Control de versiones con Git y GitHub

Sumario

Control de versiones con Git

Introducción a Git

Git es un software de control de versiones diseñado por Linus Torvalds, pensando en la eficiencia y la confiabilidad del mantenimiento de versiones de aplicaciones cuando estas tienen un gran número de archivos de código fuente.

Se llama **control de versiones** a la gestión de los diversos cambios que se realizan sobre los elementos de algún producto o una configuración del mismo. Una versión, revisión o edición de un producto, es el estado en el que se encuentra dicho producto en un momento dado de su desarrollo o modificación.

Aunque un sistema de control de versiones puede realizarse de forma manual, es muy aconsejable disponer de herramientas que faciliten esta gestión dando lugar a los llamados sistemas de control de versiones o SVC (del inglés System Version Control).

Estos sistemas facilitan la administración de las distintas versiones de cada producto desarrollado, así como las posibles especializaciones realizadas (por ejemplo, para algún cliente específico). Ejemplos de este tipo de herramientas son entre otros: Git, CVS, Subversion, SourceSafe, ClearCase, Darcs, Bazaar , Plastic SCM, Git, Mercurial, Perforce, etc.

Al principio, Git se pensó como un motor de bajo nivel sobre el cual otros pudieran escribir la interfaz de usuario o front-end como Cogito o StGIT. Sin embargo, Git se ha convertido desde entonces en un sistema de control de versiones con funcionalidad plena.

Hay algunos proyectos de mucha relevancia que ya usan Git, en particular, el grupo de programación del núcleo Linux.

En Git tendremos el **directorio de trabajo**, la **staging area** y el **directorio git** (repositorio).

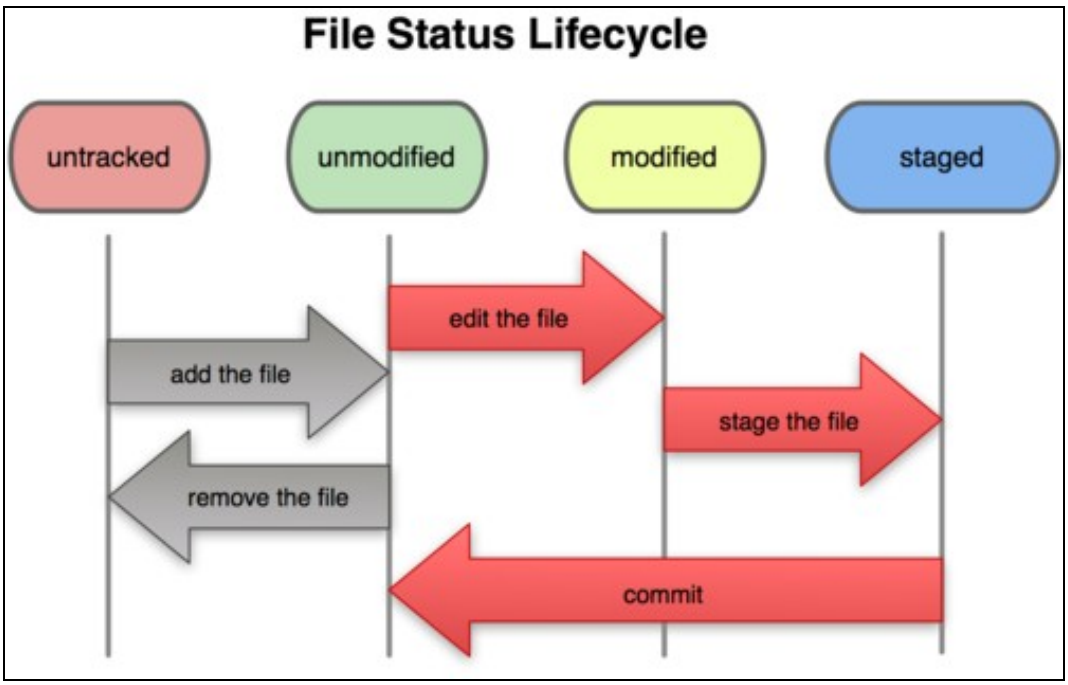
- En el **directorio de trabajo**, tendremos todos los archivos de nuestro proyecto, incluidos aquellos que no nos interesa mandar al repositorio.
- La **staging area** es una zona o área de espera, a la que mandaremos los archivos que tenemos listos para actualizar en el repositorio.
- El directorio git es dónde se almacenará el snapshot (imagen actual) de los archivos que están en la staging área.

En Git, los archivos pueden encontrarse en los siguientes estados:

- **untracked**: cuando no se ha añadido a ningún repositorio.
- **tracked**: el fichero ha sido añadido a algún repositorio.
- **staged**: el fichero ha sido añadido al repositorio, pero no ha sido enviado todavía con commit al repositorio.
- **modified**: el fichero ha sido modificado.
- **unmodified**: el fichero no ha sido modificado.
- **commited**: el fichero ya ha sido actualizado en el repositorio.

El **ciclo de vida** de un archivo en el control de versiones será el siguiente:

- En un principio cualquier nuevo archivo recién creado en nuestro directorio será un archivo **untracked**, por que no forma parte de ningún commit (actualización del repositorio).
- Cuando añadimos el archivo al proyecto, será un archivo **unmodified** (ya que no se ha realizado ningún cambio sobre él). Pasará a ser modified cuando después de añadirlo, hayamos realizado algún cambio o edición sobre ese fichero.
- Para que esos cambios pasen a formar parte del repositorio, hay que situarlo en la **staging area** (listos para ser actualizados) y entonces el archivo estará **staged** (listo para ser actualizado en el repositorio).
- Por último al hacer el **commit**, todos los cambios se almacenarán en el repositorio.
- A partir del commit, todo ese contenido pasará a ser **unmodified** y de nuevo el ciclo se repetirá.



Video de introducción a Git y GitHub (hasta el minuto 31:40):

EmbedVideo does not recognize the video service "youtubehd".

Instalación

Para instalar Git es muy sencillo.

Tendremos que acceder a la página: http://git-scm.com/download

Nos descargaremos la versión para nuestro sistema operativo y una vez instalado ya podremos crear un repositorio.

Podemos tener tres tipos diferentes de repositorio:

- **Privado** (en nuestro equipo).
- **Externo** (en un servidor externo).
- **Interno** (en un servidor interno).

