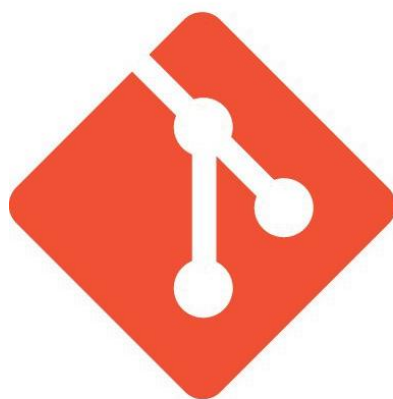


2015

Manual de Usuario Git Bash



git

Mario Alberto Martínez Calvio

Banco Central de Reserva

01/01/2015

Contenido

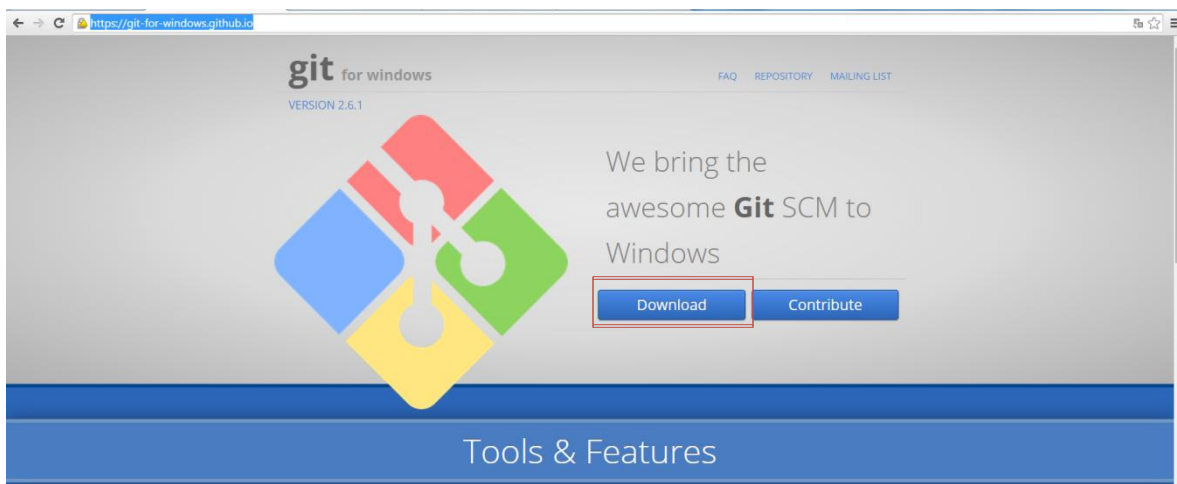
1. Instalación de Git en plataforma Windows	3
1.1 Descargando Git.....	3
2. Generando tu llave publica ssh.....	7
2.1 Verificando si ya tenemos una llave existente	7
2.2 Generando nueva llave ssh.....	7
3. Configuración de GIT	9
3.1 Tu identidad	9
4. Fundamentos de Git	10
4.1 Creando repositorios	10
4.2 Agregando archivos y primer Commit.....	11
4.3 Eliminando archivos	16
4.4 Viendo histórico de confirmaciones.....	19
4.6 Creando etiquetas	22
4.7 Deshaciendo commit.....	23
4. Trabajando con repositorios remotos	26
5.1 Mostrando los repositorios remotos	26
6. Trabajando con ramas	29
7. Fuentes de información	37

1. Instalación de Git en plataforma Windows

1.1 Descargando Git

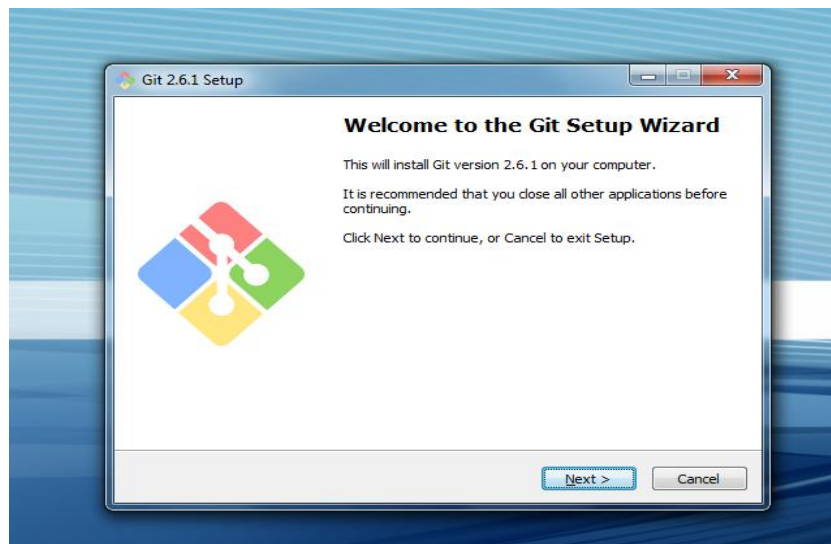
Ya que Windows es más una plataforma para ejecutables es la manera más fácil de descargar el software Git nos dirigimos a la dirección web:

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

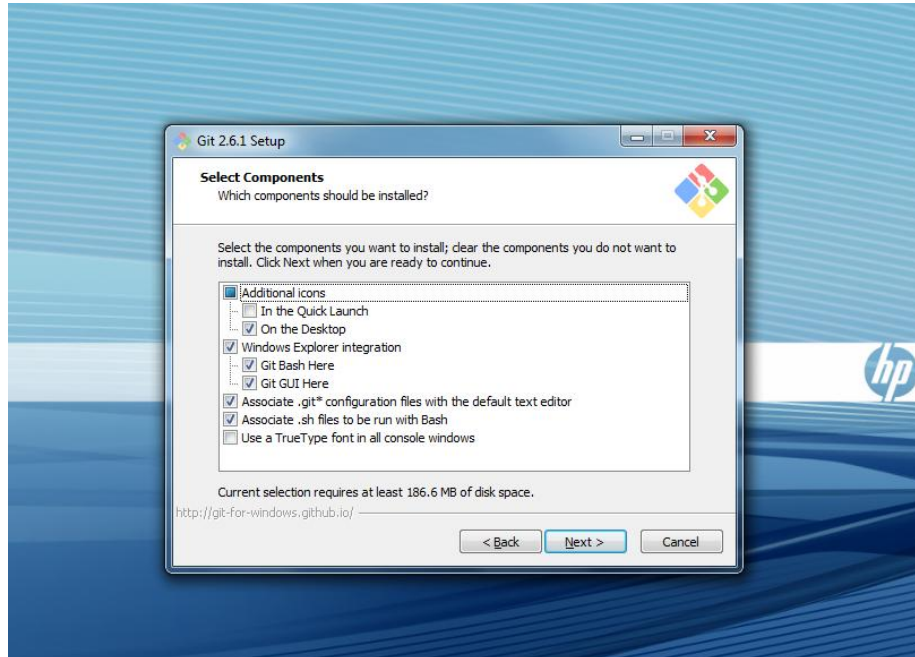


Seleccionamos descargar y al finalizar la descarga del archivo le damos doble click y corremos el ejecutable.

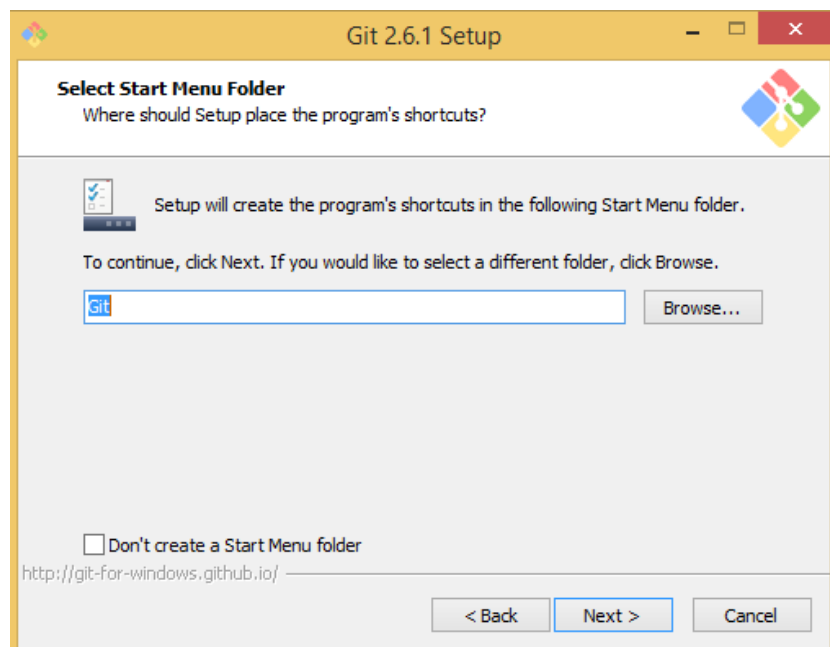
Nos mostrara la siguiente imagen:

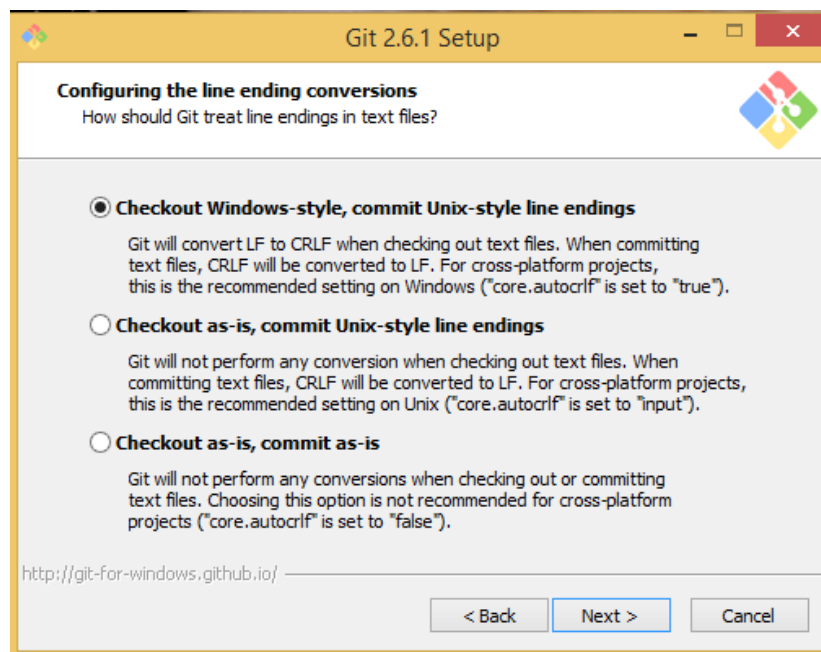
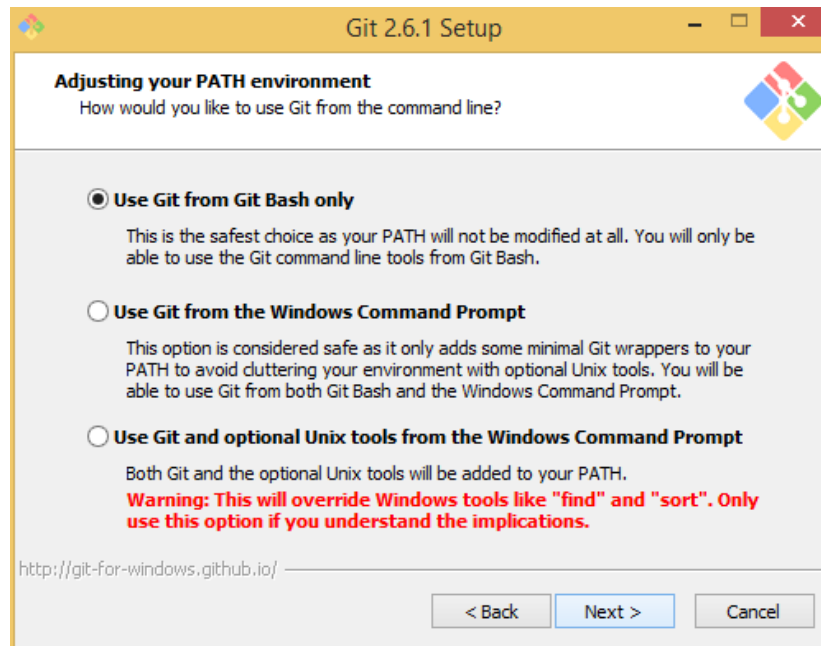


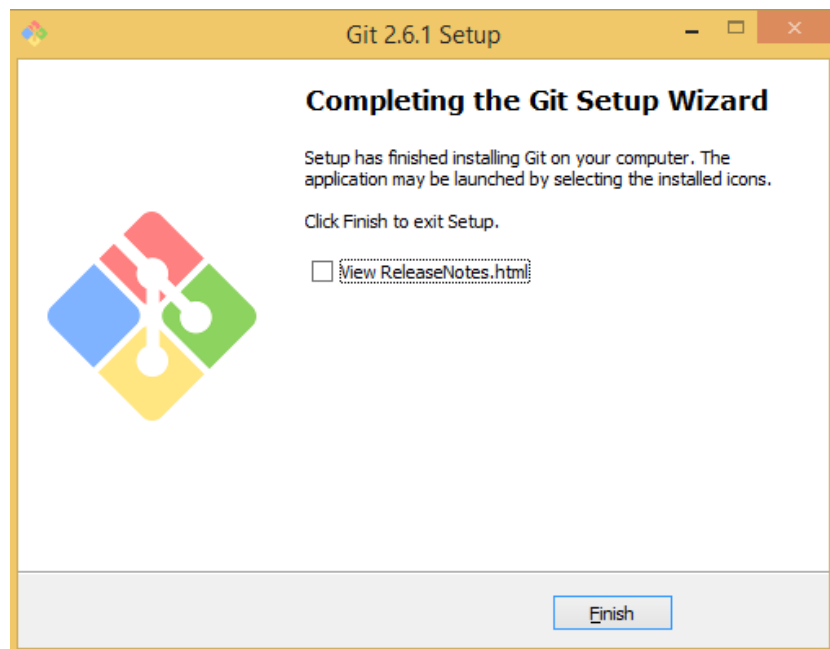
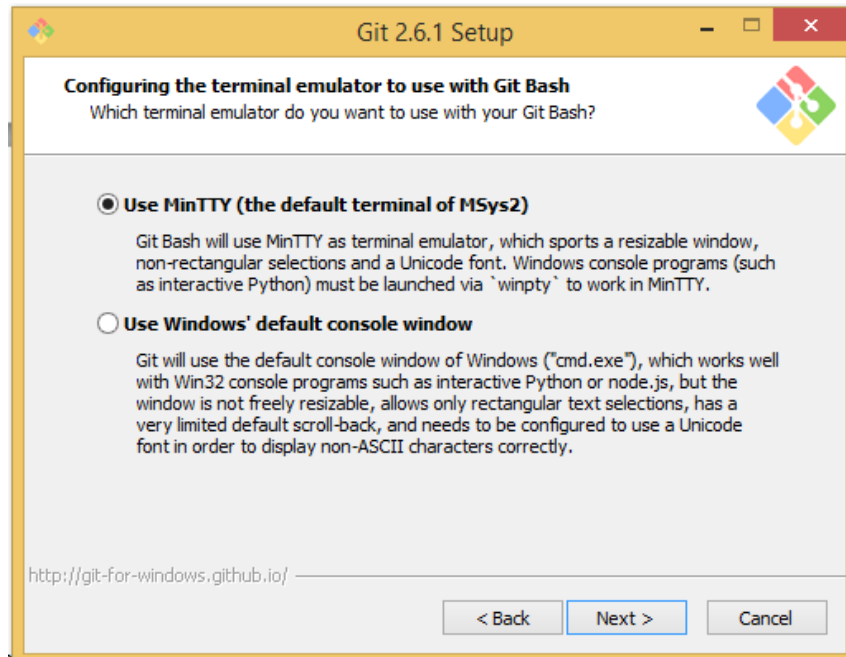
Damos click a siguiente , nos muestra la licencia del producto ya que en este caso aceptaremos los términos solo clickeamos siguiente, donde se nos mostrara una pantalla de los componentes que seleccionaremos, en este caso dejaremos los que ya están por default y si no se encuentra seleccionada pondremos que nos cree un icono en el escritorio



