RESUMEN GIT Y GITHUB DESDE CERO.

GIT = Sistema de control de versiones DISTRIBUIDO.

GIT es un sistema de control de versiones que te ayuda a guardar y manejar diferentes versiones de tus archivos. Es como una máquina del tiempo para tu código, que te permite volver atrás y comparar cambios. Puedes trabajar en diferentes ramas para probar nuevas características sin afectar la versión principal. En resumen, *GIT* te ayuda a mantener tu proyecto organizado y seguro.

Es importante no confundir *GIT* con *GITHUB*:

Git es el sistema de control de versiones en sí mismo, mientras que *GitHub* es una plataforma en línea que utiliza *Git* como base para alojar y colaborar en proyectos de desarrollo de software. *Git* se ejecuta localmente en tu máquina, mientras que *GitHub* es una plataforma en línea que te permite almacenar y compartir repositorios *Git* con otras personas. Fácil, verdad?
Antes de comenzar voy a recordarte que en la página oficial de Git, dispones de un manual y documentación gratuita. Si quieres descargar el libro GIT PRO de manera rápida, doble click en el logo de Git.

LINA DE COMANDOS BÁSICA PARA LA TERMINAL (los comandos siempre en minúscula!)

- LS = Listado de directorios.
- CD = Desplazamiento entre DIRECTORIOS.
- CD .. = Retrocede al directorio anterior.
- PWD = Sirve para ver la ruta en la que te encuentras.
- MKDIR = Crea una carpeta. Tras mkdir debes poner el nombre de la carpeta.
- CLEAR = Para limpiar la consola (o Cntrl + L si no lo quieres borrar pero si limpiar el área de trbajo)
- TOUCH = Crea un fichero.

> Turboconsejo = Cuando crees un directorio pulsa **!$** para entrar a el rápidamente!
> Turboconsejo = Escribe **code .** Para abrir VisualStudioCode directamente ahí!

CONFIGURACIÓN GIT BÁSICA

Abrir terminal:

> **git config –global user.name “NOMBRE”**
>
> **git config –global user.email “EMAIL@GMAIL.COM”**

Sin esta configuración previa no podemos ni trabajar en *GIT*. Es completamente necesaria. Te ayuda a identificarte como autor de los cambios.

A continuación vamos a ir desarrollando los diferentes comandos de GIT con el fin tanto de practicar , como de tenerlos a mano por si se olvida alguno!

GIT INIT

En el directorio raíz de nuestro código lanzaremos el comando *git init para* comenzar con GIT en nuestro proyecto. Se creará una carpeta .git (el punto antes del nombre de un directorio significa que está oculta, lanza *ls -a* para poder verla compañero).
Ahora tenemos un directorio trabajando con un control de versiones!!

Como puedes ver la consola ahora nos llama MASTER y es que eres un autentico crack pero master se refiere a que estás en la rama principal del control de *git* (Según la versión que trabajes te pondrá *master* o *main*).

Si prefieres main en lugar de master tan solo escribe... **git branch -m main**

GIT STATUS

Te indicará que cambios ha habido desde que has iniciado el proyecto. Si no has realizado ninguno te dirá …

On branch main
No commits yet

Además verás los ficheros que hayas creado pero aún no has añadido a *GIT*.

GIT ADD

Con git add estás añadiendo el fichero de guardando para la partida. Al fin y al cabo programar es un juego donde vas a tener que jugar en diferentes niveles y probablemente quieras volver a un nivel anterior. Algunos desarrolladores conocen esta fase como área de stage. Escribe:

git add helloworld.py

O cualquiera que sea el archivo que quieres guardar. Después, si realizas un *git status* de nuevo , verás que se ha guardado y ahora te sale en verde, el verde esperanza por si la lías y necesitas volver. Peeeeeeeeeeroooooooo no está guardado en la raíz principal aún. Podríamos decir que es una etapa intermedia antes de realizar el guardado. Como una manera de seleccionar los archivos para luego realizar los cambios.

Turboconsejo: Usa ‘*git add .’ y subes TODOS los archivos.*

GIT COMMIT

Con este comando guardamos los archivos seleccionados en el paso anterior. Si escribes **git commit** se abrirá el editor de texto *VIM*. Te requiere que realices un comentario para poder guardar definitivamente el archivo. (*VIM* no es como el blog de notas, Debes pulsar la I para poder insertar texto).

Si pasas de *VIM* y quieres ser más productivo es fácil, ingresa el siguiente comando:

git commit -m “Este es tu comentario”

La terminal te dirá que ha habido un cambio y asociará tu archivo a un *hash*.

Un *hash* es una representación numérica única generada a partir de un conjunto de datos mediante un algoritmo de *hash*, utilizado para verificar la integridad y la identidad de los datos.

Git status ahora nos muestra que el árbol de trabajo está limpio.

En resumen, **git add** se utiliza para seleccionar y agregar cambios al área de preparación, mientras que **git commit** se utiliza para guardar los cambios del área de preparación en el historial del repositorio.

GIT LOG

Muestra en la terminal el historial de confirmaciones o *commits*. Nos mostrará un *hash* de cada commit, el autor y la fecha. Es sobre todo útil para ver el historial de tu trabajo. Podemos facilitar la compresión siempre de la salida de *git log* con diferentes comandos:

git log –graph --Nos mostrará una especie de rama con los logs.

git log –graph –pretty=oneline --Una vista sencilla.

git log –graph –decorate –all –oneline

git reflog ‘hash’

Con *git reflog* podremos ver un registro detallado de todos los movimientos de HEAD (head = puntero que indica la posición actual en el historial de *commits* de una rama especifica) y del resto de ramas en el repositorio, incluso aquellas que ya no son visibles.

