Conmutación y Enrutamiento

Msig. Adriana Collaguazo

Introducción a Sistemas de Control de Versiones

16 de octubre de 2017

1. Acerca del Control de Versiones

¿Qué es el control de versiones y por qué deberías preocuparte? Un sistema de control de versiones – o VCS por sus siglas en inglés – es un sistema que registra cambios en un conjunto de archivos a lo largo del tiempo, de manera que puedas acceder a diferentes versiones en cualquier momento. Los VCS permiten revertir archivos o proyectos enteros a un estado previo, comparar sus cambios a lo largo del tiempo, revisar quién realizó un cambio en particular que pueda estar ocasionando un problema, entre otros. Usar un VCS implica que si en algún momento se vea comprometido el estado de los archivos, éstos puedan ser recuperados con normalidad.

2. ¿Qué es Git?

La diferencia más importante en cuanto a Git y otros VCS es la forma en la que Git maneja la data. Conceptualmente, otros sistemas almacenan la información como un conjunto de archivos así como los cambios que han tenido a lo largo del tiempo, tal como se ilustra en la Figura 1.

Por otro lado, Git trabaja la data como un conjunto de *snapshots* o copias instantáneas. Cada vez que se efectúa un *commit*, o se guarda el estado de un

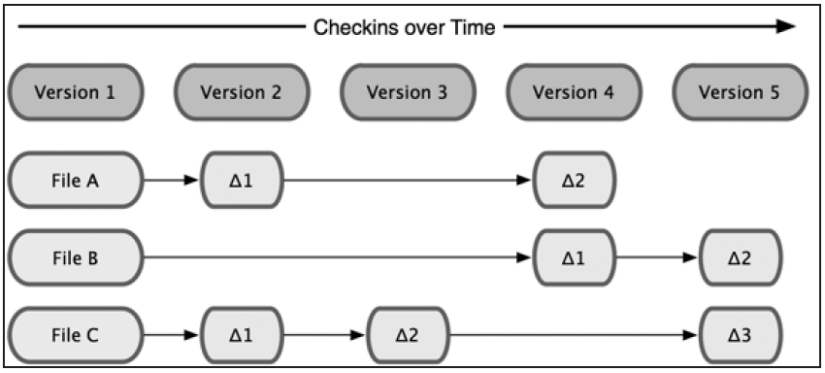

Figura 1: Otros VCS tienden a almacenar la data como cambios de la versión base de cada archivo

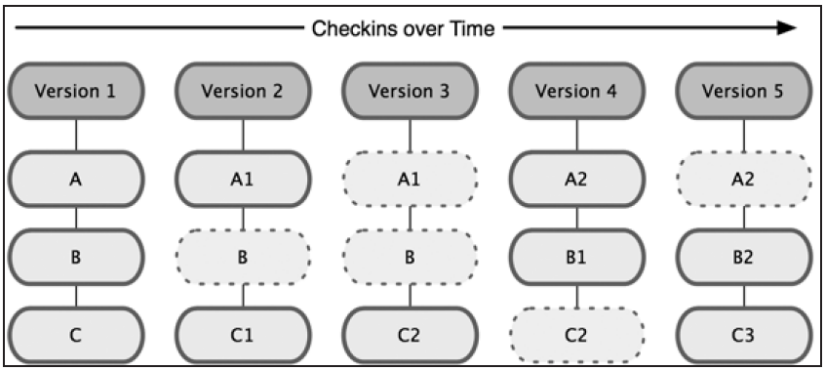

Figura 2: Git guarda la data como *snapshots* del proyecto a lo largo del tiempo.

proyecto en Git, básicamente se toma una foto de cómo todos los archivos lucen en ese preciso momento y almacena una referencia a esa copia instantánea. Para ser eficientes, si los archivos no hay cambiado, Git no almacenta los archivos de nuevo – solo toma un enlace al archivo idéntico más reciente que se haya guardado. Git piensa en la data como en la figura 2.

Git maneja tres estados sobre los archivos: *committed*, *modified* y *staged*. *Committed* significa que la data está almacenada de manera segura en nuestra base de datos local. *Modified* significa que se ha hecho cambios sobre un archivo pero no se le ha efectuado un commit en la base de datos. *Staged* significa que se ha indicado que un archivo modificado sea considerado a incluirse en el próximo commit. Esto nos lleva a presentar las tres secciones principales de un proyecto Git:

- El *directorio* `Git` es donde Git almacenta la metadata y la base de datos para nuestro proyecto.
- El *directorio de trabajo* es un simple despliegue o *checkout* de una versión del proyecto. Estos archivos son extraídos de la base de datos comprimida en el directorio `Git` y ubicados en disco para hacer uso de ellos.
- El *área de stage* es un simple archivo contenido en el directorio `Git` que almacena información acerca de lo que irá en el siguiente commit. Se lo conoce también como el *índice*.