En todos los pasos siguientes en este caso dejaremos las opciones que ya vienen por default y daremos solamente next:







Al finalizar nuestra instalación podemos observar que nuestro icono de git bash se encuentra en el escritorio de windows

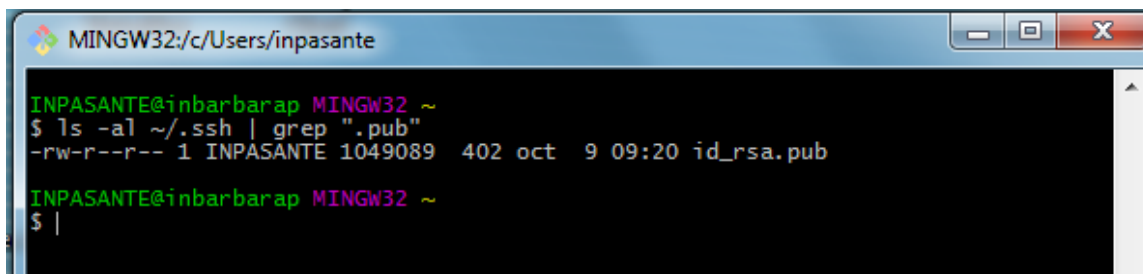
2. Generando tu llave publica ssh

2.1 Verificando si ya tenemos una llave existente

Primero Abrimos nuestra consola de Git dando doble click en el icono del escritorio GitBash

Ingresamos el comando `ls -al ~/.ssh | grep ".pub"` para verificar si tenemos una llave existente, en este caso ya poseemos una por eso nos muestra el resultado del comando en la imagen, pero si no hay ninguna generada, no nos mostrara nada o el siguiente mensaje:

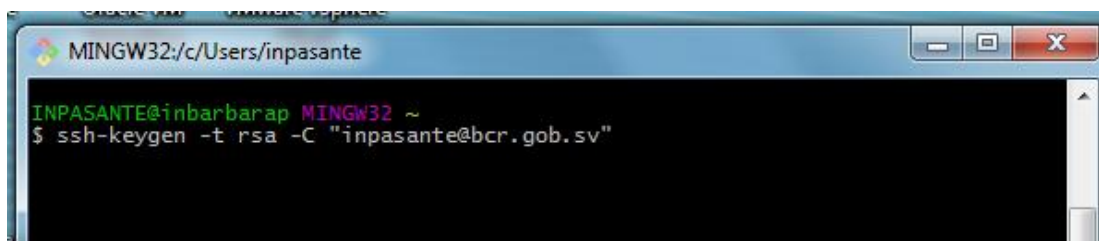
ls: cannot acces /home/fronted/.ssh: No such file or directory



```
MINGW32:/c/Users/inpasante
INPASANTE@inbarbarap MINGW32 ~
$ ls -al ~/.ssh | grep ".pub"
-rw-r--r-- 1 INPASANTE 1049089 402 oct 9 09:20 id_rsa.pub
INPASANTE@inbarbarap MINGW32 ~
$ |
```

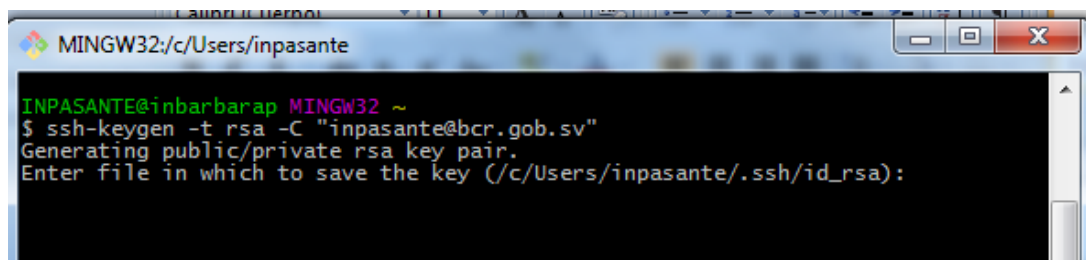
2.2 Generando nueva llave ssh

Si no tenemos la clave como verificamos en el paso anterior, crearemos una nueva, primero hay que ingresar este comando en la consola bash



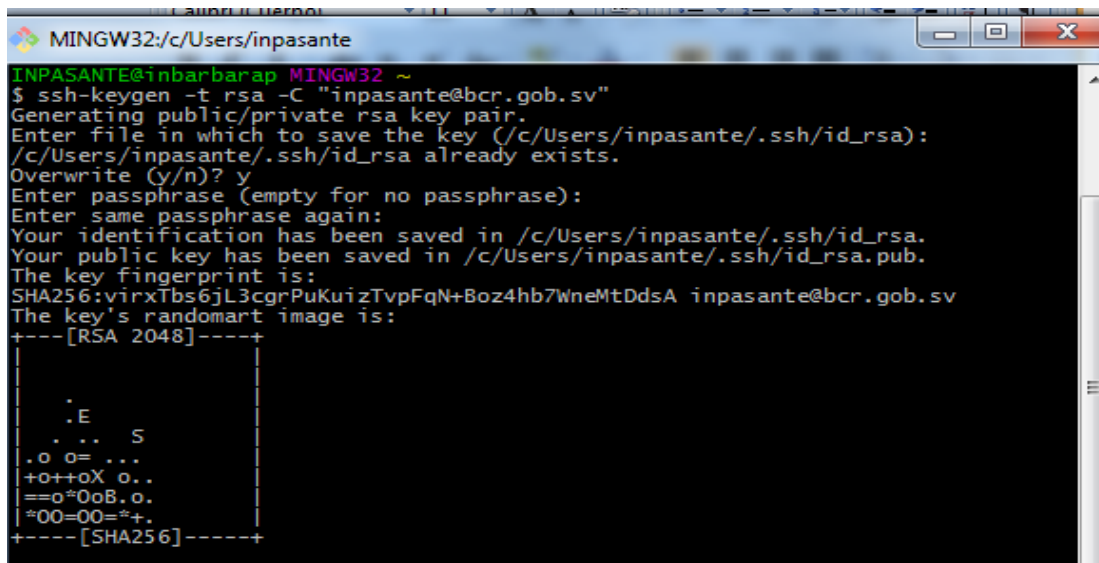
```
MINGW32:/c/Users/inpasante
INPASANTE@inbarbarap MINGW32 ~
$ ssh-keygen -t rsa -C "inpasante@bcr.gob.sv"
```

Al teclear intro empezara a ejecutar la generación de la llave y tendremos un resultado así:



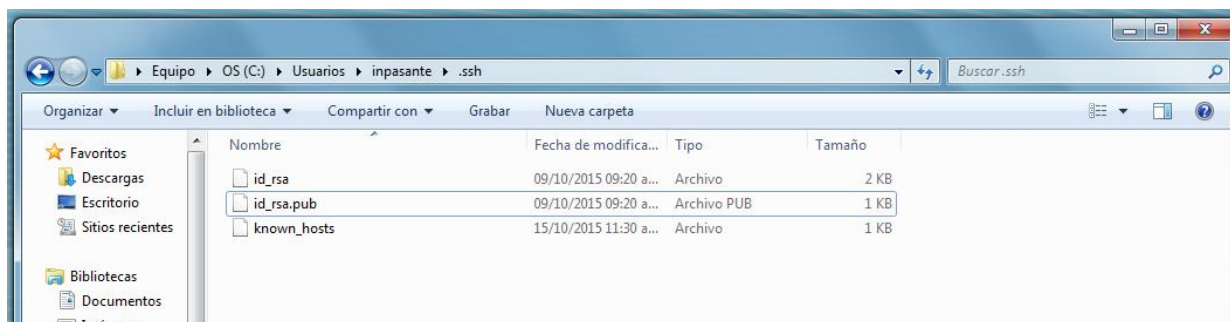
```
MINGW32:/c/Users/inpasante
INPASANTE@inbarbarap MINGW32 ~
$ ssh-keygen -t rsa -C "inpasante@bcr.gob.sv"
Generating public/private rsa key pair.
Enter file in which to save the key (/c/Users/inpasante/.ssh/id_rsa):
```

Nos pedirá una contraseña para la generación de la nueva llave en este caso solo daremos enter



```
MINGW32:/c/Users/inpasante
INPASANTE@inbarbarap MINGW32 ~
$ ssh-keygen -t rsa -C "inpasante@bcr.gob.sv"
Generating public/private rsa key pair.
Enter file in which to save the key (/c/Users/inpasante/.ssh/id_rsa):
/c/Users/inpasante/.ssh/id_rsa already exists.
Overwrite (y/n)? y
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /c/Users/inpasante/.ssh/id_rsa.
Your public key has been saved in /c/Users/inpasante/.ssh/id_rsa.pub.
The key fingerprint is:
SHA256:virxTbs6jL3cgrPuKuizTvpFqN+Boz4hb7WneMtDdsA inpasante@bcr.gob.sv
The key's randomart image is:
+----[RSA 2048]-----+
|
|.E
|..S
|.o o=...
|+o++oX o..
|==o*OoB.o.
|*OO=OO=*+.
+----[SHA256]-----+
```

Como se puede observar en este caso al ya haber una llave existente no pregunta si la queremos eliminar y hacer otra yo le puse si, aunque ya tengo el backup de la llave anterior para reemplazarla después, como se ve nuestra llave ya está generada y a partir de esto se crean dos archivos .ssh/id_rsa y .ssh/id_rsa.pub que se alojan en nuestra carpeta .ssh (por cierto la carpeta .ssh se genera en la carpeta principal del usuario), esta llave servirá para el envío y descarga de archivos del servidor que veremos más adelante, finalmente se verá así nuestra carpeta .ssh:



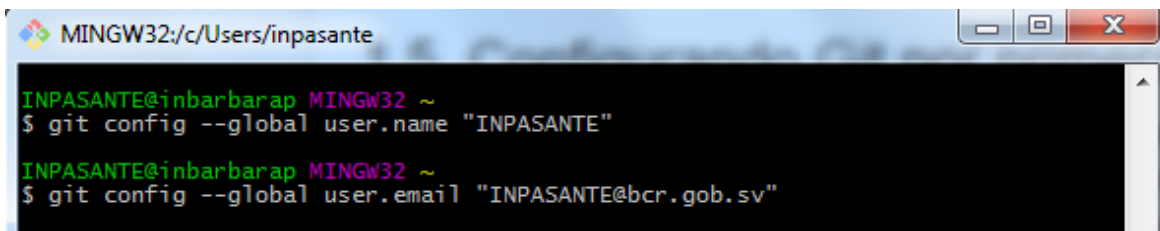
3. Configuración de GIT

Ahora que tienes instalado git en tu sistema, junto con la llave ssh hay algunas cosas que se deben hacer para personalizar tu entorno, git usa una herramienta llamada *git config* que te permite controlar el aspecto y funcionamiento de GIT

En este caso ya que estamos bajo el sistema operativo Windows este archivo se alojara en la carpeta `.git/config` a nivel de cada repositorio y en el archivo `.gitconfig` en la carpeta de usuario actual del sistema

3.1 Tu identidad

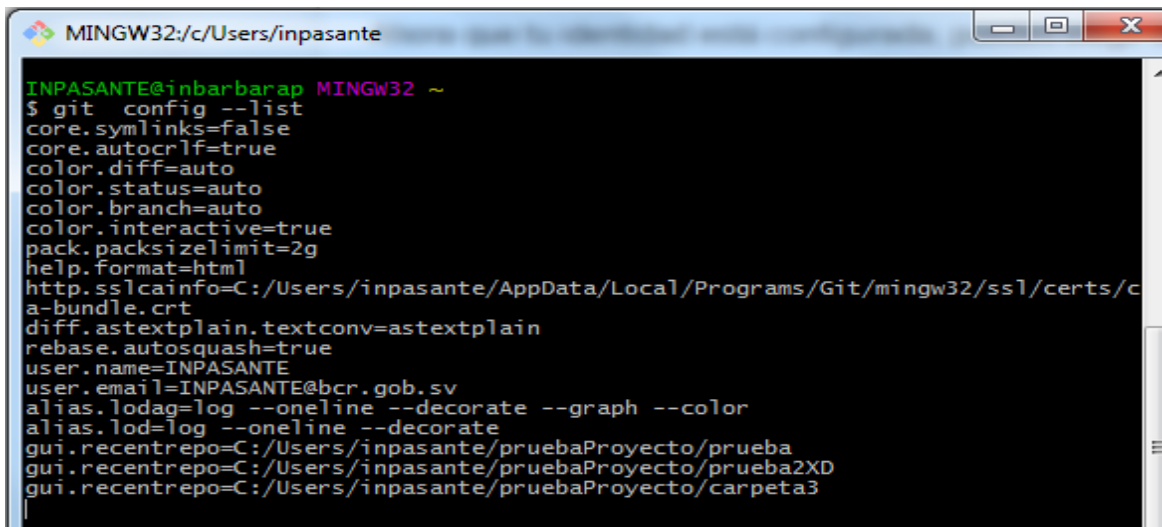
Lo primero que deberías hacer cuando instalas Git es establecer tu nombre de usuario y dirección de correo electrónico. Esto es importante porque las confirmaciones de cambios (*commits*) en Git usan esta información, y es introducida de manera inmutable en los commits que envías:

A screenshot of a Windows command prompt window titled "MINGW32:/c/Users/inpasante". The window shows two commands being executed in a terminal. The first command is `$ git config --global user.name "INPASANTE"` and the second is `$ git config --global user.email "INPASANTE@bcr.gob.sv"`. The prompt shows the user is "INPASANTE" and the current directory is "~".

```
MINGW32:/c/Users/inpasante
INPASANTE@inbarbarap MINGW32 ~
$ git config --global user.name "INPASANTE"
INPASANTE@inbarbarap MINGW32 ~
$ git config --global user.email "INPASANTE@bcr.gob.sv"
```

Puedes cambiar la herramienta de diferencias por defecto para resolver conflictos (merge) comando que se verá más adelante en el tema de ramas, pero en este caso no tocaremos eso ya que el que viene por defecto es vim

Para verificar si todas nuestras configuraciones están correctas escribimos el siguiente comando y su resultado es este:



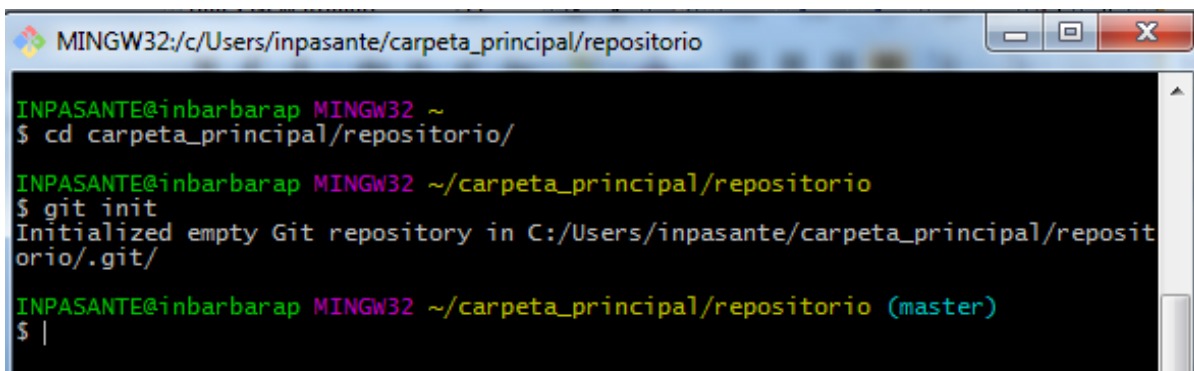
```
MINGW32:/c/Users/inpasante
INPASANTE@inbarbarap MINGW32 ~
$ git config --list
core.symlinks=false
core.autocrlf=true
color.diff=auto
color.status=auto
color.branch=auto
color.interactive=true
pack.packsizelimit=2g
help.format=html
http.sslcainfo=C:/Users/inpasante/AppData/Local/Programs/Git/mingw32/ssl/certs/ca-bundle.crt
diff.astextplain.textconv=astextplain
rebase.autosquash=true
user.name=INPASANTE
user.email=INPASANTE@bcr.gob.sv
alias.lodag=log --oneline --decorate --graph --color
alias.lod=log --oneline --decorate
gui.recentrepo=C:/Users/inpasante/pruebaProyecto/prueba
gui.recentrepo=C:/Users/inpasante/pruebaProyecto/prueba2XD
gui.recentrepo=C:/Users/inpasante/pruebaProyecto/carpeta3
```

En este caso salen unas cosas extras como es dos alias que agregue y tres repositorios existentes, las ultimas 5 líneas.

