**Chuleta de comandos git**

* **git help <command>**
* **git clone <uri> namedir** # clona usando como nombre de directorio namedir.
* **git add <dir>** # añade recursivamente todos los archivos del dir.
* **git diff --staged** # compares staged changes with last commit
* **git commit -v** # muestra el diff en el editor
* **git commit -a -m ”** # automatically stage tracked files. No hace falta git add
* **git rm --cached <file or regexp>** # Git no realiza un seguimiento del archivo, pero los deja en el directorio de trabajo. Útil cuando se olvida añadir archivos al .gitignore y ya hemos agregado dichos archivos al repositorio.
* **git rm <file>** # borrarlos con git siempre.
* **git rm -f <file>** # si ya está modificado y en el index.
* git mv <file> <renamed\_file>
* **gitk** # tcl/tk. Herramienta gráfica para git
* **git commit --amend** # Modificar el mensaje del último commit
* **git reset HEAD <file>** # to unstage
* **git checkout -- <file>** # Descartar cambios en el directorio de trabajo.

**AÑADIR ARCHIVOS**

* **git add -i** # interactive staggin
* **git add -p** # crea patch

**STASH**

* **git stash** # guarda el estado en una pila y limpia el directorio para poder cambiar de rama
* **git stash list** # muestra la pila
* **git stash apply** # vuelve al estado original del dir. Stash{n} especifica uno concreto Y --index reaplica los cambios stagged
* **git stash pop** # elimina el primero en la pila. O drop

**LOGS**

* **git log -p -2** # Muestra 2 últimos commits con diff
* **git log --stat**
* **git log --pretty <short|full|fuller>**
* **git log --pretty=format:”%h - %an, %ar : %s”**
* **git log --pretty=format;”%h %s” --graph**
* **git log --since=2.weeks**
* **git log <branch> --not master** # Muestra commit de <branch> sin incluir los de master
* **git log --abbrev-commit --pretty=oneline**
* **git diff master…contrib** # Muestra solo el trabajo que la rama contrib actual ha introducido desde su antecesor común con master
* **git log <branch1>..<branch2>** # Commits de branch2 que no están en branch1
* **git log origin/master..master** # Muestra qué commits se van a enviar al servidor
* **git log origin/master..** # Igual que el anterior. Se asume master o HEAD
* **git log refA refB --not refC** # commits en refA y refB que no están en refC
* **git log master…experiment** # commits de master o experiment, pero sin ser comunes. Con --left-right indica a qué rama pertenece cada uno

**REMOTES # repos en internet**

* **git remote -v** # lista los repos remotos
* **git remote add [shortname] [url]** # crea nuevo remote, es posible descargar el contenido de ese repo con git fetch [shortname]. Master branch en [shortcode]/master
* **git fetch <remote>** # descarga trabajo nuevo a máquina local, no sobreescribe nada tuyo. ( git pull sí hace merge automaticamente si se esta realizando un seguimiento de esa branch)
* **git push [remote-name] [branch-name]** # sii nadie ha hecho push antes
* **git remote show [remote-name]** # inspecciona remote.
* **git remote rename <old-name> <new-name>** # también renombra branches: quedaría <new-name>/master
* **git remote rm <remote-name>** # p.e si el contribuidor ya no contribuye más

**Añadir varios repositorios remotos**

* **git remote add bitbucket git@bitbucket.org:algui91/grado\_informatica\_tsi\_practicas.git** # Añadir un nuevo repositorio remoto con el nombre deseado. Por ejemplo si ya tenemos uno en github y queremos añadir otro para bitbucket
* **git push -u bitbucket –all** # Subir el proyecto a bitbucket. A partir de ahora se puede seleccionar a qué repo publicar con**\*git push nombre\_repo\_remoto**

