría de los equipos de *software* actuales. Aunque cada equipo es diferente y debería realizar su propio análisis, aquí recogemos los **principales motivos** por los que destaca el control de versiones de Git con respecto a otras alternativas:

GIT ES UNA EXCELENTE HERRAMIENTA

Git tiene la funcionalidad, el rendimiento, la seguridad y la flexibilidad que la mayoría de los equipos y desarrolladores individuales necesitan. En las comparaciones directas con gran parte de las demás alternativas, Git resulta muy ventajoso para muchos equipos.

GIT ES UN PROYECTO DE CÓDIGO ABIERTO DE CALIDAD

Git es un proyecto de código abierto muy bien respaldado con más de una década de gestión de gran fiabilidad. Los encargados de mantener el proyecto han demostrado un criterio equilibrado y un enfoque maduro para satisfacer las necesidades a largo plazo de sus usuarios con publicaciones periódicas que mejoran la facilidad de uso y la funcionalidad. La calidad del *software* de código abierto resulta sencilla de analizar y un sin número de empresas dependen en gran medida de esa calidad.

Git goza de una amplia base de usuarios y de un gran apoyo por parte de la comunidad. La documentación es excepcional y para nada escasa, ya que incluye libros, tutoriales y sitios web especializados, así como podcasts y tutoriales en vídeo.

El hecho de que sea de código abierto reduce el costo para los desarrolladores aficionados, puesto que pueden utilizar Git sin necesidad de pagar ninguna cuota. En lo que respecta a los proyectos de código abierto, no cabe duda de que Git es el sucesor de las anteriores generaciones de los exitosos sistemas de control de versiones de código abierto, SVN y CVS.

¿QUÉ ES GITHUB?

Github es un portal creado para alojar el código de las aplicaciones de cualquier desarrollador. La plataforma está creada para que los desarrolladores suban el código de sus aplicaciones y herramientas, y que como usuario no solo puedas descargarla la aplicación, sino también entrar a su perfil para leer sobre ella o colaborar con su desarrollo.

Un sistema de gestión de versiones es ese con el que los desarrolladores pueden administrar su proyecto, ordenando el código de cada una de las nuevas versiones que sacan de sus aplicaciones para evitar confusiones. Así, al tener copias de cada una de las versiones de su aplicación, no se perderán los estados anteriores cuando se va a actualizar.

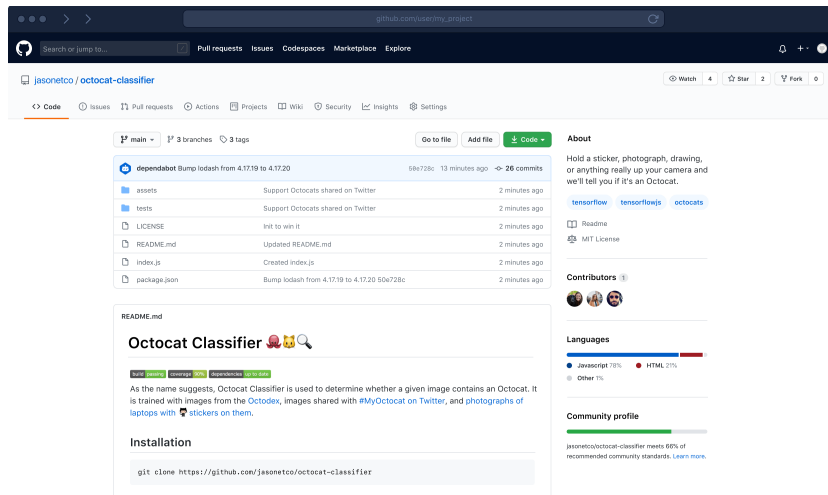
Git, al ser un sistema de control, va a ser la herramienta que nos va a permitir comparar el código de un archivo para ver las diferencias entre las versiones, restaurar versiones antiguas si algo sale mal, y fusionar los cambios de distintas versiones.

Así pues, Github es un portal para gestionar proyectos usando el sistema Git.

¿CÓMO VAMOS A LOGRAR ESO?

GitHub permite que los desarrolladores alojen proyectos creando repositorios de forma gratuita. Un repositorio es como una carpeta para tu proyecto. El repositorio de tu proyecto contiene todos los archivos de tu repositorio y almacena el historial de revisión de cada archivo. También puedes debatir y administrar el trabajo de tu proyecto dentro del repositorio.

¿Cómo se ve en GitHub?



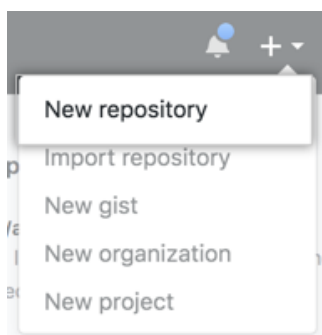
MANOS A LA OBRA!

¿COMO CREAR REPOSITORIO EN GITHUB?

Para esto primero deberemos **crearnos una cuenta en GitHub**, recomendamos ir a GitHub y **crearse una cuenta**, antes de seguir con la guía.

Para subir tu proyecto a GitHub, deberás crear un repositorio donde alojarlo. Para poder crear un repositorio deberemos crearnos una cuenta en GitHub.


- 1) En la esquina superior derecha de cualquier página, utiliza el menú desplegable + y selecciona **Repositorio Nuevo**



- 2) Escribe un nombre corto y fácil de recordar para tu repositorio. Por ejemplo: "hola-mundo".

Create a new repository

A repository contains all the files for your project, including the revision history.

Owner:  octocat ▾

Repository name: ✓


Great repository names are short and memorable. Need inspiration? How about **potential-eureka**.

Description (optional):

- 3) También puedes agregar una descripción de tu repositorio. Por ejemplo, "Mi primer repositorio en GitHub".

Create a new repository

A repository contains all the files for your project, including the revision history.


Owner:  octocat ▾ / Repository name: ✓


Great repository names are short and memorable. Need inspiration? How about **potential-eureka**.


Description (optional):

- 4) Elige la visibilidad del repositorio. Puedes restringir quién tiene acceso a un repositorio eligiendo la visibilidad de un repositorio: público o privado. Publico significa que cualquier persona puede ver ese repositorio y privado significa que solo personas autorizadas pueden verlo. Que sea publico no significa que la gente puede subir cosas a nuestro repositorio, lo único que permite es que se puedan ver los archivos.

Description (optional):

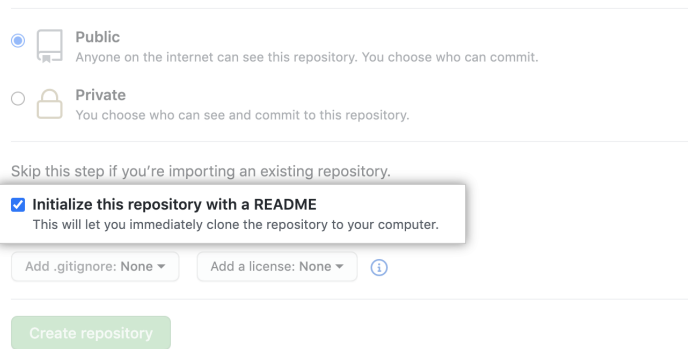
☒  **Public**
Anyone can see this repository. You choose who can commit.

☐  **Internal**
Octo Corp **enterprise members** can see this repository. You choose who can commit.

☐  **Private**
You choose who can see and commit to this repository.

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

5) Podemos crear el repositorio con un ReadMe



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

☐ **Private**
You choose who can see and commit to this repository.

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

☒ **Initialize this repository with a README**
This will let you immediately clone the repository to your computer.

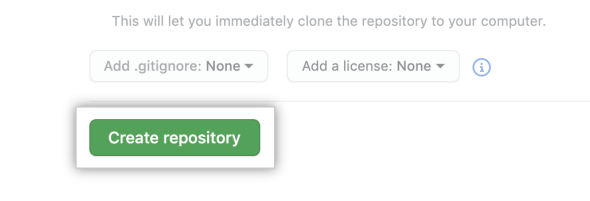
Add .gitignore: None ▾ Add a license: None ▾ ⓘ

Create repository



Un archivo README es un archivo donde puedes escribir una descripción detallada de tu proyecto.

6) Da click en Crear repositorio.



This will let you immediately clone the repository to your computer.

Add .gitignore: None ▾ Add a license: None ▾ ⓘ

Create repository



Revisemos lo aprendido hasta aquí

- Crear tu cuenta en GitHub
- Crear tu repositorio

¿CÓMO INSTALAMOS GIT?

Una vez que tenemos creado un repositorio en GitHub vamos a tener que instalar Git

INSTALADOR DE GIT PARA MAC

El modelo de Git de Apple viene preinstalado en macOS. Abra su terminal o editor de scripts de shell de su elección e **ingrese git --version** para verificar qué versión de Git está en su máquina. Si aún no está en su máquina, ejecutar **git --version** le pedirá que instale Git.