4. Fundamentos de Git

4.1 Creando repositorios

Como primer paso tenemos que convertir nuestra carpeta en la que hemos elegido para que sea nuestro repositorio, como primer paso es movernos a esa carpeta con el comando `cd`, y luego usar el comando `git init` (como mensaje aparte todo comando que usa git lleva la palabra git como inicio para ejecutarlo)



```
MINGW32:/c/Users/inpasante/carpeta_principal/repositorio
INPASANTE@inbarbarap MINGW32 ~
$ cd carpeta_principal/repositorio/

INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio
$ git init
Initialized empty Git repository in C:/Users/inpasante/carpeta_principal/repositorio/.git/

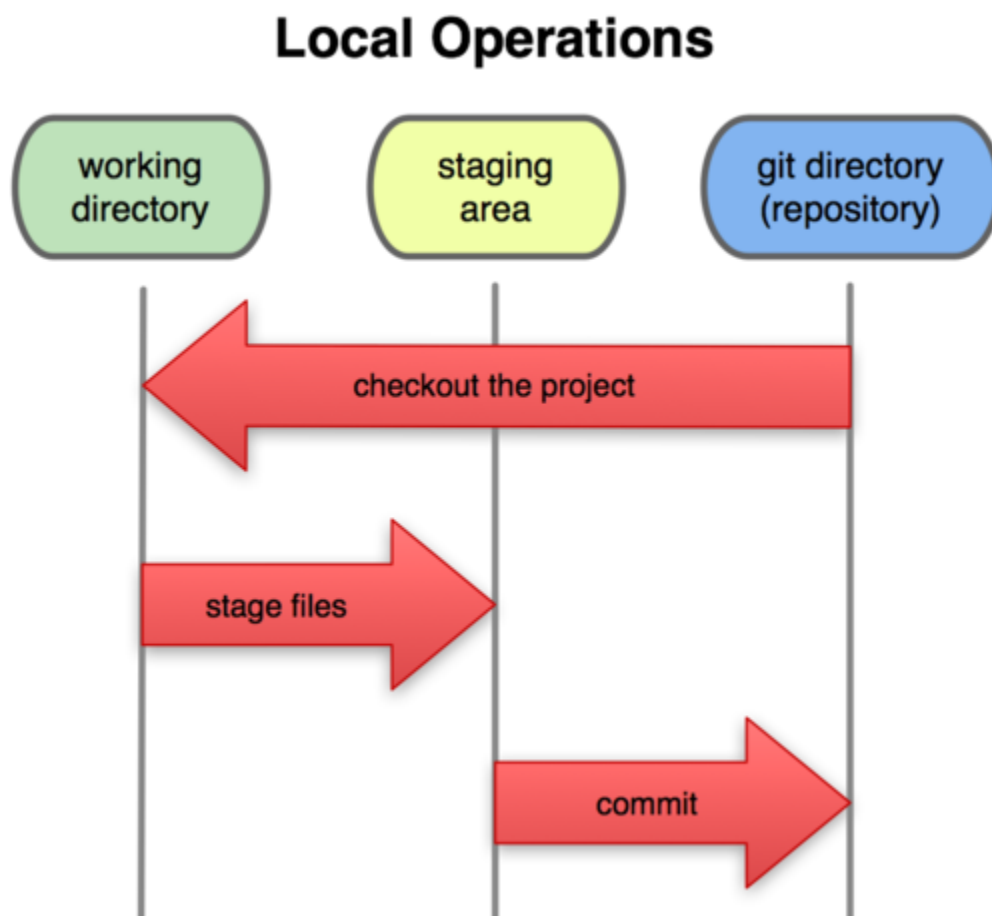
INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (master)
$ |
```

Como se observa al ejecutar el comando `git init` nos muestra un mensaje que esa carpeta actual ya está inicializada como repositorio y la línea de comandos nos muestra al final la palabra `master` que es la rama actual donde nos encontramos y es la rama principal y por defecto de git.

4.2 Agregando archivos y primer Commit

Bueno como primer punto utilizaremos el comando `git status` que sirve para observar el estado de nuestros archivos, Git trabaja con tres estados: confirmado (*committed*), modificado (*modified*), y preparado (*staged*). Confirmado significa que los datos están almacenados de manera segura en tu base de datos local. Modificado significa que has modificado el archivo pero todavía no lo has confirmado a tu base de datos. Preparado 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*), el directorio de trabajo (*working directory*), y el área de preparación (*staging area*).



El directorio de Git es donde Git almacena 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 otro ordenador.

El directorio de trabajo 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 es un sencillo archivo, generalmente contenido en tu directorio de Git, que almacena información acerca de lo que va a ir en tu próxima confirmación. A veces se denomina el índice, pero se está convirtiendo en estándar el referirse a ello como el área de preparación.

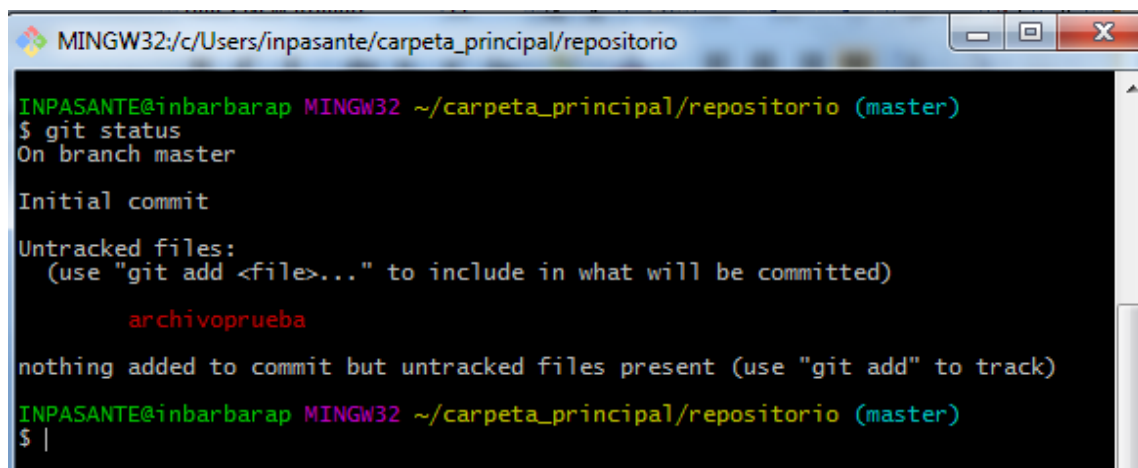
El flujo de trabajo básico en Git es algo así:

1. Modificas una serie de archivos en tu directorio de trabajo.
2. Preparas los archivos, añadiendo instantáneas de ellos a tu área de preparación.
3. Confirmas los cambios, lo que toma los archivos tal y como están en el área de preparación, y almacena esa instantánea de manera permanente en tu directorio de Git.

Si una versión concreta de un archivo está en el directorio de Git, se considera confirmada (*committed*). Si ha sufrido cambios desde que se obtuvo del repositorio, pero ha sido añadida al área de preparación, está preparada (*staged*). Y si ha sufrido cambios desde que se obtuvo del repositorio, pero no se ha preparado, está modificada (*modified*).

Como se verá a continuación usando el *git status* y otros comandos veremos los tres estados de los archivos en GIT

Al tener nuevo archivo y/o una modificación a un archivo y usamos el comando *git status*



```
MINGW32:/c/Users/inpasante/carpeta_principal/repositorio
INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (master)
$ git status
On branch master

Initial commit

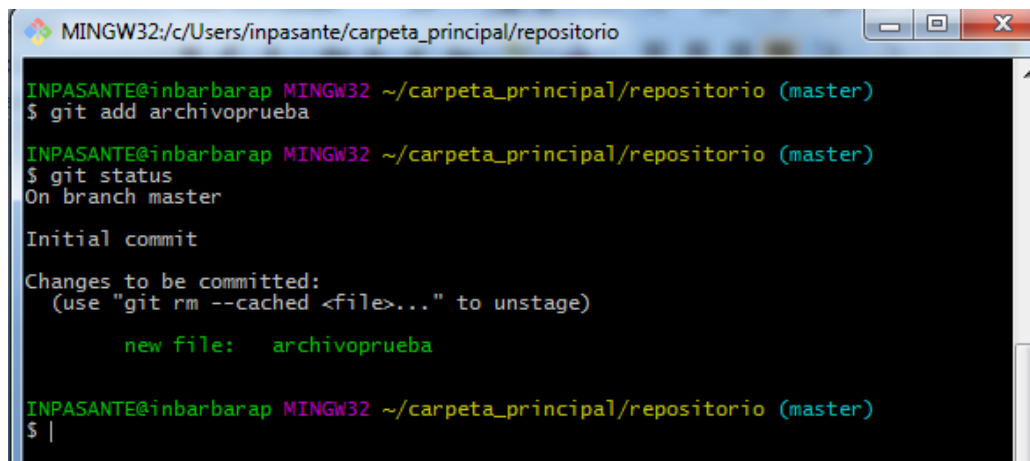
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        archivoprueba

nothing added to commit but untracked files present (use "git add" to track)
INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (master)
$ |
```

Como se puede observar el archivo `archivoprueba` se encuentra en el directorio de trabajo pero no está preparado, se encuentra actualmente en el área de (working directory)

Para agregar archivos al área de preparación (staging area) y de estado *staged*, se utiliza el comando `git add`, este comando se utiliza poniendo el nombre del archivo a “subir” al final o si se quiere varios archivos `git add “*.txt”` (asterisco. extensión de archivos a subir), y si son todos los archivos que se van a subir `git add .` (punto al final del comando) como se muestra en la pantalla.

A screenshot of a Windows command prompt window titled "MINGW32:/c/Users/inpasante/carpeta_principal/repositorio". The prompt shows the user "INPASANTE@inbarbarap" in a "MINGW32" environment at the path "~/carpeta_principal/repositorio" on the "master" branch. The user enters the command `$ git add archivoprueba`. The prompt then shows the command `$ git status`. The output indicates an "Initial commit" on the "master" branch, with "Changes to be committed:" listed as "new file: archivoprueba". The user then enters a pipe character `$ |` at the prompt.

```
MINGW32:/c/Users/inpasante/carpeta_principal/repositorio
INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (master)
$ git add archivoprueba
INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (master)
$ git status
On branch master

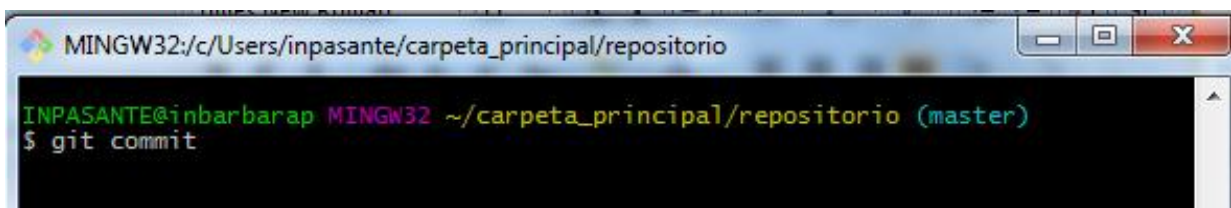
Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

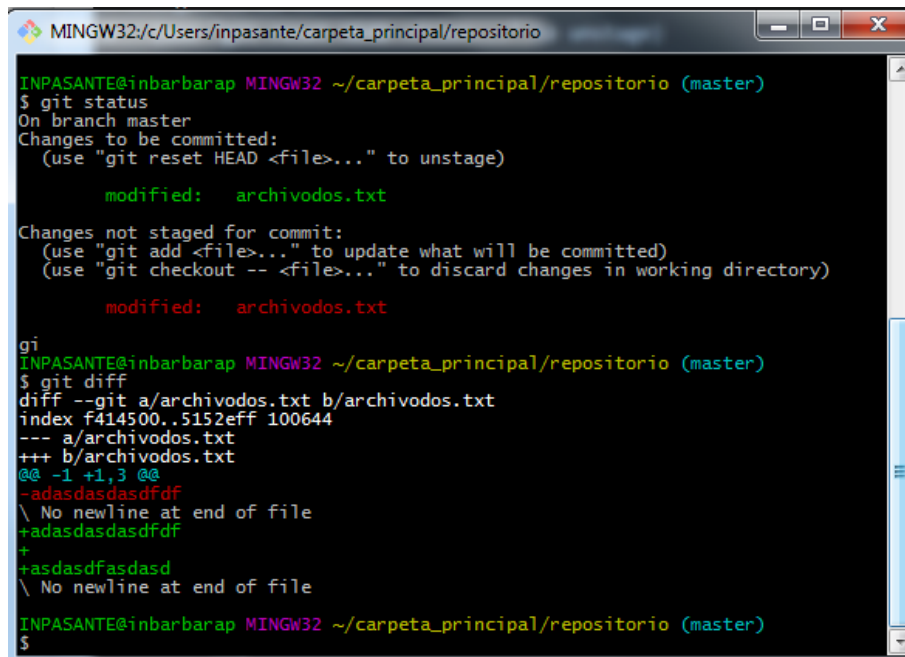
        new file:   archivoprueba

INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (master)
$ |
```

Finalmente para colocar este archivo en estado *committed* y dentro del *git directory*, el comando `git commit`, este comando se puede ingresar así tal cual, pero nos abrirá un archivo con vim en el cual tendremos que colocar una descripción de que se trata este nuevo commit o la opción de colocar el comando así `git commit -m “descripción”` donde `-m` es la opción para ingresar la descripción en vez de que abra el documento en vim

A screenshot of a Windows command prompt window titled "MINGW32:/c/Users/inpasante/carpeta_principal/repositorio". The prompt shows the user "INPASANTE@inbarbarap" in a "MINGW32" environment at the path "~/carpeta_principal/repositorio" on the "master" branch. The user enters the command `$ git commit`.

```
MINGW32:/c/Users/inpasante/carpeta_principal/repositorio
INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (master)
$ git commit
```