Video de la instalación (a partir del minuto 31:40):

EmbedVideo does not recognize the video service "youtubehd".

Creación de un repositorio privado

Todos los comandos que vamos a ver se ejecutarán sobre **Git Shell**.

El git Shell, visto desde Windows, es un aplicación que abre un entorno de línea de comandos al estilo Linux. Nos permitirá ejecutar típicos comandos de Linux, como crear carpetas, borrar archivos, editar un archivo con vi, etc.

Para crear un repositorio privado, crearemos la carpeta del proyecto y a continuación inicializaremos el repositorio con el comando **git init**. Para ello abriremos el **Git Shell**, accederemos a la carpeta dónde vamos a grabar el proyecto y allí dentro ejecutaremos el comando **git init**.

Si queremos que cualquier carpeta sea un nuevo repositorio independiente de los otros, habrá que inicializarla con git init, sino formará parte del repositorio padre del que cuelga.

Por ejemplo:

```
# Accederemos a la ruta deseada y crearemos una carpeta:
mkdir proyecto
cd proyecto

# Una vez dentro del proyecto inicializaremos el proyecto con el comando "git init":
/proyecto/git init


# Después de un mensaje de creación del repositorio, si mostramos el contenido
# de la carpeta con un ls -al veremos una carpeta oculta (.git):
ls -al
.git
```

Directorios dentro de la carpeta .git

Dentro de la carpeta .git tendremos varias subcarpetas algunas de ellas son las siguientes:

- **HEAD**: para el registro de cabeceras.
- **branches**: las diferentes versiones que vamos subiendo.
- **config**: el archivo de configuración.
- **description**: la descripción.
- **hooks**: para la ejecución de scripts necesarios para funcionamiento de git.

El primer commit

Añadiendo ficheros

Una vez que ya tenemos creado el repositorio, vamos a enviar algunos ficheros y **notificar al sistema** que existen dichos archivos (**commit**).

Para ello utilizaremos el comando **git add ficheros**.

Atención: recordar que todos los comandos hay que ejecutarlos desde el **Git Shell**.

```
# Accederemos a la carpeta del proyecto y creamos algún archivo de pruebas:

/proyecto/touch hola.html
/proyecto/touch index.html
/proyecto/touch prueba.txt

# Ahora le indicamos al sistema que hay nuevos ficheros:
# (el punto hace referencia a la carpeta actual y añadirá todos los ficheros encontrados)
/proyecto/git add .
```

Comprobación de las modificaciones en el repositorio

Para comprobar las modificaciones que se han producido en el repositorio utilizamos el comando **git status**.

```
# Ejecutamos el comando git status:
/proyecto/git status

# Se mostrará algo como el ejemplo siguiente en el que aparecen los ficheros añadidos:

git status
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#       new file:   hola.html
#       new file:   index.html
#       new file:   prueba.txt
#
```

Notificando los cambios al repositorio

Para notificar al repositorio que hemos añadido ficheros, se hace con el comando **git commit**

```
# Notificamos al repositorio los cambios que se han producido, y se envían aquellos ficheros
# que se encuentran en la staging área:
/proyecto/git commit -m "Nuestro primer commit"

# Si queremos que se envíen todos los archivos modificados desde el último commit,
# independientemente de si se encuentran o no en la staging área, usaremos -a:
/proyecto/git commit -a -m "Nuestro primer commit con todos los archivos modificados"


# Mostrará algo como lo siguiente:


[master (root-commit) 9f89cdf] Nuestro primer commit
 0 files changed
 create mode 100644 hola.html
 create mode 100644 index.html
 create mode 100644 prueba.txt
```

El log de las modificaciones

El sistema va guardando en un fichero de log todos los cambios que se van realizando, de forma que se pueden hacer recuperaciones fácilmente.

Para ver su contenido usamos el comando **git log**.

```
# Mostramos el log de los cambios:
/proyecto/git log

# Y se mostrará algo como lo siguiente:
git log

commit 9f89cdf69f653892207a651ca55de8d07d4025ee
Author: Rafa Veiga <micorreo@dominio.com>
Date:   Sat Jan 19 00:52:15 2013 +0100

    Nuestro primer commit
```

Modificando ficheros

Vamos a modificar uno de los ficheros de ejemplo (prueba.txt):

```
vi prueba.txt
Probando cambios en el fichero.

A ver si funciona git.

ESC :wq
```

A continuación comprobamos el estado de git:

```
# Comprobamos el estado del repositorio:
git status

# Veremos que nos muestra que se ha modificado un fichero:
usuario@ATLANTA /d/xampp/htdocs/web/pruebas (master)
$ git status
# On branch 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)
#
#       modified:   prueba.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
```

Sabiendo que se ha modificado un fichero podemos comprobar que cambios se han realizado. Para ello utilizaremos la herramienta **git diff --color**, que nos permitirá comparar archivos:

```
/proyecto/git diff --color

# Se mostrará algo como ésto en diferentes colores:
$ git diff --color
diff --git a/prueba.txt b/prueba.txt
index e69de29..b338220 100644
--- a/prueba.txt
+++ b/prueba.txt
@@ -0,0 +1,5 @@
+Probando cambios en el fichero.
+
+A ver si funciona git.
+
+
```

El segundo commit

Vamos a notificar al repositorio los cambios que hemos hecho en el fichero prueba.txt.

Notificando los cambios con el segundo commit

Para notificar el segundo commit lo hacemos empleando el mismo comando usado anteriormente:

```
git commit -a -m "Hemos modificado el fichero prueba.txt"

# Obtendremos un resultado similar a:
$ git commit -a -m "Hemos modificado el fichero prueba.txt"
[master dac029f] Hemos modificado el fichero prueba.txt
 1 file changed, 5 insertions(+)

# El sistema nos va dando información de los cambios realizados.
# En este caso se han realizado 5 insercciones.
```