Si bien esta compilación de Git está bien para algunos usuarios, es posible que desee instalar la versión más actualizada. Puede hacerlo de muchas formas diferentes; hemos recopilado algunas de las opciones más fáciles a continuación. Recordamos que si ya tenemos Git en nuestra Mac, no es necesario instalarlo, solo si queremos una versión más nueva.

1) Una forma es instalar Git en un Mac es mediante el instalador independiente:

A. Descarga el instalador de Git para Mac más reciente desde acá:

<https://sourceforge.net/projects/git-osx-installer/files/>

B. Sigue las instrucciones para instalar Git.

Abre un terminal y escribe el siguiente texto para comprobar que la instalación se ha realizado correctamente:

`git --version:`

```
$ git --version
git version 2.9.2
```

Nota: esta es la opción que más recomendamos

2) Otra forma es instalar Git mediante HomeBrew:

Esta opción **SOLO** la vamos a utilizar si no nos funcionó la opción anterior, de lo contrario obviar esta parte de la guía.

Homebrew instala una lista de paquetes útiles que no vienen preinstalados en Mac.

A. Pegue el siguiente comando en su terminal para instalar Homebrew:

```
/bin/bash -c "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/HEAD/install
.sh)"
```

B. El terminal le pedirá que ingrese una contraseña. Ingrese la contraseña que usa para iniciar sesión en su Mac y continuar con el proceso de instalación.

C. Una vez terminado, ingrese **brew install git** en la terminal y espere a que se descargue. Verifique que Git se instaló ejecutando **git --version**

INSTALADOR DE GIT PARA LINUX

Debian / Ubuntu (apt-get)

Los paquetes de Git están disponibles mediante APT

1. Desde tu núcleo, instala Git mediante apt-get:

```
$ sudo apt-get update
$ sudo apt-get install git
```

2. Introduce el siguiente texto para verificar que la instalación se ha realizado correctamente:

```
git --version:
```

```
$ git --version  
git version 2.9.2
```

INSTALADOR DE GIT PARA WINDOWS

1. Descarga el instalador de Git para Windows más reciente, lo podés encontrar en tu aula virtual,
2. Cuando hayas iniciado correctamente el instalador, verás la pantalla del asistente de instalación de Git.
3. Selecciona las opciones Siguiente y Finalizar para completar la instalación. Las opciones predeterminadas son las más lógicas en la mayoría de los casos.
4. Abre el símbolo del sistema (o Git Bash si durante la instalación seleccionaste no usar Git desde el símbolo del sistema de Windows).
5. Introduce el siguiente texto para **verificar que la instalación se ha realizado correctamente:**

```
git --version:
```

```
$ git --version  
git version 2.9.2
```



¿Pudiste realizar la instalación? Pregunta a tus compañeros si necesitan ayuda



Revisemos lo aprendido hasta aquí

- Instalar Git en tu ordenador

¿QUÉ ES EL GIT TOKEN EN GITHUB?

Para poder subir nuestros cambios o nuestras tareas a GitHub, vamos a tener que configurar nuestro **Personal Access Token**, este le permite a GitHub autenticar que eres tú, el que está tratando de subir cosas al GitHub. De modo que puedas realizar todo tipo de operativas, como clonar repositorios o enviar cambios a GitHub desde local a remoto.

En GitHub han declarado recientemente como obsoleta la autenticación por medio de usuario y clave y ahora los desarrolladores debemos autenticarnos a través de lo que llaman el Personal Access Token. Esta acción debes hacerla para poder conectarte y realizar cualquier tipo de operativa con el servicio de GitHub.

POR QUÉ NECESITAMOS EL PERSONAL ACCESS TOKEN

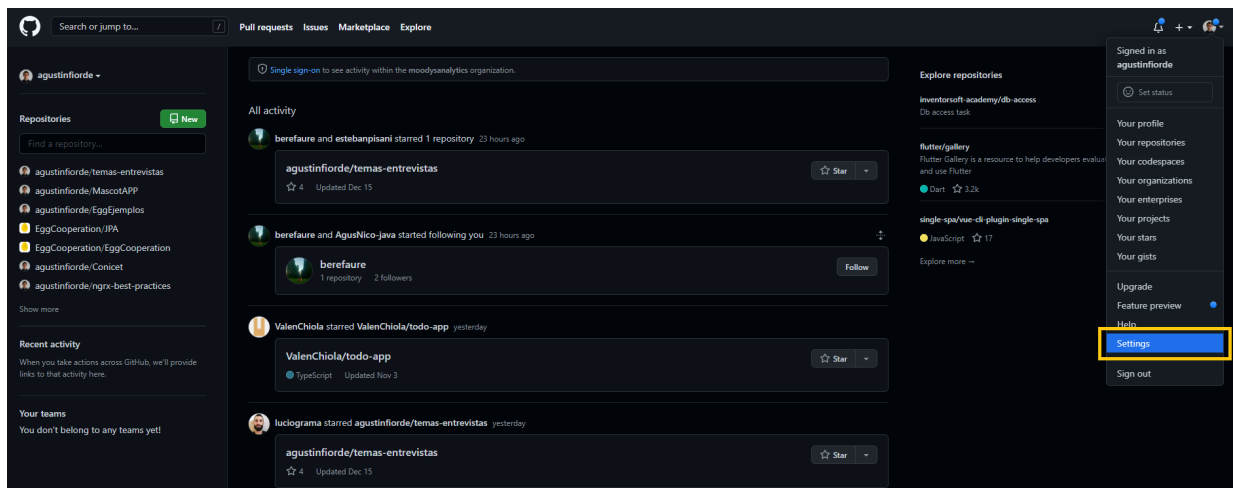
Anteriormente, cuando querías realizar cualquier operación en GitHub desde local y esa operación requería de acceso autenticado, ya sea mediante la línea de comandos o un programa de interfaz gráfica, podrías simplemente autenticarte usando el mismo usuario y contraseña que utilizas en el sitio web de GitHub. Ahora no es posible y tendrás que usar el mencionado Personal Access Token.

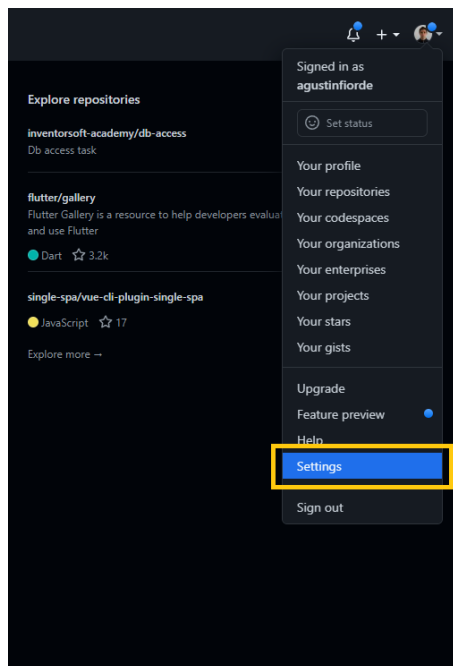
Este procedimiento lo tendrás que hacer en **todo lugar donde necesites una clave para poder utilizar GitHub desde fuera del sitio web**. Por ejemplo, clonar un repositorio privado de GitHub en local, enviar cambios a GitHub con Push, traerte cambios con Pull, etc.

CÓMO CREAR UN PERSONAL ACCESS TOKEN (PAT)

Este token se crea desde el sitio web de GitHub. El procedimiento es muy sencillo. Una vez logueado con tu usuario y contraseña en el sitio web de GitHub, accedemos a la opción **Settings** y luego a **Developer settings** y por último a **Personal Access Tokens**.

La opción de **Settings** la encuentras en el menú desplegable de tu avatar arriba a la derecha de la pantalla.