El flujo de trabajo de Git se detalla a continuación:

1. Se modifican archivos en el directorio de trabajo.
2. Se generan *snapshots* de dichos archivos y se llevan al área de stage.
3. Se efectúa el commit, el cual toma los archivos tal como se indican en el área de stage y almacena dicho *snapshot* de forma permanente en el directorio `Git`.

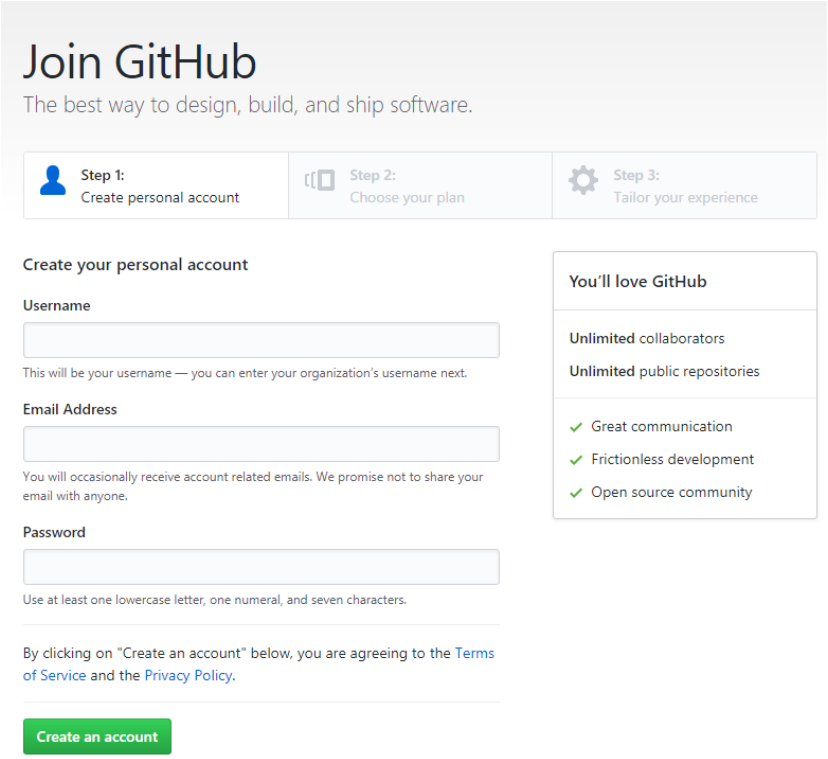

Figura 3: Ventana de sign up.

3. Instalación y configuración inicial

Antes de proceder a la instalación de Git, empezamos por crear una cuenta en Github (`https://github.com`) como se muestra en la figura 3. Github es un *forge* – o plataforma de desarrollo colaborativo – que permite alojar proyectos utilizando el sistema de control de versiones Git. Esta herramienta nos permitirá tener nuestro proyecto disponible en todo momento, así como tener un respaldo en caso que los archivos se vean comprometidos. Github es un poco diferente a muchos sitios de hosting de código fuente: es *user centric*. Esto quiere decir que cuando se hostee un proyecto `myproject` no lo encontrará en `github.com/myproject` sino en `github.com/<usuario>/myproject`. No hay una versión canónica de un proyecto, lo que permite que los proyectos puedan moverse de un usuario a otro sin importar si el primer author abandona el proyecto

Para MS Windows

Instalar Git en Windows es muy simple. El proyecto *msysGit* tiene uno de los procedimientos más simples de instalación. Simplemente descargue el archivo instalador *.exe* desde el enlace descrito abajo y ejecútelo. Luego de instalarlo, tendrá en su equipo una versión de Git por línea de comandos, así como una interfaz gráfica de usuario estándar.

https://git-for-windows.github.io/

Para Linux

Para instalar Git en Linux podemos hacerlo de dos formas: instalando desde código fuente o instalando un paquete existente para la distribución que se esté usando. Para más información, léase *What is the difference between building from source and using an install package?*: https://unix.stackexchange.com/questions/152346/what-is-the-difference-between-building-from-source-and-using-an-install-package

