Capítulo 3

# EL CONTROL DE VERSIONES

## Contenidos

El control de versiones es un sistema que registra los cambios realizados sobre un archivo o conjunto de archivos a lo largo del tiempo, de modo que podamos recuperar versiones específicas más adelante. Ese conjunto de archivos y todo su historial de cambios son lo que comunmente se reconoce como repositorio.

Existen muchos sistemas de control de versiones: CVS, Subversion, Mercurial, Git, etc. Algunos de ellos son centralizados; otros, distribuidos. Los distribuidos, al contrario de los centralizados, no exigen que el repositorio de código resida en un servidor central al que todos los usuarios del sistema de control de versiones acceden. Un sistema de control de versiones distribuido permite mantener más de un repositorio de código, incluso en servidores diferentes.

En este capítulo nos centraremos en uno de los sistemas de control de versiones más ampliamente utilizados en la actualidad: Git. Veremos, en lo sucesivo, cómo funciona y en qué consiste; además de qué herramientas tenemos a nuestro alcance.

Sección 1

# GIT

Git es un sistema de control de versiones libre y de código abierto. Se trata de un sistema de control de versiones distribuido capaz de albergar proyectos de gran envergadura de forma ágil y eficiente.

Muchas herramientas, tales como editores de código, ofrecen integración con Git, de modo que es posible hacer el control de versiones sin salir del editor.

La herramienta Git, en si misma, es una herramienta de línea de comandos. Admite una serie de órdenes que,

ejecutadas desde la ruta o carpeta del repositorio en cuestión, tendrán un efecto determinado sobre éste.

Sección 2

# COMANDOS DE GIT

Git admite diversas acciones, todas ellas a través de los comandos.

## Creación de un repositorio

Un repositorio se crea a partir de una carpeta, que puede estar vacía o no. Para crear un repositorio, escribiremos el comando **git init**:

```
git init [-q | --quiet] [--bare] [--template=<template_directory>]
    [--separate-git-dir <git dir>]
    [--shared[=<permissions>]] [directory]
```

Esto únicamente creará el repositorio. Es decir, prepara la carpeta donde estará nuestro proyecto para el control de versiones. A partir de este momento, el repositorio está listo para que podamos registrar los cambios que hagamos en nuestro proyecto como veremos más adelante.

Mención especial para la opción **--bare**, la cual implica la inicialización de un repositorio sin copia de trabajo local. Este tipo de repositorios así inicializados se utilizan a menudo en servidores de producción, en conjunción con herramientas de integración continua.

## Clonación de un repositorio

Muchas veces nos interesará replicar un repositorio para tener una copia de trabajo local. Ésta es la típica operación con la que comenzamos a trabajar sobre un repositorio alojado remotamente; así lo clonamos y lo modificamos localmente, para luego registrar esos cambios locales y remitirlos al repositorio remoto.

```
git clone [--template=<template_directory>]
    [-l] [-s] [--no-hardlinks] [-q] [-n] [--bare] [--mirror]
    [-o <name>] [-b <name>] [-u <upload-pack>]
  [--reference <repository>]
    [--dissociate] [--separate-git-dir <git dir>]
    [--depth <depth>] [--[no-]single-branch]
    [--recursive | --recurse-submodules] [--[no-]shallow-
submodules]
    [--jobs <n>] [--] <repository> [<directory>]
```

Un ejemplo de uso de este comando es el siguiente, que clonará un repositorio alojado en GitHub en una carpeta recién creada en nuestra máquina:

```
git clone https://github.com/keon/algorithms.git
```

## Registro de cambios

A medida que vamos avanzando en el desarrollo de nuestro proyecto, desearemos ir registrando los cambios que a éste le hacemos. De este modo, mantenemos un historial de cambios y hacemos que sea posible, en cualquier momento, volver atrás si por algún motivo cometemos errores en la programación y las cosas dejan de funcionar.

La primera operación que hay que efectuar para registrar un cambio es indicarle a Git que debe hacer seguimiento de los ficheros cambiados. El comando para esta acción es **git add**, el cual suele ir acompañado de los nombres de los ficheros o carpetas a incorporar en el siguiente cambio a registrar.

```
    git add [--verbose | -v] [--dry-run | -n] [--force | -f] [--
interactive | -i] [--patch | -p]

        [--edit | -e] [--[no-]all | --[no-]ignore-removal | [--update |
-u]]

        [--intent-to-add | -N] [--refresh] [--ignore-errors] [--ignore-
missing]

        [--chmod=(+|-)x] [--] [<pathspec>…]
```

Un ejemplo básico, a la par que muy socorrido, es el de incorporar toda la carpeta del proyecto al siguiente cambio.

Git sabrá qué hacer, tomando sólo los ficheros modificados:

```
git add .
```

Finalmente, hacemos que los cambios sean efectivos mediante el comando **git commit**. Esta orden es la encargada de registrar el cambio en el historial del repositorio. Es posible acompañar a este comando con un mensaje descriptivo del cambio o cambios que hemos realizado:

```
git commit -m  ‘Cambios en las hojas de estilos’
```

## Comprobación del estado del repositorio

En cualquier momento podemos inquirir al Git acerca del estado del repositorio en el que estamos trabajando. Con el comando **git status** se obtiene esa información.

Si lo que queremos es acceder al histórico de cambios efectuados al repositorio, el comando que necesitamos es **git log**.

## Más allá de lo que vemos

Hasta ahora, hemos visto cómo crear un repositorio y registrar cambios en éste. Sin embargo, no hemos tratado

aún el tema de sincronizar un repositorio local con otro remoto. Tampoco vimos cómo puede un repositorio de Git mantener diversos historiales de cambios, y cómo se puede conmutar entre unos y otros. Lo veremos en la siguiente sección.

Sección 3

# RAMAS Y ETIQUETAS

Cuando hablamos de ramas (branches), significa que partimos de la rama principal de desarrollo (master) y a partir de ahí continuamos trabajando en otra secuencia de modificaciones (commits) que divergen de esa rama principal. Uno de los puntos mas fuertes de Git es su sistema de ramas: es rápida, casi instantánea. En Git es bastante fácil moverse de una rama a otra, y rápido; lo cual lo hace idóneo para trabajar en proyectos grandes.

Git promueve un ciclo de desarrollo donde las ramas se crean y se unen entre sí con frecuencia en el día a día. Entender y manejar esta funcionalidad nos brinda una poderosa y exclusiva herramienta que puede, literalmente, cambiar la forma en la que trabajamos en equipo. Aprenderemos a crear ramas nuevas a partir de las existentes y a fusionar una rama en otra.

Por otra parte, Git tiene la habilidad de etiquetar (tag) puntos específicos en la historia como importantes. Generalmente la gente usa esta funcionalidad para marcar puntos donde se ha lanzado alguna versión (v1.0, y así sucesivamente). Veremos cómo listar las etiquetas disponibles, crear nuevas etiquetas y qué tipos diferentes de etiquetas hay.

## Creando etiquetas

Crear una etiqueta en Git es simple. La forma más fácil es especificar -a al ejecutar el comando tag:

```
$ git tag -a v1.4 -m 'my version 1.4'
$ git tag
v0.1
v1.3
v1.4
```

El parámetro -a sirve para que indicarle al comando que la etiqueta ha de tener una anotación, esto es, información adicional acerca del autor, código de comprobación (checksum), etc.

El parámetro -m especifica el mensaje, el cual se almacena con la etiqueta. Si no se especifica un mensaje

para la etiqueta, Git lanza un editor de texto para poder escribirlo.

Tanto el parámetro -a como el -m son opcionales.

## Listando etiquetas existentes

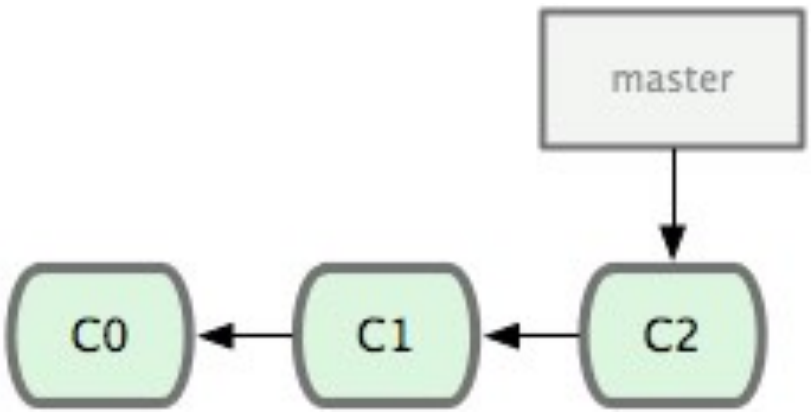
Listar las etiquetas disponibles en Git es sencillo, Simplemente escribiremos **git tag**:

```
$ git tag
v0.1
v1.3
```

Este comando lista las etiquetas en orden alfabético; el orden en el que aparecen no es realmente importante.

