LaTeX y Git aplicado a la investigación científica

Módulo 2 – Git Avanzado. Solución de problemas y aspectos avanzados

del 15 de marzo al 17 de mayo de 2022

AULA VIRTUAL

Manual: "*LaTeX y Git aplicado a la investigación científica*".

Curso Virtual

Módulo 2. *Git Avanzado. Solución de problemas y aspectos avanzados*

39 páginas.

Marzo 2022. Elaborado por Pablo Hinojosa.

Asociación Darwin Eventur

Git Avanzado

Flujos de desarrollo de software (y quizás de otras cosas)

Un sistema como git no es independiente de una organización del trabajo. Aunque a priori puedes trabajar como te dé la gana, el que te facilite el uso de ciertas herramientas hace que sea más fácil usar una serie de prácticas que son habituales en el desarrollo de software (y de otras cosas, como documentación o novelas) para hacer más productivos a los equipos de trabajo y poder predecir con más precisión el desarrollo de los mismos. Por eso, aunque se puede usar cualquier metodología de desarrollo de software con el mismo, git funciona mejor con metodologías ágiles que tienen ciclos más rápidos de producción y de despliegue de nuevas características o de arreglo de las mismas. Las metodologías ágiles son iterativas y en todas las iteraciones están presentes la mayoría de los actores del desarrollo: clientes, desarrolladores, arquitectos; incluso en algunas puede que esté la peña de marketing, a ver si se enteran de lo que está haciendo el resto de la empresa para venderlo (y no al revés, vender cosas que luego obligan al resto de la empresa a desarrollar).

El desarrollo se divide por tanto en estas fases:

1. Trabajo con el código. Modificar ficheros, añadir nuevos.
2. Prueba del código. La mayor parte de las metodologías de desarrollo hoy en día, o todas, incluyen una parte de prueba; en casi todos los casos esta prueba está automatizada e incluye test unitarios (que prueban características específicas), de integración y de cualquier otro tipo (calidad de código, existencia y calidad de la documentación).
3. Lanzamiento del producto. Cuando se han incorporado todas las características que se desean, se lanza el producto. El lanzamiento del producto, en el caso de web, incluye un *despliegue* (*deploy*) del mismo, y en el caso de tratarse de otro tipo de aplicación, de un *empaquetamiento*. Se suele hablar, en todo caso, de *despliegue* (aunque sea porque las aplicaciones web son más comunes hoy en día que las aplicaciones de escritorio).
4. Resolución de errores con el código en producción. Si surge algún error, se trata de resolver sobre la marcha (*hotfix*), por supuesto, incorporándolo al código que se va a usar para desarrollos posteriores.

En casi todas estas fases puede intervenir, y de hecho lo hace, un sistema de control de fuentes como git; en muchos casos no se trata de órdenes de git, sino de funciones a las que se puede acceder directamente desde sitios de gestión como GitHub.

Organización de un repositorio de git

No hay reglas universales para la organización de un repositorio, aunque sí reglas sobre cómo *no* debe hacerse: todo en un sólo directorio. El repositorio debe estar organizado de forma que cada persona sólo tenga que *ver* los ficheros con los que tenga que trabajar y no se *distraiga* con la modificación de ficheros con los cuales, en principio, no tiene nada que ver; también de forma que no se sienta tentado en modificar esos mismos ficheros. Vamos a exponer aquí algunas prácticas comunes, pero en cada caso el sentido común y la práctica habitual de la empresa deberá imponerse…

Qué poner en el directorio principal

Cuando se crea un repositorio en GitHub te anima a crear un README.md. Es importante que lo hagas, porque va a ser lo que se muestre cuando entres a la página principal del proyecto en GitHub y, además, porque te permite explicar, en pocas palabras, de qué va el proyecto, cómo instalarlo, qué prerrequisitos tiene, la licencia, y todo lo demás necesario para navegar por él.

Otros ficheros que suelen ir en el directorio principal:

- INSTALL por costumbre, suele contener las instrucciones para instalar. También por convención, hoy en día se suele escribir usando Markdown convirtiéndose, por tanto, en INSTALL.md.
- .gitignore posiblemente ya conocido, incluye los patrones y ficheros que no se deben considerar como parte del repositorio.
- LICENSE incluye la licencia. También se crea automáticamente en caso desde GitHub en caso de que se haya hecho así. No hay que olvidar que también hay que incluir una cabecera en cada fichero que índice a qué paquete pertenece y cuál es la licencia.
- TODO es una ventana abierta a la colaboración, así como una lista para recordarnos a nosotros mismos qué tareas tenemos por delante.
- Otros ficheros de configuración, como travis.yml, para el sistema de integración continua Travis, Makefile.PL o configure u otros ficheros necesarios para configurar la librería, y ficheros similares que haga falta ejecutar o ver al instalar la librería. Se aconseja siempre que tengan los nombres que suelan ser habituales en el lenguaje de programación, si no el usuario no sabrá cómo usarlos.

En general se debe tratar de evitar cargar demasiados ficheros, fuera de esos, en el directorio principal. Siempre que se pueda, se usará un subdirectorio.

Una estructura habitual con directorio de test

Las fuentes del proyecto deben ir en su propio directorio, que habitualmente se va a llamar src. Algunos lenguajes te van a pedir que tengan el nombre de la librería, en cuyo caso se usará el que más convenga. Si no se trata de una aplicación sino de una biblioteca, se usará liben vez de src, como en esta biblioteca llamada *NodEO* Los test unitarios irán aparte, en un directorio habitualmente llamado test. Finalmente, un directorio llamado examples o apps o scripts o bin o exe incluirá ejemplos de uso de la biblioteca o diferentes programas que puedan servir para entender mejor la aplicación o para ejecutarla directamente.

Estructura jerárquica con submódulos

Un repositorio git tiene una estructura plana, en el sentido que se trata de un solo bloque de ficheros que se trata como tal, a diferencia de otros sistemas de gestión de fuentes centralizados como CVS o Subversion en los que se podía tratar cada subdirectorio como si fuera un proyecto independiente. Pero en algunos casos hace falta trabajar con proyectos en los cuales haya un repositorio que integre el resultado del desarrollo independiente de otros, por ejemplo, una aplicación que se desarrolle conjuntamente

con una librería. En ese caso un repositorio git se puede dividir en submódulos, que son básicamente repositorios independientes pero que están incluidos en una misma estructura de directorios.

Por ejemplo, vamos a incluir el texto de este curso en el repositorio de ejemplo, para poder servirlo como una web también:

```
git submodule add git@github.com:oslugr/curso-git.git curso
```

```
Clonar en «curso»... remote: Reusing existing pack: 14, done. remote: Counting objects: 4, done. remote: Compressing objects: 100% (4/4), done. remote: Total 18 (delta 0), reused 0 (delta 0) Receiving objects: 100% (18/18), 17.26 KiB, done. Resolving deltas: 100% (4/4), done.
```

Los submódulos no se clonan directamente al clonar el repositorio. Hay que dar dos comandos: git submodule init y git submodule update dentro del directorio correspondiente; esta última orden servirá para actualizar submódulos también cada vez que haya un cambio en el repo del que dependan (y queramos actualizar nuestra copia); tras ellos habrá que decir git pull, como siempre, para traerse los ficheros físicamente.

De esta forma, el repositorio queda (parcialmente) con esta estructura de directorios:

```
├── curso
│
├── LICENSE
│
├── README.md
│
└── texto
        │
        ├── ganchos.md
        │
        ├── GitHub.md
        │
        └── mas-usos.md
```

con el subdirectorio curso siendo, en realidad, otro repositorio.

Por ejemplo, podíamos tener una estructura que incluyera subdirectorios para cliente (un submódulo) y servidor (otro submódulo). Con ambos se puede trabajar de forma independiente y, de hecho, *residen* en repositorios independientes, pero puede que, en caso de empaquetarlos o desplegarlos de alguna forma determinada (por ejemplo, a un PaaS), queramos hacerlo desde un solo repositorio, como en este caso.

Los submódulos pueden ser un ejemplo de flujos de trabajo: en este caso, hay un flujo desde "las fuentes del manantial" (que puede cambiar de forma independiente su proyecto) hasta "la desembocadura" (nuestro proyecto, que lo usa). Dividir un proyecto en módulos y dejar que personas independientes se encarguen de cada uno, integrándolo todo en un submódulo, por tanto, es una forma simple y sin demasiadas complicaciones de hacerlo.

Ramas

Las *ramas* son una característica de todos los sistemas de control de fuentes. A todos los efectos, una rama es un proyecto diferente que *surge* de un proyecto principal, aunque nada obliga a que contengan nada en común (por ejemplo, un repositorio puede incluir el código y las páginas web como una rama totalmente diferente). Sin embargo, las ramas, que más bien deberían llamarse *ramificaciones* o *caminos*, son *caminos divergentes* a partir de un tronco común que, eventualmente, pueden combinarse (aunque no es obligatorio) en uno sólo. En la práctica y como dicen aquí una rama es un nombre para un *commit* específico y todos los commits que son antecesores del mismo.

En git las ramas son también parte natural del desarrollo de un proyecto, dado que se trata de un sistema de control de fuentes distribuido. Cada usuario trabaja en su propia rama, que se *fusiona* con la rama principal en el repositorio compartido cuando se hace *push*. Por eso en alguna ocasión puede suceder que, cuando se hace *pull* o incluso *push*, si se encuentra que las ramas que hay en el repo con el que se fusiona y localmente contienen diferente número de *commits*, aparecerá un mensaje que te indicará que se está fusionando con la rama principal; todo esto, incluso aunque no se hayan creado ramas explícitamente. En realidad, *pull* es combinación de dos operaciones: fetch y merge, como ya se ha visto en el capítulo de uso básico. De hecho hay quien dice que no debe usarse nunca pull.

Por ejemplo, en caso de que se haya borrado un fichero (o, para el caso, hecho cualquier cambio) en un repositorio y se trate de hacer push desde el local, habrá un error de este estilo.

To git@github.com:oslugr/repo-ejemplo.git ! [rejected] master -> master (non-fast-forward) error: failed to push some refs to 'git@github.com:oslugr/repo-ejemplo.git' consejo: Updates were rejected because the tip of your current branch is behind consejo: its remote counterpart. Merge the remote changes (e.g. 'git pull') consejo: before pushing again. consejo: See the 'Note about fast-forwards' in 'git push --help' for details.

En este caso habrá dos ramas, en la *punta* de cada una de las cuales habrá un commit diferente. Se siguen instrucciones, es decir, git pull

jmerelo@penny:~/txt/docencia/repo-tutoriales/repo-ejemplo$ git pull

remote: Counting objects: 2, done.

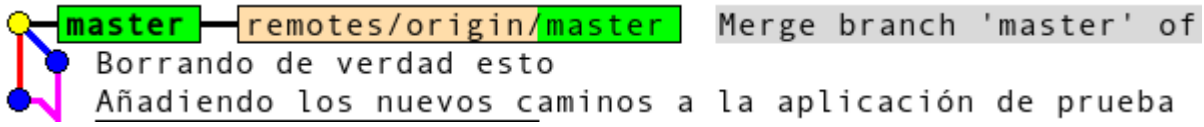
remote: Compressing objects: 100% (2/2), done.

remote: Total 2 (delta 0), reused 0 (delta 0) Unpacking objects: 100% (2/2), done.

De github.com:oslugr/repo-ejemplo 61253ec..2fd77db master -> origin/master

Eliminando Makefile Merge made by the 'recursive' strategy. Makefile | 3 --- 1 file changed, 3 deletions(-) delete mode 100644 Makefile

y aparece, efectivamente, el directorio borrado. Habrá que hacer el push de nuevo. Una vez hecho, el repositorio se ha estructurado como se muestra en la imagen:



Esta imagen, que se puede ver también en GitHub con fecha 2 de abril, y que está obtenida del programa cliente gitk, muestra cómo se ha producido la fusión. La que aparece más cerca de la fusión es la que se hizo inicialmente, borrando el fichero, y la más alejada, que aparece más a la izquierda, es la hecha a continuación. El último *commit* fusiona las dos ramas y crea una sola dentro de la rama principal.

Por eso hablamos de enramamiento (bueno, debería ser ramificación, pero esto suena mejor) *natural* en git, porque se produce simplemente porque haya dos *commits* divergentes que procedan de la misma rama. Sin embargo, se pueden usar ramificaciones adrede y es lo que veremos a continuación.

