

Curso Git y GitHub

Introducción a Git y Git Hub

Git es un sistema de control de versiones distribuido que permite realizar un seguimiento de los cambios en los archivos y volver a versiones anteriores si algo va mal. Fue diseñado por Linus Torvalds para garantizar la eficiencia y fiabilidad del mantenimiento de versiones de aplicaciones que tienen un gran número de archivos de código fuente.

1. Git está optimizado para guardar cambios de forma incremental.
2. Permite tener un historial, volver a una versión anterior y añadir funcionalidades.
3. Hace un seguimiento de los cambios que otras personas realizan en los archivos.

Git fue diseñado para funcionar en un entorno Linux. Actualmente es multiplataforma, es decir, puede compartirse con Linux, MacOS y Windows. En una máquina local es Git, se utiliza bajo el terminal o línea de comandos y dispone de comandos como merge, pull, add, commit y rebase entre otros.

¿Para que tipo de proyectos es Git?

Con Git se consigue una mayor eficiencia utilizando archivos de texto plano, ya que con los archivos binarios no se pueden guardar sólo los cambios, sino que hay que volver a guardar el archivo completo para cada modificación, por pequeña que sea, lo que aumenta demasiado el tamaño del repositorio.

Características de Git

Git te ayuda a trabajar de forma más organizada y colaborativa en proyectos de desarrollo de software. Estas son algunas de sus principales características:

1. **Control de versiones:** Git almacena la información como un conjunto de archivos. Te permite hacer un seguimiento de los cambios que realizas en tus archivos, lo que significa que siempre puedes volver a versiones anteriores si algo va mal.
2. **Ramificación:** Puedes crear ramas en tu proyecto, lo que te permite trabajar en diferentes características o aspectos de tu proyecto sin afectar al trabajo de los demás.
3. **Colaboración:** En Git, varias personas pueden trabajar en diferentes aspectos del proyecto al mismo tiempo.
4. **Seguridad:** No hay cambios, corrupción de archivos o cualquier alteración sin que Git lo sepa. Git tiene tres estados en los que se pueden encontrar los archivos: Staged, Modified y Committed.
5. **Flexibilidad:** Casi todo en Git es local. Es difícil necesitar recursos o información externa, basta con los recursos locales disponibles.
6. **Comandos:** Git tiene una sintaxis de comandos bastante sencilla y fácil de aprender, lo que lo hace accesible incluso para principiantes en programación.
- 7.

Que es un sistema de control de versiones

El VCS es un sistema que registra los cambios realizados en un archivo o conjunto de archivos a lo largo del tiempo, de modo que se puede hacer un seguimiento del ciclo de vida de un proyecto, comparar los cambios a lo largo del tiempo, ver quién los hizo o revertir todo el proyecto a un estado anterior. Cualquier tipo de archivo de un ordenador puede someterse al control de versiones.


¿Cuál es la diferencia entre Git y GitHub?

GitHub es una plataforma de desarrollo colaborativo para alojar proyectos que utilizan el sistema de control de versiones Git. Se utiliza principalmente para crear código fuente de programas informáticos. GitHub se puede considerar como la red social de código para programadores y, en muchos casos, se ve como un currículum, ya que aquí se almacena la cartera de proyectos de programación.

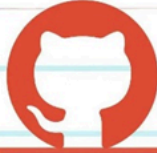
Algunas de sus características son:


1. GitHub permite alojar proyectos en repositorios de forma gratuita y pública, aunque tiene una forma de pago para la privada.
2. Puedes compartir fácilmente tus proyectos.
3. Puedes colaborar para mejorar los proyectos de otros y permitir que otros mejoren o contribuyan a los tuyos.

4. Ayuda a reducir significativamente los errores humanos, a tener un mejor mantenimiento de los diferentes entornos y a detectar fallos de forma más rápida y eficiente.
5. Es la opción perfecta para trabajar en equipo en un mismo proyecto.
6. Ofrece todas las ventajas del sistema de control de versiones Git, pero además cuenta con otras herramientas que ayudan a tener un mejor control de los proyectos.



¿QUÉ ES GIT?





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 éstas tienen un gran número de archivos de código fuente.

Sistema operativo: Unix-like, Windows, Linux
 Programado en: C, Bourne Shell, Perl
 Modelo de desarrollo: Software libre
 Escrito en: C, Perl, Tcl, Python

Importante

Directorios en Git
 Es el lugar donde se almacenan los metadatos y las bases de datos para nuestros proyectos, y es justamente lo que se copia cuando clonamos de un ordenador a otro los archivos.

GitHub es una forja para alojar proyectos utilizando el sistema de control de versiones Git. Se utiliza principalmente para la creación de código fuente de programas de ordenador. El software que opera GitHub fue escrito en **Ruby on Rails**. Desde enero de 2010, GitHub opera bajo el nombre de GitHub, Inc.


☒ **GIT**

Git es un sistema de control de versiones que originalmente fue diseñado para operar en un entorno Linux. Actualmente Git es multiplataforma, es decir, es compatible con Linux, MacOS y Windows.

Características de Git

- Git almacena la información como un conjunto de archivos.
- No existen cambios, corrupción en archivos o cualquier alteración sin que Git lo sepa.
- Casi todo en Git es local. Es difícil que se necesiten recursos o información externos, basta con los recursos locales con los que cuenta.
- Git cuenta con 3 estados en los que podemos localizar nuestros archivos: Staged, Modified y Committed


☒ **GITHUB**

 **github**


GitHub es un servicio de alojamiento que ofrece a los desarrolladores repositorios de software usando el sistema de control de versiones, Git.

Características de Github

- GitHub permite que alojemos proyectos en repositorios de forma gratuita y publica, pero tiene una forma de pago para privados.
- Puedes compartir tus proyectos de una forma mucho más fácil.
- Te permite colaborar para mejorar los proyectos de otros y a otros mejorar o aportar a los tuyos.
- Ayuda reducir significativamente los errores humanos, a tener un mejor mantenimiento de distintos entornos y a detectar fallos de una forma más rápida y eficiente.
- Es la opción perfecta para poder trabajar en equipo en un mismo proyecto.
- Ofrece todas las ventajas del sistema de control de versiones, Git, pero también tiene otras herramientas que ayudan a tener un mejor control de nuestros proyectos.



En vez de guardar un mismo archivo varias veces. Git nos ayuda a guardar solo los cambios del mismo, además maneja los cambios que otras personas hagan sobre los mismos archivos, así múltiples personas pueden trabajar en un mismo proyecto sin conflictos. Git permite rastrear que miembro realiza los cambios, además de recuperar una versión antigua de manera precisa. Github nos permite publicar un repositorio para trabajarlo de forma remota y colaborar con otros miembros dentro y/o fuera de nuestra organización.



Editores de código, archivos binarios, de texto y conceptos de Git

Un editor de código o IDE es una herramienta que proporciona muchas ayudas para escribir código, algo así como un bloc de notas muy avanzado. Los editores más populares son VSCode, Sublime Text y Atom, pero no es obligatorio utilizar ninguno de ellos para programar. Más información sobre qué es un IDE.

