



Unidad 3 / Escenario 5

Lectura fundamental

# Gestión de código

## Contenido

- 1** Estructura manejo de repositorios
- 2** Comandos
- 3** Operaciones generales de gestión y revisión de código entre varios desarrolladores
- 4** Buenas practicas

**Palabras clave:** commit, repositorio, branch, trunk.

# 1. Estructura manejo de repositorios

Los repositorios son instrumentos utilizados para almacenar datos, de tal forma que sea sencillo recuperar e instalar paquetes de software manteniendo un orden de cada versión que se haya desarrollado, mediante un completo historial de cambios. Existen tres grandes estructuras que necesitamos conocer para el manejo del mismo: (Mateo, 2007)

## 1.1. Trunk

Trunk o tronco hace referencia a la línea principal de desarrollo, en esta línea se aconseja no realizar cambios directamente, en caso de realizarse deben ser de pequeña escala, generalmente, esta línea principal es la línea de producción y es asociada a la rama master.

## 1.2. Tags:

Cada que se requiere identificar o resaltar un punto importante dentro de la historia de desarrollo se puede resaltar mediante etiquetas (tags), generalmente esta funcionalidad es utilizada para marcar puntos donde se ha lanzado alguna versión (V1.0, V2.0, V3.0).

En Git podemos encontrar dos tipos de etiquetas:

1. Etiquetas anotadas: estas etiquetas son almacenadas dentro de Git como un objeto completo en la base de datos, dentro de los atributos que pueden ser agregados son: el nombre del etiquetador, correo electrónico, fecha y un mensaje de etiquetado, para especificar una etiqueta anotada es necesario agregar `-a` al ejecutar el comando `tag`.

```
Eduardos-MacBook-Pro:ProyectoEnfasis eduardo$ git tag -a v1.0 -m 'my version 1.0'
```

**Figura 1. Pantallazo creación de tags a través de línea de comandos**

*Fuente:* Elaboración propia (2017).

En la figura 1 podemos observar cómo incluir este tipo de etiquetas, la inclusión del texto `-m` especifica el mensaje, el cual se almacena con la etiqueta; al no especificar un mensaje, Git lanzará un editor para poder escribirlo, la etiqueta `-s` se usa para firmar las etiquetas con GnuPG siempre que una clave privada sea establecida.

De igual forma, posterior a su creación y en el momento en el que queramos observar la información de una versión, podemos observar con el comando Git show y el nombre de la versión que queremos analizar.

```
Eduardos-MacBook-Pro:ProyectoEnfasis eduardo$ git show v1.0
tag v1.0
Tagger: Eduardo Oliveros Acosta <eduardo@Eduardos-MacBook-Pro.local>
Date: Thu Aug 17 22:15:43 2017 -0500

my version 1.0

commit cebb27e0b49b8b67b8e73d69352a665de6dbf745
Author: Eduardo Oliveros Acosta <eduardo@Eduardos-MacBook-Pro.local>
Date: Thu Aug 17 17:35:07 2017 -0500

    created file2.txt

diff --git a/file2.txt b/file2.txt
new file mode 100644
index 0000000..0b013f2
--- /dev/null
+++ b/file2.txt
@@ -0,0 +1 @@
+example 2
```

**Figura 2. Pantallazo Git show, descripción de la versión**

Fuente: Elaboración propia (2017).

Dentro de la información más importante, encontramos los mensajes de la versión, el desarrollador que realizó el commit, correo electrónico, datos descriptivos de fecha y documentos que fueron cambiados, entre otros.

2. **Etiquetas ligeras:** es una etiqueta simple, utilizada para especificar una versión, sin ninguna otra información para añadirla; es necesario omitir las opciones `-a`, `-s` o `-m`, esta etiqueta no es tan detallada, pero es bastante útil para mensajes concretos y descriptivos, simplificando el proceso de etiquetación.