Ramas *ligeras*: etiquetas

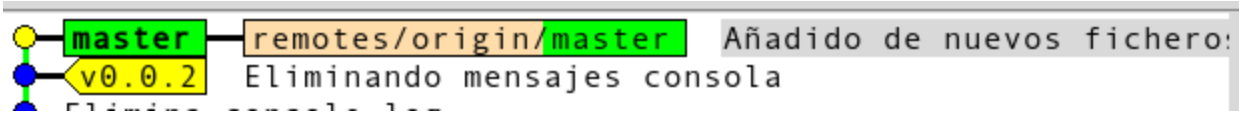
Una *etiqueta* permite *guardar* el estado del repositorio en un momento determinado, siendo como una especie de *foto* del estado el proyecto. Se suele asociar a hitos en la historia del mismo: entrada en producción, despliegue de los resultados, o versión mayor o menor.

Para etiquetar se usa la orden tag

git tag v0.0.2

tag etiqueta el último *commit*, es decir, asigna una etiqueta al estado en el que estaba el repositorio tras el último commit. La etiqueta aparecerá de forma inmediata (sin necesidad de hacer *push*, puesto que se añade al último commit y se puede listar con

git tag jmerelo@penny:~/txt/docencia/repo-tutoriales/repo-ejemplo$ git tag v0.0.1 v0.0.2



Como se ve, una etiqueta es, en realidad, un apodo para un commit determinado cuyo *hash* puede ser difícil de recordar. Pero por esa misma razón se pueden usar como salvaguarda del estado del repositorio en un momento determinado que, más adelante, se puede recuperar o fusionar con una rama.

Las ramas se pueden también anotar, lo que añade una explicación adicional a lo que ya esté almacenado en el commit correspondiente:

```
git tag -a v0.0.2.1
```

abrirá un editor (el que tengamos especificado por defecto) para añadir una anotación a esta etiqueta, lo que nos podemos ahorrar si usamos, por ejemplo:

```
git tag -a v0.0.2.1 -m "Estado estacionario del repositorio"
```

Esta información aparecerá añadida al commit correspondiente (el último que hayamos hecho) cuando hagamos, por ejemplo, git show v0.0.2.1:

```
tag v0.0.2.1 Tagger: JJ Merelo <jjmerelo@gmail.com> Date: Sun Apr 6 09:58:12 2014 +0200 Poco antes de pasar a producción el tema de tags en realidad, sólo es un ejemplo commit b958b16b8261fa3ca8159b3ae45e237ae1fa1dce Author: JJ Merelo <jjmerelo@gmail.com> Date: Sun Apr 6 09:45:38 2014 +0200 Añadido de nuevos ficheros al servidor Y edición del README para que sirva para algo
```

(Suprimidos espacios en blanco para que aparezca como un sólo mensaje). Que, como se ve, añade un pequeño mensaje (al principio) al propio del commit (a continuación).

Finalmente, git describe es una orden creada precisamente para trabajar con las etiquetas: te indica el camino que va desde la última etiqueta al commit actual o al que se le indique:

```
git describe v0.0.2.1-1-g6dd7a8c
```

que, de una forma un tanto críptica, indica que a partir de la etiqueta v0.0.2.1 hay un commit -1- y el nombre del último objeto, en este caso el único 6dd7a8c. Es otra forma de *etiquetar* un punto en la historia de una rama, o simplemente otra forma de llamar a un commit. Es más descriptivo que simplemente el hash de un commit en el sentido que te indica de qué etiqueta has partido y lo lejos que estás de ella.

Por eso precisamente conviene, como una buena práctica, etiquetar la rama principal con estas *ramas ligeras* cuando suceda un hito importante en el desarrollo. Y también conviene recordar que, dado que son anotaciones locales, hay que hacer explícitamente git push --tags para que se comuniquen al repositorio remoto.

Creando y fusionando ramas

Ya que hemos visto como se crean ramas de forma implícita y de forma *ligera* (con etiquetas), vamos a trabajar explícitamente con ramas. La forma más rápida de crear una rama es usar:

git checkout -b get-dir Switched to a new branch 'get-dir'

Esta orden hace dos cosas: crea la rama, copia todos los ficheros en la rama en la que estemos (que será la master si no hemos hecho nada) a la nueva rama y te cambia a la misma; a partir de ese momento estarás modificando ficheros en la nueva rama. Es decir, equivale a dos órdenes:

git branch get-dir git checkout get-dir

En esta rama se puede hacer lo que se desee: modificar ficheros, borrarlos, añadirlos o hacer algo totalmente diferente. En todo momento:

git status # En la rama get-dir

Nos dirá en qué rama estamos; los ficheros que físicamente encontraremos en el directorio de trabajo serán los correspondientes a esa rama. Conviene hacer siempre git status al principio de una sesión para saber dónde se encuentra uno para evitar cambios y sobre todo pulls sobre ramas no deseadas.

La rama que se ha creado sigue siendo rama local. Para crear esa rama en el repositorio remoto y a la vez sincronizar los dos repositorios haremos:

git push --set-upstream origin get-dir

donde get-dir es el nombre de la rama que hemos creado. Las ramas de trabajo se pueden listar con:

git branch * get-dir master

con un asterisco diciéndonos en qué rama concreta estamos; si queremos ver todas las que tenemos se usa:

git branch --all * get-dir master remotes/heroku/master remotes/origin/HEAD -> origin/master remotes/origin/get-dir remotes/origin/master

que, una vez más, nos muestra con un asterisco que estamos trabajando en la rama local get-dir; a la vez, nos muestra todas las ramas remotas que hay definidas y la relación que hay con las locales, pero más que nada por nombre. Si queremos ver la relación real entre ellas y los commits que hay en cada una:

jmerelo@penny:~/txt/docencia/repo-tutoriales/repo-ejemplo$ git branch -vv * get-dir 389b383 [origin/get-dir] Pasado a glob master 1a93e3d [origin/master] Añade palabros al diccionario

con -vv indicando doble verbosidad.

En este ejemplo se ha mostrado un patrón habitual de uso de las ramas: para probar nuevas características que no sabes si van a funcionar o no y que, cuando funcionen, se pasan a la rama principal. En este caso se trataba de trabajar con *todos* los ficheros del directorio en vez de los ficheros que le pasemos explícitamente.

Estas ramas se suelen denominar *ramas de características* o *feature branches* y forman parte de un flujo de trabajo habitual en git. Sobre un repositorio central, se crear una rama si quieres probar algo que no sabes si estará bien eventualmente o si realmente será útil.

De esta forma no se *estorba* a la rama principal, que puede estar desarrollando o arreglando errores por otro lado. En este flujo de trabajo, eventualmente se integra la rama desarrollada en la principal, para lo que se usa pull de nuevo.

El concepto de pull es usar primero fetch (descargarse los cambios al árbol) y posteriormente merge (incorporar los cambios del árbol al índice). En casos complicados esta división te da flexibilidad para escoger qué cambios quieres hacer, pero en un flujo de trabajo como este se puede usar simplemente. Supongamos, por ejemplo, que estamos en la rama get-dir y se han hecho cambios en la rama principal.

```
jmerelo@penny:~/txt/docencia/repo-tutoriales/repo-ejemplo$git pull origin master De github.com:oslugr/repo-ejemplo * branch master -> FETCH_HEAD Merge made by the 'recursive' strategy. .aspell.es.pws | 3 +++ README.md | 5 +++-- 2 files changed, 6 insertions(+), 2 deletions(-)
```

Este mensaje te muestra que se ha fusionado usando una estrategia determinada. git examina los commits que diferencian una rama de la otra y te los aplica; al hacer pull aparecerá el editor, en el que pondremos el mensaje de fusión. Los cambios se propagarán a la rama remota haciendo git push y las ramas quedarán como aparece en la visualización de la red con fecha 6 de abril de 2014; master se ha fusionado con get-dir.

También podemos hacer la operación inversa. Visto que los cambios de master no afectan a la funcionalidad nueva que hemos creado, fusionemos la rama get-dir en la principal. Cambiamos primero a ésta:

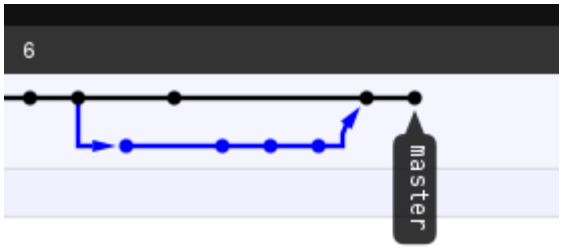
```
git checkout master
```

git checkout *saca* del árbol los ficheros correspondientes (lo que puede afectar a los editores y a las fechas de los mismos, que mostrarán la del último checkout si no se han modificado) y nos deposita en la rama principal, desde la cual podemos fusionar, usando también pull:

```
jmerelo@penny:~/txt/docencia/repo-tutoriales/repo-ejemplo$ git pull origin get-dir De github.com:oslugr/repo-ejemplo * branch get-dir -> FETCH_HEAD Updating df46a37..3705af0 Fast-
```

forward package.json | 5 +++-- web.js | 18 ++++++++++++------ 2 files changed, 15 insertions(+), 8 deletions(-)

que, dado que no hemos hecho ningún cambio en el mismo fichero, fusiona sin más problema la rama. En caso de que se hubiera modificado las mismas líneas, es decir, que los *commits* hubieran creado una divergencia, se habría provocado un conflicto que se puede solucionar como se ha visto en el apartado correspondiente. Pero, dado que no se la ha habido, el resultado final será el que se muestra en el gráfico.



La rama, una vez fusionada con el tronco principal, se puede considerar una rama muerta, así que nos la cargamos:

jmerelo@penny:~/txt/docencia/repo-tutoriales/repo-ejemplo$ git branch -d get-dir Deleted branch get-dir (was 3705af0).

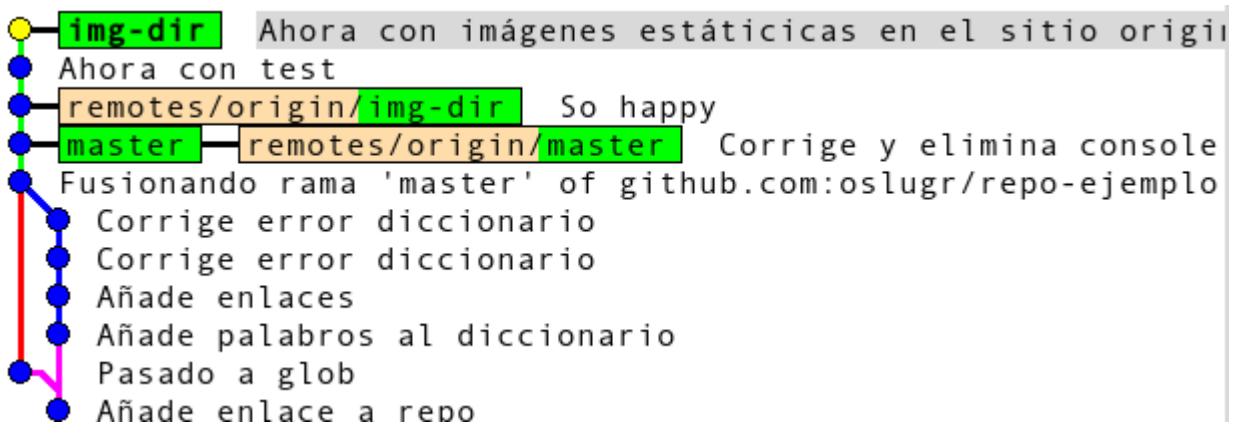
Pero eso borra solamente la rama local. Para borrarla remotamente:

jmerelo@penny:~/txt/docencia/repo-tutoriales/repo-ejemplo$ git push origin :get-dir To git@github.com:oslugr/repo-ejemplo.git - [deleted] get-dir

Una sintaxis con: que es ciertamente poco lógica, pero efectiva. Con eso tenemos la rama borrada tanto local como remotamente.

