Script taller de git /algo2

Version Control System (VCS)
Sistemas que permiten trackear (o mantener un registro de) los cambios realizados en el tiempo.

Git
Git es actualmente la implementación más popular de un sistema de control de versiones **distribuido**.

Git repositories
Un repositorio -o simplemente *repo*- de Git contiene la historia de una colección de archivos a partir de un directorio que llamaremos raíz (del repositorio). En el raíz encontraremos el archivo **.**gitignore, y el .git/config donde más tarde veremos algunas opciones de configuración para el repositorio.

Git init
Para crear repositorios nuevos, este comando crea automáticamente la carpeta .git donde se alojan archivos propios del sistema git y el archivo .git/config donde se configurarán varias opciones de nuestro repo.

```
$ mkdir tp-algo2-grupoXX
$ cd tp-algo2-grupoXX
$ git init
```

Git config --user
Ahora que tenemos nuestro repo listo para usar, es el momento de configurar algunas cuestiones básicas antes de empezar a versionar los archivos. Si usamos --global, estamos seteando estas opciones para todos los repos de nuestro filesystem, de lo contrario solo se configurarán para el repo donde corremos el comando.

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

.gitignore
En este archivo indicaremos las rutas que deseamos excluir de nuestro repositorio, es decir los archivos o directorios que no nos interesa *trackear.*

Por ejemplo si en nuestro .gitignore tenemos una línea con *.log, estaremos indicando que los archivos de extensión .log no van a ser trackeados en nuestro repo.

```
$ echo "*.log" > .gitignore
```

Crear repo en GitHub
Para trabajar en un proyecto con más personas, o bien trabajarlo en más de una computadora, vamos a crear un repo en GitHub para usarlo como repo remoto o **remote**, al cual iremos subiendo y bajando los cambios desde y a nuestro repo **local** (el que tenemos en nuestra computadora).
Cuando creamos un repo nuevo en GitHub veremos allí las indicaciones para linkearlo a un repo local en nuestra máquina.
Generalmente lo más fácil para linkear un repo de GitHub con un repo local es clonarlo, sin embargo para entender mejor lo que está sucediendo vamos a explicarlo paso a paso:

```
echo "# holaMundo" >> README.md
git init
git add README.md
git commit -m "first commit"
git remote add origin git@github.com:marin-h/holaMundo.git
git push -u origin master
```

Git clone

El proceso de copiar un repo Git existente usando las herramientas de Git se denomina clonado (cloning).
Luego de clonar un repo el usuario tiene el repositorio completo, es decir, con su historia (la de cada uno de sus archivos), en su máquina local.

```
$ git clone <url>
```

Git remote

Al clonar un repo, la url de este quedará configurado automáticamente en el repo local como **remote** de nombre "origin" en el archivo .git/config:

```
[remote "origin"]
    url = <url que usamos para clonar, puede ser http:// o ssh://>
```

Working tree, git status

Al clonar un repo, en nuestro filesystem tendremos la versión más reciente de la rama default del repositorio.
Esto quiere decir que nuestro árbol de trabajo -el **working tree**- corresponderá al checkout de último commit de la rama **master**, en nuestro caso. En caso de tener cambios hechos o no, hablaremos de un working tree limpio (clean) o sucio (dirty).

Esto lo podemos corroborar con el comando **git status:**

```
$ git status
On branch master
Your branch is up to date with 'origin/master'.

nothing to commit, working tree clean
```

Luego de agregar un archivo nuevo, corremos el mismo comando:

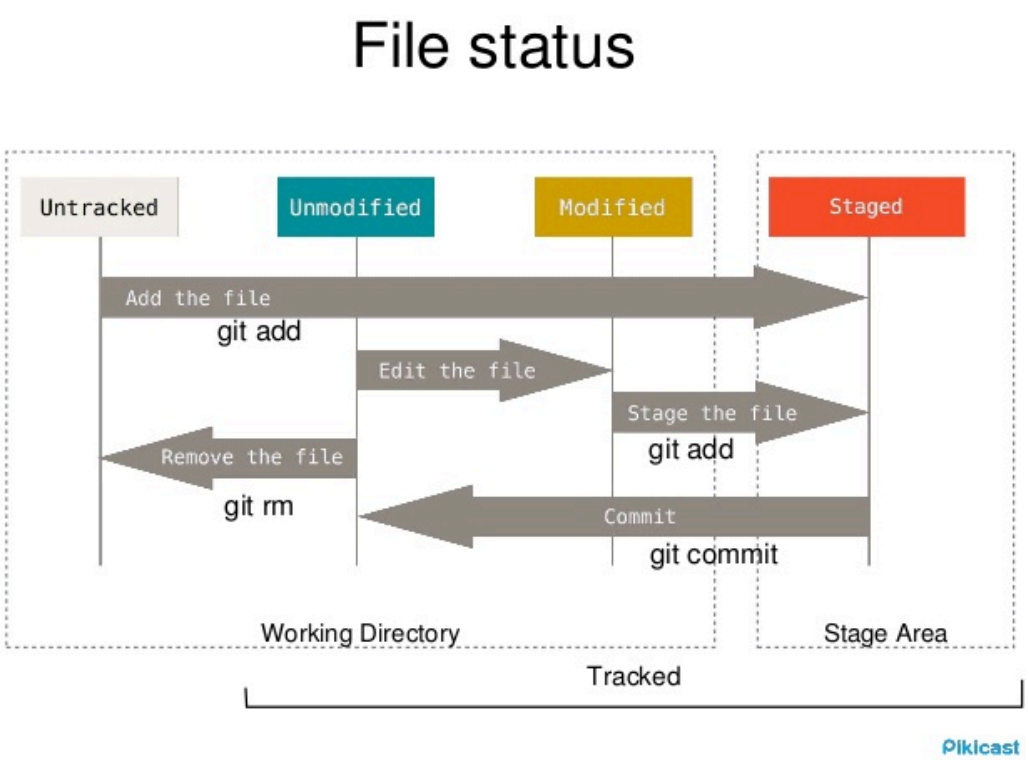
```
$ git status
On branch master
Your branch is up to date with 'origin/master'.

Untracked files:
  (use "git add <file>..." to include in what will be committed)
       archivoNuevo

nothing added to commit but untracked files present (use "git add" to track)
```

Estados
El usuario puede modificar los archivos en el working tree ya sea modificando archivos existentes, o bien creando/borrando archivos.
Un archivo en el working tree de un repo Git puede tener **distintos estados**. Esos estados son:

- New file: aún no fue agregado a ningún commit (git *ve* todos los archivos nuevos, salvo los del .gitignore)
- Staged: agregado para el próximo commit
- Modified: archivo que ya había sido agregado en un commit, ahora con modificaciones locales que aún no fueron agregadas a un nuevo commit.
- Removed: archivo que ya había sido agregado en un commit, ahora eliminado.

Además es posible diferenciar entre archivos:

- untracked: el archivo no está siendo trackeado por el repo Git aún.
- tracked: archivos que fueron modificados/agregados por un commit ya realizado.



Commit
Un objeto commit representa una versión de todos los archivos trackeados en el repositorio en el momento en el cual el commit se creó.
Luego de hacer cambios en el working tree, el usuario puede agregar sus cambios al repo local para persistirlos, con estos dos pasos:

- Agregar los cambios seleccionados al área de *staging* (conocida como *index*) con el comando **git add**.
- Hacer un commit en el repo local con los cambios agregados a staging usando el comando **git commit**.

```
$ git add <file>
$ git commit -m <file> "Commit msg"
```

Commit hash
Un commit es identificable por su hash, el cual es calculado en base al contenido del commit y su metadata (mensaje, autor, etc).
Este hash nos va a permitir interactuar con commits en la línea de comandos, para verlos, rastrearlos, compararlos, etc.

Ejemplos de comandos que utilizan el hash de un commit:

```
$ git show <hash>
$ git diff <hash>...<hash>
$ git checkout <hash>
```

Además, con **git blame** podemos inspeccionar qué commit originó cada línea de código de un archivo trackeado:

```
$ git blame <file>
```

Pull y Push, sincronizando el remoto
Git permite al usuario sincronizar su repo local con otros repositorios (**remotes**). Usuarios con permisos suficientes podrán enviar nuevas versiones desde su repo local a repositorios remotos mediante la operación *push.* También podrán traerse cambios de otros repositorios a su repositorio local utilizando las operaciones *fetch* y *pull*.

