

Manual de GIT (Básico)

Introducción

Git es un sistema de control de versiones. Estos sistemas nos permiten llevar un control exhaustivo sobre todos los cambios que se realicen en un conjunto de ficheros, normalmente código. Así, podrá verse qué modificaciones se han hecho en cada momento y quién ha realizado cada cambio, y muchísimas otras cosas que no tenemos tiempo para ver.

Aunque existen clientes gráficos para Git (el ejemplo más bonito es el cliente de GitHub), lo único que llegan a hacer es poner iconos y menús que, por detrás, ejecutarán los mismos comandos que se realizan desde una terminal. A mayores, todos esos menús e iconos utilizan la misma terminología que los comandos, por lo que si no se conocen los comandos puede ser un poco improductivo, por lo cual yo recomiendo que los ejecutéis directamente desde consola.

En la máquina virtual de Rodeiro ya viene instalado Git para que podáis utilizarlo directamente desde ella, y que así no tengáis que instalar nada nuevo en vuestros equipos locales, aunque si queréis nadie os lo impide.

Aspectos básicos

Todos los comandos de Git son invocados como parámetros dentro del comando general `git`. Por ejemplo, dado un comando imaginario “`hazmitrabajo`”, la forma de invocación sería: `git hazmitrabajo`. A su vez, esos comandos recibirán nuevos parámetros que se pasarán como parámetros a continuación del nombre, por ejemplo: `git hazmitrabajo -ahora`, implica la invocación del comando “`hazmitrabajo`” de `git` con el parámetro “`-ahora`”.

Configuración

Antes de realizar nada en Git, deberíais realizar dos pasos de configuración básicos: decirle cuál es vuestro nombre y vuestro e-mail. Git cuenta con un comando llamado “`config`” que permite realizar configuraciones directamente sin tener que editar ningún fichero de configuración. Así, los comandos que deberíais ejecutar para decirle a Git vuestro nombre y email son:

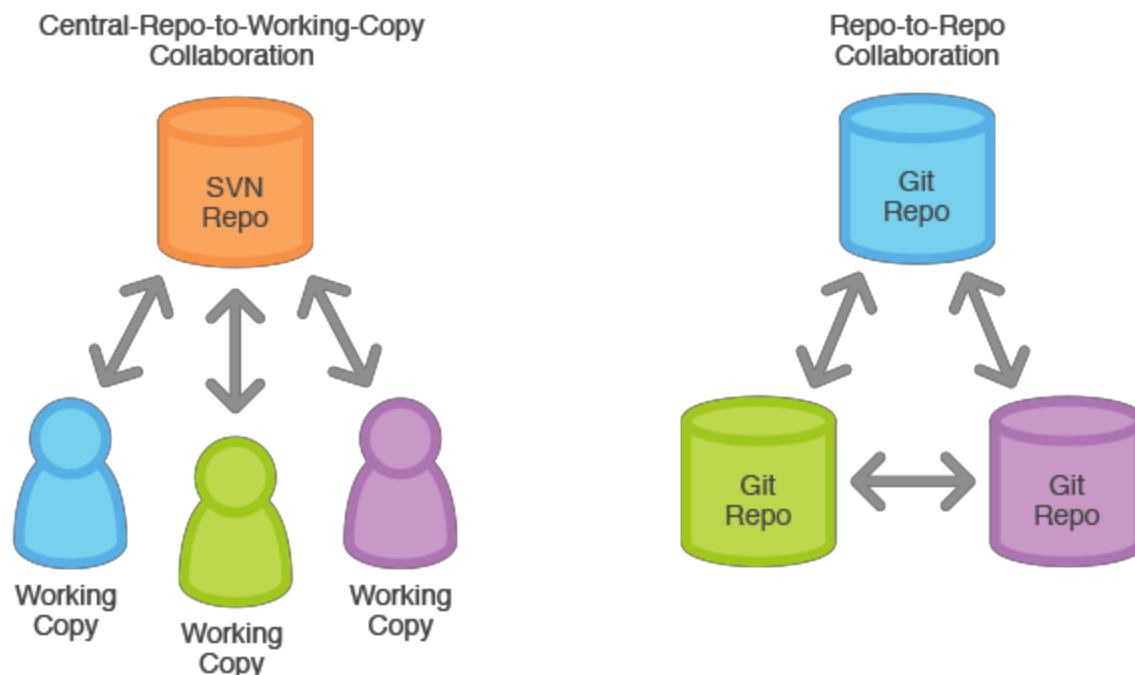
```
git config --global user.name <miNombre>
git config --global user.email <miEmail>
```

