# Curso de LinkedIn: Como hacer que tu perfil trabaje por ti

# Clase 1: Que es git

# Para que sirve git:

# 

# Control de versión:

Git es un sistema de control de versiones.

En esta aula aprendimos:

* Qué son (y para qué sirven) los sistemas de control de versiones y cómo pueden ayudar a nuestro flujo de desarrollo
* Nos ayudan a mantener un historial de cambios;
* Nos ayudan a tener control sobre cada cambio en el código;
* Nos ayudan a que un cambio de una persona no influya en el cambio realizado por otra;
* Etc.
* Qué es Git y cómo instalarlo
* Que con el comando git init pudimos crear un repositorio Git;
* Cómo analizar el estado de nuestro repositorio usando el comando git status.

# Clase 2: Iniciando los trabajos

# Guardando configuraciones:

Git status = ver el estado del repositorio

Git add . = trackear todos los archivos de la carpeta

Git rm –cached <file> = dejar de trackear archive

Git commit -m “mensaje” = crear mensaje de los cambios

Al ejecutar el comando git status, recibimos información que puede no ser tan clara, especialmente cuando nos encontramos con términos como HEAD, working tree, index, etc.

Solo para aclarar un poco, ya que entenderemos mejor cómo funciona Git durante el curso, aquí hay algunas definiciones interesantes:

* HEAD: Estado actual de nuestro código, es decir, donde nos colocó Git
* Working tree: Lugar donde los archivos realmente están siendo almacenados
* index: Lugar donde Git almacena lo que será commiteado, es decir, la ubicación entre el working tree y el repositorio de Git en sí.

# Viendo el histórico:

Git log = hash (id del commit)

Git config –local = solo para este proyecto

Git config –global = todos los proyectos en la computadora

Git config user.name = tu nombre

Git config user.email = tu email

Git log –oneline = ver commits en una línea

Git log -p = ver cambios, Para salir :q

# Cuando hacer un commit:

Deberíamos generar un commit siempre que nuestra base de código esté en un estado que nos gustaría recordar. Nunca deberíamos tener commits de código que no funcionen, pero tampoco es interesante dejar el commit solo al final de una función.

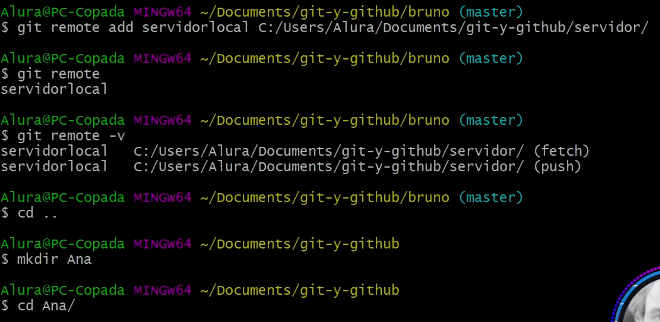
En esta aula aprendimos:

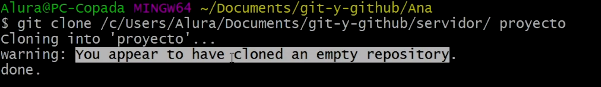
* Que un commit es la forma de guardar un estado o versión de nuestro código;
* Cómo agregar archivos para commitear con git add;
* Cómo commitear archivos usando el comando git commit;
* Cómo verificar el historial de commits, a través de git log y algunas de sus opciones:
* git log --oneline
* git log -p
* git log --pretty="parámetros de formato"
* Cómo hacer que Git no monitoree archivos, a través de .gitignore
* Que no debemos realizar commits, es decir, guardar un estado, de nuestra aplicación que no esté funcionando.

# Clase 3: Compartiendo el trabajo

# Repositorio remoto:







Git init –bare (Repositorio local)

# Sincronizando datos:

# 

# GitHub:

Origin es el repositorio por defecto.

# 

En esta aula aprendimos:

* Lo que son los repositorios remotos
* Cómo crear un repositorio de Git sin una copia de los archivos (con --bare) para usar como servidor;
* Cómo agregar links a repositorios remotos, con el comando git remote add;
* Cómo descargar un repositorio por primera vez clonándolo con el comando git clone;
* Cómo enviar nuestros cambios a un repositorio remoto, con git push;
* Cómo actualizar nuestro repositorio con los datos en el repositorio remoto, usando git pull;
* Qué es **GitHub** y para qué sirve;
* Cómo crear un repositorio en **GitHub**;
* Cómo agregar un repositorio de **GitHub** como repositorio remoto.

