Uso de repositorios Git con SourceTree

Marcos

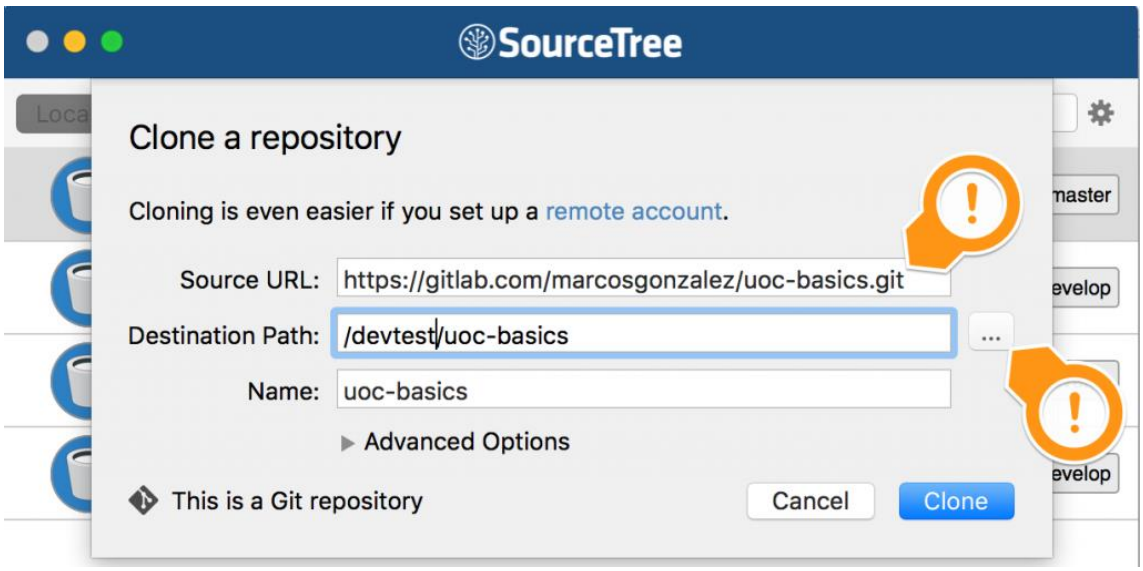
Este ejemplo parte de la base de que hemos creado el repositorio en gitlab.com, pero podría ser aplicable a repositorios creados en otros servicios como GitHub, Bitbucket, etc. En él también se da por hecho que se usa un cliente GUI para el trabajo con el repositorio, en este caso SourceTree.

1) Creamos el proyecto, en este caso como decíamos en GitLab lo que nos termina dando una URL de un repositorio. Para este ejemplo la URL es:

https://gitlab.com/marcosgonzalez/uoc-basics.git

2) Abrimos SourceTree (o la aplicación que hayamos decidido usar para trabajar con el repositorio) y accedemos a New repository > Clone from URL, para disponer de una copia local con la que trabajar.

