Git y GitHub

Introducción a Git

¿Qué es un sistema de control de versiones?

El SCV o VCS (por sus siglas en inglés) es un **sistema que registra los cambios realizados sobre un archivo o conjunto de archivos a lo largo del tiempo**, de modo que puedas llevar el historial del ciclo de vida de un proyecto, comparar cambios a lo largo del tiempo, ver quién los realizó o revertir el proyecto entero a un estado anterior. Cualquier tipo de archivo que se encuentre en un ordenador puede ponerse bajo control de versiones.

¿Qué es Git?

Git es un SCV distribuido, 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. Git está optimizado para guardar todos estos cambios de forma atómica e incremental.
Se obtiene su mayor eficiencia con archivos de texto plano, ya que con archivos binarios no puede guardar solo los cambios, sino que debe volver a grabar el archivo completo ante cada modificación, por mínima que sea, lo que hace que incremente demasiado el tamaño del repositorio.

> **"Guardar archivos binarios** en el repositorio de git **es una mala práctica**, solo debería guardarse archivos pequeños (como logos) que no sufran casi modificaciones durante la vida del proyecto. Los binarios deben guardarse en un CDN**"**

¿Qué es Github?

Es una plataforma de desarrollo colaborativo (forja) para alojar proyectos utilizando el sistema de control de versiones Git. Se utiliza principalmente para la creación de código fuente de programas de computadora.

Github puede considerarse como la red social de código para los programadores y en muchos casos es visto como tu curriculum vitae.

Comandos y Conceptos Básicos de Git

Las tres secciones principales de un proyecto de Git

- El directorio de Git (Git Directory, Repository)
- El directorio de trabajo (Working Directory)
- El área de preparación (Staging Area)

https://learngitbranching.js.org/?locale=es_AR juego online para practicar

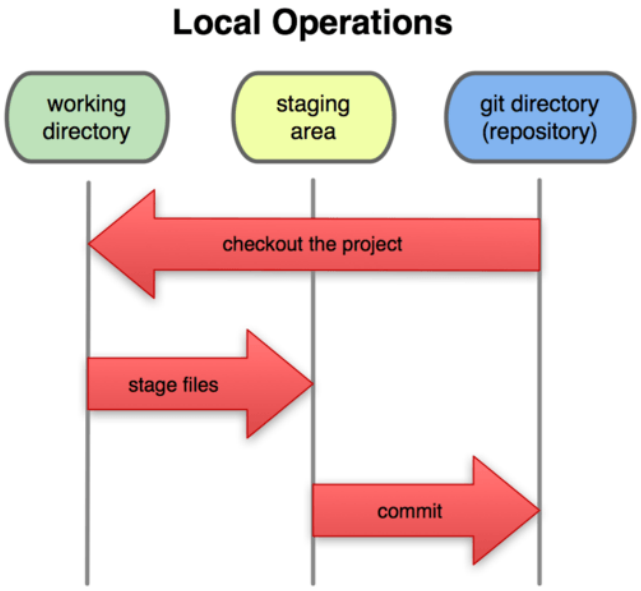
Staging Area y Git Directory

Al ejecutar el comando "`git init`" (comando para iniciar un repositorio git) ocurren dos cosas:

- Se crea una carpeta .git. El cual es el repositorio local donde Git almacena los metadatos y la base de datos de objetos para el proyecto. Es la parte más importante de Git, y **es lo que se copia cuando clonas** un repositorio desde otro ordenador.
- Se crea un archivo sencillo que define el staging area, generalmente está contenido en el directorio de Git, que almacena información acerca de lo que va a ir en tu próxima confirmación.

Ciclo básico de trabajo en Git

- Se modifica una serie de archivos en el directorio de trabajo.
- Se preparan los archivos añadiéndolos al área de preparación o staging. '`git add`'
- Se confirman los cambios: las instantáneas de los archivos que están en el área de staging se almacenan de forma permanente en el directorio de Git. '`git commit`'



Estados de un archivo

Commited: Si una versión concreta de un archivo está en el directorio de Git, se considera confirmada.
Staged: Si ha sufrido cambios desde que se obtuvo del repositorio, pero ha sido añadida al área de preparación, está preparada.
Modified: Y si ha sufrido cambios desde que se obtuvo del repositorio, pero no se ha preparado, está modificada.

Git Cheat Sheet

REBELLABS by ZEROTURNAROUND

Create a Repository

From scratch -- Create a new local repository
```
$ git init [project name]
```

Download from an existing repository
```
$ git clone my_url
```

Observe your Repository

List new or modified files not yet committed
```
$ git status
```

Show the changes to files not yet staged
```
$ git diff
```

Show the changes to staged files
```
$ git diff --cached
```

Show all staged and unstaged file changes
```
$ git diff HEAD
```

Show the changes between two commit ids
```
$ git diff commit1 commit2
```

List the change dates and authors for a file
```
$ git blame [file]
```

Show the file changes for a commit id and/or file
```
$ git show [commit]:[file]
```

Show full change history
```
$ git log
```

Show change history for file/directory including diffs
```
$ git log -p [file/directory]
```