## Creando una nueva rama

Supongamos que partimos de un repositorio con una única rama master con algunos cambios registrados (commits):



Para crear una nueva rama y saltar a ella, en un solo paso, puedes utilizar el comando git checkout con la opción -b:

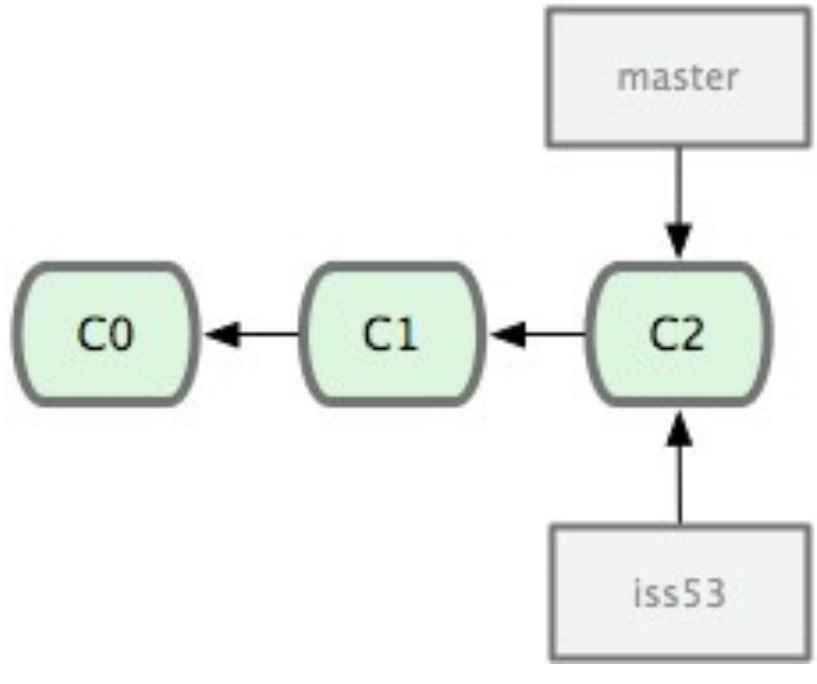
```
$ git checkout -b iss53
Switched to a new branch "iss53"
```

Esto es, en realidad, una forma abreviada de ejecutar estas dos órdenes:

```
$ git branch iss53
$ git checkout iss53
```

La primera sirve para crear la nueva rama; la segunda, para saltar a ella.

Desde este momento, estaremos trabajando en la nueva rama (denominada "iss53"), y todos los cambios que efectuemos sobre el repositorio se añadirán a la rama recién creada. El estado actual del repositorio muestra ahora una rama nueva.



Al hacer un nuevo commit o registro de cambios, el repositorio evoluciona hacia un nuevo estado, donde el último commit ya no sigue la línea de cambios de “master”, sino que cuelga de la nueva rama “iss53”.

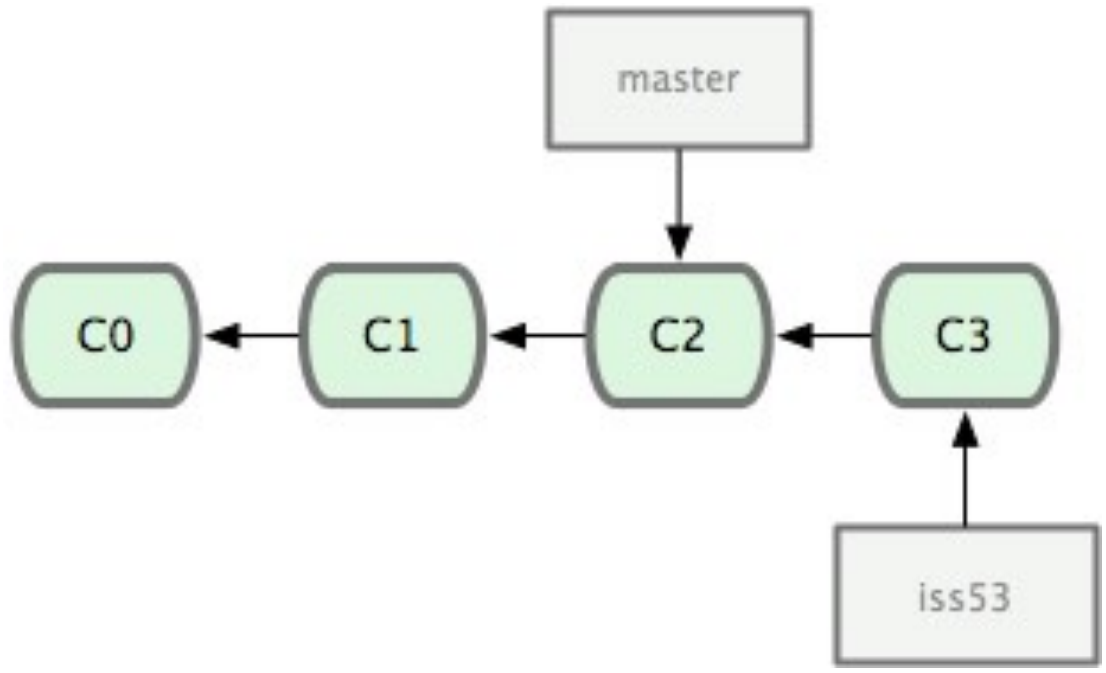
```
    git commit -m  ‘añadiendo cambios a la nueva rama’
```

## Fusionar una rama en otra

Supongamos que acabamos el trabajo en la rama “iss53” y queremos fusionarlo (merge) con la rama master. Para ello, procederemos a fusionar la rama iss53. Simplemente, cambiando (checkout) a la rama donde queremos integrar los cambios de toda esta rama y lanzando el comando git merge:

```
    $ git checkout master
    $ git merge iss53
    Merge made by recursive.
     README |    1 +
     1 files changed, 1 insertions(+), 0 deletions(-)
```

Si todo va bien, el estado del repositorio evolucionará a lo siguiente:




Sin embargo, puede ocurrir que la fusión no pueda hacerse de forma totalmente automática. Git nunca decidirá por nosotros en aquellos cambios que supongan un conflicto.

Todo aquello que sea conflictivo y no se haya podido resolver, se marca como "sin fusionar" (unmerged). Git añade a los archivos conflictivos unos marcadores especiales de resolución de conflictos. Marcadores que te guiarán cuando abras manualmente los archivos implicados y los edites para corregirlos. El archivo conflictivo contendrá algo como:

```
<<<<<<< HEAD:index.html
<div id="footer">Thank you</div>
=======
<div id="footer">
  Thanks
</div>
>>>>>>> iss53:index.html
```

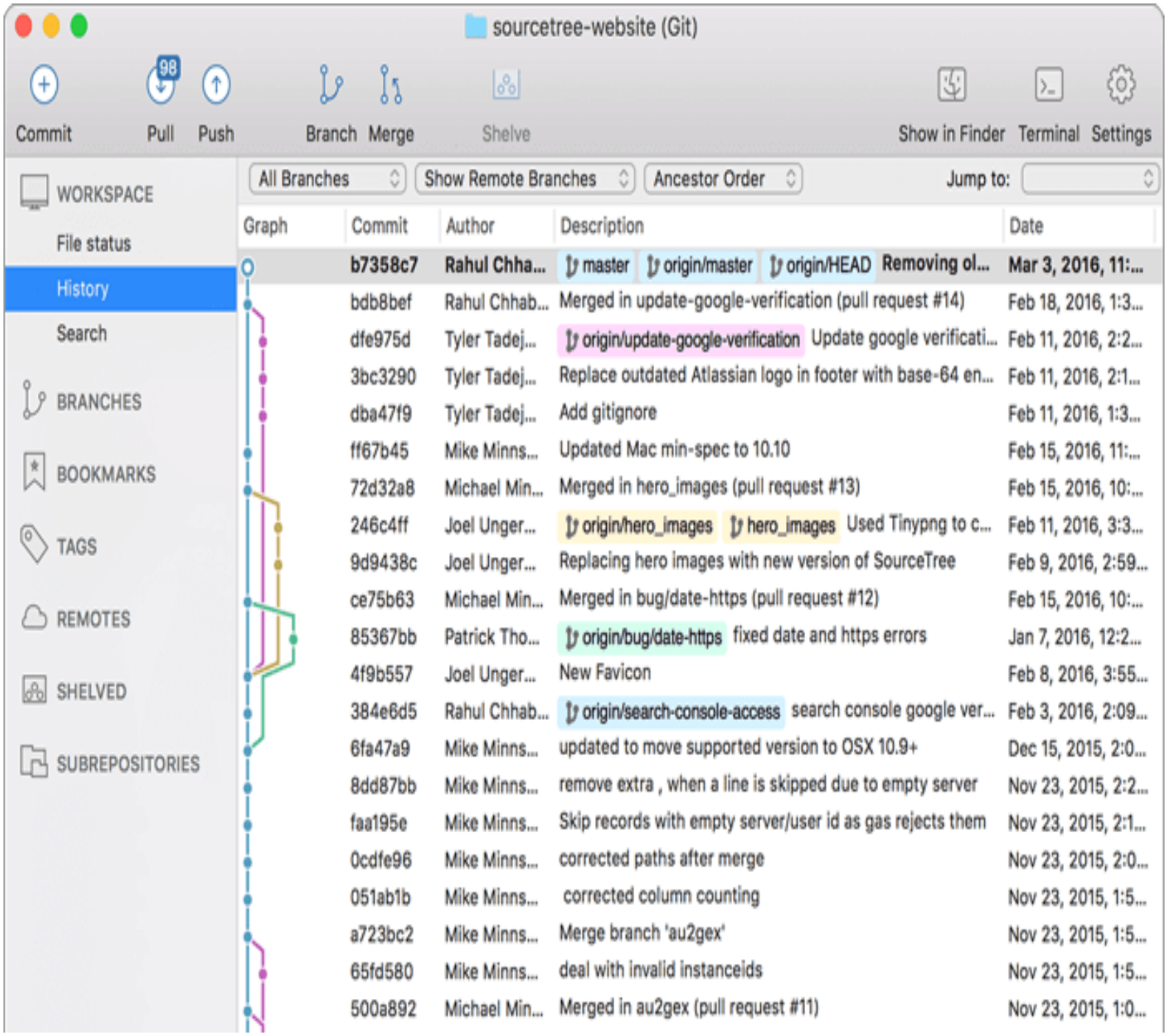
La resolución de este conflicto corre de nuestra cuenta y debemos editar el archivo en cuestión para mantener una de las partes y desechar la otra. Una vez guardado el fichero ya sin conflictos, intentaremos un nuevo git commit para registrar los cambios.