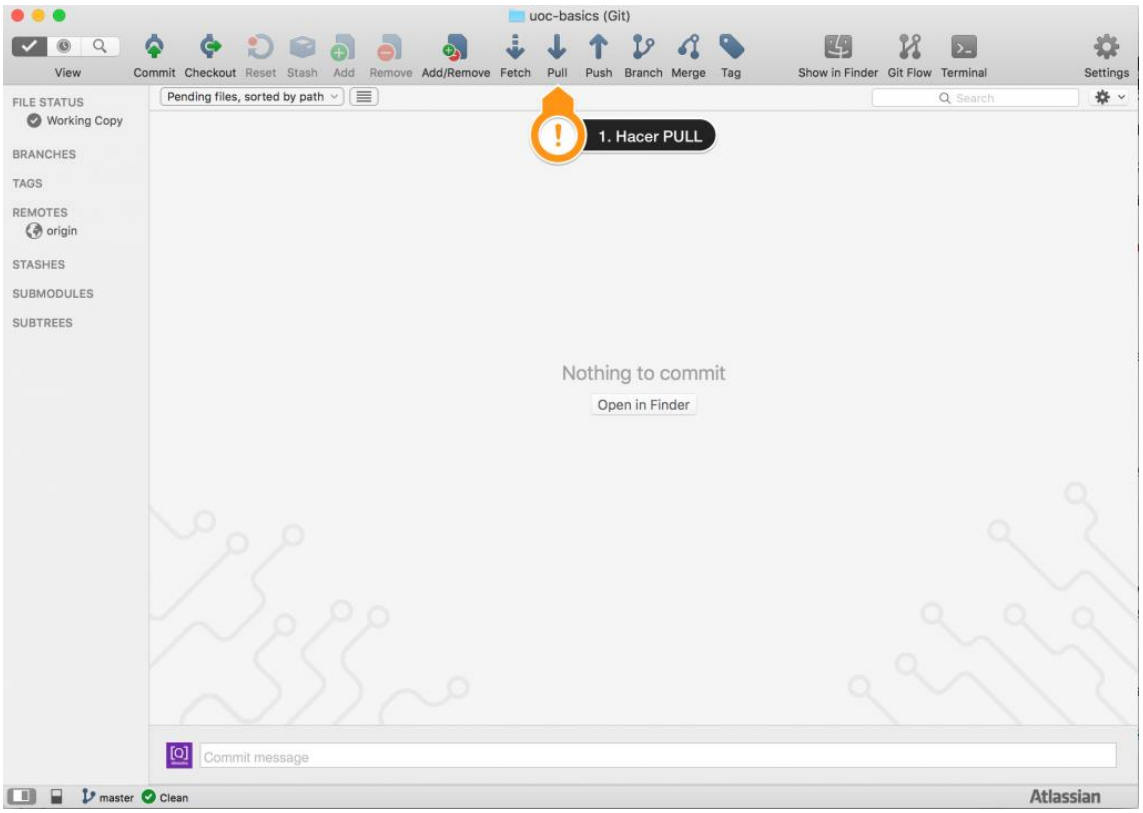
3) En el cuadro de diálogo pegamos la url anterior en el campo Source URL, y elegimos la carpeta donde almacenarlo localmente en el campo Destination path.



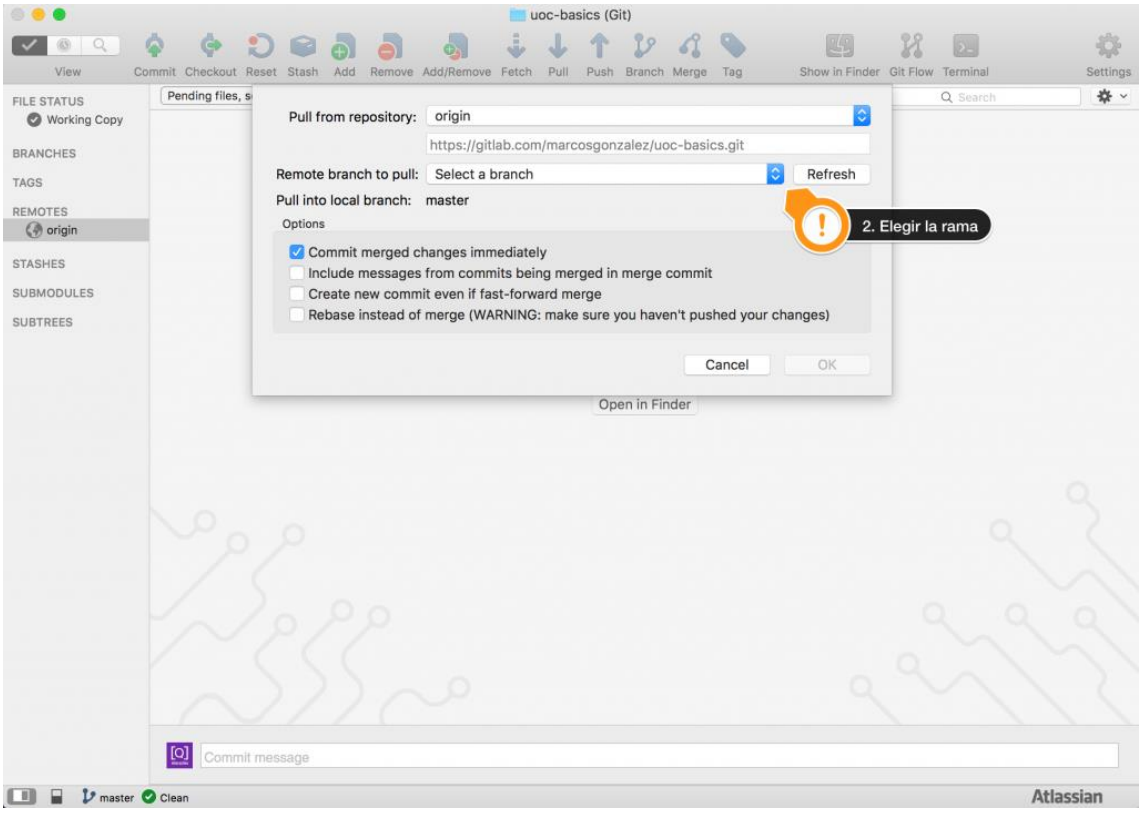
4) Pulsamos el botón Clone, con esto ya tenemos una copia local del repositorio sobre la que trabajar.