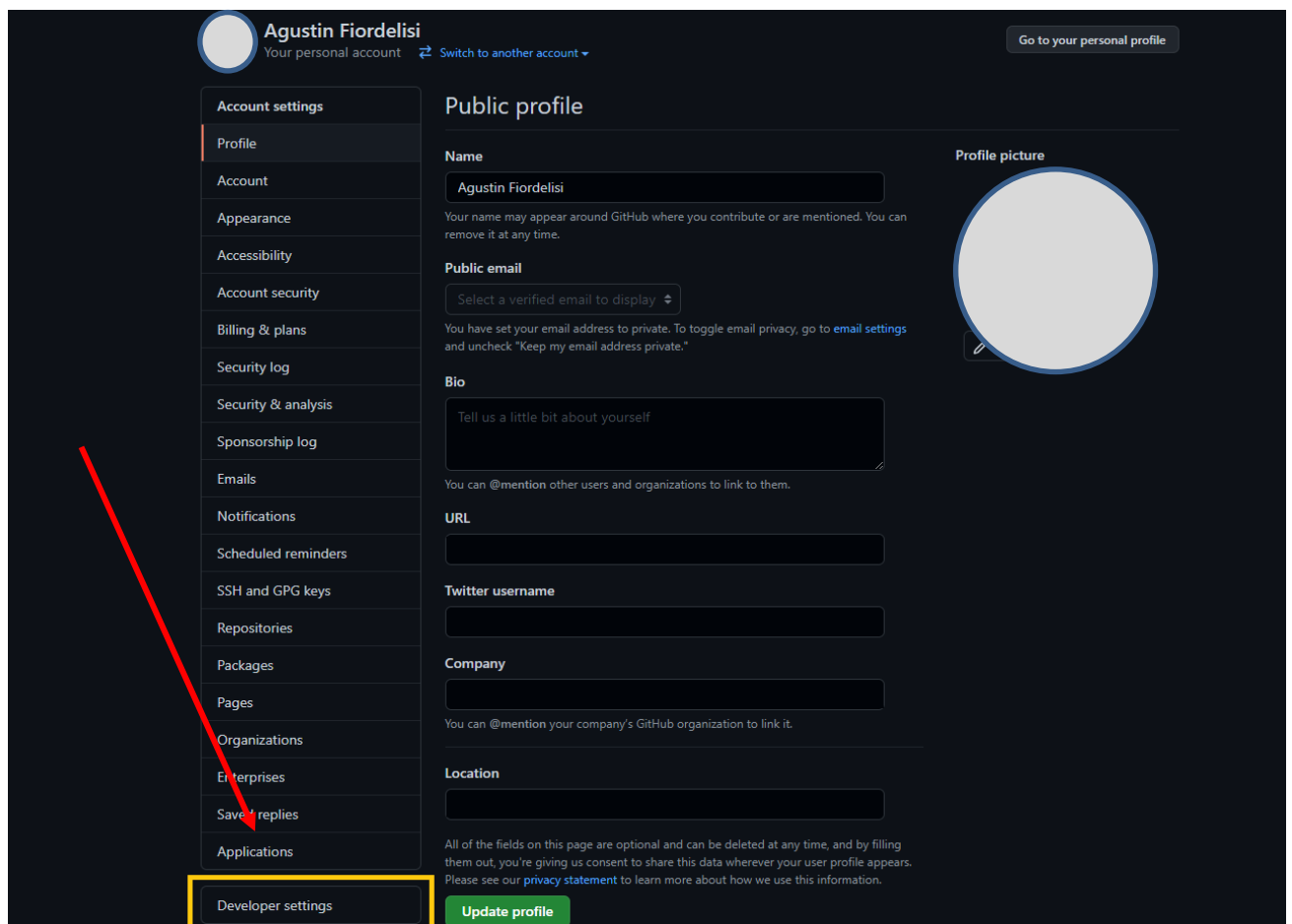


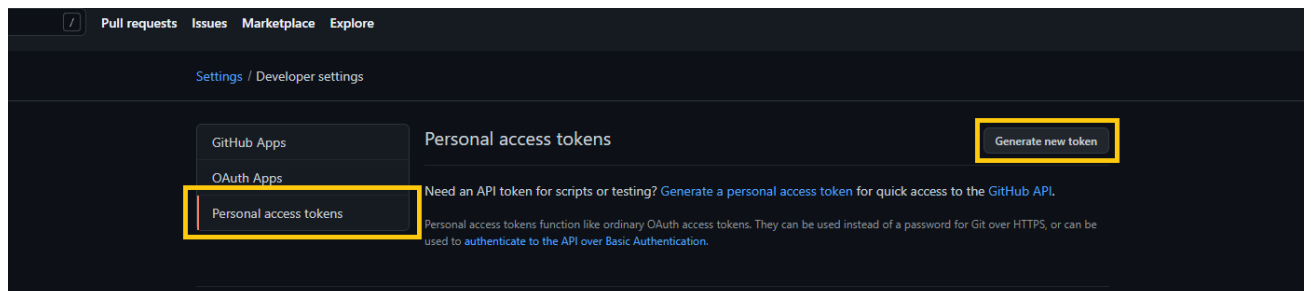


Las siguientes opciones las accedes mediante el navegador vertical de la izquierda. Ahí pulsamos la opción **Developer Settings** y luego **Personal Access Tokens**.

Entonces accederás a una página donde puedes administrar tus *Personal Access Tokens*.

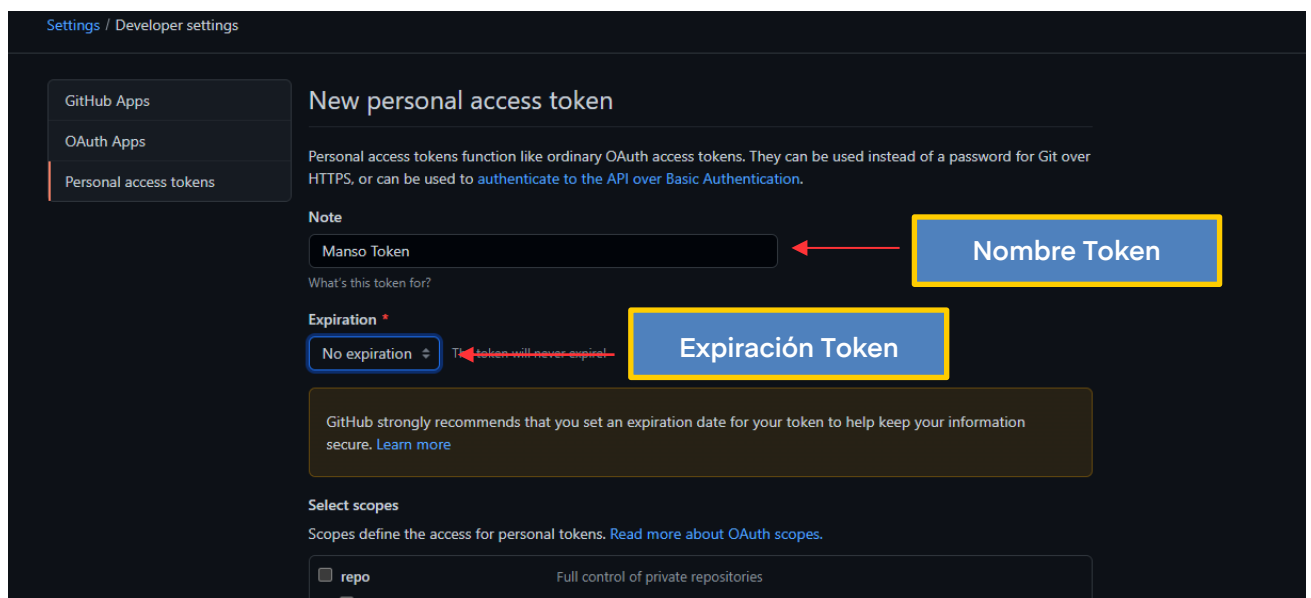
Aquí vamos a crear un nuevo token con el botón **Generate new token**.





Entonces aparecerá un formulario que debes rellenar, indicando el **nombre del token** (para saber en qué lo vas a utilizar) y otros detalles como la **expiración del token** y los **scopes**.

Para el nombre vamos a poner el **nombre que queramos**, en expiración le pondremos **No expiration (No vencimiento)** y en los scopes pondremos solo la **opción de repo**.

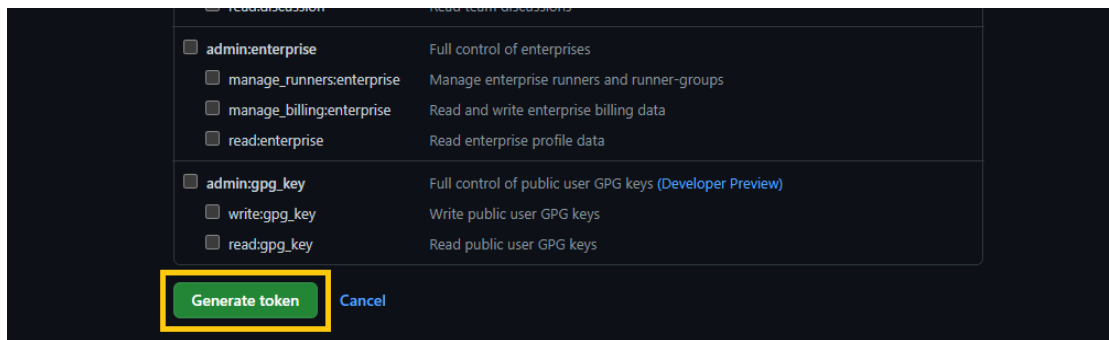


Scopes:

Select scopes

Scopes define the access for personal tokens. [Read more about OAuth scopes.](#)