Tipos de archivos y sus diferencias

1. **Archivos de texto (.txt):** Texto plano normal sin nada especial. Lo vemos igual independientemente de dónde lo abramos, ya sea con el bloc de notas o con editores de texto avanzados.
2. **Archivos RTF (.rtf):** Podemos guardar texto con diferentes tamaños, estilos y colores. Pero si lo abrimos desde un editor de código, veremos que es mucho más complejo que un simple texto plano. Esto se debe a que debe guardar todos los estilos del texto y, para ello, utiliza un código especial un poco difícil de entender y muy diferente a los textos con estilos especiales a los que estamos acostumbrados.
3. **Archivos de Word (.docx):** Podemos guardar imágenes y texto con diferentes tamaños, estilos o colores. Cuando lo abrimos desde un editor de código podemos ver que se trata de código binario, muy difícil de entender y muy diferente al texto al que estamos acostumbrados. Esto se debe a que Word está optimizado para entender este código especial y representarlo gráficamente.

Para ver el formato de un archivo, recuerda que debes habilitar la opción para ver la extensión de los archivos, de lo contrario, sólo podrás ver su nombre. La forma de hacerlo en Windows es Ver > Mostrar u ocultar > Extensiones de nombre de archivo.

Conceptos básicos de Git

Dentro de los conceptos esenciales de Git encontramos:

1. **Bug:** error en el código.
2. **Repositorio:** Donde se almacena todo el proyecto, que puede vivir en local o remoto. El repositorio guarda un histórico de versiones y, lo que es más importante, de la relación de cada versión con la anterior para poder hacer el árbol de versiones con las distintas ramas.
3. **Fork:** Si en algún momento queremos contribuir al proyecto de otra persona, o si queremos utilizar el proyecto de otra persona como punto de partida para el nuestro, esto se conoce como «fork», la cuál es una copia exacta que se usa para trabajar en el proyecto sin afectar el proyecto original.

4. **Clonar:** Copia exacta de un proyecto en un repositorio local que contiene todos los archivos y su historial.
5. **Rama:** Es una bifurcación del proyecto que se realiza para añadir una nueva funcionalidad o corregir un error.
6. **Master:** Rama donde se almacena la última versión estable del proyecto. La rama maestra es la que está en producción en todo momento (o casi) y debe estar libre de bugs. Así, la rama que está en producción, sirve de referencia para hacer nuevas funcionalidades y/o arreglar bugs de última hora.
7. **Commit:** Confirmación de guardado de cambios a un archivo, carpeta o proyecto que es identificado con un ID único que permite rastrear que cambios ocurrieron y en que momento fueron hechos.
8. **Push:** Consiste en enviar todo lo que se ha confirmado con un commit al repositorio remoto.
9. **Checkout:** Acción de navegación entre commits efectuados, permite regresar el repositorio a una versión anterior o regresar a la actual.
10. **Fetch:** Actualiza el repositorio local descargando datos del repositorio remoto al repositorio local sin actualizarlo, es decir, se guarda una copia del repositorio remoto en el repositorio actual sin fusionar la data.
11. **Merge:** La acción merge es la continuación natural de la fetch. La fusión o merge permite fusionar la copia del repositorio remoto con el repositorio local, mezclando los diferentes códigos.
12. **Pull:** Consiste en la unión del fetch y el merge, es decir, recoge la información del repositorio remoto y luego mezcla con ella el trabajo local en una sola acción.
13. **Diff:** Sirve para mostrar los cambios entre dos versiones de un mismo fichero.
14. **Log:** Acción que permite ver el histórico de commits de un archivo o del repositorio.

Como Crear un Repositorio con Git

Para crear un repositorio local primero debemos generar una carpeta de nuestro proyecto con toda la estructura y archivos pertinentes. Una vez el proyecto está ubicado en un único lugar, generamos un repositorio como una data “oculta” en donde la data del repositorio (archivos más histórico) va a almacenarse. Para crear un repositorio, debemos abrir nuestra terminal (para usuarios Windows es recomendable utilizar GitBash, para usuarios linux basta con usar la terminal nativa) y ejecutar los siguientes comandos:

- **git init:** Comando que crea e inicializa el repositorio dentro de la carpeta del proyecto.
- **git add:** Este comando, seguido de “nombrearchivo.txt” o “nombre_carpeta” nos permite enviar los archivos a la fase de staged. El comodín “git add .” agrega todos los archivos del proyecto a staged.
- **git commit -m 'mensaje' :** Guarda los cambios enviados a staging y da un ID único para ese cambio.

- **git status:** Comando para ver el estado del repositorio, si necesita ser añadido al staged o si se requiere commit
- **git conf:** Comando para ver las posibles configuraciones de Git en el repositorio.
- **git conf --list:** Comando para ver la lista de configuraciones realizadas.
- **git conf --list --show-origin:** Comando para mostrar las configuraciones y sus rutas.
- **git rm --cached nombre_archivo.txt:** Comando para eliminar el archivo de staged (ram).
- **git rm nombre_archivo.txt:** Comando para eliminar del repositorio.

Si por alguna razón te equivocaste en el nombre o email que configuraste al principio, puedes modificarlo de la siguiente manera:

- **git config --global --replace-all usuario.nombre «Aquí tienes tu nombre modificado».**

O si quieres eliminarlo y añadir uno nuevo

- **git config --global --unset-all usuario.nombre :** Elimina el nombre del usuario
- **git config --global --add usuario.nombre «Aquí va tu nombre».**

Monitorear Cambios Sobre un Repositorio

El comando **git show** nos muestra los cambios que se han realizado en un archivo y es muy útil para detectar cuándo se produjeron ciertos cambios, qué se “rompió” y cómo podemos arreglarlo. 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 commitB**. Recuerda que puedes obtener el ID de tus commits con el comando **git log**.

Comandos para analizar cambios en Git:

- **git status:** Mostrar los cambios efectuados recientemente.
- **git log filename.extension:** Historial de cambios con detalles, si se ejecuta solo git log, se muestra la historia de todos los commits hechos hasta el momento.

Supongamos que queremos explorar un poco nuestro repositorio, así pues, a continuación se enlistan unos métodos para ello que incluyen comandos básicos de terminal:

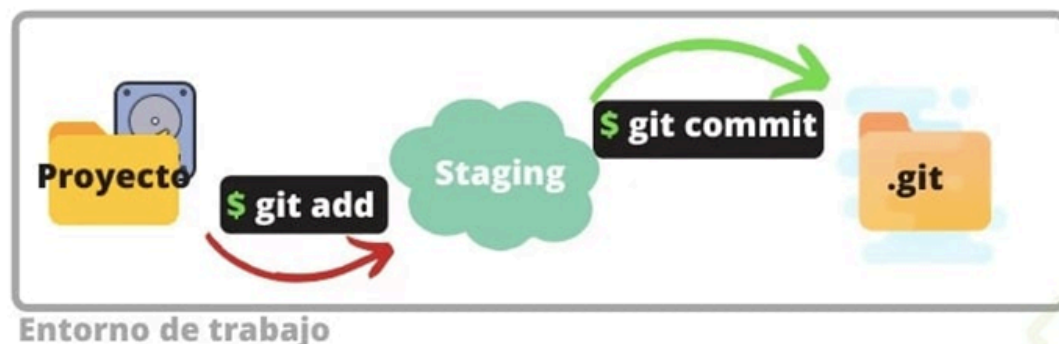
- **ls:** Lista los archivos o carpetas dentro de la dirección de trabajo actual.
- **cd “ruta”:** Cambia el directorio actual a la ruta especificada.
- **pwd:** Indica el directorio actual.
- **touch file_name.extension:** Crea un archivo vacío en el directorio actual.
- **cat file.extension:** Muestra por consola el archivo especificado.
- **history:** Muestra la historia de comandos ejecutados por terminal.
- **rm file.extension:** Elimina un archivo especificado.
- **command --help:** Muestra las funcionalidades de un comando.