Para ver qué archivos contienen conflictos tras una fusión, podemos lanzar el comando **git status**.

Sección 4

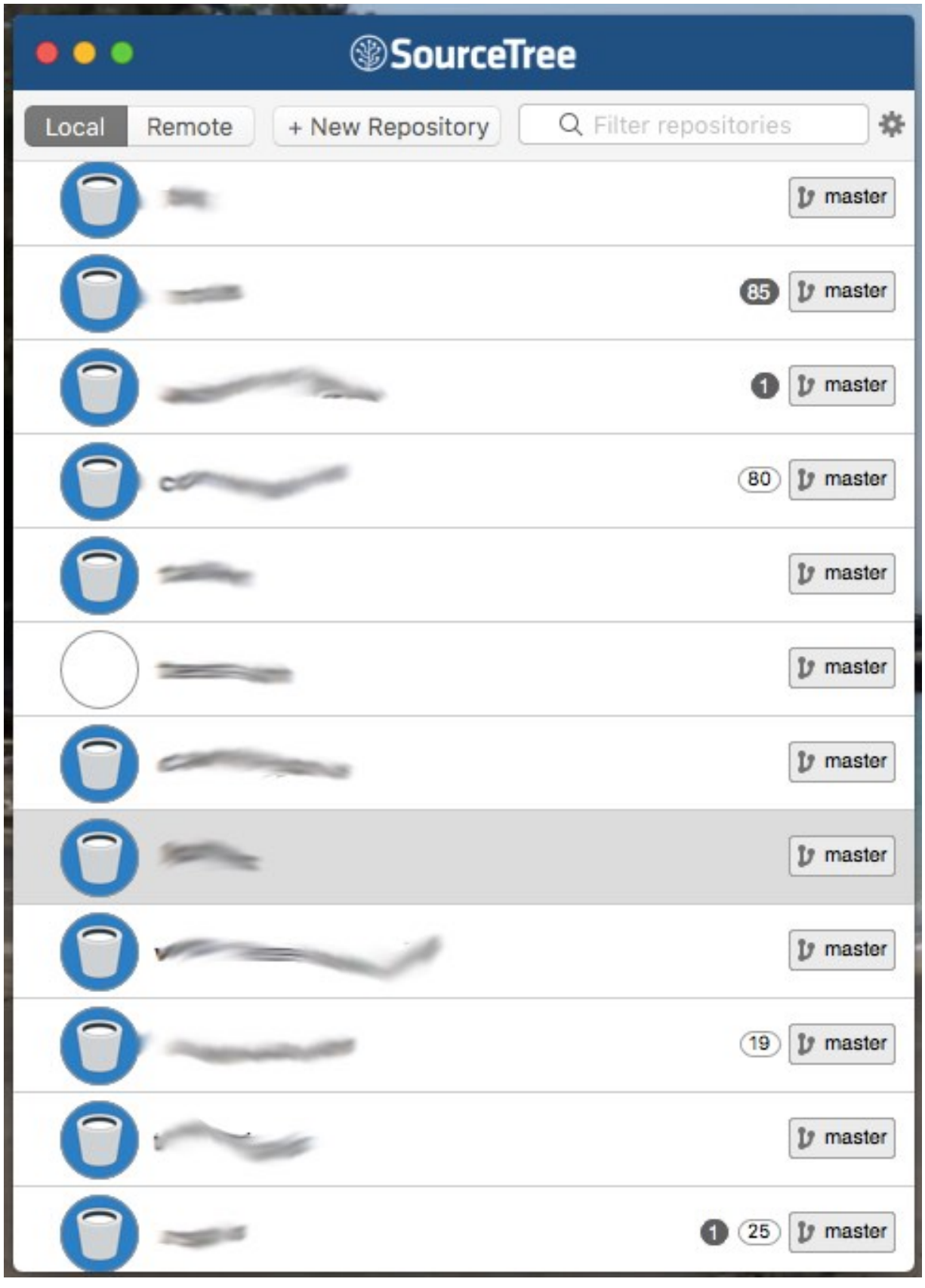
# SOURCETREE

Git es una herramienta poderosa y eficiente, pero nos exige trabajar con ella desde la línea de comandos y conocer la sintaxis de las órdenes que admite. Una forma más visual y cómoda de trabajar con Git es a través del programa SourceTree.




SourceTree nos permite trabajar con repositorios Git sin necesidad de escribir los comandos y aportando mucha información visual sobre el estado de un repositorio. Vamos a conocer esta fantástica herramienta.

## Selector de repositorios




Este es el selector de repositorios de SourceTree. Aquí veremos todos los repositorios en los que estamos trabajando. Como puede verse, hay un botón que pone “Local” y otro que pone “Remote”. El primero mostrará un listado de repositorios en nuestra máquina. El botón “Remote” tiene efecto únicamente si hemos vinculado SourceTree a alguna cuenta que tengamos en BitBucket,

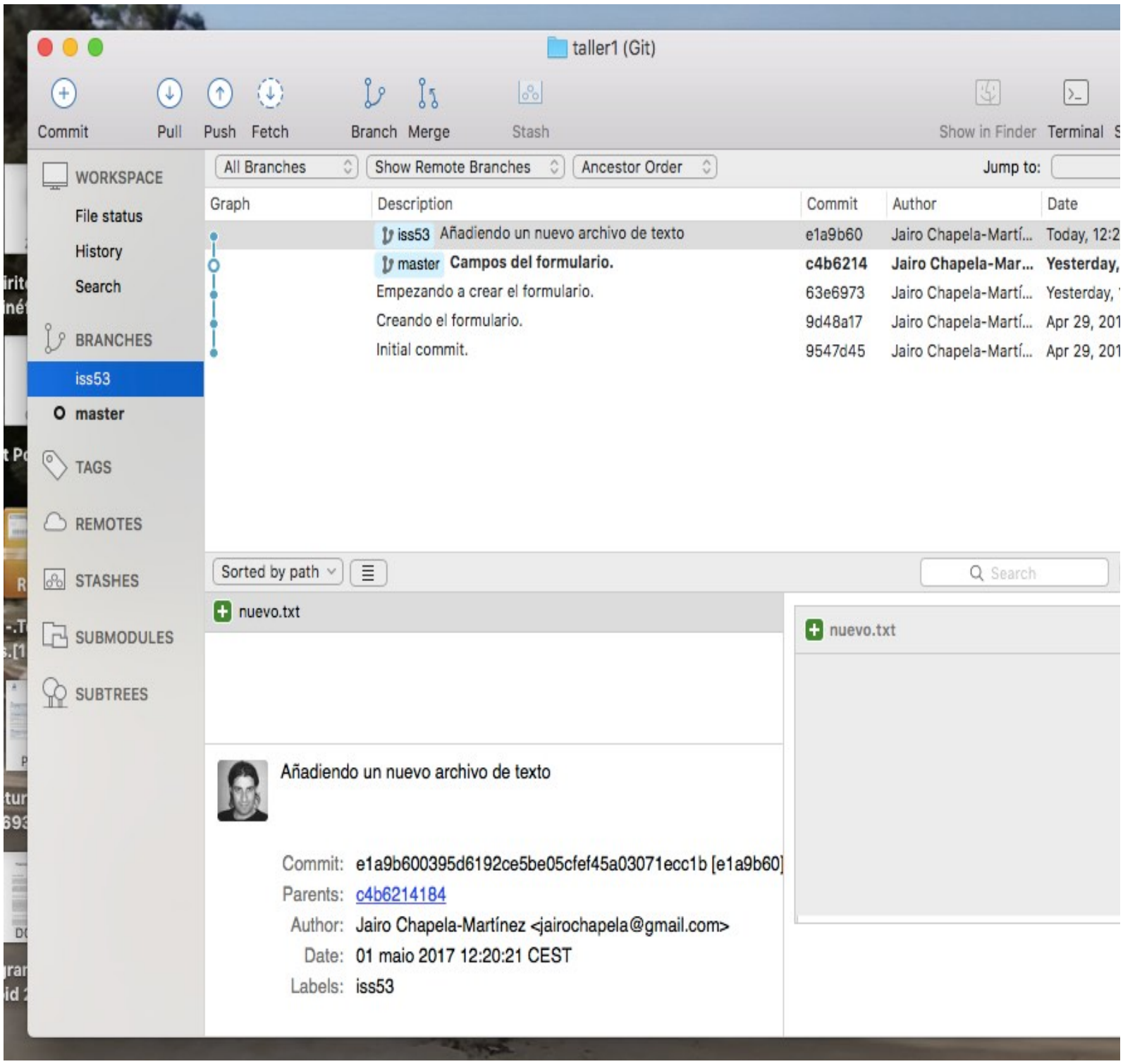
GitHub o similar; en ese caso, nos mostrará los repositorios que allí tengamos alojados.