**TAGGING # marcan puntos importantes en la histtoria del repo ( releases )**

* **git tag** # muestra las etiquetas actuales
* **git tag -l ‘v1.4.2.\*’** # acepta regex
* Dos tipos de tag:
* **Lightweight** : puntero a commit ( branch que no cambia )
* **Annotated** : se almacenan como objetos en la db, con checksum, nombre del creador, email, fecha, mensaje, posibilidad de firmarla con [GPG](https://elbauldelprogramador.com/como-cifrar-correos-con-gpg-con-mailvelope/). ( recomendada )
* **git tag -a <tagname> -m ‘mensaje’** # annotated tag
* **git show <tag-name>** # muestra información asociada.
* **git tag -s <tag-name> -m ‘message’** # la firma con gpg
* **git tag <tag-name>** # lightweight tag
* **git tag -v <tag-name>** # verifica tags firmadas
* **git tag -a <tag-name> [commit-chksum]** # crea tag para commit con dicho chksum
* Por defecto no se transfieren los tags, para subirlos al servidor:
* **git push origin [tag-name]** # una sola
* **git push origin --tags** # Enviar todas
* Para usar GPG y firmar tags, hay que subir la clave pública al repositorio:
* **gpg --list-keys** # Coges la id pública
* **gpg -a --export <id> | git hash-object -w --stdin** # Copia el SHA-1 devuelto
* **git tag -a maintainer-gpg-pub <SHA-1>**
* **git push --tags** # Comparte la clave con todos los usuarios
* **git show maintainer-gpg-pub | gpg --import** # Cada usuario importa la clave así
* **git show <tag>** # Devuelve más información sobre la etiqueta
* **git tag -d nombre\_tag** # eliminar la etiqueta
* **git push origin :refs/tags/nombre\_tag** # Eliminar la etiqueta del repositorio remoto.

**BRANCH**

Las ramas simplememte son punteros a distintos snapshots

* **git branch <nombre-rama>** # crea rama. Puntero al commit actual
* **git checkout <nombre-rama>** # cambiar a la rama especificada.
* **git checkout -b <nombre-rama>** # crea y cambia de rama
* **git merge <rama>** # Mezcla la rama actual con <rama>
* **git branch -d <rama>** # elimina la rama
* **git push origin --delete <branchName>** # Elimina una rama del servidor
* **git mergetool** # Herramienta gráfica para resolver conflictos
* **git branch** # lista ramas
* **git branch -v** # lista ramas mostrando último commit
* **git branch --merged** # lista ramas que han sido mezcladas con la actual. Si no tienen un \*, pueden borrarse, ya que significa que se han incorporado los cambios en la rama actual.
* **git branch --no-merged** # lista ramas que no han sido incorporadas a la actual.

**REMOTE BRANCHES**

* **git fetch origin** # Descarga el contenido del servidor
* **git push <remote> <branch>** # Las ramas no se suben por defecto, has de subirlas explícitamente
* **git push <remote> <branch>:<nuevoNombre>** # Igual que la de arriba, pero en el servidor se llama a la rama con nuevoNombre en lugar de branch
* **Cuando se hace un git fetch que trae consigo nuevas ramas remotas, no se disponen de ellas localmente, solo se dispone de un puntero a la rama remota que no es editable. Para poder trabajar sobre esa rama, es necesario crearla Por ejemplo:**
* **git fetch origin** # Tras ejecutarlo, notamos que se ha creado una rama nueva (rama\_nueva)
* **git checkout -b rama\_nueva origin/rama\_nueva** # Crea una rama local a partir de la remota
* **git merge origin/nueva\_rama** # Equivalente a la de arriba, pero sin establecer el tracking a la rama
* **git push [remotename] :[branch]** # elimina una rama remota
* \*\*git push [remotename] [localbranch]:[remotebranch] \*\* # La rama en el servidor tiene distinto nombre a la local

**TRACKING BRANCHES**

* **git checkout --track origin/rama** # Equivalente a -b rama\_nueva origin/rama\_nueva
* **git chekout -b <nuevo\_nombre> origin/<rama>** # Establece un nombre distinto para la rama local