Una vez hemos recordado los comandos básicos de cualquier terminal, veamos unos comandos en git para monitorear/revertir brevemente los cambios hechos:

- **git checkout:** Ayuda a recuperar los cambios hechos.
- **git log :** Muestra la historia general de commits (cambios confirmados) hechos.
- **git log :** Muestra la historia de commits de un archivo.
- **git rm --cached archivo.extension:** Sirve para devolver el archivo que tenemos en memoria ram, es decir, lo sacamos de la etapa de staging.
- **git config --list:** Muestra la lista de git config.
- **git config --list --show-origin:** rutas de git config
- **git log file.extension:** muestra el historial del archivo.
- **git show file_name:** Muestra los cambios que han existido sobre un archivo y es útil para detectar cuándo se produjeron.
- **git diff comitA_ID commitB_ID:** Muestra los cambios o diferencias entre dos commits.

Estados de Archivos en Git

El **staging** es el lugar donde se guardan temporalmente los cambios y luego se trasladan al repositorio (se almacenan los cambios en memoria ram). El repositorio es el lugar donde se almacenan todos los registros de los cambios realizados en los archivos, así pues, cuando se inicializa un repositorio con git init se crea una carpeta que almacena los cambios solo si estos son confirmados mediante un commit. Si se cierra la operación de la terminal sin antes confirmar los cambios y solo se han agregado a la fase de staging, éstos cambios se perderán. El área de staging puede verse como una fase intermedia donde nuestros archivos están a punto de ser enviados al repositorio o devueltos a la carpeta del proyecto.



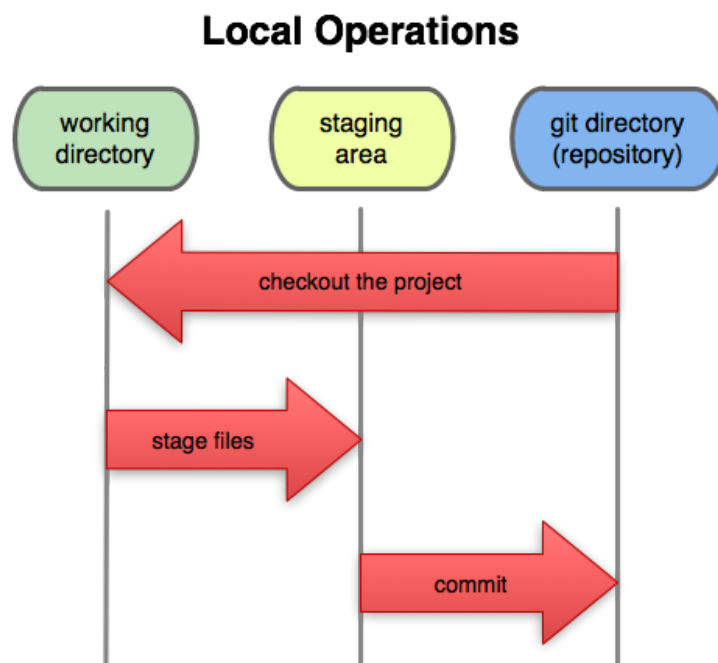
El flujo de trabajo básico en git es:

1. Modificación de un conjunto de archivos en tu directorio de trabajo.
2. Envío a staging de los archivos.

3. Confirmación de los cambios, lo que toma los archivos tal y como están en el área de preparación y almacena esa instancia de los archivos de forma permanente en tu directorio git.

El repositorio almacena los metadatos y objetos de base de datos de tu proyecto. Es la parte más importante de git (en esencia, la carpeta `.git` **es el repositorio como tal**) y es lo que se copia cuando clonas un repositorio desde otro ordenador. Las áreas o lugares presentes en un ciclo de trabajo con Git son:

- **Working Directory:** Ubicación en la máquina local de los archivos que aún no están rastreados por git.
- **Staging:** Se trata de un área que almacena información sobre lo que se incluirá en la siguiente confirmación.
- **.git directory (repository):** El repositorio almacena los metadatos y objetos de base de datos de tu proyecto.



Cuando trabajamos con git, nuestros archivos pueden vivir y moverse entre 4 estados diferentes (cuando trabajamos con repositorios remotos pueden ser más estados, pero lo estudiaremos más adelante):

- **Archivos rastreados (tracked files):** Son los archivos que viven dentro del proyecto y en el repositorio `.git`, no tienen cambios pendientes y sus últimas actualizaciones se han guardado en el repositorio gracias a los comandos `git add` y `git commit`.
- **Staged files:** Son los archivos que se encuentran dentro del área de staging. Viven dentro de git y hay constancia de ellos ya que han sido afectados por el comando `git add`, aunque no sus últimos cambios. Git sabe de la existencia de estos últimos cambios, pero aún no se han guardado definitivamente en el repositorio porque aún no se ha ejecutado el comando `git commit`.
- **Archivos unstaged:** Entiéndelos como archivos «tracked but unstaged». Son archivos que viven dentro del proyecto que poseen cambios detectados por git pero que no han sido afectados por el comando `git add`, y mucho menos por `git commit`. Git tiene un registro

de estos archivos, pero está desactualizado, sus últimas versiones sólo se almacenan en el disco duro.

- **Archivos no rastreados:** Son archivos que NO viven dentro de git, sólo en el disco duro. Nunca han sido afectados por git add, por lo que git no tiene registro de su existencia.

Recuerda que hay un caso muy raro en el que los archivos tienen dos estados al mismo tiempo: staged y untracked. Esto ocurre cuando guardas cambios en un archivo en el área de preparación (con el comando git add), pero antes de confirmar para guardar los cambios en el repositorio, haces nuevos cambios que aún no se han guardado en el área de preparación.

Comandos para moverse entre estados de Git

Para cambiar los estados de los archivos en Git, estos son los comandos más importantes que debes conocer, donde ten en cuenta que el **HEAD** es el commit más reciente de la rama actual:

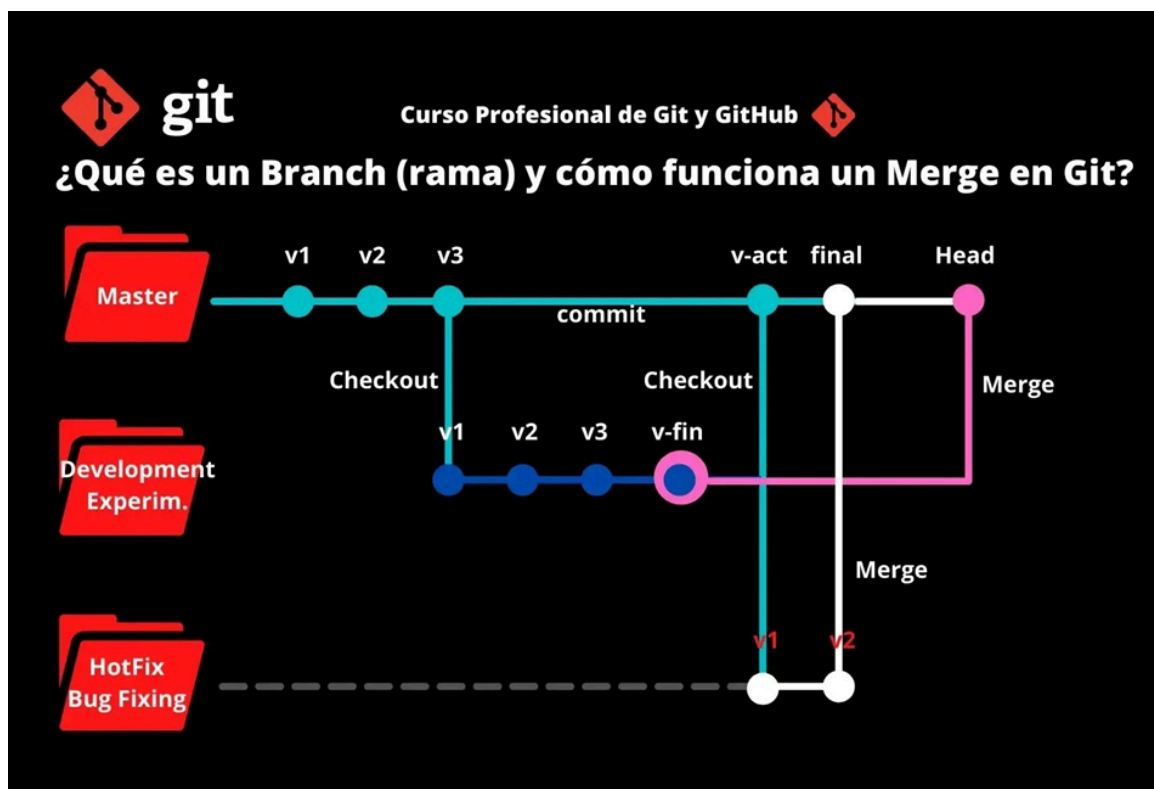
- **git status:** Permíte ver el estado de todos nuestros archivos y carpetas.
- **git add:** Mover archivos del estado untracked o unstaged al staged.
- **git reset HEAD:** Nos ayuda a sacar archivos del estado staged y devolverlos a su estado anterior. Si los archivos venían de unstaged, vuelven allí. Lo mismo ocurre si los archivos estaban en estado untracked.
- **git commit:** Nos ayuda a mover archivos de unstaged a tracked. Git nos pedirá que dejemos un mensaje para recordar los cambios que hemos hecho y podemos usar el argumento **-m** para escribirlo (git commit -m «mensaje»).
- **git rm:** Este comando necesita uno de los siguientes argumentos para ejecutarse correctamente: **git rm --cached**, mueve los archivos que especifiquemos al estado untracked. **git rm --force** elimina los archivos de git y del disco duro. Git mantiene un registro de la existencia de los archivos, por lo que podremos recuperarlos si es necesario (pero deberemos usar comandos más avanzados).

Ramas en Repositorios, Cambio de Versión y Merge

Una rama es una versión del código del proyecto en el que se está trabajando. Estas ramas ayudan a mantener el orden en el control de versiones y a manipular el código de forma segura. En otras palabras, una rama en Git es una consecución de commits que permiten mantener aisladas las versiones de un proyecto sin que entren en conflicto. Puedes imaginar a un proyecto en repositorio como un árbol donde cada rama es una versión del trabajo hecho, que tiene una rama gruesa, y una rama delgada. En la rama gruesa tenemos los commits principales (aquellos donde las versiones más maduras del proyecto están, incluso,

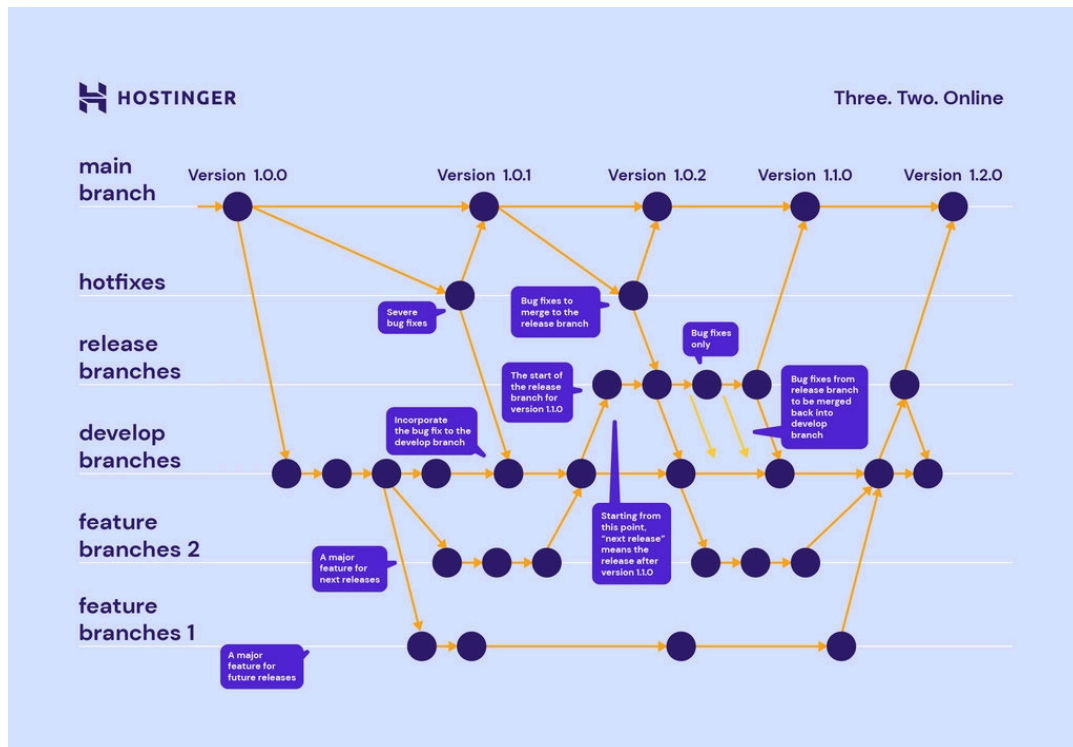
la rama main o master, se usa como la rama que contiene los proyectos que están desplegados en producción) y en la rama delgada tenemos otros commits que pueden ser hotfix, desarrollo y otros.

En la imagen anterior veremos ejemplos de tres ramas, donde la rama master es aquella que contiene la versión más importante de nuestro proyecto. Por otra parte la rama development es la parte de nuestro proyecto en donde experimentamos con cambios, implementaciones etc, mientras que hotfix es aquella rama usada para reparar bugs dentro de la rama principal. En esta imagen se ilustra como un **checkout** es un cambio de rama o commit que permite navegar entre versiones y puntos de control del proyecto. Es usual que muchas ramas creadas no sean permanentes sino temporales, donde por ejemplo, una vez terminado un experimento de desarrollo, la rama se finaliza al hacer un **merge** entre el último commit de la rama de desarrollo y la rama principal. No obstante, si en la rama principal se detecta un problema, una rama aún más temporal se crea para poder aislar los reparos al proyecto, para una vez consolidarlas, fusionarlos dentro del proyecto en su versión principal.



