Taller de Química Computacional Aplicada: Día 3

Rony J. Letona

9 de octubre de 2014

1. Ejercicios con el Sistema de Control de Revisión - Git

Al comenzar a trabajar en un proyecto en computación, generalmente se trabaja con archivos sencillos de texto. Estos tienen el código que escribimos y al compilarlos/interpretarlos y ejecutarlos, resultan en programas. Todo funciona así. A veces el código en los archivos es comprensible, a veces no lo es porque no es para que lo entendamos nosotros sino el ordenador. Sin embargo, a todos nos ha pasado que borramos un archivo importante o que cambiamos algo que no debíamos de cambiar y no hay manera de recuperarlo. Desde documentos como reportes de laboratorio hasta cartas, siempre nos pasa en alguna ocasión que perdemos información importante. Para ello se diseñó una alternativa.

Git es la solución más eficiente para ello. Es un sistema en el que cada revisión o versión de un archivo se va guardando gradualmente, se lleva registro de cuáles cambios se hicieron y de cuándo se hicieron. Es el cuaderno de laboratorio digital! No nos salvará una tarea si repentinamente desconectamos el ordenador o se interrumpe la energía eléctrica. Es un sistema para llevar automáticamente registro del trabajo realizado. Lo usan grandes empresas, así como pequeñas iniciativas en sus proyectos. Y no se tiene que limitar a software, puede tratarse de libros, artículos, bases de datos pequeñas y cualquier otra cosa que sufra cambios con el tiempo o requiera ser compartida. Por ello, y por otras razones que veremos más adelante, vamos a hacer algunos ejercicios con el sistema git.

1.1. Configuración de Git

Lo primero que debemos hacer con una nueva pieza de software es configurarla. Por ello, vamos a aprender sobre algunas de las cosas importantes que debemos tener para comenzar. Es importante que nuestro trabajo tenga nuestro nombre y que dé una dirección para contactarnos en caso de que alguien desee hacerlo. Además de eso, es importante que git nos pueda ofecer un editor de texto para hacer cambios en los registros de cada revisión. Comencemos pues con el código.

Abre una línea de comando e ingresa lo siguiente. Claro, cambia el nombre por el tuyo y la dirección de email por la tuya también.

```
git config --global user.name "Tu Nombre"
git config --global user.email "tu_direccion_de@email.com"
git config --global color.ui "auto"
git config --global core.editor "nano"
```

Con esto ya configuramos git; no necesitamos hacer esto nunca en nuestro ordenador. Revisando rápidamente lo que hicimos, porque no vamos a entrar mucho en detalle, debemos notar que cada comando de git se escribe de la siguiente forma `git verbo` . Al principio se anuncia que vamos a trabajar con git, luego el *verbo* especifica lo que vamos a hacer. Lo demás ya son parámetros. Ahora procederemos a practicar un poco con git.

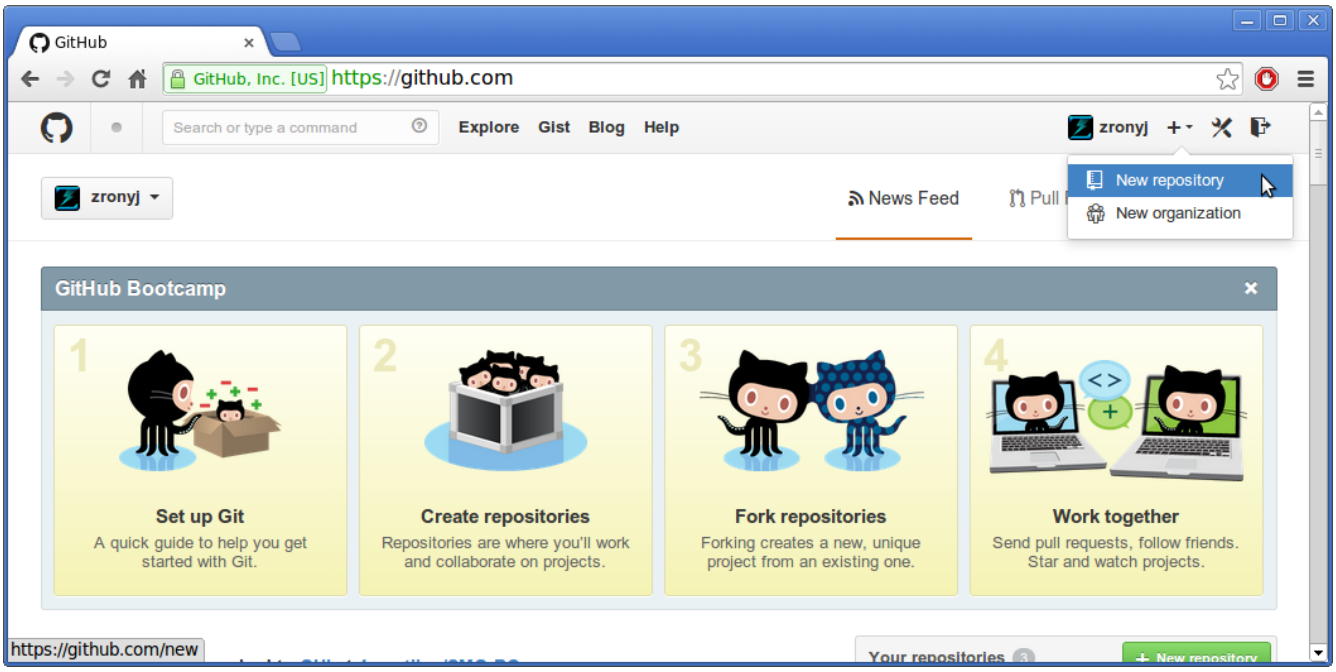
1.2. Creando y Clonando un Repositorio

Como cualquier nuevo proyecto en un laboratorio, vamos a inaugurarlo con un cuaderno nuevo: vamos a crear un repositorio nuevo localmente. Para ello crearemos un nuevo directorio y luego iniciaremos git dentro de él.

