# Tutorial básico de GIT – Parte 3

**Introducción**

En este cuarto, y último post, vamos a trabajar la operación merge en base a los conceptos introducidos en el anterior post. Crearemos un un conflicto manualmente y seguidamente lo resolveremos.

- Crear un conflicto
- Resolver un conflicto

Crear un conflicto

En esta sección, vamos a aprender a resolver un conflicto. Para ello vamos a crear manualmente un conflicto utilizando nuestros dos repositorios existentes «tutorial» y «tutorial2».

*Trabajando en tutorial*

En primer lugar, abriremos el archivo «sample.txt» en el directorio «tutorial». Añadiremos texto al fichero y haremos un commit.

```
$ echo "Each language has its purpose, but do not program in
COBOL if you can avoid it." >> sample.txt
$ git add sample.txt
$ git commit -m "added new programming advice"
[master ef95d28] added new programming advice
 1 file changed, 1 insertion(+)
```

*Trabajando en tutorial2*

A continuación, abriremos el archivo «sample.txt» en el directorio «tutorial2». Añadiremos texto al fichero y haremos un commit.

```
$ echo "The spirit and intent of the program should be retained throughout." >> sample.txt
$ git add sample.txt
$ git commit -m "added new coding advice"
[master f4ddc9c] added new coding advice
 1 file changed, 1 insertion(+)
```

Ahora haremos un push para subir los cambios de «tutorial2» al repositorio remoto.

```
$ git push
Username:
Password:
Counting objects: 3, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 356 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://gitlab.com/nebur/tutorial.git
   178cb16..f4ddc9c  master -> master
```

En nuestro repositorio remoto actual, el archivo «sample.txt» contiene la tercera línea «The spirit and intent of the program should be retained throughout.»  y se ha realizado el commit al histórico.

*Trabajando en tutorial*

Ahora, vamos a hacer un push del commit realizado en nuestro repositorio «tutorial» al repositorio remoto.

```
$ git push
Username:
Password:
To https://gitlab.com/nebur/tutorial.git
 ! [rejected]        master -> master (fetch first)
```

```
error: failed to push some refs to
'https://gitlab.com/nebur/tutorial.git'
hint: Updates were rejected because the remote contains work
that you do
hint: not have locally. This is usually caused by another
repository pushing
hint: to the same ref. You may want to first integrate the
remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help'
for details.
```

Como podemos observar, Git retorna un conflicto y rechaza el push.

## Resolver un conflicto

Con el fin de realizar el push del cambio en «tutorial» al repositorio remoto, vamos a tener que resolver el conflicto manualmente. Vamos a ejecutar un pull para adquirir los cambios más recientes desdel repositorio remoto.

### *Trabajando en tutorial*

Ejecutaremos el siguiente comando

```
$ git pull origin master
```

```
Username:
Password:
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From https://gitlab.com/nebur/tutorial
 * branch            master     -> FETCH_HEAD
   178cb16..f4ddc9c  master     -> origin/master
Auto-merging sample.txt
CONFLICT (content): Merge conflict in sample.txt
Automatic merge failed; fix conflicts and then commit the
```

result.

Debería aparecer un mensaje advirtiéndonos de un conflicto de merge.

En abrir «sample.txt», podremos ver una sección del archivo delimitada por marcadores que ha añadido Git para indicarnos donde se encuentra el conflicto.

```
$ cat sample.txt
After three days without programming, life becomes meaningless.
new line text
<<<<<<< HEAD Each language has its purpose, but do not program in COBOL if you can avoid it. ======= The spirit and intent of the program should be retained throughout. >>>>>>> f4ddc9c9a3d3329f8ad799ae582adf9efe631211
```

Vamos a resolver el conflicto aceptando los cambios y quitando el marcador.

```
$ cat sample.txt
After three days without programming, life becomes meaningless.
new line text
Each language has its purpose, but do not program in COBOL if you can avoid it.
The spirit and intent of the program should be retained throughout.
```

Una vez editado el fichero podremos proceder a realizar el commit

```
$ git add sample.txt
$ git commit -m "merge"
[master 4351226] merge
```

Ahora estamos al día con los últimos cambios del repositorio remoto.
Podemos comprobar la integridad del histórico del repositorio usando el comando «log». La opción –graph mostrará el histórico del branch en un formato gráfico y la opción

–oneline tratará de compactar el mensaje de salida.

```
$ git log --graph --oneline
*   4351226 merge
|\
| * f4ddc9c added new coding advice
* | ef95d28 added new programming advice
|/
* 178cb16 new line appened
* 0742011 first commit
```

Esto indica que las dos historias se han fusionado de manera segura con un nuevo commit del merge realizado a mano.

Podemos hacer un push con la seguridad seguridad de que este cambio no provocará un conflicto en el repositorio remoto.

**Observaciones**

Y con este último post damos por finalizada esta serie de tutoriales sobre Git. De  una forma básica y cercana hemos visto como trabajar con este sistema de control de versiones. No hemos mostrado todo su potencial, pero si introducido conceptos básico y las herramientas para empezar a trabajar con él. En una nueva serie profundizarmos en conceptos mucho más potente, por ahora podemos empezar  experimentar con el.

Ruben.

---

# Tutorial básico de GIT – Parte 2

**Introducción**

En este tercer post vamos a trabajar la comparación con un repositorio remoto e introduciremos el concepto merge. Como servicio de repositorio Git remoto usaremos GitLab por algunas razones. Principalmente porqué es un servicio liviano, sin añadidos, simple, con gestión de reportes de errores, wiki y que nos permite realizar repositorios privados. Es simple (en comparación con otros servicios mas potentes como GitHub) y nos permite realizar repositorios privados, es decir perfecto para empezar a hacer pruebas y que estas no sean visible a toda la comunidad (pues es innecesario)