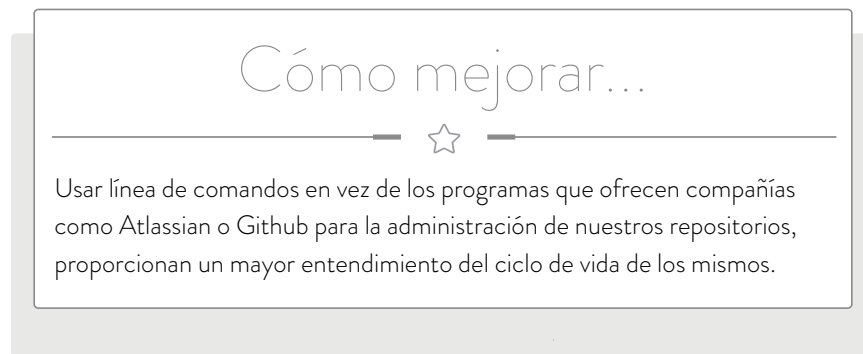
### 1.3. Branches

Los Branches o ramas, son simplemente apuntes móviles que apuntan a commits o confirmaciones de cambios realizados. Es importante crear una ramificación en el momento en el

que se vaya a realizar un cambio significativo dentro de nuestro desarrollo, cada rama tiene una copia de código de la rama de la que se cree la bifurcación, una vez terminados los cambios y habiendo realizado todas las pruebas terminadas se debe integrar los cambios en el Trunk.

## 2. Comandos

Habiendo instalado Git dentro del sistema operativo, es posible realizar la administración de nuestros repositorios a través de comandos, dentro de los comandos más utilizamos encontramos: (s.a.).



### 2.1. Git init

El comando Git init permite la inicialización de un repositorio al igual que reiniciar un repositorio existente.

```
Eduardos-MacBook-Pro:gitExample eduardo$ git init
Initialized empty Git repository in /Users/eduardo/gitExample/.git/
```

**Figura 3. Pantallazo comando de inicialización de repositorios**

Fuente: Elaboración propia (2017).

Posteriormente, al ejecutar este comando se crea una carpeta .git la cual contiene los directorios de la imagen número 2:

```
Eduardos-MacBook-Pro:~ eduardo$ ls
HEAD      config    hooks    objects
branches  description  info     refs
```

**Figura 4. Pantallazo organización archivos git**

Fuente: Elaboración propia (2017).

El archivo HEAD se refiere a la rama del commit más reciente; cuando cambias entre ramas, HEAD se actualiza refiriéndose a la nueva rama del último *commit*.

## 2.2. Git clone

Este comando clone realiza una copia local del repositorio al que se esté apuntando dentro de un nuevo directorio como se observa en la figura 5.

```
Eduardos-MacBook-Pro:gitExample eduardo$ git clone https://github.com/eduardoliveros/ProyectoEnfasis.git
Cloning into 'ProyectoEnfasis'...
remote: Counting objects: 6, done.
remote: Total 6 (delta 0), reused 0 (delta 0), pack-reused 6
Unpacking objects: 100% (6/6), done.
```

**Figura 5. Pantallazo clone local de un repositorio**

Fuente: Elaboración propia (2017).