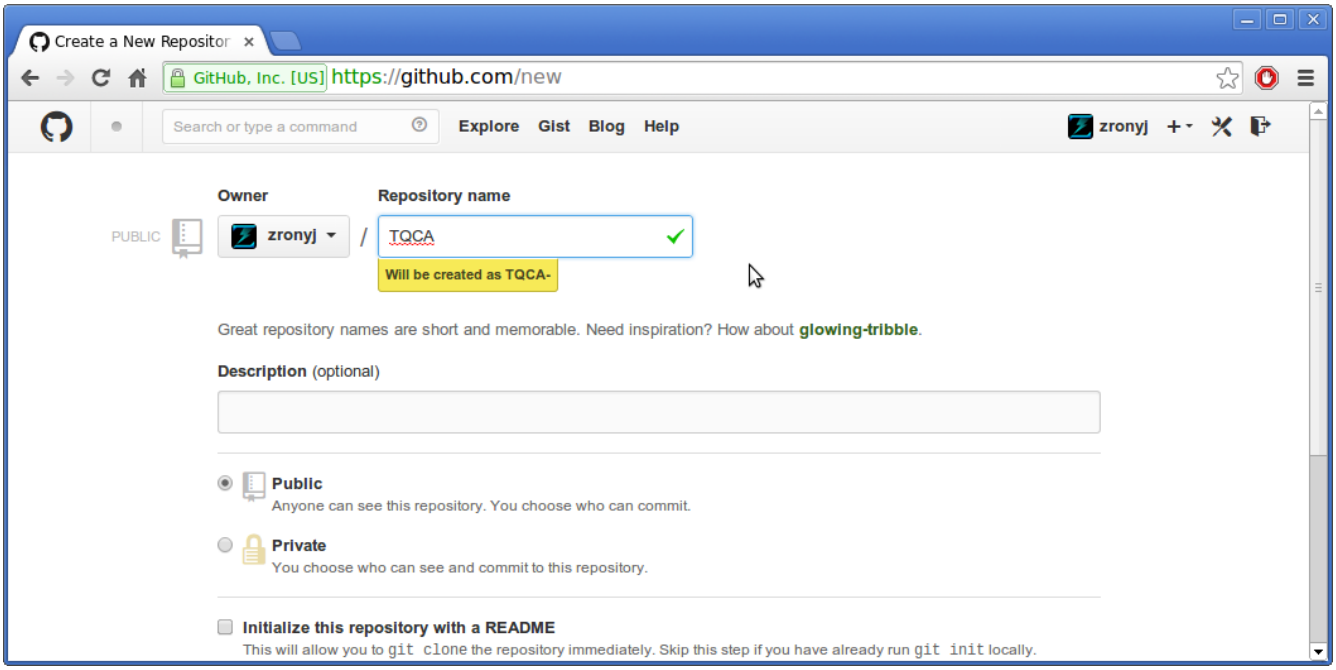
```
mkdir "Mi Proyecto"
cd "Mi Proyecto"
git init
```

Si observamos el contenido del directorio con `ls` , notaremos que no hay nada aparentemente en él. Sin embargo, si ingresamos el comando `ls -a` para mostrar *todos* los contenidos en el directorio (hasta los ocultos), nos daremos cuenta de que existe ahora un directorio `.git` en donde se irá guardando el registro de cada revisión de nuestro trabajo.

Otra forma de hacer esto, y de hecho la más cómoda, es crear un repositorio en GitHub y luego clonarlo a nuestro ordenador. Antes de comenzar con esta parte, asegúrate de tener una cuenta en GitHub. Luego, vamos a crear un nuevo repositorio presionando el botón de "New repository" que sale al presionar el + en la esquina superior derecha.



Luego vamos a crear un nuevo proyecto. Este va a contener todos los archivos y pequeños ejercicios que vamos a ir realizando durante todo el taller. Es por ello que lo vamos a nombrar **TQCA-ej**. Claro que podemos ponerle cualquier nombre, pero en este caso caso solo buscamos un lugar para llevar registro de todo el taller. Inmediatamente después hay un espacio para colocar la descripción del proyecto. Coloca allí lo que desees. Luego nos damos cuenta de que podemos tenerlo como público (a la vista del mundo) o privado (tenerlo privado implica el cobro de una mensualidad). Imagino que vamos a tenerlo público. A continuación se nos ofrece la posibilidad de incluir un archivo README. Este es una forma de instructivo sobre qué es y cómo funciona el proyecto. Además, permite que el proyecto sea clonado. Por ello, vamos a activar la casilla. Finalmente se puede activar el tener un archivo `.gitignore` , el cual no vamos a tener ahorita, y una licencia. El tema de las licencias lo tocaremos más adelante en el taller, por lo que esto también lo podemos dejar sin crear.



Finalmente ingresamos a nuestro nuevo repositorio y vemos que en él se hallan varias cosas como el nombre, los archivos en él, la descripción y una serie de opciones al lado derecho. A nosotros ahora nos interesa la opción que dice ***HTTPS*** *clone URL* en la esquina inferior derecha. Vamos a hacer click en el botón al final del campo de texto. Ahora, después de nuestra vuelta por GitHub, vamos a iniciar el proyecto en nuestro ordenador. Para ello vamos a clonar el proyecto de GitHub de la siguiente manera: en la línea de comando escribe `git clone` y pega la dirección que acabas de copiar. Al presionar Enter, te darás cuenta de que el proyecto se habrá copiado a un nuevo directorio con el nombre del proyecto. Ahora ya puedes trabajar tu proyecto de GitHub en tu ordenador.

1.3. Estado, Diferencias e Ignorados

Entre las operaciones básicas que se pueden hacer *antes* de guardar un registro, están la revisión del estado del mismo, las diferencias halladas en los archivos del proyecto y qué archivos no se deben de tomar en cuenta en el registro de una revisión. Quizá ahorita no sea tan evidente el uso de estas funciones, pero a la hora de ya estar trabajando en un proyecto y de tener que llevar registro de varias revisiones por varias personas trabajando en el mismo proyecto, estas funciones comienzan a tornarse muy útiles en la manipulación de git.

1.3.1. status

Para comenzar con esta parte, sería conveniente que copiaras todos los archivos que has producido anteriormente en el directorio del proyecto. Así podremos demostrar el uso de diferentes técnicas en git que nos permitirán trabajar mejor con las revisiones. Ahora, después de haberlos copiado, nos disponemos a revisar qué dice git de estos cambios. Para ello ingresamos en la línea de comando `git status` .

