Apuntes de GIT

1. Introducción

1.1. Qué es un sistema de control de versiones?

Un sistema de control de versiones es un software que nos permite registrar los cambios que son realizados en un archivo o un conjunto de archivos a lo largo del tiempo, no es útil por el hecho de la facilidad de que varias personas trabajen en un mismo proyecto al mismo tiempo, tambien nos ayuda a que en caso de un error podamos volver a versiones pasadas funcionales de dicho proyecto.

1.2. Fundamentos de GIT

GIT se basa en los repositorios, un repositorio no es mas que el lugar donde se alojan las versiones de un proyecto junto con el log de cambios que se han hecho en ese proyects. Dichos repositorios pueden ser tanto locales como remotos

- Locales.

  Los repositorios locales son los que tenemos en nuestra computadora.

- Remoto.

  Los repositorios remotos son los que se ubican en un servidor, los cuales permiten que varias personas vean nuestro proyecto, que realicen cambios en el y que esos cambios sean sincronizados.

1.3. Configuración inicial

Antes de comenzar a usar GIT, se debe realizar ciertas configuraciones

1.3.1. Correo electronico

Para indicarle a GIT nuestro correo electrónico debemos ejecutar este comando:

```
$ git config --global user.email "reemplaza esto por tu correo"
```

Por ejemplo:

```
$ git config --global user.email "jschavarria77@gmail.com"
```

1.3.2. Nombre

Para indicarle a GIT nuestro nombre debemos ejecutar el siguiente comando:

```
$ git config --global user.name "reemplaza esto por tu nombre completo"
```

Por ejemplo:

```
$ git config --global user.name "Jaime Sebastian Chavarria Fuertes"
```

1.4. Configurar el editor de código que abre GIT

GIT tiene configurado abrir por defecto el editor de texto `Vim` para poder modificar los archivos cuando encuentra conflictos o cuando usamos el comando `$ git commit` sin la bandera y el argumento: `-m "descripcion del commit"`. En caso de querer cambiar el editor por defecto disponemos del siguiente comando:

```
$ git config --global core.editor "el comando para abrir tu editor"
```

En mi caso voy a colocar el editor neovim, entonces el comando sería:

```
$ git config --global core.editor "nvim"
```

1.5. Comprobar la configuración de GIT

Para poder ver la configuración que tenemos en git podemos usar el siguiente comando:

```
$ git config --list
```

Que nos producira una salida de este tipo:

```
user.email=202301300@est.umss.edu
user.name=Jaime Sebastian Chavarria Fuertes
core.repositoryformatversion=0
core.filemode=true
core.bare=false
core.logallrefupdates=true
```

Podemos jugar con las banderas de `list`, por ejemplo:

- Para poder mostrar la unicamente la configuración global:

  ```
  $ git config --global --list
  ```

- Para poder mostrar la configuración del repositorio local:

  ```
  $ git config --local --list
  ```

Y muchas mas que se pueden revisar en este enlace de la documentación de GIT

1.6. Cómo inicializar un nuevo proyecto en GIT?

Para crear un nuevo repositorio local podemos utilizar el comando:

```
$ git init <direccion de la carpeta>
```

Este comando nos creara una carpeta con el nombre asignado en la dirección.

Pero si en caso de tener ya un proyecto podemos ejecutar el mismo comando dentro de la carpeta raíz de nuestro proyecto pero sin pasarle de argumento nada.

```
$ git init
```

Y asi ya habremos comenzado a utilizar GIT al fin.

2. States (Estados en GIT)

2.1. Los 3 estados en GIT

GIT maneja internamente tres estados distintos por los que pueden pasar los archivos dentro de un repositorio:

- **Modificado (modified):** El archivo ha sido cambiado en el sistema de archivos, pero esos cambios aún no han sido registrados por GIT para una futura confirmación.
- **Preparado (staged):** El archivo está listo para ser confirmado. Ha sido agregado al área de preparación, conocida como "staging area".
- **No modificado (unmodified):** El archivo está exactamente igual que en la última versión confirmada. No hay cambios pendientes.

Estos estados son importantes porque GIT nos permite controlar exactamente qué cambios queremos guardar y cuáles no, utilizando comandos que modifican el estado de los archivos.

2.2. Cómo deshacer un archivo modificado?

Si realizamos cambios en un archivo pero luego queremos descartarlos y volver a la versión original (la última registrada), podemos usar el comando:

```
$ git restore <ruta al archivo>
```

Este comando elimina los cambios hechos en el archivo y lo deja exactamente como estaba en el último registro conocido por GIT

2.3. Cómo añadimos archivos al área de staging

Para marcar un archivo como "preparado" (pasarlo al estado **staged**), usamos el comando:

```
$ git add <archivo>
```

También podemos añadir todos los archivos modificados de una vez con:

```
$ git add .
```

Esto es útil cuando tenemos varios cambios y queremos registrarlos juntos más adelante.