## 2.3. Git status

Este comando te permite observar las diferencias dentro de los archivos de la carpeta o repositorio local y los archivos del commit HEAD actual, la rama en la que se encuentra y el estado de la rama.

```
Eduardos-MacBook-Pro:ProyectoEnfasis eduardo$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        deleted:    Jenkinsfile

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        file1.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

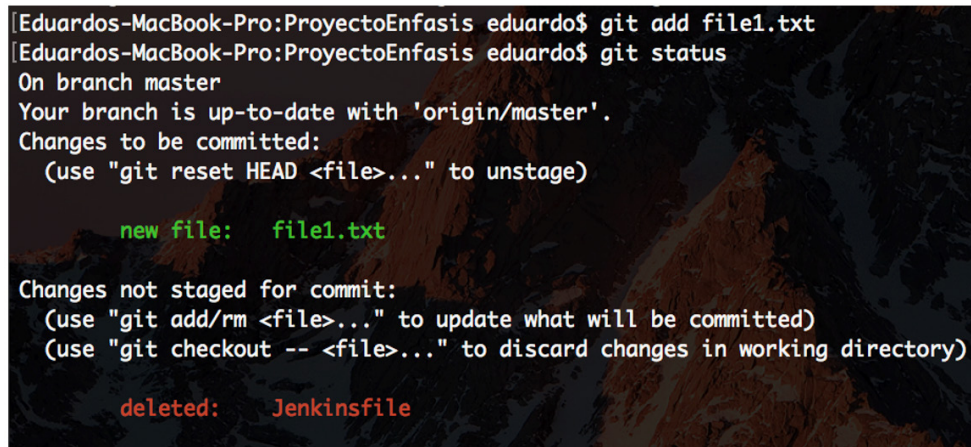
**Figura 6. Pantallazo organización servidores dentro de la integración continua**

Fuente: Elaboración propia (2017).

Como se puede observar en la figura 6, todos los cambios a los archivos serán listados, en este caso, se ha borrado un archivo (Jenkins file) y se ha creado un archivo nuevo file1.txt, el color de los archivos depende de la configuración de tu editor de texto; al agregar los archivos al commit el color cambiará indicando que se ha actualizado el commit como se observa en la Figura 7.

## 2.4. Git add

Este comando agrega los archivos al commit, en el momento en que se quieran agregar todos los cambios no es necesario especificar el nombre del archivo “(git add file1.txt)” es posible usar la instrucción “.”



```
Eduardos-MacBook-Pro:ProyectoEnfasis eduardo$ git add file1.txt
Eduardos-MacBook-Pro:ProyectoEnfasis eduardo$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   file1.txt

Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

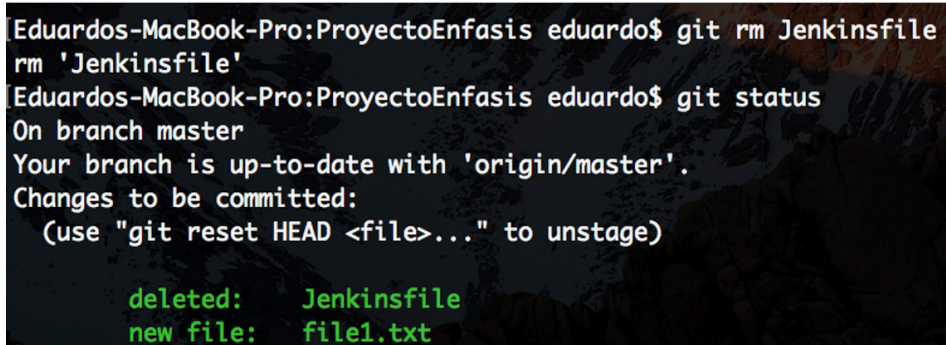
    deleted:    Jenkinsfile
```

**Figura 7. Pantallazo Git add y Git status**

Fuente: Elaboración propia (2017).

## 2.5. Git rm

Este comando es utilizado en operaciones opuestas a Git add, cuando has eliminado un archivo, para que ese cambio sea agregado al commit previamente, es necesario utilizarlo como observamos en la figura 8.



