#### Universidad Autónoma del Sur “UNASUR”

****

# Ingeniería en Informática

# 2do. Año

## GIT

##### Nombre del profesor: Diego Francisco Correa

Integrantes:

* **Sergio Chamorro (sergiochamo96@gmail.com)**
* **José Paredes (joseparedespy@icloud.com)**
* **Julio Denis (moopiojulio@gmail.com)**
* **Iván Franco (octavioivan@live.com)**

**Asunción – Paraguay**

**Año 2016.**

Tabla de contenidos

1. *INTRODUCCION 3*
2. *OJETIVO 4*
   1. Objetivo general 4
   2. Objetivos específicos 4
3. *CONTENIDO 5*
   1. Capitulo - I 5
   2. Capitulo - II 5
   3. Capitulo – (n) 5
4. *ANEXOS 6*
5. *CONCLUSION 7*
6. *BIBLIOGRAFIA O WEBLIOGRAFIA 7*
7. **INTRODUCCIÓN**

Cuando realizamos algún proyecto de cualquier tipo siempre trabajamos sobre el mismo documento y lo vamos mejorando con el tiempo, pero hay veces que queremos modificar o probar algo y creamos otra versión del documento en otra carpeta, y así sucesivamente hasta que tenemos varias carpetas con distintos cambios o versiones del mismo proyecto, que en algún momento se convierte en un lio de carpetas.

Git es un sistema de control de versiones que se encarga de guardar los cambios realizados en un proyecto, separándolos por estados o versiones del archivo y el estado actual del mismo.

Nos da la posibilidad de poder recorrer esas versiones y tener acceso a cada cambio que se hizo en el proyecto, ya sea para encontrar errores o volver a la versión inicial, entre otras.

1. **OBJETIVOS** 
   1. **Objetivo general**

* Aprender a utilizar el control de versiones Git para crear repositorios donde tengamos registrados los cambios de un archivo o conjunto de archivos.
  1. **Objetivos específicos**
* Comprender los conceptos básicos de Git y las ventajas que nos provee.
* Conocer los comandos necesarios para la utilización de Git.
* Alojar un proyecto en GitHub para repositorios remotos.

1. **CONTENIDO**

**Control de Versiones**

Un control de versiones es un sistema que se utiliza para registrar los cambios que hacemos en un documento o proyecto a medida que pasa el tiempo. Nos da la posibilidad de poder volver a cualquier versión específica de nuestro proyecto, poder ver los cambios realizados en esas versiones, comparar los cambios que hicimos durante el proceso de desarrollo, ver quien modificó y en qué momento, entre otras cosas más.

Existen 3 tipos de sistemas de control de versiones, las cuales son:

* **Sistema de control de versiones locales**: Es cuando el estado actual de un archivo y su control de versiones se encuentran en una misma computadora.
* **Sistema de control de versiones centralizado**: Es cuando el estado actual del archivo se encuentra en las computadoras de los usuarios y el control de versiones se encuentra en un servidor.
* **Sistema de control de versiones distribuido**: Es cuando, tanto las computadoras de los usuarios como el servidor almacenan el archivo actual junto con su control de versiones.

**GIT**

Git es un sistema de control de versiones distribuido diseñado por Linus Torvalds, pensado en la eficiencia y la confiabilidad del mantenimiento de versiones de aplicaciones cuando éstas tienen un gran número de archivos de código fuente, es capaz de manejar grandes proyectos de forma rápida y eficiente (como el kernel de Linux). Tiene un gran soporte de desarrollo no lineal lo que permite miles de ramas paralelas y tiene un diseño sencillo.

**Diferencias con otros Sistemas de Control de Versiones**

La principal diferencia entre Git y cualquier otro VCS (Sistema de Control de Versiones) es la forma en la que manejan sus datos. La mayoría de los otros sistemas almacenan la información como una lista de cambios en los archivos, solo guardan las modificaciones hechas a cada uno de ellos a través del tiempo.

Git no maneja ni almacena sus datos de esta forma. Git maneja sus datos como un conjunto de copias instantáneas de un sistema de archivos miniatura. Cada vez que confirmas un cambio, o guardas el estado de tu proyecto en Git, él básicamente toma una foto del aspecto de todos tus archivos en ese momento, y guarda una referencia a esa copia instantánea.

