Para iniciar un repositorio, o sea, activar el sistema de control de versiones de Git en tu proyecto, solo debes ejecutar el comando git init.

Este comando se encargará de dos cosas: primero, crear una carpeta. git donde se guardará toda la base de datos con cambios atómicos de nuestro proyecto; y segundo, crear un área en la memoria RAM, que conocemos como Staging, que guardará temporalmente nuestros archivos (cuando ejecutemos un comando especial para eso) y nos permitirá, más adelante, guardar estos cambios en el repositorio (también con un comando especial).

1. **Ciclo de vida o estados de los archivos en Git**:

Cuando trabajamos con Git, nuestros archivos pueden vivir y moverse entre 4 diferentes estados (cuando trabajamos con repositorios remotos pueden ser más estados, pero lo estudiaremos más adelante):

* **Archivos Tracked**: Son los archivos que viven dentro de Git, no tienen cambios pendientes y sus últimas actualizaciones han sido guardadas en el repositorio gracias a los comandos git add y git commit.
* **Archivos Staged**: Son archivos en Staging. Viven dentro de Git y hay registro de ellos porque han sido afectados por el comando git add, aunque no sus últimos cambios. Git ya sabe de la existencia de estos últimos cambios, pero todavía no han sido guardados definitivamente en el repositorio porque falta ejecutar el comando git commit.
* **Archivos Unstaged**: Entiéndelos como archivos *“Tracked, pero Unstaged”*. Son archivos que viven dentro de Git, pero no han sido afectados por el comando git add ni mucho menos por git commit. Git tiene un registro de estos archivos, pero está desactualizado, sus últimas versiones solo están guardadas en el disco duro.
* **Archivos Untracked**: Son archivos que NO viven dentro de Git, solo en el disco duro. Nunca han sido afectados por git add, así que Git no tiene registros de su existencia.

Recuerda que hay un caso muy raro donde los archivos tienen dos estados al mismo tiempo: Staged y Untracked. Esto pasa guardas los cambios de un archivo en el área de Staging (con el comando git add) pero, antes de hacer commit para guardar los cambios en el repositorio, haces nuevos cambios que todavía no han sido guardados en el área de Staging (en realidad, todo sigue funcionando igual, pero es un poco divertido).

**Comandos para mover archivos entre los estados de Git**:

* **git status**: Nos permite ver el estado de todos nuestros archivos y carpetas.
* **git add**: Nos ayuda a mover archivos del Untracked o Unstaged al estado Staged. Podemos usar git nombre-del-archivo-o-carpeta para añadir archivos y carpetas individuales o git add -A para mover todos los archivos de nuestro proyecto (tanto Untrackeds como unstageds).
* **git reset HEAD**: Nos ayuda a sacar archivos del estado Staged para devolverlos a su estado anterior. Si los archivos venían de Unstaged, vuelven allí. Y lo mismo se venían de Untracked.
* **git commit**: Nos ayuda a mover archivos de Unstaged a Staged. Esta es una ocasión especial, los archivos han sido guardado o actualizados en el repositorio. Git nos pedirá que dejemos un mensaje para recordar los cambios que hicimos y podemos usar el argumento -m para escribirlo (git commit -m "mensaje").
* **git rm**: Este comando necesita alguno de los siguientes argumentos para poder ejecutarse correctamente:  
  - git rm --cached: Mueve los archivos que le indiquemos al estado Untracked.  
  - git rm --force: Elimina los archivos de Git y del disco duro. Git guarda el registro de la existencia de los archivos, por lo que podremos recuperarlos si es necesario (pero debemos usar comandos más avanzados).
* ¿Qué es un Branch (rama) y cómo funciona un Merge en Git?
* Git es una base de datos muy precisa con todos los cambios y crecimiento que ha tenido nuestro proyecto. Los commits son la única forma de tener un registro de los cambios. Pero las ramas amplifican mucho más el potencial de Git.
* **Todos los commits se aplican sobre una rama**. Por defecto, siempre empezamos en la rama master (pero puedes cambiarle el nombre si no te gusta) y creamos nuevas ramas, a partir de esta, para crear flujos de trabajo independientes.
* Crear una nueva rama se trata de copiar un commit (de cualquier rama), pasarlo a otro lado (a otra rama) y continuar el trabajo de una parte específica de nuestro proyecto sin afectar el flujo de trabajo principal (que continúa en la rama master o la rama principal).
* Los equipos de desarrollo tienen un estándar: Todo lo que esté en la rama master va a producción, las nuevas features, características y experimentos van en una rama “development” (para unirse a master cuando estén definitivamente listas) y los issues o errores se solucionan en una rama “hotfix” para unirse a master tan pronto como sea posible.
* Crear una nueva rama lo conocemos como **Checkout**. Unir dos ramas lo conocemos como **Merge**.
* Podemos crear todas las ramas y commits que queramos. De hecho, podemos aprovechar el registro de cambios de Git para crear ramas, traer versiones viejas del código, arreglarlas y combinarlas de nuevo para mejorar el proyecto.
* Solo ten en cuenta que combinar estas ramas (sí, hacer “merge”) puede generar conflictos. Algunos archivos pueden ser diferentes en ambas ramas. Git es muy inteligente y puede intentar unir estos cambios automáticamente, pero no siempre funciona. En algunos casos, somos nosotros los que debemos resolver estos conflictos “a mano”.