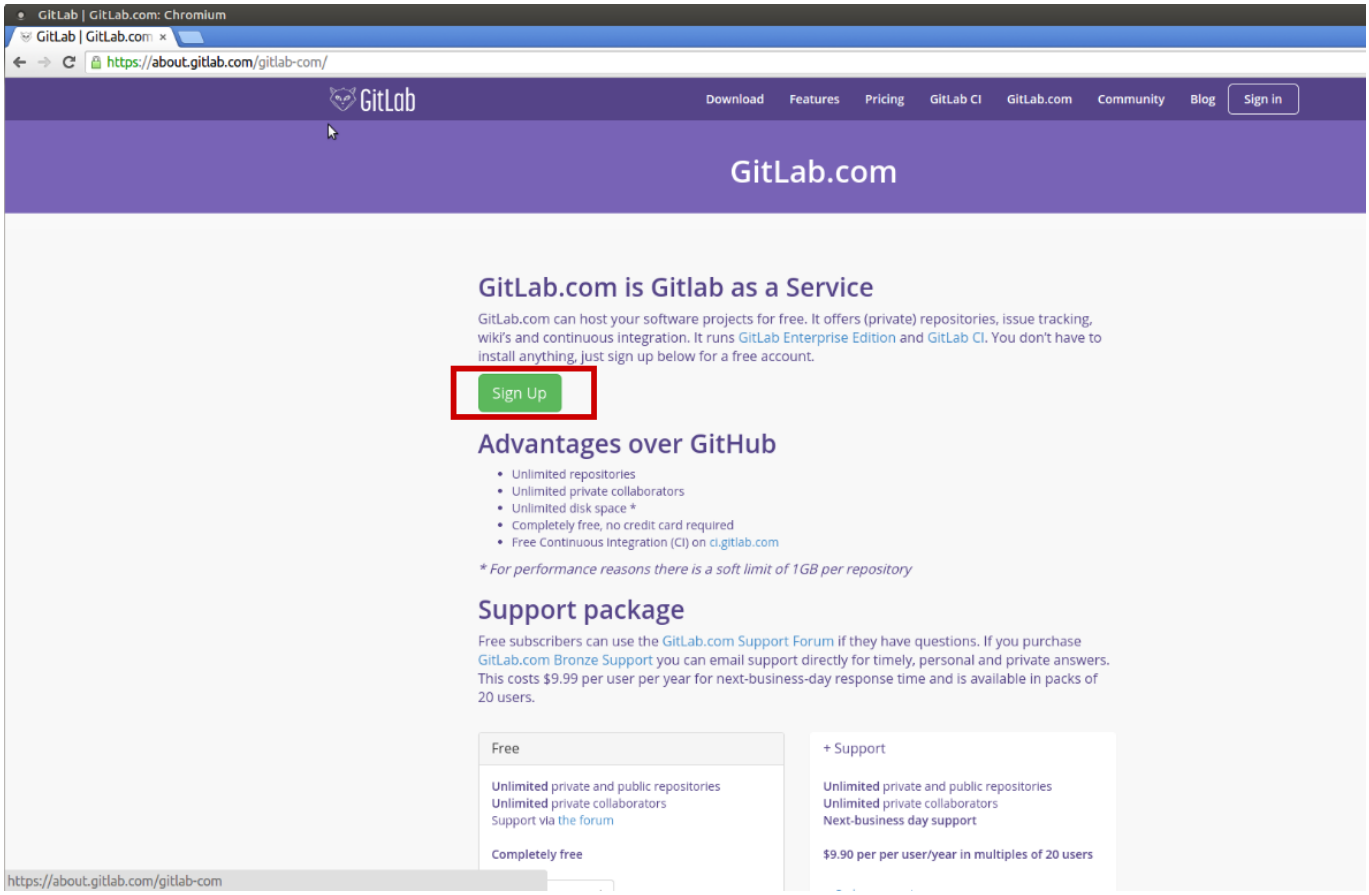
Vamos a tratar:

- Compartir un repositorio
  - Crear un repositorio remoto en GitLab
  - Hacer push a un repositorio remoto
  - Clonar un repositorio remoto
  - Realizar un push des de un repositorio clonado
  - Hacer pull des de un repositorio
- Merge de histórico
  - Hacer merge de un histórico
  - Resolver un conflicto

