**¿Por qué Git y GitHud?**

control de versiones. Nos ayuda a llevar los estados de nuestros proyectos.

Conjunto de comandos que realiza cosas especificadas.

**¿QUE ES UN SISTEMA DE CONTROL DE VERSIONES?**

Registra los cambios realizados sobre un archivo o conjunto de archivos a lo largo del tiempo.

Cuando hacemos algo en word, por ejemplo, al dar control z nos devolvemos a el estado anterior del archivo.

Realizaremos una web mientras usamos git.

**CLASIFICACIÓN:**

**Local:** Que vive en nuestra laptop.

**Centralizado**: Lo hace más grande, porque se guarda en el server. O sea, hay se almacena el proyecto permitiendo distribuirlo para múltiples dispositivos.

**Distribuidos:** Git es distribuido, entonces cada usuario que participe en el proyecto tiene una copia del proyecto que realiza, por eso no dependemos de un solo computador que almacena la información.

**GIT:**

Open source linux, se asoció con Beatkeeper. y cuando se rompen las relaciones, linux crea su propio sistema distribuidos de control de versiones. Haciéndolo veloz, sencillo, completamente distribuido.

Desarrollo no lineal: podemos abrir cosas cuando queramos a partir del proyecto principal.

Git al igual almacena una referencia de los archivos que no se han cambiado.

Cualquier trabajo es local, Entonces puedes trabajar y luego subirlo apenas tengas internet. comits.

Integridad: no se pierde la información sin que lo detecte.

**LOS 3 ESTADOS DE GIT**

**Working Directory:** Inicialmente entra aquí.

**Staging Area:** pasa por el área de preparación donde se inicia el proyecto.

**Git Repository:** commit se guarda.

**¿Qué es GitHub?**

Pueden trabajar en conjunto, pero también separado.

**Git:** el software de versiona miento.

**GitHub:** es la plataforma host para almacenar los cambios en la red.

Y además funciona como una red social.

Para instalar en WINDOWS, seguir los pases y listo.

**Linux**: Si eres usuario de linux usar la consola debe ser algo con lo que te topas en tu día a día, otra ventaja es que en muchos sistemas de Código abierto ya tiene git preinstalado así que probemos eso primero.

ve a tu terminal y escribe git --version si te muestra la versión de git todo está perfecto y puedes continuar con las clases si no es tu caso instálalo según tu distribución desde la terminal corriendo los siguientes comandos.

En Ubuntu o distribuciones derivadas:

sudo apt-get install git

**COMANDOS ÚTILES EN LA TERMINAL GIT BASH**

Tenemos que aprender algunos comandos sencillos, para trabajar con Git.

El primero es cd que nos permite movernos dentro de carpetas. Con cd … podemos salir de las carpetas.

Para crear carpetas vamos a usar mkdir y ponemos el nombre de la carpeta que vamos a crear.

Con el comando ls vamos a ver las carpetas que tenemos creadas.

Para limpiar la terminal vamos a usar el comando clear. Este nos sirve para hacer scroll e ir al principio de la terminal.

El comando touch nos permite crear archivos.

Para borrar archivos vamos a usar el comando rm y el nombre del archivo.

De la misma forma para borrar carpetas usamos rm -rf y el nombre de la carpeta.

Otra herramienta que vamos a usar dentro de Git para editar archivos con el editor que tengamos en nuestra máquina, pero dentro de la terminal.

Con el comando \_:wq \_vamos a guardar los cambios que hicimos dentro del archivo que habíamos creado.

PARA ENTRAR EN WINDOWS EN UNA CARPETA, DEBEMOS INGRESAR CARPETA A CARPETA CD

HASTA LLEGAR A LA DESTINO

/d/Alejo/CursosYLenguajes/GitHub

PARA SALIR cd ..

ls vemos las carpetas.

touch crear archivo.

rm file.txt borrar archivos

rm -rf repositorio2 borrar carpeta

editar archivos: vim archivo.txt

---PARA GUARDAR LO QUE EDITE ----

escape + : y guardar wq con esto lo guardamos.

sin guardar solo q 0

**COMANDOS GIT**

**git init:** creando repositorio. o iniciar.

**(MASTER) ES UNA RAMA:** quiere decir que está en el repositorio.

AHÍ YA PODEMOS CREAR ARCHIVOS

CARPETA OCULTA: ES DONDE VIVE, SI NOS DESHACEMOS SE DESAPARECE EL REPO.