Working with Branches

List all local branches
```
$ git branch
```

List all branches, local and remote
```
$ git branch -av
```

Switch to a branch, my_branch, and update working directory
```
$ git checkout my_branch
```

Create a new branch called new_branch
```
$ git branch new_branch
```

Delete the branch called my_branch
```
$ git branch -d my_branch
```

Merge branch_a into branch_b
```
$ git checkout branch_b
$ git merge branch_a
```

Tag the current commit
```
$ git tag my_tag
```

Make a change

Stages the file, ready for commit
```
$ git add [file]
```

Stage all changed files, ready for commit
```
$ git add .
```

Commit all staged files to versioned history
```
$ git commit -m "commit message"
```

Commit all your tracked files to versioned history
```
$git commit -am "commit message"
```

Unstages file, keeping the file changes
```
$ git reset [file]
```

Revert everything to the last commit
```
$ git reset --hard
```

Synchronize

Get the latest changes from origin (no merge)
```
$ git fetch
```

Fetch the latest changes from origin and merge
```
$ git pull
```

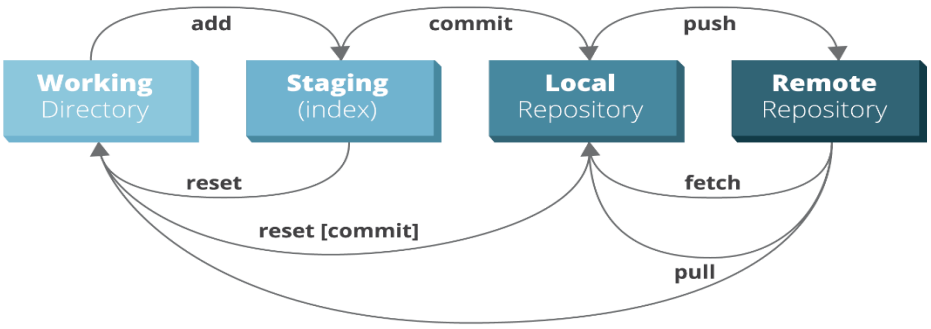
Fetch the latest changes from origin and rebase
```
$ git pull --rebase
```

Push local changes to the origin
```
$ git push
```

Finally!

When in doubt, use git help
```
$ git command --help
```

Or visit https://training.github.com/ for official GitHub training.



¿Qué es un Branch y cómo funciona un Merge en Git?

Todos los commits se aplican sobre una rama. Por convención se empieza a trabajar en la rama master (puede cambiarse el nombre de ser necesario) y se crean nuevas a partir de esta, para crear flujos de trabajo independientes.

Crear una nueva rama implica copiar un commit (de cualquier rama), pasarlo a otro lado (a otra rama) y continuar el trabajo de una parte específica de nuestro proyecto sin afectar el flujo de trabajo principal (que continúa en la rama master).

> ⚠ **Estándar o buena práctica:**
>
> - Todo lo que esté en la rama *"master"* va a producción.
> - Las nuevas features y experimentos se realizan en una rama *"development"* que se unen a master cuando estén listas.
> - Los issues o errores se solucionan en una rama *"hotfix"* para unirse a master tan pronto como sea posible.

Se puede crear todas las ramas y commits que se requieran para mantener ordenado el proyecto. Incluso puede aprovechar el registro de cambios de Git para crear ramas, traer versiones viejas del código, arreglarlas y combinarlas de nuevo para mejorar el proyecto.

Se debe tener en cuenta al combinar ramas los conflictos que puedan generarse. Git siempre intentará unir los cambios automáticamente, pero no siempre funciona bien, eventualmente se deben resolver los conflictos a mano.

Para asegurarte de que estás fusionando todas las ramas y cambios correctamente en `master`, aquí hay algunos pasos que puedes seguir:

1. Asegúrate de estar en la rama `master`: Puedes usar `git checkout master` para cambiar a la rama `master`.
2. Realiza un `git pull` para asegurarte de que tienes las últimas actualizaciones desde el repositorio remoto. Esto te ayudará a evitar conflictos innecesarios.
3. Utiliza `git merge` para fusionar tu rama de características en `master`. Por ejemplo: `git merge nombre-de-la-rama`.
4. Resuelve cualquier conflicto que surja durante la fusión.
5. Realiza un `git push` para enviar los cambios fusionados a tu repositorio remoto.

Comandos básicos

Crear repositorios y commits

- `git init`: inicializa un repositorio de GIT en la carpeta donde se ejecute el comando.
- `git add`: añade los archivos especificados al área de preparación (staging).
- `git commit -m "commit description"`: confirma los archivos que se encuentran en el área de preparación y los agrega al repositorio.
- `git commit -am "commit description"`: añade al staging area y hace un commit mediante un solo comando. (No funciona con archivos nuevos)
- `git status`: ofrece una descripción del estado de los archivos (untracked, ready to commit, nothing to commit).
- `git rm (. -r, filename) (--cached)`: remueve los archivos del index.
- `git config --global user.email <tu@email.com>`: configura un email.
- `git config --global user.name <Nombre como se verá en los commits>`: configura un nombre.
- `git config --list`: lista las configuraciones.