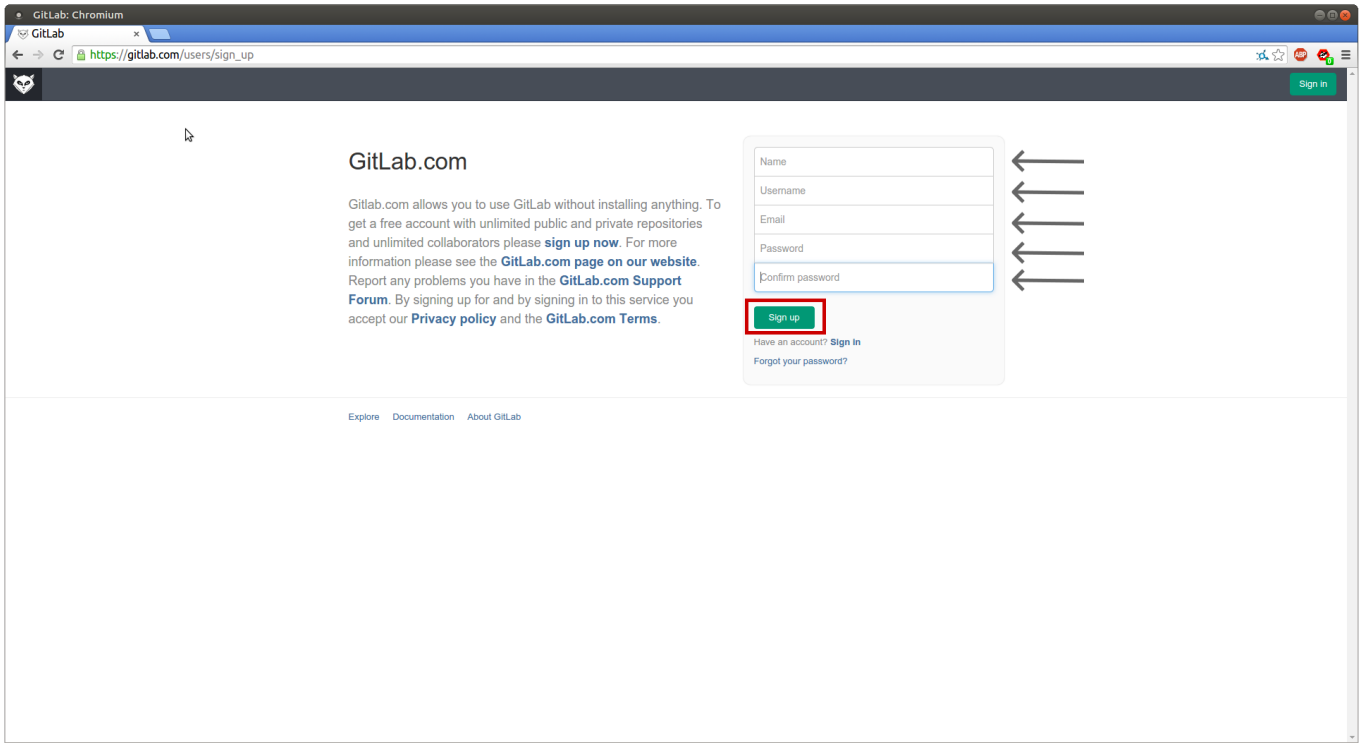
## Compartir un repositorio

### *Crear un repositorio remoto en GitLab*

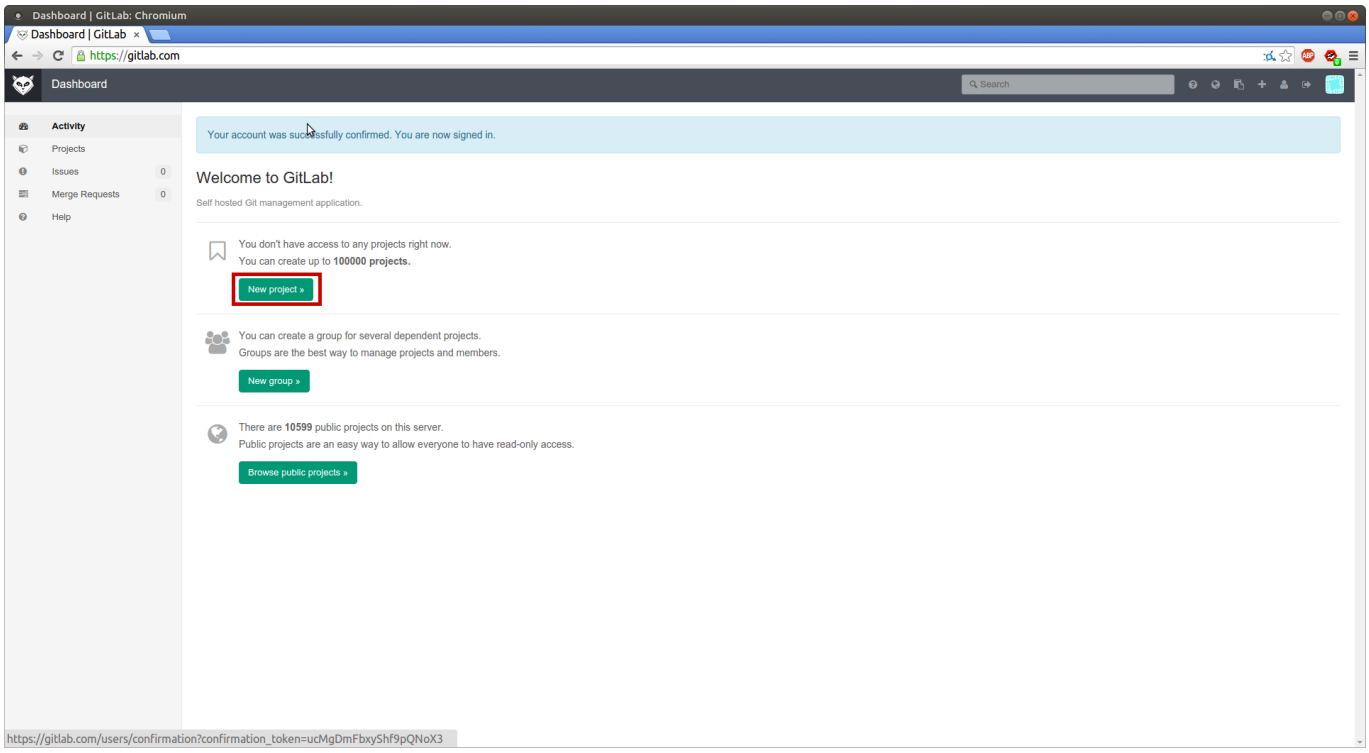
Como hemos comentado antes crearemos el repositorio en GitLab. Primera mente accederemos a la web del servicio y crearemos un usuario desde la siguiente url: https://about.gitlab.com/gitlab-com/