GIT CHECKOUT

Git checkout es un comando utilizado en Git para cambiar de rama o restaurar archivos en un repositorio. Imagina que has cambiado código y no te compila. Debes volver al último punto de guardado de la partida. Fácil, desde terminal dispara un:

git checkout archivo.py

git checkout ‘hash’ -- Vuelves al punto que selecciones. Ojo con borrar archivos!

git reset -- En este caso, deshaces el último commit.

git reset --hard -- Eliminas las ramas que hayas dejado por encima.

La diferencia entre *git reset* y *git reset –hard* es que el primero mantiene los cambios no confirmados en el área de trabajo mientras que el parámetro *–hard* descarta los cambios no

confirmados y restaura el repositorio al estado del *commit* seleccionado. Recuerda, la mejor manera de aprender es abrir la terminal y experimentar hasta que te aburras!

GIT IGNORE

El archivo *.gitignore* es una buena práctica en Git para evitar que ciertos archivos y directorios se incluyan en el control de versiones, mejorando la organización del repositorio y protegiendo información sensible.

touch .gitignore -- Con ese comando creas el archivo *gitignore*.

Recuerda que al poner el punto (.) delante del nombre del archivo, no aparecerá a simple vista sino como oculto (*ls -a* para poder verlo). Una vez creas este archivo, incluyes en el lo que no quieres que trabaje Git. Eso sí, deberás hacer un commit, es decir, imagina que trabajar desde un mac, el sistema crea el archivo ".DS_Store" que contiene metadatos.

Tan sencillo como escribir en el archivo *.gitignore*: **/.DS_Store
Después *git add* y *git commit* del archivo *.gitignore* y por obra de magia ya git no volverá a mostrar ese archivo con un *git status*. Ese archivo que quieres proteger no volverá a ser mostrado en el status de Git.

GIT DIFF

Este útil comando nos mostrará los cambios que ha habido en un archivo desde el último **commit** hasta el momento en el que te encuentras. De tal manera que con:

git diff archivo.py -- Podrás comprobar TODOS los cambios realizados.

GIT TAG

Usamos los tags para etiquetar un *commit* que es importante. En el desarrollo de aplicaciones , los tags suelen ser las versiones. En git puedes realizar un *tag* para marcar el trabajo de un día en particular o de una semana por ejemplo. Siempre nos va a ser un punto de referencia y ser ordenado y organizado es la regla número 1.

git tag primer_tag

Usando *git log*, podrás comprobar que nos aparece el tag. Si necesitas volver a este punto y reiniciar el trabajo del día o volver a una versión anterior:

git checkout tags/primer_tag --Nuestro Head vuelve al punto que creamos el tag.

GIT BRANCH

Hemos estado trabajando en la misma rama todo el tiempo (MAIN). Pero Git, es un árbol de oportunidades. Imagina que quieres seguir trabajando en el proyecto en una rama alternativa por mil razones diferentes. Para ello crearemos una rama nueva con:

git branch nombre_rama

git branch -d nombre_rama -- Para eliminar la rama una vez acabado el trabajo

Hemos creado un nuevo flujo de trabajo, pero ojo que sigues en la rama principal. Para ello:

git switch nombre_rama

Con ese sencillo comando, cambias de una rama a otra según lo necesites.

git merge main -- Fusionas la rama principal con la rama que has creado.

GIT STASH

Es como un *commit* pero en el área local. Si estas trabajando en un archivo de una rama y debes cambiar a otra rama, sin el *commit* perderás el trabajo. Pero tampoco vas a hacer un *commit* de un código erróneo o incompleto…
En ese caso tenemos este comando que nos guardará nuestro progreso.

git stash

git stash list -- Podrás ver los archivos que tienes 'guardados' pero sin *commit.*

git stash pop -- Con este comando recuperas el archivo que tenías en *stash*!

git stash drop -- Eliminar el *stash* de forma permanente.

GITHUB

Con *git* hemos trabajado de manera local. Con *github* vamos a trabajar con un servidor remoto. Es una plataforma de desarrollo colaborativo. Los repositorios se pueden clonar modificar y fusionar para facilitar el trabajo en equipo. Permite a los desarrolladores compartir y descubrir proyectos de código abierto. *Github* es una plataforma que te ofrece una documentación brutal. Aqui vamos a trasladarte conceptos básicos pero recuerda que en la web lo tienes todo!

Markdown → el markdown de GitHub es una forma sencilla y efectiva de dar formato y estructurar el contenido en la plataforma de desarrollo colaborativo GitHub, facilitando la comunicación y colaboración entre los usuarios.

GIT REMOTE

Git está pensado para trabajar en local y remoto. (Voy a dar por hecho que ya has configurado tu autentificación *ssh*, en la propia página de *github* encontrarás todos los comandos en orden).

git remote add origin … -- Emparejas tu código con el repositorio.

git push -u origin main

GIT FETCH Y GIT PULL

Git fetch descarga el historial sin los cambios, pero *git pull* descarga el historial y los cambios.

git fetch

git pull origin main

GIT CLONE

Con este comando podemos descargar el código que necesitamos. En la propia *github* tenemos la opción de de descargarlo en zip.

git clone –url

PULL REQUEST

Un pull request es una función que se encuentra en sistemas de control de versiones como *Git*. Permite proponer cambios en un repositorio y solicitar que los propietarios o colaboradores del proyecto revisen y fusionen esos cambios en una rama principal. El uso de pull requests facilita el proceso de colaboración y revisión de código en proyectos de desarrollo de software, ya que proporciona un medio estructurado para discutir, revisar y fusionar cambios de manera controlada.