Lo que git nos dice ahora es que hay nuevos archivos que no han sido agregados al registro de esa revisión. Esto significa que sí son parte del proyecto, pero git no sabe si debe de registrarlos y *seguirlos*. Inmediatamente se nos dice también cómo se agregan estos a esta revisión: `git add nombre_del_archivo.ext` . Si deseáramos agregar más de un archivo, podemos solo agregar el nombre del archivo después del primero así: `git add archivo1.txt archivo2.txt archivo3.txt` Ahora vamos a volver a correr `git status` para revisar qué ha pasado. Nos topamos con que los arhivos que hemos agregado ya están listos para ser registrados.

Finalmente, vamos a guardar este registro. Por ahora solo lo haremos sin mayor explicación; más adelante comprenderemos en su totalidad qué es lo que hace el siguiente comando. Ingresa en la línea de comando lo siguiente: `git commit -m "Probando agregar archivos"` y listo. Con esto hemos creado el registro de nuestra primera revisión en git con los archivos que hemos agregado.

1.3.2. diff

Una cosa es agregar y quitar archivos. Otra cosa es editarlos. Vamos a ingresar a alguno de nuestros archivos y vamos a cambiar algo pequeño en él. Busca algún dato que puedas cambiar, sin miedo, y cámbialo. Ahora, vamos a volver a repetir lo del ejercicio anterior: `git status` Como era de esperarse, se nos indica que uno de nuestros archivos ha sido modificado. Ahora, veremos en qué ha cambiado. Para eso corremos el nuevo comando: `git diff`

Esto es más interesante. Se nos dice qué líneas han cambiado en nuestro archivo. De hecho, se nos indica con un signo − el estado anterior de esa línea, y con un signo + el nuevo estado de la misma. Podemos ir viendo, entonces, qué cambios hemos hecho y dónde. También se nos dice en qué archivo, si nos fijamos en las primeras líneas de resultado. Si deseamos dejar esos cambios, agregamos el archivo con *add*, como habíamos visto antes. Si no, lo ignoramos.

1.3.3. .gitignore

Al crear programas nuevos es necesario ir probándolos para asegurarse de que todo funcione bien. Cada vez que un programa se traduce al lenguaje de la compu y se "ensambla", se crean varios archivos auxiliares o de soporte, además del programa en versión ya terminada. Claro, esto es importante a la hora de hacer pruebas, pero generalmente no queremos agregarlo a nuestra revisión, ya que lo importante para nosotros es el código fuente: lo que nosotros hemos escrito. Para ello, git nos ofrece un pequeño truco: `.gitignore`

`.gitignore` es un archivo "oculto", como el directorio de git en nuestro directorio de trabajo, que nos ofrece la habilidad de ignorar ciertos archivos a la hora de registrar un estado del proyecto. Cómo así? Cada vez que creamos el registro de una nueva revisión de nuestro proyecto, podemos ignorar algunos archivos (e.g. archivos auxiliares o programas semi-terminados) con el fin de solo ir guardando nuestro código fuente. Hagamos un ejercicio con esto.

En nuestro directorio del proyecto vamos a crear un archivo nuevo y escribir algo en él, cualquier cosa. Luego, en la línea de comando, abriremos el archivo con `nano` de la siguiente forma: `nano .gitignore` . Allí vamos a agregar el nombre de nuestro nuevo archivo, precedido por una diagonal: `/mi_archivo.txt` . Cerramos, guardamos y expliramos el estado de nuestro proyecto con `git status` . Inmadiatamente nos damos cuenta de que nuestro archivo no ha sido tomado en cuenta, pero `.gitignore` si. Agregamos este último al registro de nuestra nueva revisión y corremos `git commit -m "Comenzando con .gitignore"` para guardarlo.

1.4. Commit, Push, Pull y Log

Una de las cosas más interesantes que se pueden llevar a cabo con un sistema como git, es trabajar en grupo en un proyecto en particular. Esto significa que todos estarán realizando cambios sobre los mismos archivos y creando o eliminando cosas dentro del mismo proyecto. Es una buena práctica que cada vez que alguien hace algún cambio, se guarde y registre esa revisión. Por ello, a continuación veremos cómo hacer operaciones como: registrar cambios, subir los cambios registrados a nuestra cuenta en GitHub, actualizar el proyecto con cambios de alguien más y ver qué cambios se han hecho en general.

1.4.1. commit

Algo que ya hemos hecho con anteriorirdad, aunque no hemos entendido muy bien qué es lo que hace, es registrar cambios o *commitear* cambios (dado que no existe una traducción exacta para este verbo). La acción de este comando es sencilla: está guardando **solo los cambios realizados** en el proyecto. Esto lo hace en el directorio oculto que tiene nuestro proyecto. Sin embargo, esto solo sucede localmente! Los cambios solo se guardan en nuestro ordenador. Para que estos pasen a GitHub, veremos otro comando más. La ventaja es que antes de empujar cambios a GitHub, podemos hacer cualquier cantidad de commits. Veamos qué se requiere y cómo se hace esto.

El comando es relativamente sencillo cuando uno lo ve: `git commit` . Solo eso? Sí, en esencia, sí. Si queremos entender qué pasará después, debemos entender algunas cosas sobre este comando. Cada vez que registramos un cambio nuevo, git nos pide que agreguemos un mensaje. La idea es explicar a grandes rasgos qué se hizo, cómo y por qué. Hay dos formas de incluir este mensaje:

1. Esperando que después de correr `git commit` , git inicie `nano` en donde escribiremos el mensaje.

2. Agregando, después de `git commit` , la bandera `-m` y nuestro mensaje entre comillas.

Ahorita no hemos hecho ningún cambio en nuestros datos, así que, intenta editar y cambiar el archivo que habías editado en la sección de **diff** y vamos a realizar algún otro cambio que podamos registrar. Cuando ya todo esté listo, vamos a correr `git commit` , vamos a ingresar un mensaje en `nano` y vamos a finalizar cerrando y guardando. Comenta, qué ventajas y desventajas hay de usar la primera o la segunda forma?