**Gran parte de todas las operaciones se hacen de forma local**

La mayoría de las operaciones en Git sólo necesitan archivos y recursos locales para funcionar. Por ejemplo, para navegar por la historia del proyecto, Git no necesita conectarse al servidor para obtener la historia y mostrártela, simplemente la lee directamente de tu base de datos local. Esto significa que ves la historia del proyecto casi instantáneamente. Si quieres ver los cambios introducidos en un archivo entre la versión actual y la de hace un mes, Git puede buscar el archivo hace un mes y hacer un cálculo de diferencias localmente, en lugar de tener que pedirle a un servidor remoto que lo haga u obtener una versión antigua desde la red y hacerlo de manera local.

**Git tiene integridad**

Todo es verificado antes de ser almacenado, esto significa que es imposible cambiar los contenidos de cualquier archivo o directorio sin que Git lo sepa. No puedes perder información durante su transmisión o sufrir corrupción de archivos sin que Git sea capaz de detectarlo.

El mecanismo que usa Git realizar esto se conoce como hash SHA-1. Se trata de una cadena de 40 caracteres hexadecimales (0-9 y a-f), y se calcula en base a los contenidos del archivo o estructura del directorio en Git.

Un hash SHA-1 se ve de la siguiente forma:

*24b9da6552252987aa493b52f8696cd6d3b00373*

Estos valores son usados con mucha frecuencia porque Git guarda todo por el valor hash de sus contenidos y no por nombre.

**Git generalmente solo añade información**

Cuando realizas acciones en Git, casi todas ellas solo añaden información a la base de datos de Git. Es muy difícil conseguir que el sistema haga algo que no se pueda enmendar, o que de algún modo borre información. Como en cualquier VCS, puedes perder o estropear cambios que no has confirmado todavía. Pero después de confirmar una copia instantánea en Git es muy difícil de perderla, especialmente si envías tu base de datos a otro repositorio con regularidad.

**Los tres estados**

Git tiene tres estados principales en los que se pueden encontrar tus archivos:

* **Confirmado (committed):** significa que los datos están almacenados de manera segura en tu base de datos local.
* **Modificado (modified):** significa que has modificado el archivo, pero todavía no lo has confirmado a tu base de datos.
* **Preparado (staged):** significa que has marcado un archivo modificado en su versión actual para que vaya en tu próxima confirmación.

Esto nos lleva a las tres secciones principales de un proyecto de Git:

* **El directorio de Git (Git directory):** es donde se almacenan los metadatos y la base de datos de objetos para tu proyecto. Es la parte más importante de Git, y es lo que se copia cuando clonas un repositorio desde otra computadora.
* **El directorio de trabajo (working directory):** es una copia de una versión del proyecto. Estos archivos se sacan de la base de datos comprimida en el directorio de Git, y se colocan en disco para que los puedas usar o modificar.
* **El área de preparación (staging area):** generalmente está contenido en tu directorio de Git. Almacena información (un archivo modificado que has marcado como preparado) acerca de lo que va a ir en tu próxima confirmación.

**Ramas y Fusiones**

Una de las ventajas de Git es el trabajo con ramas, se pueden crear ramas de forma muy fácil y es cómodo trabajar con ellas. Es importante tener en claro que es una rama, para que sirve, así para explotar todo el potencial que proporcionan.

Otro concepto relacionado con las ramas son las fusiones, que nos permite unir el trabajo desarrollado en diferentes ramas.

Antes de explicar lo que son las ramas y fusiones debemos tener en cuenta el concepto de head. ¿Cómo sabe git en que rama estas en algún momento dado?, mediante un apuntador especial denominado head.

El concepto de head simplemente es el commit donde nosotros nos encontramos ahora, es decir si nosotros hacemos un **$ git checkout** al primer commit, segundo o cualquier commit ese va ser nuestro head.

Si vamos a nuestro proyecto y ponemos en la consola:

$ git log

Vemos todos los commit que hicimos en el proyecto, el ultimo commit es nuestro head. Un comando que nos indicará más específicamente cuál es nuestro actual head es:

$ git log --pretty --decorate

Este comando nos listará todos los commit, con sus respectivos autores, fecha de creación, además nos indica el head actual.

