**Crear un repositorio Git:**

Dos formas:

1. Dentro de un directorio existente:

git init

1. Clonar un repositorio existente:

git clone [url]. Crea el directorio donde instala el proyecto.

$ git clone git://github.com/schacon/grit.git

Podemos cambiar el nombre del proyecto:

git clone git://github.com/schacon/grid.git ProyectoGrid

**Configuración Git:**

El fichero de configuración global de Git es c:\Usuarios\usuario\.gitconfig

$ git config --global user.name "John Doe"

$ git config --global user.email [johndoe@example.com](mailto:johndoe@example.com)

$ git config --global core.editor emacs

$ git config --global merge.tool vimdiff

Para mostrar la configuracion:

git config --list

**Estados de los archivos gestionados con Git:**

- Confirmado (commited):

Los datos están almacenados de manera segura en la DB local.

- Modificado (modified):

El fichero se ha modificado pero no se ha confirmado (commit) a la DB local.

- Preparado (staged):

El fichero modificado ha sido marcado para el siguiente commit.

A un nivel superior existe otra división de los ficheros gestionados por git, tracked o untracked, que quiere decir que son seguidos o no por git.



El directorio de Git es donde Git almacena los metadatos y la base de datos de objetos para tu proyecto. Es la parte más importante de Git, y es lo que se copia cuando clonas un repositorio desde otro ordenador.

El directorio de trabajo es una copia de una versión del proyecto. Estos archivos se sacan de la base de datos comprimida en el directorio de Git, y se colocan en disco para que los puedas usar o modificar.

El área de preparación es un sencillo archivo, generalmente contenido en tu directorio de Git, que almacena información acerca de lo que va a ir en tu próxima confirmación. A veces se le denomina índice, pero se está convirtiendo en estándar el referirse a ella como el área de preparación.

**Comandos git:**

git add <fichero>, dos funciones:

* Incluye el fichero para el seguimiento de git
* Incluye el fichero para el siguiente commit

git branch , lista de branchs disponible en el repositorio local.

git brach –a, lista branchs incluso los ocultos.

git branch –d <local-branch>, elimina la rama local <local-branch>, con –D para forzar el borrado sin chequear el estado del merge.

Para eliminar una rama remota:

git push origin –-delete <remote-branch>,

git checkout branch1, para cambiar a la rama branch1

git checkout –b branch1, crea la rama branch1, copia todos los ficheros de la rama desde donde se lanza este comando y se coloca en ella (en branch1). Equivale a:

git branch branch1

git checkout branch1

CORREGIR, DESHACER

git commit –-amend, sustituye o corrige el último commit. Por ejemplo, si olvidamos incluir en el commit los cambios sobre un fichero, podemos:

git commit –m “commit incompleto”

git add ficheroquefaltaba y que no estaba en estaged (add) cuando se hizo el commit.

git commit --amend

git commit -m , prepara los ficheros para el push (repositorio remoto y base).

CORREGIR, DESHACER

git checkout –- <fichero>, para recuperar un fichero modificado después del commit, es decir, obtendremos la versión del HEAD, es decir, de lo último commiteado.

git commit –a -m , prepara los ficheros para el push aunque no se haya hecho “add“ sobre él.

git config --global –-list, configuración global establecida

git config --global --add http.proxy <https://aflucena:PASSWORD@proxy.indra.es:8080>, para definir el proxy en modo global.

git config --global -–unset http.proxy, elemina el poxy global

git config --unset http.proxy, elimina el proxy del repositorio concreto

git config --system --unset credential.helper, para borrar las credenciales (Usuario, password) que almacena window para github.com

git config --global core.editor "'C:/Program Files/Notepad++/notepad++.exe' -multiInst -notabbar -nosession -noPlugin"

git config –-global push.default, mostrar push.default

git config –-global push.default <tipo para push>, establece el modo en el que se va a elegir la rama para la subida: nothing, current, upstream, matching, simple.

git diff , muestra los cambios de los ficheros no preparados

git diff -- cached , muestra los cambios de los ficheros que están en estado preparado.

git diff [–-name-status] rama1 rama2, muestra diferencias entre dos ramas

**Para ver las diferencias con respecto al repositorio base, tenemos que hacer un git fetch origin**, que actualizará la rama oculta local con nombre remotes/origin/master.

Ahora sí podremos ver los cambios con git diff master origin/master

git diff rama1:fichero rama2.fichero, muestra las diferencias de un fichero entre dos ramas.

git diff-tree --no-commit-id --name-only -r <idcommit>, muestra los ficheros confirmados en un commit identificado por <idcommit>.