1.4.2. push

Y finalmente llegamos al momento en el que todo se une. Todos los commits que hemos estado realizando, los hemos hecho localmente. Nos toca entonces empujar todas estas revisiones a GitHub. Para ello vamos a ingresar el siguiente comando: `git push` y lo vamos a correr. Si no había cambios pendientes de registrar, se nos pedirá un usuario y una clave. Estos son los de GitHub. Finalmente, todos los cambios se guardarán en la nube y quedarán disponibles para colaboradores o usuarios de datos o el software en el que estamos trabajando.

1.4.3. pull

Ya que hemos aprendido a registrar cambios y empujar nuestras versiones a la nube, vamos a aprender cómo descargar esto para actualizar el proyecto en el que estamos trabajando, si que es este ha sufrido cambios por parte de alguien más. Si se está trabajando en un proyecto conjunto con otras personas, es muy común que haya archivos sujetos a cambio constante por varios integrantes del equipo de desarrollo y por eso, necesitas saber qué es diferente antes de empezar a trabajar.

Por eso, si estamos trabajando en un ambiente con más personas, siempre debemos de actualizar nuestras revisiones primero, i.e. hacer un *pull* o jalar cambios. Para esto, el comando es muy sencillo: `git pull` . Si hace eso ahorita con tus archivos actuales, es muy probable que git solo te diga que todo está actualizado y no hay nada que jalar. Sin embargo, intentemos hacerlo, es un buen ejercicio!

1.4.4. log

Saber qué ha pasado con nuestro proyecto a lo largo del tiempo resulta importante cuando sabemos que no solo nosotros hemos hecho cambios sobre él. También nos sirve para tener una idea del avance del proyecto. Para hacer una revisión de esto, es muy útil un pequeño comando que veremos a continuación: `git log` . Intenta ingresar esto a tu línea de comado y ver qué te arroja como resultado. Por cierto, para salirte del log, presiona *q*.

El *log* es como el índice de nuestro cuaderno de laboratorio. En este se encuentran todos los commits hechos con un número particular: su identificador o *id*. La identificación es importante! Cada commit tiene un *id* que lo identifica, permitiéndole a GitHub o a nuestro ordenador saber cuál es cuál. Más adelante veremos para qué nos puede ser útil.

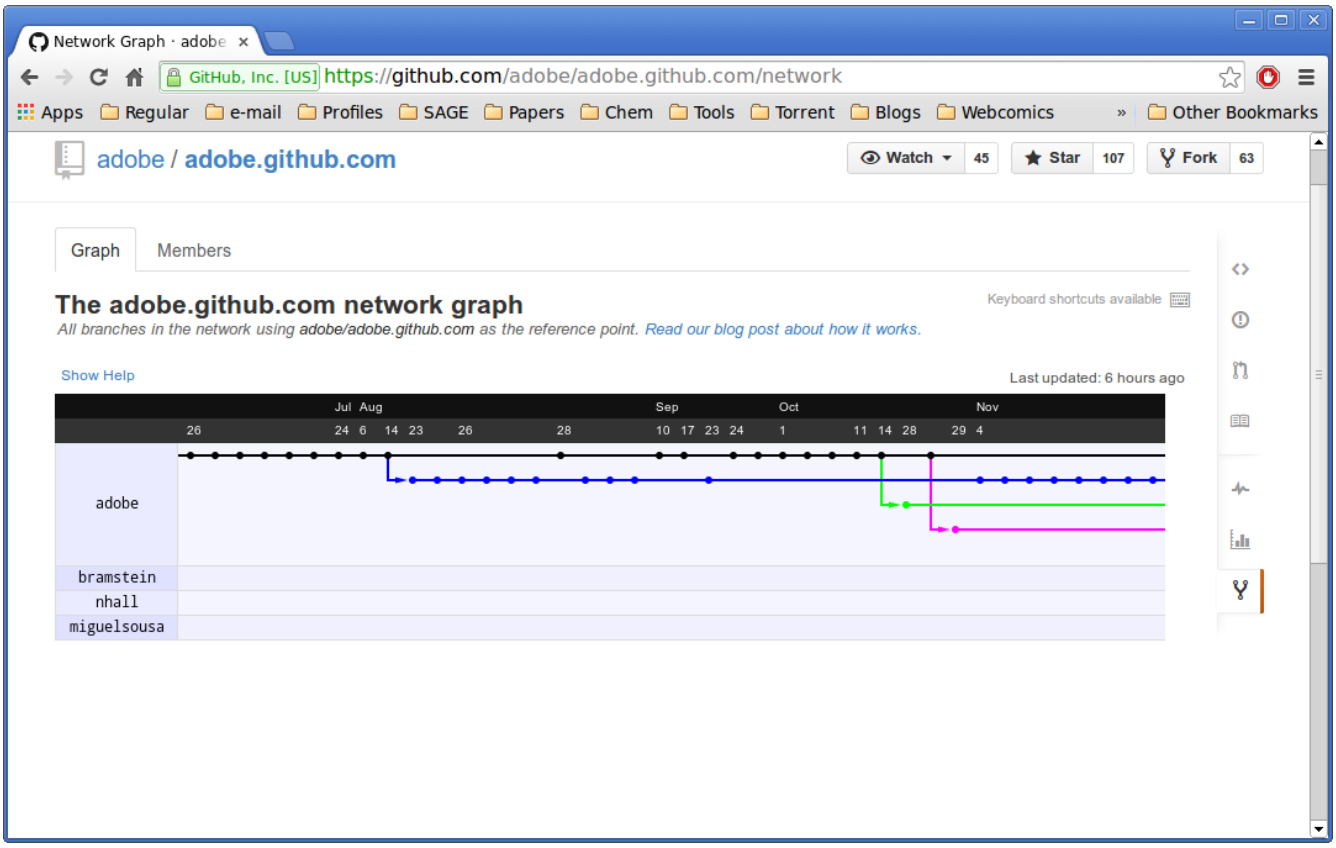
1.5. Ramas, Merge y Fork

Hasta ahora hemos revisado nuestro proyecto por cambios, hemos registrado los cambios de una revisión, hemos empujado esa información, descargado actualizaciones y revisado el índice del proyecto. Git, sin embargo, permite hacer algunas cosas mucho más interesantes. Una de ellas es la capacidad de trabajar en una versión del proyecto independientemente del resto de él. Otra es integrar esos cambios independientes al proyecto principal. Y finalmente, otra sería copiar un proyecto entero para modificarlo, sin la necesidad de descargar todo y crear algo nuevo en GitHub. Revisemos cada una de estas opciones.