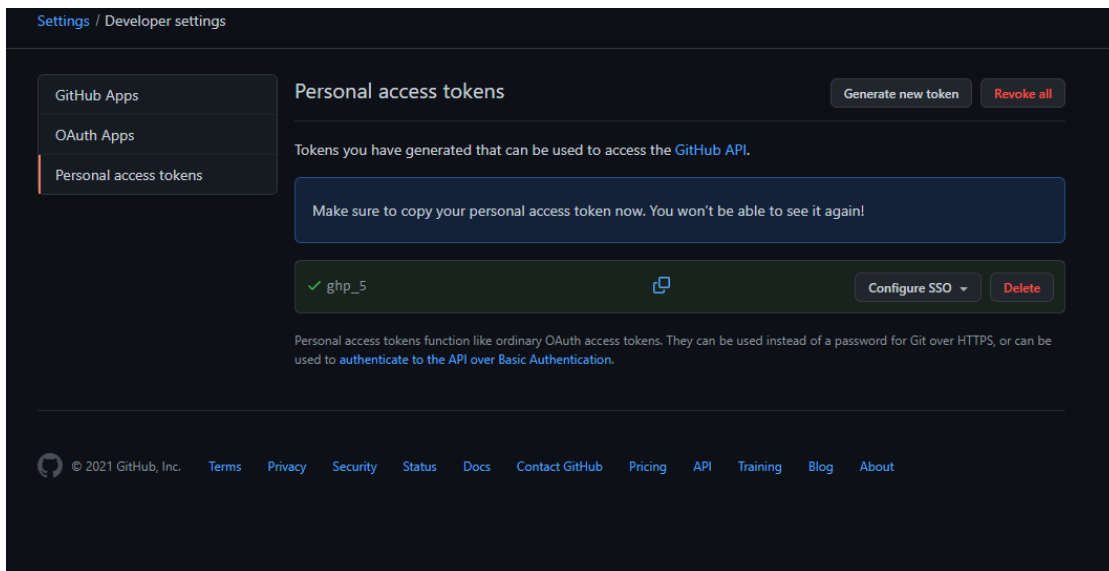
<input checked="" type="checkbox"/> repo	Full control of private repositories
<input checked="" type="checkbox"/> repo:status	Access commit status
<input checked="" type="checkbox"/> repo_deployment	Access deployment status
<input checked="" type="checkbox"/> public_repo	Access public repositories
<input checked="" type="checkbox"/> repo:invite	Access repository invitations
<input checked="" type="checkbox"/> security_events	Read and write security events
<input type="checkbox"/> workflow	Update GitHub Action workflows
<input type="checkbox"/> write:packages	Upload packages to GitHub Package Registry
<input type="checkbox"/> read:packages	Download packages from GitHub Package Registry



Los scopes son simplemente las operaciones que estarán o no permitidas desde este token. Lo más común es que des permisos para acceder a los repositorios remotos alojados en GitHub, pero existen otras operativas que puedes hacer o no tú y que podrías permitir o no con este token que vas a crear.

COPIAR EL PAT EN UN LUGAR SEGURO

Una vez generado el token **aparecerá en la página de GitHub. Puedes copiarlo y ponerlo en un lugar seguro.** Ten en cuenta que es como si fuera una clave de usuario!! Por lo que debes cerciorarte que solamente tú tengas acceso a este token.



¡Por supuesto, **no lo debes pegar en el código de ninguna aplicación,** para no dejarlo visible para otras personas!! o tus accesos a GitHub podrían verse comprometidos.

Otro detalle importante es que este token solamente **aparecerá una vez** en la página de GitHub. Si lo necesitas recuperar porque lo has perdido simplemente **no podrás hacerlo.** Tendrás que generar uno nuevo.

CÓMO USAR EL PERSONAL ACCESS TOKEN DE GITHUB

El mecanismo para usar el PAT es tan sencillo como el que anteriormente hacíamos con la clave del sitio web.

Simplemente, **cuando la operación de GitHub te pida la clave de tu usuario, en lugar de la clave colocarás el token.**

Por ejemplo, en la siguiente imagen vemos el comando de clonado de un repositorio. Cuando nos piden la clave, simplemente colocamos toda la cadena del token que hemos copiado en GitHub. Con esto habremos autenticado con el token correctamente y podremos acceder a los servicios de GitHub.

```
midesweb@MacBook-Pro ~/sites/sandbox$ git clone https://github.com/midesweb/preparar-php-apps-prof.git
Clonando en 'preparar-php-apps-prof'...
Username for 'https://github.com': midesweb
Password for 'https://midesweb@github.com':
```

CACHEO DEL PERSONAL ACCESS TOKEN

El PAT(Personal Access Token) se utilizará solo en operaciones de HTTPS, para cada operación nos pedirá que lo ingresemos de nuevo, para evitar eso podemos realizar un cacheo del PAT, de esa manera **no tendremos que ingresarlo para cada operación.**

El cacheo del Access Token lo puedes realizar en cualquier sistema operativo por medio de Git. En MacOS se hace automáticamente con la aplicación de llaves. Para los usuarios de otros sistemas operativos como Linux y Windows en la documentación de GitHub encontramos algunas otras indicaciones interesantes.

En Windows te indican que tienes que lanzar el siguiente comando:

```
git config --global credential.helper wincred
```

En Linux te informan de dos comandos útiles. Este te permite indicar que las credenciales se cacheen:

```
git config --global credential.helper cache
```

Y este otro comando te permite que las credenciales permanezcan cacheadas por el tiempo que tú estimes conveniente. El tiempo se indica en segundos. Por ejemplo, para cachear las credenciales por 3 horas lanzas el siguiente comando.

```
git config --global credential.helper 'cache --timeout=10800'
```

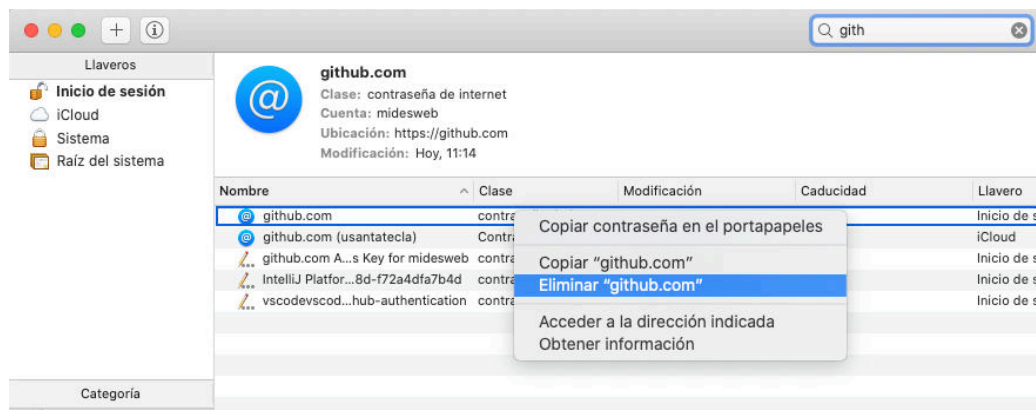
Eso es todo! Espero que con estas indicaciones puedas realizar toda la configuración de GitHub para el acceso desde el terminal en local y poder administrar tus repositorios remotos.

ACTUALIZAR CREDENCIALES EN GITHUB EN MACOS

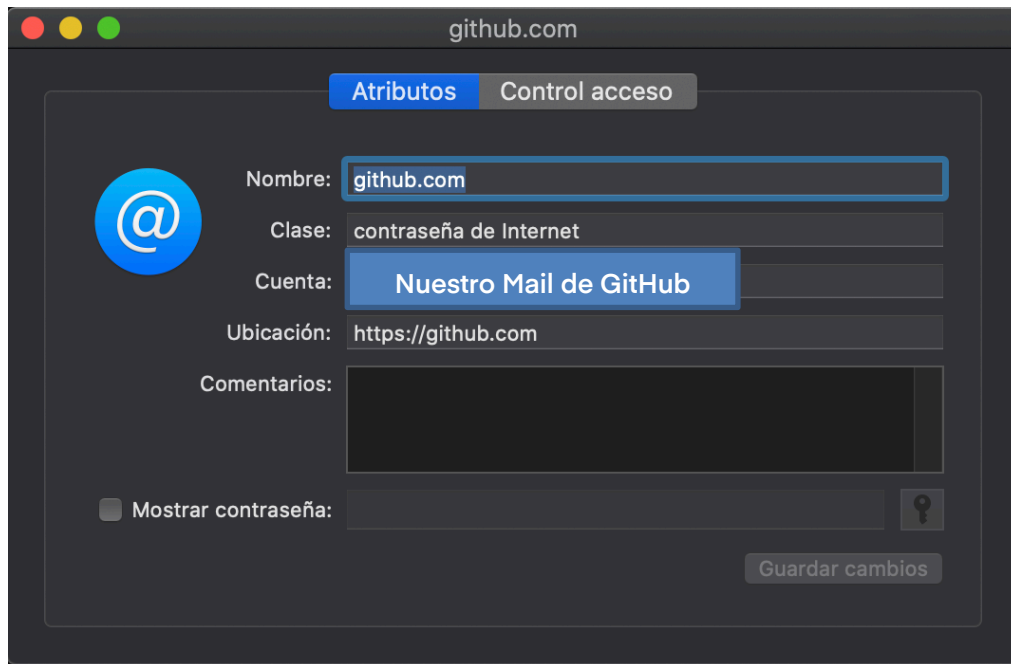
Para los usuarios de Mac, es posible que necesites actualizar credenciales de GitHub. Para ello tienes que abrir una aplicación llamada **Keychain Access**.