Los misterios del rebase

git tiene múltiples formas de reescribir la historia, como si de un régimen totalitario se tratara. Una de las más simples es *aplanar* la historia como si todos los *commits* hubieran sucedido unos detrás de otros, en vez de en múltiples ramas como es la forma habitual de trabajar (en un flujo de trabajo *rama por característica* como hemos visto anteriormente). Por ejemplo, podemos crear una rama img-dir (sobre el repositorio de ejemplo, añade a la aplicación de forma que se pueda trabajar con las imágenes del tutorial) dejando el repo en el estado que se muestra a continuación.

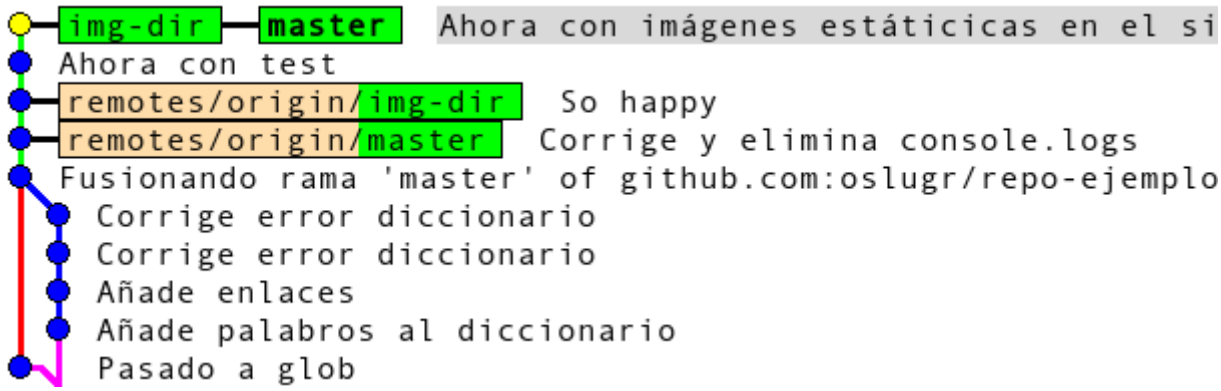


tomada desde la propia rama, donde hay una rama img-dir con un par de *commits* a partir del máster (dos puntitos azules más abajo).

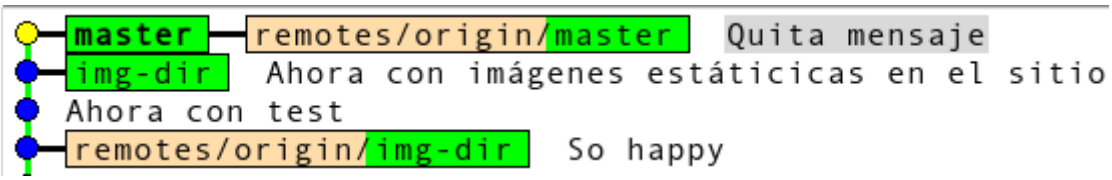
Una vez acabado el trabajo con la rama, cambiamos a master(git checkout master) y podemos hacer rebase:

git checkout master Switched to branch 'master' jmerelo@penny:~/txt/docencia/repo-tutoriales/repo-ejemplo$ git rebase img-dir First, rewinding head to replay your work on top of it... Fast-forwarded master to img-dir.

Dejando el repositorio en el estado siguiente:



El último commit es ahora parte de la rama master. No sólo se han fusionado los cambios en la rama principal, como se ve más abajo en la misma imagen e hicimos con la rama creada anteriormente, get-dir. En este caso, y a todos los efectos, se ha *reescrito la historia*, pasando los commits hecho sobre la rama anterior a formar parte de la rama principal. Una vez hecho esto, se limpia eliminando la rama creada. Sin embargo, un rebase no elimina una rama, que sigue ahí, sólo que en una parte diferente del árbol como se muestra a continuación

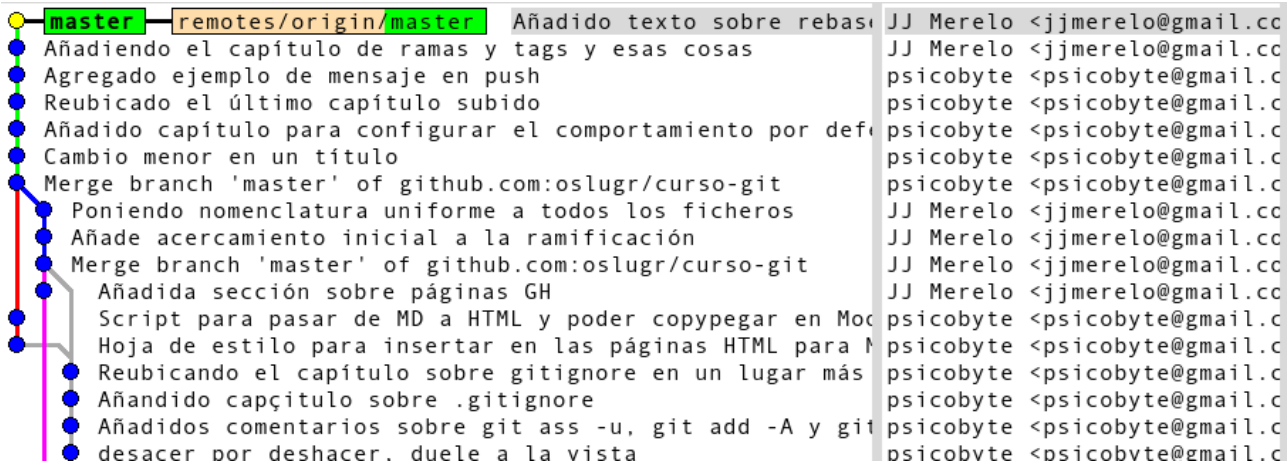


Sin embargo, ahora la rama es poco menos que un *tag* como el que hemos visto antiguamente. No estorba así que no hace falta borrarla.

Quién hizo qué

Con todas estas ramificaciones es posible que, en un momento determinado, sea difícil saber quién ha hecho qué cambio. Esto puede ser importante no sólo para repartir las culpas cuando algo falle, sino también para ver quién se responsabiliza de cada rama o característica y, eventualmente, también para asignar méritos.

La herramienta gitk que hemos usado hasta ahora te presenta en forma de árbol los cambios que se han venido haciendo en el repositorio, con un panel a la derecha que muestra quién ha hecho cada commit:



En esta imagen se ve como cada commit está asignado a uno de los autores de este tutorial, junto con los mensajes correspondientes. Con git log --pretty=short se puede conseguir un efecto similar en la línea de órdenes:

commit 3b89bd2fffbf7f5988de16b9911b14d70c9197bd Author: JJ Merelo Añadido texto sobre rebase

commit c09756d4d296fbacd9541d2d7c23e7710a5d1f09 Author: JJ Merelo Añadiendo el capítulo de ramas y tags y esas cosas

commit 8e4559325032fe1425288c4d1ab51fb7072f79b1 Author: psicobyte Agregado ejemplo de mensaje en push

(una vez más, sin líneas en blanco), con una muestra del mensaje corto y del commit junto con el autor. log es muy flexible y permite poner cualquier tipo de formato, pero hay todavía más herramientas. git blame permite hacer lo mismo sobre un fichero, viendo quién ha modificado cada una de las líneas. Por ejemplo, git blame uso_basico.md devolvería, entre otras cosas, estas líneas:

6017d70c (Manu 2014-03-28 22:30:40 +0100 70) [GUI Mac](http://mac.github.c 2feb1052 (psicobyte 2014-03-10 03:35:49 +0100 71) 6017d70c (Manu 2014-03-28 22:30:40 +0100 72) [GUI Windows](http://windows. 2feb1052 (psicobyte 2014-03-10 03:35:49 +0100 73) 01b9da5a (Manu 2014-03-28 22:36:46 +0100 74) [GUI for Linux, Windows y Mac 75b8e467 (Manu 2014-03-28 22:29:29 +0100 75) 2feb1052 (psicobyte 2014-03-10 03:35:49 +0100 76) ##Empezando a usar git

que muestran que la línea 70 y la 71 han sido modificadas por Manu mientras que el resto lo han sido por Pablo. El formato que siguen es el hash del commit seguidos por el nombre del usuario, la fecha, el número de línea; finalmente está el contenido de la línea.

Algo un poco más vistoso se puede ver en algunos repositorios como GitHub, pulsando sobre el botón *Blame* que aparece en cada uno de los ficheros:

2feb1052 » psicobyte 2014-03-10 Segundo módulo, borrador	61	###En Mac
01b9da5a » Makova 2014-03-28 Update uso_basico.md	62	Hay dos maneras de instalar Git en Mac, la más fácil es utilizar el instalado
75b8e467 » Makova 2014-03-28 Añadir Guis SO y Git para Mac	63	
6738e67d » Makova 2014-03-28 Git Mac	64	[Git for OS X](https://code.google.com/p/git-osx-installer/)
2feb1052 » psicobyte 2014-03-10 Segundo módulo, borrador	65 66 67	##Clientes GUI para Linux, Windows y Mac

Esta es simplemente una visualización del comando anterior, que presenta además un enlace al usuario en GH en caso de serlo (porque, recordemos, git es un DVCS cuyos cambios pueden haberse fusionado en local por parte de cualquier tipo de usuario, que no tiene por qué estar necesariamente en GitHub). Con blame se puede saber incluso quien modificó una línea en particular. Pero, para un uso básico, basta lo anterior.

Recursos

Algunos videos y tutoriales que pueden complementar el módulo.

Ramificaciones en Git - Procedimientos básicos para ramificar y fusionar

Video: Creación de ramas y explicación de las mismas

Manejo de ramas de desarrollo con git

GitHub, el sistema de control de fuentes social.

GitHub se ha convertido en el sitio más popular, a pesar de encontrarse entre una cantidad de sitios que alojan también proyectos y permiten usar Git como el extinto Gitorious, Gitlab, BitBucket o incluso el venerable SourceForge; Gitorious, a pesar de haber desaparecido, tiene la ventaja de que está a su vez basado en software libre, por lo que te puedes instalar tu propia copia del repositorio bajo tu control. Sea con este software o con GitLab te puedes hacer tu propia instalación de git si tienes disponible un servidor para ello. Evidentemente, para trabajar con grandes proyectos privados son una buena opción y lo más aceptable.

Sin embargo, hay buenas razones para usar GitHub. Aparte del uso de git, GitHub en realidad funciona como una comunidad de programadores que te permite interaccionar muy fácilmente con otros programadores que usen las mismas herramientas que uno. Ninguno de los otros repositorios tiene esas características; aunque todos tienen ciertas capacidades *sociales*, lo cierto es que la mayoría de los programadores de software libre usan hoy GitHub. De hecho, GitHub con sus métricas, características de *gamificación* (poner estrellas a proyectos, por ejemplo), continuo desarrollo, enganche a plataformas de programación diversas y cliente móvil es hoy en día la mejor opción para desarrollar software libre (y novelas y cursos de Git).

Adicionalmente, GitHub se usa como repositorio por defecto para módulos de diferentes lenguajes de programación; por ejemplo, npm, el gestor de módulos de Node, lo usa para alojar y descargar las fuentes; otros marcos como Joomla lo usan también para definir sus plugins. Incluso GitHub se incluye ya en flujos de trabajo de ciertos entornos, como la (futura o pasada) revista científica Push.

En resumen, GitHub es la primera, segunda y tercera mejor opción de un programador de software libre a la hora de alojar sus proyectos. La cuarta sería Gitorious, por el hecho de que efectivamente todo el software que usa está liberado y la quinta Bitbucket. Más atrás estarían el resto. En este curso veremos principalmente, por esa razón, GitHub y sus características específicas, algunas de las cuales se han venido usando ya el resto del curso.

Repositorios públicos y privados

Por omisión, los repositorios de GitHub son públicos. Para conseguir un repositorio privado hay que hacer una de dos cosas

- Pagar. Por 7 dólares al mes puedes tener cinco repositorios privados.
- Conseguir una cuenta para educación que permite, generalmente de forma limitada, tener un número determinado de cuentas gratuitas.

En realidad, las cuentas privadas son útiles para empresas que quieran desarrollar sus propias aplicaciones en privado; para software libre, aplicaciones que se creen para un proyecto las cuentas tienen espacio de almacenamiento ilimitado y un número ilimitado de colaboradores, por lo que no hace falta adquirir una cuenta ni tener un repositorio privado.