1.5.1. branch

Deseamos trabajar en un script de nuestro proyecto, pero trabajar en él implica hacer que funcione de otra forma por completo. Eso implica hacer cambios que afectan grandemente al proyecto, porque lo más probable es que algunas personas del proyecto ya dependan de que ese script funcione. Cómo hacemos eso conservando la habilidad de ir guardando registro de revisiones? Pues, se crea una rama. Una rama es como otro camino que seguiremos desde una revisión específica. Cómo así? Partiendo de un commit en particular, vamos a seguir por *otro* camino que tendrá el desarrollo de una característica en particular. Este desarrollo se estará llevando a cabo en

paralelo al desarrollo principal del proyecto, pero no lo tocará. Esto es una rama, o `branch` . La representación gráfica de esto se ve algo así.

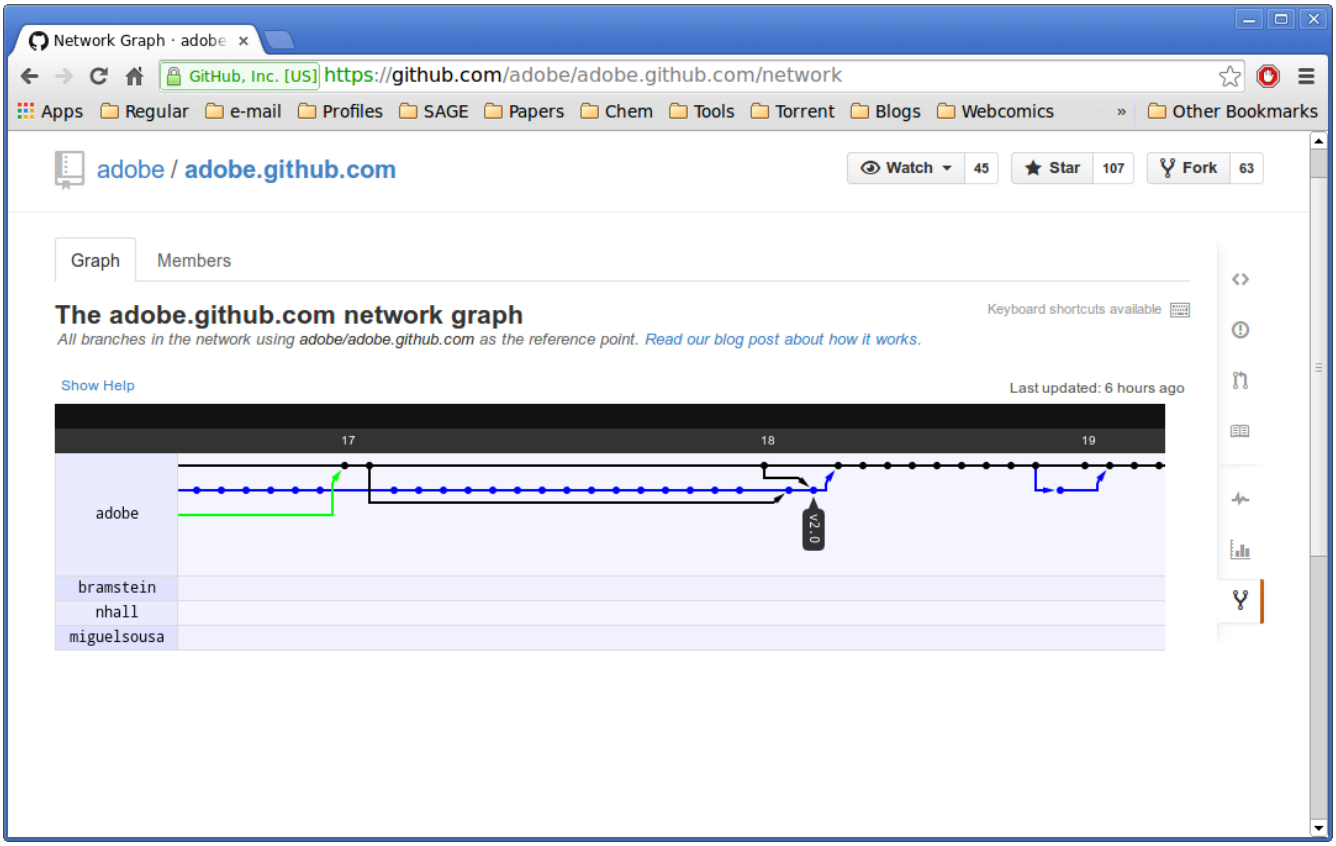


Aquí podemos ver que cada pequeño círculo en las líneas son commits, y notamos cómo es que 3 ramas se desprenden del proyecto principal (la línea negra). La idea entonces es poder desarrollar una característica en particular sin necesidad de arruinar el resto del proyecto. Lo primero que intentaremos será revisar qué ramas existen y luego crearemos una nueva.

Para revisar qué ramas existen, el comando es sencillo: `git branch` . Esto nos mostrará la rama `*master` , que por el momento es la única que existe. Ahora intentaremos crear una nueva rama y nos cambiarnos a ella. Para ello, vamos a ingresar lo siguiente en nuestra línea de comando: `git branch` *`nombre_rama`* . Esto creará la rama nueva. Para pasarnos a ella, ingresamos: `git checkout` *`nombre_rama`* . De esa forma no solo creamos la rama, sino nos situamos en ella. Para fines de practicidada, estos dos comandos se pueden fusionar en uno solo: `git checkout -b` *`nombre_rama`* . Después de haber hecho esto, piensa: Cómo te regresas a la rama principal?