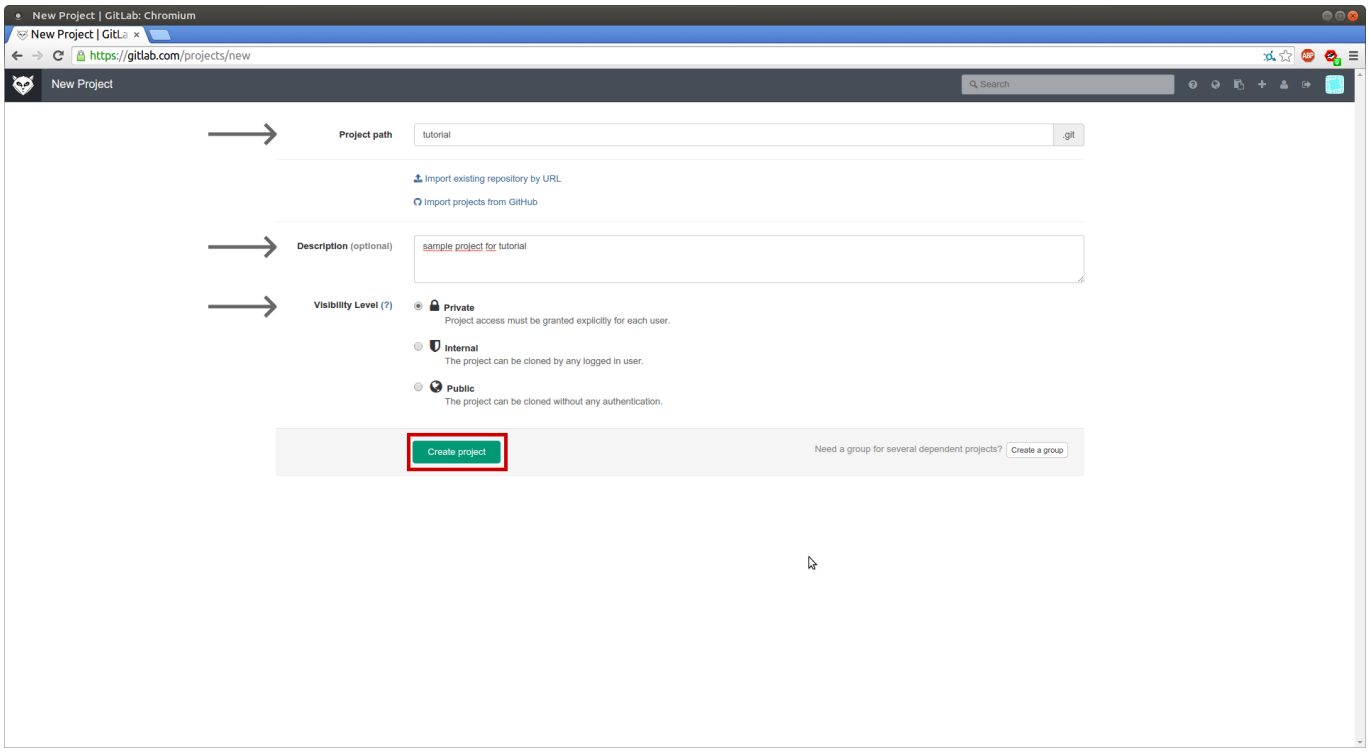

Rellenamos los campos necesarios para crear nuestra cuenta:



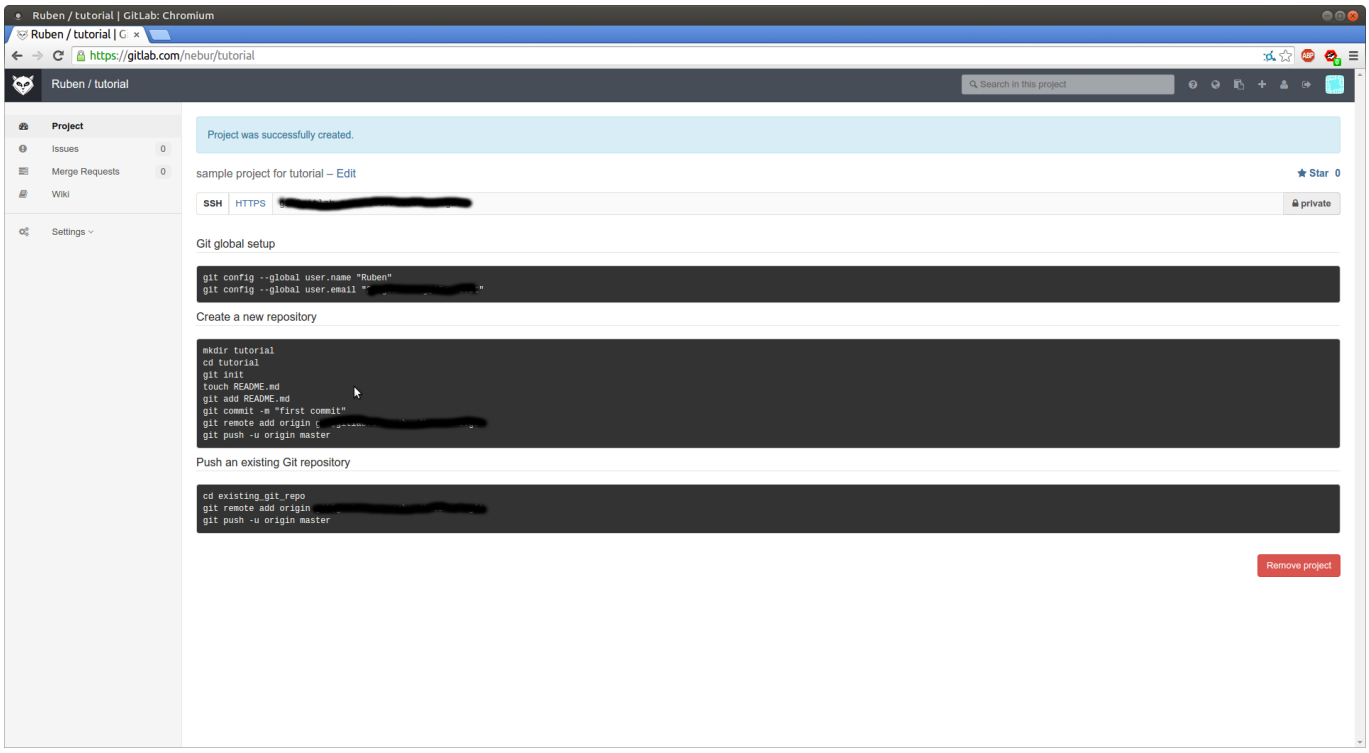

Una vez creada la cuenta entraremos con nuestro usuario y crearemos un proyecto nuevo. El proyecto será privado, puesto que no tienen ningún interés para la comunidad un repositorio donde se hacen pruebas:




Rellenamos los campos necesarios para crear nuestra nuestro repositorio como se muestra a continuación:




Finalmente podemos observar que nuestro repositorio se ha creado correctamente:




## *Hacer push a un repositorio remoto*

Haremos un push del repositorio local «tutorial» que hemos

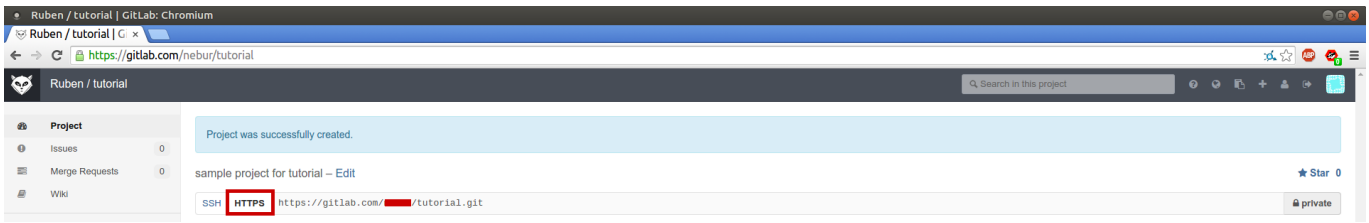
creado anteriormente en la primera parte del tutorial básico de Git.

Podemos utilizar un alias o apodo para un repositorio remoto. Esto es útil ya que no necesitamos recordar la larga dirección del repositorio remoto cada vez que tenemos la intención de hacer un push. En este tutorial, vamos a registrar un nombre de repositorio remoto como «origin».