2) Crea un repositorio de Git y haz tu primer commit

*Si quieres ver los archivos ocultos de una carpeta puedes habilitar la opción de Vista > Mostrar u ocultar > Elementos ocultos (en Windows) o ejecutar el comando ls -a.*

Le indicaremos a Git que queremos crear un nuevo repositorio para utilizar su sistema de control de versiones. Solo debemos posicionarnos en la carpeta raíz de nuestro proyecto y ejecutar el comando **git init**.

Recuerda que al ejecutar este comando (y de aquí en adelante) vamos a tener una nueva carpeta oculta llamada. git con toda la base de datos con cambios atómicos en nuestro proyecto.

Recuerda que Git está optimizado para trabajar en equipo, por lo tanto, debemos darle un poco de información sobre nosotros. No debemos hacerlo todas las veces que ejecutamos un comando, basta con ejecutar solo una sola vez los siguientes comandos con tu información:

git config --global user. email "tu@email.com"

git config --global user.name "Tu Nombre"

Existen muchas otras configuraciones de Git que puedes encontrar ejecutando el comando git config --list (o solo git config para ver una explicación más detallada).

3) Analizar cambios en los archivos de tu proyecto con Git

El comando **git show** nos muestra los cambios que han existido sobre un archivo y es muy útil para detectar cuándo se produjeron ciertos cambios, qué se rompió y cómo lo podemos solucionar. Pero podemos ser más detallados.

Si queremos ver la diferencia entre una versión y otra, no necesariamente todos los cambios desde la creación del archivo, podemos usar el comando **git diff commitA commitB**.

Recuerda que puedes obtener el ID de tus commits con el comando **git log**.

4) Volver en el tiempo en nuestro repositorio utilizando branches y checkout

Git log –stat muestra el sha quien relizo los cambio y cuanto espacio de memoria

El comando **git checkout** + ID del commit nos permite viajar en el tiempo. Podemos volver a cualquier versión anterior de un archivo específico o incluso del proyecto entero. Esta también es la forma de crear ramas y movernos entre ellas.

También hay una forma de hacerlo un poco más “ruda”: usando el comando git reset. En este caso, no solo “volvemos en el tiempo”, sino que borramos los cambios que hicimos después de este commit.

Hay dos formas de usar git reset: con el argumento --hard, borrando toda la información que tengamos en el área de staging (y perdiendo todo para siempre). O, un poco más seguro, con el argumento --soft, que mantiene allí los archivos del área de staging para que podamos aplicar nuestros últimos cambios pero desde un commit anterior.

5) Flujo de trabajo básico con un repositorio remoto

**No veas esta clase** a menos que hayas practicado todos los comandos de las clases anteriores.

Por ahora, nuestro proyecto vive únicamente en nuestra computadora. Esto significa que no hay forma de que otros miembros del equipo trabajen en él.