1.5.2. merge

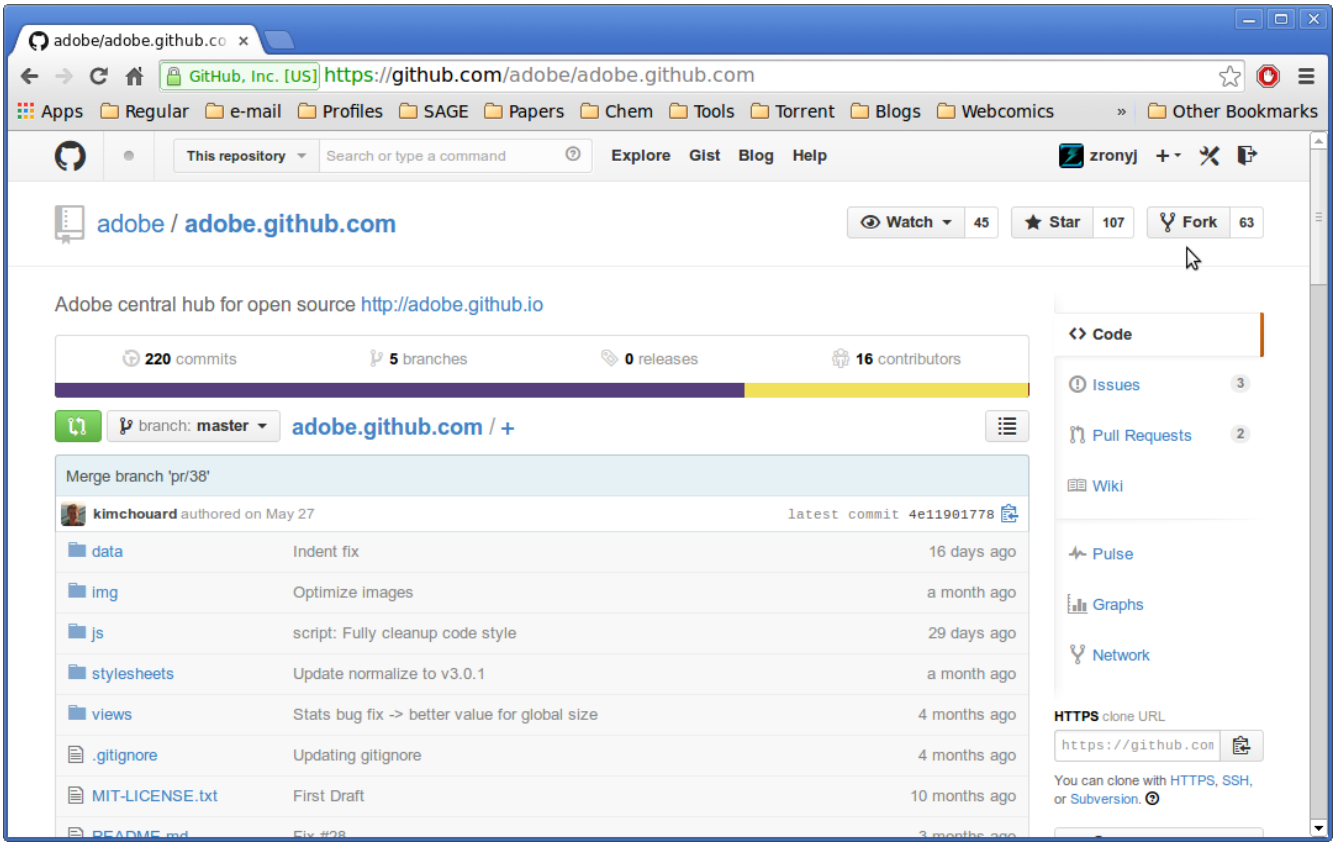
Y qué pasa cuando terminamos de modificar lo que queríamos, nuestro nuevo código funciona y lo queremos incluir en la rama principal de nuestro proyecto? En este caso, debemos de hacer que nuestra rama se una al proyecto principal. Para git, esto se hace con el comando `merge` . La idea es colocarnos en una rama particular y decirle a git que esta debe de unirse a otra rama que especificamos. La representación gráfica de esto se puede notar al observar cómo las ramas de colores verde, azul y algunas líneas negras se unen a la línea negra principal con una pequeña flecha.



Así que tomando eso en cuenta, solo parece lógico que el comando completo sea algo así `git merge master` .Con esto último uniríamos nuestra nueva rama a la rama principal. Debemos de tener cuidado, pues es común que el código sobre el que estabamos trabajando vaya a haber sido modificado en el tiempo en que nosotros trabajamos en la rama. En ese caso, git nos advertirá que eso pasó y nos dará la opción de escoger una opción. Eso, sin embargo, lo veremos más adelante.

1.5.3. fork

Se da a veces el caso en el que algún proyecto nos parece muy interesante. Por ejemplo, Linux es un proyecto que se trabaja con git. Algunas personas lo hallan muy interesante y desearían crear una versión personalizada y específica de Linux para alguna necesidad. Cómo hacer esto? Hacer una rama del proyecto principal de Linux no parece una buena idea; las ramas sirven para contribuir al proyecto. No, nosotros deseamos algo independiente. Para ello se dice que se hace un `fork` (así como tenedor en inglés). Y pues, la idea es hacer una replica exacta del repositorio de Linux dentro de nuestra cuenta personal de git. Esto nos ahorrará descargar el proyecto entero, crear un repositorio en nuestra cuenta, registrarlo con el proyecto que descargamos y volver a empujar todo a la nube. En vez de eso, en GitHub, solo hacemos click al botón del proyecto que deseamos copiar. Nada más.

