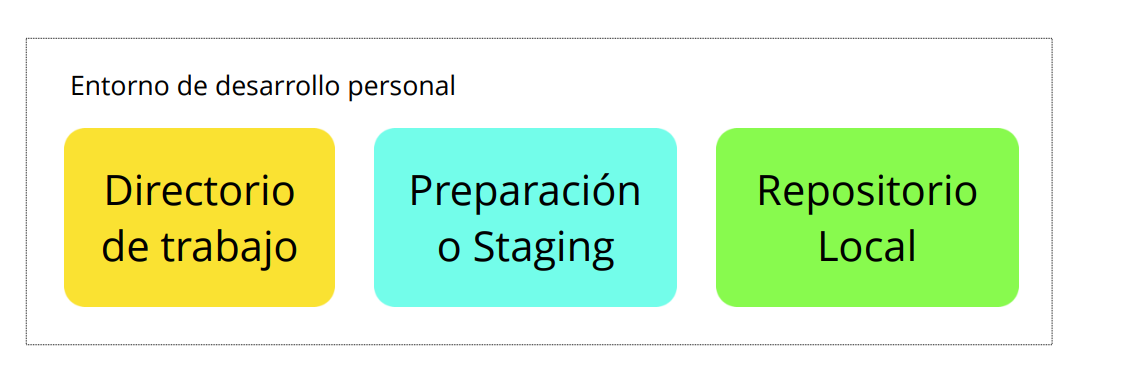
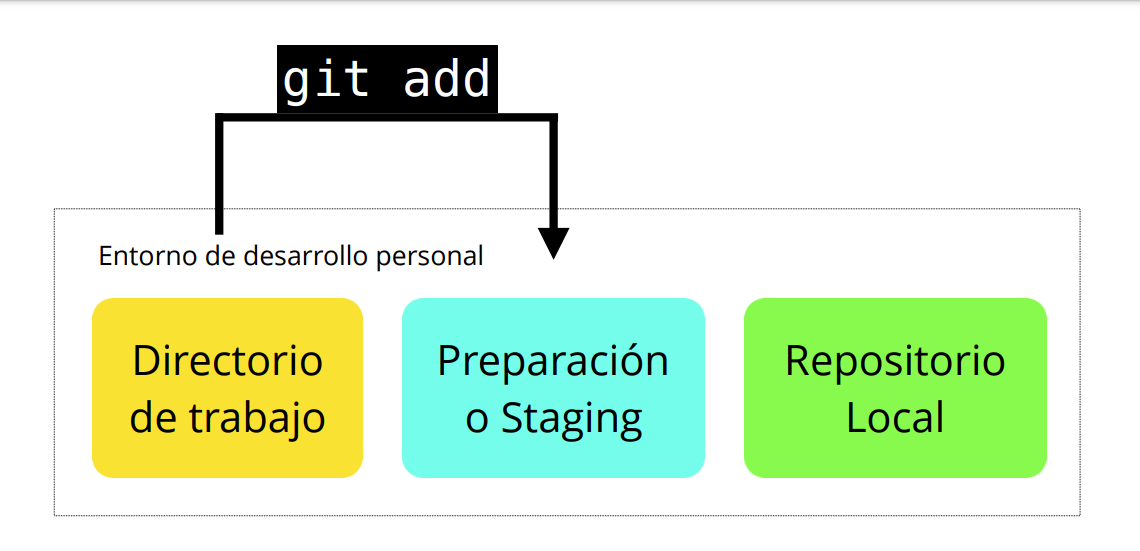
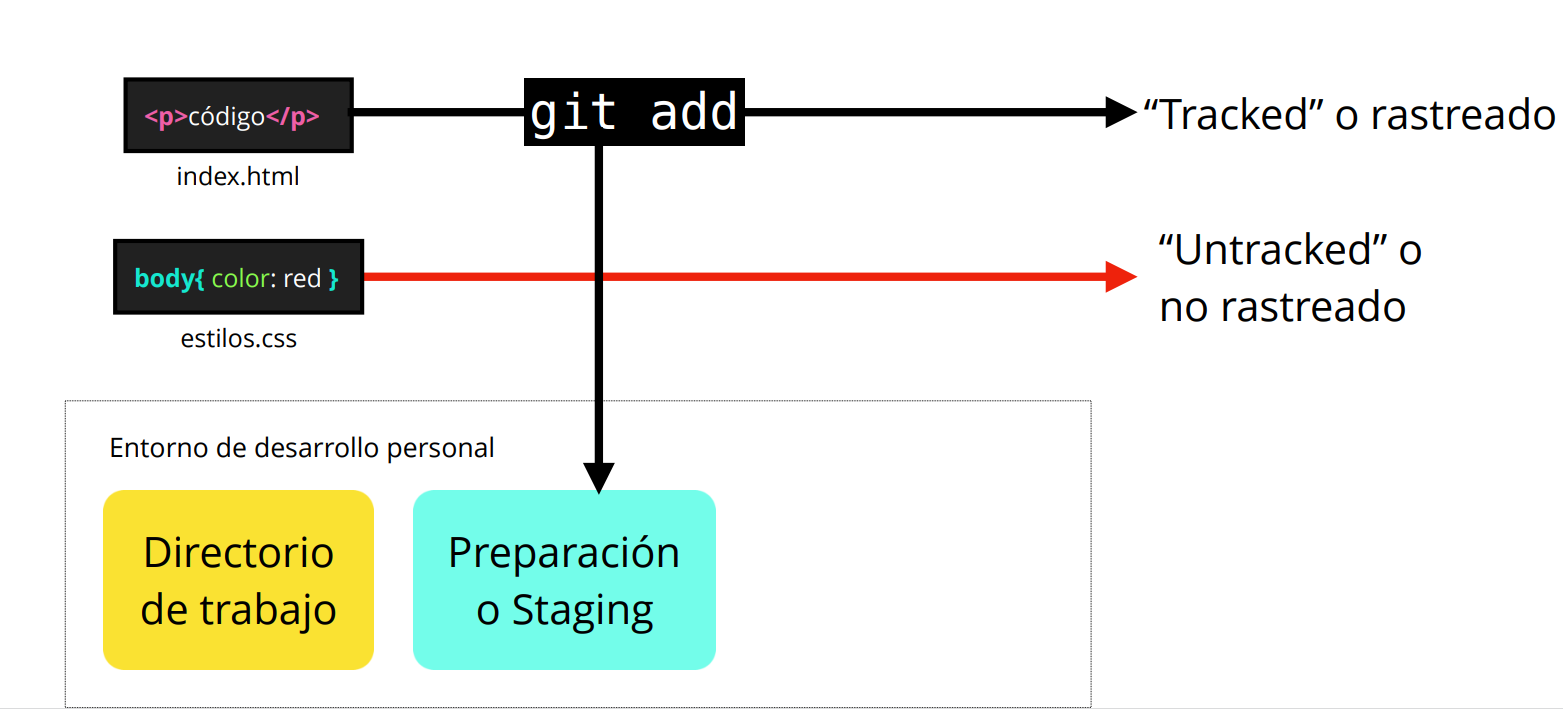
# Comandos de Git