Añadiendo más ficheros al repositorio

Por ejemplo vamos a crear un nuevo fichero y notificarlo al repositorio.

```
# Creamos el nuevo fichero
/proyecto/vi contactar.html
Información de contacto de la web.

Saludos.
ESC :wq

# Añadimos el fichero con git add fichero:
git add contactar.html

# Comprobamos los cambios con git status:
git status

# Obtendremos un resultado similar a:
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   contactar.html
#
# 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:   contactar.html
#       modified:   index.html
#       modified:   prueba.txt
#

# Ahora vemos las modificaciones:
/proyecto/git diff --color

$ git diff --color
diff --git a/contactar.html b/contactar.html
index bead4f2..0e93deb 100644
--- a/contactar.html
+++ b/contactar.html
@@ -1,4 +1,5 @@
 Informaci<F3>n de contacto de la web.

 Saludos.
+Rafa Veiga.

diff --git a/index.html b/index.html
index e69de29..357e4ea 100644
--- a/index.html
+++ b/index.html
@@ -0,0 +1 @@
+Aqui va el index
diff --git a/prueba.txt b/prueba.txt
index b338220..57fad94 100644
--- a/prueba.txt
+++ b/prueba.txt
@@ -1,5 +1,5 @@
-Probando cambios en el fichero.
+Probando cambios:
```

:

El tercer commit

Emplearemos de nuevo el comando **git commit -a -m "Texto de la modificacion"**:

```
$ git commit -a -m "Cambio en contenidos y nuevo fichero"
[master f04d938] Cambio en contenidos y nuevo fichero
 3 files changed, 8 insertions(+), 2 deletions(-)
 create mode 100644 contactar.html
```

Los branches (ramas)

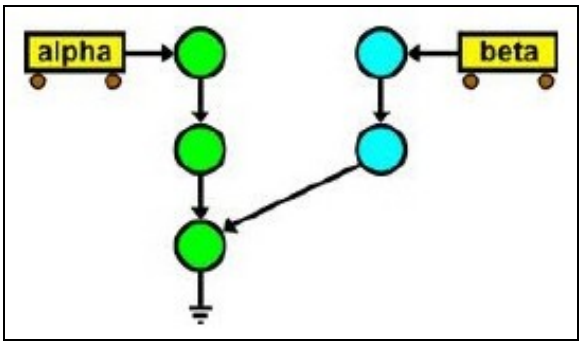
Un branch (rama) es una versión de nuestro código. Git nos permite movernos entre versiones de forma sencilla.

La creación de ramas nos permite trabajar en diferentes versiones de un mismo archivo y cuando lo consideremos podremos fusionar los cambios.

Cada vez que creamos una rama, se crea un nuevo puntero a la versión indicada o a la que estamos trabajando. Se van a emplear los comandos **git branch** y **git checkout**.

Se de?nen fácilmente y no hace falta hacerlo a priori. Los podemos crear en cualquier momento.

Para saber en que versión nos encontramos, Git mantiene un puntero a la versión en la que estamos trabajando. Este puntero lo tiene referenciado en **HEAD** (dentro de la carpeta .git).



Definir un branch

Para definir un branch se utiliza el comando **git branch** y para cambiarnos a ese branch se utiliza el comando **git checkout**.

Podríamos definir un branch y cambiarnos inmediatamente utilizando **git checkout -b nombre_branch**:

Ejemplo:

```
# Definimos un branch controlLDAP
git branch controlLDAP

# Nos cambiamos a ese branch
git checkout controlLDAP

# Podríamos hacerlo en un único paso, crear el branch y cambiarnos a la nueva versión con:
git checkout -b controlLDAP

# Podremos hacer la programación de nuevas opciones en este branch y cuando hayamos terminado
# Se enviarían los cambios y se fusionaría con la rama master, que es la que se enviaría a producción.
```

Supongamos que queremos definir como version 0.1, el primer commit que hemos hecho.

```
# Mostramos un log de los cambios realizados para ver el código del último commit.
git log
```

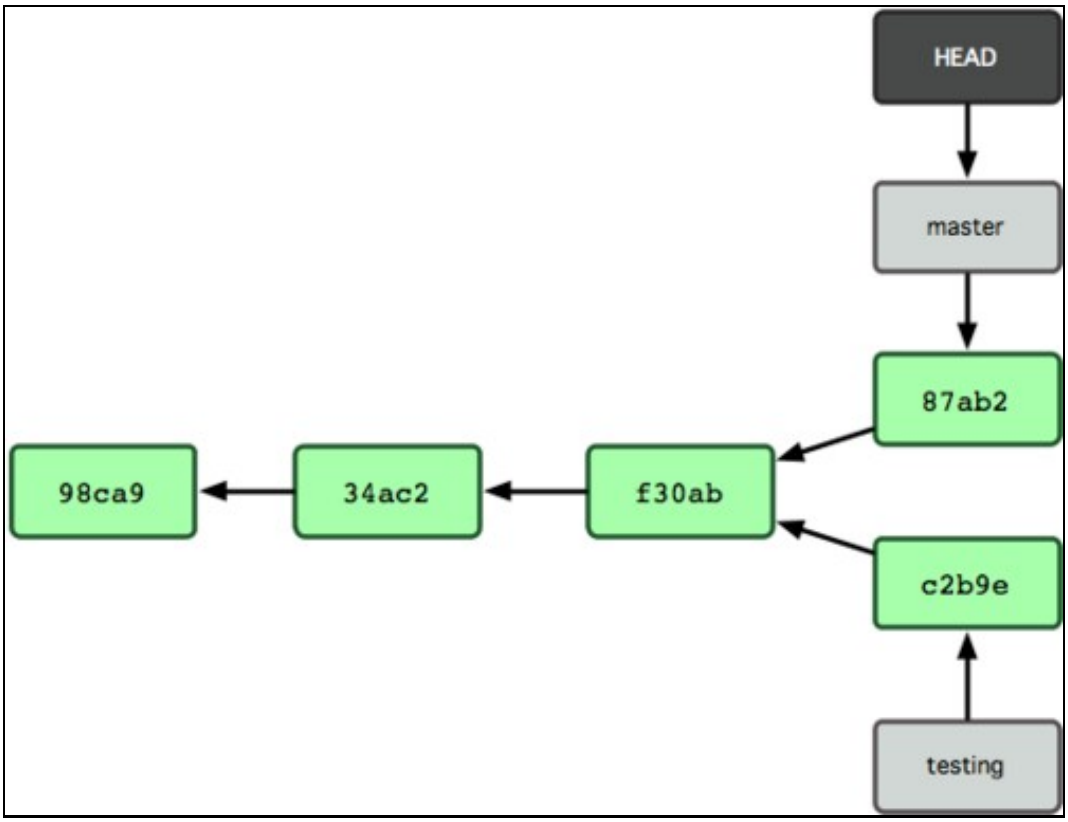
```
# Se mostrará algo como:
$ git log
commit f04d938eaec19c31e7185389a51da00869e83d1d
Author: Rafa Veiga <profesorveiga@gmail.com>
Date:   Sat Jan 19 01:24:35 2013 +0100

    Cambio en contenidos y nuevo fichero

commit dac029f1abb4f9cb43d7924b3fafeb5e2d1dfe78
Author: Rafa Veiga <profesorveiga@gmail.com>
Date:   Sat Jan 19 01:11:05 2013 +0100

    Hemos modificado el fichero prueba.txt

commit 9f89cdf69f653892207a651ca55de8d07d4025ee
Author: Rafa Veiga <profesorveiga@gmail.com>
Date:   Sat Jan 19 00:52:15 2013 +0100

    Nuestro primer commit

# Entonces ya podemos crear el branch (rama, versión 0.1, en este caso) con:
git branch 0.1 9f89cdf69f653892207a651ca55de8d07d4025ee

# El código 9f89cdf69f653892207a651ca55de8d07d4025ee es el identificador del primer commit realizado.

# Si queremos crear un nuevo branch en la versión actual en la que nos encontramos teclearemos
# directamente git branch nombre_branch. Por ejemplo:
git branch testeando

# Comprobaremos los cambios con git log --decorate:
git log --decorate

# Se mostrará algo como:
$ git log --decorate
commit f04d938eaec19c31e7185389a51da00869e83d1d (HEAD, master)
Author: Rafa Veiga <profesorveiga@gmail.com>
Date:   Sat Jan 19 01:24:35 2013 +0100

    Cambio en contenidos y nuevo fichero

commit dac029f1abb4f9cb43d7924b3fafeb5e2d1dfe78
Author: Rafa Veiga <profesorveiga@gmail.com>
Date:   Sat Jan 19 01:11:05 2013 +0100

    Hemos modificado el fichero prueba.txt

commit 9f89cdf69f653892207a651ca55de8d07d4025ee (0.1)
Author: Rafa Veiga <profesorveiga@gmail.com>
Date:   Sat Jan 19 00:52:15 2013 +0100

    Nuestro primer commit

# Dónde podemos ver que se muestra la versión 0.1 asignada al primer commit.
```