**¿Qué es una rama?**

Es una línea de tiempo en nuestro proyecto, que nos sirven para arreglar errores, experimentar, para hacer cambios de interfaz, cambios de lógica... cualquier cambio que no afecte a nuestro proyecto actual en sí.

La rama por defecto de Git es **master**. Con la primera confirmación de cambios que realicemos, se creará esta rama principal master apuntando a dicha confirmación.

En cada confirmación de cambios que realicemos, la rama irá avanzando automáticamente.

Si queremos ver las ramas con las que contamos en nuestro proyecto colocamos en la consola:

$ git branch

Ese comando nos mostrará todas las ramas de nuestro proyecto, nos marca con un asterisco [**\***] y nos colorea la rama en la que nos encontramos en este momento.

Por ejemplo:

$ git branch

**\* master**

**Crear una nueva rama**

Supongamos que estamos trabajando en un proyecto y queremos añadir una nueva funcionalidad al mismo. La forma adecuada de hacerlo con Git es crear una nueva rama con el nombre de la funcionalidad donde añadiremos nuestros cambios.

Comando para crear una nueva rama:

$ git branch Desarrollo

Una vez creada nuestra rama, en este ejemplo **Desarrollo**, podemos comprobar en que rama nos encontramos utilizando el comando **$ git branch.**

$ git branch

Desarrollo

**\* master**

Vemos que seguimos teniendo master coloreado con asterisco, eso es porque donde estamos ahora, pero ya tenemos la rama Desarrollo que habíamos creado. ¿Entonces cómo nos movemos a Desarrollo?

Para eso usamos el siguiente comando:

$ git checkout Desarrollo

Existe un atajo para crear y movernos a una rama que aún no existe usando directamente el comando checkout, que consiste en pasar el parámetro **-b** en la llamada.

$ git checkout –b Desarrollo

Si volvemos a colocar en la consola **$ git branch**, vemos que ahora la rama Desarrollo es la que tiene el asterisco y esta coloreada, eso quiere decir que es la rama en la que nos encontramos ahora.

$ git branch

**\* Desarrollo**

master

En ejemplo anterior podemos ver como nos encontramos en la rama Desarrollo. Esto significa que, si modificamos, borramos o añadimos archivos, ésta rama Desarrollo incluirá esos cambios, pero no así la rama master.

**FUSIONES**

Cuando se trabaja con ramas en Git, en algún momento el trabajo realizado en una rama será fusionado con otra rama. Por ejemplo, hemos testeado profundamente nuestros cambios y estamos listos para moverla de la rama Desarrollo a la rama principal master. Dicha acción requiere que mezclemos el contenido de una de las ramas dentro de la otra.

Para realizar la mezcla de la rama Desarrollo dentro de master vamos a situarnos en la rama master y a mezclar ambas con el siguiente comando:

$ git checkout master

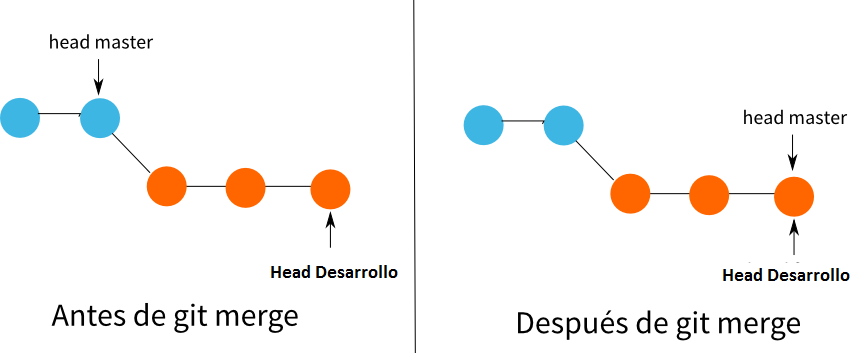
$ git merge Desarrollo

Visualizamos los commit de master y vemos que los commit de Desarrollo ya están en la rama, absorbimos los cambios de la rama Desarrollo. Una vez mezclado nuestras ramas, y en caso de que no haya conflictos, podemos eliminar la rama Desarrollo por que no vamos a necesitarla más con el siguiente comando:

$ git branch –d Desarrollo

Ahora, después que el merge se realiza con éxito en nuestro repositorio puede darse dos posibles casos: un fast-forward (merge de avance rápido) o un 3 way merge (merge a tres bandas).