Nota: para ver los commits realizados: git log

**git fetch branch, crea y carga en rama oculta el branch indicado (normalmente branch = origin/master)**

**git help,**  git help <comando>

git log, Muestra las confirmaciones (commits) realizadas, con la opcion –p muestra el detalle.

git log –-branches –-not –-remotes, muestra los commits realizados pendientes de push. A continuación podemos hacer un git show <id-commit> para ver los ficheros implicados en un commit.

git log [–-follow] <fichero>/<path>, muestra los commits realizados sobre ese fichero, <fichero> puede ser un nombre de fichero o un template que identifique a un único fichero. Con –-follow sirve para los renames.

git merge branch1, hace merge del branch1 sobre la rama activa

git mv <fichero-from> <fichero-to>

rename f1 f2 (Shell dos)

git mv f1 f2, es equivalente a :

git rm f1

git add f2

git pull origin master, importar o incorporar los cambios en repositorio remoto. git pull es equivalente a git pull origin master.

git push origin master, subir al repositorio

Importante: push sube ficheros implicados en una confirmación commit, de modo que un fichero que suba al repositorio, podría estar en ese momento en estado modificado, en el caso que se modifique después de un commit. Pero el fichero que se subirá será como estaba en el momento que se commiteo, es decir no se suben los cambios hechos después del commit.

git push origin –-delete <remote-branch>, para eliminar la rama <remote-branch> del repositorio origin

git push –-tag origin, sube también la etiqueta al repositorio remoto

git remote rm <repositorio>, para eliminar repositorio

git remote rename <repositorio-old-name> <repositorio-new-name>, para cambiar el nombre del repositorio

CORREGIR, DESHACER

git reset –-hard <hash del commit a recuperar>. Deja el directorio de trabajo local tal y como estaba con el commit indicado. Hay que tener en cuenta que elimina los ficheros creados posteriormente a ese commit indicado, por lo que puede estar bien hacer un push, antes del reset –-hard.

CORREGIR, DESHACER

git reset HEAD <file>, para pasar al estado solo modificado, es decir fuera de los preparados para commit (to unstage)

git rm <fichero>, para eliminar el fichero y sacarlo del seguimiento git

git rm –f <fichero>, para el eliminar el fichero y sacarlo del seguimiento git cuando tiene cambios.

git rm --cached <fichero>, para sacarlo del seguimiento sin eliminar, para esto mismo también se puede utilizar git reset HEAD <fichero>

Prueba hecha con rm –-cached <fichero>:

git rm –-cached sobre <fichero> + modificado <fichero> + git push origin master, lo que ha ocurrido es que se ha eliminado <fichero> en el repositorio.

git show <hash de un commit>:nombre/del/fichero > <fichero destino>, nos muestra el archive en ese momento

git show <tag>, información de situación en ese punto.

git show [--stat] --oneline HEAD^^^..HEAD, muestra los cambios en los cuatro últimos commit.

también:

git show [--stat] --oneline HEAD

git show [--stat] --oneline <id-commit>

git stash, guarda los cambios (fichero modificados y modificados en estado preparado) en una pila, de forma que puedan ser recuperados. La rama desde donde se hace el stash queda en el estado del último commit.

git stash list, puestra los stash / pilas stash generadas

git stash apply [stash@{n}] , aplica los cambios guardados, n=0 es última generada

git stash apply –-index, aplica cambios guardados manteniendo el estado de preparado para commit. Si no se indica, todos los ficheros pasan a estado modificado.

git stash branch <nombre-branch>, crea una pila con los cambios, los vuelca en la rama especificada y luego elimina la pila.

git status : muestra los ficheros preparados y no preparados que han cambiado

git tag –a V0.0.1 –m “message explicativo”, asigna etiqueta sobre el último commit.

Existe una aplicación de git para ver visualmente el histórico: gitK

**Carga inicial de un proyecto sobre repositorio de github con el branch master:**

1. Creamos el repositorio en github:

En el caso de estar el servidor github, podemos seguir las instrucciones y desde la web creamos el repositorio con nombre Proyecto1.