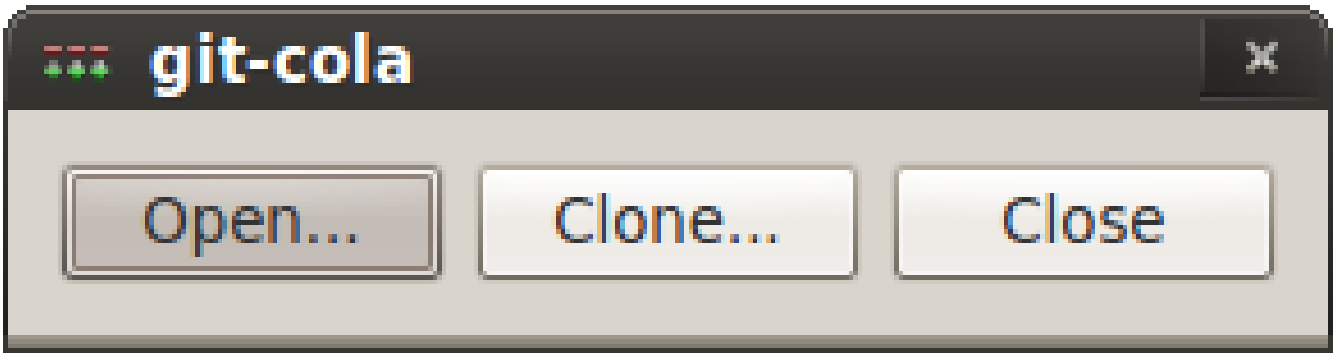


1.6. GUI y Situaciones Comunes

Aceptémoslo, la línea de comando nos permite hacer muchas cosas, pero tenemos que memorizar muchas palabras y no resulta nada práctica muchas veces. Por qué no existen soluciones más gráficas? Con campos de texto, botones, deslizadores, ventanas, etc.? Pues ... la verdad es que sí existen. Facilitan mucho el trabajo y nos ayudan a poder desarrollar más rápidamente. Por qué no las usamos muy seguido? Porque nos limitan mucho; no nos ofrecen muchas de las opciones que hemos aprendido hasta ahora en el taller. Sin embargo, como nuestro objetivo no es ser programadores, sino saber usar un ordenador para hacer química, vamos a dar una breve introducción a *un* ambiente gráfico para git: Cola.

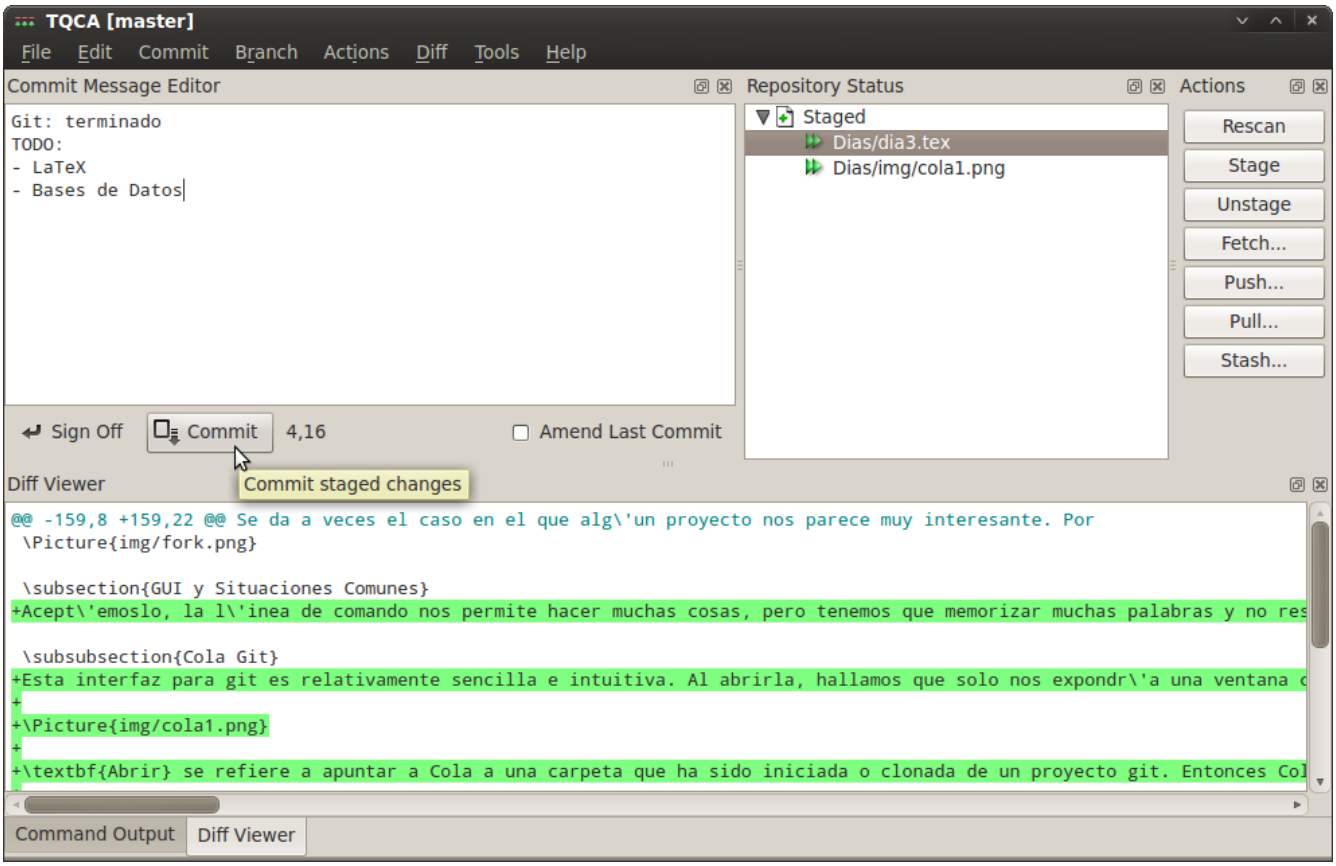
1.6.1. Cola Git

Esta interfaz para git es relativamente sencilla e intuitiva. Al abrirla, hallamos que solo nos expondrá una ventana con 3 botones: **Abrir**, **Clonar** y **Cerrar**.



Abrir se refiere a apuntar a Cola a un directorio que ha sido iniciado o clonado de un proyecto git. Entonces

Cola reconocerá esto y nos dirá en una ventana más grande: qué archivos han cambiado, nos ofrecerá agregarlos al commit, y si hacemos click en uno de ellos, en la parte inferior nos mostrará qué ha cambiado en ellos desde la última vez que se ha hecho commit. Desde acá también se puede realizar commits, se puede jalar y se puede empujar cambios.



Como ejercicio, intenta cambiar algún archivo, realiza un commit y empuja cambios desde Cola.