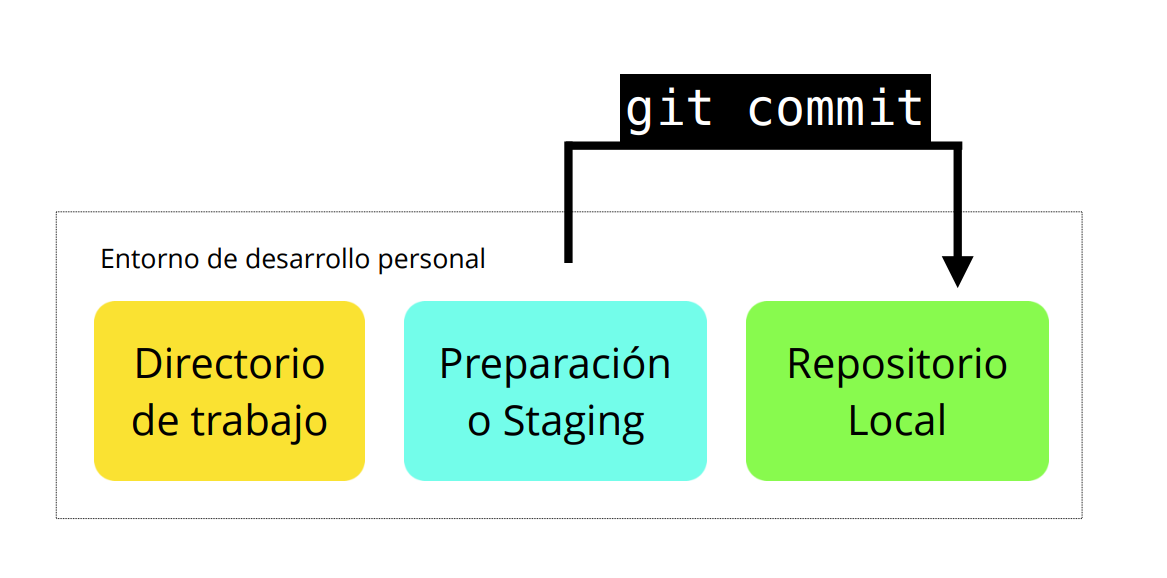
## Introducción

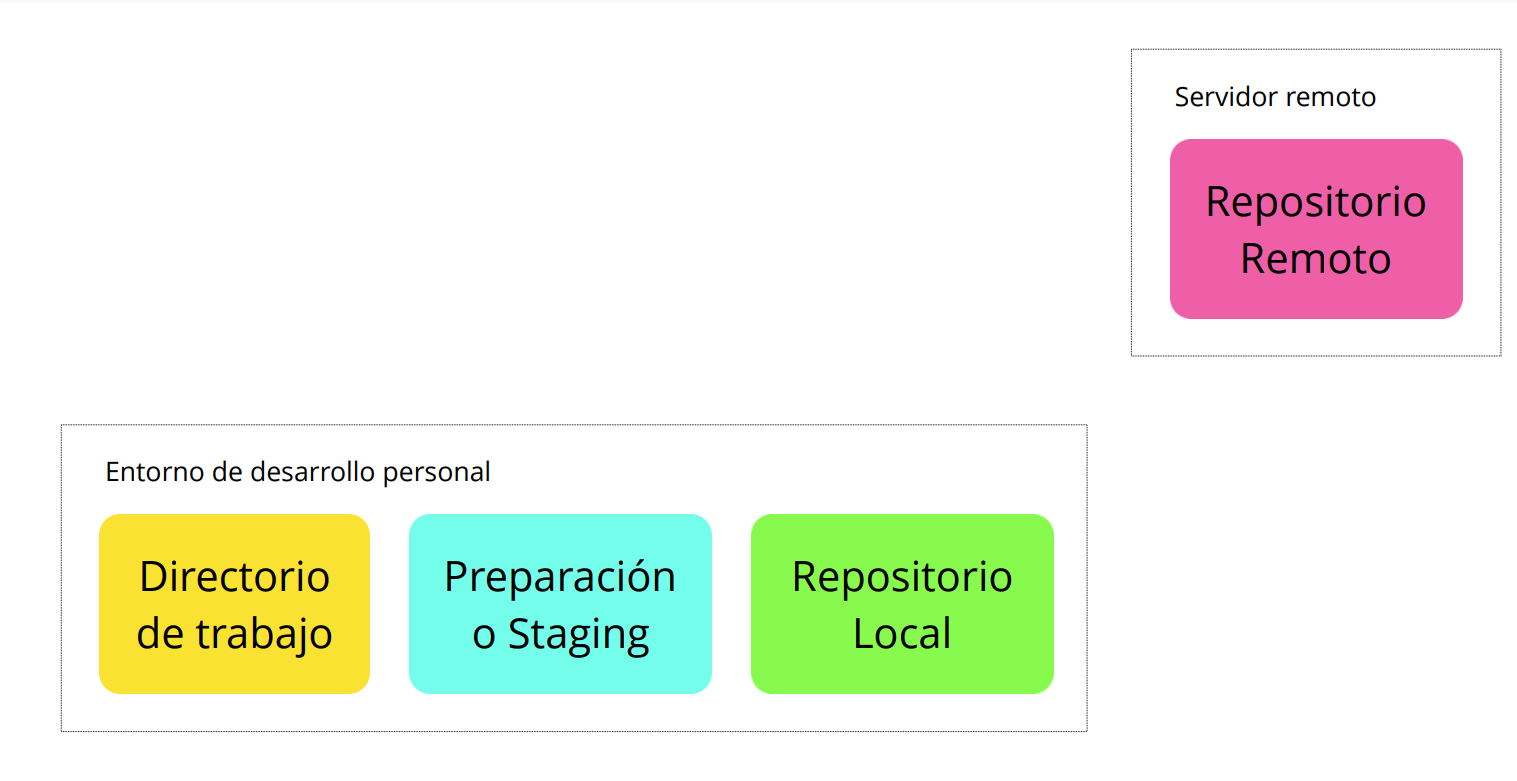
Git es un software de control de versiones diseñado por Linus Torvalds, pensando en la eficiencia y la confiabilidad del mantenimiento de versiones de aplicaciones cuando estas tienen un gran número de archivos de código fuente. En su lugar GitHub es una forma para alojar proyectos utilizando el sistema de control de versiones Git. GitHub sería la red social de código para los programadores, tu propio curriculum vitae.