A screenshot of a terminal window titled 'MINGW32: c:/Users/inpasante/carpeta_principal/repositorio'. The terminal shows the output of 'git status' and 'git diff' commands. The 'git status' output indicates that 'archivos.txt' is modified and staged for commit. The 'git diff' output shows the differences between the staged version (a/archivos.txt) and the working directory version (b/archivos.txt). The diff shows a deletion of a line and an addition of a line, both containing the text 'adadasdasdfdf'. The terminal prompt is '\$' and the user is 'INPASANTE@inbarbarap'.

```
MINGW32: c:/Users/inpasante/carpeta_principal/repositorio
INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (master)
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   archivos.txt

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

        modified:   archivos.txt

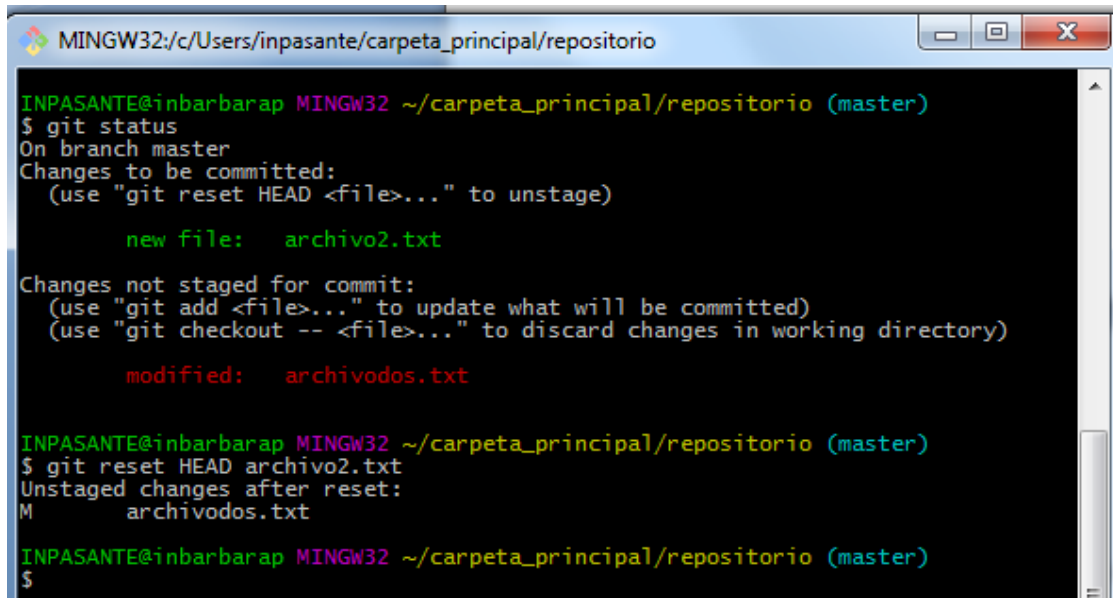
gi
INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (master)
$ git diff
diff --git a/archivos.txt b/archivos.txt
index f414500..5152eff 100644
--- a/archivos.txt
+++ b/archivos.txt
@@ -1,3 @@
-adasdasdasdfdf
\ No newline at end of file
+adasdasdasdfdf
+
+adasdfasdasd
\ No newline at end of file
INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (master)
$
```

Como se puede observar hay dos versiones del mismo archivo uno ya dentro del area de staged y uno fuera ya que acabamos de realizar una modificación y no hemos utilizado nuevamente el comando *git add* y el comando *git diff* nos muestra las diferencias entre los dos archivos las modificaciones que se hicieron

Finalmente si en el último commit nos fijamos que nos faltó una pequeña modificación en un archivo pero tiene que ir ese commit que acabamos de realizar por cuestiones de documentación o alguna razón tenemos el comando *git commit --amend* , claro terminamos de hacer todas las modificaciones necesarias y usamos el *git add* . para ingresarlos al staged area pero en vez de realizar un commit normal usamos el comando *git commit --amend* todo lo que acabamos de modificar quedara grabado en el último commit realizado

4.3 Eliminando archivos

Si se tiene un nuevo archivo y no se agregó con el comando *git add* este archivo se puede eliminar sin mayor dificultad según git ya que nunca existió en su respaldo, pero si el archivo ya fue agregado con el *git add* y se dio cuenta que ha sido un error subir ese archivo se elimina de esta manera con el comando *git reset HEAD (nombre del archivo)*

A screenshot of a terminal window titled 'MINGW32:/c/Users/inpasante/carpeta_principal/repositorio'. The terminal shows the output of 'git status' and the execution of 'git reset HEAD archivo2.txt'.

```
MINGW32:/c/Users/inpasante/carpeta_principal/repositorio
INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (master)
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   archivo2.txt

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

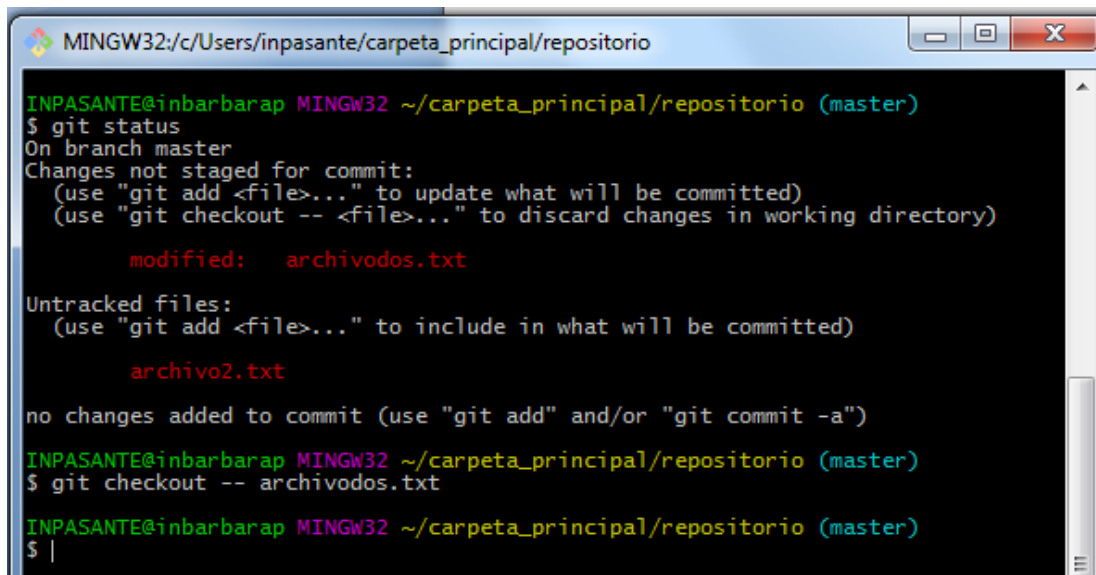
    modified:   archivodos.txt

INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (master)
$ git reset HEAD archivo2.txt
Unstaged changes after reset:
M   archivodos.txt

INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (master)
$
```

Como se puede observar el archivo *archivo2.txt* ya está en estado staged de color verde el mismo git nos dice que si queremos regresar a ese archivo a su estado anterior en el directorio de trabajo debemos usar el comando mencionado arriba, al ejecutar este comando vemos como la consola nos dice que el comando seleccionado *archivo2.txt* ya está fuera de estado staged y está el directorio de trabajo

Como se puede observar también en la pantalla anterior git nos muestra que hay un archivo modificado pero que está en el area de directorio de trabajo pero nos da la opción de regresar ese archivo a como estaba antes de la modificación con el comando *git checkout -(nombre archivo)* y siempre alojado en el directorio de trabajo

A terminal window titled 'MINGW32:/c/Users/inpasante/carpeta_principal/repositorio' showing the output of 'git status' and 'git checkout -- archivodos.txt'. The status shows 'archivodos.txt' as modified and 'archivo2.txt' as untracked. The checkout command is executed successfully.

```
MINGW32:/c/Users/inpasante/carpeta_principal/repositorio
INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (master)
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   archivodos.txt

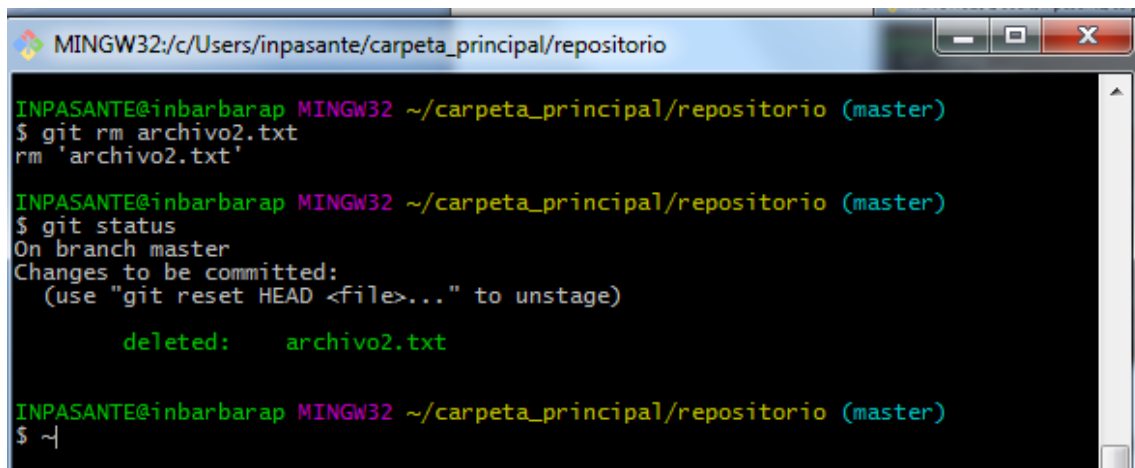
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        archivo2.txt

no changes added to commit (use "git add" and/or "git commit -a")
INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (master)
$ git checkout -- archivodos.txt
INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (master)
$ |
```

Al ejecutar el comando *git checkout* nos regresa el archivo a su contenido anterior a la modificación y siempre en el directorio de trabajo sin ser agregado

Si el archivo que queremos borrar ya lo hemos comiteado y está dentro del directorio de git se tiene que ejecutar el comando *git rm (nombrearchivo)*, este comando deja el archivo que deseamos borrar en el estado de staged lo podemos observar dando *git status*

A terminal window titled 'MINGW32:/c/Users/inpasante/carpeta_principal/repositorio' showing the execution of 'git rm archivo2.txt' and the subsequent 'git status' output. The status shows 'archivo2.txt' as deleted and staged for commit.

```
MINGW32:/c/Users/inpasante/carpeta_principal/repositorio
INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (master)
$ git rm archivo2.txt
rm 'archivo2.txt'

INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (master)
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

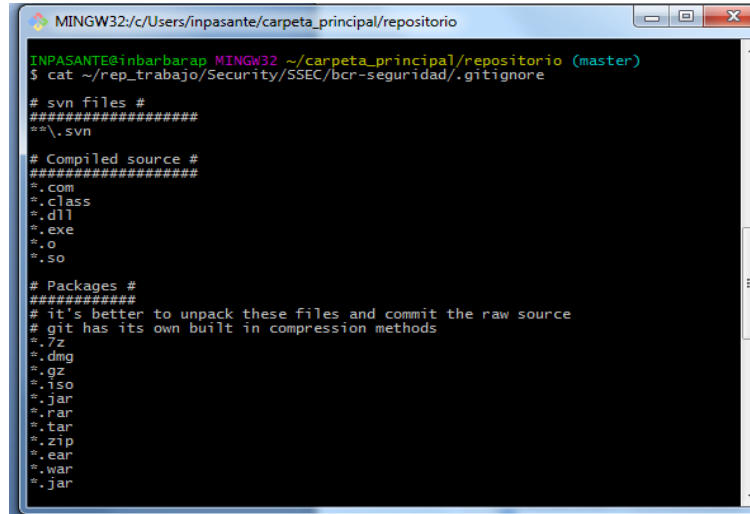
        deleted:   archivo2.txt

INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (master)
$ ~|
```

Como se puede observar el resultado del comando *git rm* borro el archivo y lo dejo en estado staged luego se puede borrar completamente de las maneras que vimos anteriormente

Una manera de poder obviar archivos que no queremos que se agreguen al repositorio como archivos log o archivos que creen por el compilador, es creando un archivo .gitignore, el cual se coloca en la carpeta principal del repositorio

Ejemplo:

A screenshot of a terminal window titled "MINGW32/c/Users/inpasante/carpeta_principal/repositorio". The prompt is "INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (master)". The command executed is "\$ cat ~/rep_trabajo/Security/SSEC/bcr-seguridad/.gitignore". The output shows the contents of the .gitignore file, which includes comments and patterns for ignoring files.

```
INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (master)
$ cat ~/rep_trabajo/Security/SSEC/bcr-seguridad/.gitignore

# svn files #
#####
**\svn

# Compiled source #
#####
*.com
*.class
*.dll
*.exe
*.o
*.so

# Packages #
#####
# it's better to unpack these files and commit the raw source
# git has its own built in compression methods
*.7z
*.dmg
*.gz
*.iso
*.jar
*.rar
*.tar
*.zip
*.ear
*.war
*.jar
```

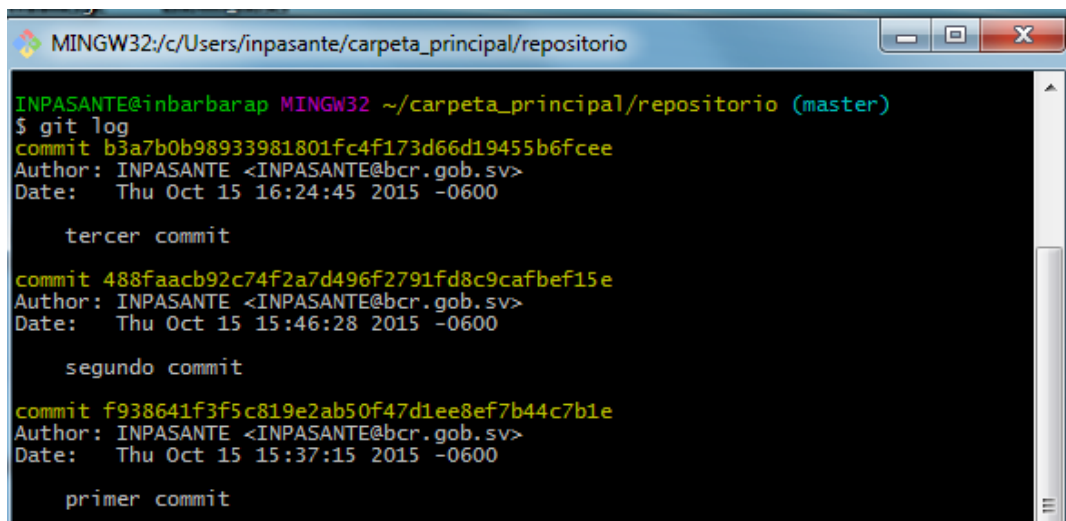
Las reglas para los patrones que pueden ser incluidos en el archivo .gitignore son:

- Las líneas en blanco, o que comienzan por #, son ignoradas.
- Puedes usar patrones glob estándar.
- Puedes indicar un directorio añadiendo una barra hacia delante (/) al final.
- Puedes negar un patrón añadiendo una exclamación (!) al principio.

Los patrones glob son expresiones regulares simplificadas que pueden ser usadas por las shells. Un asterisco (*) reconoce cero o más caracteres; [abc] reconoce cualquier carácter de los especificados entre corchetes; una interrogación (?) reconoce un único carácter; y caracteres entre corchetes separados por un guion ([0-9]) reconoce cualquier carácter entre ellos.