2.4. Cómo puedo sacar uno o varios archivos del área?

Si accidentalmente agregamos un archivo al área de staging, pero queremos quitarlo de ahí (sin perder los cambios hechos), usamos:

```
$ git restore --staged <ruta al archivo>
```

Esto lo devuelve al estado de modificado, permitiéndonos decidir luego si lo queremos preparar o no.

También podemos desagregar todos los archivos preparados de una vez:

```
$ git restore --staged .
```

Este comando limpia el área de staging, sin afectar el contenido modificado de los archivos.

3. Commits

3.1. Qué es un commit?

Un `commit` en GIT representa un punto en el tiempo que guarda el estado de los archivos que fueron preparados (staged). Es una especie de "fotografía" del proyecto en un momento determinado. Cada commit contiene información como:

- El autor que realizó el cambio.
- La fecha y hora del commit.
- Un mensaje descriptivo de los cambios.
- Un hash único que lo identifica.

Gracias a los commits, es posible regresar a versiones anteriores, comparar diferencias entre estados y colaborar con otros sin sobrescribir trabajo.

3.2. Cómo puedo hacer un commit?

Una vez que uno o varios archivos han sido agregados al `staging area`, se puede registrar ese conjunto de cambios usando el siguiente comando:

```
$ git commit -m "mensaje descriptivo del cambio"
```

La bandera `-m` sirve para escribir el mensaje directamente desde la línea de comandos.

Por ejemplo:

```
$ git commit -m "agregada la seccion de introduccion"
```

Si omitimos `-m`, GIT abrirá el editor configurado (por defecto `Vim`) para que escribamos el mensaje del commit.

Nosotros lo tenemos configurado como `nvim`

3.3. Qué es el HEAD?

`HEAD` es un puntero que indica el **commit actual** en el que estamos trabajando. Siempre apunta al último commit en la rama activa.

Cuando realizamos un nuevo commit, `HEAD` se mueve al nuevo commit creado. Si cambiamos de rama, `HEAD` apuntará al commit más reciente de esa nueva rama.

También podemos ver el contenido de `HEAD` con:

```
$ cat .git/HEAD
```

O podemos ver hash del commit exacto al que apunta:

```
$ git rev-parse HEAD
```

Saber qué es `HEAD` es importante para entender cómo funcionan los cambios de rama, los rebase, y otros comandos avanzados de GIT.

4. Ramas

4.1. Qué es una rama y para que sirve?

Una rama en GIT es una linea independiente de desarrollo dentro de un repositorio. Nos permite trabajar en características, ideas o cambios sin afectar la rama principal del proyecto. La rama por defecto en GIT se llama `master` o `main`, dependiendo de la configuracion inicial.

Las ramas son especialmente utiles para:

- Experimentar sin romper el proyecto principal.
- Desarrollar nuevas caracteristicas de forma aislada.
- Organizar el trabajo entre distintos desarrolladores.

Cuando creamos una rama, esta copia el estado actual del repositorio, y desde ese punto podemos trabajar de forma independiente. Luego, si los cambios funcionan, podemos integrarlos a la rama principal.

4.2. Trabajar con ramas

Para comenzar a trabajar con ramas, necesitamos saber como crearlas, listarlas, y cambiarnos entre ellas.

4.2.1. Crear una nueva rama

Para crear una nueva rama usamos el comando:

```
$ git branch nombre-de-la-rama
```

Esto crea la rama pero no nos mueve a ella. Para cambiarnos, usamos:

```
$ git switch nombre-de-la-rama
```

O tambien para crearla y movernos a esa rama en un solo paso agregamos la bandera `-c`

```
$ git switch -c nombre-de-la-rama
```

4.2.2. Ver las ramas disponibles

Para ver todas las ramas del repositorio local, usamos:

```
$ git branch
```

Esto mostrara un listado de ramas, y marcará con un asterisco la rama en la que estamos actualmente.

4.2.3. Cambiar entre ramas

Para cambiar a otra rama existente usamos:

```
$ git switch nombre-de-la-rama
```

4.2.4. Eliminar una rama

Para eliminar una rama local usamos:

```
$ git branch -d nombre-de-la-rama
```

Si GIT nos dice que la rama no se puede eliminar porque no ha sido fusionada (merged), y aun asi queremos borrarla, podemos usar la opcion `-D` (mayuscula):

```
$ git branch -D nombre-de-la-rama
```

5. Merge

5.1. Fusionar ramas

Una de las operaciones mas comunes en GIT es fusionar (merge) ramas. Esto significa tomar los cambios de una rama y aplicarlos en otra. Lo mas habitual es trabajar en una rama secundaria (por ejemplo, `feature-x`) y luego fusionarla con la rama principal (`main`).