Instalando desde código fuente

Si se encuentra trabajando sobre una distribución basada en Fedora (*yum*) o Debian (*apt-get*), puede usar respectivamente los siguientes comandos.

```
$ yum install curl-devel expat-devel gettext-devel \
  openssl-devel zlib-devel
```

```
$ apt-get install curl-devel expat-devel gettext-devel \
  openssl-devel zlib-devel
```

Una vez instaladas las dependencias necesarias, descargue la versión correspondiente de Git – los archivos *.tar.gz* – desde la página de https://www.kernel.org/pub/software/scm/git/. Se recomienda descargar el más reciente.

```
$ wget https://www.kernel.org/pub/software/scm/git/git-2.9.5.tar.gz
```

Luego, compilar e instalar:

```
$ tar -zxf git-2.9.5.tar.gz
$ cd git-2.9.5
$ make prefix=/usr/local all
$ sudo make prefix=/usr/local install
```

Instalando desde gestor de paquetes

Si se encuentra trabajando sobre una distribución basada en Fedora, puede emitir el comando:

```
$ yum install git-core
```

Si se encuentra trabajando sobre una distribución basada en Debian como Ubuntu, puede emitir el comando:

```
$ apt-get install git-core
```

Si se encuentra trabajando en MS Windows, tenga en cuenta que a partir de este punto se usará la herramienta Git Bash. Si se encuentra trabajando en Linux, puede continuar con el Terminal de siempre. Una vez instalado Git en el equipo se procede a aplicar una plantilla básica de configuración con la información ingresada al crear la cuenta en Github. Esta tarea se efectuará una sola vez luego de la instalación, la misma que permitirá identificarlo en todo momento.

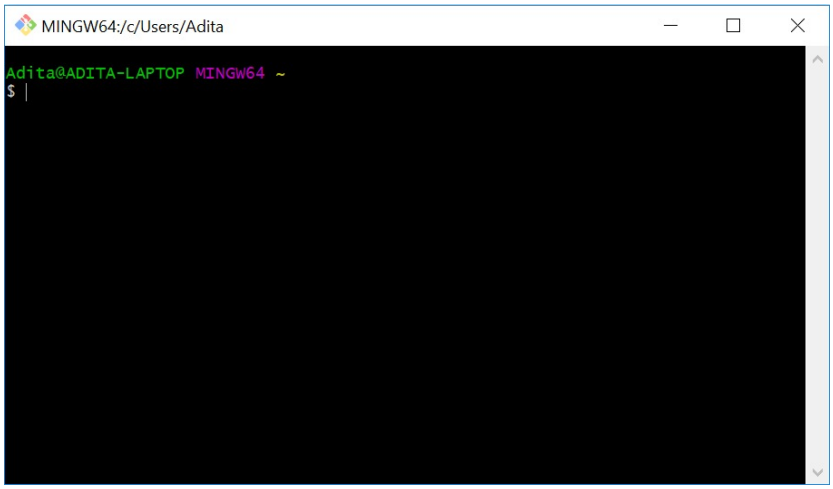

Figura 4: Git Bash en MS Windows.

```
$ git config --global user.name "YOUR_USERNAME"
$ git config --global user.email "your_email_address@example.com"
```

Si necesita ayuda mientras está usando Git, cualquiera de los siguientes comandos le ofrece información:

```
$ git help <verb>
$ git <verb> --help
```

Por ejemplo, si desea conocer información acerca del comando *config*:

```
$ git help config
```

Estos comandos son muy útiles ya que es posible acceder a ellos en todo momento, inclusive sin estar conectado a Internet. Adicional a esto, puede consultar el portal de StackExchange (`https://stackexchange.com`) con diversos contenidos de páginas especializadas como StackOverflow, Unix & Linux, AskUbuntu, entre otras.

4. Git Basics

A continuación, se revisarán los comandos básicos que se necesitan conocer para hacer la gran mayoría de las tareas que se realizan en Git. El objetivo de esta sección es lograr configurar e inicializar un repositorio, comenzar y detener el seguimiento de archivos, así como llevarlos a los estados *staged* y *commit*. En la figura 4 se muestra cómo luce la consola Git Bash en la que se trabajará si se encuentra en Windows.