5) En la ventana que nos abre SourceTree con nuestros proyectos, hacemos doble clic en el denominado uoc-basics para acceder al mismo.

6) Veremos una ventana como esta:

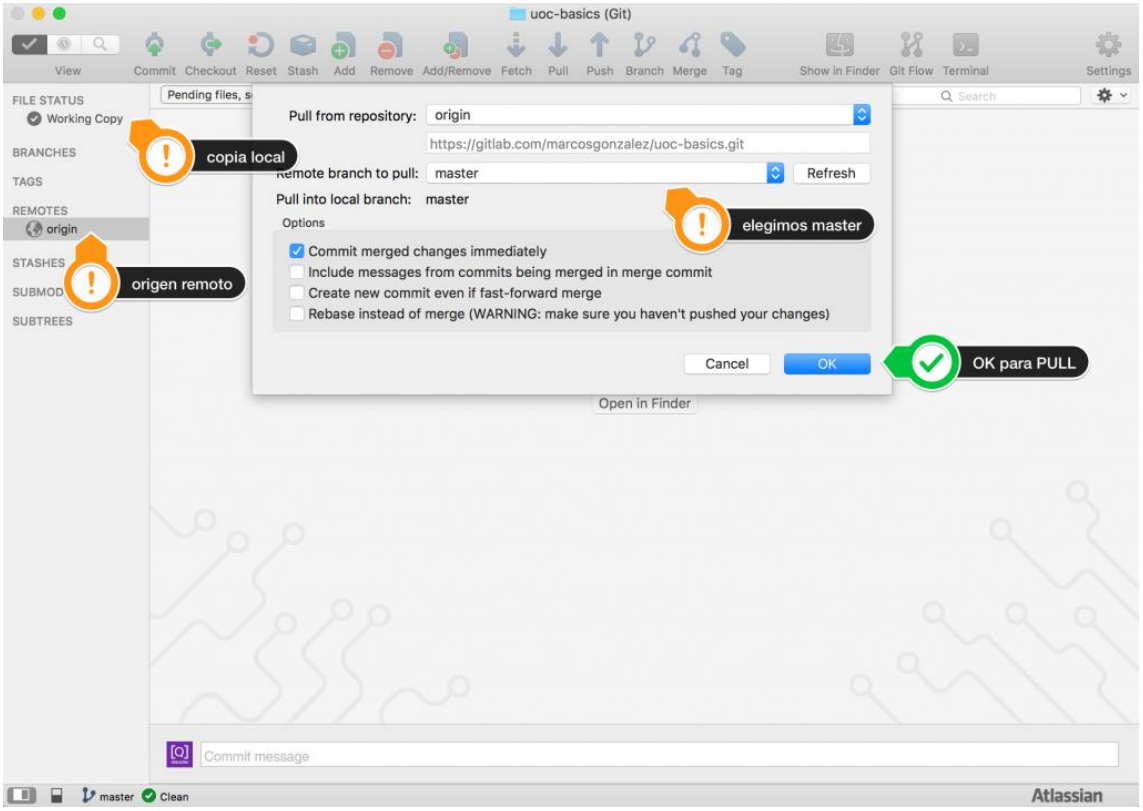


7) El primer paso antes de trabajar es asegurarnos que tenemos la última versión del repositorio actualizada, para minimizar conflictos cuando subamos nuestros cambios, por tanto siempre hacemos un PULL para comenzar, como indica la anterior imagen.