Para fusionar ramas, primero debemos cambiarnos a la rama que queremos actualizar. Por ejemplo, si queremos fusionar `feature-x` en `main`, primero hacemos:

```
$ git switch main
```

Luego ejecutamos el comando de fusion:

```
$ git merge feature-x
```

GIT intentara aplicar todos los cambios de `feature-x` sobre `main`. Si no hay conflictos, la fusion se realiza automaticamente y se genera un nuevo commit de merge.

Es importante entender que el commit de merge tiene como padres a los dos ultimos commits de las ramas involucradas, dejando un registro claro del punto en que las ramas fueron unificadas.

En caso de que GIT no pueda fusionar automaticamente, se requerira una resolucion de conflictos, lo cual veremos en la siguiente sección.

5.2. Resolver conflictos

Para resolver un conflicto, debemos abrir manualmente los archivos que GIT marco como conflictivos y decidir que cambios mantener.

GIT marca las partes en conflicto con estas señales especiales dentro del archivo:

```
<<<<<<< HEAD
contenido de la rama actual (donde estas parado)
=======
contenido de la rama que se esta fusionando
>>>>>>> feature-x
```

El contenido entre `<<<<<<< HEAD` y `=======` representa los cambios que existen en la rama actual (en la que estas parado). El contenido entre `=======` y `>>>>>>> feature-x` muestra los cambios que vienen desde la rama que estas intentando fusionar (en este caso, `feature-x`).

Debemos editar el archivo para quedarnos solamente con el contenido deseado (puede ser una mezcla de ambos o solo uno) y eliminar completamente las marcas de conflicto (`<<<<<<<`, `=======`, `>>>>>>>`).

Además, si usamos un editor de texto o vemos el conflicto en terminal con el comando:

```
$ git diff
```

GIT nos puede mostrar diferencias de esta forma:

```
<<<<<<< HEAD
- linea que esta en la rama actual
+ linea que esta en la rama que se esta intentando fusionar
=======
```

A veces también veremos prefijos como:

```
- contenido eliminado
+ contenido a\~nadido
```

O en el caso de conflicto de lineas:

```
@@ -3,7 +3,7 @@
 linea 1
-linea vieja
+linea nueva
 linea 2
```

Estos signos indican que hay diferencias linea por linea. Las lineas que comienzan con - estaban en la rama actual, y las que comienzan con + vienen desde la rama que se quiere fusionar.

Una vez resueltos todos los conflictos editando los archivos, los agregamos al area de staging con:

```
$ git add archivo.txt
```

Finalmente, para completar la fusion, debemos hacer el commit correspondiente:

```
$ git commit
```

Este commit guardara los cambios resultantes de la resolucion de conflicto. A veces GIT abre un editor para escribir un mensaje indicando que hubo conflictos resueltos manualmente.

Es muy importante verificar que los archivos funcionan correctamente despues de la resolucion para evitar errores lógicos o de ejecución.

6. Eliminar ramas

6.1. Por qué eliminar ramas?

En GIT, las ramas permiten trabajar en diferentes funcionalidades o correcciones de manera aislada. Sin embargo, si no se eliminan cuando ya no se necesitan, el repositorio puede volverse difícil de mantener.

Algunas razones para eliminar ramas:

- Reducir el desorden visual al listar ramas.
- Evitar confusiones sobre el estado actual del proyecto.
- Mantener una estructura limpia en el flujo de trabajo.

6.2. Cómo podar las ramas?

Cuando trabajamos con un repositorio remoto, muchas veces se eliminan ramas en el servidor pero siguen apareciendo en nuestro repositorio local. Para limpiar ("podar") esas referencias obsoletas podemos usar:

```
$ git remote prune origin
```

Esto eliminará las referencias locales a ramas remotas que ya no existen en el servidor remoto. Es una buena práctica hacerlo de vez en cuando si trabajamos con muchas ramas.

6.3. Cómo eliminar ramas de mi repositorio local que ya no se usan?

Para eliminar una rama local que ya no necesitamos, usamos:

```
$ git branch -d nombre-rama
```

Este comando verifica que la rama ya haya sido fusionada con la rama actual. Si no ha sido fusionada y aún así quieres eliminarla, puedes forzar la eliminación con:

```
$ git branch -D nombre-rama
```

Para asegurarte de que ya no la necesitas, puedes revisar primero si fue fusionada:

```
$ git branch --merged
```

Este comando muestra las ramas que ya han sido fusionadas a la actual.

Si deseas listar las ramas que **no** han sido fusionadas, puedes usar:

```
$ git branch --no-merged
```

7. GitHub

7.1. Git y GitHub: Son lo mismo?

Git y GitHub no son lo mismo, aunque están muy relacionados.