Borrar repositorio: rm -rf .git

**GIT ADD**

Agregado, quitar y viendo el estado de los archivos.

Creamos un archivo nuevo y para agregarlo.

git status: miramos el estado del archivo si está en rojo

es porque está en nuestro working directory y con

git add: agregamos a git los archivos.

YA EN VERDE FALTA AGREGARLO A COMMIT

sí lo quiero borrar

git rm --cached file

---- git add -A agrega todo lo que esta para ser agregado.

para quitarlos también

----- git rm -f archivos, se borra de todo lugar

--- git add -n archivo: no hace la agregación, o sea, miramos lo que en realidad esta

aquí estamos en Staging

**GIT COMMIT**

CONFIRMAMOS CAMBIOS

Pasamos a nuestro repositorio git los archivos.

got commit -m "Inicializar nuestro landing"

lo que hicimos fue agregar todo a el repository(3er estado git)

Como hago cuando realice un commit sin todo

Entonces.

git commit -m 'concateno cambios' -amend = lo que hace es concatenar con el cambio previo.

O SOLO git commit --amend

git log: ver los commit

**GIT TAG**

Etiquetando confirmaciones.

Son ligeras o anotadas: Ligeras se crean con el nombre de la etiqueta, las anotaciones son las que llevan un mensaje usando -m podemos dejar mensaje.

Si queremos borrarlas -d, si las queremos listar usamos -l

y si las queremos renombrar -f

Utilizamos Sha-1 podemos crear etiquetas anotadas haciendo referencia a commit viejos.

git tag -a 0.5 -m 'Versión estable del proyecto'

// así etiquetamos el ultimo commit.

// y así agregamos de pasados

git tag 0.3 f944a1212962e1bc1d86f3dda37469663bf1337d

// Y PARA VERLO

git tag -l

// borrar etiqueta

git tag -d 1.0

// renombrar

git tag -f -a 0.1 -m 'Iniciando provecto' 218e960adc15622a75b78099ff53da5e7b68ff8a

// luego se puede borrar la versión que no estaba bien.

**GIT LOG**

Revisa las historias de nuestro proyecto

logs muestra los códigos hexadecimales que crea el Sha-1, el auto y fecha del commit.

-- git log --oneline: pones los commits más reducidos. Lo cual al tener muchos es mejor verlos así.

--git log --oneline --graph: los graficos de los commits.

--git log -'Numero de omits?: los comits que quiero ver. La cantidad que quiera.

**GIT DIFF**

Revisa los cambios entre las versiones.

Los cambios que hay entre commits se usan los Sha-1 para ver la comparación.

git diff 218e960 PONEOS EL Sha-1 y vemos los cambios que hay, de ese commit al final en el que estamos.

Así comparamos con otro commit

git diff "Sha-1" "Sha-1"

-- En rojo aparece al comparar con la que no nuevo con el anterior y en verde, el que anterior (que no tenía) con el nuevo (que tien).

-- git diff [version 1] vs [version 2]

**GIT RESET --SOFT**

Con este comando, podemos borrar commits que no debimos guardar, pero mantiene los cambios.

Pero debemos tener cuidado de no borrar otros.

--soft [Sha-1]: nos permite identificar el commit que queremos resetear.

Entonces se pone desde el sha anterior al que quiero quitar.

Los archivos no los cambia.

O reescribir en el commit

--mixed:

--hard:

**GIR RESET --MIXED**

Se usa para hacer cambios en commits que pudieran tener errores.

Descarta cambios, pero no deja los archivos en stach.

nos dice que hacer

Cambia

**GIT RESET --HARD**

este si borra absolutamente todo

Si están en el workin directory no los borra, porque no están todavía en git.

con commits: git reset -hard Sha-1

para recuperar los datos si borre todo, se usa

el ultimo sha-1, por eso si borro algo debo tener dos ventanas, para tener los sha1 en otro lado por si se nos borra todo.

git reset -hard sha1 ultimo

git log

git reset HEAD^ --hard

**CONFIGURAR OTRO EDITOR DE TEXTO**

Como configuro cualquier editor de texto, para usar git.

git config --global core.editor vim : se abre la configuración

otro editor

git config --global core.editor "atom --wait" (o el editor de código que quiera)

**GIT BRANCH (MULTIPLES VARIABLES DEL REPOSITORIO)**

Con esto, podemos usar más ramas en el proyecto.