Para abrir la aplicación tendremos que hacer **Barra Espaciadora + Command ⌘** y ahí **escribiremos Keychain Access o Acceso a Llaveros**. Cuando aparezca le damos al **Enter**

Una vez dentro de la aplicación, arriba a la derecha, en el buscador de la aplicación buscaremos "GitHub". Veremos algo así:

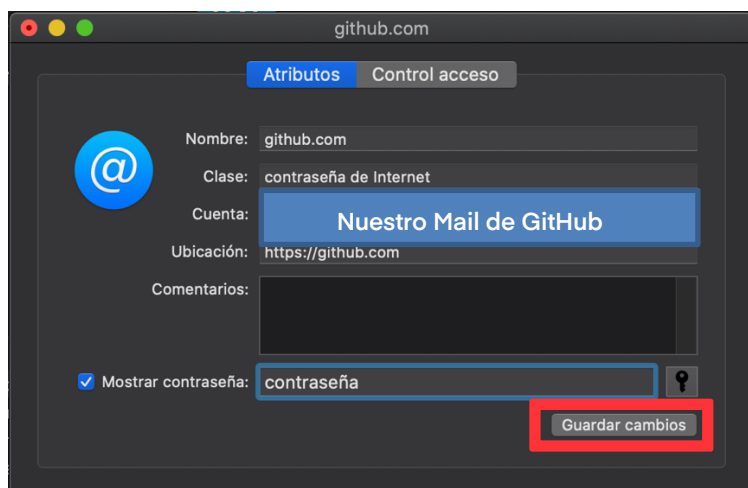


Las credenciales son las que tienen el arroba @ azul. Puede ser que nos salgan dos credenciales de GitHub, tenemos que **abrir la que tenga nuestra cuenta como credencial**. Para revisarlo le haremos doble click a la credencial y ahí nos va a mostrar la credencial y dentro podremos ver la cuenta. Se vería algo así:



Dentro de la credencial con nuestra cuenta, vamos a agregar el Token. Para esto vamos a copiar el token y dentro de la credencial vamos a darle a **Mostrar contraseña o Show Password**, nos va a pedir la contraseña de nuestra computadora, ingresamos la contraseña y le damos a **Permitir**.

Vamos a ver la contraseña de nuestra computadora así:



Borramos la contraseña de nuestra computadora y **pegamos** el Token y damos a **Guardar cambios** y así **ya tenemos el Token configurado**.

Después de realizar este paso, puede ser que al intentar hacer alguna operativa de nuevo con GitHub se pedirá tu usuario y contraseña, e introducirás tu usuario y el token.



Revisemos lo aprendido hasta aquí

- Configurar el Access Token

CONFIGURACIÓN INICIAL

Abra su terminal de Git para comenzar con la ejecución de comandos, por ejemplo, abra el programa `Git bash` en Windows para ingresar a la línea de comandos de este programa.

Una vez que ingrese, use el siguiente comando para establecer el nombre de usuario de git:

```
git config --global user.name "Jhoel Perez"
```

Recuerde sustituir el texto entre comillas por su nombre real. Ahora indique el correo electrónico del usuario para git:

```
git config --global user.email "micorreopersonal@jhoel.com"
```

Sustituyendo el texto entre comillas por su cuenta de correo electrónico. Esta configuración inicial debería ser suficiente para comenzar. Para comprobar otros valores de su configuración actual ejecute:

```
git config --list
```

Se mostrarán los nuevos valores configurados al final, y otros valores de configuración predeterminados:

...

```
color.diff=auto
```

```
color.status=auto
```

...

```
user.name=Juan Perez
```

```
user.email=micorreopersonal@juan.com
```



Revisemos lo aprendido hasta aquí

- Realizar la configuración inicial

¿CÓMO TRABAJAMOS CON GIT?

Para trabajar con Git con Github tenemos dos formas de trabajar:

- 1) Trabajar en local, en un repositorio que me cree en mi máquina (repositorio local) y vincularlo a un repositorio creado en GitHub (repositorio remoto).
- 2) Clonar un repositorio de Github (u otro hosting de repositorios) para traernos a local el repositorio completo y empezar a trabajar con ese proyecto.

Vamos a elegir de momento la opción 1) que nos permitirá comenzar desde cero y con la que podremos apreciar mejor cuáles son las operaciones básicas con Git. En este sentido, cualquier operación que realizas con Git tiene que comenzar mediante el trabajo en local, por lo que tienes que comenzar por crear el repositorio en tu propia máquina. Incluso si tus objetivos son simplemente subir ese repositorio a Github para que otras personas lo puedan acceder a través del hosting remoto de repositorios, tienes que comenzar trabajando en local.

REPOSITORIO LOCAL VS REMOTO

Tenemos dos tipos de repositorios **local y remoto**, ambos son entidades separadas y que a través de los comandos de Git podemos unificar.

Los **repositorios locales residen en las computadoras** de los miembros del equipo. Por el contrario, los **repositorios remotos se alojan en un servidor** al que pueden acceder todos los miembros del equipo, probablemente en Internet o en una red local.

Nosotros trabajaremos en nuestro repositorio local y cuando sintamos que lo que hemos hecho está bien o ya está listo, lo subiremos a nuestro repositorio remoto, de esa manera todos los cambios que hayamos hecho estarán subidos a internet y nuestros compañeros de equipo podrán verlos. Pero esto significa que mientras estemos trabajando, **estaremos trabajando en el repositorio local y después unificaremos con el remoto.**



MANOS A LA OBRA!

INICIO DE UN NUEVO REPOSITORIO: GIT INIT

Primero deberemos crear una carpeta vacía para inicializar nuestro repositorio, una vez que la tenemos creada debemos decirle a la terminal que se pare en esa carpeta, así todos los comandos de Git, afectan a esa carpeta.

En Windows podemos hacer click derecho a la carpeta y darle a **Git Bash Here**, eso nos abrirá la terminal Git Bash en la carpeta para trabajar Git.

En Mac podemos hacer lo mismo, click derecho a la carpeta y darle a la opción **Nuevo Terminal en la carpeta**.

Una vez parados en nuestra carpeta. Para crear un nuevo repositorio, usa el comando **git init**. git init es un comando que se utiliza una sola vez durante la configuración inicial de un repositorio nuevo. Al ejecutar este comando, se creará un nuevo **subdirectorio .git** en tu directorio de trabajo actual. También se creará una nueva rama principal.

`git init`

¿COMO VINCULAMOS NUESTRO REPOSITORIO CON GITHUB?

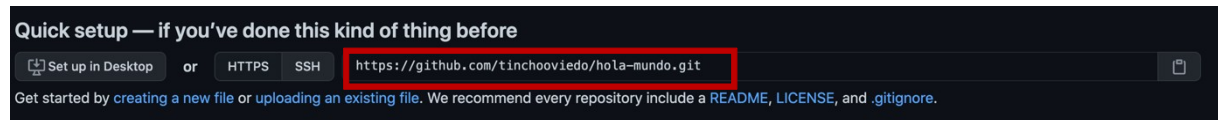
Primero deberemos crear un repositorio, sin el archivo readME.

Una vez que tenemos el archivo agregado y guardado de manera local, tenemos que vincular este repositorio local a un repositorio remoto en GitHub. Para esto vamos a utilizar el comando git remote add.

Este comando va a tomar el alias nuestro repositorio y la url de nuestro repositorio en GitHub, con esto va a vincularlo con nuestro repositorio local.

```
git remote add <name> <url>
```

El alias que vamos a utilizar para Github es origin y para obtener la url de nuestro repositorio, podemos encontrarla al principio de nuestro repositorio:



Buscamos el url de nuestro repositorio de GitHub y lo hacemos en nuestro terminal. Por lo que pondremos:

```
git remote add origin <url>
```

GUARDAR CAMBIOS EN EL REPOSITORIO: GIT STATUS GIT ADD Y GIT COMMIT

Ahora que has iniciado o clonado un repositorio, puedes realizar commits en la versión del archivo. Vamos a ir a nuestra carpeta y vamos a crear un archivo txt con nuestro nombre escrito, una vez que lo tengamos creado vamos a hacer los siguientes comandos.

Git Status

El comando de git status nos da toda la información necesaria sobre la rama actual.

```
git status
```

Podemos encontrar información como:

- Si la rama actual está actualizada
- Si hay algo para confirmar, enviar o recibir (pull).
- Si hay archivos en preparación (staged), sin preparación(unstaged) o que no están recibiendo seguimiento (untracked)
- Si hay archivos creados, modificados o eliminados

```
Cem-MacBook-Pro:my-new-app cem$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   src/App.js

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

        src/components/
```

Nota: veremos el concepto de ramas más adelante.

Hacemos un git status en nuestro repositorio para ver el estado actual.

Git Add

Cuando creamos, modificamos o eliminamos un archivo, estos cambios suceden en local y no se incluirán en el siguiente commit (a menos que cambiemos la configuración).

Necesitamos usar el comando git add para incluir los cambios del o de los archivos en tu siguiente commit.