1. En la máquina del desarrollador y en la carpeta donde tenemos el proyecto hacemos :

git init

git add .

git commit -m “primer paso”

git remote add origin <https://github.com/antonio63j/Proyecto1>

antonio63j es el Usuario con la cuenta de github.

git push origin master

**Eliminar un repositorio en github.com:**

https://github.com/*YOUR-USERNAME*/*YOUR-REPOSITORY*/settings

**Colaborar en proyecto github:**

1. fork sobre el repositorio del proyecto donde vamos a colaborar, por ejemplo repositorio antonio63j/SpringRestOneToManyManyToMany. El fork hace ya una copia en nuestro espacio github. Con el usuario github antonio63jun, hacemos el fork.
2. Nos colocamos en nuestro repositorio local, en la carpeta donde queremos colgar el proyecto y clonamos:

git clone <https://github.com/antonio63jun/SpringRestOneToManuManyToMane.git>

1. Si no nos interesan el user y el email globales , los establecemos localmente:

git user antonio63jun

git email antonio63jun@gmail.com

1. Consultamos la configuración remota:

$ git remote –v

origin https://github.com/antonio63jun/SpringRestOneToManyManyToMany.git (fetch)

origin https://github.com/antonio63jun/SpringRestOneToManyManyToMany.git (push)

1. Establecemos la ubicación del repositorio original:

$ git remote add upstream https://github.com/antonio63j[/SpringRestOneToManuManyToMany.git](https://github.com/SpringRestOneToManuManyToMany.git)

1. Actualizar la rama master:

si se han hecho cambios nos pedirá hacer commit cuando se intenta el pull

opcion1:

git pull –r upstream master

opcion2:

git fetch upstream

git merge upstream/master

fetch deja los cambios en una rama oculta que modemos ver con “git branch –a”, en este caso sería en upstream/master

o sea que git pull = git fetch + git merge

1. Crear rama desde la rama master para realizar los cambios.

git checkout –b nueva-relacion-entidades

nota: En el repositorio base crear una issue donde se expliquen los cambios que se van a hacer en esta rama.

1. Hacemos commit de los cambios sobre el nuevo branch. Indicamos el repo y el branch con los cambios:

git push origin nueva-relacion-entidades

Se crea el branch nueva-relacion-entidades en github

1. Implementados los cambios, generamos el “ pull request” para que el administrador del repo base lo revise y realice el merge. Desde github, tendremos que pinchar en pull request desde el branch “nueva-relacion-entidades”.

**Pruebas realizadas:**

Prueba 1:

Tenemos disponible el repositorio de antonio63j con nombre SpringRestOneToManyManyToMany en el servidor github. Los usuarios de github Antonio63jun y antonio63junio, van a solaparse en las subidas sobre el repositorio master de SpringRestOneToManyManyToMany perteneciente a antonio63j.

En T100, antonio63jun hace fork (click fork en github.com), crea la Issue3, clona el proyecto y:

Modifica README y Phone.java. También añade el fichero CambiosIssue3 (sin git add)

$ git config -l

*core.symlinks=false*

*core.autocrlf=true*

*core.fscache=true*

*color.diff=auto*

*color.status=auto*

*color.branch=auto*

*color.interactive=true*

*pack.packsizelimit=2g*

*help.format=html*

*http.sslcainfo=C:/Program Files/Git/mingw32/ssl/certs/ca-bundle.crt*

*diff.astextplain.textconv=astextplain*

*rebase.autosquash=true*

*credential.helper=manager*

*user.name=antonio63j*

*user.email=antonio63j@gmail.com*

*filter.lfs.clean=git-lfs clean -- %f*

*filter.lfs.smudge=git-lfs smudge -- %f*

*filter.lfs.required=true*

*http.proxy=http://aflucena:XXX@proxy.indra.es:8080*

*https.proxy=http://aflucena:XXX@proxy.indra.es:8080*

*core.repositoryformatversion=0*

*core.filemode=false*

*core.bare=false*

*core.logallrefupdates=true*

*core.symlinks=false*

*core.ignorecase=true*

*remote.origin.url=https://github.com/antonio63jun/SpringRestOneToManyManyToMany.git*

*remote.origin.fetch=+refs/heads/\*:refs/remotes/origin/\**

*branch.master.remote=origin*

*branch.master.merge=refs/heads/master*

*user.name=antonio63jun*

*user.email=antonio63jun@gmail.com*

*remote.upstream.url=https://github.com/antonio63j/SpringRestOneToManyManyToMany.git*

*remote.upstream.fetch=+refs/heads/\*:refs/remotes/upstream/\**

En T200, antonio63jun, hace un fetch y un merge :

git fetch upstream

git merge upstream/master

Los resultados son:

No se observan problemas, los cambios en README y Phone.java permanecen.

En T250, hace un pull:

*git pull –r upstream master*

Devuelve este error:

*$ git pull -r upstream master*

*error: cannot pull with rebase: You have unstaged changes.*

*error: please commit or stash them.*

En T251, hacemos un

*git add .*

*git commit -m "Primer commit con cambios en README y Phone.java, ademas se añade CambiosIssue3"*

En T260, hacemos un

git fetch upstream

git merge upstream/master, a este último responde con : Already up-to-date.