Dentro de Git, es posible crear tantas ramas como sea necesarias, de hecho, es una muy buena práctica de desarrollo, crear una rama cuando una necesidad en el proyecto en específico se presente. Por ejemplo, en la imagen siguiente, tenemos la rama main la cual contiene la versión madura de nuestro proyecto, hotfixes es usada para reparar en paralelo los errores que por ejemplo, el proyecto en producción presente inesperadamente, release branches se encarga de tener aquellas versiones maduras de nuestro proyecto que pueden o no ser pertinentes para incorporar a nuestro main. Develop es usada para integrar nuevos

experimentos e integrar nuevas características que son desarrolladas en paralelo en ramas como feature 1 y feature 2.



En el manejo y creación de ramas, los siguientes comandos se ven involucrados:

- **git branch branch_name:** Crea una nueva rama en el repositorio.
- **git checkout branch_name:** Cambia el apuntador de git (el HEAD de la rama) a una rama distinta, de modo que todo nuevo commit ocurre en la rama branch_name.
- **git branch:** Imprime por consola el nombre de la rama actual.
- **git merge branch_name:** Fusiona la rama actual con branch_name

No obstante, ¿Cómo podemos movernos entre commits y volver a versiones anteriores o incluso, retornar a una versión actual de nuestro proyecto o de un archivo?. Para ello, empleamos los siguientes comandos, donde recordemos que si queremos conocer el ID de un commit, basta con usar **git log** para ver el historial el cual nos muestra un histórico de los commits realizados:

- **git checkout ID_commit:** Permite cambiar la versión del proyecto al commit especificado por ID_commit sin eliminar algún punto del historial de commits.
- **git revert id_commit:** Revierte un commit.

Diferencias entre git reset y git rm

En el caso de usar “git checkout ID_commit”, nos permite ir a versiones previas de nuestro repositorio y con . No obstante existe una manera de devolver nuestro proyecto a versiones anteriores de una manera más drástica que **no** nos permitirá regresar a las versiones más actuales del proyecto.

- **git reset:** Con el argumento --hard, es posible borrar toda la información que tengamos en el área de staging y en commits posteriores a un commit especificado, por otra parte, con el argumento --soft, se mantienen los archivos del área de staging para que podamos aplicar nuestros últimos cambios pero desde un commit anterior. Con git reset es posible volver al pasado pero nunca al futuro. Git reset nos permite movernos entre commits para deshacer ciertos commits.

Dentro de los tipos de git reset encontramos:

1. **git reset --soft:** Borra el historial y los logs de Git de confirmaciones anteriores, pero guarda los cambios en Staging para aplicar las últimas actualizaciones a una nueva confirmación.
2. **git reset --hard:** Deshace todo, absolutamente todo. Toda la información del commit y del área de staging se elimina del historial. Si se implementa en la forma “git reset ID_commit”, se regresa el repositorio a la versión especificada pero toda información posterior se pierde.
3. **git reset --mixed:** Borra todo, exactamente todo. Todos los commits y la información del área de preparación se eliminan del historial.
4. **git reset HEAD:** El comando git reset elimina los archivos del área de preparación sin borrarlos ni realizar otras acciones. Esto evita que los últimos cambios en estos archivos se envíen al último commit. Podemos incluirlos de nuevo en el staging con git add si cambiamos de opinión.

Ten en cuenta que, si deshaces confirmaciones en un repositorio compartido en GitHub, estarás cambiando su historial y esto puede causar problemas de sincronización con otros contribuidores.

Por otra parte, tenemos que rm es:

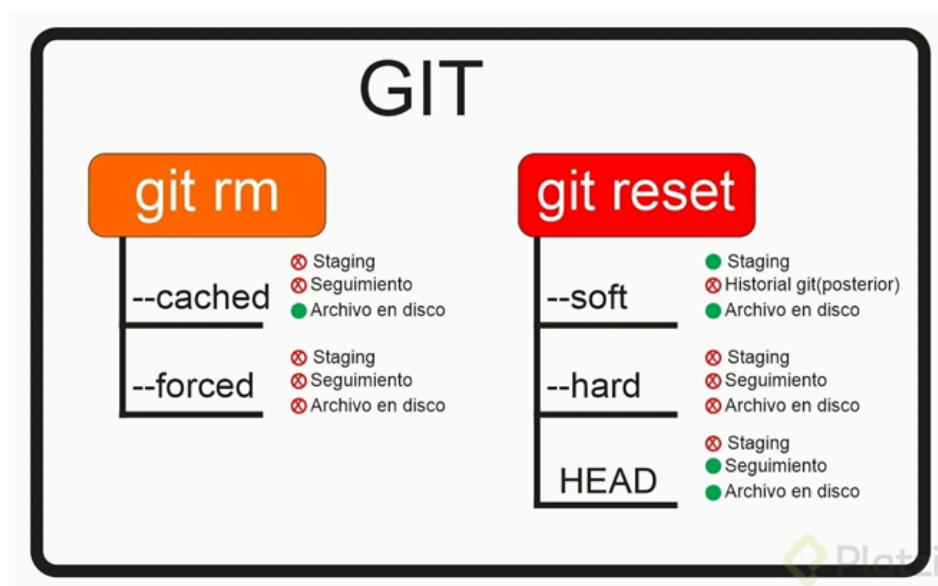
- **git rm:** Es un comando que nos ayuda a borrar archivos de Git sin eliminar su historial del sistema de versiones. Para recuperar el archivo borrado, necesitamos retroceder en el historial del proyecto, recuperar el último commit y obtener el último commit anterior al borrado del archivo. Es importante tener en cuenta que git rm no se puede utilizar sin evaluarlo antes.

Dentro de los tipos de git rm encontramos lo siguiente, no obstante que mientras git rest afecta los historiales y el staging del repositorio, rm afecta al espacio de trabajo, a los archivos como tal:

1. **git rm --cached:** Elimina los archivos del repositorio local y del área de preparación, pero los mantiene en el disco duro. Detiene el seguimiento del historial de cambios de estos archivos, por lo que permanecen en estado sin seguimiento.

2. **git rm --force:** Elimina los archivos del repositorio y del disco duro. Git lo guarda todo, así que podemos recuperar los archivos borrados si es necesario (usando comandos avanzados).
3. **git rm:** Se elimina todo, absolutamente todo, incluyendo los archivos del proyecto (eliminación de disco duro) y el historial del repositorio.

La principal diferencia entre `git rm` y `git reset` es que `git rm` elimina los archivos del repositorio y del historial del proyecto, mientras que `git reset` elimina los cambios del área de preparación y los mueve del espacio de trabajo. Note entonces que `git rm --forced` es una forma de **eliminar archivos del proyecto**, mientras que `git reset ID_commit --hard` es una manera de **restaurar forzosamente sin posibilidad de revertir, el proyecto a una versión específica**.