Añadir un único archivo:

```
git add <archivo>
```

Añadir todo de una vez:

```
git add .
```

Si revisas la captura de pantalla del git status, verás que hay nombres de archivos en rojo - esto significa que los archivos sin preparación. Estos archivos no serán incluidos en tus commits hasta que no los añadas.

Hacemos un **git add .** para agregar nuestro archivo txt. Una vez que hacemos el git add hacemos otro git status, ahora veremos que los archivos que estaban en rojo, están en verde, esto quiere decir que ya los hemos agregado para hacer nuestro commit.

```
Cem-MacBook-Pro:my-new-app cem$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   src/App.js
        new file:   src/components/myFirstComponent.js
```

Git Commit

Este sea quizás el comando más utilizado de Git. Una vez que se llega a cierto punto en el desarrollo, queremos **guardar nuestros cambios** (quizás después de una tarea o asunto específico) o subir un archivo / proyecto.

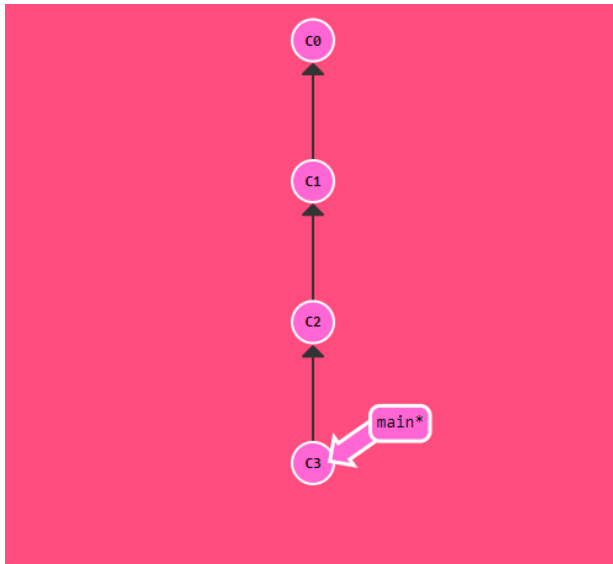
Git commit es como **establecer un punto de control en el proceso de desarrollo al cual puedes volver más tarde si es necesario.** Es como un punto de guardado en un videojuego.

También necesitamos **escribir un mensaje corto para explicar qué hemos desarrollado o modificado en el código fuente.**

```
git commit -m "mensaje de confirmación"
```

Hacemos un commit para guardar nuestro txt y le ponemos un mensaje que explique que hemos hecho.

Todos estos “puntos de guardado” se van a guardar en la **rama** donde estemos parados, en esta guía vamos a trabajar sobre la **rama Main**, más adelante explicaremos que es una rama. Pero ahora pienselo como el **lugar donde se guardarían todos esos “puntos de guardado”**.



Acá podemos ver que, sobre la misma línea, que sería la **rama Main**, tenemos el commit 0, el commit 1, el commit 2 y el commit 3, que serían todos nuestros cambios, sobre el proyecto. Nosotros podemos acceder a todos esos commit y volver hacia atrás a esos “puntos de guardado”, para tener una versión “antigua” del proyecto.

ENVIAR CAMBIOS AL REPOSITORIO: GIT PUSH

Recordemos que antes de este paso debemos haber **confirmado nuestros cambios con git commit** y haber **vinculado el repositorio** de Git local con el repositorio remoto de GitHub, usando **el git remote add origin**. Con estos dos pasos hechos el siguiente paso que quieres dar es enviar tus cambios o archivos al servidor remoto. **git push** envía tus commits al repositorio remoto.

```
git push <nombre-remoto> <nombre-de-tu-rama>
```

Nombre remoto

De nombre remoto usualmente vamos a poner la palabra **origin**, pero **¿que significa la palabra origin?**

En nuestra vida como desarrolladores cuando utilizamos Git generalmente vamos a tener varios repositorios remotos contra los que trabajamos. Bien, pues **origin** es simplemente el **nombre predeterminado o alias** que recibe el **repositorio remoto principal** contra el que trabajamos. Cuando clonamos o creamos un repositorio por primera vez desde GitHub o cualquier otro sistema remoto, el nombre que se le da a ese repositorio es precisamente *origin*.

Básicamente origin es el alias que se le da a la url del repositorio remoto, nosotros en vez de poner push, más, el url del repositorio remoto al que queremos pushear, usamos el alias **origin** para especificarle que tiene que pushear a **ese repositorio remoto concreto**.

De todos modos, ese nombre se puede cambiar, y se pueden crear más remotos contra los que hacer *push*. Pero en un porcentaje muy alto de los casos, como ese nombre por omisión no se cambia, puedes asumir que vas a enviar la información a *origin*.

El push nos quedaría de la siguiente manera:

```
git push origin <nombre-de-tu-rama>
```

Nombre de tu rama

Recordemos que usamos el alias origin para el repositorio remoto.

Cuando creamos un repositorio en GitHub, nos crea una rama por defecto llamada main, que será a la rama que tendremos que pushear.

Por lo que ahora haremos el siguiente comando en nuestra terminal:

```
git push origin main
```

Una vez que hemos hecho esto, si refrescamos nuestro repositorio vamos a ver nuestro archivo txt en nuestro repositorio de GitHub.

Este proceso se va a repetir, quitando la vinculación y la inicialización del repositorio, cada vez que nosotros hagamos un cambio dentro de nuestro repositorio. Esto puede ser modificar un archivo ya existente o agregar más archivos a la carpeta local.

¿QUÉ ES LA RAMA MAIN?

Explicamos previamente que cuando queremos pushear nuestros cambios debemos hacerlo a una rama main, pero ¿que es una rama y que es la rama main?

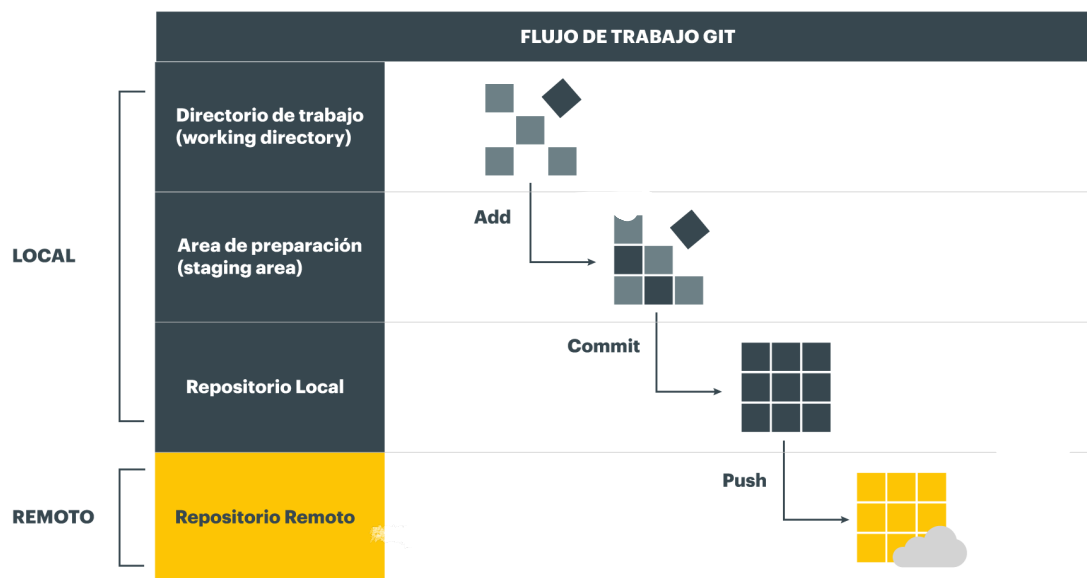
Rama en Git es similar a la rama de un árbol. De manera análoga, una rama de árbol está unida a la parte central del árbol llamada tronco. Si bien las ramas pueden generarse y caerse, el tronco permanece compacto y es la única parte por la cual podemos decir que el árbol está vivo y en pie. De manera similar, una rama en Git es una forma de seguir desarrollando y codificando una nueva función o modificación del software y aún así no afectar la parte principal del proyecto. **La rama principal o predeterminada en Git es la rama maestra o main (similar al tronco de un árbol)**. Tan pronto como se crea el repositorio, también lo hace la rama principal (o la rama predeterminada).

Básicamente la rama va a ser el lugar donde se guarden nuestros commits (nuestros cambios) y podamos ver todos los cambios que vayamos haciendo a lo largo del tiempo reflejados en esa rama.

Como nosotros **por ahora** no vamos a trabajar con ramas, ósea, no vamos a crear más ramas o borrar ramas, etc. Vamos a simplemente trabajar con la rama main, por eso cuando queramos subir nuestros cambios al repositorio remoto, vamos a utilizar la rama main, ya que es la única que tenemos disponible por ahora.

FLUJO DE TRABAJO GIT

Ahora que conocemos todos los comandos que vamos a ver en esta guía de Git con GitHub. Vamos a ver un diagrama, de como se verían todos los comandos previamente mencionados, git add, git commit y git push, en un flujo de trabajo normal.