A continuación:

*$ git pull -r upstream master*

*From https://github.com/antonio63j/SpringRestOneToManyManyToMany*

*\* branch master -> FETCH\_HEAD*

*Current branch master is up to date.*

En T270 decidimos crear una nueva rama, tal vez hubiese sido lo apropiado justo al principio de clonar el repo.

git checkout –b Issue3

modificamos fichero application-properties

git commit -a –m “para issue3”

En T309, antonio63junio hace fork, crea la issue 4, clona el proyecto y:

334 git add .

335 git config user.name "antonio63junio"

336 git config user.email "antonio63junio@gmail.com"

339 git remote add upstream https://github.com/antonio63j/SpringRestOneToManyManyToMany.git

342 git pull -r upstream master

345 git checkout -b rama-para-issue4

se modifica README

git commit -a -m "modifica AREADME cambiando la fecha de comienzo"

En T400, antonio63jun actualiza el proyecto base:

git push origin rama-issue3

y desde github.com hacemos un pull resquest para que antonio63j, lo incluya en proyecto base.

En T500, antonio63j hace el merge sobre sobre la rama base.

En T00, antonio63junio hace:

git fetch upstream

git merge upstream/master

git config --system --unset credential.helper

git push origin rama-master-issue4

En bithub.com, realizamos el pull request

En definitiva, seguimos los mismos pasos descritos anteriormente (antonio63jun) para actualizar el repositorio base (el de antonio63j en github).

Para actualizar el repositorio base en github.com de antonio63jun, hemos seguidos los pasos descritos en

[http://stackoverflow.com/questions/20984802/how-can-i-keep-my-fork-in-sync-without-adding-a-separate-remote/21131381#21131381](http://stackoverflow.com/questions/20984802/how-can-i-keep-my-fork-in-sync-without-adding-a-separate-remote/21131381%2321131381)., que parece equivalente a eliminar el repositorio y volver a hacer un fork.

Reajuste de los ficheros controlados por git:

1. ajustar fichero .gitignore
2. git rm –-cache –r .

1. git add .
2. ver modificaciones con git status
3. git –m commit

Prueba 2:

Tenemos el proyecto P en github y hacemos cambios desde dos plataformas de desarrollo pl1 y pl2.

Desde pl2 hacemos cambios en un fichero README, pero no hacemos add ., ni commit

Desde pl1 hacemos cambios en un fichero f1, add ., commint y push al repositorio origin/master de githup.

prueba 2.1:

Si desde pl2 hacemos git diff master origin/master no aparecen diferencias. **Para ver las diferencias tenemos que hacer en pl2 git fetch origin**, que actualizará la rama oculta local con nombre remote/origin/master.

Si ahora hacemos (en pl2) git diff master origin/master, sí aparecen los cambios que realizó pl1 (f1), pero no aparece como diferencia los cambios realizados por pl2 (fichero README), pues no se ha hecho commit.

A continuación se hace merge (en pl2), git merge origin/master, y actualiza f1 en pl2. También vemos que mantiene los cambios realizados en pl2 sobre el fichero README. En este punto git diff origin/master master no presenta diferencias y cuando se hacer el commit en pl2, es cuando si se muestran las diferencias entre las dos ramas, en este caso vemos los cambios del fichero README.

En este punto hacemos git push origin master.

prueba 2.2:

Con el escenario de la prueba2, hacemos cambios en f1 desde pl1, después hacemos git add .; y commit pero al intentar git push origin master, nos da error, advirtiéndonos que debemos integrar los cambios, git nos propone utilizar pull.

A continuación hacemos un git pull origin master, en este proceso se solicita etiquetar un commit para el merge. Finalizado el pull, podemos hacer el git push origin master sin problemas.

prueba 2.3:

Una vez realizados los pasos de la prueba 2, desde pl2 se modifica f1, hacemos git diff master origin/master y no muestra cambios. Hacemos git fetch origin y a continuación vemos que ya aparecen los cambios efectuados en pl1 sobre el fichero f1, pero no el que desde pl2 se ha realizado en f1. Luego hacemos un git push origin y nos devuelve error indicando que debemos integrar los cambios en la plataforma pl2. Hacemos por tanto un git merge origin/master a lo que git responde con una advertencia de que podrían sobre-escribirse los cambios realizados en f1, por lo que ejecutamos git add., git commit –m. Después de esto ya git nos muestra las modificacionesde pl1 sobre f1 y las propias (pl2 sobre f1), también ejecutamos git merge origin/master sin error.

Luego:

git tag -a V0.0.1 -m "primera versión del proyecto"

push --tag origin master