**REBASE**

**Rebase y merge se diferencian en que merge mezcla dos puntos finales de dos snapshots y rebase aplica cada uno de los cambios a la rama en la que se hace el rebase. No lo uses en repos publicos con mas colaboradores, porque todos los demas tendrán que hacer re-merges**

* **git checkout <una rama>**
* **git rebase master** # aplica todos los cambios de <una rama> a master
* **git merge master** # hay que hacer un merge de tipo fast forward
* **Tenemos 3 ramas, master, client y server, en server y client tenemos varios commit y queremos mezclar client en master pero dejar server intacta:**
* **git rebase --onto master server client** # adivina los patches del antecesor común de las ramas server y client y aplica los cambios a master.
* **git checkout master**\*
* **git merge client** # fast-forward. Client y master en el mismo snapshot
* **Si se quiere aplicar también los cambios de server, basta con:**
* **git rebase master server**\*
* **git checkout master**\*
* **git merge server**\*
* **git rebase [basebranch] [topicbranch]** # sintaxis de rebase
* **git rebase -i** # Rebase interactivo

**SERVIDOR**

* **git instawew** # Muestra una interfaz web con los commits

**GENERAR UN NÚMERO DE COMPILACIÓN (BUILD NUMBER)**

* **git describe master** # Solo funciona para tags creadas con -s ó -a

**PREPARAR UNA RELEASE**

* **git archive master -- prefix=”project/’ | gzip > `git describe master`.tar.gz**
* git archive master -- prefix=”project/’ --format=zip | `git describe master`.zip
* test/ export-ignore # Al crear el tarball no incluye el directorio test/

**GENERAR UN CHANGELOG**

* **git shortlog --no-merges master --not <tag>** # Recopila todos los commits desde <tag> y los agrupa por autor

**RECOMENDACIONES**

* Siempre hay que hacer pull antes de push en caso de que alguien haya subido cambios al servidor. Ejemplo:
* User1 clona el repo y hace cambios, realiza un commit
* User2 clona el repo, hace cambios, hace commit y sube los cambios con push
* User1 intenta hacer push, pero será rechazado con: <u>! [rejected] master -> master (non-fast forward)</u>. No puede subir los cambios hasta que no mezcle el trabajo que ha subido User2. Así que debe hacer lo siguiente:
* git fetch origin
* git merge origin/master
* git push origin master
* Mientras User1 hacía estas operaciones, User2 ha creado una rama <u>issue54</u> y realizado 3 commits, sin haber descargado los cambios de User1. Para sincronizar el trabajo, User2 debe hacer:
* **git fetch origin**
* **git log --no-merges origin/master ^issue54** # Observa qué cambios ha hecho User1
* **git checkout master**
* **git merge issue54 && git merge origin/master**\*
* **git push origin master**\*
* **git diff --check** # Antes de hacer commit, ejecutar esto para ver si hemos añadido demasiados espacios que puedan causar problemas a los demás.
* Commits pequeños que se centren en resolver un problema, no commits con grandes cambios.
* **git add --patch** # En caso de hacer varios cambios en el mismo archivo
* El mensaje del commit debe tener la estructura siguiente: Una linea de no más de 50 caracteres, seguida de otra línea en blanco seguida de una descripción completa del commit.