(Master) es la rama principal, con ella inicia el proyecto git.

cuando necesito puedo crear otras ramas, para trabajar con un equipo.

Pero lo mejor es trabajar en una rama aparte, por si hago un bug, en la principal todo seguiría bien.

git branch nombre

para verlas

git branch -l

borrar

git branch -d nombre

si tiene commit borrar

git branch -D nombre

renombrar rama

git branch -m nombre nuevoNombre

**GIT CHECKOUT (MOVIÉNDONOS ENTRE RAMAS Y VERSIONES)**

Con este comando, nos movemos en el tiempo sin borrar nada, este comando ayuda a cazar bugs o verificar el estado de nuestro proyecto en determinado momento. moviéndonos sha a sha o bueno, comit a comit.

Y también podemos crear ramas e ingresar a ella directamente:

git checkout -b

Movernos en el tiempo con los commit a través de los Sha-1

git checkout 'sha-1'

Si encontramos un bug, tenemos que crear una para solucionar el bug pero desde el master y no en la rama del bug.

**TRABAJANDO ENTRE RAMAS**

sí ocurre un error podemos ir rama a rama buscando el bug.

si no agregamos algo en una rama, ello se mueve a otra rama esperando el commit.

Ya puedo ir trabajando poco a poco versión a versión

**GIT MERGE (MEZCLANDO ENTRE RAMAS)**

REFLEJAR LAS RAMAS EN NUESTRA RAMA MASTER.

git merge 'nombre rama a mezclar'

aparece como un Fast-forward al realizarse la mezcla porque es la manera más fácil al ser el primero

para mezclar:

git merge nombreRama a maezclar

si sale fast-forward quiere decir que mezclamos cosas directamente con la master

Si vas después:

será recursivo, debemos agregar cada cambio y luego hacer un commit y listo, si no hay conflicto con archivos todo saldrá bien un commit m´ás y ya

saldrá luego todo y realiza el commit que pide realizar cambios.

merge con conflictos: git merge nuva-imagen

y ya decido con que quedarme.

**GIT REBASE**

Rescribe la historia del proyecto es otra manera de hacer mezclas de las ramas en el proyecto.

Con este solo hacemos un commit y no agregamos más con este solo tenemos una línea de tiempo con el log.

//creamos rama a mezclar con rebase

git checkout -b updateA

touch updateA

git add -A

git commit

// volvemos al master

git checkout master

creamos las ramas que necesitemos

y para agregarlas o bueno mezclarlas

// con este vemos que se agrega en curvas

git merge updateA

// con este no hay curva

git rebase updateB

// vemos los cambios

git log --oneline --graph

Con esto vemos como se agregan las cosas y se ve lineal.

git reset --hard commit

// la mezclamos interactivamente

git rebase -i updateA

y en nuestro editor veremos las opciones.

**GIT STASH**

GUARDANDO CAMBIOS TEMPORALMENTE

Si no estamos listos para confirmar el cambio, Stash nos permite mantener un estado intermedio antes de generar un commit. CAMBIOS SIN CONFIRMAR.

Entonces guardo sin hacer commit, guardamos el estado.

git stash

Si ingreso a otra rama

git checkout hotfix

puedo entrar sin problema porque tengo un estado guardado y puedo regresar a la otra rama

y restablecer el estado a la rama.

git stash list

con ese comando veré los cambios

Y los podemos eliminar también.

git stash drop stash@{0}

y si lo borre listo no pasa nada

sí creo un archivo

touch cambio2

debo agregarlo a git para poder entrar en stash

git add cambio2

Ahora si lo puedo stashear

git stash

git list

Ahora lo agregamos A COMMIT

con este aplicamos el último cambio.

git stash apply

// varios stash y con él a volvemos

git stash apply stash@{0}

**CHERRY PICK ELIGIENDO COMMITS SELECTIVAMENTE**

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

Si hacemos algo en una rama que no era:

Entonces debo sacar ese archivo de esta rama y mandarlo a la que es

y luego mezclar en master.

Entonces entramos a la rama donde si debimos hacer el commit y lo movemos con el comando.

git cherry-pick 994dded

Ahora so podemos mezclar con nuestra rama master.

git merge hotfix2

**Desafío git**

Crear un repositorio y crear un flujo git log --oneline --graph

múltiples commit por cada rama.

hotfix, css, imagenes RD y master.