Hay otro botón que pone “New Repository”. Este botón permite añadir un nuevo repositorio — local o remoto— al selector de repositorios de SourceTree.

Al seleccionar un repositorio, se abre una nueva ventana con toda la información del repositorio. Aquí podremos trabajar de una forma visual con el repositorio.

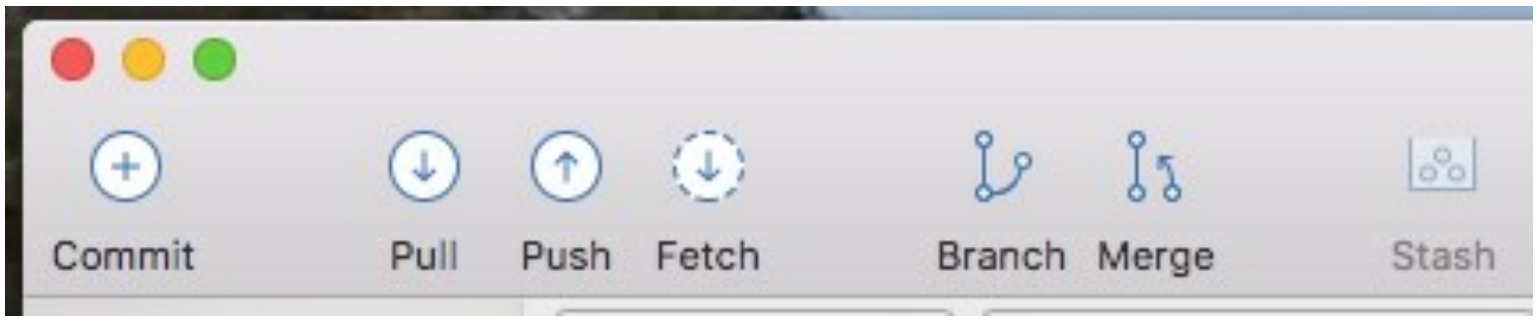
## Ventana principal

La ventana principal contiene todo lo relacionado con el repositorio.




Veamos qué opciones tenemos.

# Acciones sobre el repositorio



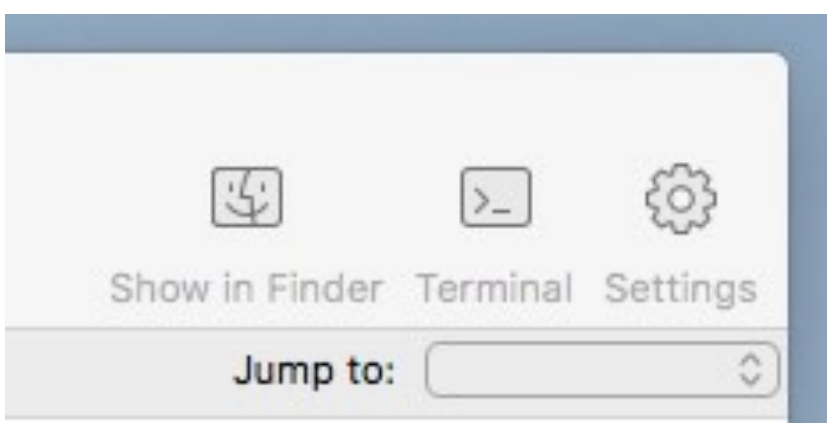
Un primer grupo de botones permiten hacer las operaciones comunes sobre el repositorio: registro de cambios (commit), recuperar los cambios actualizados desde un repositorio remoto (pull y fetch) o remitir los nuevos cambios locales a dicho repositorio remoto (push). La diferencia entre pull y fetch, es que fetch recupera los cambios remotos pero no los fusiona con el repositorio local, mientras que pull hará precisamente eso con la rama de trabajo actual.

También podemos crear ramas (branch) y fusionar ramas existentes (merge).

El último botón que aparece (stash) sirve para guardar provisionalmente un conjunto de cambios sin aplicarlos al historial de cambios. Es posible recuperar esos cambios provisionales más tarde.

## Otras acciones

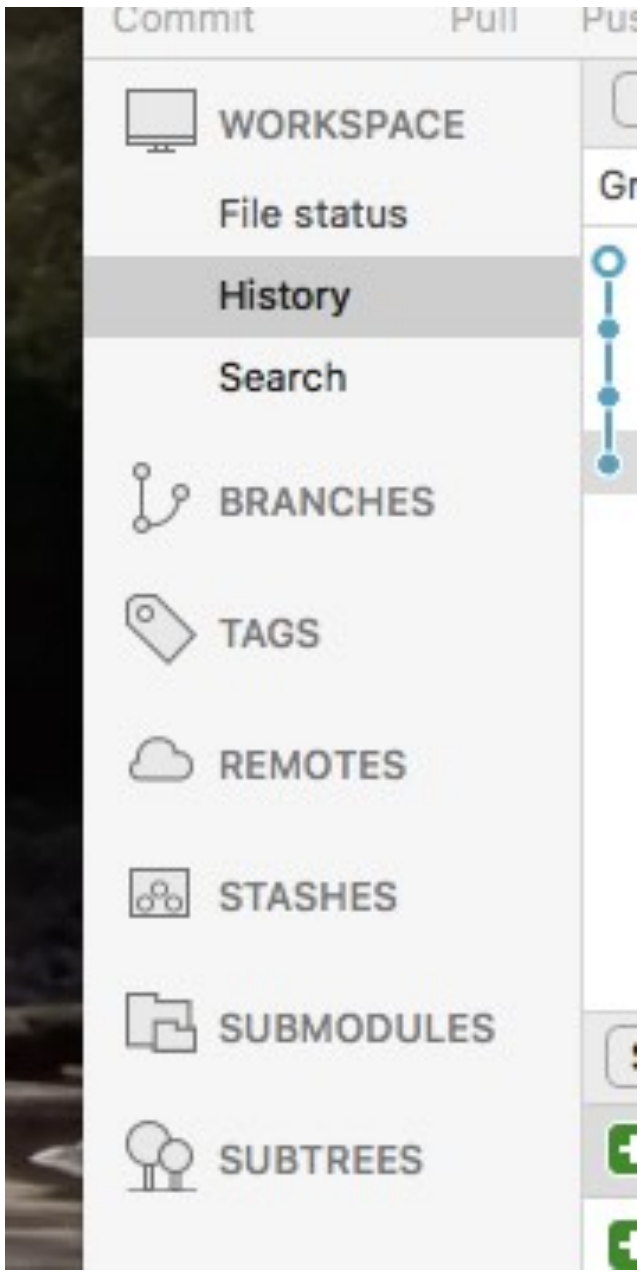
Otro grupo de botones es el siguiente:



El botón “Show in Finder” permite localizar los archivos de la copia de trabajo del repositorio en el Finder, en un ordenador Mac. Un botón parecido veremos en otras plataformas.