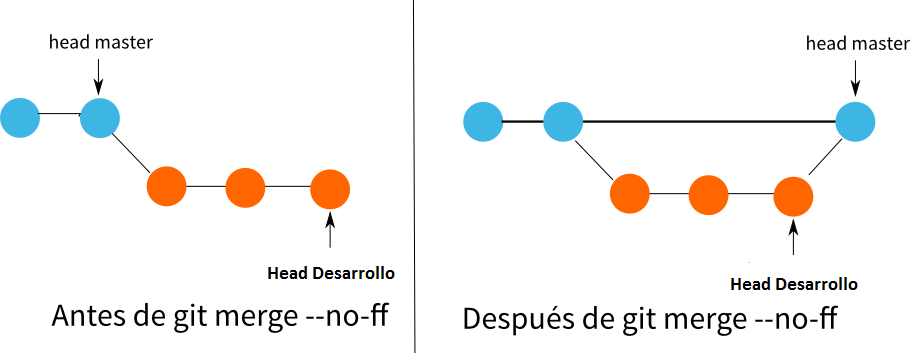
Un **fast-forward merge** es cuando al momento de hacer merge con la rama master no se ha añadido ningún commit luego de crear Desarrollo, es decir que el Head de master es el antepasado de la rama Desarrollo. Por lo tanto, en este caso no se genera un nuevo commit para agregar los commits de la rama Desarrollo, en vez de ello, el Head de master se actualiza al Head de la rama Desarrollo, sin crear un commit de merge adicional, de allí su nombre, fast-forward o avance rápido, como se muestra en la imagen:



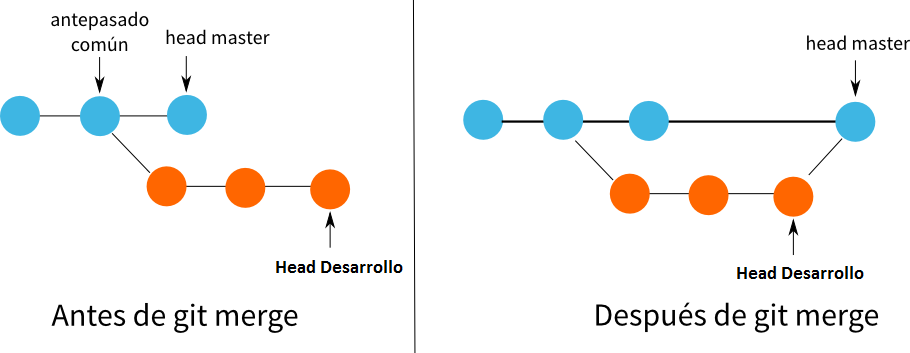
Pero si realmente queremos que se genere un commit al hacer merge para documentar o dejar constancia que se hizo merge de otra rama en master debemos usar la opción **--no-ff**, es decir

$ git merge --no-ff Desarrollo

Quedando la rama master de esta manera:



Por otro lado, si la rama master ha divergido después de haber creado la rama Desarrollo ya no es posible un fast-forward merge, debido a que el commit de la rama donde actualmente se está (master) no es un antepasado directo de la rama a fusionar (Desarrollo) por tanto, Git realiza un merge a tres bandas, es decir, que genera un commit para fusionar las dos ramas, tomando en cuenta el Head de cada una de ellas y el antepasado común de las dos:



1. **ANEXOS**

Cualquier información adicional ya sea de carácter informativo o gráfico.

1. **CONCLUSION**

Resumen de los aspectos más importantes aprendidos durante el desarrollo del trabajo.

1. **BIBLIOGRAFIA O WEBLIOGRAFIA**

* Libro “Pro Git v2 - 2014” *Scott Chacon and Ben Straub*
* <https://es.wikipedia.org/wiki/Git>
* <http://wpmallorca.com/2013/02/12/pero-que-es-github/>
* <https://es.wikipedia.org/wiki/Control_de_versiones>