El GitHub *social*

En realidad, GitHub es una red social de gente que hace cosas y escribe texto como este o aplicaciones en diferentes lenguajes de programación. Cada usuario tiene un perfil y en él puede contar cosas como dónde se encuentra, su web y poco más. Lo que importa es lo que se hace, que aparece justo al lado del logo. GitHub usa Gravatar para las fotos de perfil, por lo que tendrás que darte de alta en ese servicio *con la misma dirección de email que uses en GH* para que aparezca tu avatar junto a tus contribuciones. Se puede seguir la actividad de un usuario, pero también se puede ir un paso más allá y pulsar el botón de *Follow* con lo que, a entrar en la página principal de GitHub se te mostrará, junto con la actividad propia, la de esta persona. Una persona puede ser también añadida a un repositorio, lo que le dará privilegios para realizar todo tipo de acciones sobre él. Conviene usar esto con moderación y sólo cuando se trate de una persona ya involucrada en el proyecto.

El componente *social* también se ve en repositorios específicos: los repositorios, como este (antiguo) de ejemplo del curso, se pueden *vigilar* (*Watch*) y también poner una especie de "Me gusta", *Star*, uno al lado del otro. Finalmente, se puede hacer un *Fork* del repositorio, lo que copia el contenido completo del repositorio al propio y permite trabajar con él; equivale a una *rama* tal como la que hemos visto anteriormente. Este *fork* respeta la autoría original que sigue apareciendo en todos los *commits* que se hayan hecho originalmente, pero permite añadir uno sus propias modificaciones *sin que el autor original tenga que aprobarlas*. Los *pull requests* permiten colaboración esporádica, ya que las modificaciones que se soliciten pueden aprobarse o no por parte del autor

principal del repositorio; la persona que las haga, sin embargo, no tiene por qué estar añadida como colaborador permanente al mismo.

GitHub también permite comentar a diferentes niveles: se puede comentar un *commit*, se pueden comentar líneas de código y finalmente se pueden hacer solicitudes y comentarios a un repo completo usando *issues* o solicitudes (*to have an issue* significa literalmente tener un *asunto* o *problema*). Todas estas formas de interacción permiten tener una vida *social* más o menos rica y que haya muchas formas posibles de interaccionar con los autores de un proyecto y por supuesto también de que esos autores aumenten su *karma* a base de las conexiones, estrellas que reciban sus repositorios y comentarios, así como los *issues* resueltos.

Finalmente, los usuarios se pueden agrupar en organizaciones. Una organización es en muchos aspectos similar a un usuario; tiene las mismas limitaciones y las mismas ventajas, pero en una organización se definen *equipos* y los permisos para trabajar por repositorios se hacen usando estos *equipos*; cada repositorio puede tener uno o más equipos con diferentes niveles de privilegios y el repositorio en sí tendrá también un equipo que será el que pueda realizar ciertas acciones sobre el mismo. Generalmente se crea un equipo por repositorio, pero puede organizarse de cualquier otra forma.

Cuando uno es añadido a un equipo de una organización, se convierte en otra "personalidad" que te permite, por ejemplo, hacer *fork* como miembro de tal organización.

En resumen, la facilidad que tiene GitHub para manejar todo tipo de situaciones de desarrollo y la *gamificación* y *socialización* de la experiencia de desarrollo es lo que ha hecho que hoy en día tenga tanto éxito hasta el punto de que el perfil de uno en GitHub es su mejor carta de presentación a la hora de conseguir un trabajo en desarrollo y programación.

Cómo usar los hooks

Los *hooks* o *ganchos* son eventos que se activan cuando se produce algún tipo de acción por parte de git. En general, se usan para integrar el sistema de gestión de fuentes de git con otra serie de sistemas, principalmente de integración continua o entrega continua o, en general, cualquier tipo de sistema de notificaciones o de trabajo en grupo.

GitHub puede integrar cualquier tipo de servicio que acepte una petición REST con una serie de características (esencialmente, datos sobre el repositorio y sobre el último commit), pero tiene ya una serie de servicios, casi un centenar, configurables directamente desde el panel de control yendo a *Settings -> Webhooks & Services -> Configure services*. Todos los servicios se activan cuando se hace un push a GitHub.

Evidentemente, en local no se enteran, salvo que los configuremos explícitamente como vamos a ver en el tema siguiente.

Vamos a dividir los servicios que hay en varios grupos:

* Integración continua. Servicios como TravisCI, CircleCI, o Jenkins. Los tres se pueden configurar directamente desde GH. Estos servicios realizan una serie de test o generación de código sobre el proyecto y dan un resultado indicando qué test se han pasado o no. Para indicar que test se hacen y los parámetros del repositorio, cada uno usa un formato diferente, aunque son habituales los ficheros de formato YAML o XML.

* Servicios de mensajería diversos, que envían mensajes cuando sucede algo. Entre estos últimos está Twitter, que se puede configurar para que se cree un tweet con el mensaje del commit cada vez que se haga uno. Puede ser bastante útil, si se usa este sitio, para mantenerte al día de la actividad de un grupo de trabajo. También hay otros servicios como Jabber o Yammer o comerciales como Amazon SNS.

* Entrega continua: a veces integrados con los de, valga la redundancia, integración continua, pero que permiten directamente, cuando se hace un push sobre una rama determinada, se despliegue en el sitio definitivo. Servicios como Azure lo permiten, pero también CodeShip o Jenkins. Generalmente en este caso hay que configurar algún tipo de *secret* o *clave* que permita a GitHub acceder al sitio y depositar la *carga* que, inmediatamente, estará disponible. De hecho, es muy fácil trabajar con esto directamente desde el editor como Eclipse.

* Sistemas de trabajo en grupo, que integran GitHub con los sistemas que tengan de asignación de tareas, de resolución de incidencias incluidas por parte de clientes. Por ejemplo, Basecamp, Bugzilla o Zendesk. De hecho, el propio GitHub integra un sistema de incidencias que se puede usar fácilmente.

* Análisis del código como CodeClimate (que analiza una serie de parámetros del código), Depending (que analiza dependencias en PHP) o David-DM (que analiza dependencias para nodejs).

Lo interesante es que se puede trabajar con la mayoría de estos sistemas de forma gratuita, aunque algunos tienen un modelo *freemium* que te cobra a partir de un nivel determinado de uso (lo que es natural, si no, no podrían ofrecértelo de forma gratuita). Además, integra la mayor parte de los sistemas que se usan habitualmente en la industria del software.

Algunos *hooks* interesantes: sistemas de integración continua

GitHub resulta ideal para trabajar con cualquier sistema de integración continua, sea alojado o propio. Los sistemas de integración continua funcionan de la forma siguiente:

* Provisionan una máquina virtual con unas características determinadas para ejecutar pruebas o compilar código.

* Instalan el software necesario para llevar a cabo dichas pruebas.

* Ejecutan las pruebas, creando finalmente un informe que indique cuantas han fallado o acertado.

* Crear un *artefacto*, que puede ir desde un fichero con el informe en un formato estándar (suele ser XUnit o JUnit) hasta el ejecutable que se podrá descargar directamente del sitio; esto último puede incluir también su despliegue en la *nube*, un IaaS (Infraestructure as a Service) o PaaS (Platform as a Service) en caso de que haya pasado todos los test satisfactoriamente.

La integración continua forma parte de una metodología de desarrollo basado en test o guiado por pruebas que consiste en crear primero las pruebas que tiene que pasar un código antes de, efectivamente, escribir tal código. Las pruebas son test unitarios y también de integración, que prueban las capas de la aplicación a diferentes niveles (por ejemplo, acceso a datos, procesamiento de los datos, UI).

Todos los lenguajes de programación moderno incluyen una aplicación que crea un protocolo para llevar a cabo los test e informar del resultado y estos sistemas van desde el humilde Makefile que se usa en diferentes lenguajes compilados hasta el complejo Maven, pasando por sistemas como los test de Perl o los Rakefiles de Ruby. En cualquier caso, cada lenguaje suele tener una forma estándar de pasar los test (make test, npm test o mocha) y los sistemas de integración continua hacen muy simple trabajar con estos test estándar, pero también son flexibles en el sentido que se puede adaptar a todo tipo de programa.

Veamos como trabajar con Travis. Se hace siguiendo estos pasos:

1. Darse de alta en Travis CI usando la propia cuenta de GitHub
2. Activar el *hook* en tu perfil de Travis.
3. Se añade el fichero .travis.yml a tu repositorio. Este dependerá del lenguaje que se esté usando, aunque si lo único que quieres es comprobar la ortografía de tus documentos, lo puedes hacer como en el repositorio ejemplo
4. Hacer push.

La mayoría de estos repositorios suelen usar un fichero en formato estándar, YAML, XML o JSON. Veamos qué hace el fichero que hemos usado para el repo ejemplo:

branches: except: - gh-pages language: C compiler: - gcc before_install: - sudo apt-get install aspell-es script: OUTPUT=`cat README.md | aspell list -d es -p ./.aspell.es.pws`; if [-n "$OUTPUT"]; then echo $OUTPUT; exit 1; fi

La estructura de YAML permite expresar vectores y matrices asociativas fácilmente. En general, nos vamos a encontrar con algo del tipo variable: valor que será una clave y el valor correspondiente; el valor, a su vez, puede incluir otras estructuras similares. Por ejemplo, la primera:

branches: except: - gh-pages

es una clave, branches, que incluye otra clave, except, que a su vez apunta a un vector (diferentes valores precedidos por -) con las ramas que vamos a excluir. En este caso, gh-pages, la de las páginas. Si hubiera otra rama a excluir, iría de esta forma:

branches: except: - gh-pages - a-excluir

es decir, como un array de dos componentes. Esto hará que, sobre nuestro repositorio, se prueben todas las ramas, inclusive por supuesto la máster. A continuación, expresamos el lenguaje que se va a usar; Travis no admite cualquier lenguaje, pero algunos como C, Perl o nodejs los acepta sin problemas. Como hay varios compiladores posibles, a continuación, le decimos qué compilador se va a usar (en realidad, no se usa ninguno). En otros lenguajes habría que decir qué intérprete o qué versión; estas claves son específicas del lenguaje.

before_install: - sudo apt-get install aspell-es script: OUTPUT=`cat README.md | aspell list -d es -p ./.aspell.es.pws`; if [-n "$OUTPUT"]; then echo $OUTPUT; exit 1; fi

Como lo único que vamos a hacer en este caso es comprobar la ortografía del texto del fichero README.md, instalamos con apt-get (herramienta estándar para Linux) un diccionario en español; este instalará todas las dependencias a su vez. Finalmente, la orden marcada script es la que lleva a cabo la comprobación. Para un programa normal sería suficiente hacer make test (y definir las dependencias para este objetivo, claro). No nos preocupemos mucho por lo que es, sino por lo que hace: si hay alguna palabra que no pase el test ortográfico, fallará y enviará un mensaje de correo electrónico a la persona que haya hecho un commit indicándolo. Si lo pasa sin problemas, también enviará el mensaje indicando que todo está correcto. Este tipo de cosas resulta útil sólo por el hecho de que se ejecuten automáticamente, pero pueden servir también para hacer despliegues continuos.

Travis también proporciona un *badge* que puedes incluir en tu repositorio para indicar si pasa los test o no, que puedes incluir en tu fichero README.md (o donde quieras) con este código:

[![Build Status](https://travis-ci.org/oslugr/repo-ejemplo.svg?branch=master)](https://travis-ci.org/oslugr/repo-ejemplo)

sustituyendo el nombre de usuario y el nombre del repo por el correspondiente, claro. Este código está escrito en MarkDown, y GitHub lo interpretará directamente sin problemas, aunque lo mejor es que pinches en la imagen que aparece arriba a la derecha que te dará el código correspondiente.