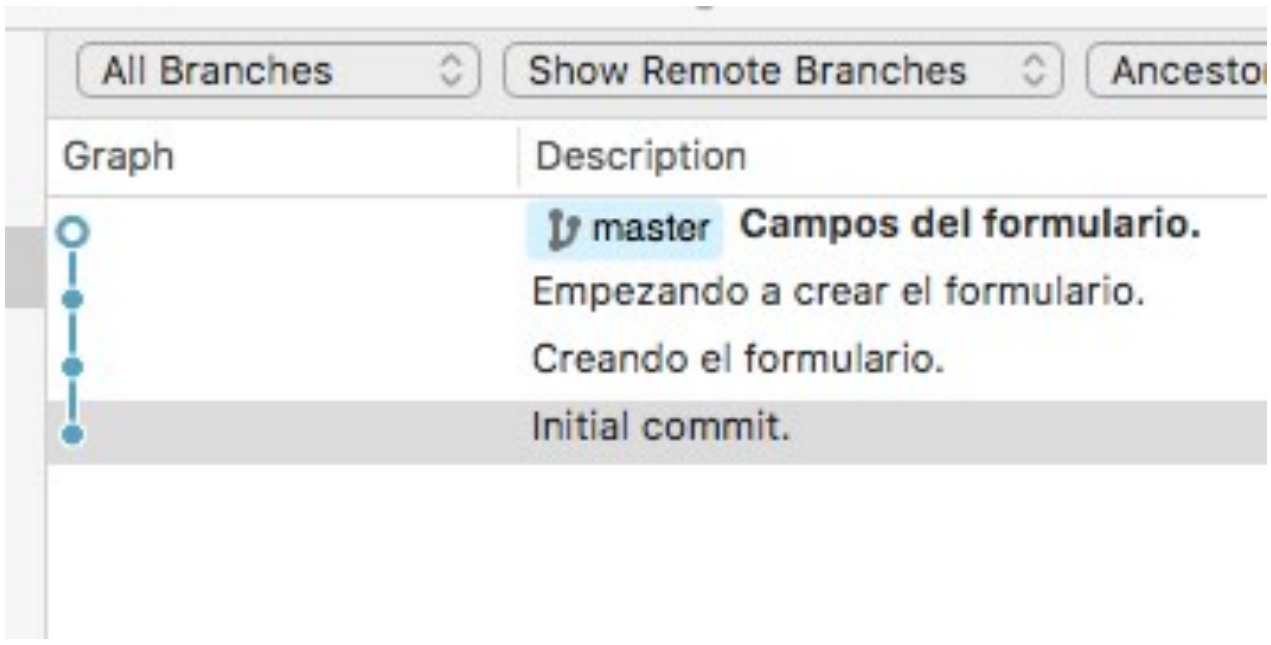
El botón “Terminal” abre una terminal para ejecutar órdenes de línea de comandos, situada en el directorio del repositorio.

El último botón, “Settings”, accede a la configuración del repositorio. Es ahí donde podemos, entre otras cosas, indicar el repositorio remoto.



El panel de la izquierda reune todo lo relativo al repositorio: el estado de la copia de trabajo (copia local), las ramas (branches), las etiquetas (tags), los repositorios remotos asociados (remotes), cambios guardados provisionalmente (stashes), submódulos y subárboles.

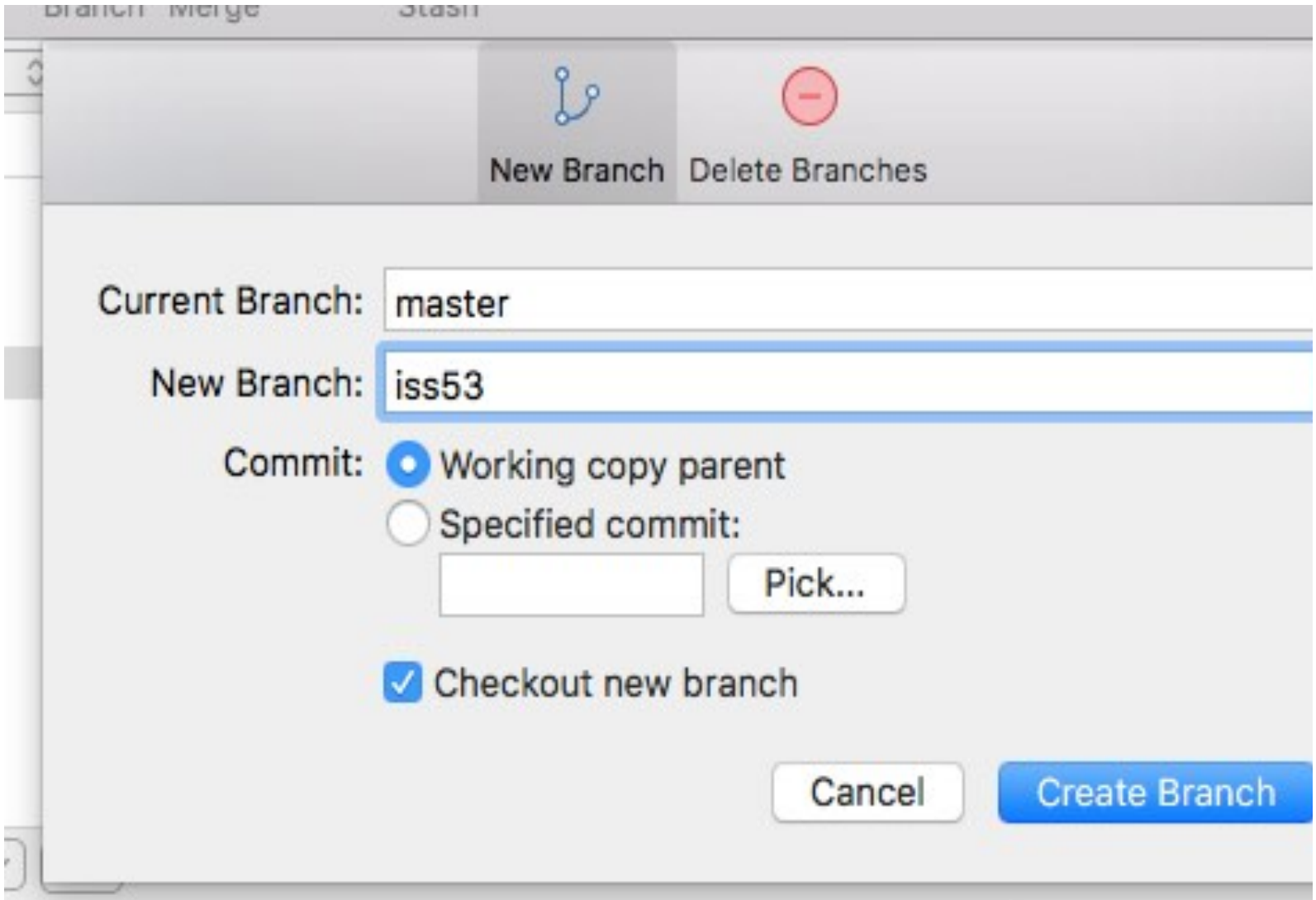
## Acceso al historial de cambios



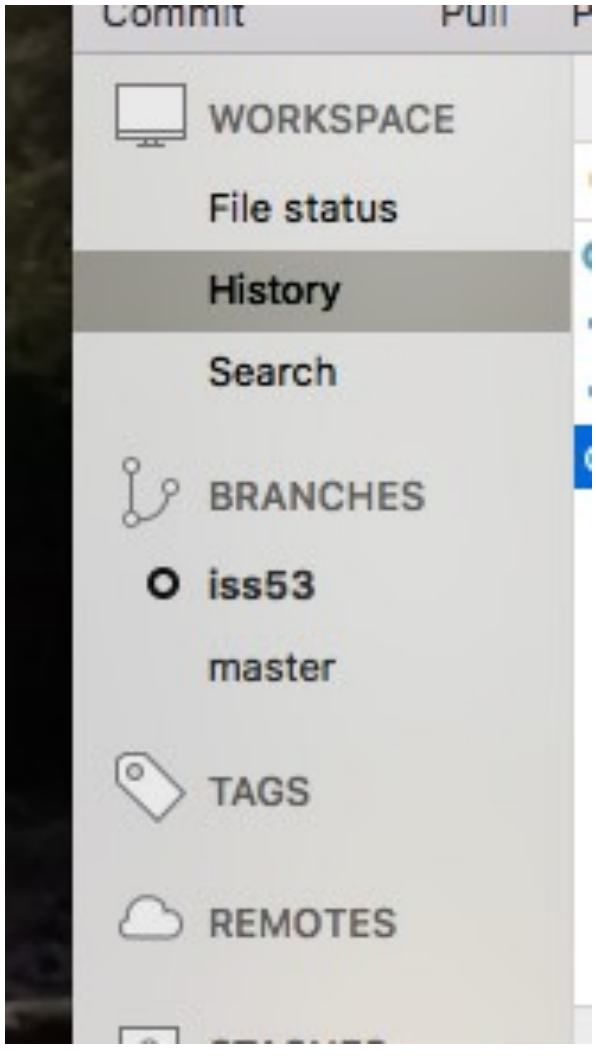

En el área central veremos todo el histórico de cambios con todas sus ramificaciones. SourceTree nos permite cambiar a un commit determinado simplemente haciendo doble click sobre él.