Analizar cambios en los archivos de un proyecto Git

- `git log`: lista de manera descendente los commits realizados.
- `git log --stat`: además de listar los commits, muestra la cantidad de bytes añadidos y eliminados en cada uno de los archivos modificados.
- `git log --all --graph --decorate --oneline`: muestra de manera comprimida toda la historia del repositorio de manera gráfica y embellecida.

- `git show filename`: permite ver la historia de los cambios en un archivo.
- `git diff <commit1> <commit2>`: compara diferencias entre en cambios confirmados.

Volver en el tiempo con branches y checkout

- `git reset <commit> --soft/hard`: regresa al commit especificado, eliminando todos los cambios que se hicieron después de ese commit.
- `git checkout <commit/branch> <filename>`: permite regresar al estado en el cual se realizó un commit o branch especificado, pero no elimina lo que está en el staging area.
- `git checkout -- <filePath>`: deshacer cambios en un archivo en estado modified (que ni fue agregado a staging)

Git rm y git reset

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.

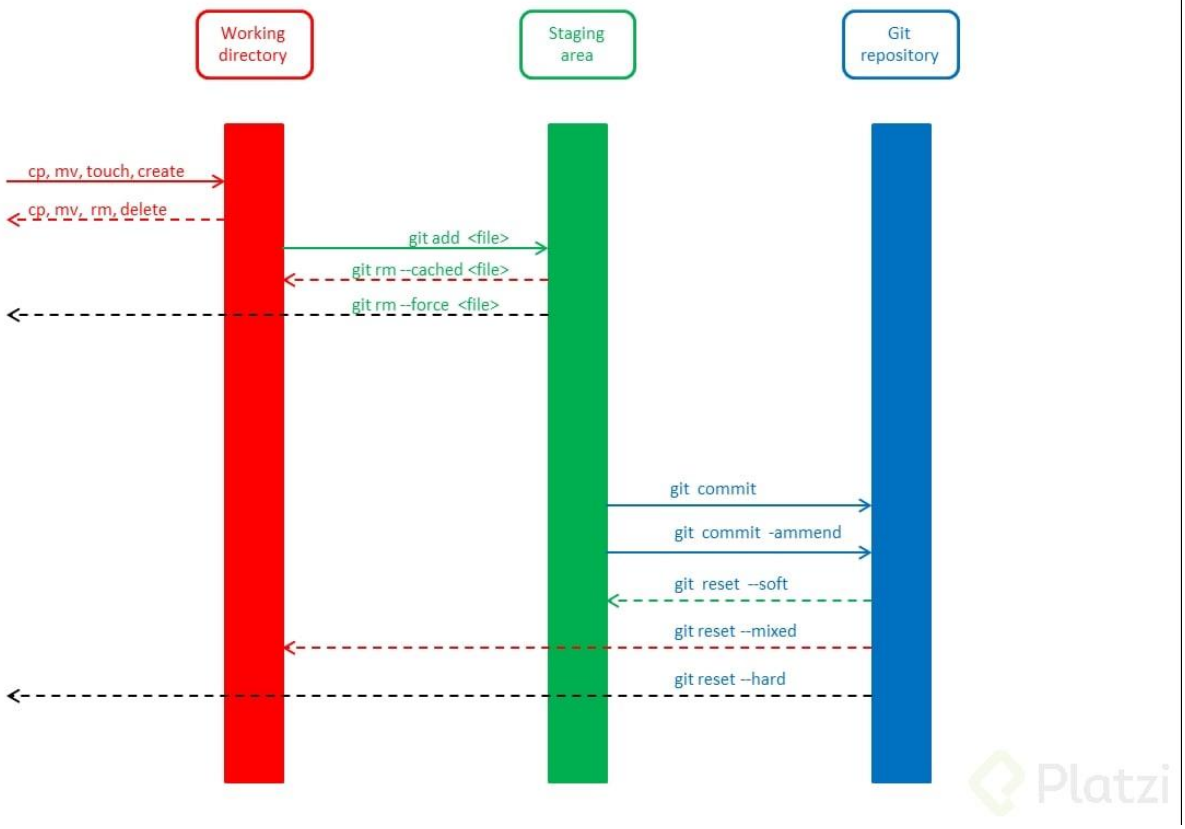
git rm no puede usarse así nomás. Se debe usar uno de los flags para indicar a Git cómo eliminar los archivos que ya no se necesitan en la última versión del proyecto:

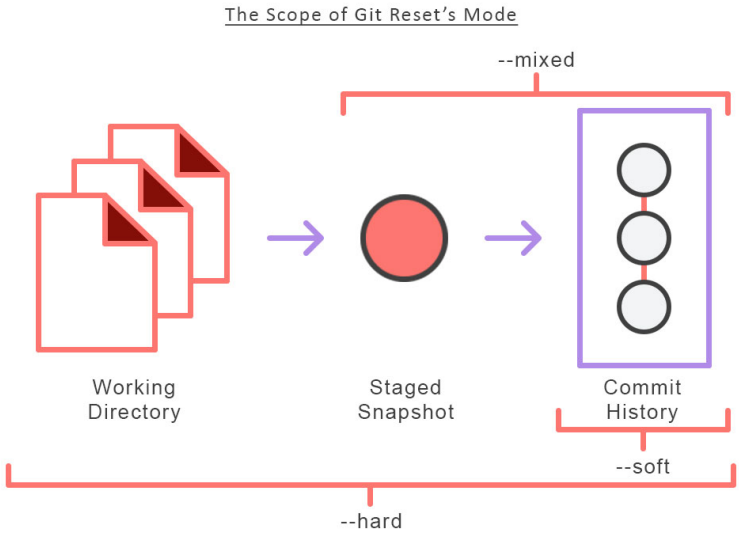
- `git rm --cached <archivo/s>`: Elimina los archivos del área de Staging y del próximo commit pero los mantiene en nuestro disco duro.
- `git rm --force <archivo/s>`: Elimina los archivos de Git y del disco duro. Git siempre guarda todo, por lo que podemos acceder al registro de la existencia de los archivos, de modo que podremos recuperarlos si es necesario (pero debemos usar comandos más avanzados).

git reset

Con git reset volvemos al pasado sin la posibilidad de volver al futuro.
Borramos la historia y la debemos sobreescribir.

- `git reset --soft`: Vuelve el branch al estado del commit especificado, manteniendo los archivos en el directorio de trabajo y lo que haya en staging considerando todo como nuevos cambios. Así podemos aplicar las últimas actualizaciones a un nuevo commit.
- `git reset --hard`: Borra absolutamente todo. Toda la información de los commits y del área de staging se borra del historial.





- `git reset HEAD`: No borra los archivos ni sus modificaciones, solo los saca del área de staging, de forma que los últimos cambios de estos archivos no se envíen al último commit. Si se cambia de opinión se los puede incluir nuevamente con `git add`.

Puedes hacer un commit desde esa versión y se actualizará como versión actual en la rama que te encuentres

Para retornar a la versión después de hacer el checkout…se aplica
`git checkout <branch> <archivo>`
ej : `git checkout master index.html`

por ejemplo para cambiar el comentario sobre un commit puedes seguir los siguientes pasos:

1. Utiliza git log para encontrar el hash del commit que deseas modificar. Luego, copia ese hash.
2. Utiliza el siguiente comando para navegar al commit que deseas modificar, reemplazando <commit-hash> con el hash que copiaste:`git checkout <commit-hash>`
3. Una vez que te encuentres en el commit correcto, utiliza el comando `git commit --amend -m "<nuevo-mensaje>"` para modificar el comentario del commit.

Ramas o Branches

Al crear una nueva rama se copia el último commit en esta nueva rama. Todos los cambios hechos en esta rama no se reflejarán en la rama master hasta que hagamos un merge.

- `git branch <new branch>`: crea una nueva rama.
- `git checkout <branch name>`: se mueve a la rama especificada.
- `git merge <branch name>`: fusiona la rama actual con la rama especificada y crea un nuevo commit de esta fusión.
- `git branch`: lista las ramas creadas.
- `git log --oneline --graph --color :` Enlista el grafo de las ramas

¿Cómo resolver conflictos en Git?

Al trabajar en dos o más ramas sobre las mismas líneas de código, ocurrirían conflictos a la hora de hacer merge. Git automáticamente nos especificará en nuestro código dónde se encuentran los conflictos.

Para resolver este problema debemos especificar la rama de donde queremos obtener el cambio, quedarnos con esas modificaciones y realizar un commit para completar el merge.

Trabajar con un repositorio remote

Recordar que antes de enviar a github las actualizaciones en local se debe crear la rama main(ya que la rama master hacía alusión a la esclavitud y decidieron cambiarla) debes crear la rama main:

```
git branch -m main
```

Si ya habías creado commits en la rama master puedes sencillamente cambiar el nombre de la rama master a main con:

git checkout master -- para cambiar a la rama master
`git branch -m main - cambia el nombre del branch actual (que es master a main)`

- `git remote add origin <link>`: enlaza el repositorio local con el repositorio remoto.
- `git config core.sparseCheckout true`: Para importar solo subdirectorios, no todo el repositorio
    - `echo "some/subdir/" >> .git/info/sparse-checkout`
    - `echo "another/sub/tree" >> .git/info/sparse-checkout`

- Importante: si hemos hecho cambios en el repositorio remote (suele ocurrir cuando se agrega README.md por defecto al crear el repositorio) o cualquier edición hecha en la nube, se deben llevar a local los cambios: `git pull origin <branch>`
- Este error típico es común que salga: `fatal: refusing to merge unrelated histories, debe ser corregido con el comando :`

  (Como ejemplo, cuando solo se hace pull al README.md suele salir:

```
$ git pull origin main --allow-unrelated-histories
From https://github.com/<usuario>/<nombre-repo>
 * branch            main       -> FETCH_HEAD
Merge made by the 'ort' strategy.
 README.md | 2 ++
 1 file changed, 2 insertions(+)
 create mode 100644 README.md
```

  A continuación es possible hacer el push o cualquier sincronización con github.