- **Git** es un sistema de control de versiones distribuido, lo que significa que puedes gestionar tu código de manera local sin necesidad de conexión a internet.
- **GitHub** es una plataforma en línea que permite almacenar proyectos Git de manera remota, facilita la colaboración y proporciona herramientas para gestionar código en equipos. Es, en esencia, un servicio de alojamiento basado en Git.

Git es la herramienta de control de versiones, mientras que GitHub es el servicio donde se aloja y comparte ese código.

7.2. Repositorios Remotos

Un **repositorio remoto** es una versión del repositorio que se encuentra alojada en un servidor, generalmente en la nube (como en GitHub). Estos repositorios permiten la colaboración en línea y sirven para tener una copia de seguridad del proyecto.

Para trabajar con un repositorio remoto, se usa el comando `git remote`. Este comando se utiliza para vincular un repositorio local con uno remoto, lo que te permitirá sincronizar los cambios entre ambos.

```
git remote add origin https://github.com/usuario/repositorio.git
```

Este comando agrega un repositorio remoto llamado `origin` que apunta a la URL del repositorio en GitHub.

7.3. Clonando un Repositorio Remoto Creado Previamente

Si ya existe un repositorio remoto en GitHub y deseas obtenerlo en tu computadora, puedes clonarlo utilizando el siguiente comando:

```
git clone https://github.com/usuario/repositorio.git
```

Este comando crea una copia local exacta del repositorio remoto. Con esto, ya puedes comenzar a trabajar en tu copia local, haciendo cambios y luego subiéndolos al repositorio remoto.

7.4. ¿Cómo Enlazar un Repositorio Local con uno Remoto?

Si ya tienes un repositorio local y quieres vincularlo a un repositorio remoto en GitHub, debes usar el siguiente comando:

```
git remote add origin https://github.com/usuario/repositorio.git
```

Este comando vincula tu repositorio local con el repositorio remoto en GitHub, de modo que puedas hacer operaciones como `push` y `pull`.

7.5. Escribiendo en el Repositorio Remoto

Para subir tus cambios al repositorio remoto, utilizas el comando `git push`. Este comando empuja los cambios locales de tu rama actual al repositorio remoto.

Si tu repositorio local está vinculado correctamente con el remoto, solo tienes que escribir:

```
git push origin main
```

Esto enviará tus cambios de la rama `main` al repositorio remoto llamado `origin`.

7.6. No me deja hacer `push`, me dice que el cambio ha sido rechazado

Este error generalmente ocurre porque hay cambios en el repositorio remoto que no están presentes en tu repositorio local. Git te impedirá hacer un `push` si tu rama local está detrás de la rama remota.

La solución más común es hacer un `git pull` para traer los cambios del repositorio remoto antes de hacer el `push`. El flujo de trabajo sería el siguiente:

```
git pull origin main
git push origin main
```

El comando `git pull` traerá los cambios del repositorio remoto y los fusionará con tus cambios locales. Después de esto, ya podrás hacer el `push` sin problemas.

7.7. Creando una Rama Remota

Para crear una nueva rama en tu repositorio local y luego subirla al repositorio remoto, debes seguir estos pasos:

Primero, crea y cambia a una nueva rama:

```
git checkout -b nueva-rama
```

Luego, empuja esta nueva rama al repositorio remoto con el siguiente comando:

```
git push -u origin nueva-rama
```

El parámetro `-u` establece la rama remota `origin/nueva-rama` como la rama de seguimiento para tu rama local. Esto significa que las futuras operaciones de `push` y `pull` se realizarán por defecto en esa rama remota.

7.8. Explicación de los Comandos `push`, `pull` y `fetch`

Los comandos `git push`, `git pull`, y `git fetch` son fundamentales para trabajar con repositorios remotos en Git.

7.8.1. git push

El comando `git push` se usa para enviar tus cambios locales al repositorio remoto. Si tienes commits en tu rama local que aún no han sido subidos, puedes usar este comando para "empujarlos" al servidor remoto.

Ejemplo:

```
git push origin main
```

Este comando sube tus cambios de la rama `main` al repositorio remoto `origin`.

7.8.2. git pull

El comando `git pull` se utiliza para traer los cambios desde el repositorio remoto y fusionarlos con tu rama local. Este comando es una combinación de `git fetch` (que descarga los cambios) y `git merge` (que los fusiona con tu rama local).

Ejemplo:

```
git pull origin main
```

Este comando descarga y fusiona los cambios desde la rama `main` del repositorio remoto `origin` con tu rama local.

7.8.3. git fetch

El comando `git fetch` solo descarga los cambios desde el repositorio remoto sin hacer ninguna fusión con tu rama local. Es útil cuando solo quieres ver qué cambios han ocurrido en el repositorio remoto sin que se apliquen automáticamente a tu código local.

Ejemplo:

```
git fetch origin
```

Este comando descarga todos los cambios del repositorio remoto sin alterar tu rama local.