**CREANDO CUENTA GITHUB**

Vamos a la web de GitHub que nos brinda el repositorio remoto.

Creamos la cuenta y nos loguemos.

¿Qué es? Una red social para desarrolladores, que nos permite trabajar y colaborar con la comunidad.

**GIT CLONE/FORK (CLONANDO REPOSITORIOS REMOTOS)**

el \*\*fork \*\*en github nos hace una copia del proyecto en nuestro perfil/repositorios para poder hacerle lo que queramos sin afectar el proyecto original.

Vamos a crear nuestro primer proyecto en GitHub.

Nuevo repositorio

Asigna un nombre a tu repositorio

Asigna una descripción a tu repositorio

Puedes adicionar un ““read me””

Crea el repositorio

En la parte derecha encuentras una opción que se llama clone o download si no es nuestra intención compartirlo.

Un fork va a clonar el proyecto y nos va a crear un nuevo proyecto en el GitHub.

CREAMOS EL REPO: PODEMOS INVOCARLO DESDE EL LOCAL CON UNA CLONACIÓN.

PUEDO DESCARGARLO ZIP PARA USAR ALGO O CLONAR PARA TENERLO EN LOCAL.

\*\*\*DENTRO DE LA CONSOLA PONEMOS git clone https://github.com/crojastriana3/invie-github.git

\*\* SI NO SOY DUEÑO LE HAGO UN Fork, nos permite clonar el proyecto y me agrega un proyecto en mi propio GitHub para poder trabajar en él.

EN EL YO PUEDO HACER CAMBIOS Y SOLICITAR AL DUEÑO QUE LOS ACEPTE.

**AÑADIENDO CLAVE SSH S GITHUB**

Si quieres mantener tus seguros dentro de GitHub es importante usar llaves ssh con el comando:

ssh-keygen -t rsa -b 4096 -C "correo"

Con esto, podemos comunicarnos con el local de manera segura.

Así no nos pedirá usuario y contraseña siempre.

la creamos en consola: ssh-keygen -t rsa -b 4096 -C "correo"

TIENE QUE SE EL CORREO QUE TIENE EN GitHub

\*\* PARA COPIAR LA LLAVE USAMOS:

cat ~/.ssh/id\_rsa.pub

**GIT REMOTE (AÑADIENDO UN REPOSITORIO REMOTO A UNO LOCAL)**

Tenemos que crear una conexión entre tu repositorio local y tu repositorio remoto para que puedas compartirlo con la comunidad. Para esto vamos a usar un nuevo comando que en este caso es git remote. Por convención lo nombramos “origin”. Para borrarlo podemos usar git remote remove y el nombre del repositorio.

\*\*\* git remote add origin link ssh o hhtps

\*\* verlo git remote -v

\*\* eliminarlo git remote remove origin

**GIT PULL/FETCH (TRAYENDO CAMBIOS DESDE EL REPOSITORIO REMOTO)**

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

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

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

\*\*Al tener los cambios en el repo y tener conectado el local con el remoto, debemos ahora.

\*\* Si queremos descargar archivos del remoto: git fetch origin master

\*\*ahora los mezclamos: git merge origin/master --allow-unrelated-histories

\*\* O PARA NO HACER FETCH Y MERGE SOLO USAMOS:

pull: git pull origin master

--Con esto ya tenemos en local las cosas para trabajar con el remoto.

**GIT PUSH (ENVIANDO CAMBIOS AL REPOSITORIO REMOTO)**

Para enviar nuestro código al repositorio remoto vamos a usar el comando git push. Así logramos que otras personas tengan acceso a nuestro proyecto.

Cuando esto termine tendremos acceso a nuestro trabajo desde la plataforma de GitHub.

Lo primero que debemos hacer es enviar nuestros tags en el proyecto.

también recordemos que podemos evitar ramas, lo que también podemos hacer a través del uso del comando git push [nombre de la rama].

\*\* Enviar al remoto solo el master: git push origin master

\*\*\* Para enviar las demás ramas y tags: git push origin master --tags -- Otra rama: git push origin RD

**EXPLORANDO LA PLATAFORMA DE GITHUB**

Podemos calificar los proyectos que hemos creado.

Podemos ver las estadísticas de nuestro proyecto.

También las personas que le han hecho fork.

Podemos proteger ciertas ramas del proyecto.