## Creación de ramas

Crear ramas también es muy sencillo en SourceTree. Pulsando el botón “Branch” podemos hacerlo, indicándole el nombre de la rama y a partir de qué rama sale.

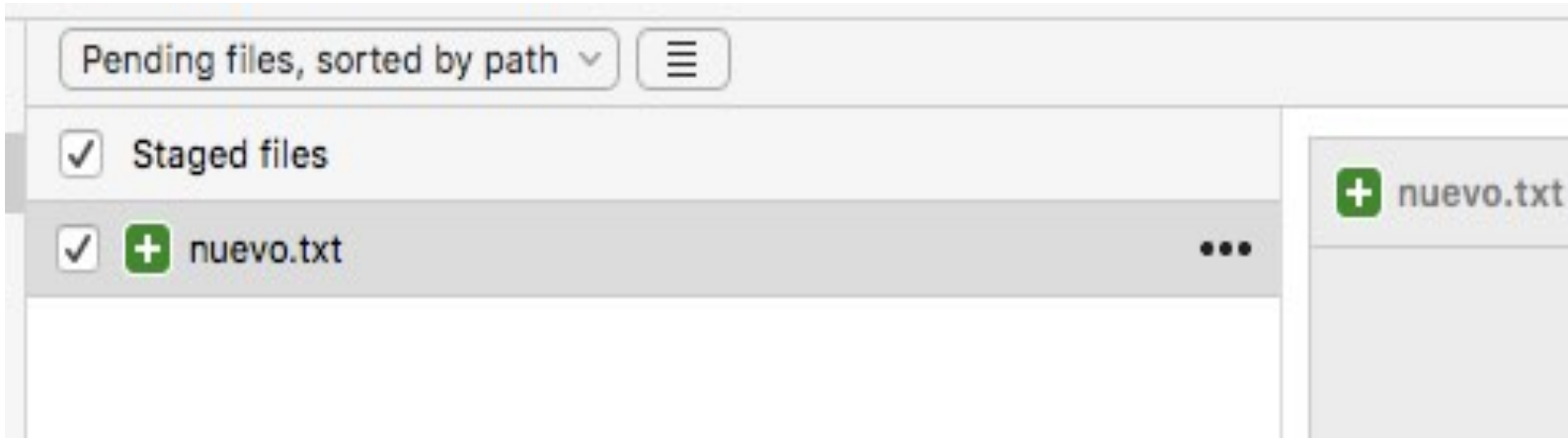



Las ramas creadas aparecen en la sección “Branches” del panel lateral.

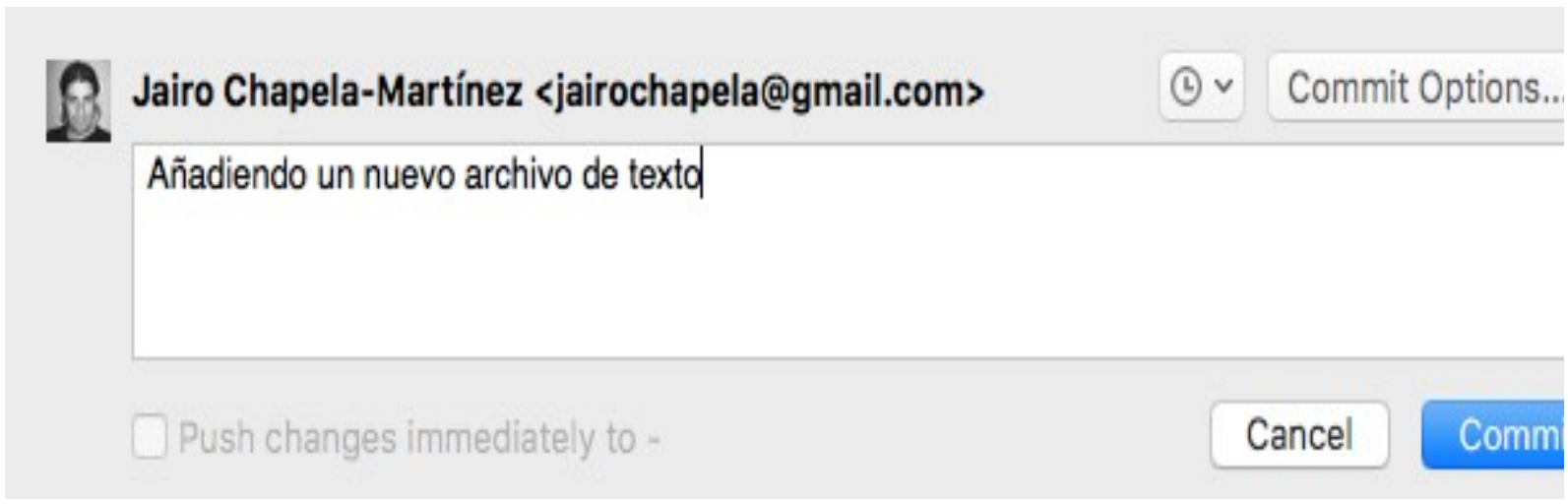



# Registrando cambios

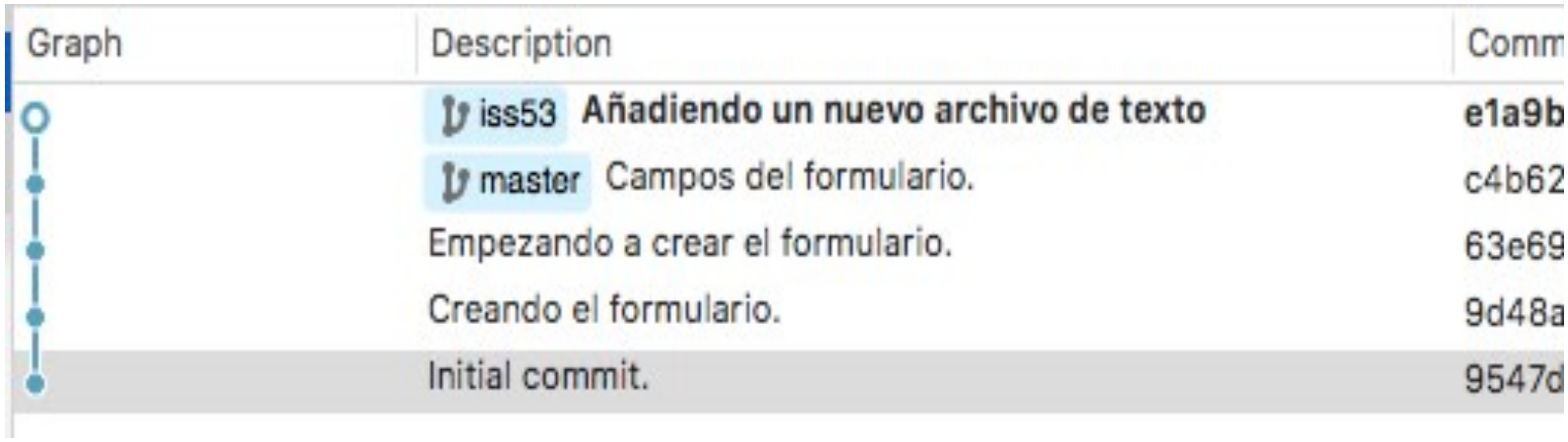
Podemos editar con nuestro editor favorito cualquier archivo del repositorio, e incluso añadir nuevos o borrar existentes. Automáticamente los cambios se reflejarán en el panel inferior de SourceTree. Podemos marcar esos cambios para que sean incluidos en el próximo commit. Esto es equivalente al comando git add.




A la hora de hacer el commit, podemos indicar un comentario acerca de los cambios que vamos a registrar. Pulsando el botón “Commit” haremos efectiva nuestra acción.