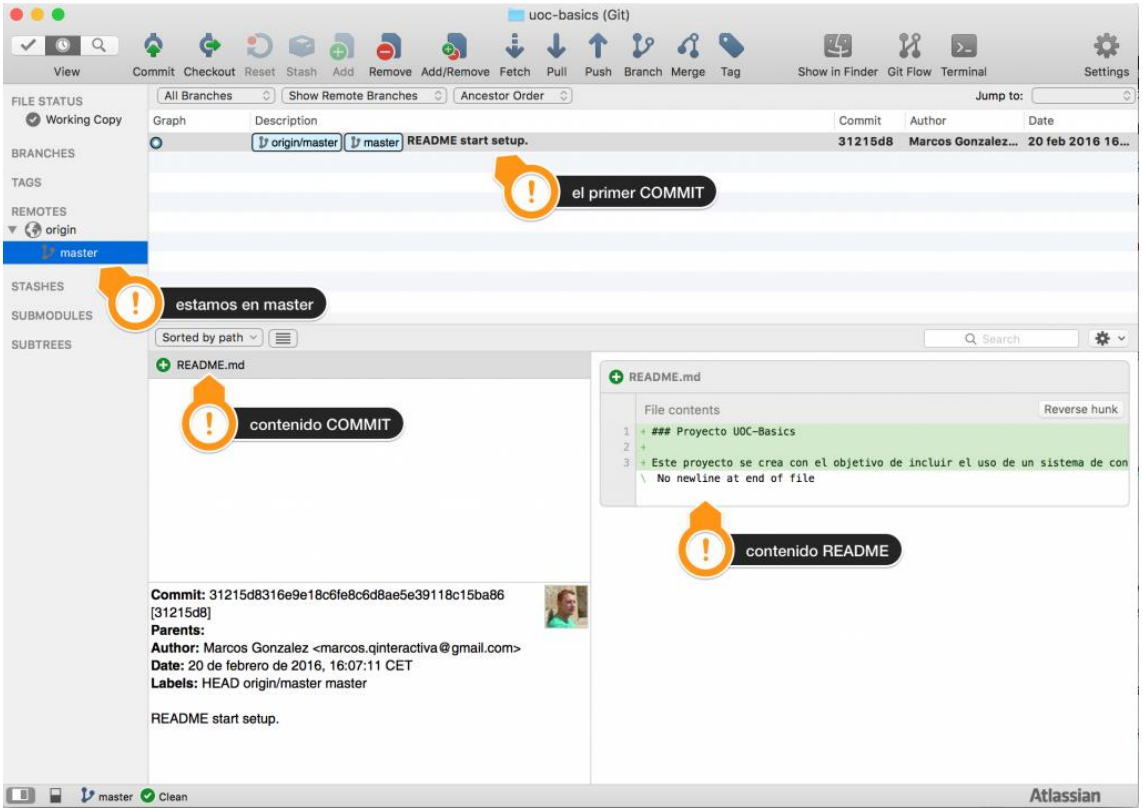


Es posible que no nos muestre ninguna rama y por tanto no nos deje hacer el PULL, si eso fuera así podemos ir al repositorio original en Gitlab y crear un README para que pulsando el botón Refresh, podamos acceder a la rama por defecto del repositorio: master.

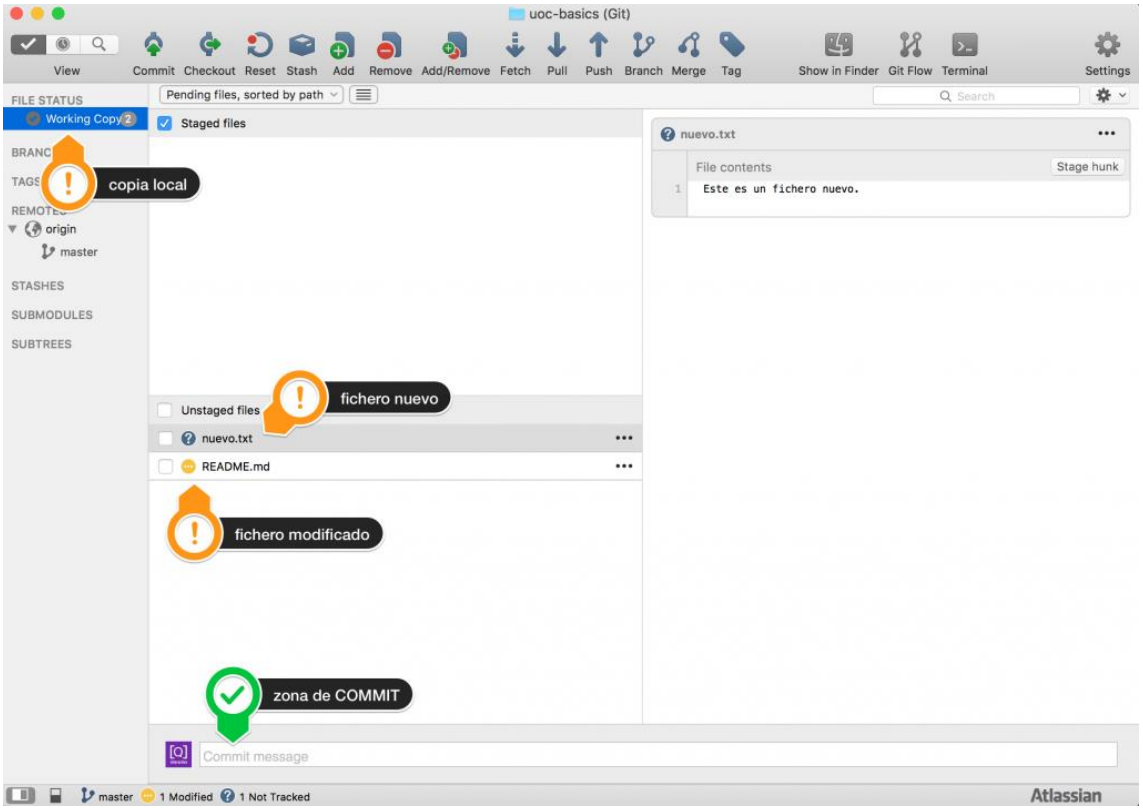
Una vez logrado esto, elegiríamos la rama master y ya podríamos dar al botón OK, para obtener la última versión de los ficheros de dicha rama en nuestra copia local.