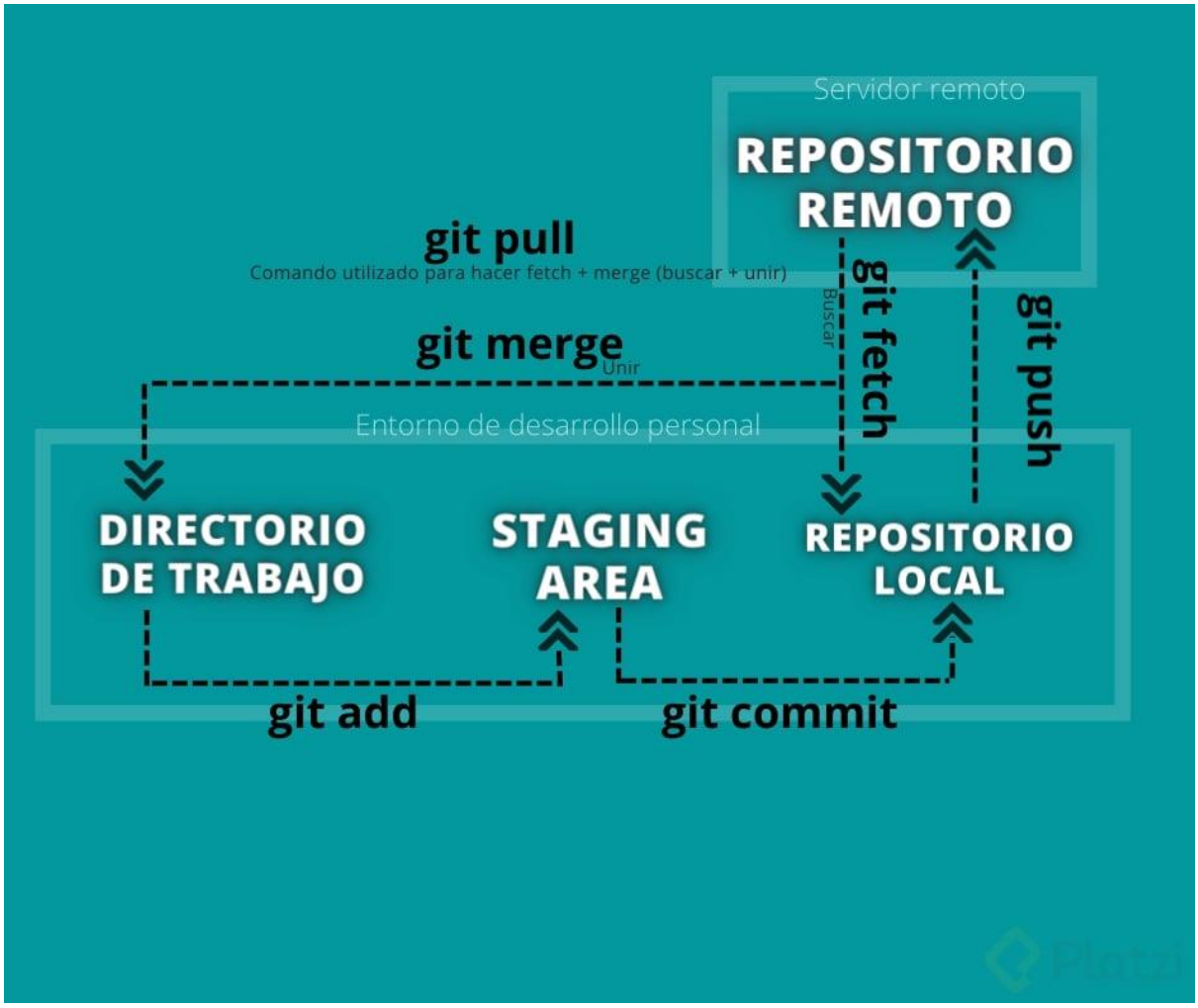
- `git push origin <branchName>`: exportar los archivos confirmados en el repositorio local al repositorio remoto.