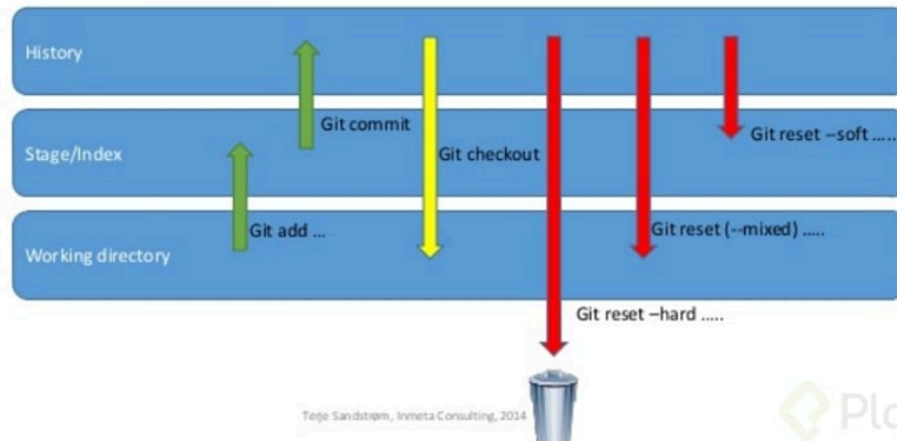


En éste sentido, para reescribir el historial del repositorio y eliminar confirmaciones previas, se usa `git reset`. Para deshacer de forma segura los cambios en confirmaciones previas sin modificar el historial del repositorio, se usa `git revert`.

Para evitar problemas en el trabajo, conviene comprender las implicaciones y los riesgos de cada comando y elegir el enfoque adecuado en función de las necesidades y el flujo de trabajo del proyecto. **Con `git rm` borramos un archivo de Git, pero conservamos su historial de cambios.** Si no queremos borrar un archivo, sino dejarlo como está y actualizarlo más tarde, no deberíamos usar este comando en este commit. Usando `git reset HEAD`, movemos los cambios de Staging a Unstaged, pero mantenemos el HEAD en el repositorio con los últimos cambios que confirmamos. Así, no perdemos nada relevante.

La imagen a continuación ilustra como se cambian los estados del repositorio para los comandos vistos en la presente sección.

Git tree movements visualized



Merge de ramas

Producir una nueva rama se conoce como checkout. La unión de dos ramas se conoce como merge. Cuando fusionas estas ramas con el código principal, tu código se fusiona en una nueva versión de la rama maestra (o principal) que ya tiene todos los cambios que aplicaste en tus experimentos o correcciones de errores.

No obstante hay que tener en cuenta que combinar estas ramas (merge) puede generar conflictos. Algunos archivos pueden ser diferentes en ambas ramas o simplemente, dos colaboradores han trabajado sobre la misma línea de código, Git es muy eficiente y puede intentar fusionar estos cambios automáticamente, pero no siempre funciona. En algunos casos, somos nosotros los que tenemos que resolver estos conflictos a mano.

Antes de realizar un merge, debes **estar en la rama donde deseas aplicar los cambios**. Supongamos que estás en la rama main y deseas combinar los cambios de la rama feature. Con el comando merge, Git intenta combinar la rama feature con la rama main.

- **git checkout** main
- **git merge** feature

Si hay conflictos durante el merge, Git te notificará y pausará el proceso hasta que resuelvas esos conflictos. Los archivos con conflictos tendrán marcadores especiales en el código para ayudarte a identificar las diferencias. Para resolver conflictos, abre los archivos en un editor, selecciona las partes del código que quieres conservar, y elimina los marcadores.

Por otra parte, si queremos ver el historial de merges ejecutamos lo siguiente:

- **git log --merges**

Si durante el merge decides que no quieres continuar con el proceso, puedes abortarlo usando:

- `git merge --abort`

Repositorios Remotos y GitHub

Conceptos básicos de Git y repositorios remotos

Cuando empiezas a trabajar en un entorno local, el proyecto sólo vive en tu ordenador. Esto significa que no hay forma de que otros miembros del equipo trabajen en él. Para solucionar esto, utilizamos servidores remotos: un nuevo estado que deben seguir nuestros archivos para conectarse y trabajar con equipos de cualquier parte del mundo.

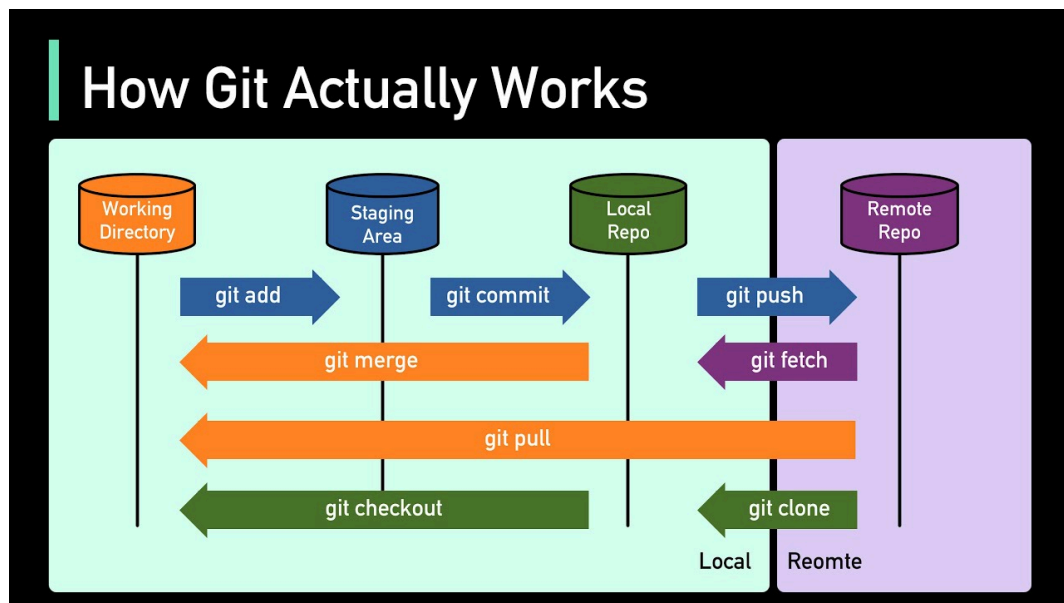
Estos servidores remotos pueden estar alojados en **GitHub**, **GitLab**, **BitBucket**, entre otros. Lo que harán será guardar el mismo repositorio que tienes en tu ordenador y darnos una URL con la que todos podremos acceder a los archivos del proyecto. Entonces el equipo puede descargarlos, hacer cambios y enviarlos de vuelta al servidor remoto para que otras personas puedan ver los cambios, comparar sus versiones y crear nuevas propuestas para el proyecto.

Comandos para el trabajo remoto con Git:

- **`git remote add origin <url_del_repositorio>`**: Conecta tu repositorio local con un repositorio remoto.
- **`git remote add another_origin_name <url_del_repositorio>`**: Conecta tu repositorio local con un origen remoto que has llamado `another_origin_name`.
- **`git clone url_desde_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`**: Después de hacer `git add` y `git commit`, debemos ejecutar este comando para enviar los cambios al servidor remoto.
- **`git push origin <nombre_de_la_rama>`**: Enviar tus cambios locales a una rama específica en el repositorio remoto.
- **`git push -u origin <nombre_de_la_rama>`**: Si es la primera vez que haces push a una rama, y deseas que se cree una rama remota.
- **`git fetch`**: Lo utilizamos para obtener actualizaciones del servidor remoto y guardarlas en nuestro repositorio local (en caso de que las haya, claro).
- **`git fetch origin <nombre_de_la_rama>`**: Trae al repositorio local una rama específica sin hacer un merge automático.
- **`git pull`**: Se usa para hacer un fetch y merge al mismo tiempo dentro del repositorio local.
- **`git pull origin <nombre_de_la_rama>`**: Obtener y fusionar automáticamente los cambios de una rama remota en tu rama actual.
- **`git checkout main` seguido de `git merge origin/feature-branch`**: Hace merge entre la rama local principal y la rama remota especificada (en este caso `feature-branch`).
- **`git remote remove origin`**: Elimina un origen remoto.