\*\*Siempre protejan la rama master. Para que vaya a master tiene que tener su proceso del pull request.

Unwatch (Nos permite que nos lleguen notificaciones si hay cambios en el repositorio)

Not watching (Se te notifica cuando participas o mencionas)

Ignoring (Nunca ser notificado)

Star (así como en Facebook se vive a base de likes, aquí se vive a base de estrellas, cuantas más estrellas más valorado es ese proyecto, y a su misma vez tiene muchos forks)

Fork (Indica la cantidad de gente que hizo copia de su repositorio para poderles enviar cambios)

Podemos ver las estadísticas de nuestro proyecto en Insights

contributors (Nos permite ver el avance de nuestro proyecto)

traffic (Nos muestra el tráfico de nuestro proyecto)

punch card (Muestra los días que ha estado trabajando)

network (Vemos el flujo de ramas)

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

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

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

Settings: Configuraciones del GitHub.

collaborators (aquí se añade a más gente para que colabore en este proyecto)

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

CONSEJO: Protege siempre la rama master (Una vez protegida ningún cambio va a ir directamente a master tenemos que pasar por cierto proceso para que un Código vaya a master)

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

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

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

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

Pull requests (Vemos una lista de pull requests)

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

Proyects (Nos aparece nuestros proyectos)

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

**PROYECTOS GITHUB**

A cada repositorio se le ´puede crear un proyecto.

Ejemplo para el Resposive Design seria uno y así para cada parte grande de un proyecto.

Y se puede orgranizar por columnas las tareas.

**CREANDO UN TEMPLATE PARA ISSUES**

Lo ideal es que siempre que creemos un proyecto tengamos un template para enviar pull requests o cuando tengamos issues.

En este, pedimos la información mediante preguntas de lo que queremos saber de un issue o un pull.

**Creando un template para Pull Request**

Nos permite tener pautas con los colaboradores.

Con estas pautas pueden solicitar un Pull Request.

**GITIGNORE (IGNORANDO ARCHIVOS NO DESEADOS)**

Si tienes archivos que no pueden ser públicos, como archivos de configuración con contraseñas, lo ideal es que no los subas a tu repositorio, estos archivos los puedes poner en el archivo. gitignore.

En el agregamos todos los archivos que requerimos en local pero no en el remoto.

En esta web podemos encontrar el gitignore de nuestro proyecto-

https://www.toptal.com/developers/gitignore

**Pull Request (Colabora a proyectos externos)**

Un Pull Request es una solicitud para que el dueño del repositorio realice los cambios que estas proponiendo. Estos nunca se hacen a la rama master, para evitar inconvenientes.

Es la base de la colaboración Open Source en github

Se tiene en cuenta que, si bloqueamos la rama master, todo cambio se debe subir desde una rama diferente y luego pedir el pull Request.

**ISSUES Y MILESTONES(REPORTANDO Y MONITOREANDO ERRORES EFICIENTEMENTE)**

Issues: Es la manera de hacer reportes o cambios en el repositorio.

Milestone: Una forma de agrupas conjuntos de Issues o PullRequest.

Labels: Son etiquetas para nuestros Issues.

**¿Cómo funciona un Pull Request en GitHub?**

Debemos crear un fork, que lo que hace es crear una copia del original y tener nuestra propia versión.

Si yo quiero puedo solicitar el PullRequest a el líder del proyecto al que se le hizo el fork, y si es aprobado eres parte o eres uno de los desarrolladores que aporta a ese proyecto.

**GITHUB PAGES, HOSTING GRATUITO DE ARCHIVOS ESTÁTICOS**

GitHub nos permite publicar nuestros proyectos a través de GitHub Pages.

Los GitHubs pages almacenan código estático, así mismo podemos agregarle nuestro dominio personalizado agregándolo a un archivo CNAME.

En el proveedor de dominio agregamos las ip de GitHub.

**GITHUB DESKTOP**, todo lo que necesitas de Git en una sola herramienta gráfica

Interfaz de escritorio en desktop.github.com para que puedas trabajar gráficamente desde tu PC. Desarrollada en Electron. Lo primero que debes hacer es iniciar sesión, luego clonar el proyecto.

Puedes ver todo el historial de tu proyecto de una manera muy amigable.

Quien hace los cambios, quien los aprueba.

-También puedes navegar en todos tus repositorios.

Puedes hacer cambios.