4.4 Viendo histórico de confirmaciones

Después de haber hecho varias confirmaciones(commit) o clonado repositorios que ya tenía un histórico de confirmaciones, para poder ver todos estos commit realizados se utiliza el comando *git log*.



```
MINGW32:/c/Users/inpasante/carpeta_principal/repositorio
INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (master)
$ git log
commit b3a7b0b98933981801fc4f173d66d19455b6fcee
Author: INPASANTE <INPASANTE@bcr.gob.sv>
Date: Thu Oct 15 16:24:45 2015 -0600

    tercer commit

commit 488faacb92c74f2a7d496f2791fd8c9cafbef15e
Author: INPASANTE <INPASANTE@bcr.gob.sv>
Date: Thu Oct 15 15:46:28 2015 -0600

    segundo commit

commit f938641f3f5c819e2ab50f47d1ee8ef7b44c7b1e
Author: INPASANTE <INPASANTE@bcr.gob.sv>
Date: Thu Oct 15 15:37:15 2015 -0600

    primer commit
```

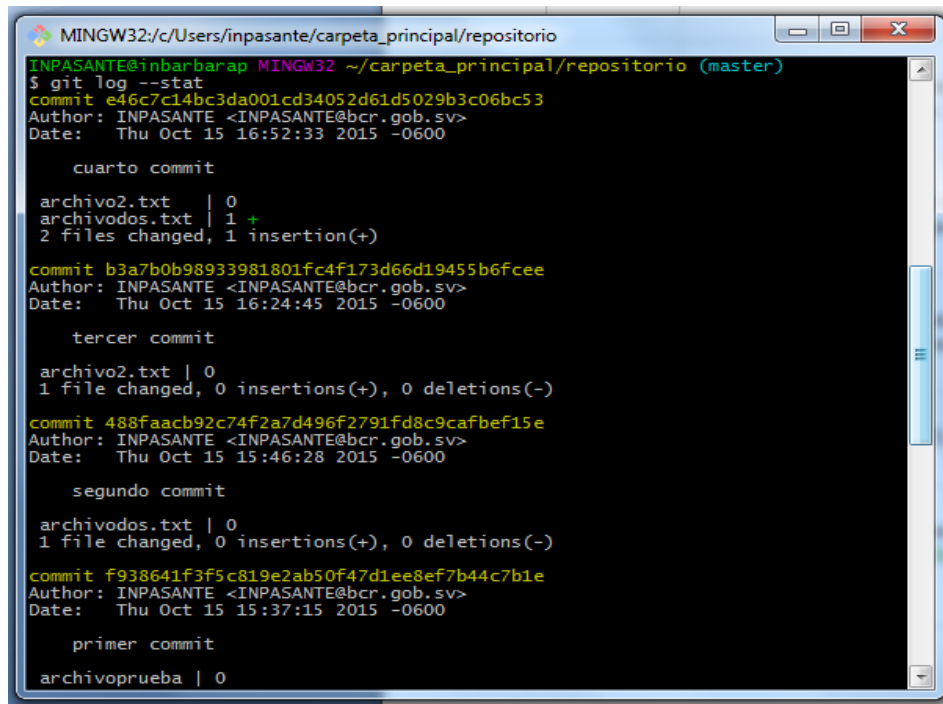
Como se puede ver al ejecutar *git log* se muestra cierta información de los commit como se puede ver en las líneas amarillas cada commit se identifica con un número de identificador único creado con SHA-1, además de mostrar el autor y la fecha que fue realizado, con el comentario puesto en la opción *-m* del commit

El comando *git log* tiene varias opciones como se muestran en la siguiente tabla:

Opción	Descripción
<i>-p</i>	Muestra el parche introducido en cada confirmación.
<i>--stat</i>	Muestra estadísticas sobre los archivos modificados en cada confirmación.
<i>--shortstat</i>	Muestra solamente la línea de resumen de la opción <i>--stat</i> .

Opción	Descripción
<code>--name-only</code>	Muestra la lista de archivos afectados.
<code>--name-status</code>	Muestra la lista de archivos afectados, indicando además si fueron añadidos, modificados o eliminados.
<code>--abbrev-commit</code>	Muestra solamente los primeros caracteres de la suma SHA-1, en vez de los 40 caracteres de que se compone.
<code>--relative-date</code>	Muestra la fecha en formato relativo (por ejemplo, “2 weeks ago” (“hace 2 semanas”)) en lugar del formato completo.
<code>--graph</code>	Muestra un gráfico ASCII con la historia de ramificaciones y uniones.
<code>--pretty</code>	Muestra las confirmaciones usando un formato alternativo. Posibles opciones son <code>oneline</code> , <code>short</code> , <code>full</code> , <code>fuller</code> , y <code>format</code> (mediante el cual puedes especificar tu propio formato).

Como se muestra en las siguientes pantallas algunos ejemplos



```
MINGW32:/c/Users/inpasante/carpeta_principal/repositorio
INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (master)
$ git log --stat
commit e46c7c14bc3da001cd34052d61d5029b3c06bc53
Author: INPASANTE <INPASANTE@bcr.gob.sv>
Date: Thu Oct 15 16:52:33 2015 -0600

    cuarto commit

    archivo2.txt | 0
    archivosdos.txt | 1 +
    2 files changed, 1 insertion(+)

commit b3a7b0b98933981801fc4f173d66d19455b6fcee
Author: INPASANTE <INPASANTE@bcr.gob.sv>
Date: Thu Oct 15 16:24:45 2015 -0600

    tercer commit

    archivo2.txt | 0
    1 file changed, 0 insertions(+), 0 deletions(-)

commit 488faacb92c74f2a7d496f2791fd8c9cafbef15e
Author: INPASANTE <INPASANTE@bcr.gob.sv>
Date: Thu Oct 15 15:46:28 2015 -0600

    segundo commit

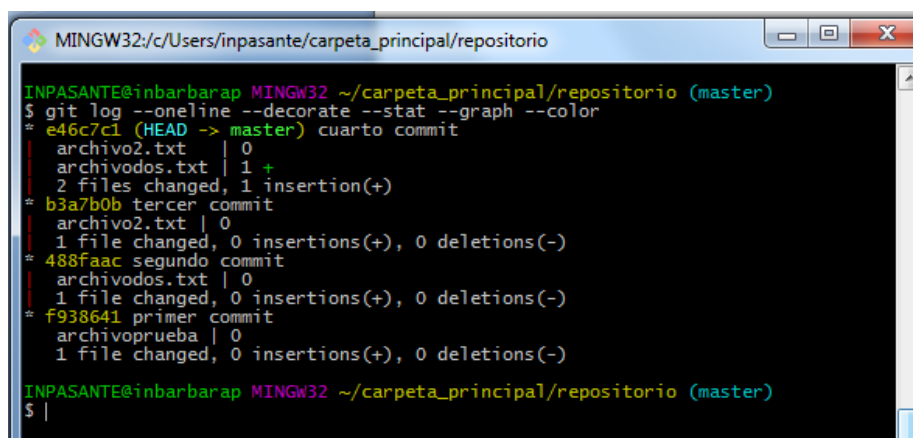
    archivosdos.txt | 0
    1 file changed, 0 insertions(+), 0 deletions(-)

commit f938641f3f5c819e2ab50f47d1ee8ef7b44c7b1e
Author: INPASANTE <INPASANTE@bcr.gob.sv>
Date: Thu Oct 15 15:37:15 2015 -0600

    primer commit

    archivoprueba | 0
```

Además estos comandos se pueden combinar para mostrar de una manera más entendible para el usuario:



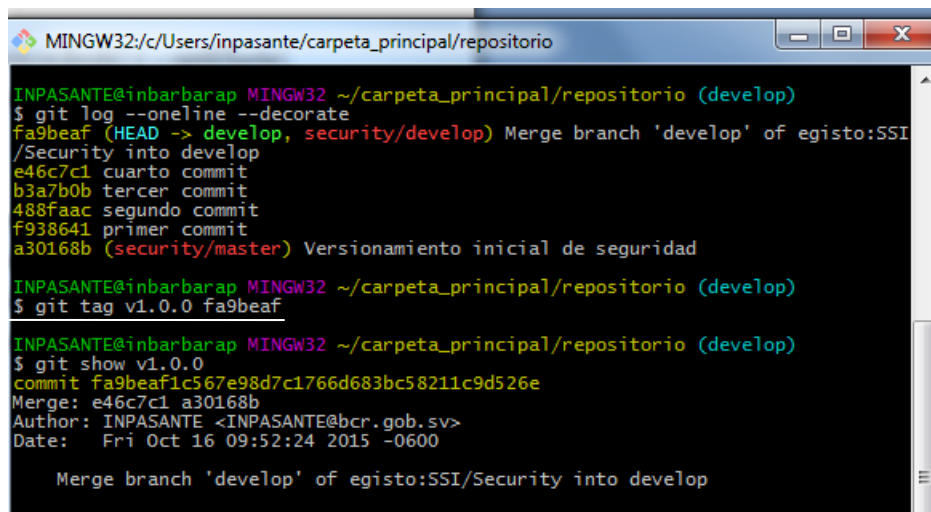
```
MINGW32:/c/Users/inpasante/carpeta_principal/repositorio
INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (master)
$ git log --oneline --decorate --stat --graph --color
* e46c7c1 (HEAD -> master) cuarto commit
  archivo2.txt | 0
  archivosdos.txt | 1 +
  2 files changed, 1 insertion(+)
* b3a7b0b tercer commit
  archivo2.txt | 0
  1 file changed, 0 insertions(+), 0 deletions(-)
* 488faac segundo commit
  archivosdos.txt | 0
  1 file changed, 0 insertions(+), 0 deletions(-)
* f938641 primer commit
  archivoprueba | 0
  1 file changed, 0 insertions(+), 0 deletions(-)
INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (master)
$ |
```

En este ejemplo usamos el `--oneline` para que muestre en una sola línea, `--decorate` para observar que rama hizo los commit, el `--stat` muestra los cambios hechos en cada commit, `--graph` para que me muestre el grafico de cómo se han ido realizando los commits y `--color` para darle color a esos gráficos

Además para limitar las salidas del log hay varias opciones como `--since(desde)` y `--until(hasta)`, como las opciones `-author(búsqueda por author)` `-grep(búsqueda de información en los mensajes)`

4.6 Creando etiquetas

Git tiene la facultad de crear etiquetas para eventos importantes estas etiquetas se aplican a los commit que vamos realizando, la manera de crear etiquetas es con el comando `git tag`, de la siguiente forma `git tag (nombre del tag) (hash del commit)`, en git existen 2 tipos de git etiquetas ligeras y etiquetas anotadas eso depende de las opciones que se coloque con el `git tag`.



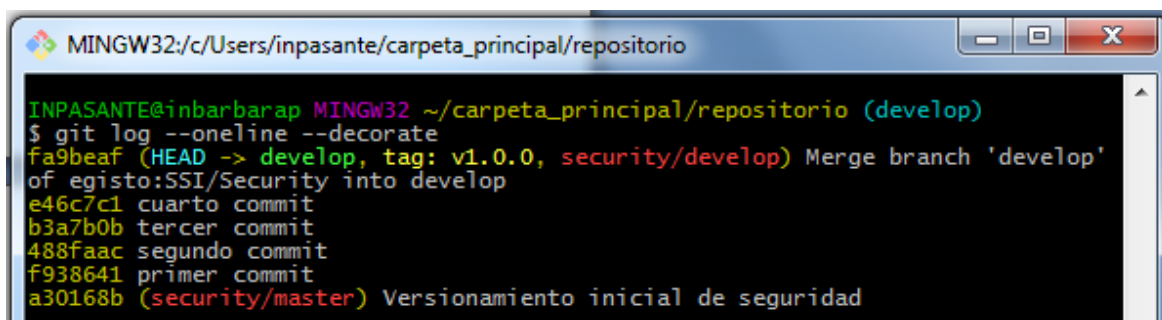
```
MINGW32:/c/Users/inpasante/carpeta_principal/repositorio
INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (develop)
$ git log --oneline --decorate
fa9beaf (HEAD -> develop, security/develop) Merge branch 'develop' of egisto:SSI
/Security into develop
e46c7c1 cuarto commit
b3a7b0b tercer commit
488faac segundo commit
f938641 primer commit
a30168b (security/master) Versionamiento inicial de seguridad

INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (develop)
$ git tag v1.0.0 fa9beaf

INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (develop)
$ git show v1.0.0
commit fa9beaf1c567e98d7c1766d683bc58211c9d526e
Merge: e46c7c1 a30168b
Author: INPASANTE <INPASANTE@bcr.gob.sv>
Date: Fri Oct 16 09:52:24 2015 -0600

Merge branch 'develop' of egisto:SSI/Security into develop
```

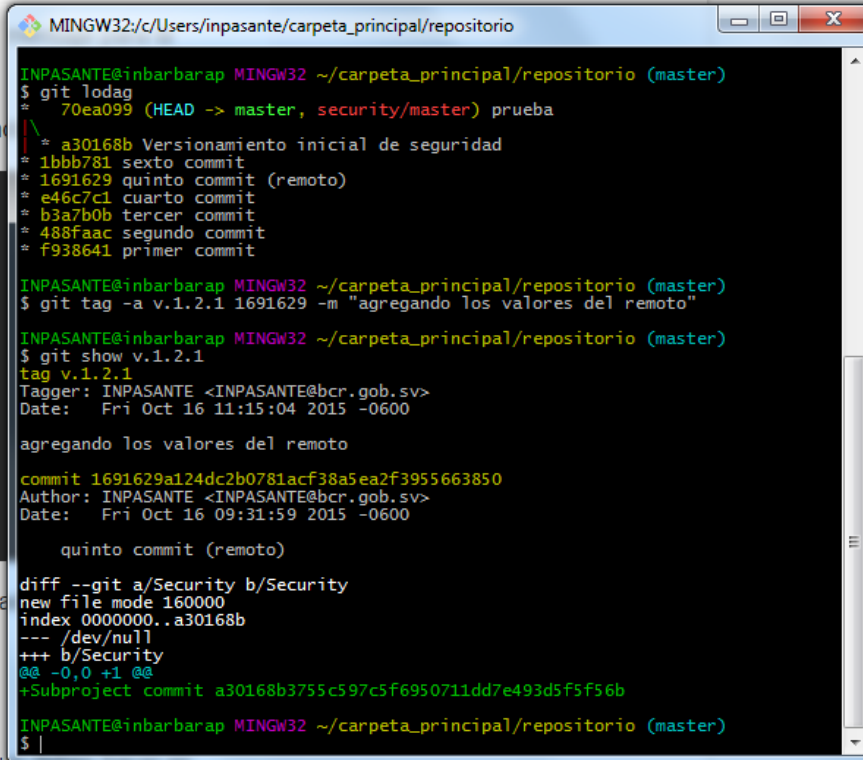
Como vemos usamos el `git log` para observar los commit hechos y el commit que necesitamos un tag y usamos el comando `git tag (nombre tag)(hash commit)`, y como podemos ver podemos usar el `git show` con el tag que acabamos de crear para ver la información del commit y como se verá abajo haciendo un nuevo `git log` se ve que ya tiene asignado su tag



```
MINGW32:/c/Users/inpasante/carpeta_principal/repositorio
INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (develop)
$ git log --oneline --decorate
fa9beaf (HEAD -> develop, tag: v1.0.0, security/develop) Merge branch 'develop'
of egisto:SSI/Security into develop
e46c7c1 cuarto commit
b3a7b0b tercer commit
488faac segundo commit
f938641 primer commit
a30168b (security/master) Versionamiento inicial de seguridad
```

Aclarando el commit que de `security/master` es el merge que se uso al unir la rama remota con la rama local que teníamos, estos conceptos se aclaran más en el tema de ramas

Para los tags anotadas que lo se agrega una descripción del tag con la opción `-a` pero para agregar esos mensajes se necesita también `-m` al igual que los commit



```
INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (master)
$ git log
* 70ea099 (HEAD -> master, security/master) prueba
|
| * a30168b Versionamiento inicial de seguridad
| * 1bbb781 sexto commit
| * 1691629 quinto commit (remoto)
| * e46c7c1 cuarto commit
| * b3a7b0b tercer commit
| * 488faac segundo commit
| * f938641 primer commit
|
INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (master)
$ git tag -a v.1.2.1 1691629 -m "agregando los valores del remoto"

INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (master)
$ git show v.1.2.1
tag v.1.2.1
Tagger: INPASANTE <INPASANTE@bcr.gob.sv>
Date: Fri Oct 16 11:15:04 2015 -0600

agregando los valores del remoto

commit 1691629a124dc2b0781acf38a5ea2f3955663850
Author: INPASANTE <INPASANTE@bcr.gob.sv>
Date: Fri Oct 16 09:31:59 2015 -0600

    quinto commit (remoto)

diff --git a/Security b/Security
new file mode 160000
index 0000000..a30168b
--- /dev/null
+++ b/Security
@@ -0,0 +1 @@
+Subproject commit a30168b3755c597c5f6950711dd7e493d5f5f56b

INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (master)
$ |
```