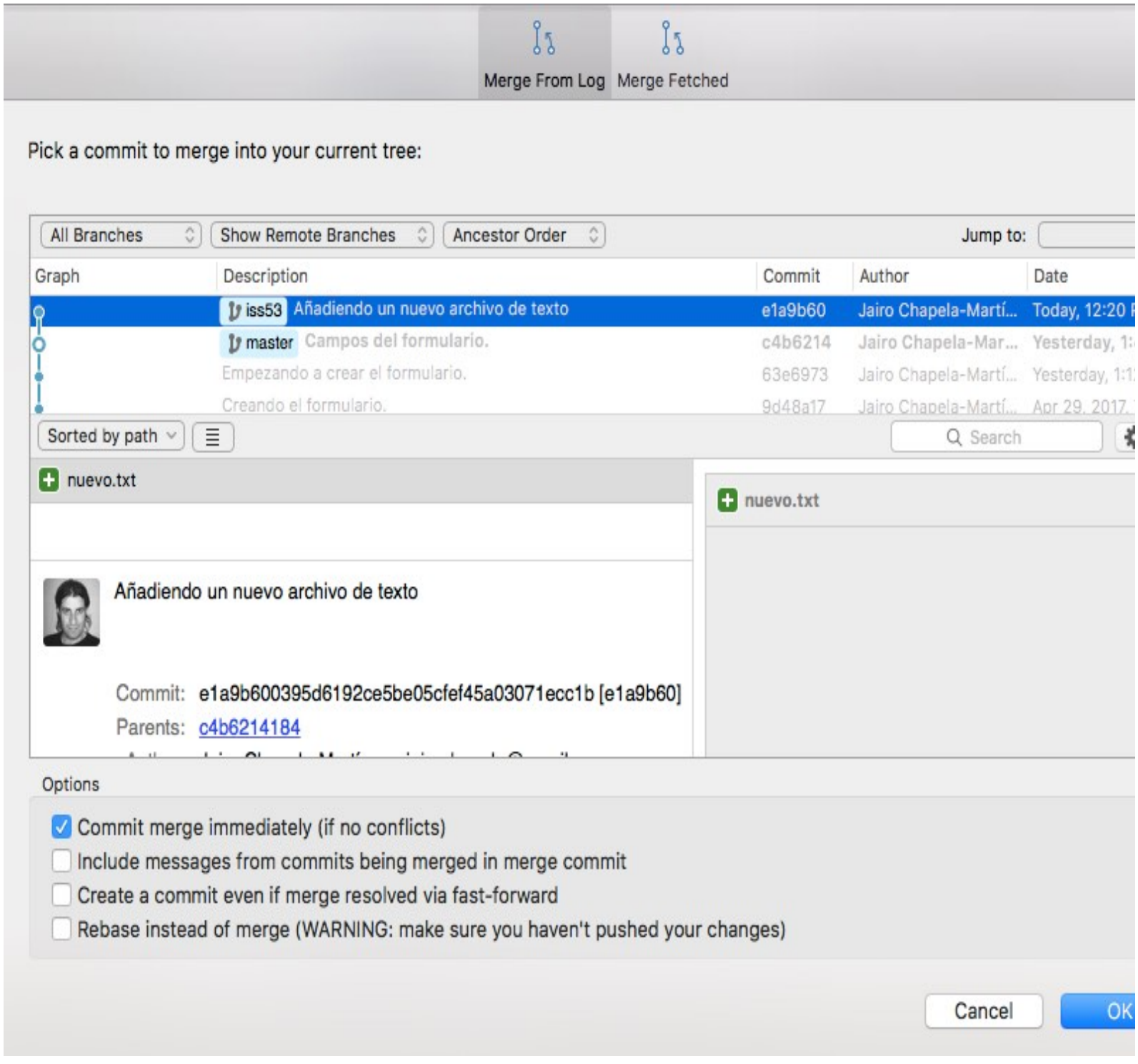



Y los cambios se reflejarán en el historial.




# Fusionar ramas

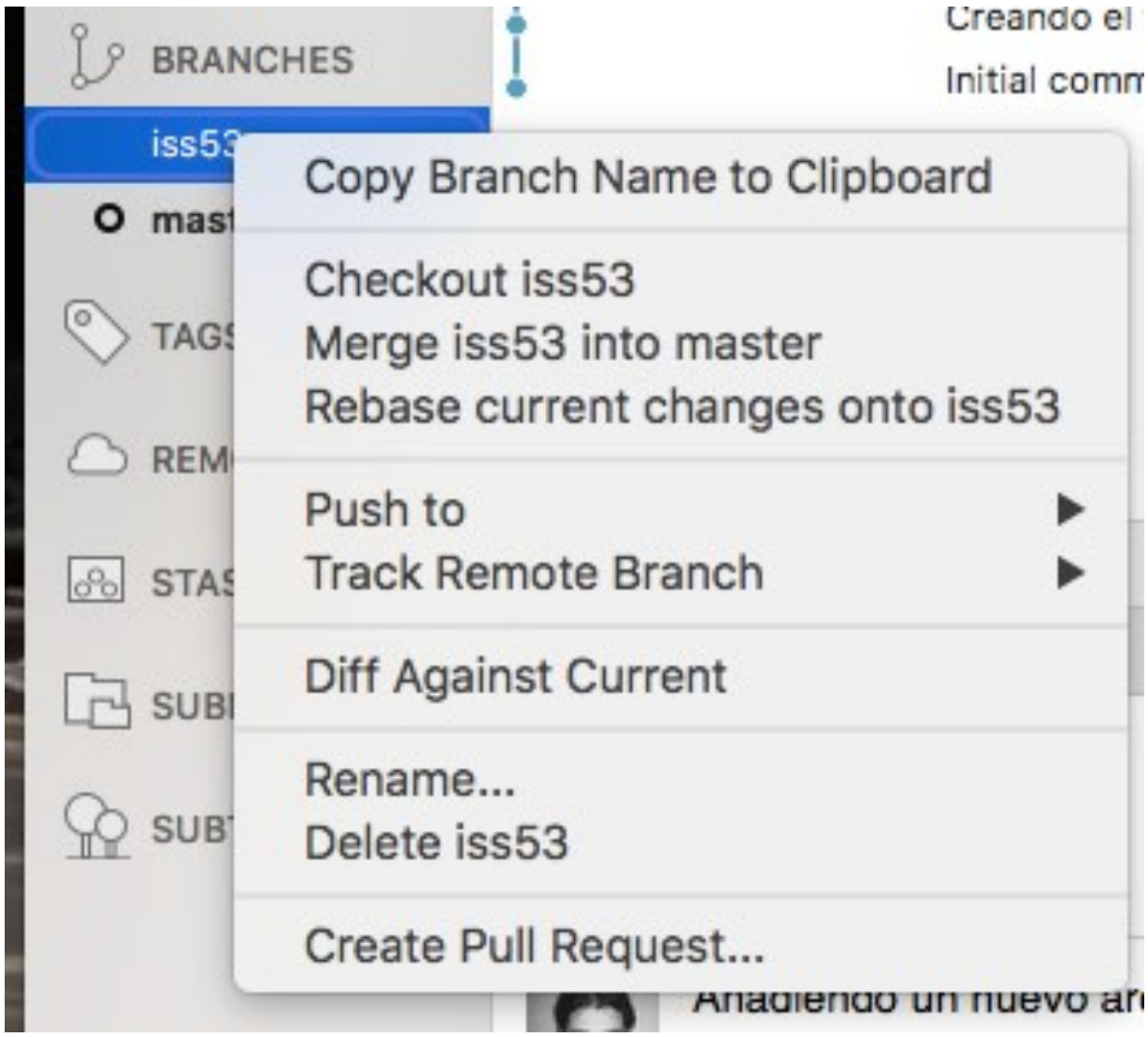
Es posible fusionar ramas en SourceTree, pulsando el botón “Merge”. Nos aparecerá un cuadro de diálogo en el cual seleccionaremos qué rama queremos fusionar, apuntando al commit que queramos.




# Operaciones con las ramas

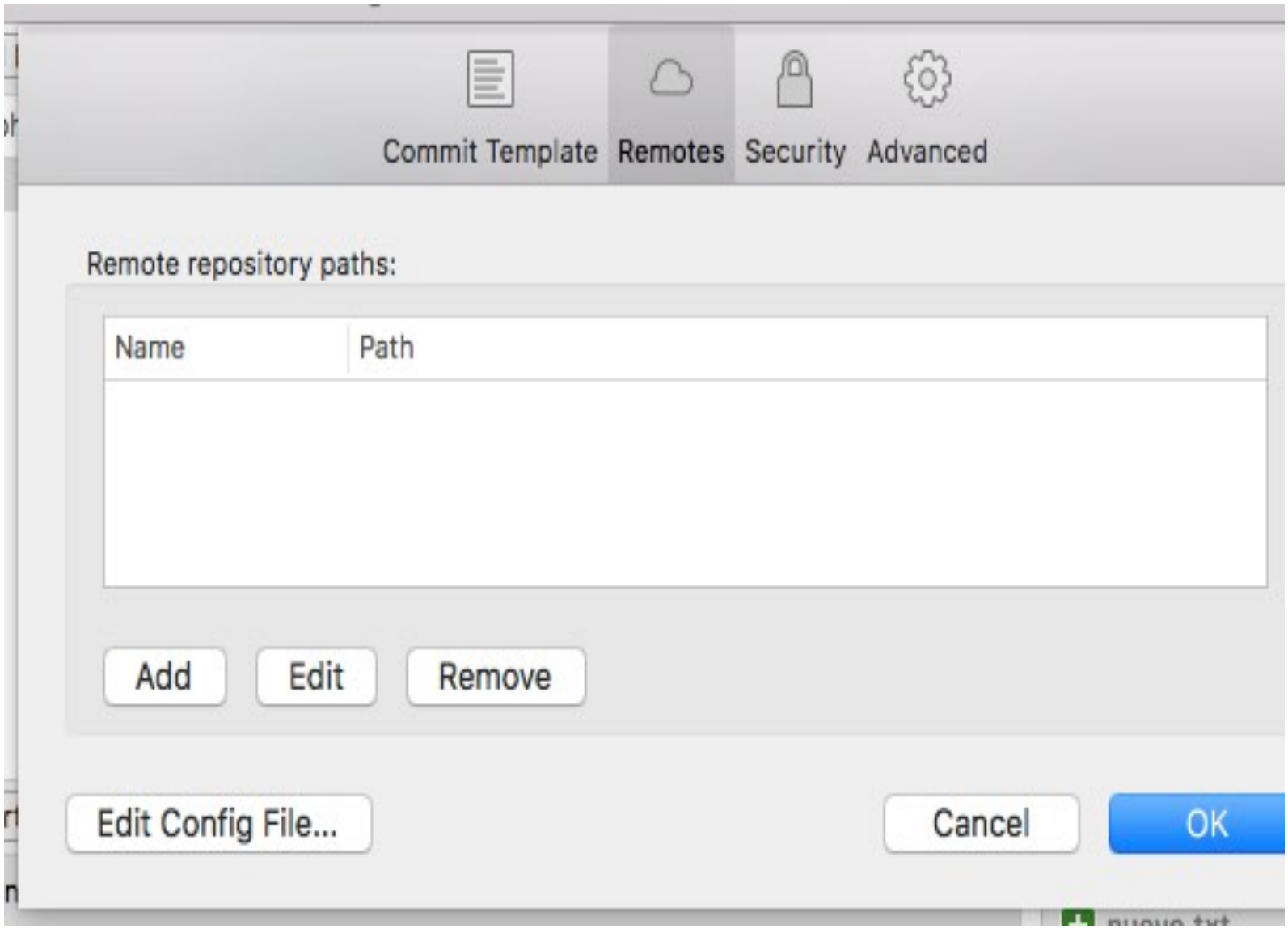
Muchas opciones están disponibles en el menú contextual de las ramas. Haciendo click con el botón secundario sobre