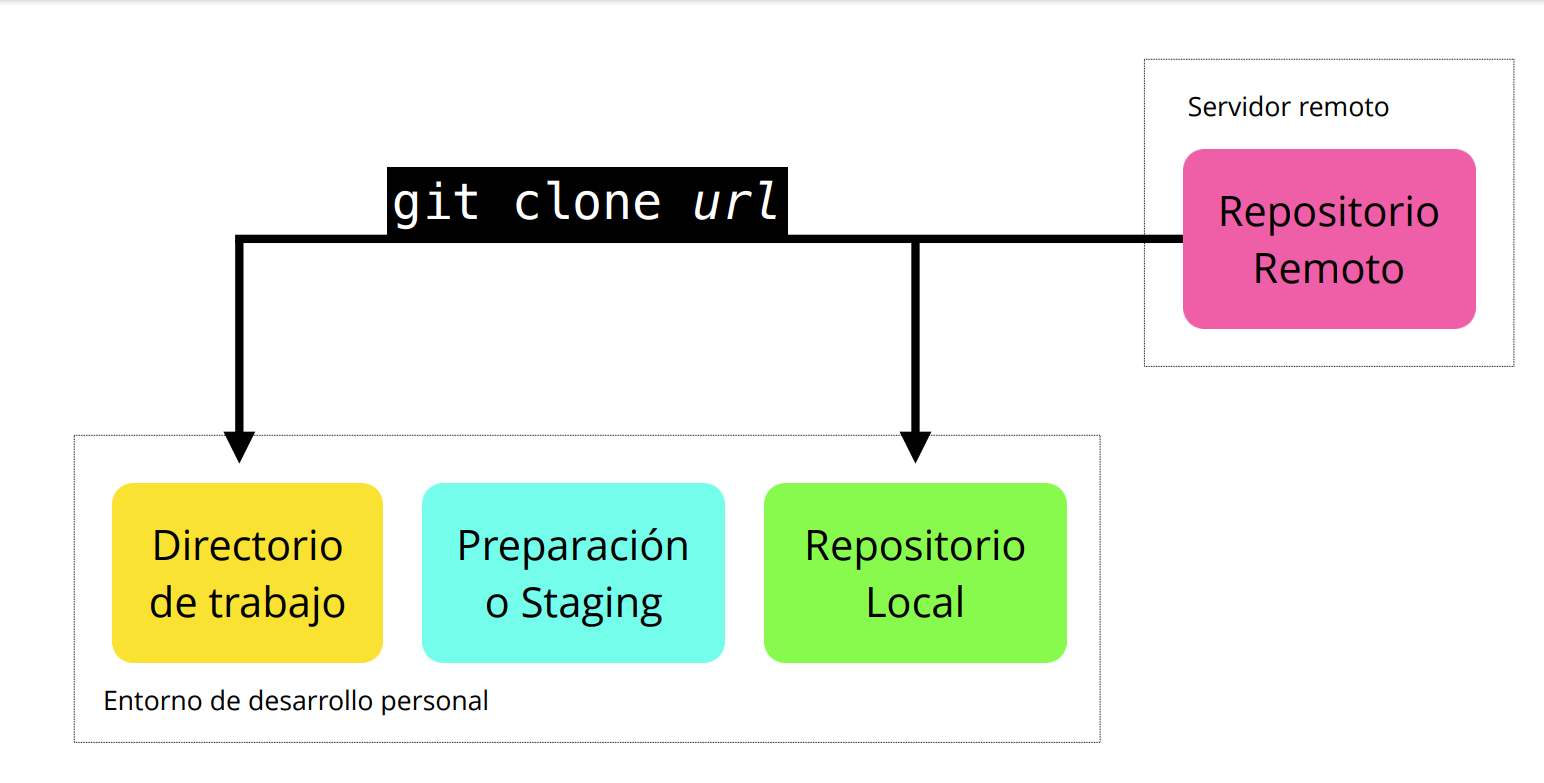


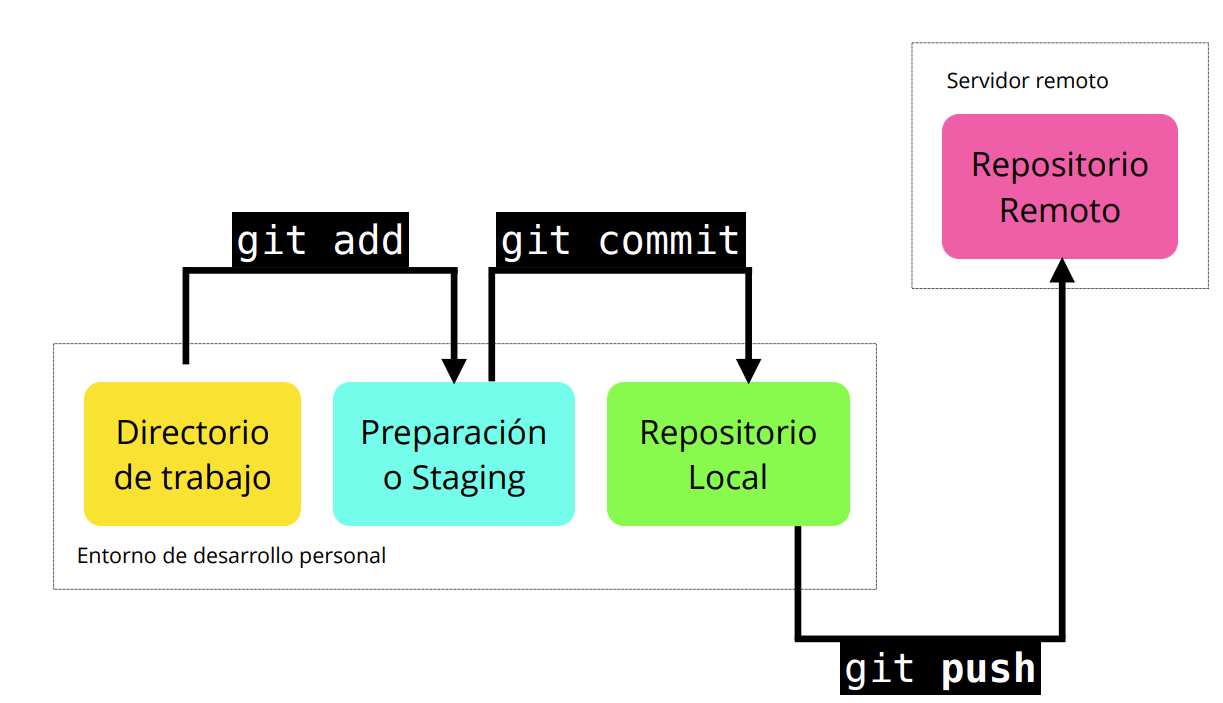


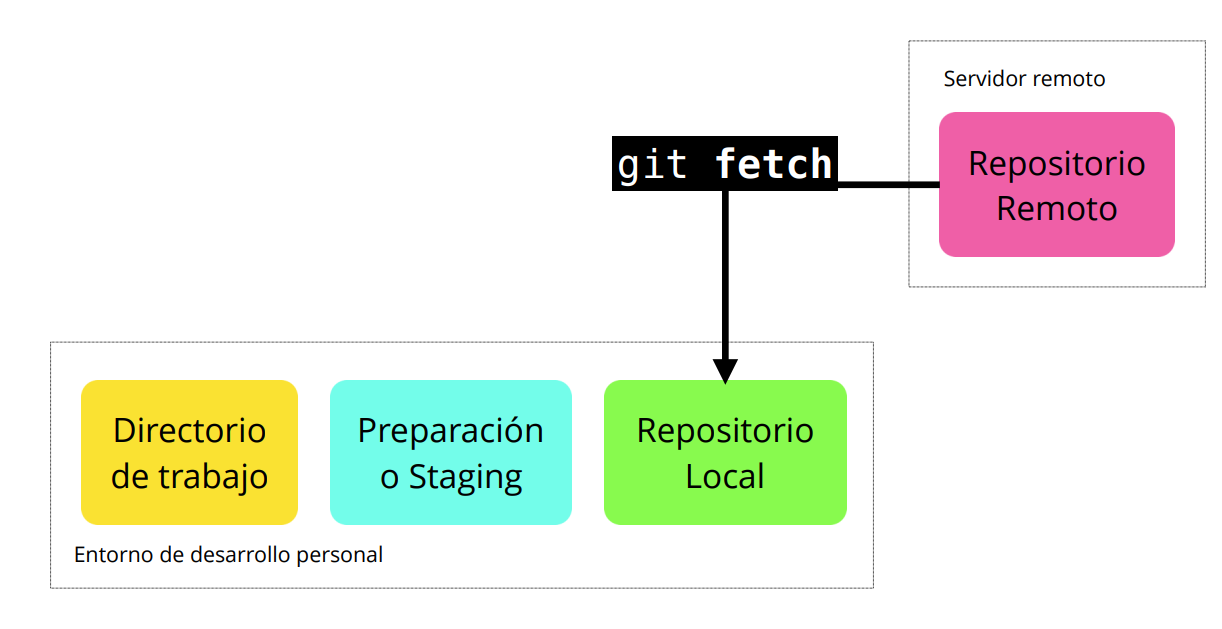


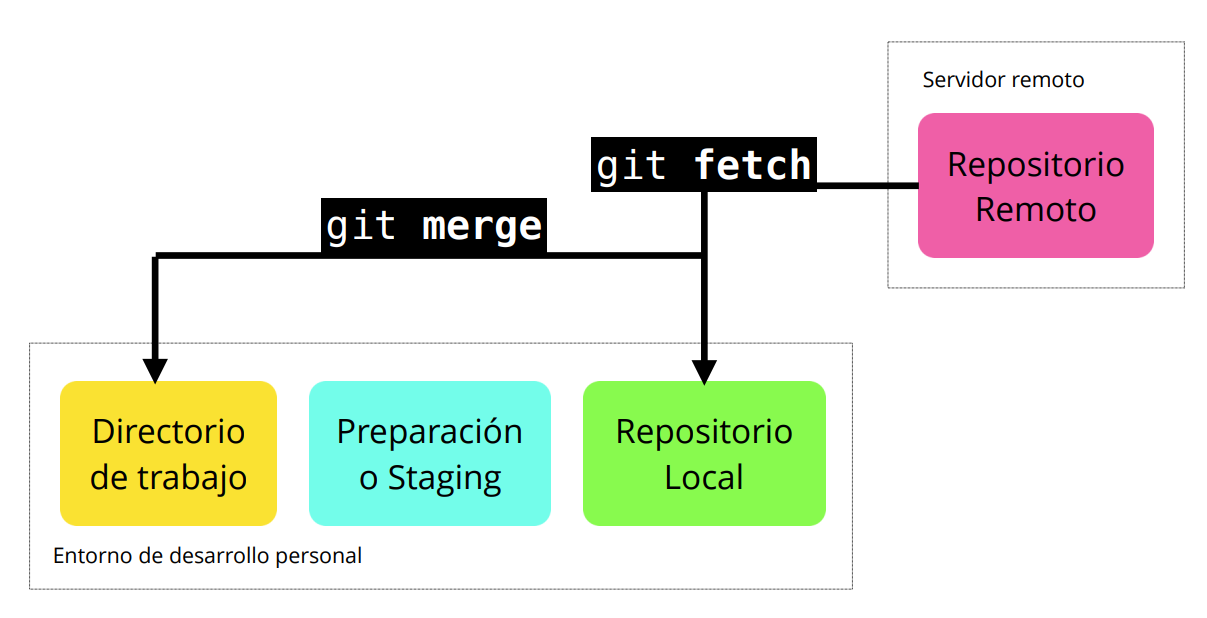


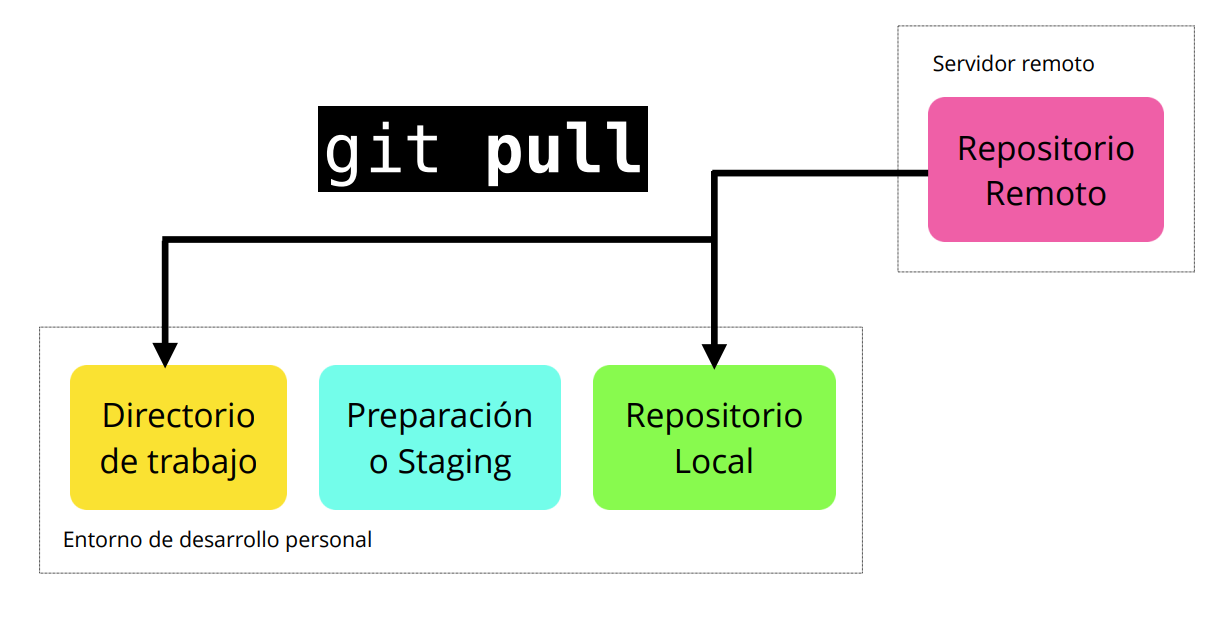


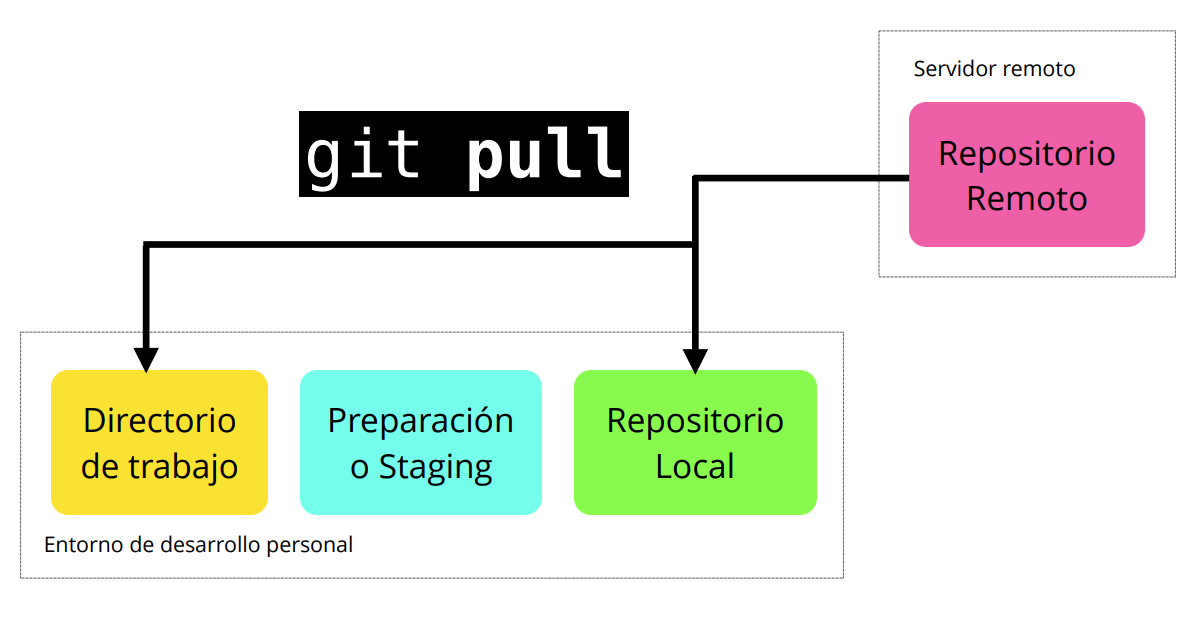


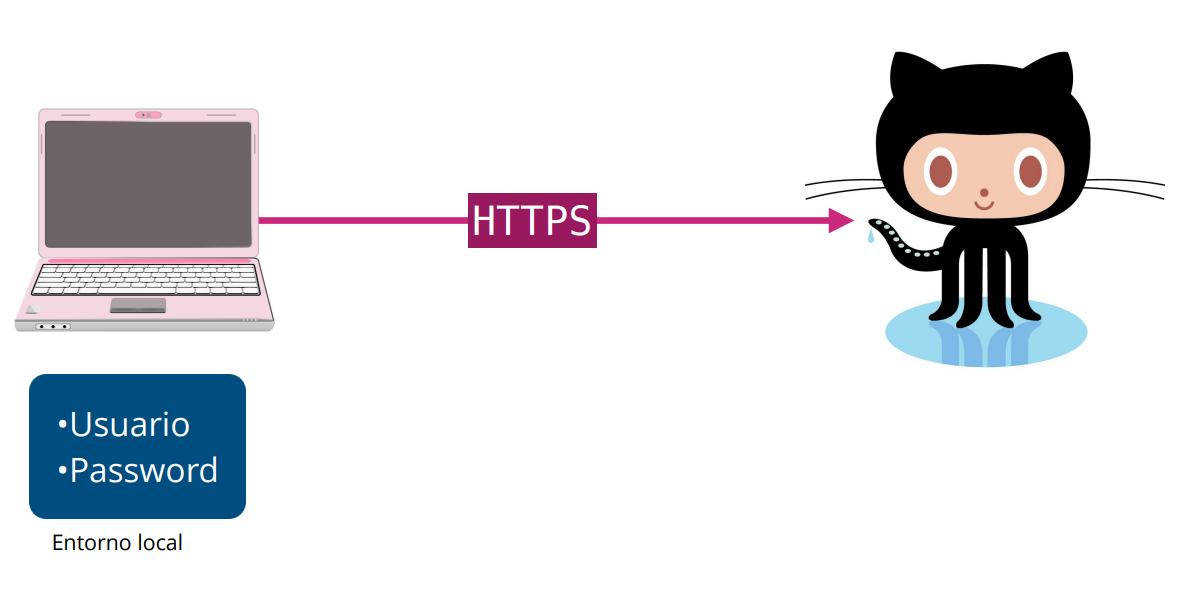


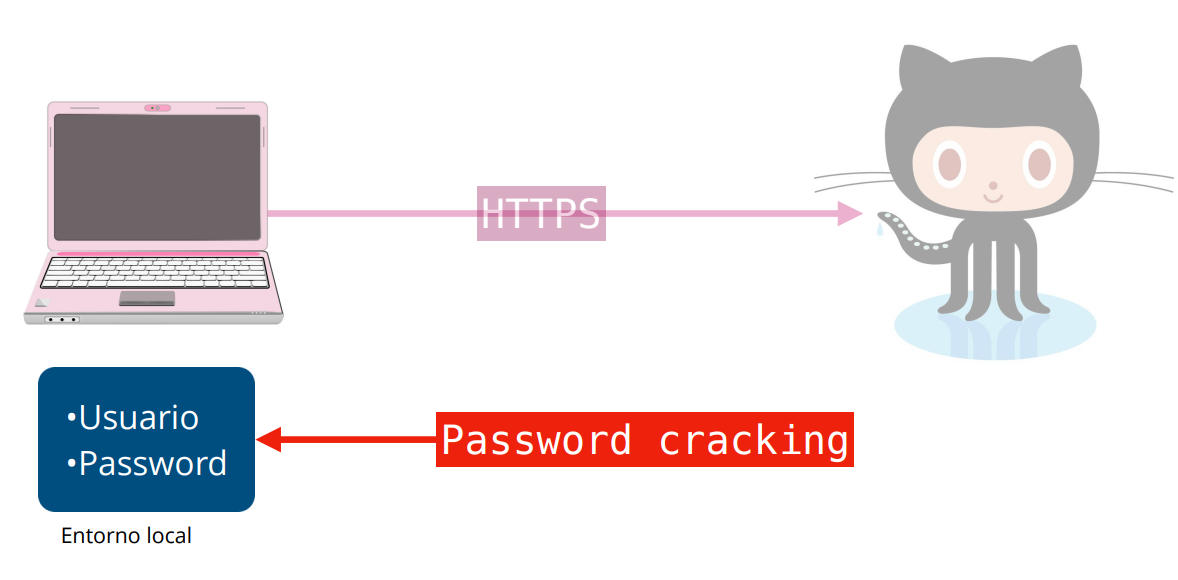




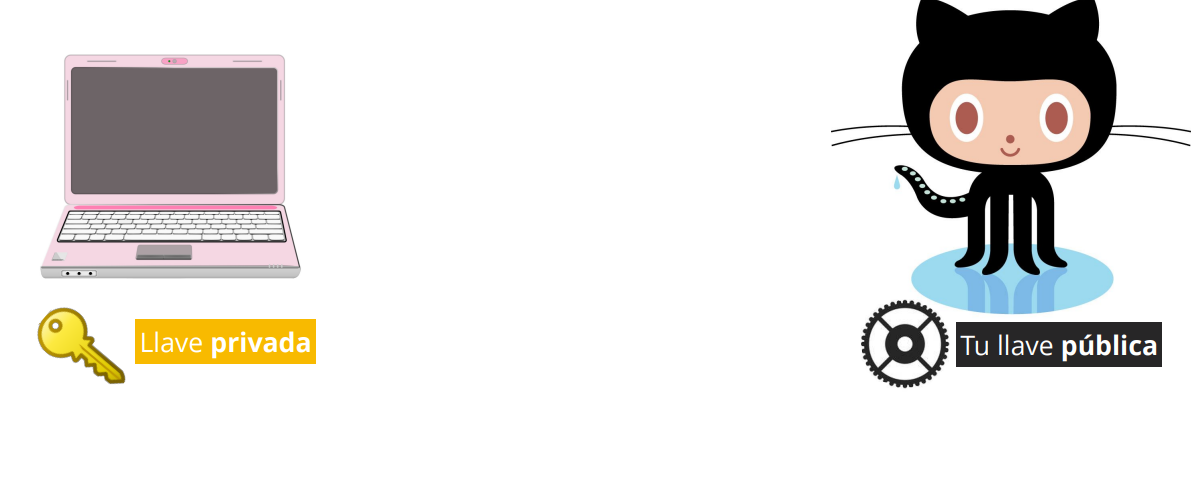


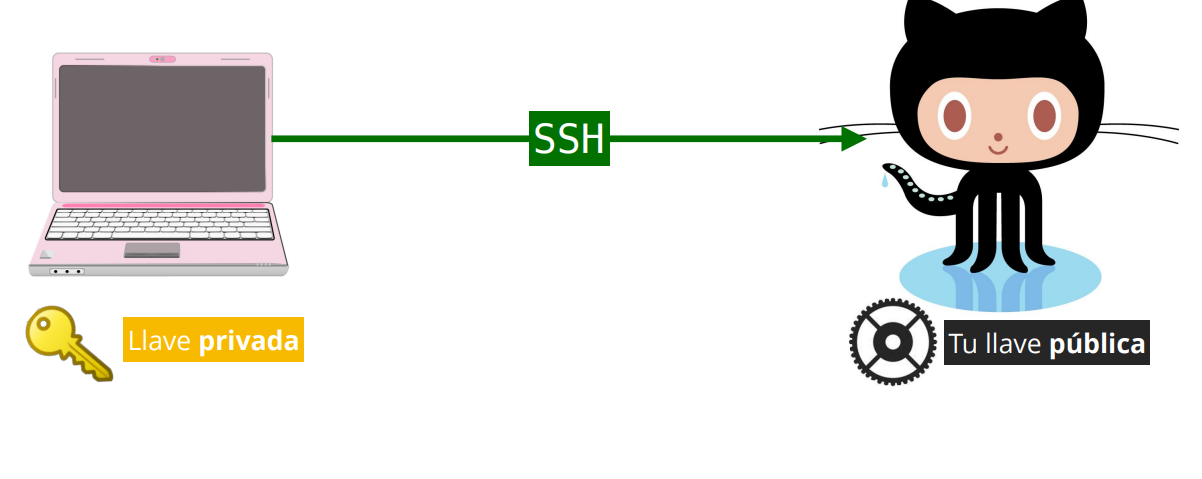


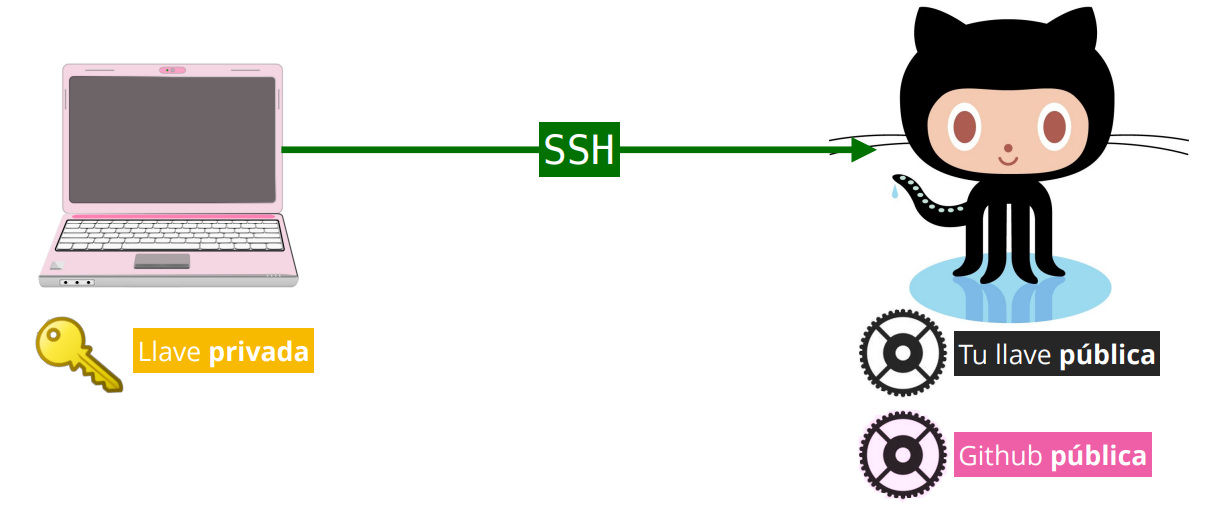


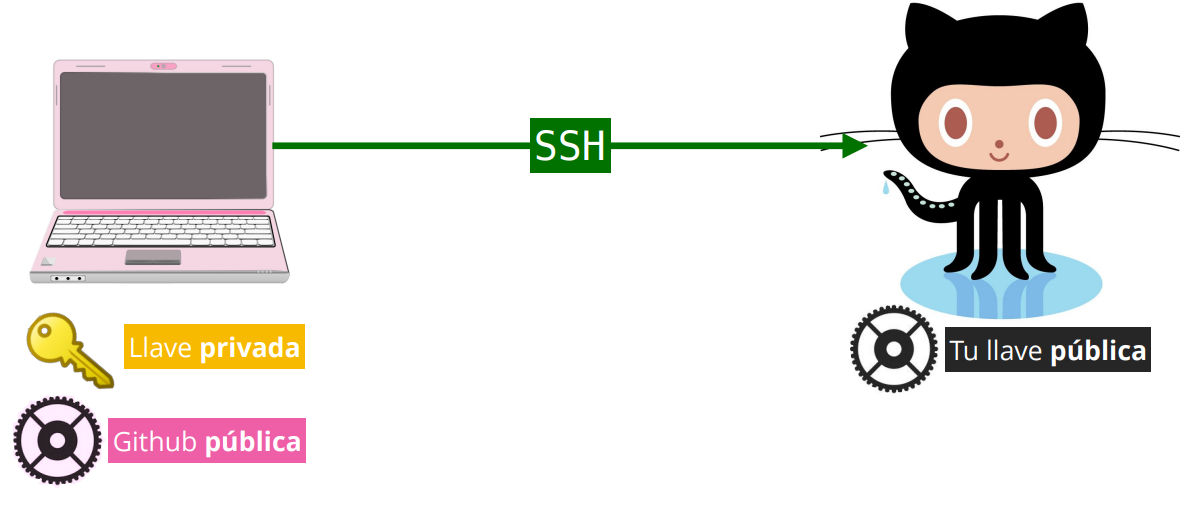


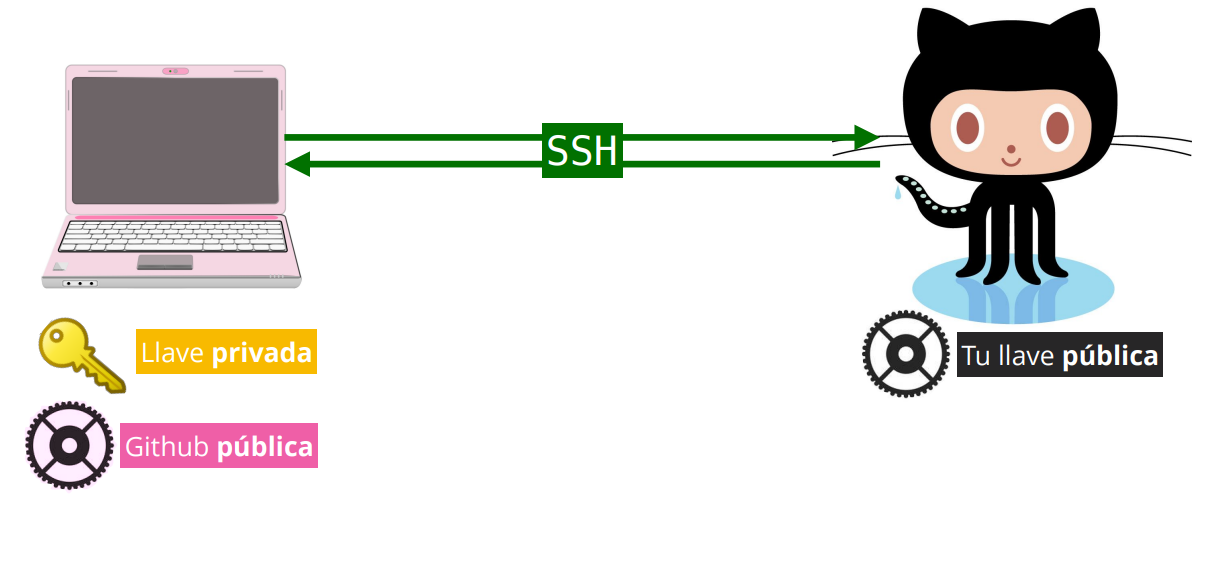












### Inicio de un repositorio (init,add, commit show, log, push, pull)

Un sistema de control de versiones como Git nos ayuda a guardar el historial de cambios y crecimiento de los archivos de nuestro proyecto.

En realidad, los cambios y diferencias entre las versiones de nuestros proyectos pueden tener similitudes, algunas veces los cambios pueden ser solo una palabra o una parte específica de un archivo específico. Git está optimizado para guardar todos estos cambios de forma atómica e incremental, o sea, aplicando cambios sobre los últimos cambios, estos sobre los cambios anteriores y así hasta el inicio de nuestro proyecto.

El comando para iniciar nuestro repositorio, o sea, indicarle a Git que queremos usar su sistema de control de versiones en nuestro proyecto, es

git init.

El comando para que nuestro repositorio sepa de la existencia de un archivo o sus últimos cambios es git add. Este comando no almacena las actualizaciones de forma definitiva, solo las guarda en algo que conocemos como “Staging Area” (no te preocupes, lo entenderemos más adelante).