Para añadir un repositorio remoto, utilizaremos el comando «remote». <name> se utiliza como un alias de un repositorio remoto, seguido de <url> con la URL del repositorio remoto.

```
$ git remote add
```

Ejecutamos el comando utilizando la url del repositorio repoto que hemos creados anteriormente. El nuevo repositorio remoto tendrá el alias «origin»




```
$ git remote add origin https://gitlab.com/[user_name]/tutorial.git
```

El repositorio remoto llamado «origin» se utiliza por defecto si se omite el nombre remoto al hacer push/pull. Esto se debe a «origin» es comúnmente usado como un nombre remoto por convención.

Para hacer un push de nuestros cambios al repositorio remoto, utilizaremos el comando «push». Asignaremos la dirección en <repository> y el nombre del branch en <refspec>, al cual queremos hacer el push. Hablaremos de lo branchs en Git en el tutorial avanzado de Git más adelante.

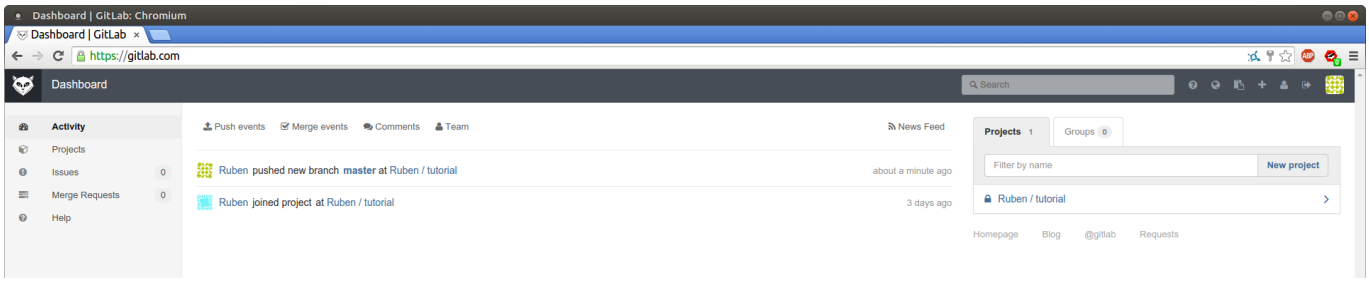
```
$ git push  ...
```

Ejecutaremos el siguiente comando para insertar un commit al repositorio remoto «origin». Si especifica la opción -u al ejecutar el comando, se puede omitir el nombre del branch la próxima vez que se hagamos un push al repositorio remoto. Cuando empujas a un remoto vacante sin embargo, se debe especificar el repositorio remoto y nombre de la rama.

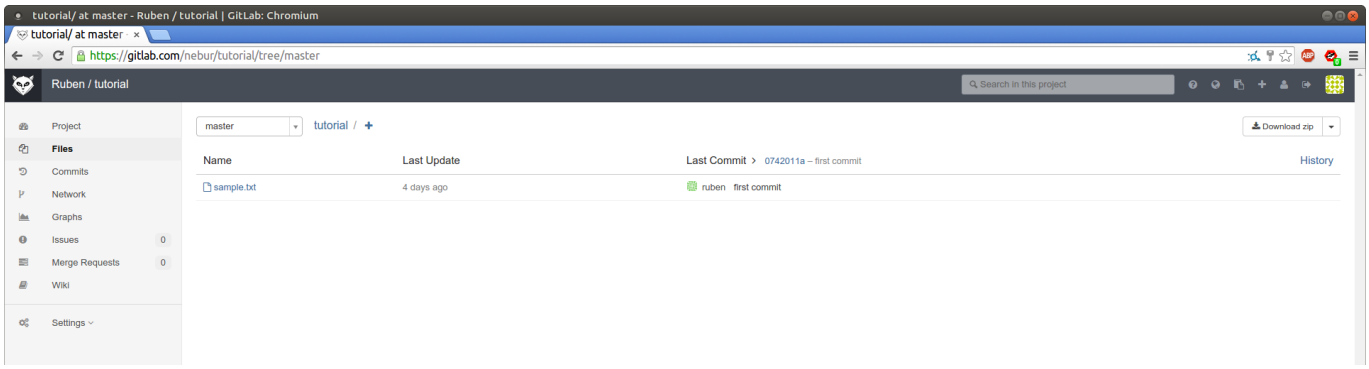
Cuando se nos pregunte por el nombre de usuario y contraseña, introduzca los creados en GitLab.

```
$ git push -u origin master
Username:
Password:
Counting objects: 3, done.
Writing objects: 100% (3/3), 245 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://monkey.backlogtool.com/git/BLGGIT/tutorial.git
 * [new branch]      master -> master
```

Abra la página de Git en Cartera. Usted encontrará que una nueva actualización que corresponde a su empuje al repositorio remoto se ha incluido en las actualizaciones recientes.




El archivo que hemos subido mediante push también se ha añadido en la lista de archivos del repositorio remoto y podemos observar el comentario.



*Clonar un repositorio remoto*

Realizaremos una copia de un repositorio remoto para poder empezar a trabajar con él en el equipo local.

Ahora, vamos a asumir el papel de otro miembro en el equipo y clonar el repositorio remoto existente en otro directorio llamado «tutorial 2».

Para ello utilizaremos el comando «clon» para copiar un repositorio remoto como se muestra en el siguiente ejemplo. Substitiremos <repository> con la URL del repositorio remoto y <directory> con el nombre del nuevo directorio en el que se descargaran los contenidos remotos.

```
$ git clone <repository> <directory>
```

Ejecutando el siguiente comando, el repositorio remoto se copiará en el directorio tutorial2.

```
$ git clone  https://gitlab.com/<user_name>/tutorial.git tutorial2
Cloning into 'tutorial2'...
Username:
Password:
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
Checking connectivity... done.
```

Para comprobar que la clonación se ha ejecutado correctamente, ejecutaremos el siguiente comando para comprobar el contenido de sample.txt del directorio clonado «tutorial2».

```
$ cat sample.txt
After three days without programming, life becomes meaningless.
```

## *Realizar un push des de un repositorio clonado*

Vamos a realizar un push des de un reporitorio clonado. Para ello trabajaremos en el repositorio clonado **tutorial2**