```
Eduardos-MacBook-Pro:ProyectoEnfasis eduardo$ git rm Jenkinsfile
rm 'Jenkinsfile'
Eduardos-MacBook-Pro:ProyectoEnfasis eduardo$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    deleted:    Jenkinsfile
    new file:   file1.txt
```

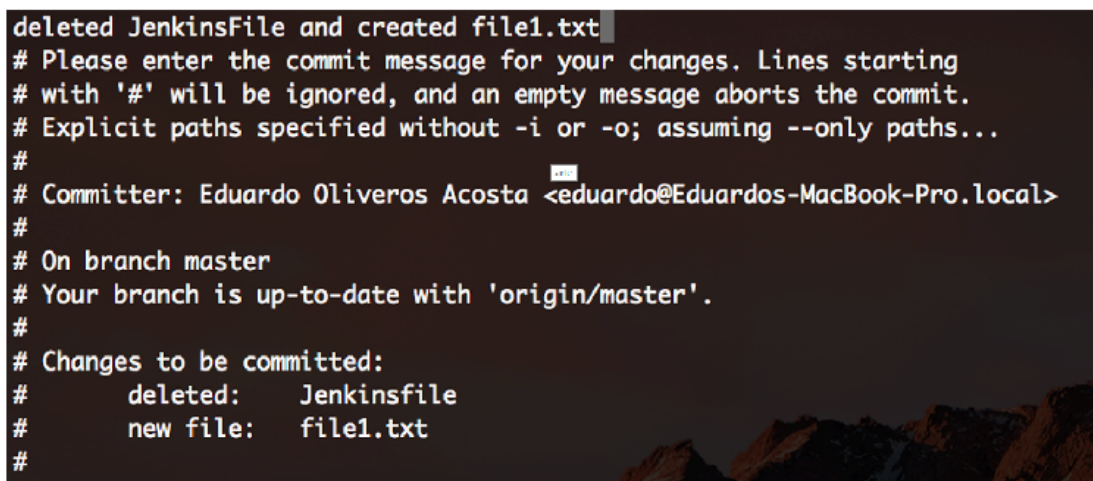
**Figura 8. Pantallazo Comando rm**

Fuente: Elaboración propia (2017).



## 2.6. Git commit

El comando Git commit registra los cambios realizados dentro del directorio contenedor, dentro del repositorio local, de esta forma solo queda actualizar el repositorio para que los cambios se vean reflejados en el repositorio global; cada commit debe tener una descripción específica, que dé al lector un fácil entendimiento del fin de ese cambio en específico; en caso de ser corto y conciso, ayuda a la revisión posterior del mismo.

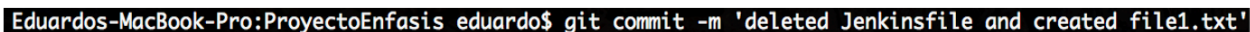
A screenshot of a terminal window showing the initial Git commit message editor. The text is as follows:

```
deleted JenkinsFile and created file1.txt
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# Explicit paths specified without -i or -o; assuming --only paths...
#
# Committer: Eduardo Oliveros Acosta <eduardo@Eduardos-MacBook-Pro.local>
#
# On branch master
# Your branch is up-to-date with 'origin/master'.
#
# Changes to be committed:
#   deleted:   Jenkinsfile
#   new file:  file1.txt
#
```

**Figura 9. Pantallazo ejemplo mensajes commit inicial**

Fuente: Elaboración propia (2017).

Una forma rápida de insertar mensajes al commit en una sola línea es usando el `-m` (Figura 10) esto agrega un atajo para poner el texto, sin hacer uso de un editor, siempre teniendo en cuenta que el mensaje sea concreto y explicativo.

A terminal screenshot showing the command to create a commit with a message using the -m flag. The text is:

```
Eduardos-MacBook-Pro:ProyectoEnfasis eduardo$ git commit -m 'deleted Jenkinsfile and created file1.txt'
```

**Figura 10. Pantallazo implementación `-m`**

Fuente: Elaboración propia (2017).

## 2.7. Git push

Este comando es utilizado para actualizar las referencias globales del repositorio, es necesario especificar la rama a la que se vaya a realizar la actualización; en el caso de la figura es la rama principal, dentro del log de este comando mostrara el nombre del commit del repositorio y el número de archivos que se han modificado, como se observa en la figura 11.

```
Eduardos-MacBook-Pro:ProyectoEnfasis eduardo$ git push origin master
Counting objects: 3, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 344 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/eduardoliveros/ProyectoEnfasis.git
bc19c8b..82f62d8 master -> master
```

**Figura 11. Pantallazo comando git push**

Fuente: Elaboración propia (2017).

## 2.8. Git branch

Este comando tiene distintos usos, es utilizado para listar, crear o eliminar ramas; en el caso de listar las ramas usadas localmente, únicamente es necesario agregar algo más que el comando Git branch, como se muestra en la figura 12.