Se puede obtener un repositorio Git de dos maneras. La primera es tomar un proyecto o directorio existente e importarlo a Git. La segunda es clonar un repositorio Git existente hacia otro equipo. Para empezar el *tracking* de un proyecto existente en Git, se necesita ir al directorio del proyecto e ingresar el comando:

```
$ git init
```

Este comando crea un nuevo subdirectorio llamado .git que contiene todos los archivos necesarios del repositorio – un *esqueleto* Git. En este punto, nada en el proyecto figura como *tracked* La herramienta principal para determinar qué archivos se encuentran *staged* es el comando `git status`.

```
$ git status
# On branch master
nothing to commit (working directory clean)
```

Esto significa que tenemos un directorio de trabajo limpio, es decir libre de archivos que figuren como *tracked* y *modified*. Ahora considere que desea agregar un nuevo archivo al projecto, un simple `README`. Un archivo `README` no es más que un resumen introductorio de lo que aloja el repositorio, y será lo primero que aparezca cuando se acceda a la ruta raíz del repositorio en Github (para más información de como tener un `README` estilizado y personalizado puede visitar: `https://github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet`). Si el archivo no existía antes y se emitió el comando `git status` veremos lo siguiente:

```
$ vim README
$ git status
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#    README
nothing added to commit but untracked files present (use "git add"
to track)
```

Podemos ver que el archivo `README` figura como *untracked*, es decir que Git ve un archivo que no constaba en el commit anterior; asimismo Git no va a incluirlo en los siguientes *snapshots* hasta que lo indique de forma explícita. Para ello, se usa el comando `git add`:

```
$ git add README
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#     new file:   README
#
```

Podemos ver que el archivo `README` figura como *staged* ya que se encuentra bajo la sección Çhanges to be committed". Si se realiza un commit en este punto, la versión del archivo al momento en que se ejecutó `git add` es la que irá en el historial de `snapshots`. Ahora, considere que se hace un cambio en un archivo que consta como *tracked*. Si se modifica un archivo *tracked* llamado `datos.txt` y se revisa el estado del proyecto, veremos lo siguiente:

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#     new file:   README
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
#     modified:   datos.txt
#
```

El archivo `datos.txt` aparece bajo la sección Çhanged but not updated", lo cual indica que el archivo *tracked* ha sido modificado en el directorio de trabajo pero no se encuentra como *staged*. Para ello, emitimos el comando `git add`:

```
$ git add datos.txt
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#     new file:   README
#     modified:   datos.txt
#
```

Ambos archivos figuran como *staged* y están aptos para ir en el siguiente commit. En este punto, suponga que recordó realizar un pequeño cambio en el archivo `datos.txt` antes de realizar el commit:

```
$ vim datos.txt
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#     new file:   README
#     modified:   datos.txt
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
#     modified:   datos.txt
#
```

¿Por qué razón el archivo `datos.txt` aparece como *staged* y *unstaged* a la vez? Sucede que Git marca un archivo como *staged* tan pronto como se emita el comando `git add`. Si se realiza un commit ahora, la versión de `datos.txt` que era cuando se emitió el comando `git add` es la que irá en el commit, no la versión del archivo como luce en el directorio de trabajo cuando únicamente se envíe el comando `git commit`. Si se modifica un archivo luego de emitir `git`

`add`, se deberá enviar dicho mando de nuevo para tener registro de la última versión del archivo.

Si bien es cierto que es muy útil personalizar lo que irá en los commits, el área de stage puede resultar en ocasiones tediosa y compleja. En caso de desear saltarse dicha etapa, Git provee un atajo: simplemente enviando el argumento `-a` al comando `git commit` hace que Git automáticamente etiquete como *staged* a cada archivo del proyecto que figure como *tracked*, permitiendo de esa manenera saltarse la parte `git add`.

```
$ git status
# On branch master
#
# Changed but not updated:
#
#     modified:   datos.txt
#
$ git commit -a -m 'agregando mas datos'
[master 83e38c7] agregando mas datos
 1 files changed, 5 insertions(+), 0 deletions(-)
```

Para remover un archivo de Git, debe removerse de los archivos *tracked* – exactamente, del área de stage – y luego realizar el commit. El comando `git rm` realiza ésto y también lo remueve del directorio de trabajo, así ya no aparecerá como archivo *untracked* en próximas ocasiones.

```
$ git rm oldREADME
rm 'oldREADME'
$ git status
# On branch master
#
# Changed to be committed:
#   (use "git reset HEAD <file>..." to ustage)
#
#     deleted:   oldREADME
#

$ git rm --cached login.txt
```

Adicional a esto, si lo que desea es un ambiente gráfico para visualizar su historial de commit y demás, puede usar el programa `gitk` que viene incorporado en la instalación inicial. Puede emitir el comando `gitk` y se mostrará la ventana inicial del programa.