El comando para almacenar definitivamente todos los cambios que por ahora viven en el staging area es git commit. También podemos guardar un mensaje para recordar muy bien qué cambios hicimos en este commit con el argumento

git commit -m "Mensaje del commit"

Por último, si queremos mandar nuestros commits a un servidor remoto, un lugar donde todos podamos conectar nuestros proyectos, usamos el comando git push.

Git init : inicia el repositorio en git

Git add nombreArchivo : agrega el archivo en el stage para prepararlo para el commit

Git add . : agrega a todo lo archivos dentro del stage

Gir commit –m “mensaje”

Git show muestra todos los cambios que se han realizado

Git log : miestra la historia de los cambios

Git push: enviar todos los cambios al repositorio web

Git pull traer los cambios del repositorio web

### Introducción a la terminal y línea de comandos

Diferencias entre la estructura de archivos de Windows, Mac o Linux.

La ruta principal en Windows es C:\, en UNIX es solo /.

Windows no hace diferencia entre mayúsculas y minúsculas pero UNIX sí.

Recuerda que GitBash usa la ruta /c para dirigirse a C:\ (o /d para dirigirse a D:\) en Windows. Por lo tanto, la ruta del usuario con el que estás trabajando es /c/Users/Nombre de tu usuario

Comandos básicos en la terminal:

pwd: Nos muestra la ruta de carpetas en la que te encuentras ahora mismo.

mkdir: Nos permite crear carpetas (por ejemplo, mkdir Carpeta-Importante).

touch: Nos permite crear archivos (por ejemplo, touch archivo.txt).

rm: Nos permite borrar un archivo o carpeta (por ejemplo, rm archivo.txt). Mucho cuidado con este comando, puedes borrar todo tu disco duro.