Cómo cambiar de versión o branch

Podemos cambiarnos entre versiones o branches fácilmente. De esta forma podremos añadir o ver los cambios realizados fácilmente en cada una de las versiones.

ATENCION: *Tenemos que tener en cuenta que se recomienda aplicar cambios en el branch en el que estamos trabajando, antes de cambiarnos a un branch distinto.*

Con la instrucción **git checkout nombre_branch** nos cambiamos de rama (branch):




```
# Por ejemplo en base a la imagen anterior:
# Nos podemos cambiar a la versión master con:
git checkout master

# Nos muestra como resultado de la ejecución del comando anterior:
git checkout master
Already on 'master'

# Si nos queremos cambiar a la versión testing, teclearemos:
git checkout testing
Switched to branch 'testing'

# Nos mostrará un mensaje indicando que se ha cambiado a la versión 0.1.
```

Clonando repositorios

Para hacer una copia (clon) de un repositorio, se realiza con una instrucción bastante sencilla.

La instrucción **git clone destino**, realizará una copia del repositorio actual en el destino indicado.

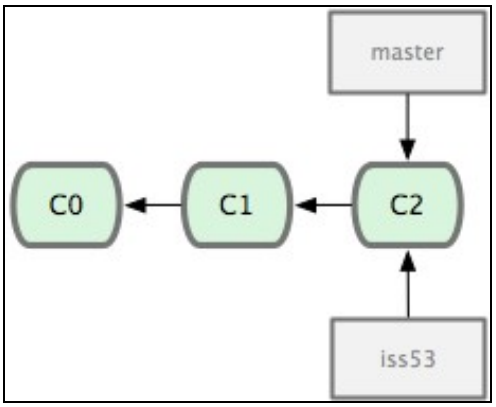
```
# Copiamos el proyecto actual en la carpeta /copiaproyecto
# Desde la carpeta de proyecto ejecutaremos:
git clone /copiaproyecto
```

Usando merge

Merge nos va a permitir fusionar diferentes versiones. Para ello se utilizará la instrucción **git merge nombre_rama**.

Vamos a ver un ejemplo sencillo de ramas, versiones o branches (como quieras llamarlo) y cómo podemos usarlo en nuestro flujo normal de trabajo.

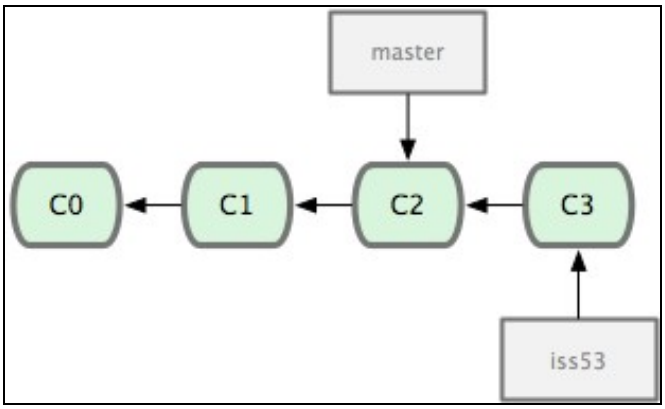
- Estamos trabajando en una web.
- Creamos una nueva rama para una nueva opción que queremos implementar.
- Trabajamos un poco en esa nueva opción.




```
# Por ejemplo, creamos una rama llamada iss53 en la que vamos a programar nuevas opciones del proyecto.
git checkout -b iss53

# El comando anterior es la versión reducida de
# git branch iss53
# git checkout iss53
```

Trabajamos en la nueva rama (por ejemplo añadiendo un nuevo pie al fichero index.html) y hacemos un commit para enviar los cambios.