Para solucionar esto están los **servidores remotos**: un nuevo estado que deben seguir nuestros archivos para conectarse y trabajar con equipos de cualquier parte del mundo.

Estos servidores remotos pueden estar alojados en GitHub, GitLab, BitBucket, entre otros. Lo que van a hacer es guardar el mismo repositorio que tienes en tu computadora y darnos una URL con la que todos podremos acceder a los archivos del proyecto para descargarlos, hacer cambios y volverlos a enviar al servidor remoto para que otras personas vean los cambios, comparen sus versiones y creen nuevas propuestas para el proyecto.

Esto significa que debes aprender algunos nuevos comandos:

* **git clone url\_del\_servidor\_remoto**: Nos permite descargar los archivos de la última versión de la rama principal y todo el historial de cambios en la carpeta .git.
* **git push**: Luego de hacer de hacer git add y git commit debemos ejecutar este comando para mandar los cambios al servidor remoto.
* **git fetch**: Lo usamos para traer actualizaciones del servidor remoto y guardarlas en nuestro repositorio local (en caso de que hayan, por supuesto).
* **git merge**: También usamos el comando git fetch con servidores remotos. Lo necesitamos para combinar los últimos cambios del servidor remoto y nuestro directorio de trabajo.
* **git pull**: Básicamente, git fetch y git merge al mismo tiempo.
* git log --oneline - Te muestra el id commit y el título del commit.
* git log --decorate- Te muestra donde se encuentra el head point en el log.
* git log --stat - Explica el número de líneas que se cambiaron brevemente.
* git log -p- Explica el número de líneas que se cambiaron y te muestra que se cambió en el contenido.
* git shortlog - Indica que commits ha realizado un usuario, mostrando el usuario y el titulo de sus commits.
* git log --graph --oneline --decorate y
* git log --pretty=format:"%cn hizo un commit %h el dia %cd" - Muestra mensajes personalizados de los commits.
* git log -3 - Limitamos el número de commits.
* git log --after=“2018-1-2” ,
* git log --after=“today” y
* git log --after=“2018-1-2” --before=“today” - Commits para localizar por fechas.
* git log --author=“Name Author” - Commits realizados por autor que cumplan exactamente con el nombre.
* git log --grep=“INVIE” - Busca los commits que cumplan tal cual está escrito entre las comillas.
* git log --grep=“INVIE” –i- Busca los commits que cumplan sin importar mayúsculas o minúsculas.
* git log – index.html- Busca los commits en un archivo en específico.
* git log -S “Por contenido”- Buscar los commits con el contenido dentro del archivo.
* git log > log.txt - guardar los logs en un archivo txt
* git log --all --graph --decorate --oneline muetra

comandos  
<!-- basico -->  
git init -> inicia un repositorio  
git add $nombreArchivo -> añade un archivo al repositorio local  
git add . -> añade todos los cambios al repositorio local  
git commit -m “comentario” -> ejecuta un commit  
git commit -am “comentario” -> ejecuta un commit y ejecuta el add al mismo tiempo (pero no agrega archivos nuevos, solo cambios a existentes)

<!-- rollback -->  
git reset $idDeEstado --hard -> hace un rollback de un estado en repositorio local, elimina todos los commit hechos  
git reset $idDeEstado --soft -> hace un rollback de un estado en repositorio local, No elimina los commits, estos quedan guardados en repositorio local  
git checkout $idDelEstado --> hace un rollback a un estado

<!-- branch -->  
git branch -> muestra todas las ramas  
git branch $nombreDeLaNuevaRama -> crea un branch nuevos  
git checkout $nombreDeBranch -> Cambia de branch  
git merch $nombreDeBranch -> fuciona dos ramas

Git show-branch muestra las ramas y su historia.