Sustituyendo el `<miNombre>` (si hay espacios, tendréis que ponerlo entre comillas) y `<miEmail>` por lo correspondiente, por ejemplo:

```
git config --global user.name "Alberto Gutiérrez Jácome"
git config --global user.email agjacome@gmail.com
```

Clonación

Git, al igual que cualquier otro sistema de control de versiones, utiliza “repositorios” para el almacenamiento y control de ficheros. Cada repositorio puede verse como un simple directorio donde se almacena todo lo que debe ser controlado y todos los ficheros propios que Git utiliza para ello. Una característica de Git que lo diferencia a otros sistemas (ejemplo: SVN, que se utilizaba en IU hasta este año, somos los primeros que utilizamos Git), es que es descentralizado. Esta descentralización implica que cada copia del repositorio es una copia completa, que no depende de ningún otro repositorio para funcionar correctamente, mientras que en uno centralizado existirá una copia central completa y el resto de copias serán versiones “reducidas” en las que se necesitará de la existencia del repositorio central para su correcto funcionamiento. La versión descentralizada tiene mayor tolerancia a fallos, puesto que si el repositorio central deja de existir (el servidor que lo contiene peta y no hay copia de respaldo, o se cae temporalmente), el sistema no permitirá realizar ningún cambio ni, lo que es más importante, recuperar toda la información desde ninguno de los otros repositorios, cosa que sí es posible en Git.



SourceForge, donde ya os habéis registrado, lo que proporciona es un repositorio central. Como ya he dicho, la diferenciación entre un repositorio central y uno que no es central en Git es nula (realmente no, pero al nivel que lo vamos a utilizar, es prácticamente lo mismo), la ventaja principal de tener un repositorio central es el control unificado de todos los cambios, evitando que un cambio concreto se quede en un repositorio local y que el resto de desarrolladores del proyecto no tengan acceso al mismo.

Git proporciona un comando para copiar un repositorio remoto (en nuestro caso, el de SourceForge) en un repositorio local, para poder realizar cambios sobre los ficheros del mismo, que posteriormente serán integrados en el repositorio central. Su sintaxis es:

```
git clone <direccion-remota> <directorio-local>
```

Si se omite el directorio local, se creará dentro de un directorio con el mismo nombre que el repositorio remoto (eg: http://midominio.com/git/asd/mi_repo.git creará un directorio "mi_repo"). Ejemplo de uso:

```
git clone https://github.com/moot/riotjs.git riot_js
```

```
[agjacome]-[Caronte]-[~/projects/temporal]
[✓]→ git clone https://github.com/moot/riotjs.git riot_js
Cloning into 'riot_js'...
remote: Counting objects: 117, done.
remote: Compressing objects: 100% (78/78), done.
remote: Total 117 (delta 40), reused 107 (delta 31)
Receiving objects: 100% (117/117), 304.52 KiB | 143.00 KiB/s, done.
Resolving deltas: 100% (40/40), done.
Checking connectivity... done
[agjacome]-[Caronte]-[~/projects/temporal]
[✓]→ cd riot_js/
[agjacome]-[Caronte]-[.../projects/temporal/riot_js]-[master]
[✓]→ ls
compare lib test todomvc Makefile README.md riot.js riot.min.js
[agjacome]-[Caronte]-[.../projects/temporal/riot_js]-[master]
[✓]→
```

Tras la ejecución de ese comando, dentro del directorio “riot_js” se encontrará el repositorio de Riot.js (un framework MVP, parecido a MVC, para JavaScript, que ocupa sólo 1KiB).

Realizando cambios

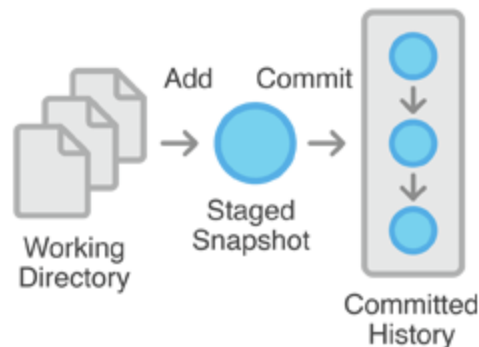
Git cuenta con dos comandos, add y commit, que suponen la base central para todo el trabajo que se realice. Entender su funcionamiento requiere entender el modelo de trabajo de Git.