**PASOS A SEGUIR PARA CONTRIBUIR A PROYECYOS AJENOS, MEDIANTE FORK**

* git clone <url>
* git checkout -b featureA
* git commit
* git remote add myFork <url>
* git push myFork featureA
* git request-pull origin/master myFork # enviar la salida por mail al propietario del proyecto, o hacer click en pull request.
* Buena practica tener siempre una rama master que apunte a origin/master, para estar siempre actualizado con los ultimos cambios en el proyecto original.
* **Separar cada trabajo realizado en topic branch, que trackeen a origin/master**
* git checkout -b featureB origin/master
* (Hacer cambios)
* git commit
* git push myFork featureB
* (Contactar con el propietario del proyecto)
* git fetch origin
* **Otro ejemplo, el propietario del proyecto quiere aceptar un pull tuyo, pero quiere que hagas algunos cambios, aprovechas la oportunidad y mueves tu trabajo para basarlo en el contenido actual de la rama origin/master, aplastas los cambios en**featureB**, resuelves conflictos, y haces push:**
* git checkout -b featureBv2 origin/master
* git merge --no-commit --squash featureB
* (cambiar la implementacion)
* git commit
* git push myFork featureBv2
* **--squash coge todo el trabajo de la rama mezclada y la aplasta en un no-merge commit encima de la rama en la que estas. --no-commit no registra el commit automaticamente. Así puedes realizar todos los cambios necesarios y luego hacer el commit**

**REFLOG**

En segundo plano, git crea un log de a donde han estado referenciando HEAD y el resto de ramas en los últimos meses.

* **git reflog**\*
* \*\*git show HEAD@{n} \*\* # Muestra información sobre el reflog número n
* **git log -g master** # Muestra el log formateado como la salida de reflog
* \*\*git show master@{yesterday} \*\* # Muestra los commits de ayer.

**UTILIDADES**

* **git show <short-SHA-1>** # Es posible ver un commit pasando la versión abreviada del SHA-1
* **git rev-parse <branch>** # A qué SHA-1 apunta una rama
* \*\*git show HEAD^ \*\* # Muestra commit padre
* **git show HEAD^2** # Muestra segundo padre
* **git show HEAD~2** # El primer padre del primer padre
* **git filter-branch --tree-filter ‘rm -f <file>’ HEAD** # elimina el archivo de todos los commits

**DEPURACIÓN**

* File anotation
* **git blame -L 12,22 <archivo>** # muestra cuando y por quién se modificaron de la linea 12 a la 22
* **git blame -C -L 141,153 <file>** # cuando renombras un archivo o lo refactorizas en varios, muestra de donde vino originalmente.
* Búsqueda Binaria: Cuando hay un bug que no puedes localizar, usas bisect para dererminar en qué commit empezó a producirse el bug.
* **git bisect start**\*
* **git bisect bad** # marcas el commit actual como roto
* \*\*git bisect good [commit bueno] \*\* # último commit conocido que funcionaba
* Ahora irá preguntando hasta que encuentres el commit culpable. Si esta bien indicas git bisect good. De lo contrario git bisect bad. Al terminar hay que resetear.
* **git bisect reset**

**SUBMODULOS**

* **git submodule add <url>** # crea un directorio que contiene el comtenido de otro proyecto.
* Clonar un repo con submodulos
* git clone url
* git submodule init
* git submodule update

**CONFIGURATION**

* **git config --global <opcion> <valor>** # global para usuario, system todos y sin nada, especifico para el repo.
* \*\*git config {key} \*\* # muestra el valor de key
* **git config --global core.editor <editor>** # cambia el editor por defecto
* **git config --global commit.template $HOME/.gitmessage.txt** # plantilla para commits

|  |  |
| --- | --- |
| \*\*git config --global core.pager ‘more | less’\*\* # paginador por defecto, puedes usar cualquiera |

* **git config --global user.signingkey <gpg-key-id>** # clave gpg para firmar tags
* **git config --global core.excludesfile <file>** # como gitignore
* **git config --global help.autocorrect 1** # autocorrige cuando se escribe un comando incorrecto. Solo en git >= 1.6.1

|  |  |  |
| --- | --- | --- |
| **git config --global color.ui true** # colorea la salida de git. Valores: true | false | always |