8) Esto nos lleva a disponer de los ficheros del repositorio en nuestro equipo, e información de todos los cambios de la rama que queramos del repositorio, como se puede ver en la siguiente captura para la rama master:

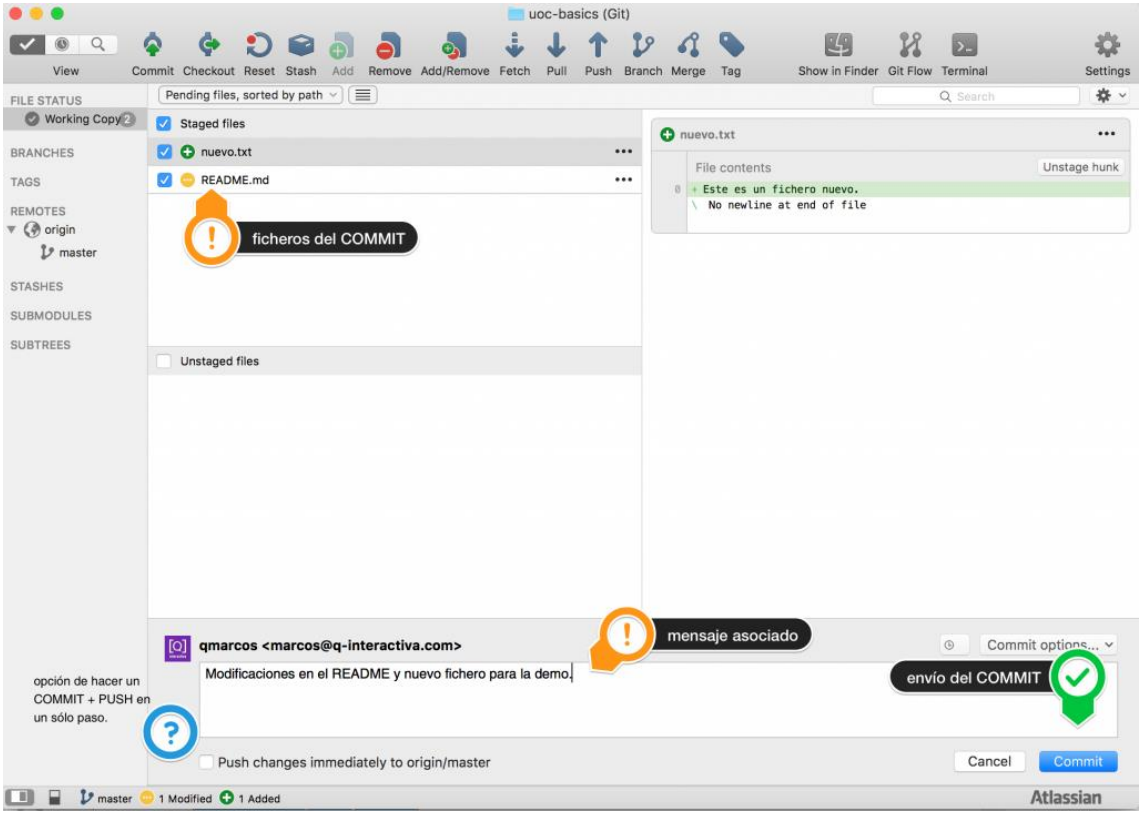


9) Si trabajamos con los archivos del proyecto, seleccionamos la copia local (Working Copy) y modificamos, eliminamos o añadimos alguno, veremos que aparecen los ficheros que han cambiado. En nuestro caso hemos modificado el archivo README.md y creado un fichero nuevo.txt.