Git show-branch –all muestra igual que git show branch pero mas detallado

<!-- comandos para traer data del repositorio remoto -->  
git fetch -> descarga un estado del repositorio remoto al repositorio local  
git merge -> ejecuta en los archivos de un repositorio local  
git pull -> descarga un estado del repositorio remoto al repositorio local y a los archivos (osea combina un fetch y un merge)

<!-- comandos para ver estados -->  
git status -> muestra el estado del commit antes de enviarse  
git log -> muestra un registro de los commits ejecutados

6) Introducción a las ramas o branches de Git

Las ramas son la forma de hacer cambios en nuestro proyecto sin afectar el flujo de trabajo de la rama principal. Esto porque queremos trabajar una parte muy específica de la aplicación o simplemente experimentar.

La cabecera o HEAD representan la rama y el commit de esa rama donde estamos trabajando. Por defecto, esta cabecera aparecerá en el último commit de nuestra rama principal. Pero podemos cambiarlo al crear una rama (git branch rama, git checkout -b rama) o movernos en el tiempo a cualquier otro commit de cualquier otra rama con los comandos (git reset id-commit, git checkout rama-o-id-commit).

7) GitHub es una plataforma que nos permite guardar repositorios de Git que podemos usar como servidores remotos y ejecutar algunos comandos de forma visual e interactiva (sin necesidad de la consola de comandos).

Luego de crear nuestra cuenta, podemos crear o importar repositorios, crear organizaciones y proyectos de trabajo, descubrir repositorios de otras personas, contribuir a esos proyectos, dar estrellas y muchas otras cosas.

El README.md es el archivo que veremos por defecto al entrar a un repositorio. Es una muy buena práctica configurarlo para describir el proyecto, los requerimientos y las instrucciones que debemos seguir para contribuir correctamente.

Para clonar un repositorio desde GitHub (o cualquier otro servidor remoto) debemos copiar la URL (por ahora, usando HTTPS) y ejecutar el comando git clone + la URL que acabamos de copiar. Esto descargara la versión de nuestro proyecto que se encuentra en GitHub.

Sin embargo, esto solo funciona para las personas que quieren empezar a contribuir en el proyecto. Si queremos conectar el repositorio de GitHub con nuestro repositorio local, el que creamos con git init, debemos ejecutar las siguientes instrucciones:

# Primero: Guardar la URL del repositorio de GitHub

# con el nombre de origin

git remote add origin URL

# Segundo: Verificar que la URL se haya guardado

# correctamente:

git remote

git remote -v

# Tercero: Traer la versión del repositorio remoto y

# hacer merge para crear un commit con los archivos

# de ambas partes. Podemos usar git fetch y git merge

# o solo el git pull con el flag --allow-unrelated-histories:

git pull origin master --allow-unrelated-histories

# Por último, ahora sí podemos hacer git push para guardar

# los cambios de nuestro repositorio local en GitHub:

git push origin master

8) Cómo funcionan las llaves públicas y privadas

Las llaves públicas y privadas nos ayudan a cifrar y descifrar nuestros archivos de forma que los podamos compartir archivos sin correr el riesgo de que sean interceptados por personas con malas intenciones.

La forma de hacerlo es la siguiente:

1. Ambas personas deben crear su llave pública y privada.
2. Ambas personas pueden compartir su llave pública a las otras partes (recuerda que esta llave es pública, no hay problema si la “interceptan”).
3. La persona que quiere compartir un mensaje puede usar la llave pública de la otra persona para cifrar los archivos y asegurarse que solo puedan ser descifrados con la llave privada de la persona con la que queremos compartir el mensaje.
4. El mensaje está cifrado y puede ser enviado a la otra persona sin problemas en caso de que los archivos sean interceptados.
5. La persona a la que enviamos el mensaje cifrado puede usar su llave privada para descifrar el mensaje y ver los archivos.

Puedes compartir tu llave pública pero nunca tu llave privada.

En la siguiente clase vamos a crear nuestras llaves para compartir archivos con GitHub sin correr el riesgo de que sean interceptados.