Para traer cambios del repo remoto:

```
$ git pull <remote> <rama>
```

Para subir cambios del repo remoto:

```
$ git push <remote> <rama>
```

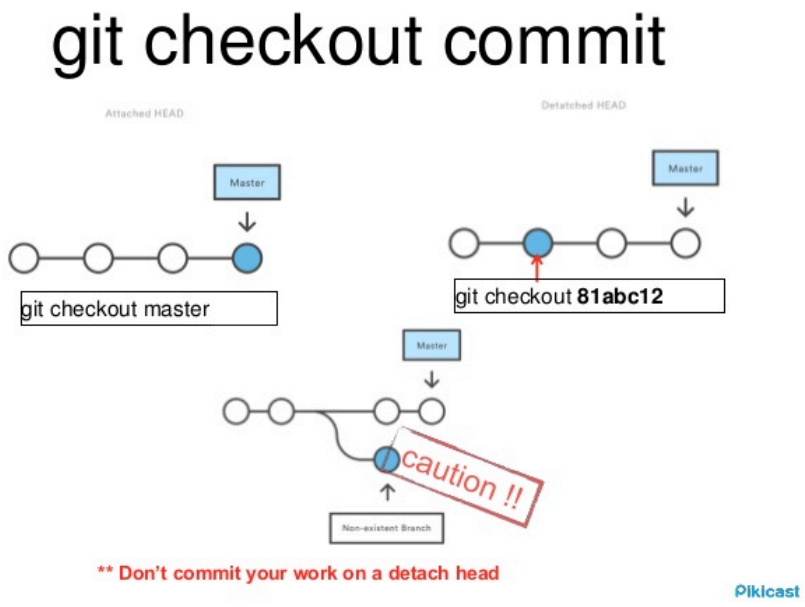
Stash
Cuando tenemos cambios en nuestro working tree, e intentemos hacer git pull para traernos lo último del repo remoto, Git no nos permitirá pullear. Esto es porque nuestros cambios del working tree serían sobreescritos con la operación de merge que implica el pull.
Si se da esta situación, es posible salvarla de manera elegante usando **git stash** (para pasar nuestros cambios a la pila de stash y así limpiar el working tree), luego **git pull**, y luego **git stash pop.** De esa manera volvemos a tener nuestros cambios en el working tree luego de pullear.

```
$ git stash
$ git pull <remote> <rama>
$ git stash pop
```

Checkout
Usando el comando **git checkout** podemos movernos entre distintos branches o commits, haciendo que nuestro working tree refleje el estado de una rama o de un determinado commit.

Ramas
Git permite el uso de ramas (branches), es decir que se puede trabajar sobre diferentes versiones de la colección de archivos. Las ramas permiten trabajar simultáneamente, cambios independientes entre sí dentro de nuestros archivos, para luego combinarlos si se quiere (para esto vamos a ver en un rato rebase, merge, cherry-pick).
El ejemplo más básico de esto son las ramas master y develop. Con un **branching model** típico en **master** generalmente vamos a tener el código más estable, por ej. El que ya está en producción disponible para los usuarios. En **develop** trabajaremos a diario para desarrollar progresivamente los nuevos features, o mejoras de la aplicación que construimos. En algún momento, vamos a querer que los cambios hechos en develop sean replicados en **master**, para esto generalmente usaremos **git merge**.

HEAD, detached HEAD
HEAD es una referencia simbólica al punto sobre el cual se para el working tree. Al cambiar de rama, HEAD apuntará al puntero de la rama actual, el cual termina apuntando a un commit. Si se hace checkout a un commit en particular, HEAD apuntará a este commit directamente (en modo *detached HEAD*). En ese estado, no se está trabajando sobre ninguna rama, por lo que los commits que se hagan no quedarán asociados a ninguna rama (y serán fácilmente perdidos).



Merge
El comando **git merge** se utiliza para aplicar los cambios de una rama en otra rama. Generalmente se realiza desde una rama 'padre' para mergear los cambios de alguna de sus ramas 'hijas'.