Usando un cliente de línea de órdenes de GitHub

GitHub también mantiene un cliente de GitHub, escrito en Ruby y llamado hub, que se puede usar para sustituir a git o por sí mismo. En realidad, es como git salvo que tiene ya definidos por omisión una serie de características específicas de GitHub, como los nombres de los repositorios o los usuarios de los mismos. Tras instalarlo puedes usarlo, por ejemplo, para clonar el repo de ejemplo usado aquí con hub clone oslugr/repo-ejemplo en vez de usar el camino completo a git; el formato sería siempre usuario/nombre-del-repo. Más órdenes que añade a git (y que se pueden usar directamente desde git si se usa, como se indica en las instrucciones, git como un alias de hub):

- hub browse: abre un navegador en la página del repositorio correspondiente. Por ejemplo, hub browse -- issues lo abriría en la página correspondiente a las solicitudes de ese proyecto.
- hub fork: una vez clonado un repositorio de otro usuario, no hace falta hacer *fork* desde la web, se puede hacer directamente desde el repo. Se crea un origen remoto con el nombre de tu usuario, al que se puede hacer *push* de la forma normal. Desde la misma línea de órdenes se puede hacer un *pull request* al repositorio original también.
- Como usuario puedes aplicar también los *pull requests* desde línea de órdenes.
- hub compare permite comparar entre diferentes tags o versiones o ramas.

En general, ya que se tiene GitHub, conviene usar este cliente, sea o no con un alias a git. Por lo menos su uso es conveniente.

Clientes de línea de órdenes y lanzamientos

Listo para el lanzamiento: publicación en GitHub

GitHub, como cualquier otro repositorio git, permite usar una rama específica para depositar las versiones descargables; una forma de hacerlo, por ejemplo, es usar la rama gh-pages para no mezclarlo con el resto de los ficheros que están en el repositorio y, por tanto, versionados. Sin embargo, no es una buena idea poner ficheros binarios bajo control de git, porque es muy fácil provocar conflictos con ellos y no tan fácil resolverlos (o es un fichero o es otro, los algoritmos de diferencias de texto no trabajan sobre ficheros que no sean binarios); además, en general, estos ficheros se generan a partir del fuente de una forma más o menos automática: usando Makefiles, por ejemplo, en C, o en general compilando; si se trata de paquetes, cada lenguaje tiene mecanismos específicos para crearlos a partir de los fuentes, por lo tanto no es necesario colocar tales ficheros en el repositorio. Por eso es mejor colocarlos *fuera* del repositorio, y es lo que hace GitHub, en un apartado llamado *archivos*.

Crear un lanzamiento es fácil en GitHub: simplemente se crea una etiqueta como se ha visto anteriormente, con git tag. Por ejemplo, este es el fichero correspondiente a la etiqueta v0.0.1 que se definió en el repositorio de ejemplo. Al pinchar en *Releases* te aparecen las versiones que ya has creado o un botón con *Draft a new release* para crear una nueva.

Desde este interfaz web se puede añadir alguna información más que desde la línea de comandos: se puede crear la etiqueta si no existe y se pueden añadir imágenes, ficheros binarios generados de cualquier otra forma (o automáticamente) y, en general, lo que uno desee. También se puede marcar como *pre-release* y darle un título como a las versiones de Ubuntu, con animalitos o nombres de días de la semana o lo que sea.

En general, si no se usa ningún repositorio de módulos o aplicaciones para publicar la aplicación, o simplemente se quiere publicar junto con las fuentes, manuales y lo que se desee, es conveniente usar esta característica de GitHub para mantener un archivo de versiones descargables de la misma y también para que puedan acceder a ella fácilmente quienes no quieran usar simplemente git para descargársela.

> Vais a decir que ya podían instalarse git y demás herramientas necesarias para compilar o ejecutar la aplicación, pero en muchos casos no tiene por qué ser fácil o factible; no se va a instalar uno un compilador de fortran simplemente para compilar una aplicación nuestra, por ejemplo.

Recursos adicionales

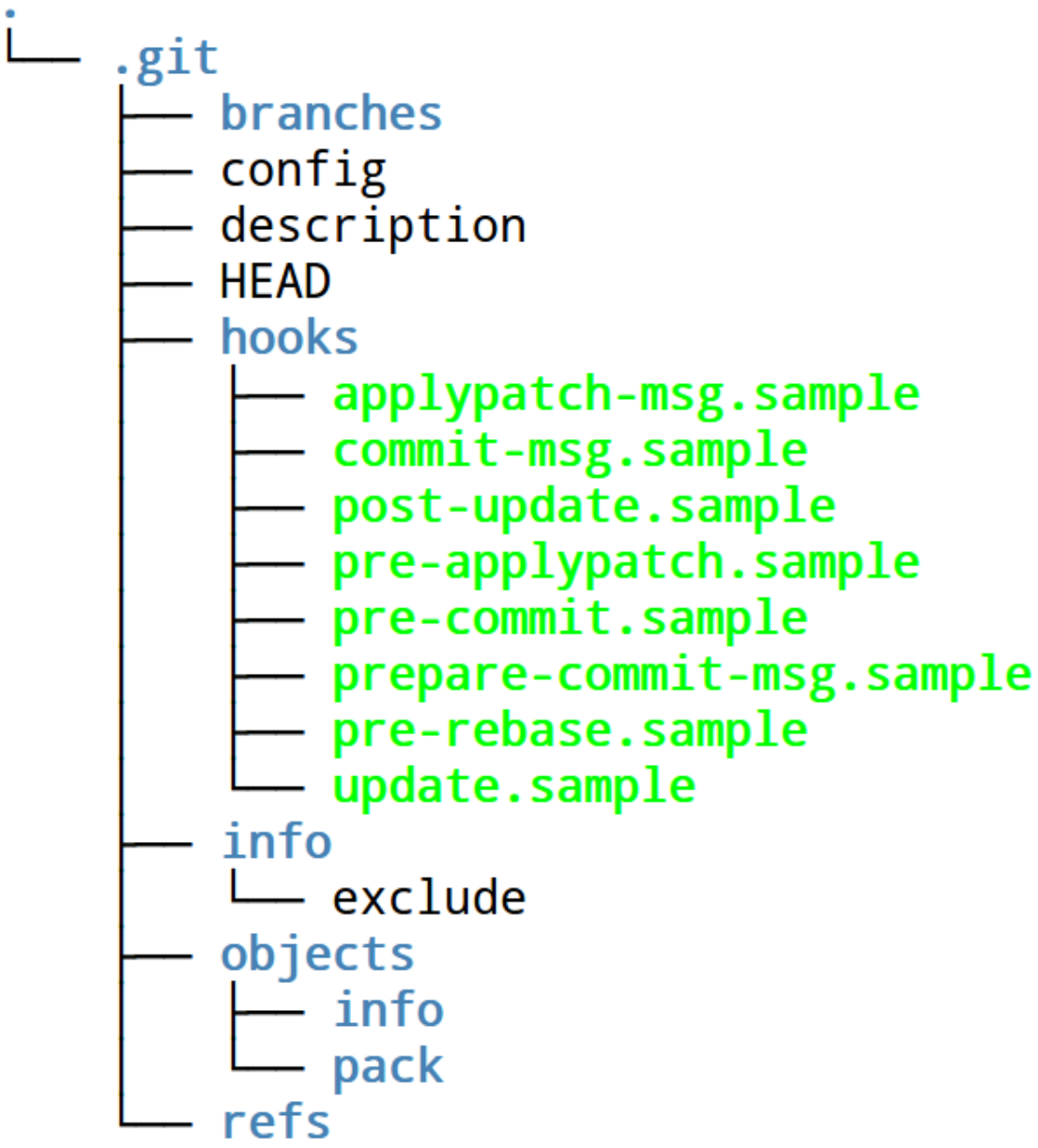
No son imprescindibles, pero pueden ayudarte a entender este tema:

Tutorial de Git y Github

Guapo Linux estrena GitHub

Viendo las cañerías: estructura de un repositorio git

Cuando se crea por primera vez un repositorio veremos que aparecen misteriosamente una serie de ficheros con esta estructura dentro del directorio .git.



branches lo dejamos de lado porque ya no se usa (aunque por alguna razón se sigue creando). config, HEAD, refs y objects son ficheros o directorios que almacenan información dinámica, por ejemplo, config almacena las variables de configuración (que se han visto anteriormente). El resto de los ficheros y directorios se copian de una *plantilla*; esta plantilla se instala con git y se usa cada vez que hacemos clone o init, y contiene hooks, description, branches e info y los ficheros que se encuentran dentro de ellos.

Esta plantilla la podemos modificar y cambiar. La plantilla que se usa por omisión se encuentra en /usr/share/git-core/templates/ y contiene una serie de ficheros junto con ejemplos (*samples*) para ganchos. Sin embargo, podemos personalizar nuestra plantilla editando (con permiso de superusuario) estos ficheros o bien usando la opción --template <nombre de directorio> de clone o init. En ese caso, en vez de copiar los ficheros por omisión, copiará los contenidos en ese directorio.

Por ejemplo, se puede usar esta plantilla que elimina los ficheros de ejemplo, sustituye por otro y traduce los contenidos de los otros ficheros al castellano; también mete en los patrones ignorados (sin necesidad de usar .gitignore) los ficheros que terminan en ~, que produce Emacs como copia de seguridad.

Estos ficheros forman parte de las cañerías de git y podemos cambiar su comportamiento editando config como ya se ha visto en el capítulo de uso básico; de hecho, existe también un fichero de configuración a nivel global, .gitconfig que sigue el mismo formato y que ya hemos visto:

[alias] ci = commit st = status [core] editor = emacs

Estos ficheros de configuración siguen un formato similar al de los ficheros .ini, es decir, bloques definidos entre corchetes y variables con valor, dentro de ese bloque, a las que se le asigna usando =. En este caso definimos dos alias y un editor o, mejor dicho, *el* editor. Esto podemos hacerlo tanto en el fichero global como en el local si queremos que afecte sólo a nuestro repositorio.

Otro fichero dentro de este directorio que se puede modificar es. git/info/exclude; es similar a .gitignore, salvo que en este caso afectará solamente a nuestra copia local del repositorio y no a todas las copias del mismo. Por ejemplo, podemos editarlo de esta forma

git ls-files --others --exclude-from=.git/info/exclude # Lines that start with '#' are comments. # For a project mostly in C, the following would be a good set of # exclude patterns (uncomment them if you want to use them): # *.[oa] *~ para excluir los ficheros de copia de seguridad de Emacs (que hemos definido antes como editor) que nos

interesa evitar a nosotros, pero que puede que tengan un significado especial para otro usuario del repo y que por tanto no quiera evitar.

Por supuesto, el tema principal de este capítulo está en el otro directorio, *hooks*, cuyo contenido tendremos que cambiar si queremos añadir ganchos al repositorio. Pero para usarlo necesitamos también conocer algunos conceptos más de git, empezando por cómo se accede a más cañerías.

Paso a paso

Si miramos en el directorio .git/objects encontraremos una serie de directorios con nombres de dos letras, dentro de los cuales están los objetos de git, que tienen otras 38 letras para componer las 40 letras que componen el nombre único de cada objeto; este nombre se genera a partir del contenido usando SHA1.

git almacena toda la información en ese directorio y de hecho está organizado para acceder a la información almacenada por contenido. Por eso tenemos que imaginarlo como un sistema de ficheros normal, con una raíz (que es HEAD, el punto en el que se encuentra el repositorio en este momento) y una serie de ramas que apuntan a ficheros y a diferentes versiones de los mismos.

git, entonces, procede de la forma siguiente.

1. Crea un SHA1 a partir del contenido del fichero cambiado o añadido. Este fichero se almacena en la zona temporal en forma de *blob*.
2. El nombre del fichero se almacena en un árbol, junto con el nombre de otros ficheros que estén, en ese momento, en la zona temporal. Un árbol almacena los nombres de varios ficheros y apunta al contenido de los mismos, almacenado como *blob*. De este objeto, que tiene un formato fijo, se calcula también el SHA1 y se almacena en .git/objects. Un árbol, a su vez, puede apuntar a otros árboles creados de la misma forma.
3. Cuando se hace *commit*, se crea un tercer tipo de objeto con ese nombre. Un *commit* contiene enlaces a un árbol (el de más alto nivel) y metadatos adicionales: quién lo ha hecho, cuándo y por supuesto el mensaje de commit.

Veremos más adelante cómo se listan ficheros de todos estos tipos, pero por lo pronto la idea es que un comando de git de alto nivel involucra varias órdenes de bajo nivel que, eventualmente, van a parar a información que se almacena en un directorio determinado con nombres de fichero que se calculan usando SHA1, aparte de que se puede actuar a diferentes niveles, desde el más bajo de almacenar un objeto directamente en un árbol o crear un commit “a mano” hasta el más alto (que es el que estamos acostumbrados). Este sistema, además, asegura que no se pierda ninguna

información y que podamos acceder al contenido de un fichero determinado hecho en un momento determinado de forma fácil y eficiente. Pero para poder hacerlo debe haber una forma única y también compacta de referirse a un elemento determinado dentro de ese repositorio. Es lo que explicaremos a continuación.