```
$ gitk
```

Para poder colaborar en cualquier proyecto Git, es necesario saber administrar los repositorios remotos. Los *repositorios remotos* son versiones de un proyecto que son hosteados en Internet o en un recurso de red. Estos repositorios pueden ser de sólo-lectura o de lectura/escritura. El trabajo colaborativo involucra administrar estos repositorios, así como extraer (pull) o actualizar (push) data desde y hacia ellos. Para ver qué servidores remotos se han configurado, ejectuar el comando `git remote`; en el siguiente ejemplo se procede a clonar un proyecto Git y se muestra la URL donde Git lo tiene alojado.

```
$ git clone git://github.com/schacon/ticgit.git
$ cd ticgit
$ git remote
origin
$ git remote -v
origin     git://github.com/schacon/ticgit.git
```

Para añadir un nuevo repositorio Git se procede con el comando `git remote add [shortname] [url]`:

```
$ git remote add pb git://github.com/<username>/ticgit.git
$ git remote -v
origin     git://github.com/schacon/ticgit.git
pb     git://github.com/acollaguazo/ticgit.git
```

Ahora puede usar el string `pb` en la linea de comando en lugar de la URL completa. Por ejemplo, si desea obtener toda la información que el usuario `acollaguazo` tiene pero que usted no tiene en su repositorio, puede correr el comando `git fetch pb`:

```
$ git fetch pb
remote: Counting objects 58, done.
remote: Compressiong objects: 100% (41/41), done.
remote: Total 44 (delta 24), reused 1(delta 0)
Unpacking objects:100% (44/44), done,
From git://github.com/acollaguazo/ticgit.git
 * [new_branch]     master     -> pb/master
 * [new_branch]     ticgit     -> pb/ticgit
```

La rama master de `acollaguazo` está ahora accesible de manera local como `pb/master` – puede fusionarla en una de sus ramas, o puede checkout una rama local en ese punto si desea inspeccionarla. A través del comando `git fetch [remote-name]`, se puede ir hacia el proyecto remoto y bajarse toda la data que aún no se tiene. Una vez hecho esto, se tienen referencias hacia todas las ramas desde aquella remota, la cual podemos fusionar o inspeccionar en cualquier momento.

Si hemos clonado un repositorio, el comando automáticamente añade ese repositorio remoto bajo el nombre `origin`, de manera que la instrucción `git fetch origin` descarga cualquier novedad que se haya actualizado en ese servidor desde que fue clonado. Es importante tener en cuenta que el comando `fetch` baja la data a nuestro repositorio local, mas no lo fusiona con alguna rama o trabajo previo que nos encontremos realizando; para ello, hay que hacerlo de forma manual.

Cuando se tiene un proyecto a un punto donde se lo desee compartir, se debe de efectuar un `upstream`. El comando necesario es `git push [remote-name] [branch-name]`. Si desea realizar push de su rama master hacia el servidor `origin` debe tener permisos de escritura sobre el servidor al que desea acceder.

```
$ git push origin master
```

Para renombrar una referencia, puede ejecutar el comando `git remote rename [former-name] [new-name]`. Por ejemplo, si desea cambiar de `pb` a *myname*:

```
$ git remote rename pb myname
$ git remote
origin
myname
```

Vale destacar que esta acción modifica el nombre de las ramas. Es decir que lo que estaba referenciado hacia `pb/master` está ahora hacia `myname/master`. En caso de querer remover una referencia por alguna razón – migración de servidor o tal vez un colaborador no contribuirá más, puede usar el comando `git rm`:

```
$ git remote rm myname
$ git remote
origin
```

5. Resumen

Para más información de los recursos a utilizar en el curso, así como futuras referencias y proyectos puede acceder al siguiente enlace: `https://github.com/acollaguazo`. Además, puede consultar más temas específicos de Git en la documentación online: `https://git-scm.com/book/en/v2`.

Configuración inicial

```
$ git config --global user.name "<usuario-github>"
$ git config --global user.email "<correo-github>"
```

Crear un nuevo repositorio

```
$ git init
```

Crear una copia de trabajo de un repositorio local

```
$ git clone <ruta-local>
```

Crear una copia de trabajo Github

```
$ git clone git://github.com/<usuario>/<nombre-repositorio>
```

Llevar un archivo al área de stage

```
$ git add <nombre-archivo>
```

Realizando un commit

```
$ git commit -m "mensaje de commit"
```

Conectar un repositorio a un servidor remoto de Github

```
$ git remote add <atajo> https://github.com/<usuario>/<repo-existente>
```

Eliminar cualquier referencia remota al servidor

```
$ git remote rm origin
```

Efectuando *pushing* de los cambios

```
$ git push origin master
```

Para actualizar un repositorio local al commit más reciente

```
$ git pull
```