Git utiliza tres áreas dentro de cada repositorio, el directorio de trabajo (working directory), el directorio de git (git directory) y el área de preparación (staging area). Cada una de esas tres áreas se corresponde con cada uno de los tres estados posibles para cada fichero dentro del repositorio: confirmado (committed), modificado (modified) y preparado (staged). Committed implica que los datos están almacenados de forma segura dentro de la base de datos de Git en el repositorio local. Modified implica que se han realizado cambios a un fichero del repositorio, pero éstos aún no han sido confirmados en la base de datos. Y Staged significa que el fichero se ha marcado como modificado en su versión actual, para que se incluya en la próxima confirmación a la base de datos.

El flujo de trabajo de Git, en base a estos estados es algo parecido a lo siguiente:

1. Se realiza alguna modificación sobre uno o varios ficheros del repositorio, cambiando así su estado a “modified”.
2. Se marcan los ficheros que se quiera almacenar como modificados, incluyéndose dentro de la “staging area”.

3. Se confirman los cambios, tomando los ficheros tal como estén en la “staging area” y almacenando dichas modificaciones de forma permanente en el “git directory”.



Para comprobar, en cualquier momento, el estado de todos los ficheros del repositorio, Git proporciona el comando status:

```
git status
```

Git comprobará constantemente el estado de todos los ficheros, y si ve algún cambio en alguno lo marcará automáticamente como “modified”, sin que se tenga que realizar ninguna acción. Para añadir esos cambios al área de preparación, Git proporciona el comando “add”, cuya sintaxis es la siguiente:

```
git add ruta_a_fichero1 ruta_a_fichero2 ...
```

Tras añadirlos a la staging area (equivalentemente, marcarlos como “staged”), si se ejecuta de nuevo el comando “status”, se verá que el estado de los ficheros ha cambiado. Una vez marcados como preparados para confirmación, se le puede indicar a Git que almacene dichos cambios en la base de datos local mediante el comando “commit”:

```
git commit -m <mensaje>
```

Git obliga a la inclusión de un mensaje siempre que se confirme algún cambio, para escribir dicho mensaje de forma cómoda se puede utilizar el parámetro “-m” seguido de un pequeño mensaje (que si contiene espacios debe ir entre comillas). Si dicho flag “-m” no es proporcionado, Git abrirá el editor de textos por defecto del sistema, que es probable que sea vim o cualquier otro editor de texto en consola, para que se proporcione el mismo por lo cual seguramente resulte más cómodo escribir el mensaje directamente con el parámetro “-m”. Tras la confirmación de los cambios, si se vuelve a invocar el comando “status” se verá que los ficheros preparados ya no aparecen en el mismo, puesto que status sólo nos informa de ficheros modificados y preparados.

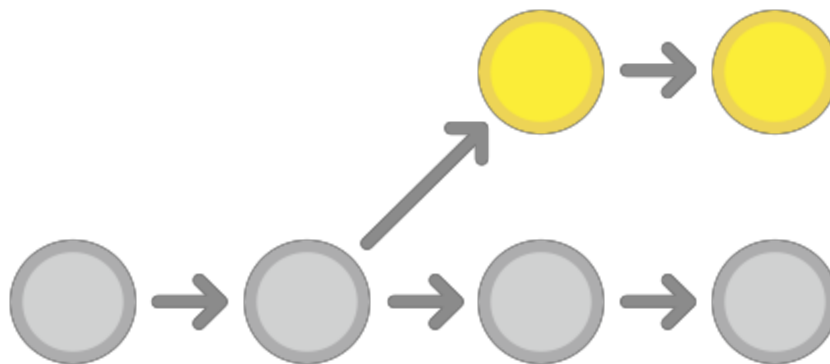
A continuación se muestra una captura de pantalla realizando y confirmando un cambio sobre el repositorio de Riot.js descargado en el punto anterior:

```
[agjacome]-[Caronte]-[~/projects/temporal/riot_js]-[master]
[/]→ echo "2013 © IU Grupo-2 Inc." > todomvc/copy.html
[agjacome]-[Caronte]-[~/projects/temporal/riot_js]-[master!]
[/]→ git status
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       todomvc/copy.html
nothing added to commit but untracked files present (use "git add" to track)
[agjacome]-[Caronte]-[~/projects/temporal/riot_js]-[master!]
[/]→ git add todomvc/copy.html
[agjacome]-[Caronte]-[~/projects/temporal/riot_js]-[master!]
[/]→ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   todomvc/copy.html
#
[agjacome]-[Caronte]-[~/projects/temporal/riot_js]-[master!]
[/]→ git commit -m 'added copyright file'
[master 1440ada] added copyright file
1 file changed, 1 insertion(+)
create mode 100644 todomvc/copy.html
[agjacome]-[Caronte]-[~/projects/temporal/riot_js]-[master]
[/]→ git status
# On branch master
# Your branch is ahead of 'origin/master' by 1 commit.
#   (use "git push" to publish your local commits)
#
nothing to commit, working directory clean
[agjacome]-[Caronte]-[~/projects/temporal/riot_js]-[master]
[/]→
```