El nombre de las cosas: refiriéndonos a objetos en git.

Como ya hemos visto antes, todos los objetos (sean *blobs*, árboles o *commits*) están representados por un SHA1. Si conocemos el SHA1, se puede usar show, por ejemplo, para visualizarlo. Haciendo git log veremos, por ejemplo, los últimos commits y si hacemos show sobre uno de ellos,

git show fe88e5eefff7f3b7ea95be510c6dcb87054bbcb commit fe88e5eefff7f3b7ea95be510c6dcb87054bbcb0 Author: JJ Merelo <jjmerelo@gmail.com> Date: Thu Apr 17 18:29:11 2014 +0200 Añade layout diff --git a/views/layout.jade b/views/layout.jade new file mode 100644 index 0000000..36cc059 --- /dev/null +++ b/views/layout.jade @@ -0,0 +1,6 @@ [....]

El mismo resultado que obtendríamos si hacemos git show HEAD, que recordemos que es una referencia que apunta al último commit. También obtendremos lo mismo si hacemos git show master. En cualquiera de los casos, lo que está mostrando es un objeto de tipo *commit*, el último realizado.

Pero veremos cómo funciona este último ejemplo. Al lado del directorio objects está el directorio refs, que almacena referencias y que es como git sabe a qué commit corresponde cada cosa. Este comando:

~/txt/docencia/repo-tutoriales/repo-ejemplo<master>$ tree .git/refs/

├── heads

│ ├── img-dir

│ └── master

├── remotes

│ ├── heroku

│ │ └── master

│ └── origin

│ ├── HEAD

│ ├── img-dir

```
|       └── master
└── tags
        ├── v0.0.1
        ├── v0.0.2
        ├── v0.0.2.1
        └── v0.0.3 5 directories, 10 files
```

muestra todo lo que hay almacenado en este directorio: referencia a las ramas locales en heads y a las remotas en remotes. Si mostramos el contenido de los ficheros:

~/txt/docencia/repo-tutoriales/repo-ejemplo<master>$ cat .git/refs/heads/master fe88e5eefff7f3b7ea95be510c6dcb87054bbcb0

Que muestra que, efectivamente, el hash del commit es el que corresponde:

> Podemos mirar en .git/objects/fe a ver si efectivamente se encuentra; puedes hacerlo sobre tu copia del repositorio repo-ejemplo, ya que los hash son iguales en todos lados.

Como hemos visto anteriormente, un *commit* apunta a un árbol. Podemos indicarle a show que nos muestre este árbol de esta forma:

~/txt/docencia/repo-tutoriales/repo-ejemplo<master>$ git show master^{tree} tree master^{tree} .aspell.es.pws .gitignore .gitmodules .travis.yml LICENSE Procfile README.md curso package.json shippable.yml test/ views/ web.js

En este caso el formato es rama (circunflejo o *caret*) {tree}; el circunflejo se usa en la selección de referencias de git para cualificar lo que se encuentra antes de ella, pero no hay muchas más opciones aparte de tree, pero sí podemos acceder a versiones anteriores del repositorio y a sus ficheros.

~/txt/docencia/repo-tutoriales/repo-ejemplo<master>$ git show master~1

commit 5be23bb2a610260da013fcea807be872a4bd6981 Author: JJ Merelo <jjmerelo@gmail.com> Date: Thu Apr 17 17:42:39 2014 +0200 Aclara una frase [...]

La tilde ~ indica un ancestro, es decir, el *padre* del commit anterior, que, como vemos corresponde al commit 5be23bb. Podemos ir más allá hasta que nos aburramos: ~2 accederá al padre de este y así sucesivamente. Y, por supuesto, podemos cualificarlo con ^{tree}^ para que nos muestre el árbol en el estado que estaba en ese commit. Y también para que nos muestre un fichero sin necesidad de sacarlo del repositorio:

~/txt/docencia/repo-tutoriales/repo-ejemplo<master>$ git show master~2:README.md

repo-ejemplo ========== Ejemplo de repositorio para trabajar en el [curso de `git`](http://cevug.ugr.es/git) el contenido del cual está [...]

> En esta sección hemos usado show para mostrar las capacidades de los diferentes selectores de git, pero se pueden usar con cualquier otra orden, como checkout o cualquiera que admita, en general, una referencia a un objeto.

Comandos de alto y bajo nivel: *fontanería* y *loza*

Para entendernos, todas las órdenes que hemos usado hasta ahora son *loza*. Es decir, es el *interfaz* del usuario de toda la instalación de fontanería que lleva a cabo realmente la labor de quitar de en medio lo que uno deposita en las instalaciones sanitarias. Pero por debajo de la loza y pegado a ella, están las cañerías y toda la instalación de fontanería.

Los comandos de git se dividen en dos tipos: *fontanería* o *cañería*, que son comandos que *generalmente* no ve el usuario y *loza*, que son los que ve y los que usa. Sin embargo, este capítulo trata realmente de esa fontanería, porque van a ser una serie de órdenes que se van a llevar a cabo *después* de que se ejecuten las órdenes de *loza*, o, quizás *dentro* de esas órdenes de loza.

Pero antes de usar esas órdenes de fontanería tenemos que entender cómo son las cañerías. Una parte se ha visto anteriormente: el *index* o índice que contiene todos los objetos a los que git debe prestarle atención a la hora de hacer un commit. Pero existen además los objetos y las referencias.

Los *objetos* son, en general, información que está almacenada en el repositorio. Incluyen, por supuesto, los ficheros que almacenamos en el mismo, pero también los mensajes de commit, las etiquetas y los *árboles*. Los ficheros almacenados están *divididos*: el *contenido* del fichero se almacena en un *blob* y el nombre del fichero se almacena en el árbol. Hay, pues, cuatro tipos de objetos: *blob*, *tree*, *commit* y *tag*.

La orden ls-tree nos permite ver qué tipos de objetos tenemos almacenados y sus códigos SHA1, que son los nombres de ficheros calculados a partir del contenido del mismo. Aunque todos están almacenados en el directorio .git/objects, esta orden nos permite ver también de qué tipo son:

```
~/txt/docencia/repo-tutoriales/repo-ejemplo<master>$ git ls-tree HEAD 100644 blob
a6f69e4284566cd84272c6a4e4996f64643afbea    .aspell.es.pws    100644    blob
a72b52ebe897796e4a289cf95ff6270e04637aad    .gitignore    100644    blob
cc5411b5557f43c7ba2f37ad31f8dc34cccda075    .gitmodules    100644    blob
4e7b6c1b5a6cb3a962ea05874d10c943c1923f39    .travis.yml    100644    blob
d5445e7ac8422305d107420de4ab8e1ee6227cca    LICENSE    100644    blob
```

d1913ebe4d9e457be617ee0e786fc8c30a237902 Procfile 100644 blob
c5badda0c484c989e958ea4e27dfe11d69f3c8ef README.md 160000 commit
fa8b7521968bddf235285347775b21dd121b5c11 curso 100644 blob
f8c35adaf57066d4329737c8f6ec7ce6179cc221 package.json 100644 blob
08827778af94ea4c0ddbc28194ded3081e7b0f87 shippable.yml 040000 tree
39da6b155c821af1e6a304daca9b66efb1ac651f test 100644 blob
94f151d9ef9340c81989b0c3fa8c517c068e1864 web.js

En este caso tenemos objetos de tres tipos: blob, commit y tree. A ls-tree se le pasa un *tree-ish*, es decir, algo que apunte a dónde esté almacenado un árbol, pero para no preocuparnos de qué se trata esto, usaremos simplemente HEAD, que apunta como sabéis a la punta de la rama en la que nos encontramos ahora mismo. También nos da el SHA1 de 40 caracteres que representa cada uno de los ficheros.

Si queremos que se expandan los tree para mostrar los ficheros que hay dentro también usamos la opción -r:

~/txt/docencia/repo-tutoriales/repo-ejemplo<master>$ git ls-tree -r HEAD 100644 blob
a6f69e4284566cd84272c6a4e4996f64643afbea .aspell.es.pws 100644 blob
a72b52ebe897796e4a289cf95ff6270e04637aad .gitignore 100644 blob
cc5411b5557f43c7ba2f37ad31f8dc34cccda075 .gitmodules 100644 blob
4e7b6c1b5a6cb3a962ea05874d10c943c1923f39 .travis.yml 100644 blob
d5445e7ac8422305d107420de4ab8e1ee6227cca LICENSE 100644 blob
d1913ebe4d9e457be617ee0e786fc8c30a237902 Procfile 100644 blob
da5b5121adb42e990b9e990c3edb962ef99cb76a README.md 160000 commit
fa8b7521968bddf235285347775b21dd121b5c11 curso 100644 blob
f8c35adaf57066d4329737c8f6ec7ce6179cc221 package.json 100644 blob
08827778af94ea4c0ddbc28194ded3081e7b0f87 shippable.yml 100644 blob
9920d80438d42e3b0a6924a0fcace2d53a6af602 test/route.js 100644 blob
36cc059186e7cb247eaf7bfd6a318be6cffb9ea3 views/layout.jade 100644 blob
97c32024cda29e0fb6abebf48d3f6740f0acb9e2 web.js

que muestra solo los objetos de tipo blob (y un commit) con el camino completo que llega hasta ellos.

Si editamos un fichero tal como el README.md, tras hacer el commit tendrá esta apariencia:

~/txt/docencia/repo-tutoriales/repo-ejemplo<master>$ git ls-tree HEAD100644 blob
a6f69e4284566cd84272c6a4e4996f64643afbea .aspell.es.pws [...] 100644 blob
da5b5121adb42e990b9e990c3edb962ef99cb76a README.md

Como vemos, ha cambiado el SHA1. Pero ls-tree va más allá y te puede mostrar también cuál es el estado del repositorio hace varios commits. Por ejemplo, podemos usar HEAD^ para referirnos al commit anterior y git ls-tree HEAD^ nos devolvería exactamente el mismo estado en el que estaba antes de hacer la modificación a README.md. De hecho, podemos usar también la abreviatura del commit de esta forma git ls-tree 5be23bb, siendo este último una parte del SHA1 (o hash) del último commit; nos devolvería el último resultado.