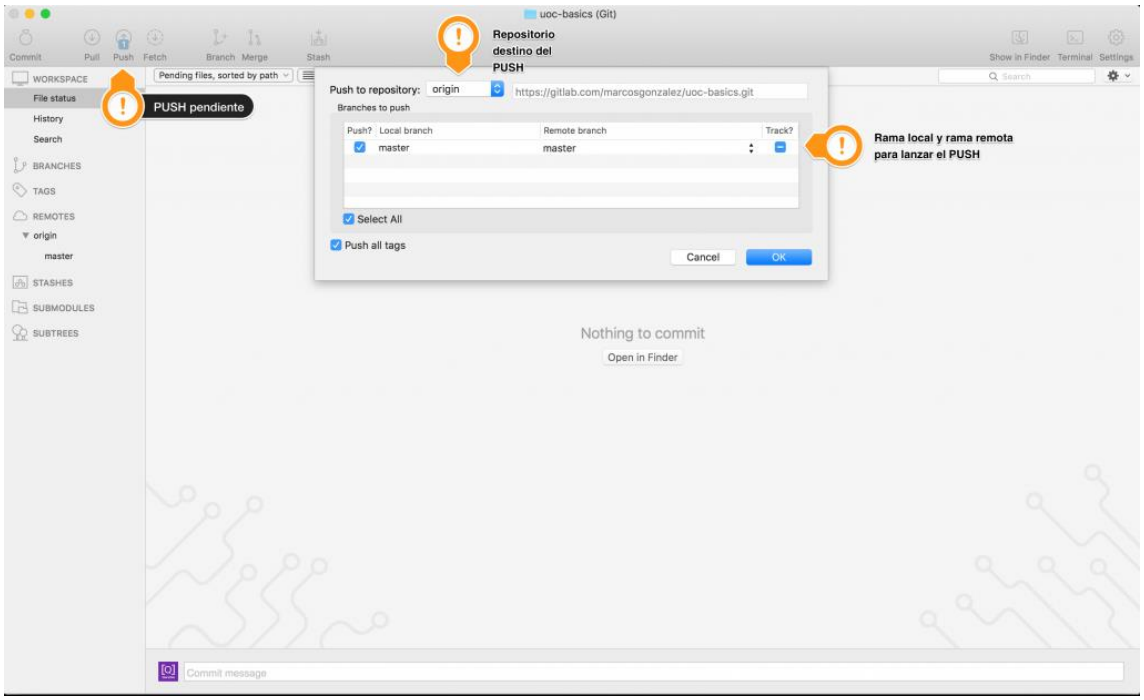
9) Si queremos enviar los cambios al repositorio para que queden archivados en el histórico de control de versiones, hemos de hacer un COMMIT que se complementa con un PUSH para que tengan efecto en la rama remota (en este caso master).

Para ello lo primero que haremos es marcar los ficheros de «Unstaged files» que queremos procesar para pasarlas a «Staged files», que serán las que tenga en cuenta el COMMIT. Según las vamos marcando irán quedando como «Staged» y cuando tengamos todas las que queramos (en este caso los dos cambios) podemos escribir el mensaje de COMMIT que queremos asociar al mismo, que ha de ser lo más representativo posible de los cambios que contiene.



10) Una vez seleccionados los ficheros y escrito el mensaje podemos enviar el COMMIT, y como se indica en la captura anterior, marcar incluso que se traslade a la rama en el origen correspondiente con un PUSH inmediatamente después. Lo normal tras el COMMIT es lanzar el PUSH, con lo que esta opción es un atajo al siguiente paso.

11) Como hemos dicho una vez que los cambios se han enviado al repositorio con COMMIT, veremos el indicador en el botón de PUSH que existen elementos por trasladar a la rama en el origen (en este caso master). Si queremos que tengan efecto debemos hacer PUSH mediante dicho botón que nos muestra el siguiente cuadro de trabajo, en el que debemos pulsar OK una vez configurados los parámetros (normalmente los que ofrece son los que queremos aplicar)



Podéis apreciar un cambio en la interfaz de las capturas, debido a que la nueva versión de SourceTree ha salido muy recientemente y propone una interfaz simplificada y más limpia. De hecho la botonera superior se ha reducido a los comandos básicos lo que permite menos confusión al trabajar con la herramienta a un nivel más básico.