Ramificaciones

Una de las características más diferenciadoras de Git con respecto a otros sistemas es el uso que proporciona a las ramificaciones (branches). Mientras que en otros sistemas (como SVN) éstas branches implican, a grandes rasgos, la clonación completa de un repositorio en otro con distinto nombre, en Git las ramificaciones forman parte del mismo repositorio, y no implican la copia del proyecto entero en otro “apartado”, sino que son un simple puntero a un punto concreto del repositorio.

Una rama o branch representa una línea independiente de desarrollo. Las branches sirven como una abstracción sobre el proceso editar/preparar/confirmar presentado en el punto anterior. Pueden verse, a grandes rasgos, como la creación de un nuevo directorio de trabajo, directorio de git y área de preparación independientes del resto del repositorio. Los nuevos commits realizados en una branch se almacenan en la base de datos de la propia branch, sin afectar a todas las demás existentes en el repositorio.



Git, en todo repositorio, por defecto, crea una rama principal llamada “master”, donde se realizarán todos los cambios a menos que se especifique lo contrario. El nombre de ésta rama master puede modificarse en cualquier momento y no afectará al funcionamiento del repositorio, pero es una convención establecida entre todos los usuarios de Git que la rama “master” existe y que además es la principal del proyecto, donde se almacena la versión estable del código.

Gestionando ramas

Para listar todas las ramas existentes en el repositorio:

```
git branch
```

La rama actual en uso estará marcada con un asterisco dentro del listado de branches.

Para crear una nueva rama:

```
git branch <nombre>
```

Toda rama debe tener un nombre, por tanto debe proporcionarse el nombre de la rama a crear, sustituyendo <nombre> por el mismo, por ejemplo:

```
git branch mi_rama
```

Las ramas son creadas a partir del directorio de git, esto es: almacenan toda la historia de la rama desde donde son creadas hasta el último commit realizado antes de su creación. Todo cambio que se realice en la nueva rama, partirá desde los cambios confirmados (previos a la creación) de la rama original.

Para eliminar una rama:

```
git branch -d <nombre_rama>  
git branch -D <nombre_rama>
```

La diferencia entre ambos comandos (d minúscula o mayúscula) es el modo de borrado de la rama. La versión con “-d” minúscula implica un borrado seguro, desde el cual Git se encargará de comprobar que todas las modificaciones realizadas en la branch a eliminar han sido traspasadas (con un merge, se verá posteriormente) a otra rama, es decir, que borrar la rama seleccionada no implique la pérdida de información no existente en ningún otro lugar. Por su contra, la versión con “-D” implica el borrado forzado, desde el cual Git no realizará la comprobación y eliminará irrevocablemente todo el contenido de la rama, esta versión forzada debería utilizarse sólo cuando se haya creado un branch para realizar algún tipo de cambio “loco” para probar algo en sucio, y no impida el correcto desarrollo futuro del proyecto.