Pero podemos ir todavía más profundamente dentro de las tuberías. ls-tree sólo lista los objetos que ya forman parte del árbol, del principal o de alguno de los secundarios. Puede que necesitemos acceder a aquellos objetos que se han añadido al índice, pero todavía no han pasado a ningún árbol. Para eso usamos ls-files. Tras añadir un fichero que está en un subdirectorio views con add, podemos hacer:

git ls-files --stage 100644 a6f69e4284566cd84272c6a4e4996f64643afbea 0 .aspell.es.pws 100644 a72b52ebe897796e4a289cf95ff6270e04637aad 0 .gitignore 100644 cc5411b5557f43c7ba2f37ad31f8dc34cccda075 0 .gitmodules 100644 4e7b6c1b5a6cb3a962ea05874d10c943c1923f39 0 .travis.yml 100644 d5445e7ac8422305d107420de4ab8e1ee6227cca 0 LICENSE 100644 d1913ebe4d9e457be617ee0e786fc8c30a237902 0 Procfile 100644 da5b5121adb42e990b9e990c3edb962ef99cb76a 0 README.md 160000 fa8b7521968bddf235285347775b21dd121b5c11 0 curso 100644 f8c35adaf57066d4329737c8f6ec7ce6179cc221 0 package.json 100644 08827778af94ea4c0ddbc28194ded3081e7b0f87 0 shippable.yml 100644 9920d80438d42e3b0a6924a0fcace2d53a6af602 0 test/route.js 100644 36cc059186e7cb247eaf7bfd6a318be6cffb9ea3 0 views/layout.jade 100644 94f151d9ef9340c81989b0c3fa8c517c068e1864 0 web.js

Que nos devuelve, en penúltimo lugar, un fichero que todavía no ha pasado al árbol. Evidentemente, tras el commit:

~/txt/docencia/repo-tutoriales/repo-ejemplo<master>$ git ls-tree HEAD [...] 040000 tree fd3846c0d6089437598004131184c61aea2b6514 views

Este listado nos muestra el nuevo objeto de tipo tree que se ha creado y nos da su SHA1, que podemos usar para examinarlo con ls-tree:

~/txt/docencia/repo-tutoriales/repo-ejemplo<master>$ git ls-tree fd3846c 100644 blob 36cc059186e7cb247eaf7bfd6a318be6cffb9ea3 layout.jade

que, si queremos ver en una vista más normal, hacemos lo mismo con ls-file:

~/txt/docencia/repo-tutoriales/repo-ejemplo<master>$ git ls-files views
views/layout.jade

Hay un tercer comando relacionado con el examen de directorios y ficheros locales, cat-file, que muestra el contenido de un objeto, en general. Por ejemplo, en este caso, para listar el contenido de un objeto de tipo tree:

~/txt/docencia/repo-tutoriales/repo-ejemplo<master>$ git cat-file -p fd3846c 100644 blob 36cc059186e7cb247eaf7bfd6a318be6cffb9ea3 layout.jade

nos muestra que ese objeto contiene un solo fichero, layout.jade, y sus características. Pero más curioso aún es cuando se usa sobre objetos de tipo *commit* (no sobre el objeto *commit* que aparece arriba, que se trata de un *commit* de *otro repositorio* al contener el directorio curso un submódulo de git. Por ejemplo, podemos hacer:

~/txt/docencia/repo-tutoriales/repo-ejemplo<master>$ git cat-file -p HEAD tree 1c40899a32c2b5ec7f930bd943e5dbb98562d373 parent 5be23bb2a610260da013fcea807be872a4bd6981 author JJ Merelo <jjmerelo@gmail.com> 1397752151 +0200 committer JJ Merelo <jjmerelo@gmail.com> 1397752151 +0200 Añade layout

que, dado que HEAD apunta al último commit, nos muestra en modo *pretty-print* toda la información sobre el último *commit* y muestra el árbol de ficheros correspondiente, que podemos listar con:

~/txt/docencia/repo-tutoriales/repo-ejemplo<master>$ git cat-file -p 1c40899a 100644 blob a6f69e4284566cd84272c6a4e4996f64643afbea .aspell.es.pws 100644 blob a72b52ebe897796e4a289cf95ff6270e04637aad .gitignore 100644 blob cc5411b5557f43c7ba2f37ad31f8dc34cccda075 .gitmodules 100644 blob 4e7b6c1b5a6cb3a962ea05874d10c943c1923f39 .travis.yml 100644 blob d5445e7ac8422305d107420de4ab8e1ee6227cca LICENSE 100644 blob d1913ebe4d9e457be617ee0e786fc8c30a237902 Procfile 100644 blob da5b5121adb42e990b9e990c3edb962ef99cb76a README.md 160000 commit fa8b7521968bddf235285347775b21dd121b5c11 curso 100644 blob f8c35adaf57066d4329737c8f6ec7ce6179cc221 package.json 100644 blob 08827778af94ea4c0ddbc28194ded3081e7b0f87 shippable.yml 040000 tree 39da6b155c821af1e6a304daca9b66efb1ac651f test 040000 tree fd3846c0d6089437598004131184c61aea2b6514 views 100644 blob 97c32024cda29e0fb6abebf48d3f6740f0acb9e2 web.js

En general, si queremos ahondar en las entrañas de un punto determinado en la historia del repositorio, trabajar con ls-files, cat-file y ls-tree permite obtener toda la información contenida en el mismo. Esto nos va a resultar útil un poco más adelante.

Viva la diferencia

En muchos casos para procesar los cambios dentro de un gancho necesitaremos saber cuál es la diferencia con versiones anteriores del fichero. Hay que tener en cuenta que esas diferencias, dependiendo del estado en el que estemos, estarán en el árbol o en el índice preparadas para ser enviadas al repositorio, En general, son una serie de órdenes con diffen ellas. La más simple, git diff, nos mostrará la diferencia entre los archivos en el índice y el último commit:

~/txt/docencia/repo-tutoriales/repo-ejemplogit diff diff --git a/views/layout.jade b/views/layout.jade index 36cc059..2a66d58 100644 --- a/views/layout.jade +++ b/views/layout.jade @@ -1,6 +1,11 @@ -!!! 5 +doctype html +html(lang="en")

-body +html + head + title #{title} + body

-#wrapper - block content \ No newline at end of file + h1 Curso de git + + block content \ No newline at end of file diff –git a/web.js b/web.js index 97c3202..93b6255 100644 — a/web.js +++ b/web.js @@ -27,6 +27,12 @@ app.get(‘/’, function(req, res) { res.send(routes[‘README’]); });

+app.get(‘/curso/:ruta’, function(req, res) { + var ruta = “curso/texto/”+req.params.ruta; + console.log(“Request “+req.params.ruta + “ doc “ + ruta + “ contenido “ + file_conte + res.render(‘doc’, { content: routes[ruta], title: ruta }); +}) + app.get(‘/curso/texto/:ruta’, function(req, res) { // console.log(“Request “+req.params.ruta); var ruta_toda = “curso/texto/”+req.params.ruta;

Esta vista de diff las diferencias sigue el formato habitual en la utilidad diff, que permite generar parches para aplicarlos a conjuntos de ficheros. En concreto, muestra qué ficheros se están comparando (pueden ser diferentes ficheros, si se ha cambiado el nombre) los SHA1 de los contenidos correspondientes, y luego un + o - delante de cada una de las líneas que hay de diferencia. Este fichero se podría usar directamente con la utilidad diff de Linux, pero realmente no nos va a ser de mucha utilidad a la hora de saber, por ejemplo, qué ficheros se han modificado. Para hacer esto, simplemente:

~/txt/docencia/repo-tutoriales/repo-ejemplo<master>$ git diff --name-only views/layout.jade web.js

que se puede hacer un poco más completa con --name-status:

~/txt/docencia/repo-tutoriales/repo-ejemplo<master>$ git diff --name-status M views/layout.jade M web.js ~/txt/docencia/repo-tutoriales/repo-ejemplo<master>$ git diff --name-status --cached A views/doc.jade

que nos muestra, en una sola letra, qué tipo de cambio han sufrido. En el primer caso nos muestra que han sido Modificados, y en el segundo caso, además usamos otra

opción, --cached que, en este caso, nos muestra los ficheros que han sido preparados para el commit; es decir, la diferencia que hay entre la cabeza y el índice; en este caso, "A" indica que se trata de un fichero añadido. Podemos ver todo junto con:

~/txt/docencia/repo-tutoriales/repo-ejemplo<master>$ git diff --name-status HEAD A views/doc.jade M views/layout.jade M web.js

que nos muestra la diferencia entre el commit más moderno (HEAD) y el área de trabajo ahora mismo: hemos cambiado dos ficheros y añadido uno. Un resultado similar obtendremos con diff-index, cuya principal diferencia es que compara siempre el índice con algún *árbol*, sin tener ningún valor por omisión:

~/txt/docencia/repo-tutoriales/repo-ejemplo<master>$ git diff-index HEAD :000000 100644 0000000000000000000000000000000000000000 67e6d7e1ecbb64ff7d467dc2103fa2b2fead49d1 A views/doc.jade :100644 100644 36cc059186e7cb247eaf7bfd6a318be6cffb9ea3 0000000000000000000000000000000000000000 M views/layout.jade :100644 100644 97c32024cda29e0fb6abebf48d3f6740f0acb9e2 0000000000000000000000000000000000000000 M web.js

Además del estado muestra el hash inicial y final de cada uno de los ficheros. En este caso, como todavía no le hemos hecho commit, muestra 0. dif-tree, sin embargo, sí muestra el SHA1 puesto que trabaja con el árbol:

~/txt/docencia/repo-tutoriales/repo-ejemplo<master>$ git diff-tree HEAD fe88e5eefff7f3b7ea95be510c6dcb87054bbcb0 :000000 040000 0000000000000000000000000000000000000000 fd3846c0d6089437598004131184c61aea2b6514 A views :100644 100644 94f151d9ef9340c81989b0c3fa8c517c068e1864 97c32024cda29e0fb6abebf48d3f6740f0acb9e2 M web.js

Aunque en este caso muestra un árbol, views, que ha sido cambiado porque se le ha añadido un fichero nuevo, views/doc.jade. En el momento que se haga el commit y pase por tanto del índice a la zona de *staging*, los hash ya están calculados y cambia la salida:

~/txt/docencia/repo-tutoriales/repo-ejemplo<master>$ git diff-tree HEAD 637c2820013188f1c4951aef0c21de20440a6fbb :040000 040000 fd3846c0d6089437598004131184c61aea2b6514 6bb4560a218c008bbc468f23f36f26ff639eb2e8 M views :100644 100644 97c32024cda29e0fb6abebf48d3f6740f0acb9e2 93b625533c2d1752d9a8e789878512919cf92e17 M web.js

diff-index, sin embargo, no devolverá nada puesto que todos los cambios que se habían hecho han pasado al árbol.

En general, lo que más nos va a interesar a la hora de hacer un gacho es qué ficheros han cambiado. Pero conviene conocer toda la gama de posibilidades que ofrece git, sobre todo para poder entender su estructura interna.

Los dueños de las tuberías

No todo el contenido que hay en el repositorio son los ficheros que forman parte del mismo. Hay una parte importante de la fontanería que son los metadatos del repositorio. Hay dos órdenes importantes, var y config. Con -l nos listan todas las variables o variables de configuración disponibles:

~/txt/docencia/repo-tutoriales/curso-git/texto<master>$ git var -l user.email=jjmerelo@gmail.com user.name=JJ Merelo filter.obj-add.smudge=cat push.default=simple rerere.enabled=true core.repositoryformatversion=0 core.filemode=true core.bare=false core.logallrefupdates=true remote.origin.url=git@github.com:oslugr/curso-git.git remote.origin.fetch=+refs/heads/*:refs/remotes/origin/* branch.master.remote=origin branch.master.merge=refs/heads/master GIT_COMMITTER_IDENT=JJ Merelo <jjmerelo@gmail.com> 1399019391 +0200 GIT_AUTHOR_IDENT=JJ Merelo <jjmerelo@gmail.com> 1399019391 +0200 GIT_EDITOR=editor GIT_PAGER=pager

Todas excepto las cuatro últimas variables son variables de configuración que, por tanto, se pueden obtener también con git config -l. Por sí sólo, config o var listan el valor de una variable:

~/txt/docencia/repo-tutoriales/curso-git/texto<master>$ git config user.name JJ Merelo

La mayoría de estos valores están disponibles o como variables de entorno o en ficheros; sin embargo, estas órdenes dan un interfaz común para todos los sistemas operativos.

Todavía nos hacen falta una serie de órdenes para tomar decisiones sobre ficheros y sobre dónde estamos en el repositorio. La veremos a continuación:

Simplemente, rev-parse

La tersa descripción del comando rev-parse, “recoge y procesa parámetros” esconde la complejidad del mismo y su potencia, que va desde el procesamiento de parámetros hasta la especificación de objetos, pasando por la búsqueda de diferentes directorios dentro del repositorio git. Por ejemplo, se puede usar para verificar si un objeto existe o no:

~/txt/docencia/repo-tutoriales/repo-ejemplo<master>$ git rev-parse --verify HEAD 637c2820013188f1c4951aef0c21de20440a6fbb

Nos muestra el SHA1 de la cabeza actual del repositorio de ejemplo, verificando que actualmente existe. No lo hará si acabamos de crear el repositorio, por ejemplo:

/tmp/pepe<>$ git init Initialized empty Git repository in /tmp/pepe/.git/ /tmp/pepe<>$ git rev-parse --verify HEAD fatal: Needed a single revision

De hecho, con él podemos encontrar todo tipo de objetos usando la notación que permite especificar revisiones:

~/txt/docencia/repo-tutoriales/repo-ejemplo<master>$ git rev-parse HEAD@{1.month} 61253ecba351921c96a1553f6c5b7f9910f286f3

que correspondería a este commit del 9 de marzo y que podemos listar usando show, describe o cualquiera de los otros comandos que se pueden aplicar a objetos. En general, para esto se usará dentro de los *garfios*: para poder acceder a un objeto determinado o a sus metadatos a la hora de ver las diferencias con el objeto actual.