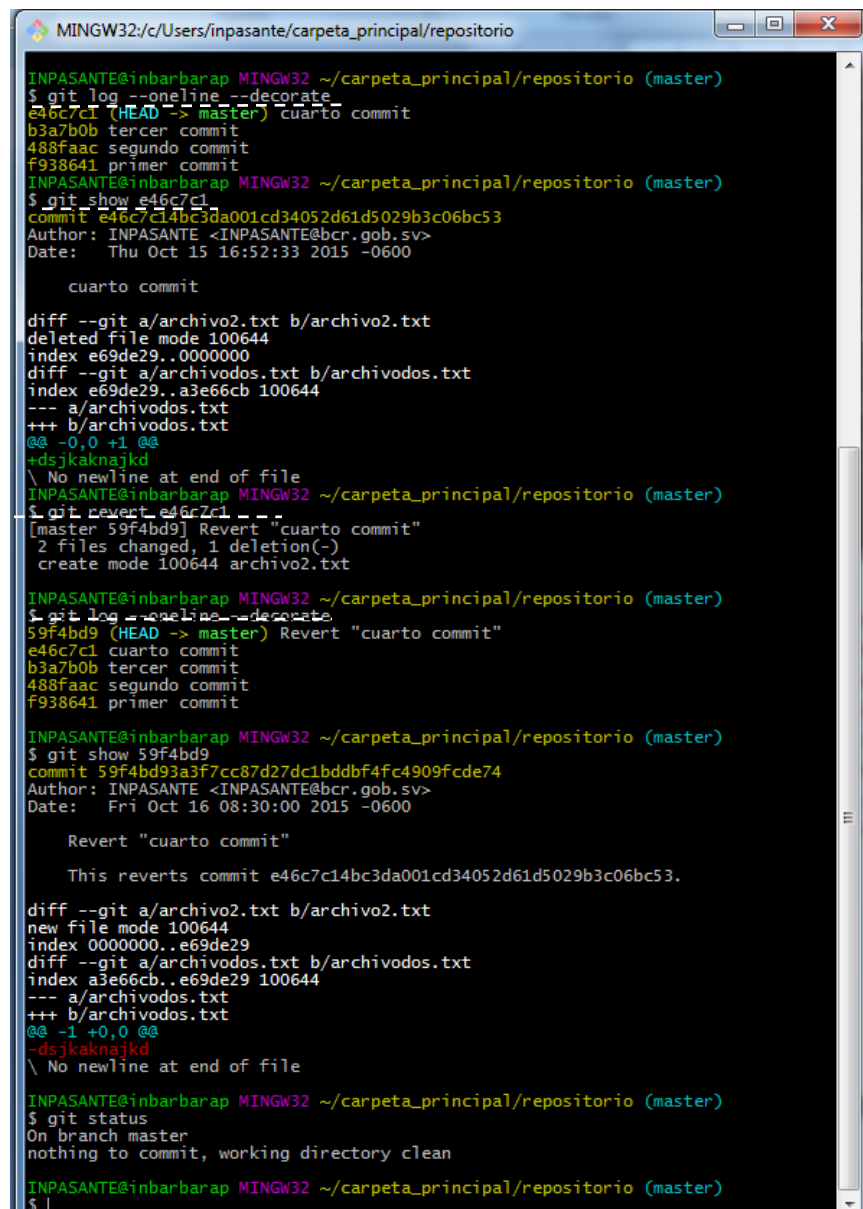
En este caso observamos al hacer el `git log` (ese comando es un alias) vemos los commit y sus hashes y ejecutamos el comando `git tag` como se menciona arriba y creamos nuestra etiqueta anotada con la descripción que queríamos

4.7 Deshaciendo commit

Como hemos visto anteriormente en la sección de eliminando archivos se pueden eliminar un archivo que ha sido commitiado pero que sucede que se realizó un commit que no era necesario o que el commit realizado se fueron muchas modificaciones que no eran necesarias, o en un caso extremo que nos dimos cuenta varios commit después que nos faltó una parte de la modificación y ya teníamos mucho trabajo realizado, pues para este caso git tiene dos comandos el comando *git reset* y el comando *git revert*

Ambos comandos funcionan colocando el comando *git reset (hash del commit)*, *git revert(hash del commit)*, poseen la misma funcionalidad pero la diferencia entre ellos es que `git reset` es un comando destructivo que del commit seleccionado por el hash, todos los commit siguientes se

eliminan, por lo cual no se recomienda usar si estas en un ambiente de red, solo a manera local en cambio el comando `git revert` funciona creando otro commit que se encuentra al comienzo pero deshaciendo los cambios del commit que se selecciono, por supuesto para ver los hash de cada commit utilizamos el comando `git log` que vimos anteriormente, algo que debemos saber que `reset` no funcionan a menos que no sea en la línea principal ya que las líneas secundarias se crean al hacer un merge con otra rama y esos commit de la rama secundaria vienen anclados a la línea principal que se genero en la otra rama



```
MINGW32:/c/Users/inpasante/carpeta_principal/repositorio
INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (master)
$ git log --oneline --decorate
e46c7c1 (HEAD -> master) cuarto commit
b3a7b0b tercer commit
488faac segundo commit
f938641 primer commit
INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (master)
$ git show e46c7c1
commit e46c7c14bc3da001cd34052d61d5029b3c06bc53
Author: INPASANTE <INPASANTE@bcr.gob.sv>
Date: Thu Oct 15 16:52:33 2015 -0600

    cuarto commit

diff --git a/archivo2.txt b/archivo2.txt
deleted file mode 100644
index e69de29..0000000
diff --git a/archivos.txt b/archivos.txt
index e69de29..a3e66cb 100644
--- a/archivos.txt
+++ b/archivos.txt
@@ -0,0 +1 @@
+dsjkaknajakd
\ No newline at end of file
INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (master)
$ git revert e46c7c1
[master 59f4bd9] Revert "cuarto commit"
2 files changed, 1 deletion(-)
create mode 100644 archivo2.txt

INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (master)
$ git log --oneline --decorate
59f4bd9 (HEAD -> master) Revert "cuarto commit"
e46c7c1 cuarto commit
b3a7b0b tercer commit
488faac segundo commit
f938641 primer commit

INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (master)
$ git show 59f4bd9
commit 59f4bd93a3f7cc87d27dc1bddbf4fc4909fcde74
Author: INPASANTE <INPASANTE@bcr.gob.sv>
Date: Fri Oct 16 08:30:00 2015 -0600

    Revert "cuarto commit"

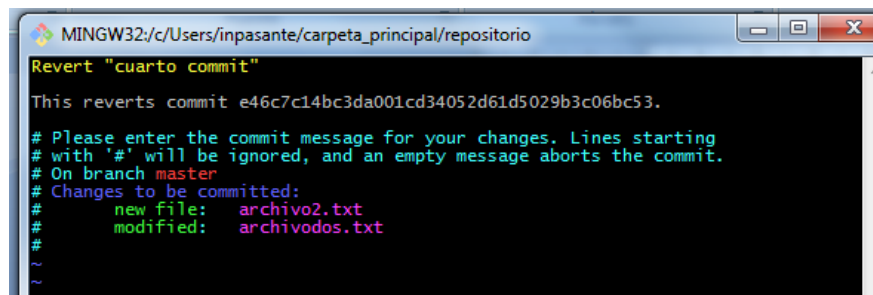
    This reverts commit e46c7c14bc3da001cd34052d61d5029b3c06bc53.

diff --git a/archivo2.txt b/archivo2.txt
new file mode 100644
index 0000000..e69de29
diff --git a/archivos.txt b/archivos.txt
index a3e66cb..e69de29 100644
--- a/archivos.txt
+++ b/archivos.txt
@@ -1,0 +0,0 @@
-dsjkaknajakd
\ No newline at end of file

INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (master)
$ git status
On branch master
nothing to commit, working directory clean

INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (master)
$
```


En la siguiente captura de pantalla podemos observar como es este proceso de usar revert, primero comenzamos con el comando log para ver los diferentes commit realizados y sus hash, usamos el comando git show que sirve para ver lo que se realizo en el commit seleccionado siempre con su hash, como se puede observar se agrego una línea al archivo archivodos.txt y se borro el archivo archivo2.txt, luego ejecutamos el revert al ejecutarlo nos muestra la pantalla que está abajo para agregar un comentario al nuevo comit que se genera regresando a un estado anterior los archivos que se utilizaron en ese commit, guardando el archivo regresamos a la consola principal donde nos muestra como resultado lo que se realizo, volvemos a hacer un git log para ver nuestra actividad y vemos que se creó un nuevo commit llamado revert “cuarto commit” que es la ejecución del revert y al observar con git show el contenido de este vemos que las modificaciones que hicimos en el commit anterior ya no están y al hacer git status vemos que nuestros archivos siguen dentro del directorio de trabajo de git sin regresar a sttaged

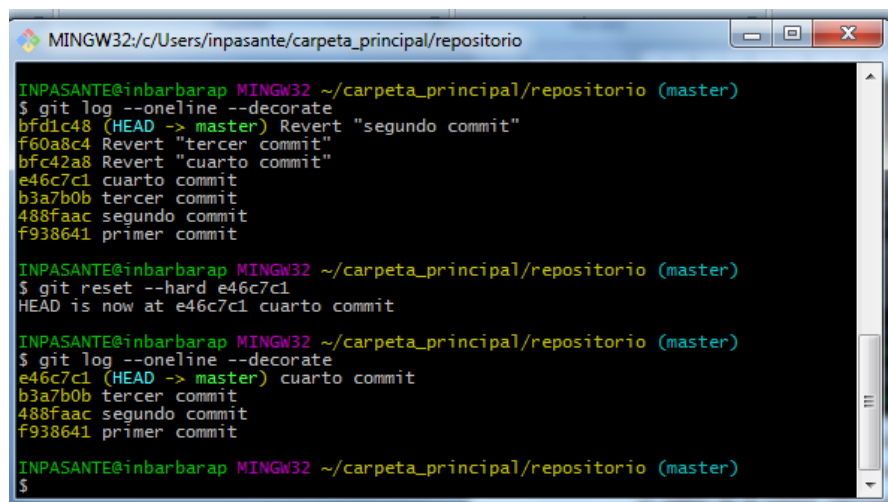


```
MINGW32:/c/Users/inpasante/carpeta_principal/repositorio
Revert "cuarto commit"

This reverts commit e46c7c14bc3da001cd34052d61d5029b3c06bc53.

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Changes to be committed:
#   new file:   archivo2.txt
#   modified:   archivodos.txt
#
~
~
```

Con reset es un proceso parecido veamos un ejemplo:



```
MINGW32:/c/Users/inpasante/carpeta_principal/repositorio (master)
$ git log --oneline --decorate
bfd1c48 (HEAD -> master) Revert "segundo commit"
f60a8c4 Revert "tercer commit"
bfc42a8 Revert "cuarto commit"
e46c7c1 cuarto commit
b3a7b0b tercer commit
488faac segundo commit
f938641 primer commit

INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (master)
$ git reset --hard e46c7c1
HEAD is now at e46c7c1 cuarto commit

INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (master)
$ git log --oneline --decorate
e46c7c1 (HEAD -> master) cuarto commit
b3a7b0b tercer commit
488faac segundo commit
f938641 primer commit

INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (master)
$
```

En este caso se observa lo mismo del ejemplo anterior solo que se agregaron mas revert para que sea más evidente la funcionalidad del reset, con el git log vemos el hash del commit que deseamos resetear y al usar el comando git reset utilizamos --hard que es para forzar el reset y eliminarlo desde el árbol observamos a hacer nuevamente el git log que todos los commit realizados después del commit usado en el reset han sido eliminados

4. Trabajando con repositorios remotos

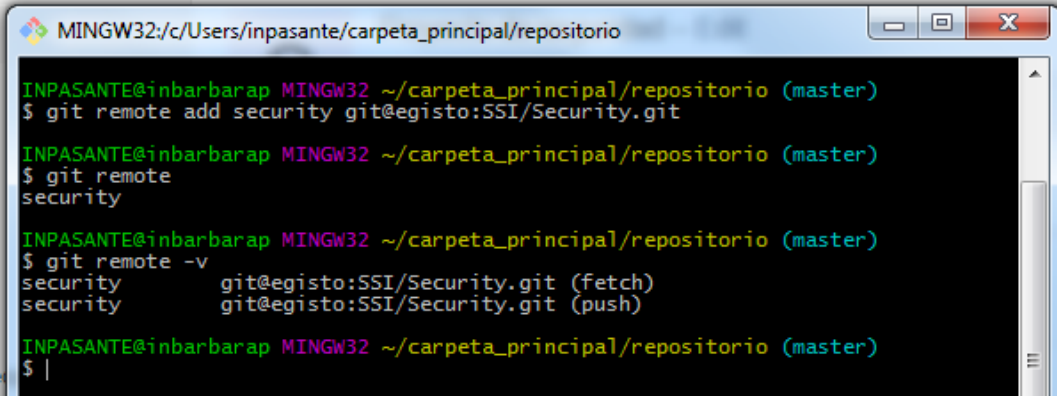
5.

Para poder colaborar en cualquier proyecto Git, necesitas saber cómo gestionar tus repositorios remotos. Los repositorios remotos son versiones de tu proyecto que se encuentran alojados en Internet o en algún punto de la red. Puedes tener varios, cada uno de los cuales puede ser de sólo lectura, o de lectura/escritura, según los permisos que tengas. Colaborar con otros implica gestionar estos repositorios remotos, y mandar (*push*) y recibir (*pull*) datos de ellos cuando necesites compartir cosas.

Gestionar repositorios remotos implica conocer cómo añadir repositorios nuevos, eliminar aquellos que ya no son válidos, gestionar ramas remotas e indicar si están bajo seguimiento o no, y más cosas.