Como última posibilidad, podemos renombrar la rama actual en la que se encuentra Git a través del siguiente comando:

```
git branch -r <nuevo_nombre>
```

Para cambiar entre las distintas ramas del repositorio, Git proporciona el comando “checkout”, con la siguiente sintaxis:

```
git checkout <nombre_rama>
```


En la siguiente captura se muestra un ejemplo de éstos comandos, ejecutados sobre el repositorio de Riot.js anteriormente utilizado:

```
[agjacome]-[Caronte]-[.../projects/temporal/riot_js]-[master]
[✓]→ git branch
* master
[agjacome]-[Caronte]-[.../projects/temporal/riot_js]-[master]
[✓]→ git branch testing
[agjacome]-[Caronte]-[.../projects/temporal/riot_js]-[master]
[✓]→ git branch
* master
testing
[agjacome]-[Caronte]-[.../projects/temporal/riot_js]-[master]
[✓]→ git checkout testing
Switched to branch 'testing'
[agjacome]-[Caronte]-[.../projects/temporal/riot_js]-[testing]
[✓]→ git branch -m qwerty
[agjacome]-[Caronte]-[.../projects/temporal/riot_js]-[qwerty]
[✓]→ git branch
master
* qwerty
[agjacome]-[Caronte]-[.../projects/temporal/riot_js]-[qwerty]
[✓]→ git checkout master
Switched to branch 'master'
Your branch is ahead of 'origin/master' by 1 commit.
(use "git push" to publish your local commits)
[agjacome]-[Caronte]-[.../projects/temporal/riot_js]-[master]
[✓]→ git branch -d qwerty
Deleted branch qwerty (was 1440ada).
[agjacome]-[Caronte]-[~/projects/temporal/riot_js]-[master]
[✓]→
```

Fusión de ramas

Los cambios realizados en una rama pueden introducirse de nuevo en la rama original a través del comando “merge” de Git. Merge actualizará la rama actual cogiendo todos los commits realizados en otra rama e introduciéndolos en el directorio git actual. Su sintaxis es la siguiente:

```
git merge <nombre_rama>
```

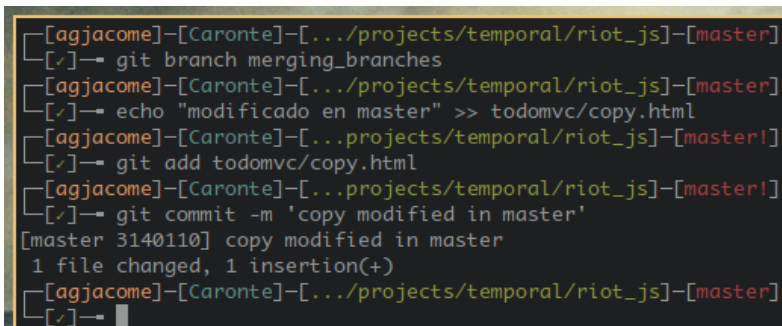
Git generará automáticamente un nuevo commit tras la fusión de las ramas, que normalmente tendrá un mensaje del estilo “Merge branch X”.

Git es un poco inteligente, así que en la mayoría de casos la ejecución de un merge será completamente limpia sin que el usuario final tenga que realizar ninguna acción a mayores. Pero, puede darse el caso de que sí sea necesario realizar una pequeña acción. Al nivel que nosotros trabajaremos con Git, esto podrá ocurrir solamente dado el caso de que tanto en la rama a fusionar y la rama original (posteriormente a la ramificación), se haya modificado el mismo fichero en líneas muy cercanas. En dichos casos, Git no sabrá con cuál de las dos

versiones quedarse, si con la de la rama original o la de la rama a fusionar, y tendremos que indicar cuál es la que nos interesa.

El proceso es un poco complicado para explicarlo, así que vamos a verlo con un pequeño ejemplo realizado sobre el repositorio de Riot.js descargado y modificado en los pasos anteriores de este manual.

Primero, vamos a crear una nueva rama a partir de la rama master. Y, antes de movernos a ésta nueva rama con un “checkout”, vamos a realizar y confirmar una modificación sobre un fichero. Al no haber cambiado de rama, dicho cambio se almacenará en la rama “master”.



```
[agjacome]~[Caronte]-[.../projects/temporal/riot_js]-[master]
[✓]→ git branch merging_branches
[agjacome]~[Caronte]-[.../projects/temporal/riot_js]-[master]
[✓]→ echo "modificado en master" >> todomvc/copy.html
[agjacome]~[Caronte]-[.../projects/temporal/riot_js]-[master!]
[✓]→ git add todomvc/copy.html
[agjacome]~[Caronte]-[.../projects/temporal/riot_js]-[master!]
[✓]→ git commit -m 'copy modified in master'
[master 3140110] copy modified in master
1 file changed, 1 insertion(+)
[agjacome]~[Caronte]-[.../projects/temporal/riot_js]-[master]
[✓]→
```