Programando ganchos

Concepto de *hooks*

Un *hook*. literalmente *garfio* o *gancho* es un programa que se ejecuta cuando sucede un evento determinado en el repositorio. Los *webhooks* de GitHub, por ejemplo, son un ejemplo: cuando se lleva a cabo un *push*, se envía información al sitio configurado para que ejecute un programa determinado: pase unos test, publique un tweet, o lleve a cabo una serie de comprobaciones.

Los *ganchos* no son estrictamente necesarios en todo tipo de instalaciones; se puede trabajar con un repositorio sin tener la necesidad de usarlos. Sin embargo, son tremendamente útiles para automatizar una serie de tareas (como los test que se usan en integración continua), implementar una serie de políticas para todos los usuarios de un repositorio (formato de los mensajes de *commit*, por ejemplo) y añadir información al repositorio de forma automática.

Los *hooks* son, por tanto, programas ejecutables. Cualquier programa que se pueda lanzar puede servir, pero generalmente se usa o guiones del *shell* (si uno es suficientemente masoquista) o lenguajes de *scripting* tales como Perl, Python, Ruby, Javascript o, si uno es *realmente* masoquista, PHP. En realidad, a git le da igual qué lenguaje se use.

Los *ĥooks* van en su propio directorio, .git/hooks que se crea automáticamente y que tiene, siempre, una serie de *scripts* ejemplo, ninguno de ellos activados. Sólo se admite un *hook* por evento, y ese *hook* tendrá el nombre del evento asociado; es decir, un programa llamado post-merge en ese directorio se ejecutará siempre cada vez que se

termine un *merge* con éxito. Como generalmente uno quiere que los scripts tengan un nombre razonable, la estrategia más general es usar un *enlace simbólico* de esta forma:

ln -s nombre-real-del-script.sh post-merge y, en todo caso, no se debe olvidar:

chmod +x nombre-real-del-script.sh

para hacerlo ejecutable.

> Windows seguramente tendrá su forma particular de hacer lo mismo, o ninguna. Por cualquiera de esas dos razones, nunca recomendamos Windows como una plataforma para desarrollo. Úsala para la declaración de la renta o para jugar al Unreal, pero para desarrollar usa una plataforma para desarrolladores: Linux (o Mac, que tiene un núcleo Unix por debajo).

Los *hooks* se activarán cuando se ejecute un comando determinado y recibirán una serie de parámetros como argumento o en algún caso como entrada estándar. Este cuadro resume cuando se ejecutan y también qué reciben como parámetro. En general, también tendrán influencia en si tiene éxito o no el comando determinado: salir con un valor no nulo, en algunos casos, parará la ejecución del comando con un mensaje de error. Por ejemplo, un *hook applypatch-msg*, que se aplica desde el comando git am antes de que se ejecute, parará la aplicación del parche si se sale con un valor 1.

De todos los *hooks* posibles sólo veremos los que se refieren al *commit*. Son los que se pueden usar en local (los referidos a *push* sólo se programan en remoto, y los que se aplican a git am o git gc quedan fueran de los temas de este libro. Hay sólo cuatro de estos, que veremos a continuación.

Programando un *hook* básico

En general, un *hook* hará lo siguiente:

1. Examinar el entorno y los parámetros de entrada
2. Hacer cambios en el entorno, los ficheros o la salida
3. Salir con un mensaje si hay algún error, o ninguno.

No son diferentes de ningún otro programa, en realidad, salvo que los parámetros de entrada y cómo se debe salir está establecido de antemano.

Miremos un script simple, que actúa como gancho para preparación de un mensaje de commit (prepare-commit-msg, incluido en el repositorio de ejemplo de este curso:

#!/bin/sh SOB=$(git config github.user) grep -qs "^$SOB" "$1" || echo ". Cambio por @$SOB" >> "$1"

Este script tiene toda la simplicidad de estar en dos líneas y toda la complicación de estar escrito para el *shell*. Esta introducción venerable te puede ayudar a empezar a trabajar con él, pero en este capítulo no pretendemos que aprendas a programar, sólo que tengas las nociones básicas para echar a andar y posiblemente modificar ligeramente un gancho.

Empecemos por la primera línea: es común a todos los guiones del shell. Simplemente indica el camino en el que se encuentra el mismo, con #! indicando que se trata de un fichero ejecutable (junto con el chmod +x, que se lo indica al sistema de ficheros.

La siguiente línea define una variable, SOB, que no es acrónimo de nada, cuidadito. Esa variable usa el formato de ejecución de un comando del shell $() para asignar la salida de dicho comando a la variable.

La tercera línea, en resumen, comprueba si el nombre del usuario ya está en el mensaje y si no lo está le añade una "firma" que lo incluye. Lo hace de la forma siguiente: grep -qs "^$SOB" "$1" comprueba si el valor de la variable no (de ahí el caret ^) está ya en el mensaje (cuyo nombre de fichero está en el primer argumento, $1, según vemos en la chuleta de ganchos de git. Si no lo está (de ahí el ||, es decir, *O*, se escribe (echo) el valor de la variable añadiéndose (>>) al final del fichero cuyo nombre está en la variable $1.

La variable github.user no tiene por qué estar definida siempre. Se puede sustituir por user.email, por ejemplo, o por user.name, que sí se suele definir siempre cuando se crea un repositorio.

Este otro ejemplo es todavía más minimalista, y también añade información al commit (que, como en el caso anterior, se puede eliminar si se edita el fichero de commit):

STATS=$(git diff --cached --shortstat) echo ". Cambios en este commit\n ${STATS}" >> "$1"

Es interesante notar, en este caso, que se usa diff --cached ya que en este caso los cambios estarán ya *staged* o *cached* y la diferencias (que suelen aparecer de todas formas en el mensaje que da el comando) serán entre HEAD y lo que hay ya almacenado el área de preparación de los ficheros, no entre HEAD y lo que hay en el sistema de ficheros; esto es así porque ya estamos *dentro* del commit y los ficheros están ya preparados para ser procesados en las tuberías de git por sus comandos-tubería. En resumen, esta orden da una mini-estadística que dice el número de ficheros y líneas cambiadas, produciendo resultados sobre este mismo fichero tales como este:

Mini-corrección . Cambios en este commit 1 file changed, 1 insertion(+)

Otro *hook* puede servir para comprobar que los mensajes de *commit* son correctos. Como se ha visto anteriormente, una buena práctica es usar una primera línea de 50 caracteres (que aparecerán como título) seguida por una línea vacía y el resto del

mensaje. Esto se puede aplicar mediante un programa en Python o el siguiente en Node, una versión de Javascript.

#!/usr/bin/env node var fs = require('fs'); var msg_file = process.argv[2]; fs.readFile(msg_file, 'utf8', function(err, data) { if (err) throw err; var lines = data.split("\n"); if (lines[0].length > 50) { console.log("[FORMATO] Primera línea > 50 caracteres"); process.exit(1); } })

La primera línea es común a los scripts, y a continuación se carga el módulo necesario para usar el sistema de ficheros (fs) y se lee el argumento que se pasa por línea de órdenes, el nombre del fichero que contiene el mensaje del commit (este mensaje estará ahí, aunque lo hayamos pasado con -m desde la línea de órdenes).

El resto, usando el modo asíncrono que es común en Node, lee el fichero (creando una excepción si hay algún error), lo divide en líneas usando split y a continuación comprueba si la primera línea (lines[0]) tiene más de 50 caracteres, en cuyo caso sale del proceso con un código de error (1), de esta forma:

~/txt/docencia/repo-tutoriales/curso-git<master>$ git commit -am "Añadiendo un montón de cosas al capítulo de los ganchos y testeándolo a la vez" [FORMATO] Primera línea > 50 caracteres

Si no es así, simplemente deja pasar el mensaje. En este *hook*, curiosamente, no se usa más comando de git que el mensaje almacenado en el fichero; realmente, no es imprescindible usarlo, sólo cuando vayamos a usar información de git en el mismo.

El programa anterior es un ejemplo de cómo se pueden implementar políticas de formato o de cualquier otro tipo sobre un repositorio; sin embargo, la capacidad que tienen es limitada, ya que se aplican sólo sobre los mensajes. En realidad, el que actúen de esa forma es convencional, porque los programas que ejecutan los *ganchos* se diferencian solamente en el momento en el que actúan, no en lo que pueden hacer. Sin embargo, si queremos implementar una política sobre los nombres de ficheros o el contenido de los mismos, hay que usar un *gancho* que actúe cuando se añadan al repositorio como el siguiente gancho pre-commit escrito en Perl:

#!/usr/bin/env perl my $is_head = `git rev-parse --verify HEAD`; my $last_commit = $is_head?"HEAD":"4b825dc642cb6eb9a060e54bf8d69288fbee4904"; my @changes = `git diff-index --name-only $last_commit`; my %policies = (no_underscore => qr/_/); for my $f (@changes) { for my $p (keys %policies) { if ($f =~ /$policies{$p}/) { die "[FORMATO]: $p "; } } }

Este programa es similar a los anteriores, pero para empezar usa un comando *cañería* que se encuentra mucho: rev-parse. Las dos primeras órdenes comprueban si nos encontramos en el primer *commit* de un repositorio (en cuyo caso HEAD no existe y

tendremos que usar el *commit primigenio* que es igual en todos los repositorios de git; lo que pretenden esas dos líneas es encontrar el "último commit" para ver cuál es la diferencia con respecto a él. Como vamos a tratar con ficheros que acaban de ser añadidos al índice, usamos diff-index en la siguiente línea con un formato que nos devolverá solamente los ficheros cambiados desde el último commit (o desde el primero) sin que nos interese nada más sobre ellos.

La siguiente línea define un *hash* con un nombre y una expresión regular que será la que se tiene que implementar. Si no conoces el lenguaje no te preocupes, pero si lo conoces (ese u otro) es relativamente fácil añadir políticas nuevas, como por ejemplo que no se permitan .pdfs, simplemente añadiéndole una línea.

El bucle for posterior es el que va recorriendo cada uno de los cambios y cada una de las políticas (aunque en este caso habrá una sola) y saldrá con die si alguno de los ficheros tiene un nombre incorrecto.

~/txt/docencia/repo-tutoriales/repo-plantilla<master>$ git commit -am "A ver si me deja" [FORMATO]: no_underscore at .git/hooks/pre-commit line 13. ~/txt/docencia/repo-tutoriales/repo-plantilla<master>$ git status # En la rama master # Cambios para hacer commit: # (use «git reset HEAD <archivo>...«para eliminar stage) # # archivo nuevo: uno_que_no

Lo que se puede hacer ahora no es tan trivial: puedes cambiar el fichero de nombre a pelo, pero ya está en el índice, por eso es mejor hacer algo como git mv (o git rm --force).

Algunos *hooks* útiles explicados

Hay múltiples posts en blogs que explican diferentes ejemplos de *hooks* y lo que se puede hacer con ellos. Por ejemplo, en este conjunto usa programas externos y su código de salida ($?) para comprobar si un programa es correcto o no; también usa extensivamente git stash para almacenar todos los ficheros que se hayan modificado y luego los recupera con git stash pop. Este, por ejemplo, crea una plantilla para usarla en los mensajes de commit, Pero este gist incluye algunos *hooks* útiles que vamos a ver. Por ejemplo, el siguiente comprueba si se encuentra la palabra debugger en el fichero (es decir, comentarios de depuración):

if git-rev-parse –verify HEAD >/dev/null 2>&1; then against=HEAD else against=4b825dc642cb6eb9a060e54bf8d69288fbee4904 fi for FILE in `git diff-index --check --name-status $against -- | cut -c3-` ; do # Check if the file contains 'debugger' if ["grep 'debugger' $FILE"] then echo $FILE ' contains debugger!' exit 1 fi done

Lo más complicado que tiene este fichero es el uso de filtros del shell (algo que, por otro lado, debería aprenderse); esos filtros hacen algo similar a lo que se ha hecho antes con

Perl: recorre el nombre de los ficheros y le aplica el buscador de cadenas, grep, buscando la palabra debugger; si la encuentra, sale. Quizás está mejor explicado en la historia original.

Recursos

Unos cuantos tutoriales, en castellano.

Tutorial de ganchos de git

Una web dedicada a los hooks

Algunos ganchos en PHP