la rama que queramos en el panel lateral aparecerán todas las acciones permitidas.

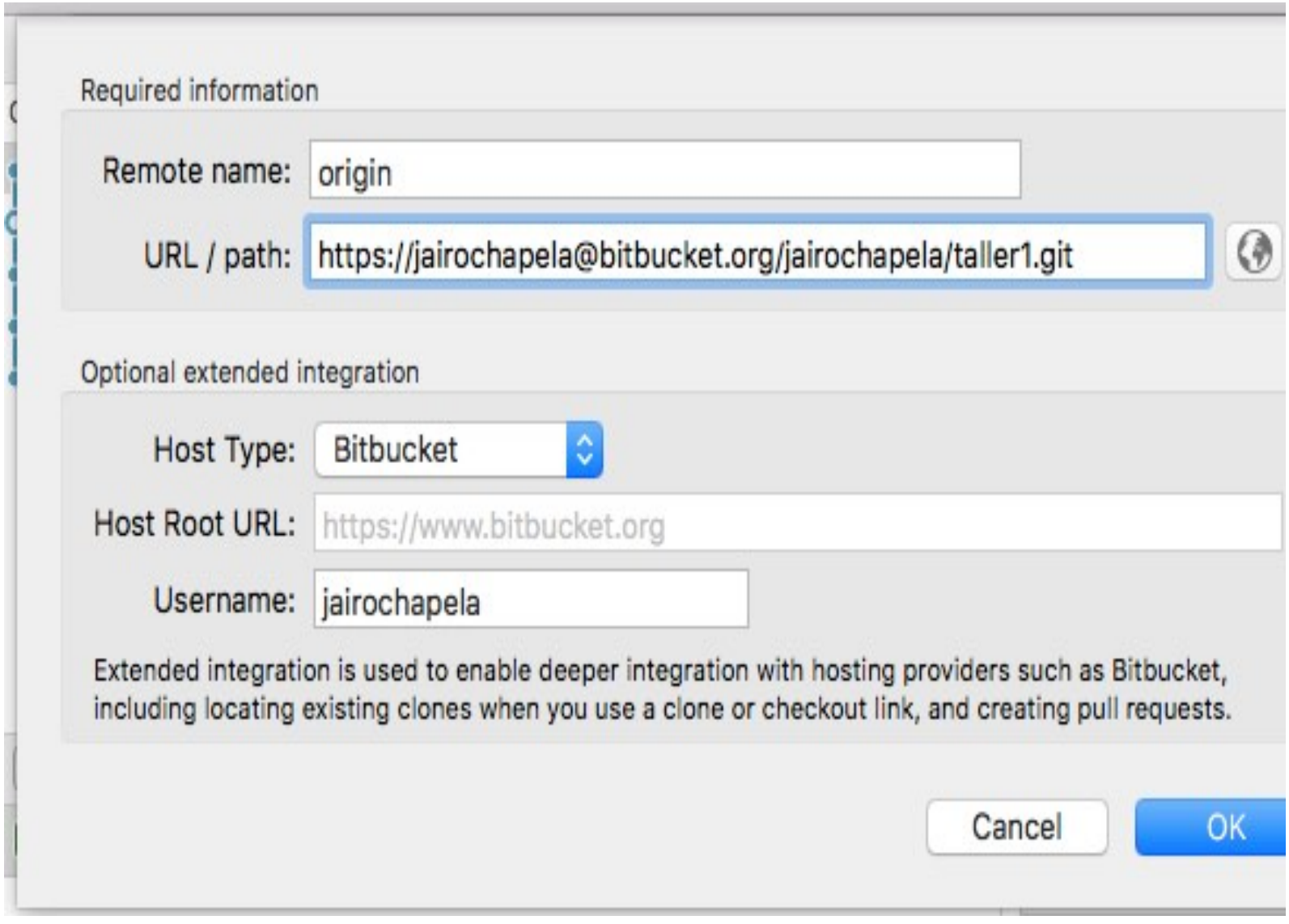



## Ajustes del repositorio

Pulsando el botón “Settings” accedemos a los ajustes del repositorio. El ajuste más significativo es el de añadir un repositorio remoto, alojado en algún servidor o en un servicio en la nube. Este ajuste está en la pestaña “Remotes” y nos permite añadir uno o varios remotos.




Si queremos añadir uno, pulsaremos el botón “Add” e introduciremos los datos de acceso al repositorio.


Required information
Remote name:
origin
URL / path:
https://jairochapela@bitbucket.org/jairochapela/taller1.git
Optional extended integration
Host Type:
Bitbucket
Host Root URL:
https://www.bitbucket.org
Username:
jairochapela
Extended integration is used to enable deeper integration with hosting providers such as Bitbucket, including locating existing clones when you use a clone or checkout link, and creating pull requests.
Cancel
OK

# CONCLUSIÓN

El control de versiones es fundamental para establecer una metodología de trabajo consistente. Los sistemas de control de versiones distribuidos, tales como Git, permiten establecer estos flujos de trabajo a través de un entorno colaborativo del que el repositorio de código es el eje central.

Git es uno de los sistemas de control de versiones más populares en la actualidad y de gran repercusión en el mundo del *open source*. Es además una opción predilecta para las empresas de software, por su capacidad para gestionar repositorios de proyectos de cualquier envergadura.

La interfaz de Git es básicamente por línea de comandos. Algunos editores de código e IDE se integran con Git, facilitando a través de su interfaz el acceso a la mayoría de las acciones que se pueden llevar a cabo con la

herramienta de línea de comandos. Aun así, es recomendable conocer al menos los comandos más básicos para tener la capacidad de resolver cualquier incidencia.

La unidad elemental de un repositorio Git es el *commit*. Un *commit* es un registro de un conjunto de cambios, único e indivisible; es cada uno de los eslabones de una cadena de sucesos que constituyen el historial de cambios. Un *commit* tiene siempre un antecesor, que es otro *commit*, a excepción del que inicia el historial de cambios. Al último *commit* efectuado se le considera cabecera o *head*.

Por claridad, es posible etiquetar algunos *commits*. La utilidad es bien clara: marcar algunos registros de cambios como clave o para señalar hitos dentro del historial de cambios. Un uso típico de las etiquetas o *tags* es el marcado de versiones.

También es posible hacer diverger un repositorio Git en ramas, para mantener diversos historiales de cambios que atienden a evoluciones distintas del proyecto. Es habitual crear ramas para desarrollar nuevas funcionalidades de un proyecto que, por estar todavía en fase experimental, no deben figurar todavía en la rama principal o *master*. Más adelante, es posible fusionar una rama en otra; se puede hacer converger cualquiera de esas ramas creadas con otras ya existentes (como la *master*) y

así integrar esos cambios que evolucionaban en paralelo en el desarrollo principal del proyecto.

Por último, cabe destacar que utilizar una buena aplicación visual para gestionar repositorios Git aporta claros beneficios al desarrollador. SourceTree es una de esas herramientas, y hace un gran aporte visual.