Ahora, nos moveremos con “checkout” a la rama anteriormente creada, y modificaremos el mismo fichero, añadiendo también una nueva línea, pero de contenido distinto, confirmando posteriormente el cambio.



```
[agjacome]~[Caronte]-[.../projects/temporal/riot_js]-[master]
[✓]→ git checkout merging_branches
Switched to branch 'merging_branches'
[agjacome]~[Caronte]-[...temporal/riot_js]-[merging_branches]
[✓]→ echo "modificado en merging_branches" >> todomvc/copy.html
[agjacome]~[Caronte]-[...temporal/riot_js]-[merging_branches!]
[✓]→ git add todomvc/copy.html
[agjacome]~[Caronte]-[...temporal/riot_js]-[merging_branches!]
[✓]→ git commit -m 'copy modified in merging_branches'
[merging_branches 0460ab6] copy modified in merging_branches
1 file changed, 1 insertion(+)
[agjacome]~[Caronte]-[...temporal/riot_js]-[merging_branches]
[✓]→
```

De nuevo, volveremos a la rama original, master, para intentar fusionar los cambios realizados en la rama merging_branches dentro de la misma, para ello usaremos “merge”.

```
[agjacome]~[Caronte]-[...temporal/riot_js]-[merging_branches]
[✓]→ git checkout master
Switched to branch 'master'
Your branch is ahead of 'origin/master' by 2 commits.
(use "git push" to publish your local commits)
[agjacome]~[Caronte]-[...projects/temporal/riot_js]-[master]
[✓]→ git merge merging_branches
Auto-merging todomvc/copy.html
CONFLICT (content): Merge conflict in todomvc/copy.html
Automatic merge failed; fix conflicts and then commit the result.
[agjacome]~[Caronte]-[...projects/temporal/riot_js]-[master!]
[✗]→
```

Como podemos ver en el mensaje de respuesta que Git nos da al hacer un merge, existe un conflicto en el fichero que ambas ramas han modificado, y Git no sabe con qué modificación ha de quedarse. Para poder indicar cuál es la que nos interesa, o realizar los cambios necesarios para integrar ambas ramas, bastará que abramos el fichero (o ficheros) en los que Git nos indica que existen conflictos con nuestro editor de texto favorito y ver que el contenido presentará este aspecto:

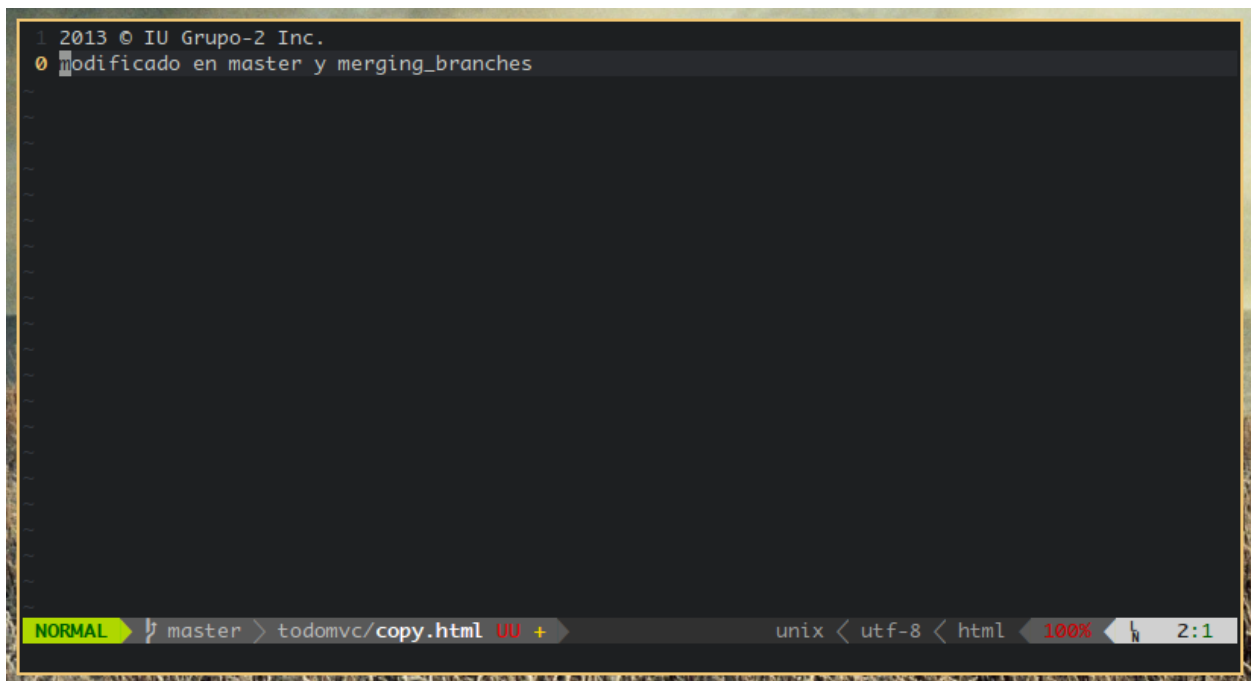
```
0 2013 © IU Grupo-2 Inc.
1 <<<<<< HEAD
2 modificado en master
3 =====
4 modificado en merging_branches
5 >>>>>> merging_branches

NORMAL master > todomvc/copy.html UU
unix < utf-8 < html 17% 1:1
"todomvc/copy.html" 6L, 122C
```