```
Eduardos-MacBook-Pro:ProyectoEnfasis eduardo$ git branch
* master
```

**Figura 12. Comando git branch**

Fuente: Elaboración propia (2017).

En el caso en que se desee crear una rama, únicamente es necesario agregar a este comando el nombre de la nueva rama; en nuestro caso develop. El nombre de la rama puede ser bastante descriptivo dependiendo del equipo de desarrollo; un ejemplo de nombre eficaz sería JP/bug32 este podría indicar las siglas del nombre del desarrollador (Juan Pérez) y el bug que ha solucionado dentro de esta rama.

```
Eduardos-MacBook-Pro:ProyectoEnfasis eduardo$ git branch develop
```

**Figura 13. Pantallazo creación de ramas**

Fuente: Elaboración propia (2017).

Al querer listar las ramas de nuevo, se va a ver reflejado el cambio de nuevo.



```
Eduardos-MacBook-Pro:ProyectoEnfasis eduardo$ git branch  
develop  
* master
```

**Figura 14. Pantallazo Listar ramas existentes**

Fuente: Elaboración propia (2017).

En el caso de querer eliminar una rama, es utilizado el tag `-d` como vemos en la figura 14, en este caso es utilizado cuando la rama está directamente a una funcionalidad específica que ya ha sido solucionada.

```