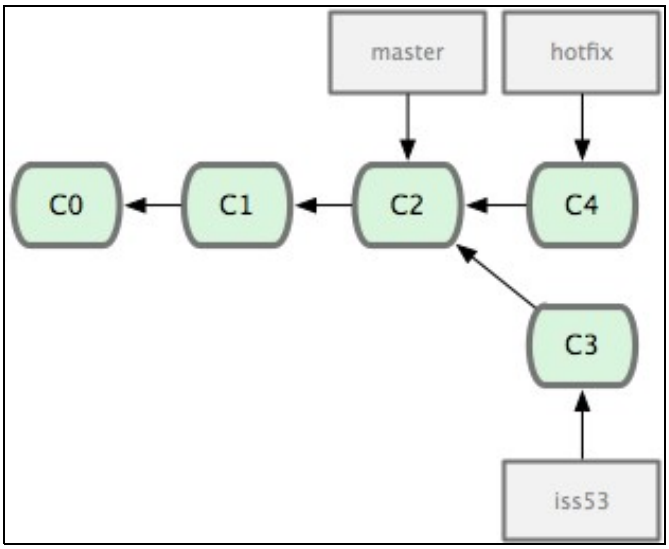
```
vim index.html
git commit -a -m 'Añadido nuevo footer a la web [requisito 53]'
```



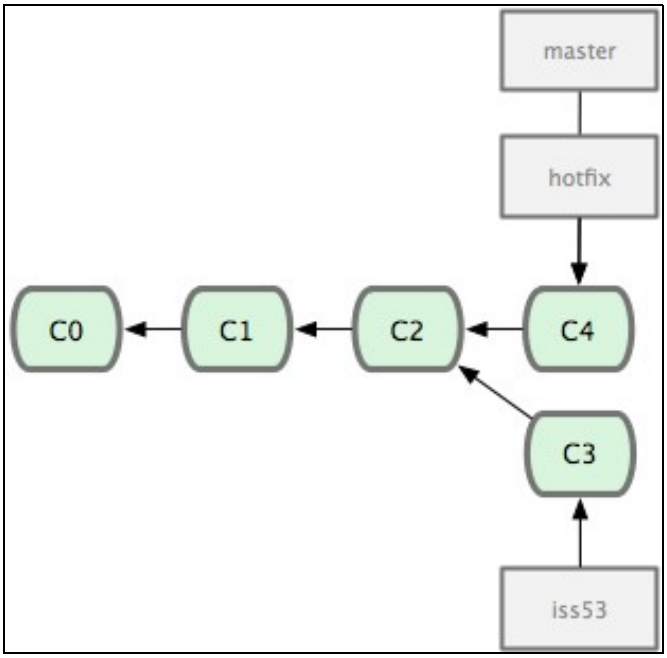
En ese momento recibimos una llamada y necesitamos realizar una corrección de un fallo de forma urgente en la web. Realizaremos lo siguiente:

- Volveremos de nuevo a la rama de producción.
- Creamos una rama para añadir la corrección.
- Después de haber comprobado que la corrección funciona, fusionamos la rama con el master y la enviamos a producción.
- Volvemos a la rama de la nueva opción en la que estábamos inmersos, y continuamos con nuestro trabajo.

```
# Volveremos de nuevo a la rama de producción.
git checkout master

Switched to branch "master"

# Creamos una rama para añadir la corrección.
git checkout -b hotfix

Switched to a new branch "hotfix"

vim index.html

git commit -a -m 'fixed the broken email address'[hotfix]:
created 3a0874c: "fixed the broken email address"
1 files changed, 0 insertions(+), 1 deletions(-)
```

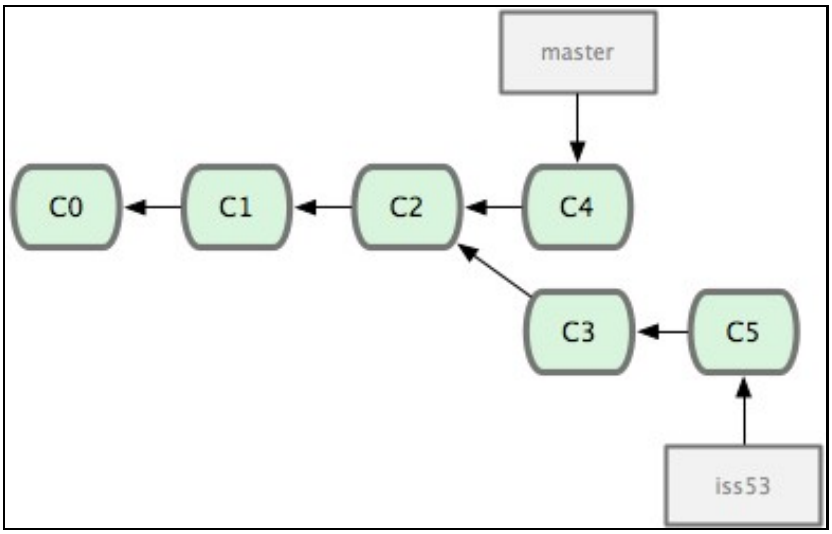



```
# Después de haber comprobado que la corrección funciona, fusionamos la rama con el master
# para enviarla a producción.
git checkout master
git merge hotfix

Updating f42c576..3a0874cFast forward
 README |    1 -
 1 files changed, 0 insertions(+), 1 deletions(-)

# La palabra Fast Forward indica que el commit al que hemos apuntado en el merge, está a continuación
# del commit master y no hay bifurcaciones en medio, es decir se puede llegar a esa rama
# siguiendo un simple commit del historial.
# Git simplifica las cosas moviendo el puntero hacia adelante, ya que
# no hay trabajos en el medio en otras bifurcaciones que sea necesario fusionar.
```