cat: Ver el contenido de un archivo (por ejemplo, cat nombre-archivo.txt).

ls: Nos permite cambiar ver los archivos de la carpeta donde estamos ahora mismo. Podemos usar uno o más argumentos para ver más información sobre estos archivos (los argumentos pueden ser -- + el nombre del argumento o - + una sola letra o shortcut por cada argumento).

- ls -a: Mostrar todos los archivos, incluso los ocultos.

- ls -l: Ver todos los archivos como una lista.

cd: Nos permite navegar entre carpetas.

- cd /: Ir a la ruta principal:

- cd o cd ~: Ir a la ruta de tu usuario

- cd carpeta/subcarpeta: Navegar a una ruta dentro de la carpeta donde estamos ahora mismo.

- cd .. (cd + dos puntos): Regresar una carpeta hacia atrás.

- Si quieres referirte al directorio en el que te encuentras ahora mismo puedes usar cd . (cd + un punto).

history: Ver los últimos comandos que ejecutamos y un número especial con el que podemos repetir su ejecución.

! + número: Ejecutar algún comando con el número que nos muestra el comando history (por ejemplo, !72).

clear: Para limpiar la terminal. También podemos usar los atajos de teclado Ctrl + L o Command + L.

Todos estos comandos tiene una función de autocompletado, o sea, puedes escribir la primera parte y presionar la tecla Tab para que la terminal nos muestre todas las posibles carpetas o comandos que podemos ejecutar. Si presionas la tecla Arriba puedes ver el último comando que ejecutamos.

Recuerda que podemos descubrir todos los argumentos de un comando con el argumento --help (por ejemplo, cat --help).