Eduardos-MacBook-Pro:ProyectoEnfasis eduardo$ git branch -d develop  
Deleted branch develop (was 82f62d8).
```

**Figura 15. Pantallazo eliminar ramas**

Fuente: Elaboración propia (2017).

## 2.9. Git checkout

El comando checkout tiene distintos usos; uno de los más importantes podemos observarlo en la figura 15, el cual permite la migración a diferentes ramas para continuar el trabajo sobre la misma.

```
Eduardos-MacBook-Pro:ProyectoEnfasis eduardo$ git checkout develop  
Switched to branch 'develop'  
Eduardos-MacBook-Pro:ProyectoEnfasis eduardo$ git branch  
* develop  
master
```

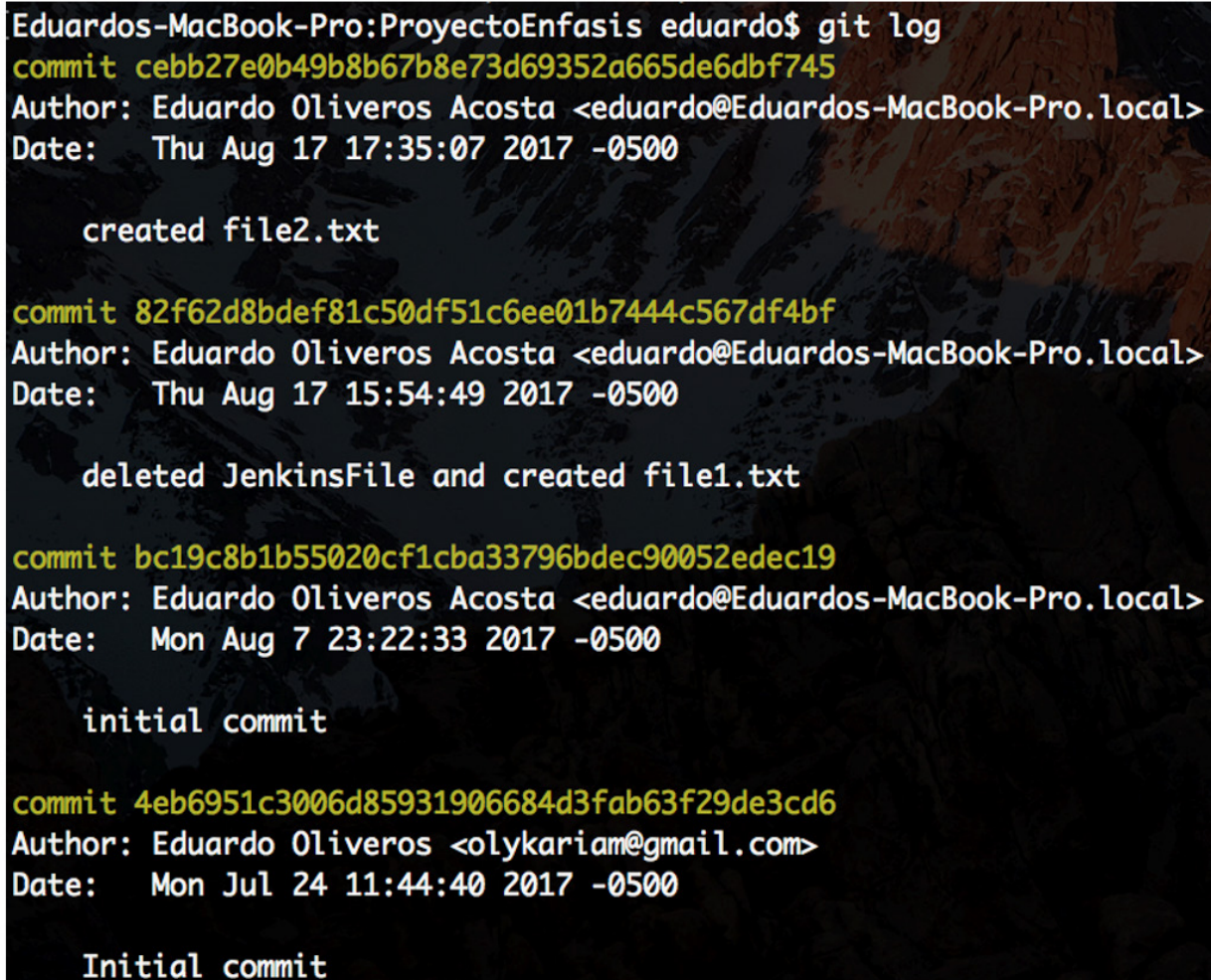
**Figura 16. Pantallazo cambio de ramas**

Fuente: Elaboración propia (2017).

En caso de que no se quiera aplicar cambios dentro de un archivo específico, es posible utilizar este comando en el archivo, de esta forma no será agregado en el siguiente commit.

## 2.10. Git log

Este comando es utilizado para observar el listado de commits realizados a través del tiempo, de la rama actual; los commits listados tienen como datos identificadores, autor, fecha y mensaje, es utilizado para recuperar versiones y restaurar cambios que se deseen recuperar.