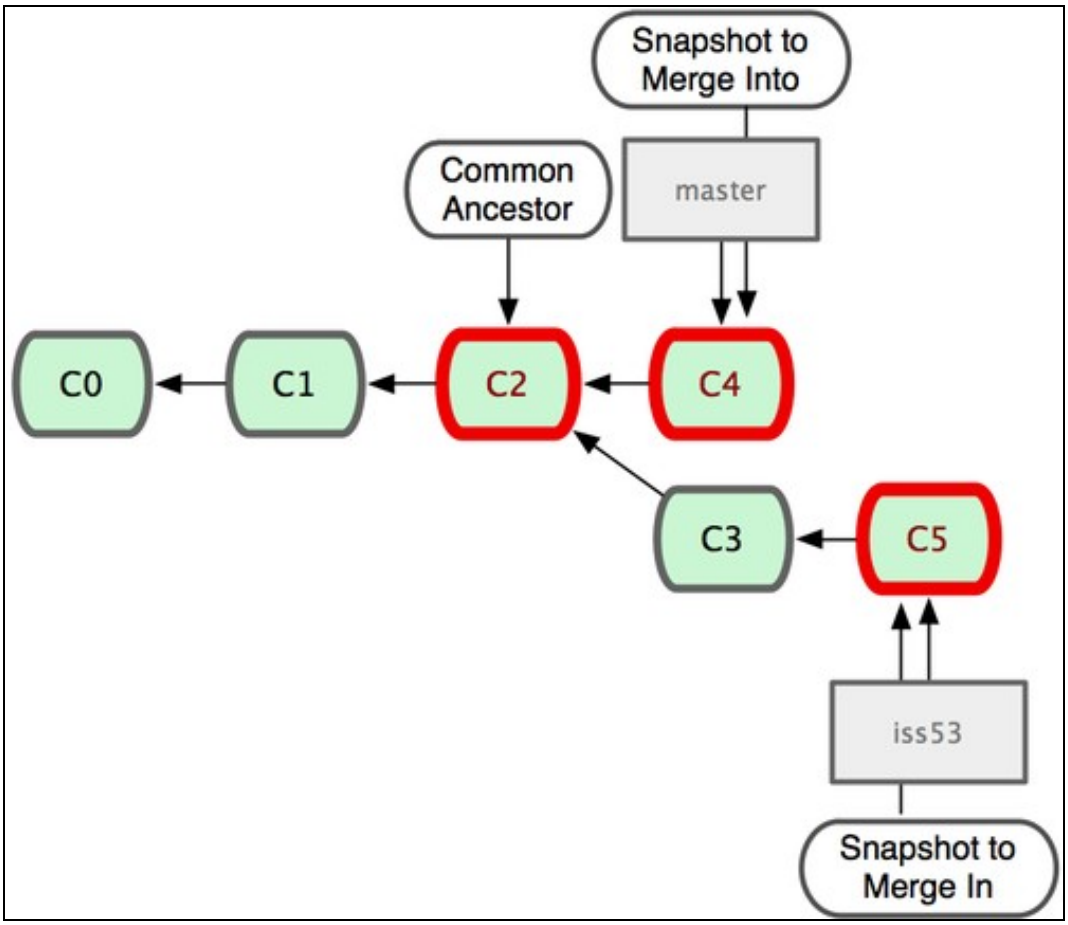


Una vez hemos terminado de enviar nuestros cambios

```
# Volvemos a la rama de la nueva opción en la que estábamos inmersos, y continuamos con nuestro trabajo.

# Pero para ello eliminaremos la branch hotfix, ya que no la necesitaremos más.
# Usaremos entonces el parámetro -d para eliminar esa rama.
git branch -d hotfix
Deleted branch hotfix (3a0874c)

# Continuamos trabajando en la rama iss53
git checkout iss53
Switched to branch "iss53"

vim index.html
git commit -a -m 'finished the new footer [issue 53]'
[iss53]: created ad82d7a: "finished the new footer [issue 53]"
 1 files changed, 1 insertions(+), 0 deletions(-)
```

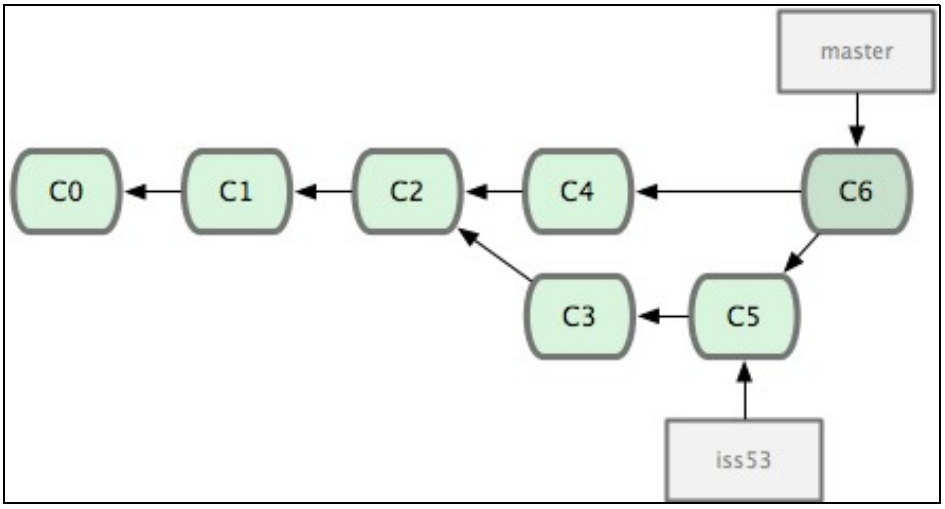


Si hemos terminado la programación de la nueva opción, se trataría ahora de fusionarla con el master. Para ello Git lo que hace, debido a que tenemos ramas intermedias y el padre de la nueva rama no está en conexión directa con el master, es buscar el mejor padre para hacer la fusión.



Y como resultado final obtendremos algo como lo siguiente, después de haber realizado las instrucciones:

```
git checkout master

git merge iss53
Merge made by recursive.
README |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
```



Por último sólo nos faltaría eliminar la branch iss53 ya que no la necesitamos, y podemos borrar el ticket en nuestro sistema de tracking con la nueva opción solicitada.

```
git branch -d iss53
```

Conflictos con merge

Si cuando se produce el merge existe algún conflicto, por ejemplo que hemos hecho una modificación de la misma parte de un fichero en dos ramas distintas, éste no continuará y se mostrará un mensaje de error.

Consultando con **git status** podremos ver dónde se ha producido el conflicto.

Tendremos que solucionar esa diferencia de contenido manualmente, añadir de nuevo cada uno de los ficheros con git add para marcarlos como solucionados y ejecutar de nuevo el merge.

Más información sobre conflictos en Merge en: http://git-scm.com/book/en/Git-Branching-Basic-Branching-and-Merging#Basic-Merge-Conflicts

GitHub un repositorio para usar con Git

Cuando estamos programando, a veces nos interesa compartir ese trabajo con la comunidad o con otro grupo de programadores que estén en el mismo proyecto que nosotros.

Para ello disponemos de diferentes opciones:

- Instalar nuestro propio servidor.
- Utilizar repositorios libres como github.

GitHub dispone de diferentes planes de precios para uso individual o de empresas. Si queremos que nuestro proyecto sea del tipo open-source, entonces GitHub es gratuito sin límite de proyectos y colaboradores. El único "pero" es que ese proyecto estará visible para todo el mundo, lo cuál es la filosofía propia de open-source.