* **git config --global core.autocrlf input** # para que usuarios linux no tengan problemas con los retornos de carro de windows
* **git config --global core.autocrlf true** # para usuarios de windows
* **git config --global core.whitespace trailing-space, space-before-tab, indent-with-non-tab, cr-at-eol** # respectivamente: busca espacios al final de línea, busca espacios al inicio de tabulación, busca líneas con 8 o más espacios en lugar de tabulaciones, acepta retornos de carro
* **git apply --whitespace=warn <patch>** # advierte de errores de espacios antes de aplicar el patch. Con --whitespace=fix intenta arreglarlos

**GIT ATTRIBUTES**

Archivo en .gitattributes en el directorio de trabajo o en .git/info/attributes para no committearlo

**Identificando archivos binarios**  
Muchos archivos son para uso local y no aportan información al repositorio. Para decirle a git qué archivos son binarios hacer añadir al archivo atributes:  
**<nombre archivo o regexp> -crlf -diff** # git no intentará corregir problemas de crlf ni mostrará los cambios con diff. En versiones >= 1.6 se pueden sustituir estos dos valores por la macro binary

**Diffing binary files**  
En ocasiones es útil mostrar diffs de archivos binarios, como una archivo de word:  
**\*.doc diff=word**  
#tras esto hay que definir el filtro word para que git convierta archivos word a texto:  
**git config diff.word.textconv strings**

Es posible hacer lo mismo para imágenes jpeg, es necesario instalar**exiftool**para extraer los metadatos y luego hacer:  
**echo ‘\*.jpeg diff=exif’ » .gitattributes**  
**git config diff.exif.textconv exiftool**

**Procesar archivos antes de hacer commit y antes de hacer checkout**: Es posible crear tus propios filtros para hacer sustitución. Estos filtros se llaman**smudge**y**clean**. Los puedes configurar para distintos directorios y luego escribir un script que procesará cada archivo antes de que sea checkeado (smudge) y commiteado (clean). Para ello,escribe en el .gitattributes: (En caso que quieras procesar código C)

**\*.c filter=indent**Luego:

**git config --global filter.indent.clean indent**

**git config --global filter.indent.smudge cat**

Otro ejemplo interesante es la expansión de la palabra clave**$Date$**. Para ello hay que escribir un script en ruby que recibe un archivo, encuentra la fecha de su último commit e inserta dicha fecha en el archivo:

#! /usr/bin/env ruby

data = STDIN.read

last\_date = `git log &#45;&#45;pretty=format:"%ad" &#45;1`

puts data.gsub('$Date$', '$Date: ' + last\_date.to\_s + '$')

Puedes nombrar este script como**expand\_date**. Crea un filtro en git, llamado dater y dile que use el script anterior:  
**git config filter.dater.smudge expand\_date**  
**git config filter.dater.clean ‘perl -pe “s/\$Date[^\$]\*\$/\$Date\$/”‘**

Para usar el filtro, simplemente escribe la palabra clave en los archivos que desees:

echo '# $Date$' > date\_test.txt

echo 'date.txt filter=dater' » .gitattributes

git add date\_test.txt .gitattributes

git commit -m "Testing date expansion in Git"

rm date\_test.txt

git checkout date\_test.txt

cat date\_test.txt

$Date: Tue Apr 21 07:26:52 2009 -0700$

**GIT HOOKS**

Hay dos tipos, de lado cliente y servidor, se guardan en el directorio .git/hooks. Para activarlos basta con que sean ejecutables.

**CONCEPTOS**

Fast forward: cuando se hace un merge y el commit de la rama a mezclar esta justo un commit adelantado, simplemente se hace apuntar la rama en la que se iba a mezclar al commit del merge.

**GITIGNORE:**

**a comment - this is ignored**

***.a\*\* # no .a files*  
\*\*!lib.a** # but do track lib.a, even though you’re ignoring .a files above\*  
**/TODO** # only ignore the root TODO file, not subdir/TODO\*  
**build/** # ignore all files in the build/ directory\*  
**doc/\*.txt** # ignore doc/notes.txt, but not doc/server/arch.txt