```
Eduardos-MacBook-Pro:ProyectoEnfasis eduardo$ git log
commit cebb27e0b49b8b67b8e73d69352a665de6dbf745
Author: Eduardo Oliveros Acosta <eduardo@Eduardos-MacBook-Pro.local>
Date: Thu Aug 17 17:35:07 2017 -0500

    created file2.txt

commit 82f62d8bdef81c50df51c6ee01b7444c567df4bf
Author: Eduardo Oliveros Acosta <eduardo@Eduardos-MacBook-Pro.local>
Date: Thu Aug 17 15:54:49 2017 -0500

    deleted JenkinsFile and created file1.txt

commit bc19c8b1b55020cf1cba33796bdec90052edec19
Author: Eduardo Oliveros Acosta <eduardo@Eduardos-MacBook-Pro.local>
Date: Mon Aug 7 23:22:33 2017 -0500

    initial commit

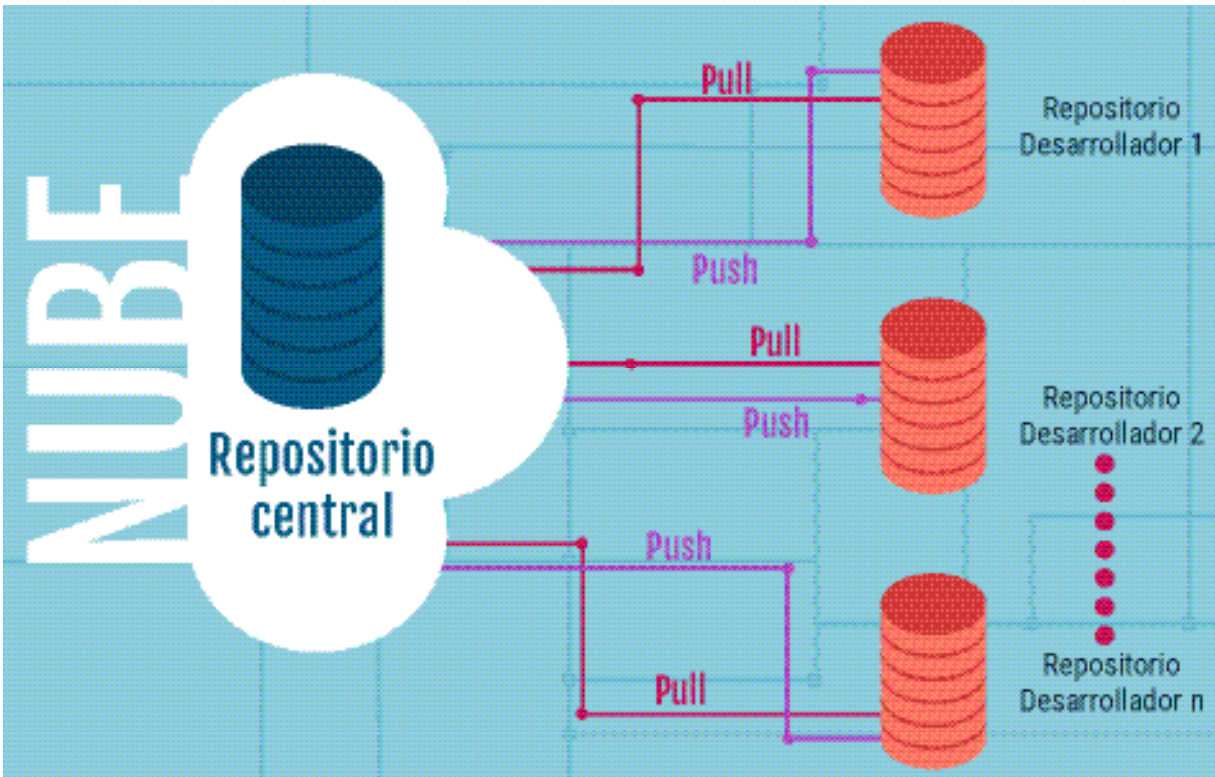
commit 4eb6951c3006d85931906684d3fab63f29de3cd6
Author: Eduardo Oliveros <olykariam@gmail.com>
Date: Mon Jul 24 11:44:40 2017 -0500