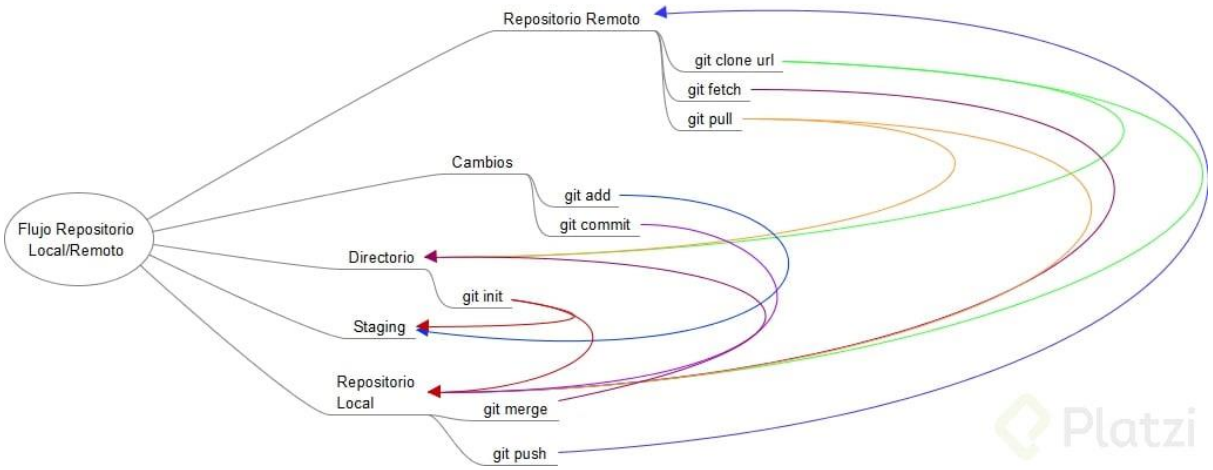
- `git pull origin <branchName>`: importa los archivos del repositorio remoto al repositorio local y al working directory.
- `git fetch`: importa los archivos remotos al repositorio local pero no al working directory.
- `git merge`: una vez hecho el git fetch, hace falta hacer un git merge para que los archivos importados aparezcan en el working directory.

En ocasiones el merge se hace en estado “Fast-forward” y no permite incluír comentario ni hará commit de integración, se recomienda hacer el merge con el comando:

```
git merge --no-ff -m "feature1 into develop" feature1
```

el cual indica un ejemplo de un merge de la rama feature1 into develop (se debe hacer checkout antes a la rama develop) y de esta manera se incluirá el comentario en el merge y se creará el commit





GitHub - Repositorios remotos

Por seguridad y practicidad, para trabajar con repositorios remotos lo ideal es no enviar las credenciales de loging cada vez que se transfiera información con la plataforma, para eso se configuran las llaves SSH, siendo el método más seguro.

Llaves SSH

1. Generar las llaves SSH. Si bien no es obligatorio, se recomienda proteger la llave privada con una contraseña cuando lo solicita el proceso de generación.
   `ssh-keygen -t rsa -b 4096 -C <tu@email.com>`
   -t rsa es el algoritmo elegido de cifrado (acrónimo de Rivest-Shamir-Adleman creadores del algoritmo)
   -b 4096 son los bits que tendrá la llave. 2048 suele ser suficiente pero con 4096 se extrema la seguridad.
   -C <comentario a elección>
2. Terminar de configurar según sistema operativo.
   a. En Windows y Linux:
      # Encender el "servidor" de llaves SSH local:
      `eval $(ssh-agent -s)`
      # Añadir la llave privada SSH a este "servidor":
      `ssh-add <ruta-a-la-llave-privada>`
   b. En Mac:
      # Encender el "servidor" de llaves SSH local:
      `eval "$(ssh-agent -s)"`
      # Para versiones de OSX superior a Mac Sierra (v10.12)
      # se debe crear o modificar un archivo "config" en la carpeta
      # del usuario con el siguiente contenido (respetar las mayúsculas):

```
Host *
     AddKeysToAgent yes
     UseKeychain yes
     IdentityFile
     ruta-a-la-llave-privada
```

```
# Añadir la llave privada SSH al "servidor" de llaves SSH local
# (en caso de error se puede ejecutar este mismo comando
# pero sin el argumento -K):
ssh-add --apple-use-keychain <ruta-a-la-llave-privada>
```

Conexión a GitHub con SSH

Luego de crear las llaves SSH se debe entregar la llave pública a GitHub para realizar la comunicación de forma segura y sin necesidad de escribir el usuario y contraseña.

Para esto entrar a la Configuración de Llaves SSH en GitHub, crear una nueva llave con el nombre deseado y el contenido de la llave **pública** de tu computadora.

Luego actualizar en nuestra pc la URL del repositorio remoto, cambiando la URL con HTTPS por la URL con SSH:

```
git remote set-url origin <url-ssh-del-repositorio-en-github>
```

Tags y versiones en Git y GitHub

Los tags o etiquetas permiten asignar versiones a los commits con cambios más importantes o significativos del proyecto.
En GitHub esto crea releases, versiones descargables del proyecto en ese preciso estado.

Comandos para trabajar con etiquetas:

- Crear un nuevo tag y asignarlo a un commit:
  ```
  git tag -a <nombre-del-tag> -m <mensaje del commit> <id-del-commit-al-que-asignar-la-etiqueta>
  ```
- Borrar un tag en el repositorio local:
  ```
  git tag -d nombre-del-tag
  ```
- Es possible hacer un tag a un commit recien creado escribiendo el commando siguiente justo después de la creación del commit:
  ```
  git tag <nombre_del_tag>
  ```

- Listar los tags de nuestro repositorio local:
  ```
  git tag
  ```
  Listar los tags indicando a qué commit se asignó cada uno
  ```
  git show-ref --tags
  ```
- Publicar un tag en el repositorio remoto:
  ```
  git push origin --tags
  ```
- Borrar un tag del repositorio remoto:
  ```
  git tag -d nombre-del-tag
  git push origin :refs/tags/nombre-del-tag
  ```

Manejo de ramas en GitHub

Puedes trabajar con ramas que nunca envias a GitHub, así como pueden haber ramas importantes en GitHub que nunca usas en el repositorio local.