# Clase 4: Trabajando en equipo

# Branches: (Ramas)

Git Branch nombre = crear rama

Git checkout nombre = cambiar de rama

Git checkout -b lista = crear y cambiar rama

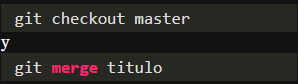
Las branches ("ramas") se utilizan para desarrollar funcionalidades aisladas entre sí. La branch master es la branch "predeterminada" cuando creas un repositorio.

Es interesante separar el desarrollo de funcionalidades en diferentes branches, para que los cambios en el código de una no influyan en el funcionamiento de otra.

# Uniendo el trabajo:

Git merge nombre = Unir las dos ramas (en la rama main) salimos con :x

Cómo podemos hacer el merge de la *Branch* titulo hacia la *Branch* master:



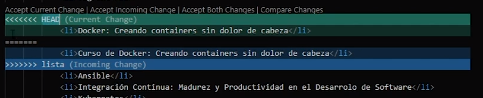
# Actualizando la Branch:

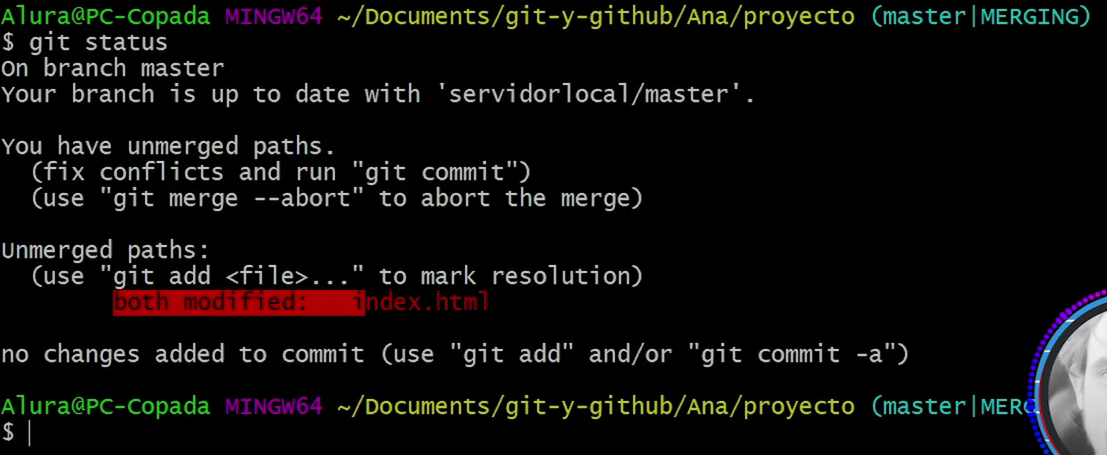
Git rebase titulo = llevamos los commits de titulo a master.

El merge junta los trabajos y genera un merge commit. El rebase aplica los *commits* de otra Branch en la *Branch* actual.

# Resolviendo conflictos:

# 





Vimos lo simple que es resolver los conflictos identificados por Git al intentar hacer merge.

Ahora, genera un conflicto y, en lugar de usar merge, usa rebase para actualizar el master:

* Vé a la *branch* titulo
* *Commitea* algo
* Vé a la *branch* master, *commitea* un cambio en la misma línea
* Ejecuta git rebase titulo

Mira la salida de Git y usa la información que te da; después de corregir el conflicto, continúa con el rebase.

En esta aula aprendimos:

* Que una *branch* (o rama) es una línea de *commits* separada, y que se puede usar para desarrollar funcionalidades independientes;
* Que con *branches* separadas, podemos evitar que el código de una funcionalidad interfiera con otra;
* Cómo traer el trabajo realizado en una *branch* a otra *branch*, como la master, usando el comando git merge;
* Que git merge genera un nuevo *commit*, informando que hubo una mezcla entre dos *branches*;
* Cómo traer *commits* de una *branch* a otra con git rebase
* Que git rebase no genera un *commit* de merge, lo que simplifica nuestro *log*;
* Cómo presenta Git los conflictos;
* Cómo resolver conflictos y conservar solo los cambios deseados con Git.

# Clase 5: Manipulando versiones

# Ctrl + z en Git:

# Clase 6: Generando entregas