12) Si accedemos a nuestro repositorio en GitLab podremos ver como en el apartado Actividad refleja el push y en Commits podemos ver los existentes y en cada uno de ellos navegar para ver los ficheros implicados y los cambios realizados. Haciendo clic sobre el texto del commit, podemos ver dicha situación:

Forma de trabajo habitual

Como hemos comentado, la forma de trabajo habitual es hacer siempre un PULL antes de comenzar a trabajar, y antes de cada COMMIT. No es mala práctica hacer varios COMMITS en una misma sesión de trabajo, pero tampoco es conveniente hacer un COMMIT a cada cambio.

Si todos los miembros del equipo trabajan de esta forma se reducen las probabilidades de que se den situaciones que requieran de un MERGE, que se produce cuando un miembro del equipo modifica un fichero que a la hora de volcarse al repositorio, se detecta que no es la última versión… es decir mientras hemos hecho cambios en un fichero, otro miembro del equipo lo ha modificado y vuelto a subir, por lo que la versión que nosotros subimos no está cambiada sobre la última versión disponible, y toca seleccionar con qué partes de cada zona del fichero cambiada nos quedamos para unificar los cambios de los dos miembros.

Con respecto a las ramas, una buena política de trabajo es crear ramas para desarrollar bloques de funcionalidad concretos, de manera que se pueda trabajar sobre ella con libertad, y el proceso de unificación sea sobre el conjunto de los cambios una vez finalizado. De esta forma se reducen también las colisiones de código entre desarrollos que no pertenecen a la misma funcionalidad.

En realidad existen diferentes estrategias de gestión de ramas para el trabajo y cada equipo puede encontrar una mejor que otra en función de la tipología del proyecto y de la propia manera de trabajar sobre el repositorio, aunque en equipos pequeños estas estrategias pierden relevancia al simplificarse bastante la probabilidad de conflictos y trabajos simultáneos.

Uno de los recursos más reconocidos para la gestión de ramas de forma avanzada es lo que se conoce como Git-flow (de Vincent Driessen)

¿Qué es git-flow?

Empezamos una serie de artículos sobre git-flow, conjunto de extensiones de git que facilitan la gestión de ramas y flujos de trabajo.

Si quieres seguir esta serie, debes disponer de una máquina con git instalado:

Windows: msysgit que puedes descargar de este enlace

Mac: a través de homebrew o macports

Linux: a través del gestor de paquetes de tu distribución

Flujos de trabajo

Hace unos días participé en el Open Space de Calidad del Software organizado en Madrid este mes de febrero. En la reunión se abordaron varios temas que iban desde responder a preguntas como ¿qué se calidad del software? ¿cuánto cuestan los tests funcionales? o ¿cómo hacer testing de desarrollos para dispositivos móviles? pasando por otros tan exóticos como el Pirata Roberts, llegando incluso a plantearse hasta la eliminación de los responsables de calidad de la faz de la tierra.

En casi todas las conversaciones en las que tuve la oportunidad de participar había un denominador común: las ramas. Se hablaba de ramas para hacer hot-fixes urgentes, ramas para desarrollar nuevas versiones separadas de las ramas maestras donde está

la versión en producción. Ramas para probar nuevas versiones, ramas y repositorios para trabajar con proveedores externos, ramas para hacer pruebas en pre-producción, ramas para que los departamentos de calidad hagan sus pruebas antes de liberar nuevas versiones. Con git podemos crear ramas “como churros” y ese fin de semana tuve la oportunidad de compartir con varios colegas de profesión cómo utilizar las ramas para hacer el bien. Sin embargo, esta facilidad para crear ramas también se puede utilizar para hacer el mal y sembrar el terror. Más de una vez he visto ramas creadas sin ningún criterio, sin ningún flujo de información detrás que las sustente. Esta situación suele llevar al repositorio al caos más absoluto.

Para no acabar en el caos, debemos establecer unas “reglas del juego” que todo el equipo debe respetar. Aunque a grandes rasgos casi todos los proyectos pueden utilizar unas reglas de base comunes, las reglas deben ser flexibles para adaptarse a los cambios que puedan surgir en el tablero de juego; al fin y al cabo, las necesidades y particularidades de cada equipo, empresa o proyecto no son las mismas.