5.1 Mostrando los repositorios remotos

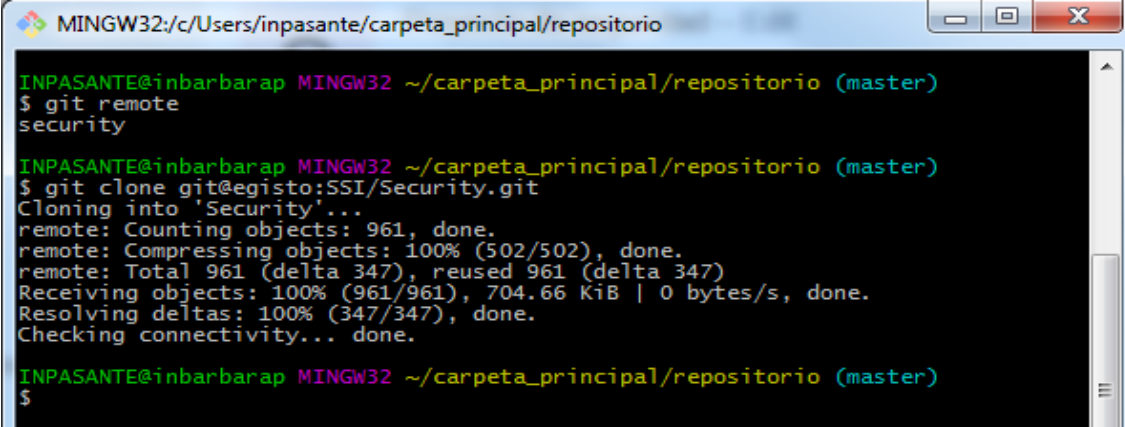
Para ver tus repositorios remotos tienes configurados se hace utilidad del comando *git remote*, para ver la dirección exacta del remoto en la opción *-v*, además se puede agregar remotos con la opción *add*, así *git remote add (nombreremoto) (direccionweb)*, de esta manera se pueden hacer dos cosas clonar un repositorio a distancia o hacer push que es enviar tu repositorio local y subirlo al servidor al repositorio en red.



```
MINGW32:/c:/Users/inpasante/carpeta_principal/repositorio
INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (master)
$ git remote add security git@egisto:SSI/Security.git
INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (master)
$ git remote
security
INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (master)
$ git remote -v
security          git@egisto:SSI/Security.git (fetch)
security          git@egisto:SSI/Security.git (push)
INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (master)
$ |
```

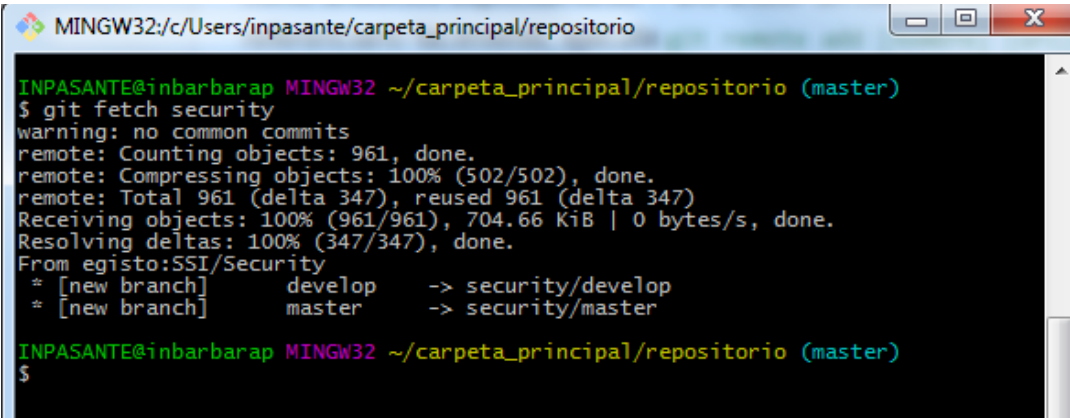
Aquí vemos como creamos primeramente el remoto con el comando *git remote add*, para verificar que el remoto ha sido creado usamos el comando *remote*, y para ver toda la dirección la opción *-v* en este caso vemos dos cosas al final de la dirección *fetch* y *push* *fetch* es para ver si hay actualizaciones en el repositorio remoto y descargarlas, mientras *push* es para enviar nuestro repositorio al repositorio remoto

Ahora veremos cómo bajar repositorios del servidor así como enviar nuestro repositorio al servidor. Para bajar un repositorio por primera vez podemos usar el comando *git clone* (dirección del repositorio)



```
MINGW32:/c/Users/inpasante/carpeta_principal/repositorio
INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (master)
$ git remote
security
INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (master)
$ git clone git@egisto:SSI/Security.git
Cloning into 'Security'...
remote: Counting objects: 961, done.
remote: Compressing objects: 100% (502/502), done.
remote: Total 961 (delta 347), reused 961 (delta 347)
Receiving objects: 100% (961/961), 704.66 KiB | 0 bytes/s, done.
Resolving deltas: 100% (347/347), done.
Checking connectivity... done.
INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (master)
$
```

Como vemos al ejecutar el comando *git clone* nos descarga el repositorio almacenado y este se aloja en la carpeta del repositorio en el que estamos ubicados donde todavía estará en el directorio de trabajo y será necesario hacer *git add* y *git commit* para enviarlo al directorio de git y tenerlo en nuestro repositorio.



```
MINGW32:/c/Users/inpasante/carpeta_principal/repositorio
INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (master)
$ git fetch security
warning: no common commits
remote: Counting objects: 961, done.
remote: Compressing objects: 100% (502/502), done.
remote: Total 961 (delta 347), reused 961 (delta 347)
Receiving objects: 100% (961/961), 704.66 KiB | 0 bytes/s, done.
Resolving deltas: 100% (347/347), done.
From egisto:SSI/Security
 * [new branch]      develop -> security/develop
 * [new branch]      master  -> security/master
INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (master)
$
```

Utilizamos el comando *git fetch* el remoto *security* vemos que no hay commit nuevos en el repositorio remoto y nos manda de nuevo el repositorio que ya estaba, y como se ve las ramas que existen en este repositorio, pero este comando solo baja el repositorio no lo une a la rama con la cual estamos trabajando para eso usamos el comando *git pull* que nos junta las dos versiones de archivo en una sola y somos capaces de seguir modificando y luego hacer un *push*.

```
INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (develop)
$ git push -u security develop
Counting objects: 11, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (9/9), done.
Writing objects: 100% (11/11), 1.23 KiB | 0 bytes/s, done.
Total 11 (delta 0), reused 0 (delta 0)
To git@egisto:SSI/Security.git
   a30168b..fa9beaf  develop -> develop
Branch develop set up to track remote branch develop from security.

INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (develop)
$ |
```

Utilizando push podemos subir nuestro repositorio local al repositorio remoto, como se ve en la pantalla aquí enviamos la rama develop a la rama que esta por defecto en el servidor

```
MINGW32:/c/Users/inpasante/carpeta_principal/repositorio
INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (master)
$ git pull prueba develop
From egisto:inpasante/prueba
 * branch                develop      -> FETCH_HEAD
Auto-merging index.html
CONFLICT (add/add): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.

INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (master|MERGING)
```

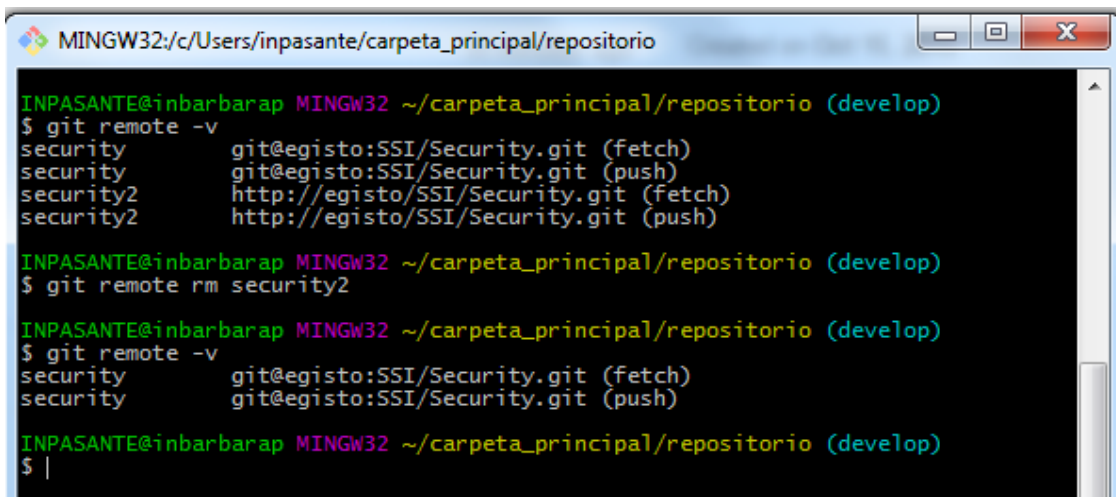
Como se menciona en una descripción de imagen más arriba el git pull funciona para descargar la rama que nosotros elijamos y fusionarla mediante un merge que se verá en la sección de ramas más abajo obteniendo los archivos de nuestro repositorio en el servidor aquí también se puede observar que hay un conflicto entre archivos tema también que se ve en la sección de ramas en el numeral 6

En caso queramos ver la configuración y la actividad de git con sus repositorios remotos usamos el comando *git remote show* (ubicación remota)

```
MINGW32:/c/Users/inpasante/carpeta_principal/repositorio
INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (develop)
$ git remote show security
* remote security
Fetch URL: git@egisto:SSI/Security.git
Push URL: git@egisto:SSI/Security.git
HEAD branch: develop
Remote branches:
  develop tracked
  master tracked
Local branch configured for 'git pull':
  develop merges with remote develop
Local refs configured for 'git push':
  develop pushes to develop (up to date)
  master pushes to master (local out of date)

INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (develop)
$ |
```

y si finalmente ya no necesitaremos una ubicación remota la borraremos la opción rm



```
MINGW32:/c/Users/inpasante/carpeta_principal/repositorio
INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (develop)
$ git remote -v
security          git@egisto:SSI/Security.git (fetch)
security          git@egisto:SSI/Security.git (push)
security2         http://egisto/SSI/Security.git (fetch)
security2         http://egisto/SSI/Security.git (push)

INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (develop)
$ git remote rm security2


INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (develop)
$ git remote -v
security          git@egisto:SSI/Security.git (fetch)
security          git@egisto:SSI/Security.git (push)

INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (develop)
$ |
```

6. Trabajando con ramas

Git usa puntos de control para ubicarse mientras “navega” dentro del árbol de archivos que genera cada commit(el commit apunta a la raíz del árbol), y cada vez que hay una confirmación se guarda un apuntador a esa confirmación, la rama es sencillamente un apuntador móvil a una de esas confirmaciones, la rama por defecto de git es master, cada confirmación que vayamos haciendo la rama ira avanzando automáticamente, y la rama master apuntara siempre a la ultima confirmación realizada.

Para ver primeramente las ramas que actualmente tenemos usaremos el comando *git branch*, nos muestra las ramas que están creadas y en la cual estamos actualmente, algo que debemos saber que para cambiar de rama debemos tener nuestro directorio git limpio quiere decir ya todo con un commit respectivo, otra cosa que tenemos que tener claro que en las diferentes ramas se pueden hacer diferentes modificaciones y que solo estarán disponibles en la rama que se tiene en ese momento y para funcionar esa rama con la que se desea se usa el comando merge, pero siguiendo con el comando *git branch* también se puede utilizar para crear una nueva rama como se a continuación:

A terminal window titled 'MINGW32:/c/Users/inpasante/carpeta_principal/repositorio' with standard Windows window controls. The terminal shows a user named 'INPASANTE@inbarbarap' in a 'MINGW32' environment. The first command is '\$ git branch', which lists 'develop' and 'master' with a green arrow pointing to 'master'. The second command is 'git br', which is partially shown. The third command is '\$ git branch prueba_rama'. The fourth command is '\$ git branch', which lists 'develop', 'master', and 'prueba_rama' with a green arrow pointing to 'master'.

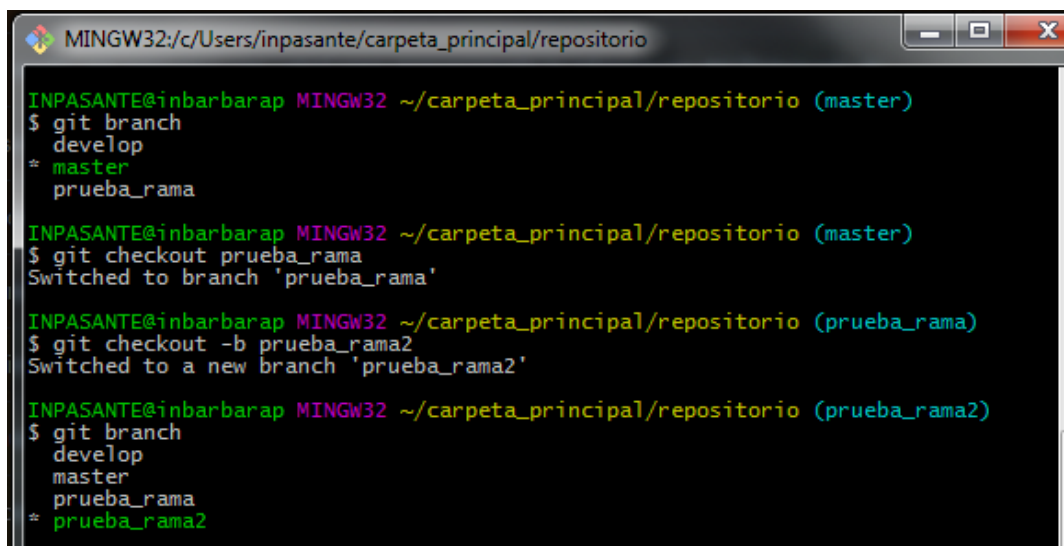
```
MINGW32:/c/Users/inpasante/carpeta_principal/repositorio

INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (master)
$ git branch
  develop
* master
git br
INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (master)
$ git branch prueba_rama

INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (master)
$ git branch
  develop
* master
  prueba_rama
```

Como se observa primero usamos *git branch* para ver las ramas actuales, después usamos el mismo comando para crear una nueva rama llamada *prueba_rama* y finalmente verificamos con el mismo *git branch* si ya existe

Ahora si queremos cambiarnos a esa rama para no tocar las *master* y poder modificar archivos del *git* directory usamos el comando *checkout* (rama a usar) este comando como se vio anteriormente tiene varias funcionalidades otra funcionalidad es crear ramas nuevas y cambiar a esa rama todo en el mismo paso con la opción *-b*

A terminal window titled 'MINGW32:/c/Users/inpasante/carpeta_principal/repositorio' with standard Windows window controls. The terminal shows a user named 'INPASANTE@inbarbarap' in a 'MINGW32' environment. The first command is '\$ git branch', which lists 'develop', 'master', and 'prueba_rama' with a green arrow pointing to 'master'. The second command is '\$ git checkout prueba_rama', which outputs 'Switched to branch 'prueba_rama''. The third command is '\$ git checkout -b prueba_rama2', which outputs 'Switched to a new branch 'prueba_rama2''. The fourth command is '\$ git branch', which lists 'develop', 'master', 'prueba_rama', and 'prueba_rama2' with a green arrow pointing to 'prueba_rama2'.

```
MINGW32:/c/Users/inpasante/carpeta_principal/repositorio

INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (master)
$ git branch
  develop
* master
  prueba_rama

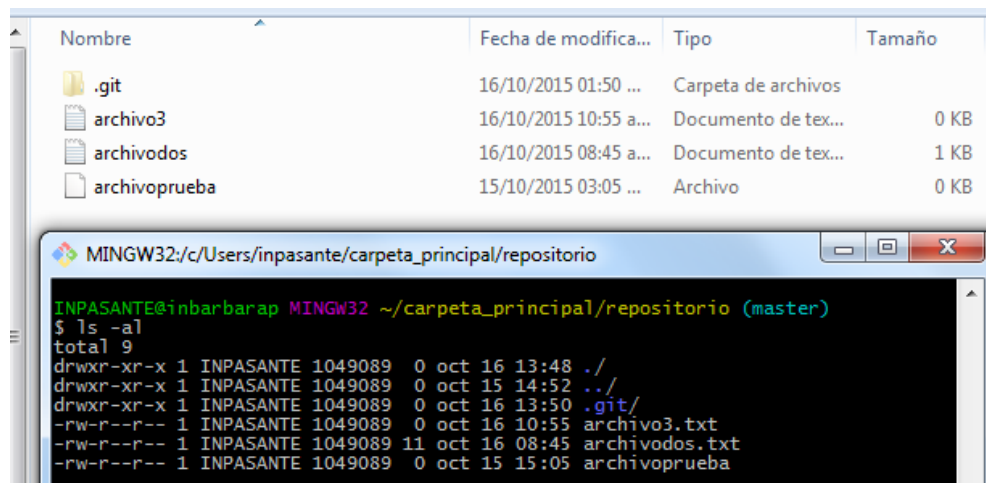
INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (master)
$ git checkout prueba_rama
Switched to branch 'prueba_rama'

INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (prueba_rama)
$ git checkout -b prueba_rama2
Switched to a new branch 'prueba_rama2'

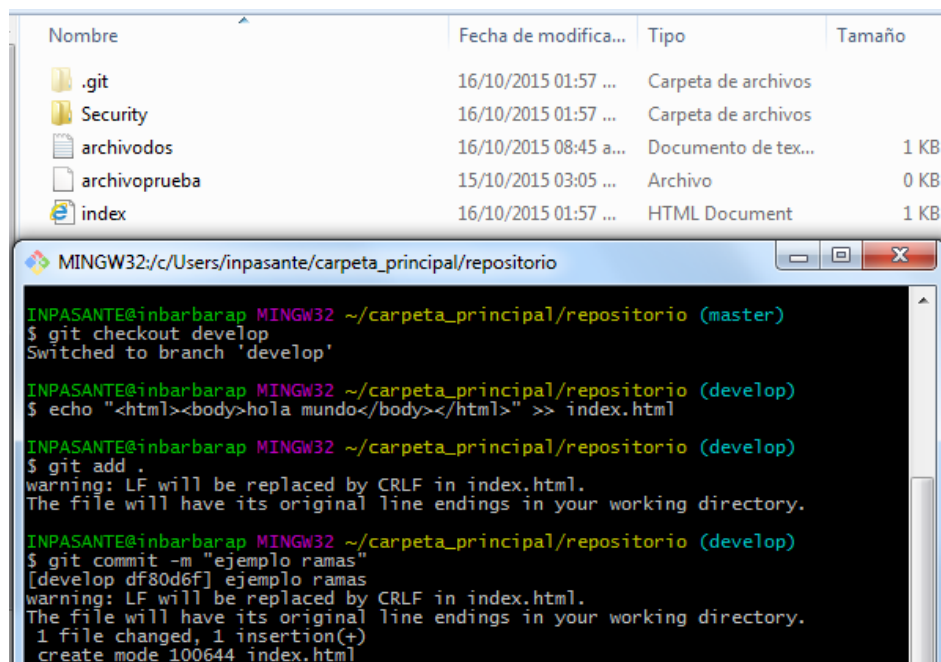
INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (prueba_rama2)
$ git branch
  develop
  master
  prueba_rama
* prueba_rama2
```

Usamos el comando *git branch* para ver las ramas actuales, luego con el *git checkout* cambiamos a la rama *prueba_rama* y como se puede ver en la consola entre los paréntesis azules nos muestra en que rama estamos actualmente, con el comando *git checkout -b* creamos una nueva rama y en una sola ejecución nos cambiamos a ella, y finalmente usamos el *branch* para observar las ramas que tenemos creadas y en cual estamos actualmente.