Clonar se refiere a clonar un repositorio; un proyecto. Cola inmediatamente te pide una dirección web de dónde descargar el proyecto, te pregunta a qué directorio lo deseas descargar y te da la opción de descargarlo a través de él.

Si lo notas, todo lo que puedes hacer con Cola, lo puedes hacer desde la línea de comando. En el caso de Cola, puedes hacerlo quizá más rápido, pero la línea de comando te permite hacer más cambios y con mucho más control.

1.6.2. Deseo ver cómo estaban mis archivos en una revisión particular

No es tan común como las dos siguientes, pero sin embargo, nos pasa. Deseamos ver qué habíamos escrito en determinado punto del proyecto o cómo se veía un archivo en particular porque "entonces funcionaba, y ahora no". Para hacer esto se trabaja de manera muy similar a las ramas, pero con un truco leve. Lo primero que debemos saber es el *id* del commit que queremos revisar. Para eso es útil usar `git log`. Una vez hallado el *id*, entonces procedemos a situarnos en ese punto del proyecto mediante: `git checkout <id-commit>`. De esta forma llegamos a **ESE** momento en particular.

Ahora, resulta útil hacer esto, pero ... y cómo regresamos? Aquí vale la pena mencionar nuevamente el caso de la cabeza del proyecto: `HEAD`. Nosotros podemos navegar entre diferentes puntos del proyecto refiriéndonos solamente a dónde se halla la cabeza o a qué distancia de la cabeza se halla algún commit. Para regresar a la cabeza, el comando es sencillo: `git checkout HEAD`. Sin embargo, mencionamos lo demás, porque para movernos a un commit en particular, a veces no es necesario el *id*. El commit anterior a la cabeza se llama `HEAD~1`, el anterior a ese `HEAD~2`, el anterior a ese `HEAD~3` y así sucesivamente. De esta manera, si deseamos retroceder 5 pasos,

solo debemos de ingresar `HEAD~5` a la línea de comando y *voilá*, hemos llegado!

1.6.3. Hice commit de cambios que no deseo en GitHub

Para este problema, existen dos soluciones. La primera es responsabilizarse por el error y dejarlo allí para la posteridad. La segunda es decirle a git que ignore eso y borre lo que acabas de registrar. Si se dice que la *cabeza* del sistema de revisiones es el último commit en haber sido agregado, esta acción de ignorar lo que acabas de hacer y borrar todo desde cierto punto en adelante se conoce como mover la cabeza hasta X punto (en donde X es el identificador del commit hecho). Hacer esto es **MUY** peligroso y no se recomienda, porque se perderán los cambios de todos a partir del commit escogido como nueva cabeza. Sin embargo, esta es una pregunta común y por eso dejaremos el código para hacerlo: `git reset --hard <id-commit>` Si ya empujaste los cambios que no deseabas a GitHub, hay algunas maneras de eliminar los commits hechos, pero no se cubrirán aquí.

1.6.4. Ambos editamos el mismo archivo en el mismo lugar

Es común, aunque a pocos les guste, que dos personas cambien el mismo archivo al mismo tiempo. Una intenta hacer `pull` antes de registrar cambios y resulta que BAM! hay un choque! Git nos avisa que el mismo archivo ha sido editado en las mismas líneas. Que desastre! Y ahora, qué debemos hacer? Pues la verdad, este asunto se resuelve de una manera bastante simple. Git nos permite ignorar cambios, o escoger cuál deseamos mantener y cual desechar. Hay herramientas especializadas para esto como KDiff. Este programa nos permite ver el código antes de ser editado, la edición de una persona, y la nuestra a modo de ver los cambios y escoger cuáles se quedan.

1.7. Comentarios Finales

Si observamos bien, no hicimos ningún ejercicio grande y especial con git. Esto se debe a que este sistema no sirve para hacer algo muy interesante dentro de un proyecto, más que organizarlo y mantener copias de seguridad de cada paso en el desarrollo de un proyecto. Más adelante en el taller, vamos a continuar utilizando git a modo de ir guardando nuestros avances y practicar su uso.

Por ahora, felicidades! Has completado la introducción al uso de un sistema de control de revisiones. Si deseas profundizar en el tema, se te recomienda hacerlo. Saber usar a mayor profundidad las ventajas que te ofrece git es algo de lo que jamás te arrepentirás. Y pues claro, llevar un registro decente y ordenado es una buena práctica en toda ciencia.

Felicidades de nuevo, y ánimo! Mañana toca aprender a utilizar una herramienta que hará que tus reportes nunca vuelvan a ser iguales.

2. Glosario de comandos sencillos

- **git config**: configura git con tu nombre, tu dirección de correo y tu editor de texto preferido.
- **git init**: inicia un nuevo proyecto en git de manera local; un nuevo repositorio.
- **git clone**: descarga los contenidos de un repositorio ya existente a tu ordenador y los deja listos para poder trabajar con ellos.
- **git status**: revisa el estado actual de los archivos en el repositorio en comparación al último commit.
- **git diff**: muestra claramente las diferencias entre el estado actual de los archivos en el repositorio con respecto al último commit. Muestra las diferencias línea por línea.
- **.gitignore**: archivo que contiene el listado de todos los archivos del proyecto que deben ser ignorados por git.
- **git commit**: guarda los cambios realizados a los archivos como una revisión. Crea un registro y un *id* que identifica a esa revisión.
- **git push**: empuja (sube) los commits locales a GitHub.
- **git pull**: jala (descarga) los archivos del proyecto en GitHub, actualizando el estado de estos en nuestro ordenador.
- **git log**: muestra toda la actividad de commits que ha habido en un repositorio/proyecto.
- **git branch**: muestra a todas las ramas existentes o, si se le agrega un nombre después del comando, se crea una rama nueva en el proyecto.
- **git merge**: une a una rama existente con otra rama que se debe de especificar como argumento del comando.
- **git checkout**: si como argumento se agrega el nombre de una rama, se habrá trasladado a esa rama. Si como argumento se agrega el *id* de un commit o la distancia de la cabeza a un commit con `HEAD~#` , entonces nos situaremos en el commit seleccionado y podremos ver el estado del proyecto en ese momento.

Licencia