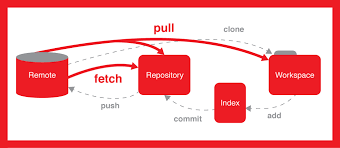
EJERCICIO 8: GIT





Andrés Cuevas Rodríguez

**¿Qué es git y qué debemos saber sobre 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. Su propósito es llevar registro de los cambios en archivos de computadora y coordinar el trabajo que varias personas realizan sobre archivos compartidos.

Al principio, Git se pensó como un motor de bajo nivel sobre el cual otros pudieran escribir la interfaz de usuario o front end como Cogito o StGIT. 3​Sin embargo, Git se ha convertido desde entonces en un sistema de control de versiones con funcionalidad plena. 4​Hay algunos proyectos de mucha relevancia que ya usan Git, en particular, el grupo de programación del núcleo Linux.

Entre las características más relevantes se encuentran:

-Fuerte apoyo al desarrollo no lineal, por ende rapidez en la gestión de ramas y mezclado de diferentes versiones. Git incluye herramientas específicas para navegar y visualizar un historial de desarrollo no lineal. Una presunción fundamental en Git, es que un cambio será fusionado mucho más frecuentemente de lo que se escribe originalmente, conforme se pasa entre varios programadores que lo revisan.

-Gestión distribuida. Al igual que Darcs, BitKeeper, Mercurial, SVK, Bazaar y Monotone, Git le da a cada programador una copia local del historial del desarrollo entero, y los cambios se propagan entre los repositorios locales. Los cambios se importan como ramas adicionales y pueden ser fusionados en la misma manera que se hace con la rama local.

-Los almacenes de información pueden publicarse por HTTP, FTP, rsync o mediante un protocolo nativo, ya sea a través de una conexión TCP/IP simple o a través de cifrado SSH. Git también puede emular servidores CVS, lo que habilita el uso de clientes CVS pre-existentes y módulos IDE para CVS pre-existentes en el acceso de repositorios Git.

-Los repositorios Subversion y svk se pueden usar directamente con git-svn.

-Gestión eficiente de proyectos grandes, dada la rapidez de gestión de diferencias entre archivos, entre otras mejoras de optimización de velocidad de ejecución.

-Todas las versiones previas a un cambio determinado, implican la notificación de un cambio posterior en cualquiera de ellas a ese cambio (denominado autenticación criptográfica de historial). Esto existía en Monotone.

-Resulta algo más caro trabajar con ficheros concretos frente a proyectos, eso diferencia el trabajo frente a CVS, que trabaja con base en cambios de fichero, pero mejora el trabajo con afectaciones de código que concurren en operaciones similares en varios archivos.

-Los renombrados se trabajan basándose en similitudes entre ficheros, aparte de nombres de ficheros, pero no se hacen marcas explícitas de cambios de nombre con base en supuestos nombres únicos de nodos de sistema de ficheros, lo que evita posibles y desastrosas coincidencias de ficheros diferentes en un único nombre.

-Realmacenamiento periódico en paquetes (ficheros). Esto es relativamente eficiente para escritura de cambios y relativamente ineficiente para lectura si el reempaquetado (con base en diferencias) no ocurre cada cierto tiempo.

**GitHub y GitLab. Definiciones y diferencias.**

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. Anteriormente era conocida como Logical Awesome LLC. El código de los proyectos alojados en GitHub se almacena típicamente de forma pública.

Gitlab es un servicio web de control de versiones y desarrollo de software colaborativo basado en Git. Además de gestor de repositorios, el servicio ofrece también alojamiento de wikis y un sistema de seguimiento de errores, todo ello publicado bajo una Licencia de código abierto.

GitLab es una suite completa que permite gestionar, administrar, crear y conectar los repositorios con diferentes aplicaciones y hacer todo tipo de integraciones con ellas, ofreciendo un ambiente y una plataforma en cual se puede realizar las varias etapas de su SDLC/ADLC y DevOps.

Tanto GitHub como GitLab permiten mantener repositorios de código, tanto públicos como privados. La diferencia estriba en que, mientras que GitLab es siempre gratuito, GitHub solo te permite repositorios privados bajo suscripción. Pero no solo esto, también presentan otras diferencias:

-Niveles de autentificación. Mientras que GitHub solo te da la opción de lectura o escritura en un repositorio, con GitLab puedes proporcionar acceso a determinadas partes del proyecto.

-GitLab permite incluir adjuntos en un issue o problema.

-GitLab permite marcar un proyecto como que está en desarrollo, lo que avisará a otros usuarios de la situación del mismo. Esto permite evitar que otros usuarios integren tu proyecto antes de tiempo…

-GitHub tiene una comunidad de usuarios y una cantidad de proyectos muy superior a GitLab. Mientras que en GitLab hay poco mas de 100.000 proyectos, en GitHub superan los 35.000.000.