- **git remote -v:** Muestra que repositorios remotos están conectados al repositorio local.

Con los conceptos anteriores en mente, el flujo de trabajo extendido a repositorios remotos de Git es el siguiente:

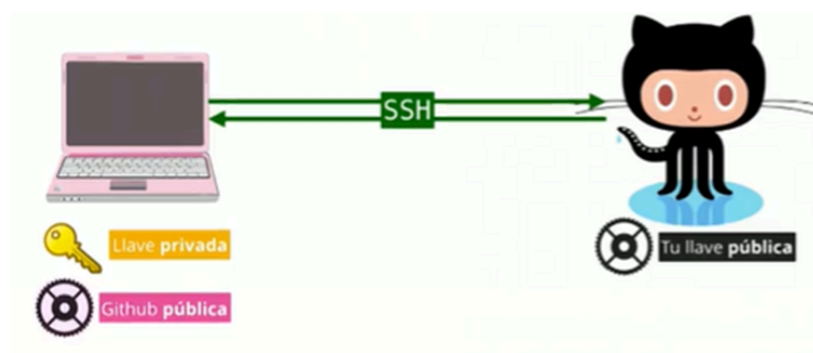


Llaves públicas y privadas

Las llaves públicas y privadas también conocidas como cifrado asimétrico de un solo camino, permiten enviar mensajes privados entre varios nodos los cuáles permiten cifrar y descifrar archivos, mensajes de manera que la información pueda ser compartida a través de un canal de comunicación sin que terceros mal intencionados puedan interceptarla y usarla.

- Tanto el receptor como el emisor del mensaje deben generar tanto una llave pública y privada.
- Ambas personas pueden compartir la llave pública a otras personas. La persona que quiere compartir un tipo de información puede usar la llave pública de la otra persona para cifrar los archivos y asegurarse que solo puedan ser descifrados con la llave privada del receptor.

Con esto en mente, es posible generar una llave pública y una llave privada para poder comunicarnos seguramente con GitHub a través del protocolo SSH.



Para generar estas llaves se ejecuta en la terminal de comandos los siguientes comandos, no obstante primero tenemos que ubicarnos en el directorio donde queremos que las llaves sean guardadas y allí, abrir la terminal:

- **ssh-keygen -t rsa -b 4096 -C 'micorreoelectronicodeinteres@gmail.com'**

La creación exitosa de un par de llaves SSH luce de la siguiente manera en la terminal.

```
Usuario@DESKTOP-QK5HS4B MINGW64 ~
$ ssh-keygen -t rsa -b 4096 -C [redacted]
Generating public/private rsa key pair.
Enter file in which to save the key (/c/Users/Usuario/.ssh/id_rsa):
Created directory '/c/Users/Usuario/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /c/Users/Usuario/.ssh/id_rsa
Your public key has been saved in /c/Users/Usuario/.ssh/id_rsa.pub
The key fingerprint is:
[redacted]
The key's randomart image is:
+---[RSA 4096]-----+
|
|..o..o..S o..
|++o+* *..
|o+=O.=.
|o.*oO*o+
|.oBXE*o.
+---[SHA256]-----+
```

Una vez generadas las llaves, tenemos que asegurarnos de que estas sean reconocidas por el ambiente local de nuestro ordenador para poder usarlas y así tener una conexión segura con GitHub, para ello, primero ejecutamos el siguiente comando el cual verifica que el protocolo SSH esté activado. De ser así, la consola arrojará “agent” que nos dice que en efecto el protocolo SSH está corriendo, y “pid” o identificador de proceso.

- **eval \$(ssh-agent -s)**

```
Usuario@DESKTOP-QK5HS4B MINGW64 ~
$ eval $(ssh-agent -s)
Agent pid 625
```

Finalmente, agregamos nuestra llave privada a las variables de entorno de nuestro ordenador, para ello, ejecutamos el siguiente comando. Nota a demás, que para tener una sana conexión con github es recomendable usar **el mismo correo de tu cuenta de github para crear las llaves SSH**.

- **ssh-add ruta-donde-guardamos-la-llave**

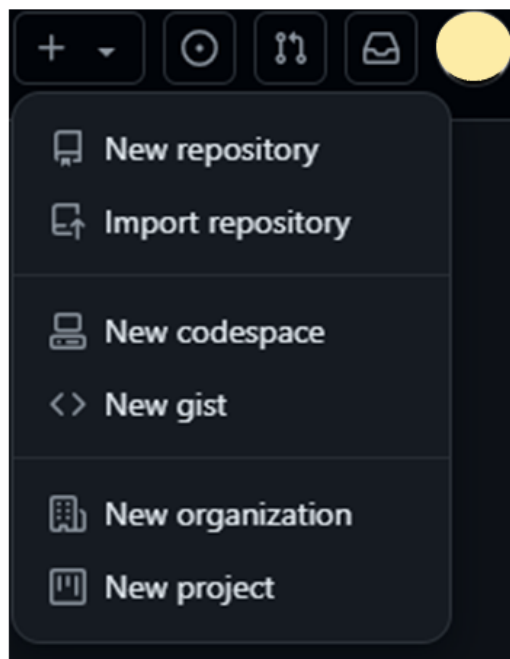
```
Usuario@DESKTOP-QK5HS4B MINGW64 ~
$ ssh-add ~/.ssh/id_rsa
Identity added: /c/Users/Usuario/.ssh/id_rsa ([redacted])
```

GitHub

Git Hub es un sitio web que funciona como un gran servidor de Git el cuál funciona a su vez como una interfaz “gráfica” de nuestros repositorios. A demás, es una de las herramientas colaborativas más importantes del desarrollo de software ya que funciona como un portafolio de desarrollo que permite a múltiples entidades conocer tu trabajo y establecer una plataforma de colaboración.

Una vez hemos creado una cuenta de GitHub, veremos las siguientes opciones dentro de la esquina superior derecha.

- New repository: Crea un nuevo repositorio vacío.
- Import repository: Importa un repositorio desde otros sistemas incluyendo sistemas que no están basados en Git.
- New Gist: Nuevo Script de código.
- New Project: Nuevo proyecto, colección de repositorios.
- New Organization: Nueva organización registrada a tu nombre, colección de proyectos.



Para poder generar un repositorio en la nube, primero tenemos que crearlo dentro de GitHub en **New Repository**, como se muestra a continuación.

Create a new repository
A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)

Required fields are marked with an asterisk (*).

Owner * Repository name *

Great repository names are short and memorable. Need inspiration? How about [super-duper-carnival](#)?

Description (optional)

☒ **Public**
Anyone on the internet can see this repository. You choose who can commit.

☐ **Private**
You choose who can see and commit to this repository.

Initialize this repository with:
☐ **Add a README file**
This is where you can write a long description for your project. [Learn more about READMEs.](#)

☐ **Add a README file**
This is where you can write a long description for your project. [Learn more about READMEs.](#)

Add .gitignore
.gitignore template: **None**
Choose which files not to track from a list of templates. [Learn more about ignoring files.](#)

Choose a license
License: **None**
A license tells others what they can and can't do with your code. [Learn more about licenses.](#)

📘 You are creating a public repository in your personal account.