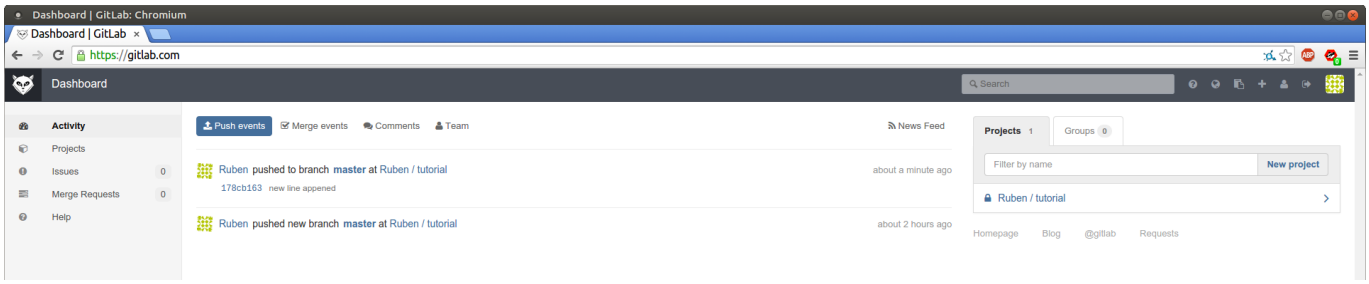
Primeramente añadiremos texto al fichero sample.txt

```
echo "new line text" >> sample.txt
$ git add sample.txt
$ git commit -m "new line appened"
[master 178cb16] new line appened
 1 file changed, 1 insertion(+)
```

Ahora realizaremos el push de este nuevo commit al repositorio remoto. Podemos omitir el repositorio y el branch cuando hacemos un push en el directorio de un repositorio clonado.

```
$git push
Username:
Password:
Counting objects: 3, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 282 bytes | 0 bytes/s, done.
Total 3 (delta 1), reused 0 (delta 0)
To https://gitlab.com/nebur/tutorial.git
   0742011..178cb16  master -> master
```

Podemos observar el nuevo push en GitLab. Podremos verlo en el Dashboard.




## *Hacer pull des de un repositorio*

En esta sección, vamos a hacer un pull del último cambio del

repositorio remoto a nuestro repositorio local (**en el directorio tutorial**).

Ahora que nuestro repositorio remoto está actualizado con los cambios de «tutorial2», vamos a hacer un pull para obtener el cambio y sincronizar nuestro repositorio inicial en el directorio «tutorial».

Para ejecutar un pull, utilizaremos el comando «pull». Si no se incluye el nombre del repositorio, el pull se hará en el repositorio con alias «origin».

```
$ git pull  ...
```

Ejecutaremos el siguiente comando

```
$ git pull origin master
Username:
Password:
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From https://gitlab.com/nebur/tutorial
 * branch            master     -> FETCH_HEAD
   0742011..178cb16  master     -> origin/master
Updating 0742011..178cb16
Fast-forward
 sample.txt | 1 +
 1 file changed, 1 insertion(+)
```

Ahora vamos a comprobar que el historico esta actualizado con el comando log

```
$ git log
commit 178cb1635855ea757a3bf6ed748f0096cdc1f6de
Author: ruben <ruben11set@hotmail.com>
Date:   Tue Feb 3 21:13:52 2015 +0100

    new line appened

commit 0742011aaf70e0e3a611fb22500c73d633f755c1
```

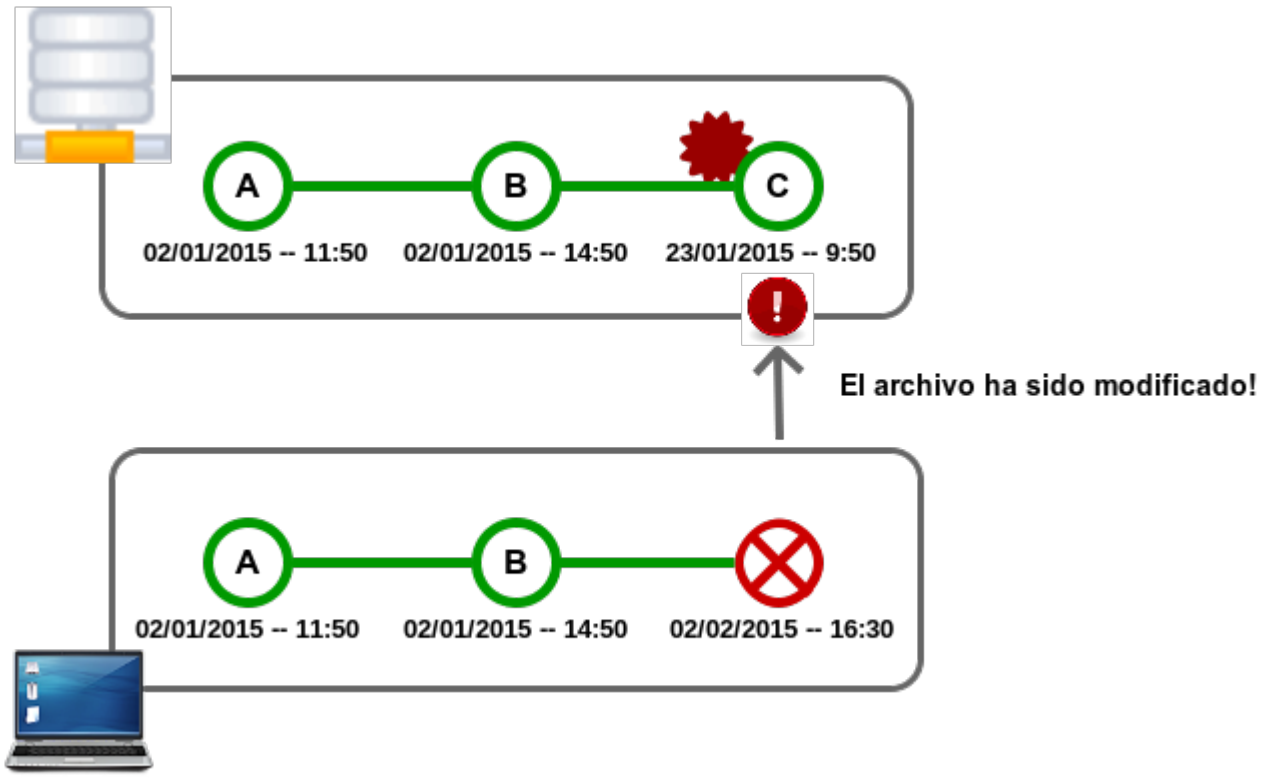
```
Author: ruben <ruben11set@hotmail.com>
Date:   Fri Jan 30 18:08:42 2015 +0100

    first commit
```