Como se ve en la captura, Git ha incluido unos separadores para indicar qué cambios se han realizado en cada rama. Así <<<<<< HEAD indica que a continuación vendrán las líneas modificadas tal como están en la rama actual (en el caso del ejemplo, master). Tras ese bloque, Git incluye un separador ===== que indica que el bloque a continuación son las líneas

modificadas tal como están en la rama a fusionar (en el caso del ejemplo, merging_branches), y terminará el bloque de conflictos con >>>>>> rama_a_fusionar.

La forma de resolver este conflicto es que nosotros, como usuarios de Git, modifiquemos dicho bloque de conflictos dejando solamente el contenido que nos interesa, y eliminando todos los separadores. En algunos casos simplemente nos interesará quedarnos con una de las versiones, en otras tendremos que realizar cierto trabajo de integración entre ambas. Por ejemplo, en el caso del ejemplo, quizás nos interese marcar que ambas ramas han modificado el contenido, para ello dejaríamos el fichero con este contenido:



Tras esa modificación, para marcar como resuelto finalmente el conflicto tendremos que hacer un commit con el mismo. La forma de hacerlo es con el mismo ciclo add-commit que se ha visto anteriormente. Y tras ello, la fusión de ramas habrá terminado y podremos continuar con nuestro trabajo como nos convenga.

```
[agjacome]~[Caronte]-[...projects/temporal/riot_js]-[master!]  
[✓]→ vim todomvc/copy.html  
[agjacome]~[Caronte]-[...projects/temporal/riot_js]-[master!]  
[✓]→ git status  
# On branch master  
# Your branch is ahead of 'origin/master' by 2 commits.  
#   (use "git push" to publish your local commits)  
#  
# You have unmerged paths.  
#   (fix conflicts and run "git commit")  
#  
# Unmerged paths:  
#   (use "git add <file>..." to mark resolution)  
#  
#       both modified:   todomvc/copy.html  
#  
no changes added to commit (use "git add" and/or "git commit -a")  
[agjacome]~[Caronte]-[...projects/temporal/riot_js]-[master!]  
[✓]→ git add todomvc/copy.html  
[agjacome]~[Caronte]-[...projects/temporal/riot_js]-[master!]  
[✓]→ git commit  
[master 65cfbdf] Merge branch 'merging_branches'  
[agjacome]~[Caronte]-[.../projects/temporal/riot_js]-[master]  
[✓]→ █
```

Repositorios remotos

Git, como se ha comentado anteriormente, es un sistema de control de versiones descentralizado. Y, a diferencia de, por ejemplo, SVN, donde todo commit es enviado automáticamente al repositorio central, Git permite que se realicen en el repositorio local todos los commits que se desee, y que posteriormente puedan enviarse en bloque hacia algún repositorio remoto (en nuestro caso, se enviarán al repositorio de SourceForge).

Descarga de modificaciones

Git, tras hacer un “clone”, crea automáticamente un puntero hacia el repositorio remoto que se ha clonado en local, dicho puntero recibe siempre el nombre de “origin”. Al igual que con el nombre de la rama master, dicho nombre puede modificarse, pero es una convención que se mantenga como “origin” el puntero hacia el repositorio remoto principal (central).