    Initial commit
```

**Figura 17. Pantallazo listar cambios realizados**

Fuente: Elaboración propia (2017).

### 3. Operaciones generales de gestión y revisión de código entre varios desarrolladores



**Figura 18. Organización general de repositorios**

Fuente: Elaboración propia (2017).

Como observamos en la figura, la organización de trabajo de gestión de un equipo de desarrollo es distribuida de acuerdo al número de repositorios de desarrollo o número de desarrolladores asignados al proyecto; dentro del caso mínimo se encuentra un escenario administrado por 2 desarrolladores.

Como podemos observar, cada repositorio (local) está aislado de la comunicación entre ellos, de igual forma cada repositorio mantiene la comunicación con el repositorio central el cual puede ser administrado por herramientas como *Github* y *Bitbucket*. La comunicación entre los repositorios locales y el repositorio central alojado en la nube, será a través de operaciones *push* y *pull*.

Para una correcta configuración y administración de cambios de cada uno de los desarrolladores, generalmente se presentan 5 tipos de ramas:

1. **Trunk:** generalmente la rama master de desarrollo, esto quiere decir, la rama principal, la cual estará ligada a la versión de producción del *software*. En esta rama se busca no realizar cambios directamente, debido a que una falla puede llevar a la dilatación de un proyecto, incluso al fracaso.
2. **Hotfix:** esta rama debe ser utilizada para realizar operaciones de solución de errores presentados en la rama master por lo que el ciclo de operación es bastante pequeño y al momento de realizar los cambios dentro de esta rama temporal, aseguramos de no interferir la rama principal. Hasta no estar completamente seguros de la solución de errores, no se debe integrar a la rama principal nuevamente; al momento de integrar los cambios, todos los desarrolladores deben ser informados para realizar la actualización de sus repositorios locales.
3. **Release:** esta rama es la previa al lanzamiento, debe contener cada nueva mejora, tarea o funcionalidad correctamente testeada, de esta forma es posible descargar cada avance funcional de la versión que se está construyendo.
4. **Develop:** cada funcionalidad o feature terminada, debe ser indexada a esta rama hasta el siguiente lanzamiento (release). Al momento que todas las funcionalidades estén listas en esta rama, se prueba como versión “estable” para ser añadida a la rama release.
5. **Feature:** cada feature, mejora, *bug* o advertencia, puede ser realizada en ramas individuales, de esta forma se puede hacer un seguimiento individual a cada tarea, en caso de querer a una versión estable será más fácil.

¿Sabía qué...?



La división del proyecto entre varias ramas permite la recuperación de versiones en el caso de ser necesario, siempre es mejor previo a migrar los cambios a la rama siguiente tener seguridad del correcto funcionamiento.



**Figura 19. Ciclo de vida ramas dentro del proyecto**

Fuente: Elaboración propia (2017).

## 4. Buenas practicas:

Dentro del manejo de repositorios es importante tener en cuenta:

1. Antes de comenzar a trabajar, es necesario hacer update de la rama en la que se está trabajando, o *checkout* hacia la misma para empezar a trabajar. Es probable que mientras no se trabaje sobre esa rama, el actualizar tu rama permite no hacer caso omiso a los cambios desarrollados por otro desarrollador.
2. El desarrollador siempre debe tener en cuenta que es parte de un equipo de desarrollo y debe pensar como equipo.
3. Cuando exista un conflicto por la modificación simultánea de archivos, un desarrollador debe revisar los cambios con el mejor criterio, teniendo en cuenta la opinión de los demás desarrolladores involucrados.



4. Los mensajes de commit deben ser claros y directos, en una sola línea se debe saber la finalidad del mismo y qué se solucionó con ese nuevo desarrollo.
5. Es necesario hacer commit al terminar el cambio o funcionalidad que se requiera, al terminar, se debe informar al equipo de desarrollo interesado del proyecto.
6. El commit debe limitarse a solucionar la tarea en específico, al combinar diferentes cambios en un mismo *commit* sin un objetivo compartido, puede dificultar la revisión y documentación de los mismos.

## En síntesis...

Tener presente que se es parte de un equipo siempre es la mejor práctica, estar en constante comunicación y verificar el estado de las ramas, mitigan el riesgo de pérdida de información.



# Referencias

Documentación Git (s.d). Recuperado de: <https://git-scm.com/doc>

Laguiar. (2016). *Wiki.genexus.com*. Recuperado de: [https://wiki.genexus.com/commwiki/servlet/wiki?29683,Manejo+de+ambientes+\(Desarrollo%2C+Testeo%2C+PreProduccion%2C+Producci%C3%B3n\)](https://wiki.genexus.com/commwiki/servlet/wiki?29683,Manejo+de+ambientes+(Desarrollo%2C+Testeo%2C+PreProduccion%2C+Producci%C3%B3n))

(Desarrollo%2C+Testeo%2C+PreProduccion%2C+Producci%C3%B3n)

Martinfowler. (2006). *Martinfowler.com*. Recuperado de: <https://martinfowler.com/articles/continuousIntegration.html>

Mateo, F. G. (2007). *Estructura manejo de repositorios trunk branch*. Recuperado de: <https://www.um.es/atika/documentos/NORPuestaEnLineaAplicacionesWeb.pdf>

## INFORMACIÓN TÉCNICA



**Módulo:** Énfasis Profesional I (Integración continua)

**Unidad 3:** Administración avanzada de repositorios

**Escenario 5:** Gestión de Código

**Autor:** Eduardo Enrique Oliveros Acosta

**Asesor Pedagógico:** Amparo Sastoque Romero

**Diseñador Gráfico:** Paola Andrea Melo

**Asistente:** Eveling Peñaranda

*Este material pertenece al Politécnico Gran Colombiano. Por ende, es de uso exclusivo de las Instituciones adscritas a la Red Ilumino. Prohibida su reproducción total o parcial.*