En este diagrama se muestra cual sería el flujo de trabajo de Git con GitHub. Para poder entender este diagrama, vamos a primero explicar los tres estados de nuestros archivos que existen en Git.

Tres estados

Git tiene tres estados principales en los que se pueden encontrar tus archivos: confirmado (committed), modificado (modified), y preparado (staged).

- **Confirmado:** significa que los datos están almacenados de manera segura en tu base de datos local.
- **Modificado:** significa que has modificado el archivo, pero todavía no lo has confirmado a tu base de datos.
- **Preparado:** significa que has marcado un archivo modificado en su versión actual para que vaya en tu próxima confirmación.

Esto nos lleva a las tres secciones principales de un proyecto de Git: El **directorio de Git** (*Git directory*), el **directorio de trabajo** (*working directory*), y el **área de preparación** (*staging area*).

El **directorio de Git** es donde se almacenan los metadatos y la base de datos de objetos para tu proyecto. Es la parte más importante de Git, es el **subdirectorio .git** que se crea cuando ejecutamos el comando `git init`. Y también, es lo que se copia cuando clonas un repositorio desde otra computadora.

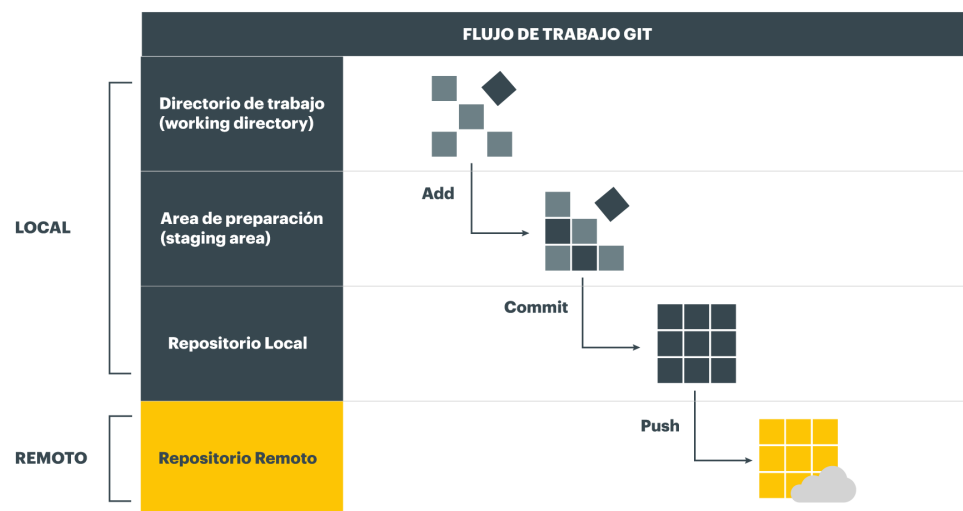
El **directorio de trabajo** es una copia de una versión del proyecto. Estos archivos se sacan de la base de datos comprimida en el directorio de Git, y se colocan en disco para que los puedas usar o modificar. O esto sería el proyecto que tenemos de manera local y se convierte en directorio de trabajo cuando le hacemos un `git init`, ya que, como dijimos el proyecto pasa a tener un directorio de git, cuando ejecutamos dicho comando.

El **área de preparación es un archivo**, generalmente contenido en tu directorio de Git, que almacena información acerca de lo que va a ir en tu próxima confirmación (commit). A veces se le denomina índice ("index"), pero se está convirtiendo en estándar el referirse a ella como el área de preparación. Esto se generaría con el comando **git add**, porque "agregaríamos" que información queremos enviar en el commit.

El flujo de trabajo básico en Git es algo así:

1. Modificas una serie de **archivos en tu directorio de trabajo**.
2. Preparas los archivos, añadiéndolos a tu área de preparación. (Git Add)
3. Confirmas los cambios, lo que toma los archivos tal y como están en el área de preparación y almacena esa copia instantánea de manera permanente en tu directorio de Git. (Git Commit)

Ahora volvamos a ver el diagrama:



Podemos ver como el cuadrado, empieza en el directorio de trabajo, pasa al área de preparación con el Git Add, después se confirman los cambios con el Git Commit y pasa al repositorio local y por último el push envía ese cuadrado al repositorio remoto, que sería el repositorio que tenemos en GitHub



Revisemos lo aprendido hasta aquí

- Vincular el repositorio local con el remoto
- Utilizar los comandos **status, add y commit**.
- Actualizar el repositorio remoto utilizando el comando **git push**

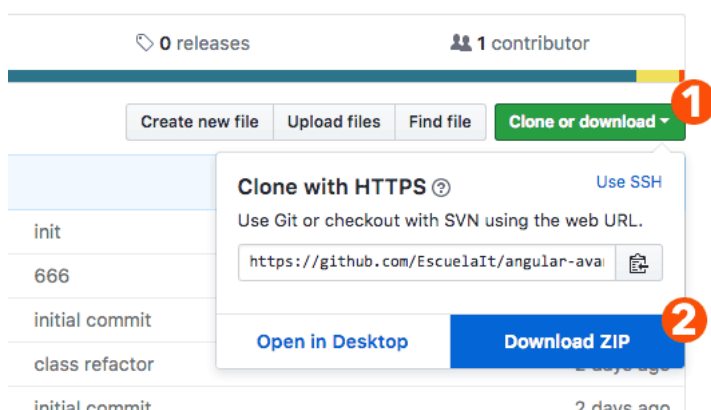
CLONAR UN REPOSITORIO

Ahora vamos a hablar de la operativa de clonado de un repositorio, el proceso que tienes que hacer cuando quieres traerte el código de un proyecto que está publicado en GitHub y lo quieres restaurar en tu ordenador, para poder usarlo en local, modificarlo, etc.

Este paso es bastante básico y muy sencillo de hacer, pero es esencial porque lo necesitarás realizar muchas veces en tu trabajo como desarrollador. Además, intentaremos complementarlo con alguna información útil, de modo que puedas aprender cosas útiles y un poquito más avanzadas.

DESCARGAR VS CLONAR

Al inicio de uso de un sitio como GitHub, si no tenemos ni idea de usar Git, también podemos obtener el código de un repositorio descargando un simple Zip. Esta opción la consigues mediante el botón de la siguiente imagen.



Sin embargo, descargar un repositorio así, aunque muy sencillo no te permite algunas de las utilidades interesantes de clonarlo, como:

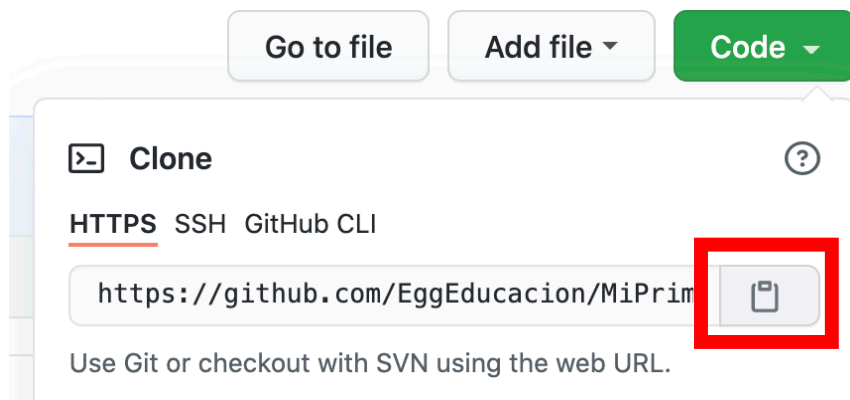
- No crea un repositorio Git en local con los cambios que el repositorio remoto ha tenido a lo largo del tiempo. Es decir, te descargas el código, pero nada más.
- No podrás luego enviar cambios al repositorio remoto, una vez los hayas realizado en local.

En resumen, no podrás usar en general las ventajas de Git en el código descargado. Así que es mejor clonar, ya que aprender a realizar este paso es también muy sencillo.

CLONAR EL REPOSITORIO GIT

Entonces veamos cómo debes clonar el repositorio, de modo que sí puedas beneficiarte de Git con el código descargado. El proceso es el siguiente.

Primero copiarás la URL del repositorio remoto que deseas clonar (ver el icono "Copy to clipboard" en la siguiente imagen).



Luego abrirás una ventana de terminal, para situarte sobre la carpeta de tu proyecto que quieras clonar. Yo te recomendaría crear ya directamente una carpeta con el nombre del proyecto que estás clonando, o cualquier otro nombre que te parezca mejor para este repositorio. Te sitúas dentro de esa carpeta y desde ella lanzamos el comando para hacer el clon, que sería algo como esto:

```
git clone https://github.com/EggEducacion/MiPrimerRepositorio.git
```

El último punto, después de la url copiada desde git, le indica que el clon lo vas a colocar en la carpeta donde estás situado, en tu ventana de terminal. La salida de ese comando sería más o menos como tienes en la siguiente imagen:

```
→ Git git clone https://github.com/EggEducacion/MiPrimerRepositorio.git .
Cloning into '.'...
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
→ Git git:(master)
```

De esta manera nosotros ya tenemos el repositorio remoto para trabajar local y podremos hacer los cambios que queramos y subir los cambios con los comandos que explicamos previamente.