Si queremos que nuestro proyecto sea privado, tendremos que acogernos a algún tipo de plan ofertado por github.

¿Cómo funciona github?

GitHub es un repositorio en línea que usa Git. Necesitamos lo siguientes requisitos antes de poder integrar GitHub con nuestro repositorio local:

- 1.- Tendremos que instalar Git.
- 2.- Tenemos que crear una cuenta en [GitHub].
- 3.- Crearemos un nuevo repositorio en GitHub.
- 3.- Configuraremos nuestro git para usar el repositorio de GitHub que hemos creado.

```
# Configuramos el nombre Global
git config --globar user.name "Tu nombre"

# Configuramos nuestro e-mail:
git config --global user.email pruebas@gmail.com
```

- Github nos da una dirección con la que poder trabajar.
- Tendremos que añadir el repositorio a nuestro sistema. (usaremos **git remote add**)
- Una vez comenzado el trabajo, lo podemos compartir con la comunidad. (usaremos **git push**)
- Para descargarnos un repositorio externo. (usaremos **git pull**)

```
# Añadimos el repositorio remoto.
# Copiando la url de nuestro repositorio remoto a partir del parámetro origin:
git remote add origin git://github.com/usuario/repositorio.git

# En lugar de poner como protocolo git:// se podría poner https:, etc..

# Una vez añadido el repositorio remoto podremos enviar nuestro repositorio local con:
```

```
# El parámetro master es la referencia al branch que queremos subir.
git push -u origin master

# Nos pedirá el usuario y la password.
# Enviará nuestro repositorio a GitHub.

# Si queremos descargarnos el proyecto de ese repositorio lo haremos con el comando: git pull
git pull --all
```

Cómo clonar un repositorio de github

Para clonar un repositorio ya existente en github a nuestra unidad local, haremos lo siguiente:

```
# Accedemos al Git Bash, a la carpeta o unidad dónde queramos clonar.
# Una vez allí dentro teclearemos el comando:
git clone https://github.com/usuario/repositorio.git

# Si queremos que lo clone en un directorio distinto, haremos:
git clone https://github.com/usuario/repositorio.git nuevodirectorio

# Si queremos comprobar los repositorios remotos que tenemos configurados:
git remote

# "origin" ?es el nombre predeterminado que le da Git al servidor del que clonaste

# Si queremos ver las URL de esos repositorios remotos:
git remote -v
```

Herramientas gráficas de gestión de github

Github dispone de aplicaciones gráficas gratuitas para su gestión en: https://github.com/

- Versión Windows
- Versión Mac

Libro recomendadísimo sobre Git

En la siguiente dirección tenéis un libro completísimo sobre el uso de Git, dónde se complementan todas las opciones y apartados que no han sido vistos aquí:

http://git-scm.com/book

Otras referencias

Servicios de host en la nube que usan Git:

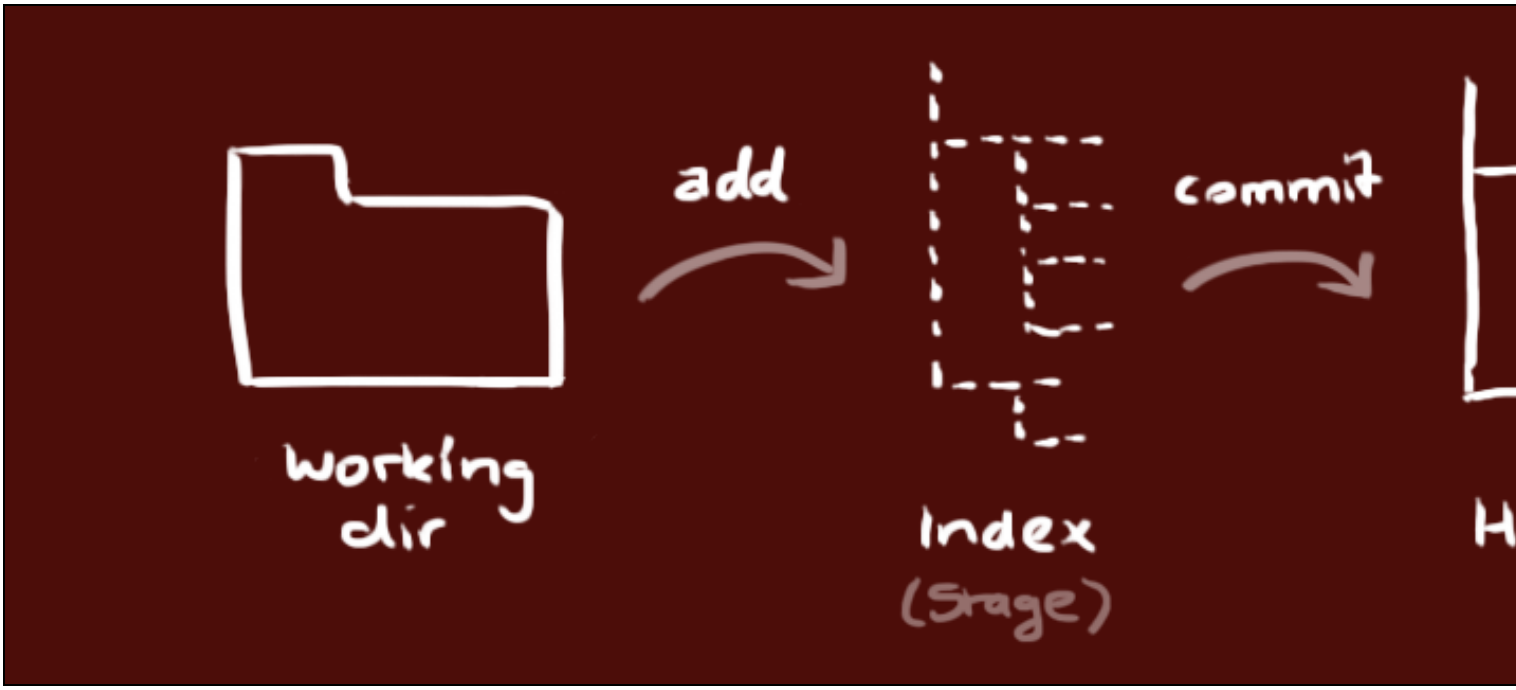
- http://rogerdudler.github.com/git-guide/index.es.html Guia sencilla sobre Git
- http://www.genbetadev.com/herramientas/referencia-rapida-de-comandos-de-git Referencia rápida de comandos Git.
- http://www.heroku.com/ (podrás subir aplicaciones de Node.js, python, etc..
- http://nodester.com/ (para aplicaciones de Node.js)

RESUMEN Y USO BÁSICO de Git por comandos

- Guía Sencilla de Git: http://rogerdudler.github.io/git-guide/index.es.html
- http://backlogtool.com/git-guide/en/

Un repositorio git esta compuesto por tres "árboles" administrados por git.