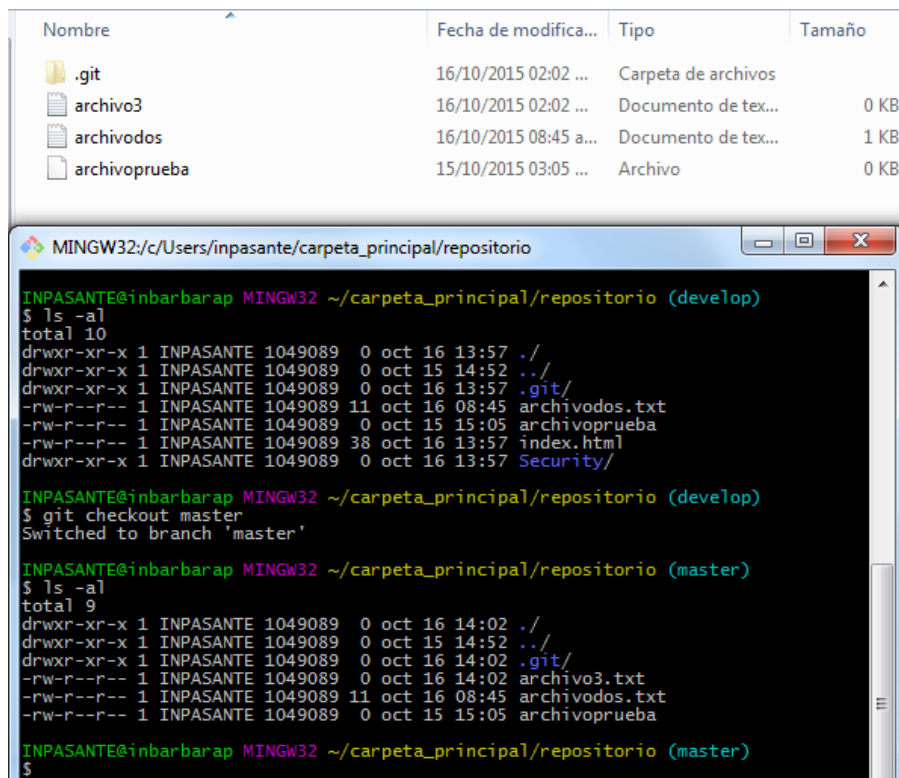
Como se menciono antes al navegar entre ramas podemos hacer modificaciones diferentes en cada rama las cuales serán independientes de las otras ramas que se tengan, ejemplo:



Actualmente estamos en la rama master y tenemos solamente tres archivos, cambiaremos a una rama en este caso develop y crearemos un archivo nuevo y añadiremos algo nuevo después del commit, para luego regresar a master y ver que esa creación no está en la otra rama

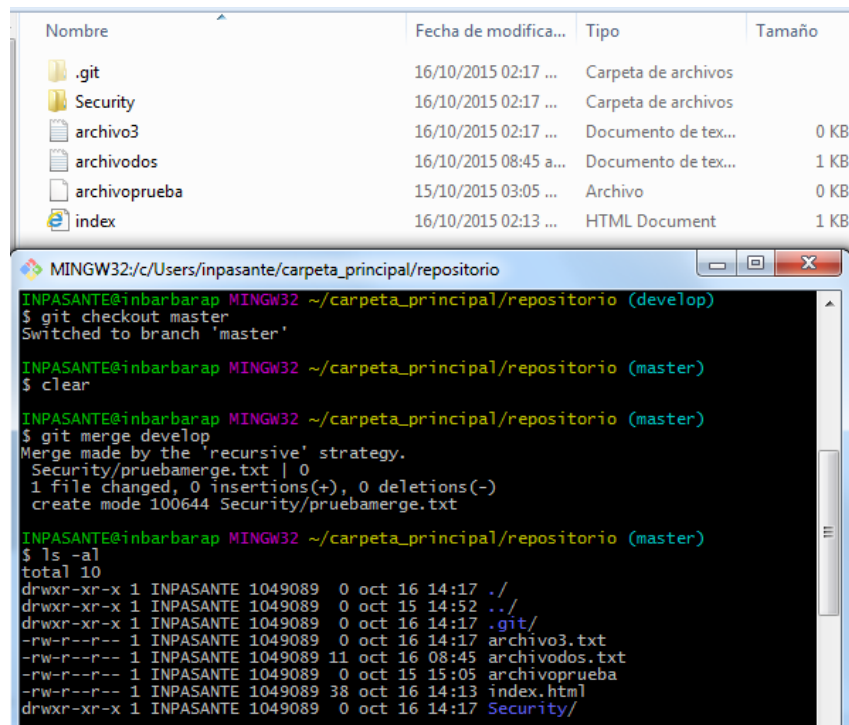


Como mencionamos cambiamos a la rama develop y creamos un archivo index.html en esta rama, como se dijo antes no se puede navegar entre ramas a menos que tengamos el directory git limpio con un commit ya hecho de los cambios que se realizaron, como se puede observar también en las imágenes hemos eliminado la carpeta security de la rama master pero al cambiar a develop esta sigue ahí

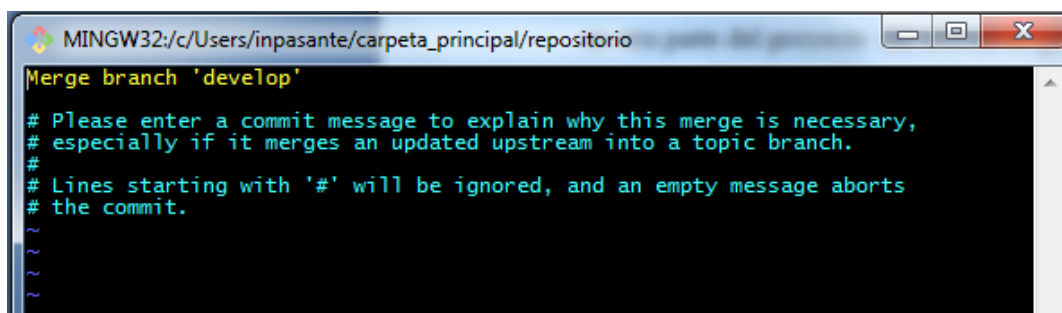


Finalmente en esta imagen podemos comparar los archivos que existen en una rama y los archivos que existen al cambiarse a otra rama.

Esto nos conviene mucho a la hora de trabajar en diferentes archivos para un proyecto ya que sabemos que tenemos los archivos originales o que podemos simplemente crear otra rama y trabajar en otra parte del proyecto sin afectar lo que ya tenemos hecho, pero para reunir las partes que tenemos realizadas en diferentes ramas y tener el proyecto completo necesitamos funcionarlas como se hace esto, pues con el comando *git merge*, este funciona entrando a la rama en la que queremos que quede el proyecto completo y ejecutamos el *git merge (nombre rama)* el nombre de la rama es el nombre de aquella donde hemos modificado el proyecto y toda la información ahí caerá encima de los archivos en la rama destino, no es reemplazando sino sobrescribiendo los archivos y agregando las líneas que hagan falta del archivo que ya se encontraba en la rama destino o copiando los archivos que no estaban ahí pero si en la rama de la cual copiamos ejemplo:



Como vemos al hacer el git merge desde la rama develop hacia la rama master vemos que todos los archivos que estaban en la rama develop han pasado a la rama master, la cual antes no tenía el archivo index.html ni la carpeta Security, al ejecutar el comando merge se nos abre una ventana de texto donde nos permite escribir un comentario porque el efecto de funcionar dos ramas con merge se considera un commit si damos git log veremos que aparece



También durante un merge pude haber un conflicto es raro si hablamos de desarrollo de código ya que rara vez se va estar tocando el mismo archivo de configuración o código fuente una vez estando en el server pero si llega a suceder a la hora de hacer un merge se puede originar un conflicto ya que ambas ramas en esta caso local modificaron el mismo archivo, ejemplo:

```
MINGW32:/c/Users/inpasante/carpeta_principal/repositorio

INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (master)
$ cat index.html
<html>
<head>
<style type="text/css">
body
{
    background-color:#CCCCCC;
}
div.class
{
    background-color:#999999;
padding: .5em;
}
</style>
</head>
<body>
<div class="class"><h1> mi primera pagina web</h1></div>
</body>
</html>

INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (master)
$ |
```

```
MINGW32:/c/Users/inpasante/carpeta_principal/repositorio

INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (master)
$ git checkout develop
Switched to branch 'develop'
c
INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (develop)
$ cat index.html
<html><body>hola mundo</body></html>

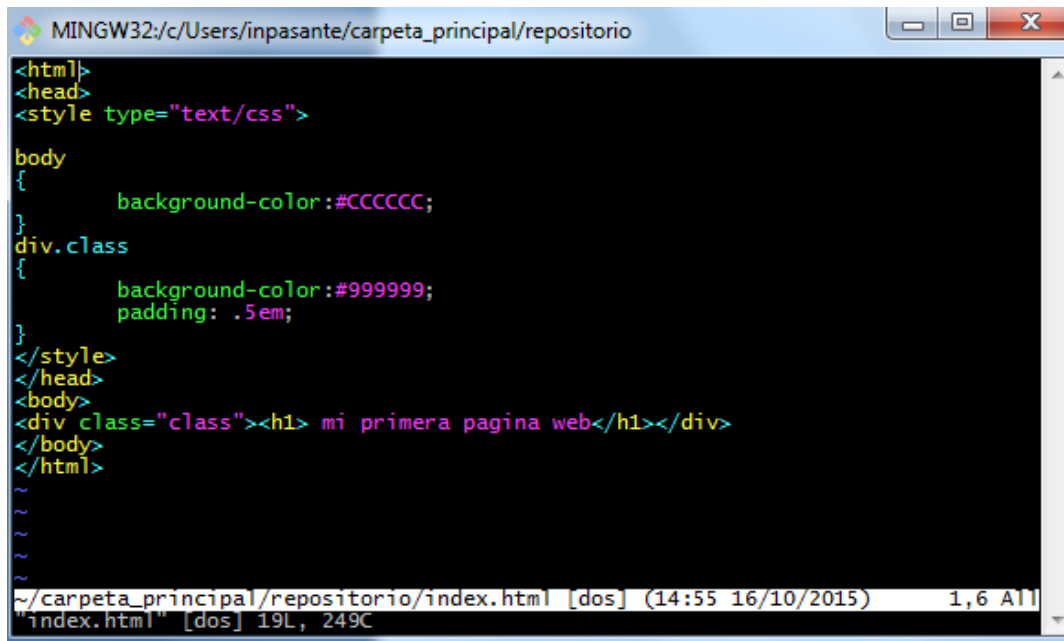
INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (develop)
$ vim index.html

INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (develop)
$ cat index.html
<html>
<head>
<style type="text/css">
body
{
    background-color:#555555;
}
div.tema
{
    background-color:#CCCCCC;
width:1600px;
height:250px;
}
</style>
</head>
<body>
<div class="tema">
<h1>Mi primera pagina web</h1>
</div>
</body>
</html>

INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (develop)
$
```

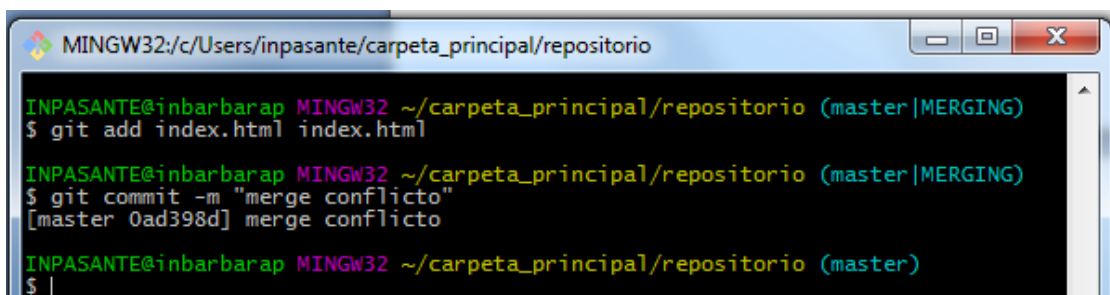
Hemos modificado ambos archivos index.html tanto en master como es develop

Como se puede observar al abrir el archivo `index.html` con `vim` el mismo `git` nos muestra la parte del código que nos da conflicto iniciando en `<<<<head` (`head` es el nombre del ultimo commit realizado donde apunta también la rama actual en este caso `master`), en medio del conflicto nos separa con triple raya y nos coloca la parte del código de la otra rama en este caso `develop`, como se pueden dar cuenta `git` coloca los dos códigos de ambos archivos para el usuario elija cual debe usar y eliminar el otro



```
<html>
<head>
<style type="text/css">
body
{
    background-color:#CCCCCC;
}
div.class
{
    background-color:#999999;
    padding: .5em;
}
</style>
</head>
<body>
<div class="class"><h1> mi primera pagina web</h1></div>
</body>
</html>
~
~
~
~/carpeta_principal/repositorio/index.html [dos] (14:55 16/10/2015) 1,6 All
index.html [dos] 19L, 249C
```

La manera de solucionar un conflicto dependerá del usuario ya que puede borrar la sección de código que ama le convenga y dejar la más funcional o dejar las dos versiones del código depende de las necesidades, una vez arreglado el archivo hacemos lo que ya sabemos usamos `git add` para agregar el archivo y `commit` para enviarlo a `git` directory y el merge se completa con éxito



```
INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (master|MERGING)
$ git add index.html index.html

INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (master|MERGING)
$ git commit -m "merge conflicto"
[master 0ad398d] merge conflicto

INPASANTE@inbarbarap MINGW32 ~/carpeta_principal/repositorio (master)
$
```

7. Fuentes de información

Para obtener mayor información visitar:

http://librosweb.es/libro/pro_git/

Y agregamos un curso online en youtube

<https://www.youtube.com/watch?v=jSJ8xhKtfP4&list=PLTd5ehlJ0goMCnj6V5NdzSIHBgrlXckGU>