Git y GitHub

Comenzando a usar Git

Comandos básicos

- `git init`: se utiliza para iniciar nuestro repositorio.
- `git add ArchivoEjemplo.js`: crea el archivo pero no lo guarda de forma definitiva, lo almacena en (Staging Area).
- `git commit -m "versión 1"`: aquí se generan cambios de "Staging Area" y con (-m "") se deja un mensaje que nos sea útil.
- `git add .`: Agrega los archivos actualizados al repositorio, pero únicamente en la carpeta que te encuentras.
- `git commit -m "Cambios v1"`: sirve para generar cambios sobre la versión antigua.
- `git status`: sirve para revisar si has modificado o guardado los cambios hechos.
- `git log "archivo.txt"`: sirve para ver el historial del archivo.
- `git push`: sirve para enviar cambios al repositorio remoto.
- `git pull`: sirve para recibir cambios de repositorio remoto a local.

Ciclos 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 remoto 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 por que 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 por que falta ejecutar el comando git commit.
- **Archivos *Unstaged***: entiendelos como archivos "Traked 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 esta desactualizado, sus últimas versiones solo estan guardadas en su disco duro.

- **Archivos *Untracked***: son archivos que NO viven dentro de Git, solo en el disco duro. Nunca han sido afectados por git add, asi que Git no tiene registro de su existencia.

Recuerda que hay un caso muy raro donde los archivos tienen dos estados al mismo tiempo: Staged y Untracked. Esto pasa cuando guardas los cambios de un archivo en el área de Staging (con el comando git commit), 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 los archivos del estado Staged para devolverlos a su estado anterior. Si los archivos venian de Unstaged, vuelven alli. Y lo mismo si venian de Untracked.
- `git commit`: nos ayuda a mover archivos de Unstaged a Tracked. Esta es una ocasión especial, los archivos han sido guardados o actualizados en el repositorio. Git nos pedira que dejemos un mensaje para recordar los cambios que hicimos y podemos usar el argumento -m para describirlo (git commit -m “mensaje”).
- `git rm`: este comando necesita algunos de los 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 y cómo funciona un Merge?

Checkout es para cambiar de rama. Sólo la crea con el modificador -b. Unir dos Ramas lo conocemos como Merge.

Estándar de equipos de desarrollo…

- Rama ***Master*** o ***Main***: va a producción.
- Rama ***Development***: se alojan las nuevas features, características y experimentos
  (para unirse al Máster cuando estén definitivamente listas).
- Rama ***Hotfix***: issues o errores se solucionan aquí para unirse al Master tan pronto sea posible.

Crea un repositorio de Git y haz tu primer commit

Le indicaremos a Git que queremos crear un nuevo repositorio para utilizar un 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 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 vez los siguientes comandos con tu información:

```
git config --global user.name "tu nombre"
git config --global user.email "tu@email.com"
```

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

Analizar cambios en los archivos de tu proyecto

- `git log`: muestra la identificación de los commits.
- `git show`: nos muestra los cambios que han existido sobre un archivo y es muy útil para detectar cuando se produjeron ciertos cambios, qué se rompió y cómo lo podemos solucionar. Pero podemos ser mas detallados.
- `git diff`: nos muestra la diferencia entre una version y otra, no necesariamente todos los cambios desde la creación. (Gif diff commitA commitB).

Reset y checkout

- `git checkout + Id del commit`: podemos volver a cualquier versión anterior de un archivo específico o incluso de nuestro proyecto entero. Esta es también es la forma de movernos entre ramas.
- `git log --stat`: commit descriptivo con cantidad de líneas agregadas y removidas por archivo.
- `git reset --soft 'commit'`: mantiene los archivos en el área de Staging para que podamos aplicar nuestros últimos cambios desde un commit anterior.
- `git reset --hard 'commit'`: borra toda la información que tengamos en el área de Staging. (Perdiendo todo para siempre).
- `git checkout master 'archivo.txt'`: volvemos a la versión madre.

Ramas o branches

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).

```
git branch "nombre de rama" // crea una rama
git checkout "nombre de la rama" // cambiarse de rama
```

NOTA: Hacer commit antes de cambiar de ramas para no perder los datos.

Fusión de ramas con Git Merge

El comando git merge nos permite crear un nuevo commit con la combinación de dos ramas, la rama donde nos encontramos cuando ejecutamos el comando y la rama que le pasamos después del comando.

```
git merge "cualquier otra rama"
```

Resolución de conflictos al hacer un Merge

Git nunca borra nada a menos que se lo indiquemos. Cuando usamos los comandos git merge o git checkout estamos cambiando de rama pero creando un nuevo

commit, no borrando ramas ni commits (recuerda que puedes borrar commits con `git reset` / `git branch -d`)

Los archivos con conflictos por el comando git merge entran en un nuevo estado que conocemos como ***Unmerged***.
Funcionan muy parecido a los archivos en estado ***Unstaged***, algo así como un estado intermedio entre ***Untracked*** y ***Unstaged***, solo debemos ejecutar `git add` para pasarlos al área de ***Staging*** y git commit para aplicar los cambios en el repositorio.

Uso de GitHub

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 consola de mandos).

Luego de crear nuestra cuenta, podemos crear o importar repositorios, crear organizadores y proyectos de trabajo, descubrir respositorios 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 en 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 el 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 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
```

Manejo de ramas en GitHub

Puedes trabajar con ramas que nunca enviamos a GitHub, así como pueden haber ramas importantes en GitHub que nunca usas en el repositorio local. Lo importante es 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`.

Configurar múltiples colaboradores en un repositorio de GitHub

Por defecto, cualquiera 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.

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.

Ignorar archivos en el Repositorio con *.gitignore*

No todos los archivos que agregas a un proyecto deberían ir a un repositorio, por ejemplo cuando tienes un archivo donde están tus contraseñas que comúnmente tienen la extensión ***.env*** o cuando te estas conectando a una base de datos, son archivos que nadie debe ver.

Reconstruir commits en Git con *amend*

A veces hacemos un commit, pero resulta que no queríamos mandarlo porque faltaba algo más. Utilizamos `git commit --amend`, amend en inglés es remendar y lo que hará es que los cambios que hicimos nos los agrega al commit anterior.

Git *reset* y *reflog*: úsese en caso de emergencia

¿Qué pasa cuando todo se rompe y no sabemos qué está pasando?

Con git reset hashDelHEAD nos devolvemos al estado en que el proyecto funcionaba.

- `git reset --soft <hashDelHEAD>` te mantiene lo que tengas en Staging ahí.
- `git reset --hard <hashDelHEAD>` resetea absolutamente todo incluyendo lo que tengas en Staging.

Git reset es una mala práctica, no deberías usarlo en ningún momento; debe ser nuestro último recurso.

Buscar en archivos y commits de Git con *grep* y *log*

A medida que nuestro proyecto se hace grande vamos a querer buscar ciertas cosas.

Por ejemplo: ¿Cuántas veces en nuestro proyecto utilizamos la palabra color?

Para buscar utilizamos el comando git grep color y nos buscará en todo el proyecto los archivos en donde está la palabra color.

- **Con git grep -n color** nos saldrá un output el cual nos dirá en que línea está lo qué estamos buscando.
- **Con git grep -c color** nos saldrá un output el cual nos dirá cuantas veces se repite esa palabra y en qué archivo.

Si queremos buscar cuántas veces utilizamos un atributo de HTML lo hacemos con **git grep -c "atributo"**.