1. El primero es tu **Directorio de trabajo** que contiene los archivos.
2. El segundo es el **Index** que actua como una zona intermedia.
3. El último es el **HEAD** que apunta al último commit realizado.



Crear un repositorio

```
# Para crear un repositorio:
# 1.- Acceder a la carpeta que deseemos.
# 2.- Teclear git init
# 3.- Se habrá creado una carpeta oculta .git que gestionará todas las copias y control de versiones.
git init
```

Comprobar modificaciones en el repositorio

```
git status
```

Notificar cambios al repositorio

```
# Para notificar que se han modificado ficheros en el repositorio y enviarlos a la Stage Área o área temporal:
git add nombrefichero

# O podemos añadir TODOS los ficheros modificados tecleando:
git add .

# Para ver esa notificación teclear:
git status
```

Commit

```
# Para confirmar los cambios registrados en los archivos y hacer una nueva copia
git commit -m "Mi primer commit."

# Si queremos enviar todos los ficheros modificados independientemente de si se han enviado a la
# staging áre usaremos el parámetro -a:
git commit -a -m "Mi primer commit con todos los archivos modificados."
```

Envío de cambios al repositorio remoto en GitHub

```
# Nuestros cambios están en el HEAD local de nuestra máquina.
# Para enviar los cambios al repositorio remoto ejecutar:
git push origin master  (reemplazar master por la rama a dónde queremos enviar los cambios)
```

Configuración de Git para poder usar GitHub.com

```
# Configuramos el nombre Global
git config --globar user.name "Tu nombre"

# Configuramos nuestro e-mail:
git config --global user.email pruebas@gmail.com
```

Añadir repositorio remoto de GitHub

```
# Añadimos el repositorio remoto.
# Copiando la url de nuestro repositorio remoto a partir del parámetro origin:
git remote add origin git://github.com/usuario/repositorio.git

# En lugar de poner como protocolo git:// se podría poner https:, etc..

# Una vez añadido el repositorio remoto podremos enviar nuestro repositorio local con:
# El parámetro master es la referencia al branch que queremos subir.
git push -u origin master

# Nos pedirá el usuario y la password.
# Enviará nuestro repositorio a GitHub.

# Si queremos descargarnos el proyecto de ese repositorio lo haremos con el comando: git pull
git pull --all

# Si queremos comprobar los repositorios remotos que tenemos configurados:
git remote

# "origin" ?es el nombre predeterminado que le da Git al servidor del que clonaste
# Si queremos ver las URL de esos repositorios remotos:
git remote -v

# Para obtener los datos del repositorio remoto:
git fetch

# Este comando recupera todos los Datos del Proyecto (no los ficheros) remoto que no tengas todavía.
# Después de hacer esto, deberías tener referencias a todas las ramas del repositorio remoto, que puedes unir o inspeccionar en cual

# Si clonas un repositorio, el comando añade automáticamente ese repositorio remoto con el nombre de "origin".
# Por tanto, git fetch origin recupera toda la información enviada a ese servidor desde que lo clonaste (o desde la última vez que e
# Es importante tener en cuenta que el comando fetch sólo recupera la información y la pone en tu repositorio local
# no la une automáticamente con tu trabajo ni modifica aquello en lo que estás trabajando. Tendrás que unir ambos manualmente a post
```

Clonado de un repositorio remoto de GitHub.com a local

```
# Accedemos al Git Bash, a la carpeta o unidad dónde queramos clonar.
# Una vez allí dentro teclearemos el comando:
git clone https://github.com/usuario/repositorio.git

# Si queremos que lo clone en un directorio distinto, haremos:
git clone https://github.com/usuario/repositorio.git nuevodirectorio
```

```
# Si clonas un repositorio, el comando añade automáticamente ese repositorio remoto con el nombre de "origin".
```

Actualizar tu repositorio local al commit más nuevo

```
# Ejecutar en el directorio de trabajo para bajar y fusionar con los cambios remotos.
git pull
```

Revertir los cambios de un commit concreto

```
# Vemos el log de los commit:
git log --pretty=oneline

# Se muestra algo como ésto
5542cd7844e3a035542cd7848c19482029963813 cambios 3
0bbfc3d4b41c8cb572ad780a346ddfbb0f5cfed5 2
f03af74db09f6f1802024af5c1371756b5b9a557 1
f64da54ea58649f863cbddfdeadf67dbd046d7d8 :boom::camel: Added .gitattributes & .gitignore files

# Para revertir el commit escribiremos:
git revert f03af74db09f6f1802024af5c1371756b5b9a557

# Éste cambio teóricamente no afectará a los commits siguientes. Deshace todo lo que se hizo en el commit identificado.
# pero mantiene todo lo que se ha hecho después de éste. El commando revert hace un nuevo commit, que se podría volver a deshacer.
```

Volver a una versión anterior de Git perdiendo lo posterior a ese commit

```
# Vemos el log de los commit:
git log --pretty=oneline

# Se muestra algo como ésto
5542cd7844e3a035542cd7848c19482029963813 cambios 3
0bbfc3d4b41c8cb572ad780a346ddfbb0f5cfed5 2
f03af74db09f6f1802024af5c1371756b5b9a557 1
f64da54ea58649f863cbddfdeadf67dbd046d7d8 :boom::camel: Added .gitattributes & .gitignore files

# Si queremos volver a como estaba todo cuando hicimos el commit 1 (perdiendo lo posterior al commit 1)
git reset --hard f03af74db09f6f1802024af5c1371756b5b9a557

# No hacer git push al directorio remoto por que perderemos todo lo que teníamos.

# Para volver al presente basta con hacer un git pull. De esta forma se sincronizará el local con el contenido del repositorio remot
```

Anotaciones

```
cd /home/forge/default
git pull origin master
composer install
php artisan migrate
```

--Veiga (discusión) 18:44 14 abr 2015 (CEST)