Create repository

Una vez creado nuestro repositorio en GitHub, se verá de la siguiente manera:

AsorKy / UPTC_Diplomado_en_Ciencia_de_Datos_Cohorte_I_2024

Code Issues Pull requests Actions Projects Wiki Security Insights Settings

UPTC_Diplomado_en_Ciencia_de_Datos_Cohorte_I_2024 Public

Set up GitHub Copilot
Use GitHub's AI pair programmer to autocomplete suggestions as you code.
[Get started with GitHub Copilot](#)

Add collaborators to this repository
Search for people using their GitHub username or email address.
[Invite collaborators](#)

Quick setup — if you've done this kind of thing before
Set up in Desktop or **HTTPS** SSH `https://github.com/AsorKy/UPTC_Diplomado_en_Ciencia_de_Datos_Cohorte_I_2024.git`
Get started by [creating a new file](#) or [uploading an existing file](#). We recommend every repository include a [README](#), [LICENSE](#), and [.gitignore](#).

...or create a new repository on the command line

```
echo "# UPTC_Diplomado_en_Ciencia_de_Datos_Cohorte_I_2024" >> README.md
git init
git add README.md
git commit -m "first commit"
git branch -M main
git remote add origin https://github.com/AsorKy/UPTC_Diplomado_en_Ciencia_de_Datos_Cohorte_I_2024.git
git push -u origin main
```

Quick setup — if you've done this kind of thing before
Set up in Desktop or **HTTPS** SSH `https://github.com/AsorKy/UPTC_Diplomado_en_Ciencia_de_Datos_Cohorte_I_2024.git`
Get started by [creating a new file](#) or [uploading an existing file](#). We recommend every repository include a [README](#), [LICENSE](#), and [.gitignore](#).

...or create a new repository on the command line

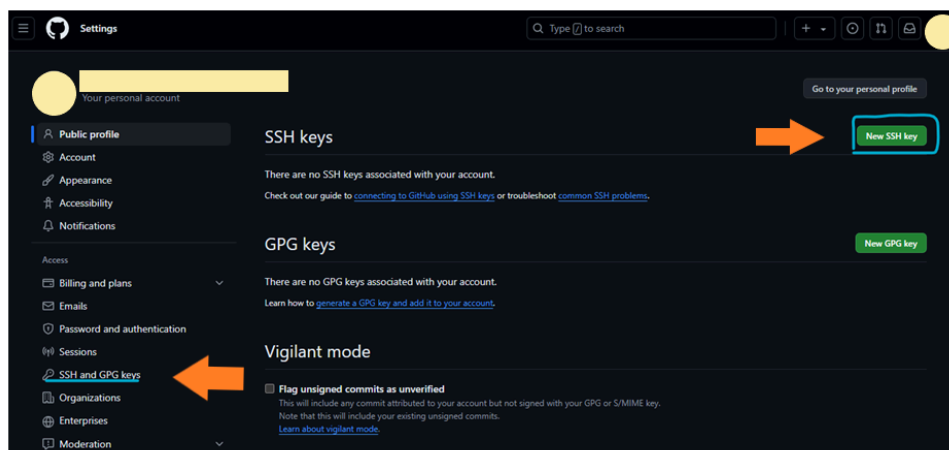
```
echo "# UPTC_Diplomado_en_Ciencia_de_Datos_Cohorte_I_2024" >> README.md
git init
git add README.md
git commit -m "first commit"
git branch -M main
git remote add origin https://github.com/AsorKy/UPTC_Diplomado_en_Ciencia_de_Datos_Cohorte_I_2024.git
git push -u origin main
```

...or push an existing repository from the command line

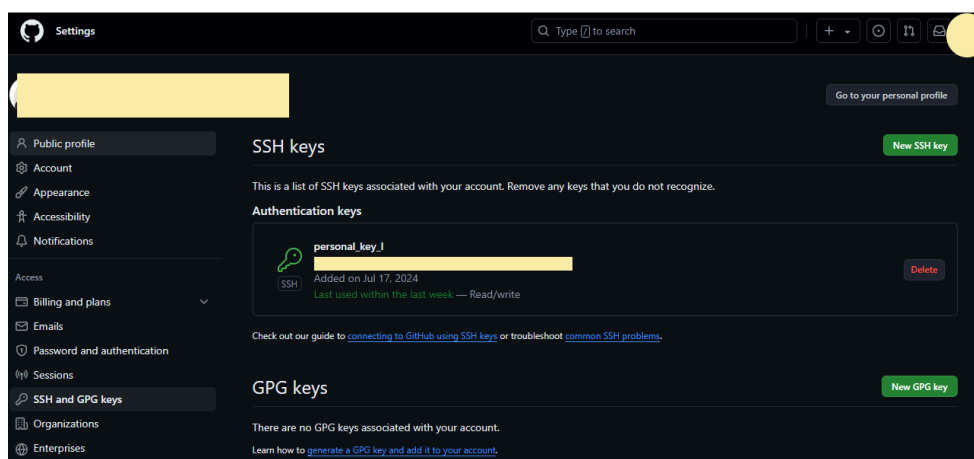
```
git remote add origin https://github.com/AsorKy/UPTC_Diplomado_en_Ciencia_de_Datos_Cohorte_I_2024.git
git branch -M main
git push -u origin main
```

Debido a que nuestra intención es subir nuestro código al nuevo repositorio, podemos hacerlo mediante la interfaz gráfica de GitHub o mediante una conexión HTTPS haciendo uso del url que nos presenta el repositorio y usando los comandos para repositorios remotos mostrados en esta sección unos párrafos arriba. No obstante, vamos a hacer esto de manera profesional mediante el uso de nuestras llaves públicas y privadas de manera que podamos realizar esta acción mediante una conexión segura. Para ello, ejecutamos los siguientes pasos ilustrados en las imagenes siguientes:

- Nos dirigimos a Configuraciones (panel derecho al seleccionar nuestra foto de perfil).
- Seleccionamos la opción SSH and GPG keys
- Seleccionamos la opción New SSH key
- Ingresamos nuestras credenciales.
- GitHub nos pedirá ingresar la llave, así que nos dirigimos a la carpeta en nuestro ordenador donde la hemos guardado y abrimos el archivo de la **llave pública**, **NO** la llave privada. Dentro de la carpeta donde hemos guardado las llaves, la llave privada es aquel archivo de texto plano que al abrirlo explícitamente dice “-----BEGIN OPENSSH PRIVATE KEY-----”, así pues, seleccionamos el archivo **id_rsa** que al abrirlo con un editor de texto, empieza por **ssh-rsa**, ésta es nuestra llave publica que debemos copiar y pegar en github.



Una vez sea aceptada la llave pública de nuestro ordenador, GitHub se verá de la siguiente manera:



Una vez establecida una conexión segura SSH, haremos un push de un proyecto mediante el siguiente comando:

- **git remote add origin <url HTTPS o SSH del repositorio creado>**
- **git branch main**
- **git add .**
- **git commit -m 'Mi primer commit de mi proyecto remoto'**
- **git push origin main**

Una vez el push ha sido exitoso, si refrescamos la página de nuestro repositorio, podremos ver los archivos de nuestro repositorio local pero ahora en la nube.