Para ver un listado de todos los punteros a repositorios remotos configurados en el repositorio actual puede utilizarse el comando “remote”:

```
git remote
```

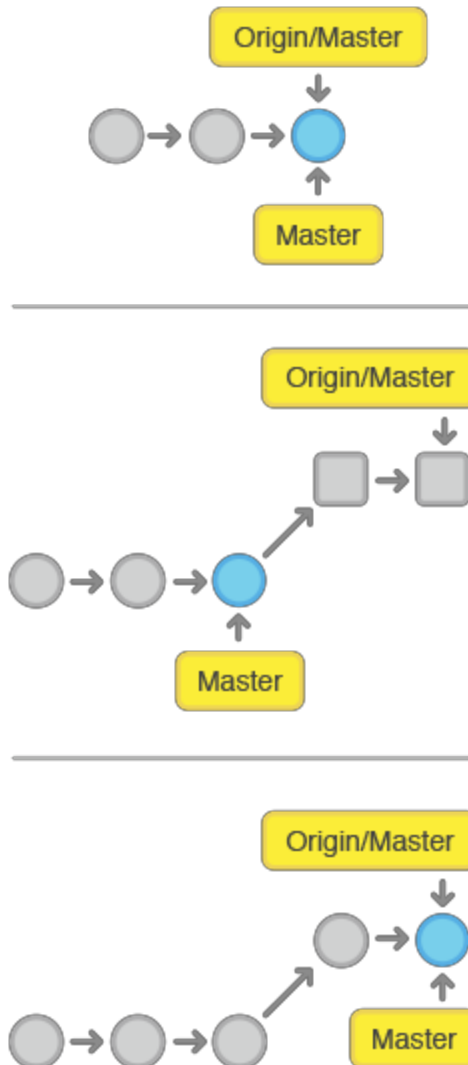
En nuestro caso, sólo tendremos un repositorio remoto, apuntando a SourceForge.

Es posible, en cualquier momento del desarrollo, que el repositorio remoto contenga nuevos commits que no tengamos en local. Es decir, que el repositorio remoto ha sido actualizado (con “push”, lo veremos a continuación). Para poder descargar esas modificaciones, Git proporciona el comando “pull”, cuya sintaxis es la siguiente:

```
git pull <puntero_a_repositorio>
```

En nuestro caso, dado que contaremos con un único repositorio remoto, apuntado por “origin”, esta sintaxis se traduce en:

```
git pull origin
```



Hay que tener en cuenta que pull descargará los cambios que se hayan realizado dentro de la branch actual. Por lo tanto si estamos en la branch master, y hacemos un pull, lo que hará internamente será descargar los commits en la branch remota master y fusionarlos (con un “merge”) en la branch master local. Es posible, que al igual que en un merge normal, surjan conflictos a la hora de realizar un pull, puesto que internamente está realizando un merge con una branch remota, dichos conflictos se resuelven del mismo modo que con merge. De cualquier modo, si las cosas se hacen bien, estos conflictos deberían ser prácticamente inexistentes.

Subiendo cambios

Una vez que estemos conformes con un conjunto de commits realizados sobre una branch, y estemos dispuestos a publicarnos, utilizaremos el comando “push” de Git para enviarlos a un repositorio remoto.

La sintaxis de push es la siguiente:

```
git push <repositorio-remoto> <branch-remota>
```

Por ejemplo:

```
git push origin master
```

Push envía los commits realizados en la branch actual hacia la branch remota del repositorio remoto especificado. Normalmente, si la branch remota aún no existe, se creará en el momento de realizar el push.

Es necesario tener en cuenta que Git nos obliga a estar actualizados a la última versión del repositorio remoto para poder subir cualquier cambio. Así se evita que sobreescribamos ficheros editados por otras personas. Por tanto, el ciclo de envío de cambios a un repositorio remoto es:

1. Se realiza un git pull para descargar todos los cambios.
2. Se solucionan los posibles conflictos surgidos de la fusión de los cambios remotos con los locales.
3. Se envían los cambios realizados al repositorio remoto con git push.

Existen formas de forzar a Git para que nos permita sobreescribir lo que existe en el repositorio remoto. No deberíais ni acercaros a google para buscar cómo se hace, y quien lo haga con el repositorio de SourceForge, que se atenga a las consecuencias.