**Comandos existentes y para qué sirven.**

-Configurar Nombre que salen en los commits:

git config --global user.name "omar"

-Configurar Email:

git config --global user.email [omar69@gmail.com](mailto:omar69@gmail.com)

-Marco de colores para los comando:

git config --global color.ui true

-Iniciamos GIT en la carpeta donde está el proyecto:

git init

-Clonamos el repositorio de github o bitbucket:

git clone <url>

-Añadimos todos los archivos para el commit:

git add .

-Hacemos el primer commit:

git commit -m "Texto que identifique por que se hizo el commit"

-Subimos al repositorio:

git push origin master

-Clonamos el repositorio de github o bitbucket:

git clone <url>

-Añadimos todos los archivos para el commit:

git add .

-Añadimos el archivo para el commit:

git add <archivo>

-Añadimos todos los archivos para el commit omitiendo los nuevos:

git add --all

-Añadimos todos los archivos con la extensión especificada:

git add \*.txt

-Añadimos todos los archivos dentro de un directorio y de una extensión específica:

git add docs/\*.txt

-Añadimos todos los archivos dentro de un directorios

git add docs/

-Cargar en el HEAD los cambios realizados

git commit -m "Texto que identifique por que se hizo el commit"

-Agregar y Cargar en el HEAD los cambios realizados

git commit -a -m "Texto que identifique por que se hizo el commit"

-De haber conflictos los muestra

git commit -a

-Agregar al último commit, este no se muestra como un nuevo commit en los logs. Se puede especificar un nuevo mensaje

git commit --amend -m "Texto que identifique por que se hizo el commit"

-Subimos al repositorio

git push <origien> <branch>

-Subimos un tag

git push --tags

-Muestra los logs de los commits

git log

-Muestras los cambios en los commits

git log --oneline --stat

-Muestra los cambios realizados a un archivo

git diff

-Agregar repositorio remoto

git remote add origin <url>

-Cambiar de remote

git remote set-url origin <url>

-Remover repositorio

git remote rm <name/origin>

-Muestra lista repositorios

git remote -v

-Muestra los branches remotos

git remote show origin

-Crea un branch

git branch <nameBranch>

-Lista los branches

git branch

-Comando -d elimina el branch y lo une al master

git branch -d <nameBranch>

-Elimina sin preguntar

git branch -D <nameBranch>

-Muestra una lista de todos los tags

git tag

-Crea un nuevo tags

git tag -a <verison> - m "esta es la versión x"

-Lista un estado actual del repositorio con lista de archivos modificados o agregados

git status

-Quita del HEAD un archivo y le pone el estado de no trabajado

git checkout -- <file>

-Crea un branch en base a uno online

git checkout -b newlocalbranchname origin/branch-name

-Busca los cambios nuevos y actualiza el repositorio

git pull origin <nameBranch>

-

Cambiar de branch

git checkout <nameBranch/tagname>

-

Une el branch actual con el especificado

git merge <nameBranch>

-Verifica cambios en el repositorio online con el local

git fetch

-Borrar un archivo del repositorio

git rm <archivo>

-Descargar remote de un fork

git remote add upstream <url>

-Merge con master de un fork

git fetch upstream

git merge upstream/master

-Saca un archivo del commit

git reset HEAD <archivo>

-Devuelve el último commit que se hizo y pone los cambios en staging

git reset --soft HEAD^

-Devuelve el último commit y todos los cambios

git reset --hard HEAD^

-Devuelve los 2 último commit y todos los cambios

git reset --hard HEAD^^

-Rollback merge/commit

git log

git reset --hard <commit\_sha>

**Ramas, tipos y usos.**

Las ramas son una de las principales utilidades que disponemos en Git para llevar un mejor control del código. Se trata de una bifurcación del estado del código que crea un nuevo camino de cara a la evolución del código, en paralelo a otras ramas que se puedan generar.

Las ramas nos pueden servir para muchos casos de uso. Por ejemplo, para la creación de una funcionalidad que queramos integrar en un programa y para la cual no queremos que la rama principal se vea afectada. Esta función experimental se puede realizar en una rama independiente, de modo que, aunque tardemos varios días o semanas en terminarla, no afecte a la producción del código que tenemos en la rama principal y que permanecerá estable.

El trabajo con ramas resulta muy cómodo en el desarrollo de proyectos, porque es posible que todas las ramas creadas evolucionen al mismo tiempo, pudiendo el desarrollador pasar de una rama a otra en cualquier momento según las necesidades del proyecto. Si en un momento dado el trabajo con una rama nos ha resultado interesante, útil y se encuentra estable, entonces podemos fusionar ramas para incorporar las modificaciones del proyecto en su rama principal.

La rama por defecto es master. También está la rama main y alguna que queramos crear.

**Resolución de conflictos.**

Algunas veces la unión de dos ramas no resulta tan bien, sino que ocurre un conflicto, esto cuando los commits de la rama a fusionar y la rama actual modifican la misma parte en un archivo en particular y git no puede decidir cuál versión elegir, y te avisa que tu debes resolverlo.

Para resolverlo hay que hacer un cambio de rama y commit, y posteriormente un merge.