- Crear una rama en el repositorio local:
  `git branch nombre-de-la-rama` o
  `git checkout -b nombre-de-la-rama`
- Publicar una rama local al repositorio remoto:
  `git push origin nombre-de-la-rama`

Se puede ver gráficamente el entorno y flujo de trabajo local con Git usando el comando `gitk`.

Flujo de trabajo profesional con Pull requests

En un entorno profesional normalmente se bloquea la rama master, se desarrolla en para enviar código a dicha rama pasa por un code review y luego de su aprobación se unen códigos.

Para realizar pruebas enviamos el código a un "staging server" (servidor de prueba), una vez pasadas las pruebas tanto del código como de la aplicación, se pasan al servidor de producción mediante un pull request (GitHub y Bitbucket) o merge request (GitLab).

Eliminar una rama

- Eliminar una rama local
  `git branch -d <localBranchName>`
- Eliminar una rama remota
  `git push origin --delete remoteBranchName`

Readme.md es una excelente práctica

README.md es una excelente práctica en los proyectos, md significa Markdown, es una especie de código que permite cambiar la manera en que se ve un archivo de texto.
Funciona en muchas páginas, por ejemplo la edición en Wikipedia es un lenguaje intermedio que no es HTML, no es texto plano, es una manera de crear textos formateados.

Datos a tener en cuenta para escribir un buen README.md

1. **Nombre:** Especificamos cómo se llama nuestro proyecto.
2. **Descripción:** es donde diremos para qué exactamente es el proyecto, qué problemas resuelve y cualquier información relevante.
3. **Instalación:** muestra los pasos específicos para instalar el proyecto. Por lo general se muestra un pedazo del código necesario para la instalación.
4. **Cómo usar:** describe rápidamente casos de uso en los cuales se puede usar el proyecto, además de mostrar funcionalidades.

5. **Cómo contribuir:** si es un proyecto open source se describe acá la forma en la que deberían crearse las contribuciones.
6. **Licencia:** muestra la licencia que tiene el proyecto.

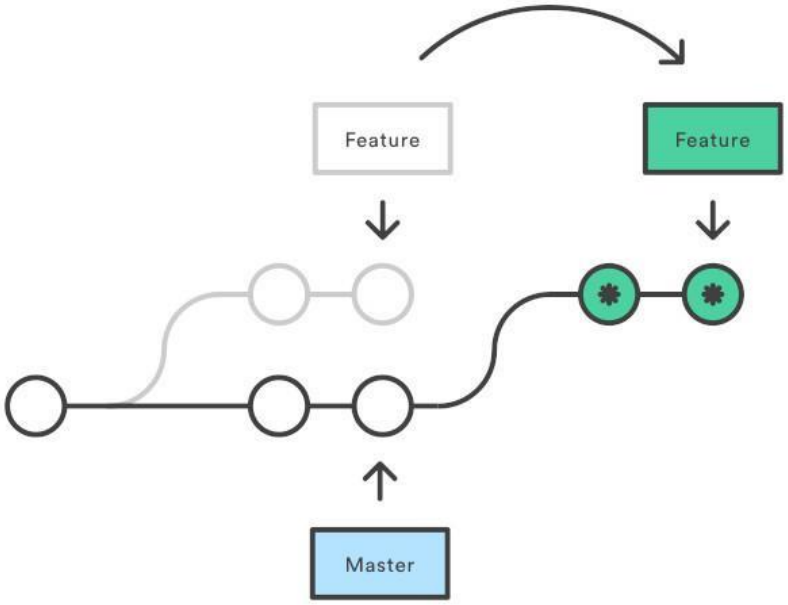
Sitio web público con GitHub Pages

GitHub tiene un servicio de hosting gratis llamado GitHub Pages, se puede tener un repositorio donde el contenido del repositorio se vaya a GitHub y se vea online.

Múltiples entornos de trabajo

Rebase: Reorganizando el trabajo realizado

Con rebase se puede recoger todos los cambios confirmados en una rama y ponerlos sobre otra.

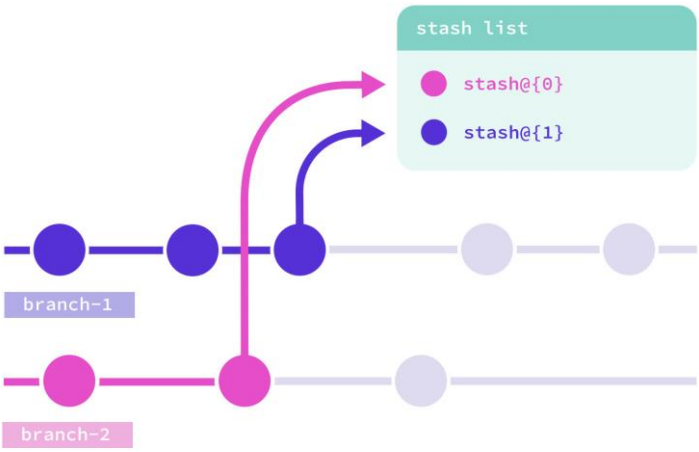


1. Cambiar a la rama que queremos traer los cambios
   `git checkout experiment`
2. Aplicar rebase para traer los cambios de la rama que queremos
   `git rebase master`

> **"rebase es una mala práctica** no debería usarse a menos que no quede otra opción.**"**

Stash: Guardar cambios en memoria y recuperarlos después

Sirve para cuando se necesita recordar el estado actual del directorio de trabajo y del índice, pero se requiere volver y limpiar el directorio de trabajo.