¿Y cuáles son estas reglas base comunes? En enero de 2010 Vincent Driessen publicó en su blog un artículo en el que compartía con la comunidad un flujo de trabajo que a él le estaba funcionando: “A successful Git branching model”. Como él mismo cuenta en el artículo (te recomiendo encarecidamente que lo leas) Vincent propone una serie de “reglas” para organizar el trabajo del equipo.

Ramas master y develop

El trabajo se organiza en dos ramas principales:

Rama master: cualquier commit que pongamos en esta rama debe estar preparado para subir a producción

Rama develop: rama en la que está el código que conformará la siguiente versión planificada del proyecto

Cada vez que se incorpora código a master, tenemos una nueva versión.

Además de estas dos ramas, Se proponen las siguientes ramas auxiliares:

Feature

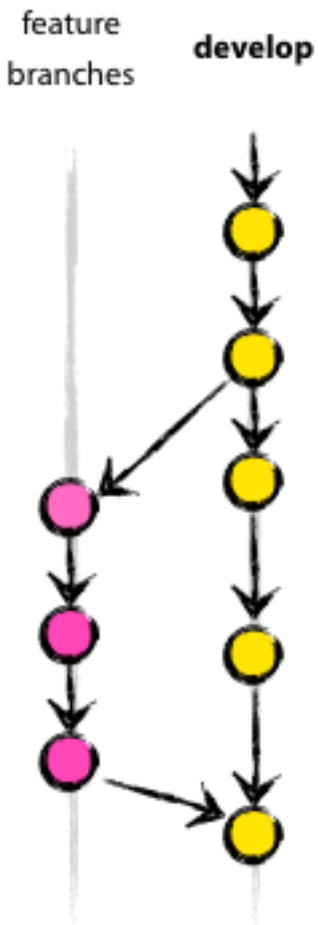
Release

Hotfix

Cada tipo de rama, tiene sus propias reglas, que resumimos a continuación.

Feature or topic branches

feature branches



fuente: nvie http://nvie.com/posts/a-successful-git-branching-model/

Se originan a partir de la rama develop.

Se incorporan siempre a la rama develop.

Nombre: cualquiera que no sea master, develop, hotfix-* o release-*

Estas ramas se utilizan para desarrollar nuevas características de la aplicación que, una vez terminadas, se incorporan a la rama develop.

Release branches

Se originan a partir de la rama develop

Se incorporan a master y develop

Nombre: release-*

Estas ramas se utilizan para preparar el siguiente código en producción. En estas ramas se hacen los últimos ajustes y se corrigen los últimos bugs antes de pasar el código a producción incorporándolo a la rama master.

Hotfix brancheshotfix branches

Se origina a partir de la rama master

fuente http://nvie.com/posts/a-successful-git-branching-model/

Se incorporan a la master y develop

Nombre: hotfix-*

Esas ramas se utilizan para corregir errores y bugs en el código en producción. Funcionan de forma parecida a las Releases Branches, siendo la principal diferencia que los hotfixes no se planifican.

¿**Qué es git-flow?**

Si queremos implementar este flujo de trabajo, cada vez que queramos hacer algo en el código, tendremos que crear la rama que corresponda, trabajar en el código, incorporar el código donde corresponda y cerrar la rama. A lo largo de nuestra jornada de trabajo necesitaremos ejecutar varias veces al día los comandos git, merge, push y pull así como hacer checkouts de diferentes ramas, borrarlas, etc. Git-flow son un conjunto de extensiones que nos ahorran bastante trabajo a la hora de ejecutar todos estos comandos, simplificando la gestión de las ramas de nuestro repositorio.

La flexibilidad de git…y el sentido común

Las «reglas» que Vincent plantea en su blog son un ejemplo de cómo git nos permite implementar un flujo de trabajo para nuestro equipo. Estas no son reglas absolutas, bien es cierto que pueden funcionar en un gran número de proyectos, aunque no siempre será así. Por ejemplo ¿qué pasa si tenemos que mantener dos o tres versiones diferentes de una misma aplicación? digamos que tenemos que mantener la versión 1.X, la 2.X y la 3.X. El tablero de juego es diferente así que necesitaremos ampliar y adaptar estas reglas para poder seguir jugando.

git es una herramienta que nos permite modificar estas reglas y, lo que es más importante, irlas cambiando y adaptando a medida que el proyecto avanza y el equipo madura. Una vez más, una buena dosis de sentido común será nuestra mejor aliada para responder las preguntas que nos surjan durante el camino.