El nuevo commit que hemos añadido en «tutorial2» ahora aparece en la lista de registro de la histórico del repositorio «tutorial». Con el siguiente comando comprobaremos el contenido del fichero sample.txt

```
$ cat sample.txt
After three days without programming, life becomes meaningless.
new line text
```

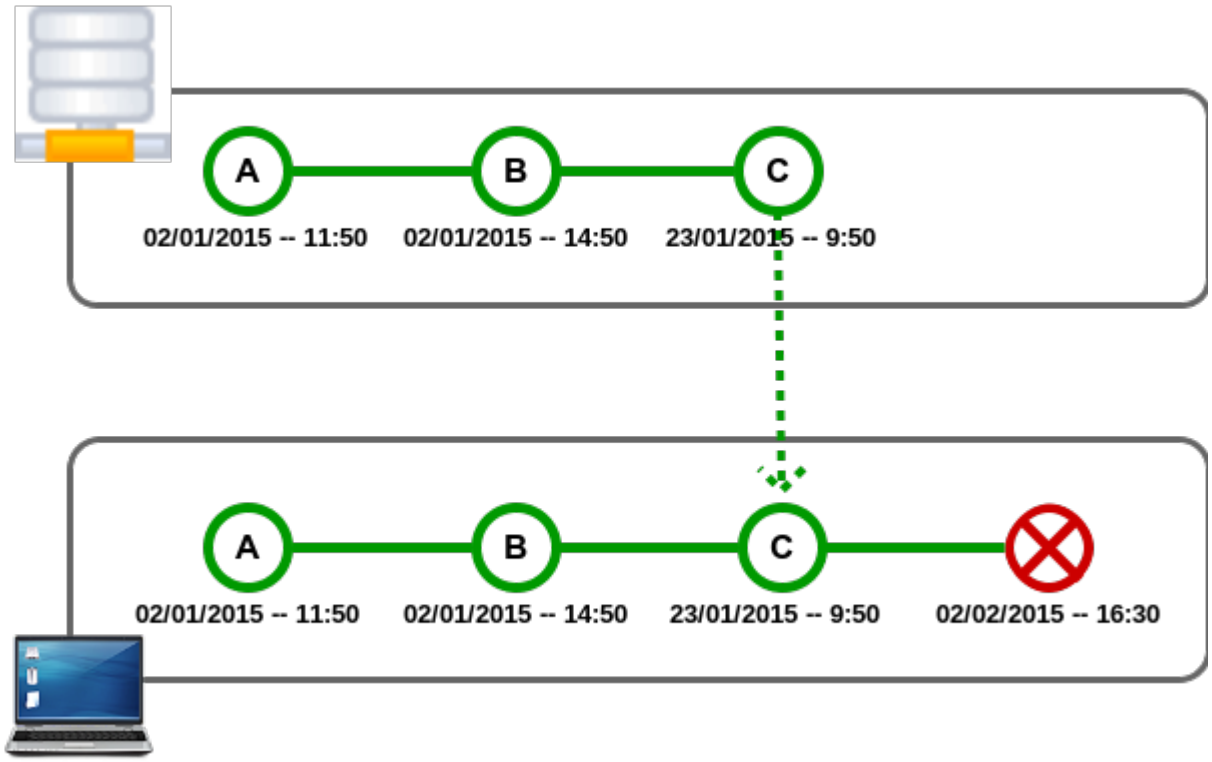
## Merge de histórico

### *Hacer merge de un histórico*




Un push puede ser rechazado si nuestro repositorio local no

está actualizado, posiblemente porque hay algunos cambios añadidos por otros en el repositorio remoto que aún no tenemos en nuestro repositorio local todavía.




Si ese es el caso, hacer un «merge» deberemos obtener el último cambio del repositorio remoto antes de hacer un push. Git de esta manera esto para asegurarse de que los cambios realizados por otros miembros quedan retenidos en el histórico (Commit C en la figura anterior).

Durante un «merge», Git intentará aplicar automáticamente los cambios de la historia y los combina con la rama actual. Sin embargo, si hay un conflicto debido a los cambios, Git retornará un error. En este caso se nos indicará que debemos resolver el conflicto de forma manual.

*Resolver un conflicto*

Como se describe en el apartado anterior, Git intentará aplicar automáticamente los cambios que enlazará con un

historial de cambios existente cuando se ejecuta «merge».

Aveces un «merge» puede fallar y puede suceder cuando hay un conflicto. Si dos o más miembros hacen cambios en la misma parte de un archivo en las dos branchs (remota y local en este caso) que está siendo fusionadas, Git no será capaz de hacerlo automáticamente y retornará un conflicto de combinación.