Merge conflicts

Conflictos durante el proceso de merge pueden darse si hay modificaciones realizadas en las mismas líneas de código en ambas ramas. En ese caso, Git se ocupará de avisarnos del merge conflict y nos dejará el merge a medio camino para que manualmente decidamos qué código debe quedar y cual se eliminará.
Por ejemplo, al presentarse un conflicto de merge en un archivo, al inspeccionar el contenido del archivo veremos algo así:

```
<<<<<<< HEAD
my version
=======
the other version
>>>>>>> other branch
```

Esta es la forma que usa Git para indicar los detalles del conflicto:

- Entre las marcas <<<<<<< y =======, estarán los cambios propios de tu rama actual.
- Entre las marcas ======= y >>>>>>>, estarán los cambios de la rama que se intenta mergear.
- Convenientemente, luego de las marcas <<<<<<< y >>>>>>>, habrá 'hints' acerca de a qué commit pertenece esa parte del conflicto, siendo HEAD por supuesto la referencia a tu revisión actual.

Tags
El comando **git tag** va a permitirnos etiquetar o ponerle un nombre legible a un commit, para luego poder hacer checkout, diff o show usando ese nombre.

PR, review
Luego de realizar cambios en un branch específico, es posible abrir un pull request para solicitar a tus colaboradores o al administrador del repositorio que revisen tus cambios (review) antes de hacer el merge.
A través de distintas reglas también es posible establecer un número mínimo de reviews y/o approvals de un PR antes de que sea posible efectuar el merge.

Comandos git

- Status
- Git diff
- Add
- Rm
- Commit
- Log, mencionar: —graph
- Checkout
- Merge, merge conflicts
- Rebase, mencionar: —interactive, squash
- Cherry-pick
- Stash
- Show
- Blame

Buenas prácticas

- Commits chiquitos

- Mensajes de commit cohesivos, que sigan cierta convención
- Tener claro el branching model

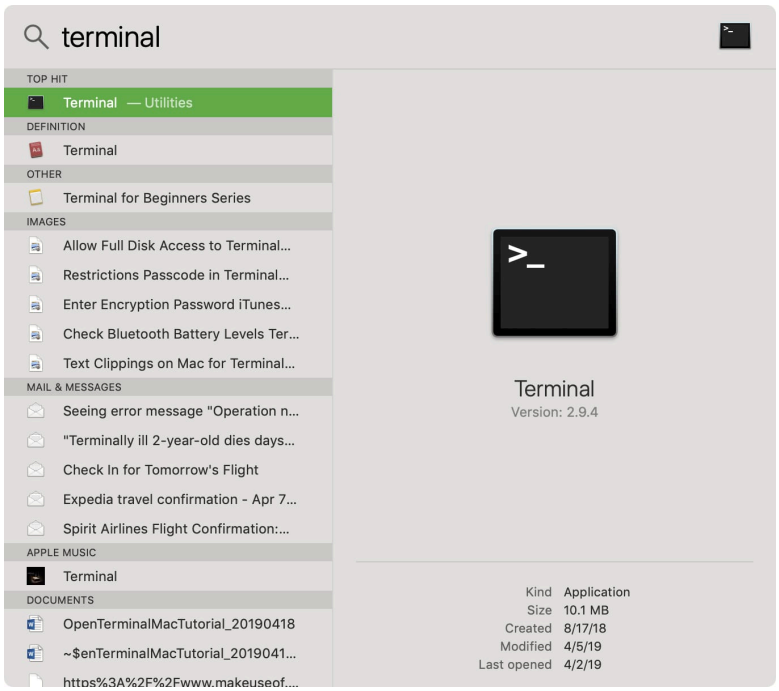
Más recursos
Git-it - Desktop App for Learning Git
Learn Git Branching

Documentación
https://www.atlassian.com/es/git (en español)
https://git-scm.com/doc

Windows
Git bash para Windows
https://gitforwindows.org/

OSX y Ubuntu
En OSX y Ubuntu *generalmente* git viene instalado por defecto, solo tienen que abrir una terminal y usarlo.
Si no estuviera instalado, pueden seguir los pasos de https://git-scm.com/downloads.

Abrir una terminal en OSX
Usando spotlight (command + espacio) y tipear term:



Abrir una terminal en Ubuntu/Debian:
- Abrir desde el menú de aplicaciones, tipeando "term"y eligiendo la opción que aparece.
- Con el shortcut Ctrl - Alt + T

Para chequear desde cualquier terminal si git está instalado correctamente:

```
$ git --version
```

Si la salida es la versión de git, por ej "git version 2.25.0", todo bien :)
De lo contrario, hay que revisar por qué no se instaló correctamente, reinstalarlo, etc.