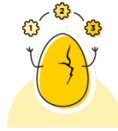


Revisemos lo aprendido hasta aquí

- Clonar un repositorio
- Diferenciar la descarga de la clonación



Anota los comandos básicos para tenerlos a mano. Por ejemplo: pegarlos con un post it al lado de la pantalla, puedes agregar el orden en que debes ejecutarlos.



Importante

Para que sigas practicando esta herramienta tan importante, los ejemplos de las siguientes guías para descargar, van a estar alojados en el siguiente portal de tu curso con repositorios. La idea es que practiquen descargar esos ejemplos usando Git y así no perder la practica. ¡¡Recomendamos que se guarden este link para siempre tenerlo a mano!!

GitHub Egg: <https://github.com/EggCooperation>

EJERCICIOS DE APRENDIZAJE

Ahora es momento de poner en practica todo lo visto en la guía.



VIDEOS: Te sugerimos ver los videos relacionados con este tema, antes de empezar los ejercicios, los podrás encontrar en tu aula virtual o en nuestro canal de YouTube.

PRIMER EJERCICIO

- 1) Tendremos que crear desde GitHub un repositorio SIN ARCHIVO README y copiar la URL directa hacia el repositorio.
- 2) Crear una carpeta que contenga un archivo .txt con un mensaje inicial
- 3) Recordar que deberemos inicializar el repositorio haciendo click derecho sobre la carpeta que contiene el archivo y abrir Git . O también podremos hacer click derecho en un lugar en blanco al costado del archivo .txt. Inicializar el repositorio con:
- 4) git init ✓
- 5) Vincular y conectar nuestro repositorio local con el remoto usando:
- 6) git remote add origin URL ✓
- 7) Luego controlo los cambios que están bajo observación en la rama principal con:
- ✓ 8) git status ✓
- 9) Luego añado los cambios de todos los archivos con:
- ✓ 10) git add . ✓
- 11) Luego comiteo los cambios con:
- ✓ 12) git commit -m "Mensaje" ✓
- 13) Y finalmente hago un git push origin para mandar mis cambios al repositorio remoto ✓
- ✓ 14) Podría cambiar el mensaje del .txt y repetir los pasos desde el 2 al 10 y debería ver en la página de git mis cambios

Todo lo que hacemos desde el paso 2 al 9 debemos repetirlo en todos los ejercicios que quisiéramos guardar en nuestro Git personal.



Desde Egg sugerimos hacerlo con todos los siguientes ejercicios y cada vez que terminemos nuestra jornada de estudio. Lo recomendamos por dos motivos: uno, para practicar Git desde ahora hasta la última guía y dos, para que nuestros ejercicios no se pierdan ante algún eventual problema en nuestra computadora.

✓ SEGUNDO EJERCICIO

- 1) Tendremos que pedirle un repositorio publico a algún estudiante de nuestro curso.
- 2) Copiaremos la URL principal del repositorio del estudiante
- 3) Tendremos que abrir Git en alguna carpeta y correr el comando git clone URL_ESTUDIANTE



De esta manera podremos no solo aprender del código de otros programadores, sino que también podremos recuperar información propia de nuestros proyectos desde git.

```
EUGE@DESKTOP-IR3PA1E MINGW64 ~/Desktop/programacion/Backend 1 Java/GIT - repositorio local
$ git init
Initialized empty Git repository in C:/Users/EUGE/Desktop/programacion/Backend 1 Java/GIT - repositorio local/.git/
EUGE@DESKTOP-IR3PA1E MINGW64 ~/Desktop/programacion/Backend 1 Java/GIT - repositorio local (master)
$ git remote add origin https://github.com/EugeniaMarinZalda/Encuentro1Egg.git
```

```
EUGE@DESKTOP-IR3PA1E MINGW64 ~/Desktop/programacion/Backend 1 Java/GIT - repositorio local (master)
$
EUGE@DESKTOP-IR3PA1E MINGW64 ~/Desktop/programacion/Backend 1 Java/GIT - repositorio local (master)
$ git status
On branch master
```

```
No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
  committejer1.txt

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

```
EUGE@DESKTOP-IR3PA1E MINGW64 ~/Desktop/programacion/Backend 1 Java/GIT - repositorio local (master)
$ git add committejer1.txt
```

```
EUGE@DESKTOP-IR3PA1E MINGW64 ~/Desktop/programacion/Backend 1 Java/GIT - repositorio local (master)
$ git status
On branch master
```

```
No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   committejer1.txt
```

```
EUGE@DESKTOP-IR3PA1E MINGW64 ~/Desktop/programacion/Backend 1 Java/GIT - repositorio local (master)
$ git status
On branch master
```

```
No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   committejer1.txt
```

```
EUGE@DESKTOP-IR3PA1E MINGW64 ~/Desktop/programacion/Backend 1 Java/GIT - repositorio local (master)
$ git commit -m primercommit
[master (root-commit) 042a1dc] primercommit
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 committejer1.txt
```

```
EUGE@DESKTOP-IR3PA1E MINGW64 ~/Desktop/programacion/Backend 1 Java/GIT - repositorio local (master)
$ git status
On branch master
nothing to commit, working tree clean
```

```
EUGE@DESKTOP-IR3PA1E MINGW64 ~/Desktop/programacion/Backend 1 Java/GIT - repositorio local (master)
$ git push origin main
error: src refspec main does not match any
error: failed to push some refs to 'https://github.com/EugeniaMarinZalda/Encuentro1Egg.git'
```

```
EUGE@DESKTOP-IR3PA1E MINGW64 ~/Desktop/programacion/Backend 1 Java/GIT - repositorio local (master)
$ git help
usage: git [-v] [-version] [-h] [-help] [-C <path>] [-c <name>=<value>]
  [-exec-path=<path>] [--html-path] [--man-path] [--info-path]
  [-p | --paginate] [-P | --no-pager] [--no-replace-objects] [--bare]
  [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
  [--super-prefix=<path>] [--config-env=<name>=<envvar>]
  [<command>] [<args>]
```

These are common Git commands used in various situations:

```
start a working area (see also: git help tutorial)
clone      Clone a repository into a new directory
init       Create an empty Git repository or reinitialize an existing one
```

```
work on the current change (see also: git help everyday)
add        Add file contents to the index
mv         Move or rename a file, a directory, or a symlink
restore    Restore working tree files
rm         Remove files from the working tree and from the index
```

```
examine the history and state (see also: git help revisions)
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.
```

```
EUGE@DESKTOP-IR3PA1E MINGW64 ~/Desktop/programacion/Backend 1 Java/Introduccion - Guia 1/clonar 2 (master)
$ git init
Reinitialized existing Git repository in C:/Users/EUGE/Desktop/programacion/Backend 1 Java/Introduccion - Guia 1/clonar 2/.git/
```

```
EUGE@DESKTOP-IR3PA1E MINGW64 ~/Desktop/programacion/Backend 1 Java/Introduccion - Guia 1/clonar 2 (master)
$ git remote add origin https://github.com/EggCooperation/CalendarRedQADCTeamVersion.git
error: unknown subcommand: 'origin'
usage: git remote [-v] [--verbose]
  or: git remote add [-t <branch>] [-m <master>] [-f] [--tags | --no-tags] [--mirror=<fetch|push>] <name> <url>
  or: git remote rename [-m <master>] <old> <new>
  or: git remote remove <name>
  or: git remote set-head <name> [-a] --auto [-d] --delete | <branch>
  or: git remote [-v] [--verbose] show [-n] <name>
  or: git remote remove <name>
  or: git remote prune [-n] [--dry-run] <name>
  or: git remote [-v] [--verbose] update [-p] [--prune] [--group] | <remote>...
  or: git remote set-branches [--add] <name> <branch>...
  or: git remote get-url [--push] [--all] <name>
  or: git remote set-url [--push] <name> <newurl> [<oldurl>]
  or: git remote set-url --add <name> <newurl>
  or: git remote set-url --delete <name> <url>
```

```
-v, --verbose      be verbose; must be placed before a subcommand
```

```
EUGE@DESKTOP-IR3PA1E MINGW64 ~/Desktop/programacion/Backend 1 Java/Introduccion - Guia 1/clonar 2 (master)
$ git clone https://github.com/EggCooperation/CalendarRedQADCTeamVersion.git
Cloning into 'CalendarRedQADCTeamVersion'...
remote: Enumerating objects: 40, done.
remote: Counting objects: 100% (40/40), done.
remote: Compressing objects: 100% (40/40), done.
remote: Total 40 (delta 9), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (40/40), 443.03 KiB | 869.00 KiB/s, done.
Resolving deltas: 100% (9/9), done.
```

```
EUGE@DESKTOP-IR3PA1E MINGW64 ~/Desktop/programacion/Backend 1 Java/Introduccion - Guia 1/clonar 2 (master)
$ git remote -v
```