### Configuración del ambiente de trabajo antes de hacer un commit

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 sola vez los siguientes comandos con tu información:

Configaraciones

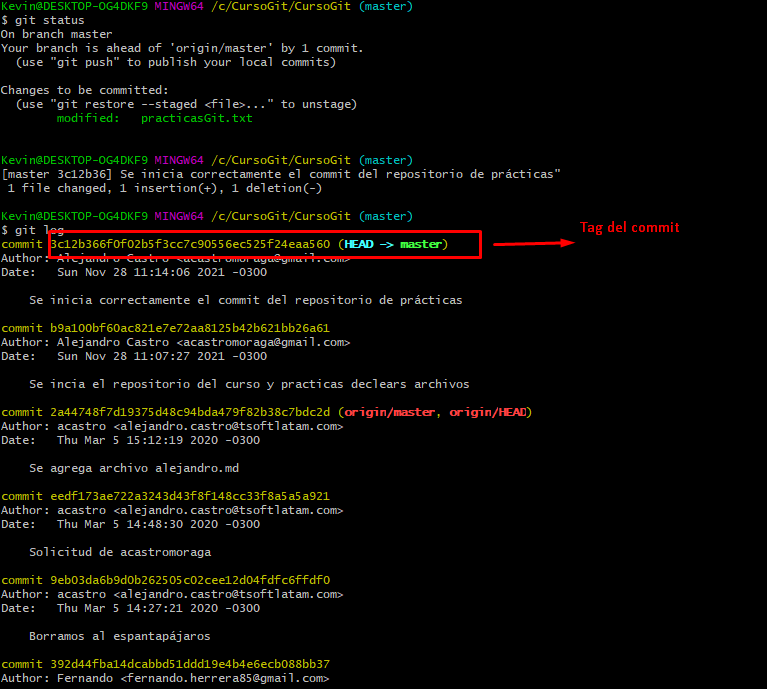
git config --global user.email [tu@email.com](mailto:tu@email.com)

git config --global user.name "Tu Nombre"

git config –list: muestra todas las configuraciones realizadas

git config - - list - - show-origin: muestra dónde se encuentran guardadas estas configuraciones

un solo guión se usa para las letras dos guiones palabras

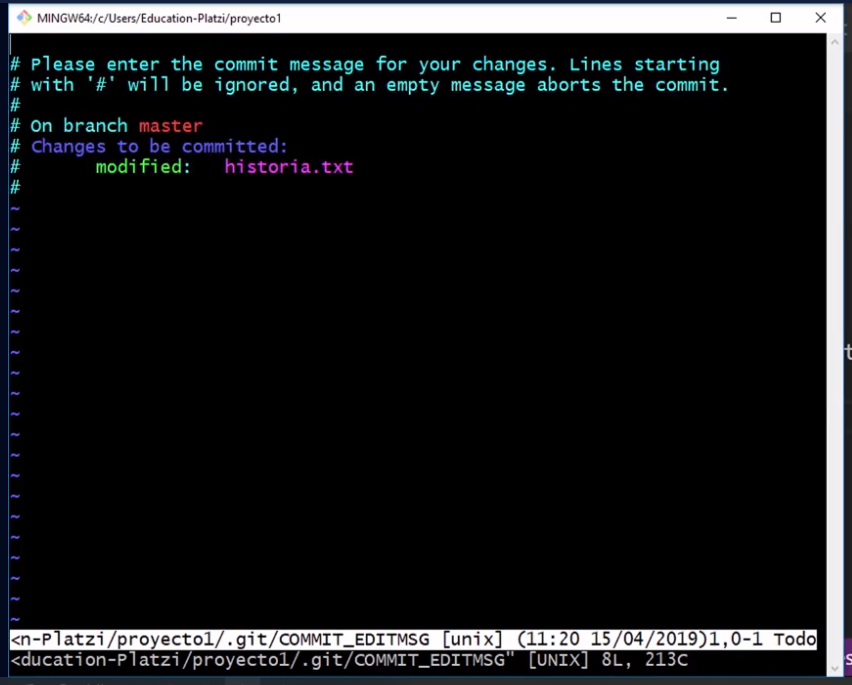


### Analizar cambios en los archivos de tu proyecto con Git, consola de commit

El comando git show nos muestra los cambios que han existido sobre un archivo y es muy útil para detectar cuándo se produjeron ciertos cambios, qué se rompió y cómo lo podemos solucionar. Pero podemos ser más detallados.

Si queremos ver la diferencia entre una versión y otra, no necesariamente todos los cambios desde la creación del archivo, podemos usar el comando

git diff commitA (tag commit nueva) commitB(tag commit antigua). S epueden cambiar



Editor de texto dentro de la linea d ecomando vim

#son lo scomentarios

Apretar “i” (insert) para ingresar texto

Para guar y salir: scc -> shift -> zz

### ¿Qué es el staging y los repositorios? Ciclo básico de trabajo en Git

Para iniciar un repositorio, o sea, activar el sistema de control de versiones de Git en tu proyecto, solo debes ejecutar el comando git init.

Este comando se encargará de dos cosas: primero, crear una carpeta .git, donde se guardará toda la base de datos con cambios atómicos de nuestro proyecto; y segundo, crear un área que conocemos como Staging, que guardará temporalmente nuestros archivos (cuando ejecutemos un comando especial para eso) y nos permitirá, más adelante, guardar estos cambios en el repositorio (también con un comando especial).

Ciclo 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 con repositorios remotos 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 porque 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 porque falta ejecutar el comando git commit.

Archivos Unstaged: entiéndelos como archivos “Tracked 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 está desactualizado, sus últimas versiones solo están guardadas en el disco duro.

Archivos Untracked: son archivos que NO viven dentro de Git, solo en el disco duro. Nunca han sido afectados por git add, así que Git no tiene registros 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 add), 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).

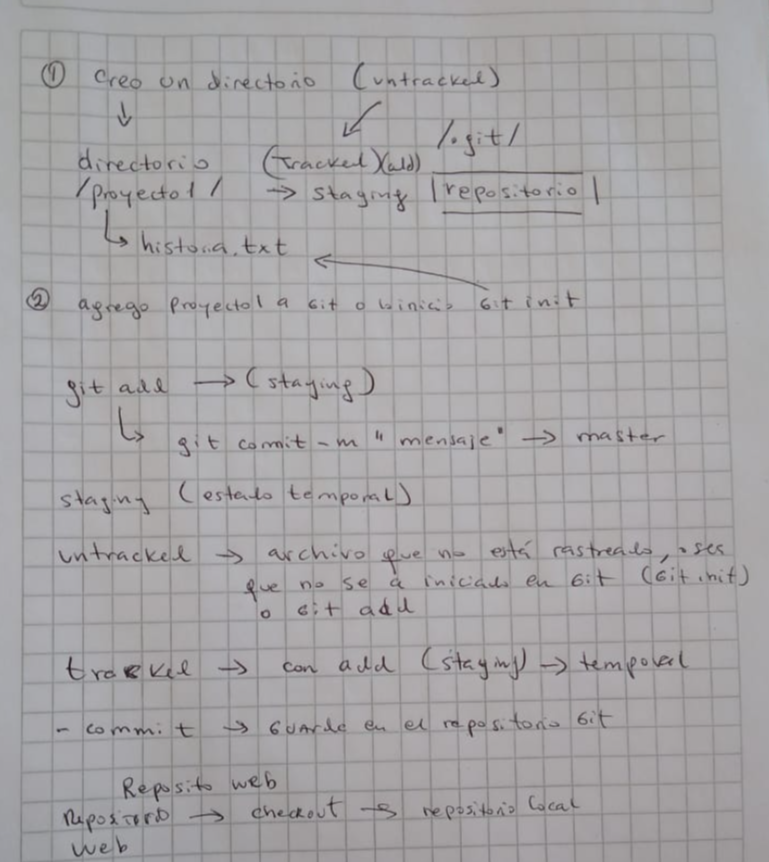
git reset HEAD: nos ayuda a sacar archivos del estado Staged para devolverlos a su estado anterior. Si los archivos venían de Unstaged, vuelven allí. Y lo mismo se venían de Untracked.

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 pedirá que dejemos un mensaje para recordar los cambios que hicimos y podemos usar el argumento -m para escribirlo (git commit -m "mensaje").

git rm: este comando necesita alguno de los siguientes 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).



### Volver en el tiempo en nuestro repositorio utilizando reset y checkout

vo específico o incluso del proyecto entero. Esta también es la forma de crear ramas y movernos entre ellas.

También hay una forma de hacerlo un poco más “ruda”: usando el comando git reset. En este caso, no solo “volvemos en el tiempo”, sino que borramos los cambios que hicimos después de este commit.

Hay dos formas de usar git reset: con el argumento --hard, borrando toda la información que tengamos en el área de staging (y perdiendo todo para siempre). O, un poco más seguro, con el argumento --soft, que mantiene allí los archivos del área de staging para que podamos aplicar nuestros últimos cambios pero desde un commit anterior.

Git diff antes del add, se verán los cambios realizados en los archivos

Git log –stat se verán los cambios específicos en los archivos

Git checkout tagcommit nombreArchivo

Si se hace un git status se puede ver que el archivo tiene modificaciones, por lo que se puede volver a guardar esos cambios, pero si quiero volver a la versión que tenía y que se encuentra en el master

Volver a la última versión

Git checkout master nombrearchivo

Git reset y git rm son comandos con utilidades muy diferentes, pero aún así se confunden muy fácilmente.

git rm

Este comando nos ayuda a eliminar archivos de Git sin eliminar su historial del sistema de versiones. Esto quiere decir que si necesitamos recuperar el archivo solo debemos “viajar en el tiempo” y recuperar el último commit antes de borrar el archivo en cuestión.

Recuerda que git rm no puede usarse así nomás. Debemos usar uno de los flags para indicarle a Git cómo eliminar los archivos que ya no necesitamos en la última versión del proyecto:

git rm --cached: Elimina los archivos de nuestro repositorio local y del área de staging, pero los mantiene en nuestro disco duro. Básicamente le dice a Git que deje de trackear el historial de cambios de estos archivos, por lo que pasaran a un estado untracked.

git rm --force: Elimina los archivos de Git y del disco duro. Git siempre guarda todo, por lo que podemos acceder al registro de la existencia de los archivos, de modo que podremos recuperarlos si es necesario (pero debemos usar comandos más avanzados).

git reset

Este comando nos ayuda a volver en el tiempo. Pero no como git checkout que nos deja ir, mirar, pasear y volver. Con git reset volvemos al pasado sin la posibilidad de volver al futuro. Borramos la historia y la debemos sobreescribir. No hay vuelta atrás.