9) Configura tus llaves SSH en local

**Primer paso: Generar tus llaves SSH**. Recuerda que es muy buena idea proteger tu llave privada con una contraseña.

ssh-keygen -t rsa -b 4096 -C "tu@email.com"

**Segundo paso**: Terminar de configurar nuestro sistema.

**En Windows y Linux**:

# Encender el "servidor" de llaves SSH de tu computadora:

eval $(ssh-agent -s)

# Añadir tu llave SSH a este "servidor":

ssh-add ruta-donde-guardaste-tu-llave-privada

**En Mac**:

# Encender el "servidor" de llaves SSH de tu computadora:

eval "$(ssh-agent -s)"

# Si usas una versión de OSX superior a Mac Sierra (v10.12)

# debes crear o modificar un archivo "config" en la carpeta

# de tu usuario con el siguiente contenido (ten cuidado con

# las mayúsculas):

Host \*

AddKeysToAgent yes

UseKeychain yes

IdentityFile ruta-donde-guardaste-tu-llave-privada

# Añadir tu llave SSH al "servidor" de llaves SSH de tu

# computadora (en caso de error puedes ejecutar este

# mismo comando pero sin el argumento -K):

ssh-add -K ruta-donde-guardaste-tu-llave-privada

10) Conexión a GitHub con SSH

Luego de crear nuestras llaves SSH podemos entregarle la llave pública a GitHub para comunicarnos de forma segura y sin necesidad de escribir nuestro usuario y contraseña todo el tiempo.

Para esto debes entrar a la [Configuración de Llaves SSH en GitHub](https://github.com/settings/keys), crear una nueva llave con el nombre que le quieras dar y el contenido de la llave pública de tu computadora.

Ahora podemos actualizar la URL que guardamos en nuestro repositorio remoto, solo que, en vez de guardar la URL con HTTPS, vamos a usar la URL con SSH:

git remote set-url origin url-ssh-del-repositorio-en-github

11) Tags y versiones en Git y GitHub

Los tags o etiquetas nos permiten asignar versiones a los commits con cambios más importantes o significativos de nuestro proyecto.

Comandos para trabajar con etiquetas:

* Crear un nuevo tag y asignarlo a un commit: **git tag -a nombre-del-tag id-del-commit**.
* Borrar un tag en el repositorio local: **git tag -d nombre-del-tag**.
* Listar los tags de nuestro repositorio local: **git tag** o **git show-refs --tags**.
* Publicar un tag en el repositorio remoto: **git push origin --tags**.
* Borrar un tag del repositorio remoto: git tag -d nombre-del-tag y **git push origin :refs/tags/nombre-del-tag**.

Crea los alias para resumir comandos muy largos

alias ejemplo = “comando”

12) Manejo de ramas en GitHub

Puedes trabajar con ramas que nunca envias a GitHub, así como pueden haber ramas importantes en GitHub que nunca usas en el repositorio local. Lo importantes que aprendas a manejarlas para trabajar profesionalmente.

* Crear una rama en el repositorio local: **git branch nombre-de-la-rama** o **git checkout -b nombre-de-la-rama**.
* Publicar una rama local al repositorio remoto: **git push origin nombre-de-la-rama**.

Recuerda que podemos ver gráficamente nuestro entorno y flujo de trabajo local con Git usando el comando gitk.

13) Configurar múltiples colaboradores en un repositorio de GitHub

Por defecto, cualquier persona puede clonar o descargar tu proyecto desde GitHub, pero no pueden crear commits, ni ramas, ni nada.

Existen varias formas de solucionar esto para poder aceptar contribuciones. Una de ellas es añadir a cada persona de nuestro equipo como colaborador de nuestro repositorio.

Solo debemos entrar a la configuración de colaboradores de nuestro proyecto (Repositorio > Settings > Collaborators) y añadir el email o username de los nuevos colaboradores.

14) Flujo de trabajo profesional con Pull requests

En un entorno profesional normalmente se bloquea la rama **master**, y para enviar código a dicha rama pasa por un *code review* y luego de su aprobación se unen códigos con los llamados *merge request*.

Para realizar pruebas enviamos el código a servidores que normalmente los llamamos *staging develop* (servidores de pruebas) luego de que se realizan las pruebas pertinentes tanto de código como de la aplicación estos pasan a el servidor de producción con el ya antes mencionado *merge request*.