- `git stash`
  guarda las modificaciones locales en memoria y revierte el directorio de trabajo para coincidir con el estado del commit de HEAD.
  Es típico cuando se hacen cambios que no merecen una rama o no merecen un rebase, sino que simplemente se está probando algo y luego se quiere volver rápidamente a la versión anterior, que es la correcta.
- `git stash branch <nombre-nueva-rama>`
  Crea un nuevo branch con las modificaciones guardadas en memoria con stash.

Clean: Limpiar el proyecto de archivos no deseados

A veces se crean archivos durante la realización de un proyecto, que realmente no forman parte del directorio de trabajo y no se debería agregar.

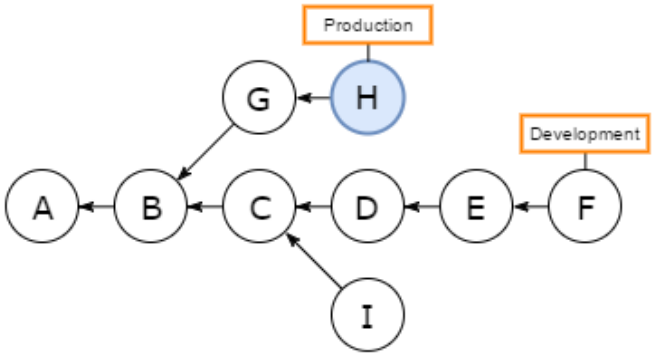
- Para saber qué archivos se borrará
  `git clean --dry-run`
- Para borrar todos los archivos listados (que no son carpetas)
  `git clean -f`

Cherry-pick: Traer commits viejos al head de un branch

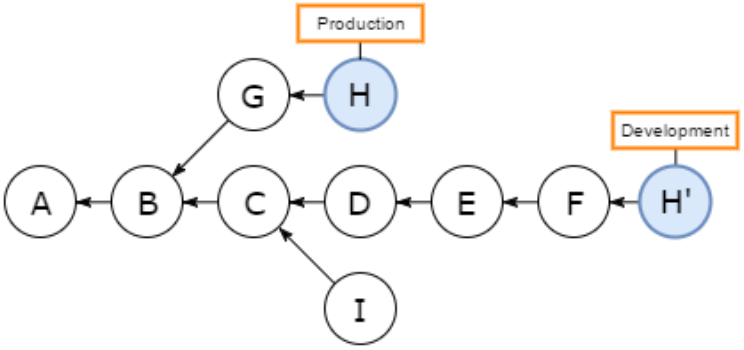
Cherry-pick trae las modificaciones realizadas en un commit específico de otra rama.

`git cherry-pick <IDCommit>`

Un caso de uso válido sería el siguiente:
Se encuentra un problema en la rama de producción, ésta tiene modificaciones hechas que no están en la rama de desarrollo (commit G), se desarrolla un fix de urgencia (commit H).



Si se requiere incorporar el fix a la rama de desarrollo pero **sin las modificaciones del commit G**, se realiza un cherry-pick del commit H a desarrollo (commit H').



> **"Cherry-pick suele ser una mala práctica** porque significa que estamos reconstruyendo la historia. Debe usarse como último recurso.**"**

Amend reconstruir commits

Amend (remendar - reconstruir) agrega cambios al último commit, tanto de archivos como del mensaje.

- `git add <archivos a agregar>`
  `git commit --amend`

Buscar en archivos y commits de Git con Grep y log

A medida que nuestro proyecto se hace grande vamos a querer buscar ciertas cosas.

- `git grep color`
  busca en todo el proyecto los archivos en donde está la palabra color.
- `git grep -n color`
  indicará en qué línea está la palabra color.
- `git grep -c color`
  indicará cuántas veces se repite la palabra color y en qué archivo.
- `git grep -c "<p>"`
  indicará cuántas veces se utiliza el atributo <p> de HTML

Reset y Reflog: "Úsese en caso de emergencia" 🧯

¿Qué pasa cuando todo se rompe y no sabemos qué está pasando?

- Volver al estado en que el proyecto funcionaba
  `git reset <HashDelHEAD>`
- Mostrar todos los cambios del HEAD.
  `git reflog`
- Mantener lo que haya en staging
  `git reset --soft <HashDelHEAD>`
- Resetear absolutamente todo incluyendo lo que haya en staging
  `git reset --hard <HashDelHEAD>`

> 🚑 **"reset es una mala práctica**. Debe ser el último recurso.**"**