Este comando es muy peligroso y debemos usarlo solo en caso de emergencia. Recuerda que debemos usar alguna de estas dos opciones:

Hay dos formas de usar git reset: con el argumento --hard, borrando toda la información que tengamos en el área de staging (y perdiendo todo para siempre). O, un poco más seguro, con el argumento --soft, que mantiene allí los archivos del área de staging para que podamos aplicar nuestros últimos cambios pero desde un commit anterior.

git reset --soft: Borramos todo el historial y los registros de Git pero guardamos los cambios que tengamos en Staging, así podemos aplicar las últimas actualizaciones a un nuevo commit.

git reset --hard: Borra todo. Todo todito, absolutamente todo. Toda la información de los commits y del área de staging se borra del historial.

¡Pero todavía falta algo!

git reset HEAD: Este es el comando para sacar archivos del área de staging. No para borrarlos ni nada de eso, solo para que los últimos cambios de estos archivos no se envíen al último commit, a menos que cambiemos de opinión y los incluyamos de nuevo en staging con git add, por supuesto.

¿Por qué esto es importante?

Imagina el siguiente caso:

Hacemos cambios en los archivos de un proyecto para una nueva actualización. Todos los archivos con cambios se mueven al área de staging con el comando git add. Pero te das cuenta de que uno de esos archivos no está listo todavía. Actualizaste el archivo, pero ese cambio no debe ir en el próximo commit por ahora.

¿Qué podemos hacer?

Bueno, todos los cambios están en el área de Staging, incluido el archivo con los cambios que no están listos. Esto significa que debemos sacar ese archivo de Staging para poder hacer commit de todos los demás.

¡Al usar git rm lo que haremos será eliminar este archivo completamente de git! Todavía tendremos el historial de cambios de este archivo, con la eliminación del archivo como su última actualización. Recuerda que en este caso no buscábamos eliminar un archivo, solo dejarlo como estaba y actualizarlo después, no en este commit.

git reset HEAD “nombrearchivo”

En cambio, si usamos git reset HEAD, lo único que haremos será mover estos cambios de Staging a Unstaged. Seguiremos teniendo los últimos cambios del archivo, el repositorio mantendrá el archivo (no con sus últimos cambios pero sí con los últimos en los que hicimos commit) y no habremos perdido nada.

Conclusión: Lo mejor que puedes hacer para salvar tu puesto y evitar un incendio en tu trabajo es conocer muy bien la diferencia y los riesgos de todos los comandos de Git.

### Introducción a las ramas o branches de Git

Git commit –am “mensaje para commit”, solo funciona con archivos dónde se la ha hecho add previamente

La branch se hace desde la rama principal de donde estoy, se deb hacer desde la master

1. Hacer git status para ver dónde estoy, debe ser la master inicialmente
2. Crear la nueva rama

git branch “nombreRama”

1. Verificar la correcta creación d ela nueva rama

git show

1. Cambiar de rama

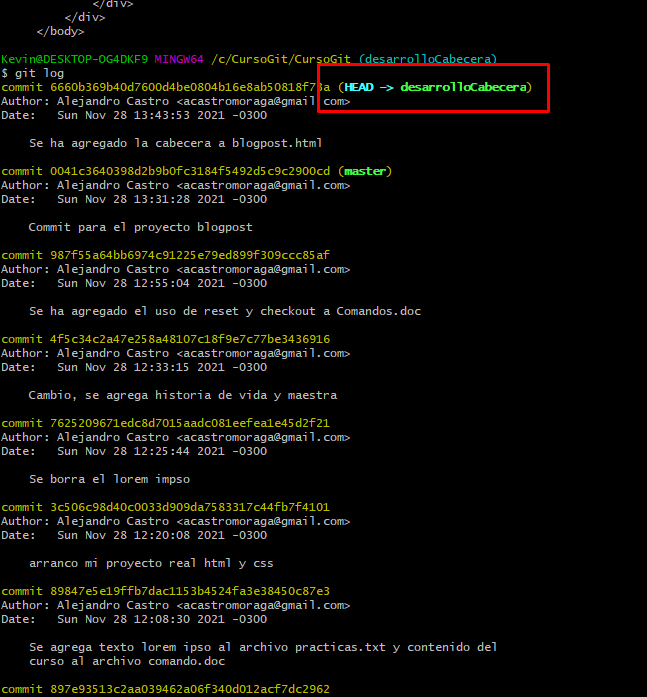
Git checkout “nombrerama”

1. Hacer los cambios y verificar la rama donde me encuentro

git show

1. Para verificar los cambios globales dentro del pryecto y verificar los commit en todos las ramas

Git log



1. Para volver a la rama master sin hacerle push

Git checkout master

Por ahora, nuestro proyecto vive únicamente en nuestra computadora. Esto significa que no hay forma de que otros miembros del equipo trabajen en él.

Para solucionar esto están los servidores remotos: un nuevo estado que deben seguir nuestros archivos para conectarse y trabajar con equipos de cualquier parte del mundo.

Estos servidores remotos pueden estar alojados en GitHub, GitLab, BitBucket, entre otros. Lo que van a hacer es guardar el mismo repositorio que tienes en tu computadora y darnos una URL con la que todos podremos acceder a los archivos del proyecto para descargarlos, hacer cambios y volverlos a enviar al servidor remoto para que otras personas vean los cambios, comparen sus versiones y creen nuevas propuestas para el proyecto.

Esto significa que debes aprender algunos nuevos comandos:

git clone url\_del\_servidor\_remoto: Nos permite descargar los archivos de la última versión de la rama principal y todo el historial de cambios en la carpeta .git.

git push: Luego de hacer git add y git commit debemos ejecutar este comando para mandar los cambios al servidor remoto.

git fetch: Lo usamos para traer actualizaciones del servidor remoto y guardarlas en nuestro repositorio local (en caso de que hayan, por supuesto).

git merge: También usamos el comando git merge con servidores remotos. Lo necesitamos para combinar los últimos cambios del servidor remoto y nuestro directorio de trabajo.

git pull: Básicamente, git fetch y git merge al mismo tiempo.

### Introducción a las ramas o branches de Git

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).

### 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 indiquemos después del comando).

# Crear un nuevo commit en la rama master combinando

# los cambios de la rama cabecera:

git checkout master

git merge cabecera

# Crear un nuevo commit en la rama cabecera combinando

# los cambios de cualquier otra rama:

git checkout cabecera

git merge cualquier-otra-rama

git branch muestra las ramas que están contenidas dentro del repositorio y donde me encuentro posicionado

1. Debo traerme los cambios que se hicieron en desarrollo y desde la rama de master hacer el merge

### Resolución de conflictos al hacer un merge

### Uso de GitHub

1. Traer todos los archivos del repositorio remoto , se debe traer en los caso a partir del 2020 los archivos del main

git remote add origin <https://github.com/acastromoraga/FrameWorkVersionTrabajo.git>

luego se debe hacer un git pull para traer las cosas del repositorio remoto

git pull origin (master/mainI)

si existe algún archive de conflict por ya existir

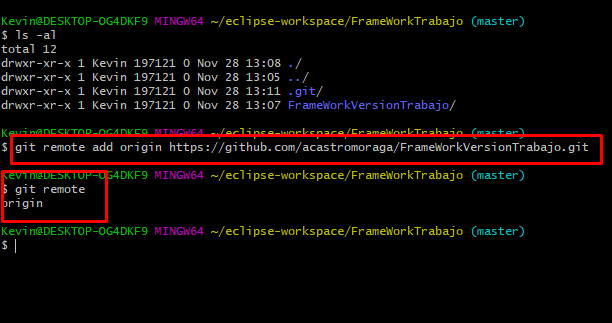
git pull origin master/main –allow-unrelated-histories

origin master, fucionar lo de la rama y lo del repositorio real

porner yes

1. Verificar que s etrajo los cambios

git remote



Explicacion verbal

git remote –v

hacer push a la rama del repositorio (master/main)

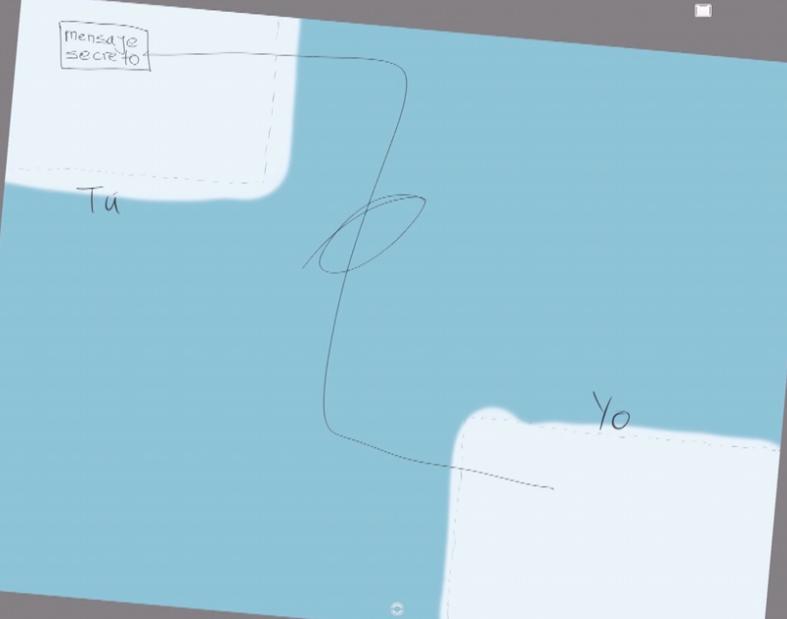
git push origin master/main

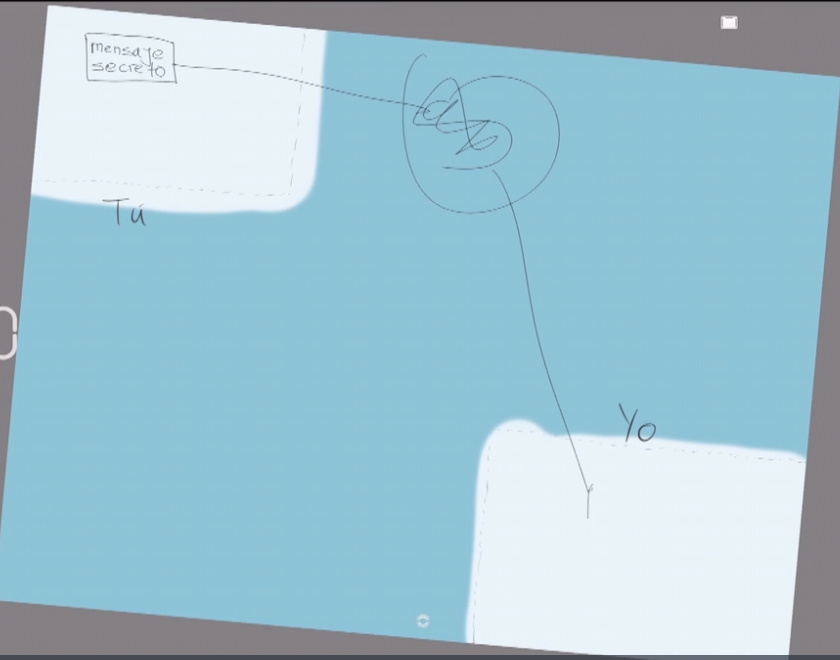
En git hub en la sección de blame se puede ver el historial de los commit

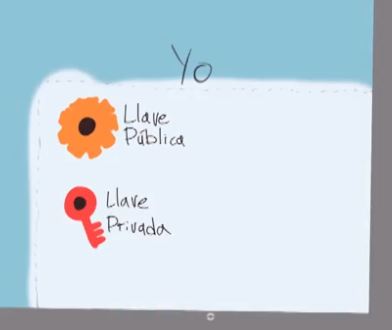
Luego puedo bajar los cambios a mi repositorio local

Git pull origin master

### Cómo funcionan las llaves públicas y privadas







Las llaves publicas se comparten, los mensajes se convieten en mensajes secretos

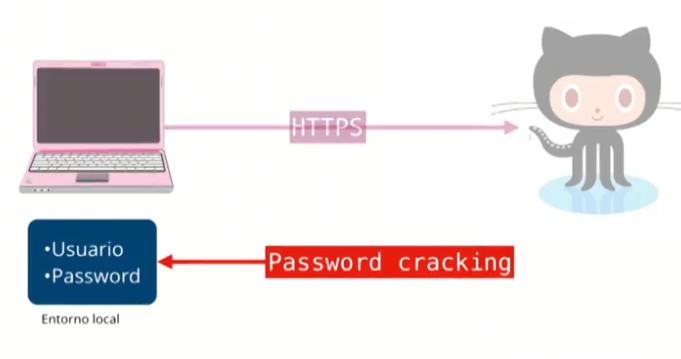
La llave publica queda conectada con la llave privada, la otra persona copa mi llave publica

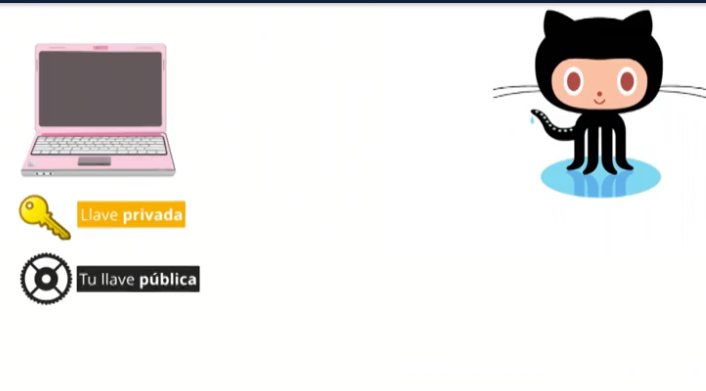
Ejemplo otro me envía un mesaje (él tiene mi llave publica, la cual copio de algun lado o yo se la envié, la cual esta conectada con mi llave privada) el pasa el mensaje por mi llave publica y me envía un mensaje

Copio el mensaje y uso mi llave privada para decifrar el mensaje, todo se rige que me envién el mensaje con mi llave publica y yo lo decifro con mi llave privada

El otro cifra su mensaje con mi llave, me envía el mensaje y yo lo decifro con mi llave privada

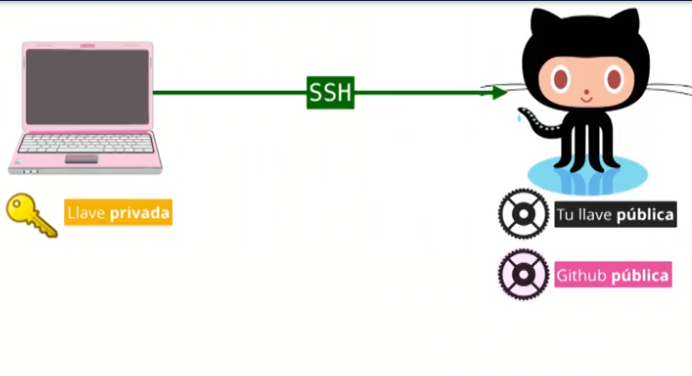
### Configura tus llaves SSH en local

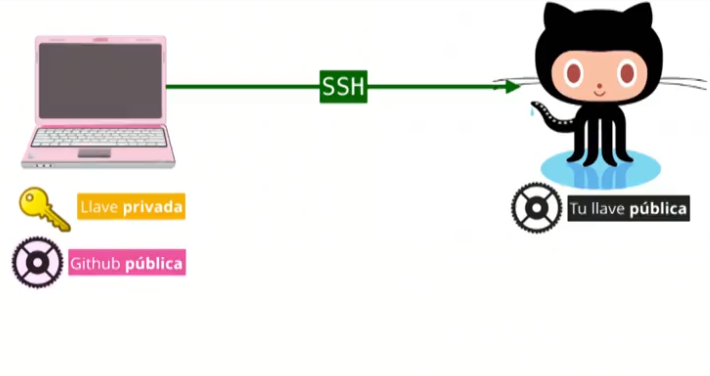




En entorno local se crea las llaves, se envia la llave publica a github, se conecta por un protocolo SSH,

En la primera conexión git hub una vez le llega mi llave publica, me envía su llave publica

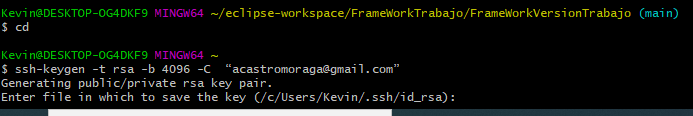




La llave privada mia le puedo poner otra contraseña

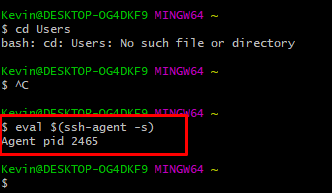
1. Primer paso: Generar tus llaves SSH. Recuerda que es muy buena idea proteger tu llave privada con una contraseña.

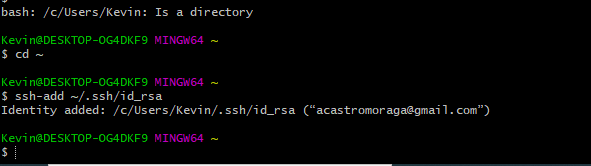
ssh-keygen -t rsa -b 4096 -C “[tu@email.com](mailto:tu@email.com)”

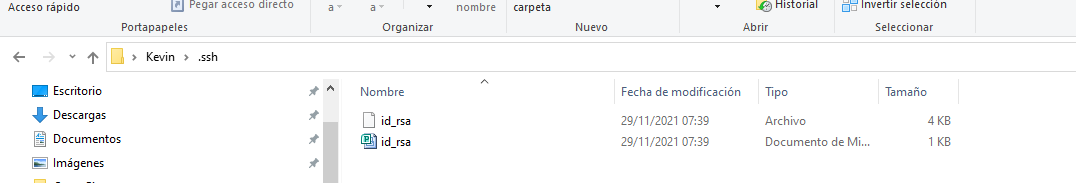


SHA256:o7Kw4IBQx1gXUokytTo4JqZop9SMurIAL24UOhv7VdU [acastromoraga@gmail.com](mailto:acastromoraga@gmail.com)

Sin contraseña







1. Segundo paso: Terminar de configurar nuestro sistema.

En Windows y Linux:

# Encender el "servidor" de llaves SSH de tu computadora:

eval $(ssh-agent -s)

1. # Añadir tu llave SSH a este "servidor":

ssh-add ruta-donde-guardaste-tu-llave-privada

En Mac:

# Encender el "servidor" de llaves SSH de tu computadora:

eval "$(ssh-agent -s)"

# Si usas una versión de OSX superior a Mac Sierra (v10.12)

# debes crear o modificar un archivo "config" en la carpeta

# de tu usuario con el siguiente contenido (ten cuidado con

# las mayúsculas):

Host \*

AddKeysToAgent yes

UseKeychain yes

IdentityFile ruta-donde-guardaste-tu-llave-privada

# Añadir tu llave SSH al "servidor" de llaves SSH de tu

# computadora (en caso de error puedes ejecutar este

# mismo comando pero sin el argumento -K):

ssh-add -K ruta-donde-guardaste-tu-llave-privada

Las llaves no es por proyecto sino por persona