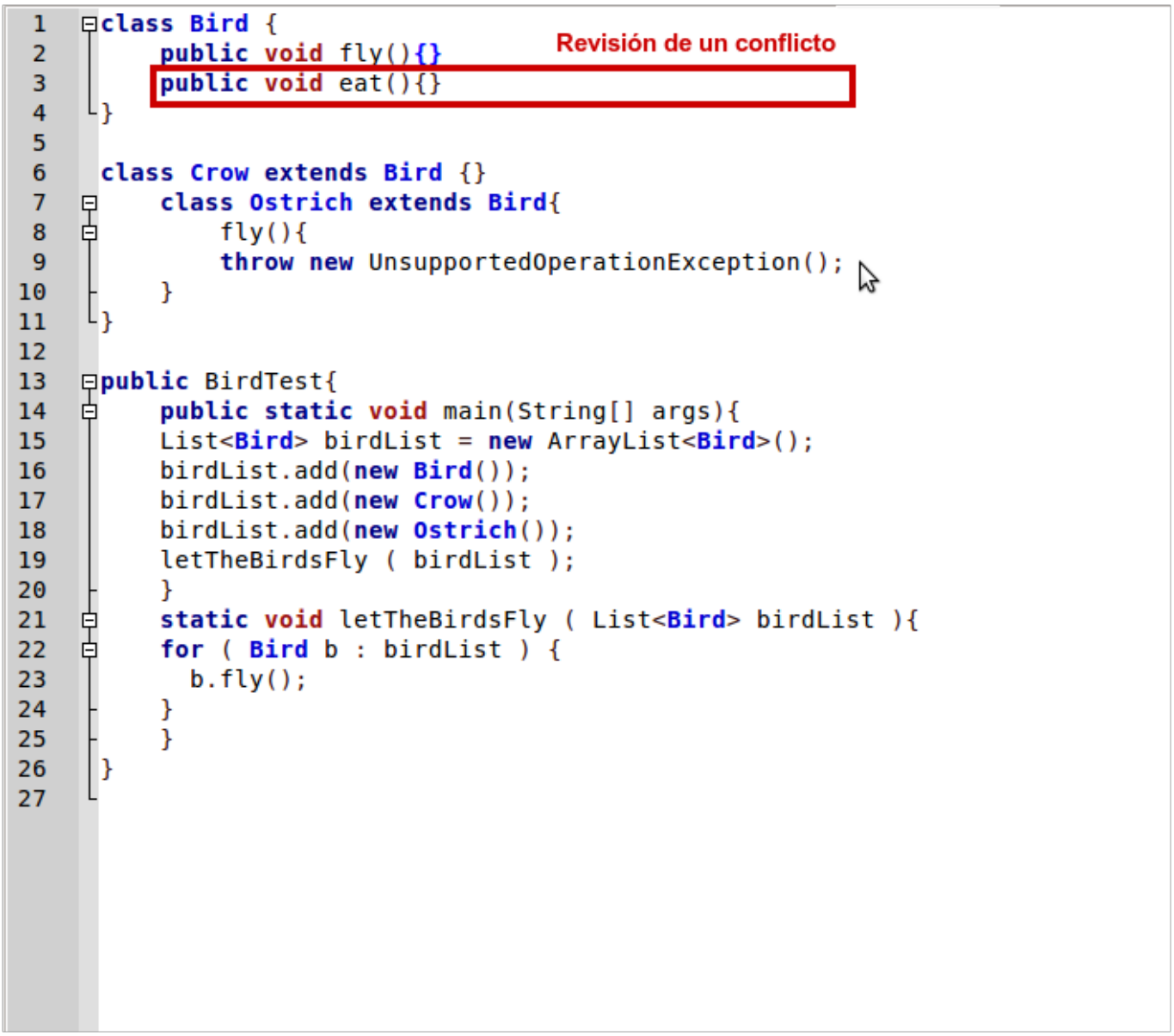
Cuando esto suceda, Git agregará algunos marcadores de resolución de conflictos al archivo. Los marcadores actúan como un indicador para ayudarnos a entender las secciones en el contenido del archivo en conflicto que debemos resolver manualmente.

Ejemplo de un conflicto

```
1  class Bird {
2      public void fly(){}
3      <<<<<<< HEAD
4      eatOnce
5      =======
6      public void eat(){}
7      >>>>>>> 12jh76093675093h78fjn76575a76556f8h9
8  }
9
10 class Crow extends Bird {}
11     class Ostrich extends Bird{
12         fly(){
13         throw new UnsupportedOperationException();
14     }
15 }
16
17 public BirdTest{
18     public static void main(String[] args){
19     List<Bird> birdList = new ArrayList<Bird>();
20     birdList.add(new Bird());
21     birdList.add(new Crow());
22     birdList.add(new Ostrich());
23     letTheBirdsFly ( birdList );
24     }
25     static void letTheBirdsFly ( List<Bird> birdList ){
26     for ( Bird b : birdList ) {
27       b.fly();
28     }
29     }
30 }
31
```

Todo lo anterior «=====» es su contenido local, y todo lo que se encuentra a continuación se trata de la rama remota.

Podemos resolver las partes en conflicto tal y como se muestra a continuación. Ahora ya está listo para proceder con la creación de un «merge commit».




**Observaciones**

En este tercer post hemos visto como trabajar con un repositorio remoto y hemos introducido el concepto merge. En

el próximo, y último  post de esta serie, mostraremos paso a paso como trabajar con merge.

Ruben.

---

# Tutorial básico de GIT – Parte 1

**Introducción**

En este segundo post vamos a preparar el entorno para poder trabajar y realizar algunas operaciones básica. También introduciremos algunos conceptos nuevos para trabajar con repositorios remotos, los cuales profundizaremos en el próximo post

Vamos a tratar:

- Operaciones básicas
  - Instalando Git
  - Configuraciones por defecto
  - Crear un nuevo repositorio

        - Hacer commit con un fichero
    - Trabajar con un repositorio remoto
        - Hacer push a un repositorio remoto
        - Hacer clone de un repositorio remoto
        - Hacer pull de un repositorio remoto

## Operaciones básica

### *Instalando Git*

Bien antes de empezar a trabajar con Git deberemos preparar el entorno de trabajo. Esto es realmente fácil simplemente lo instalamos en GNU\Linux haciendo uso del sistema de paquetes

Para sistema basados en rpm como Fedora:

```
$ yum install git
```

Para sistemas basados en deb como Debian:

```
$ sudo apt-get install git
```

Acabada la instalación y a modo de prueba podemos ejecutar el siguiente comando que nos devolverá la versión instalada:

```
git --version
git version 2.1.0
```

### *Configuraciones por defecto*

Ahora, vamos a configurar el nombre de usuario por defecto y dirección de correo electrónico para que Git identifique la persona que commitea los cambios. Esta configuración sólo hay que hacer una vez.

La configuración de la consola Git se guarda en el archivo .gitconfig en el directorio home del usuario. Se puede editar

manualmente el archivo, pero en este tutorial vamos a utilizar el comando «config».

```
$ git config --global user.name "<Username>"
$ git config --global user.email "<Email address>"
```

Configuramos la salida de color de Git

```
$ git config --global color.ui auto
```

También puede configurar los alias para los comandos de Git. Por ejemplo, se puede abreviar «checkout» para «co» y usarlo para ejecutar el comando.

```
$ git config --global alias.co checkout
```

*Crear un nuevo repositorio*

Vamos a empezar por la creación de un nuevo repositorio local. Nuestro objetivo es crear un directorio de pruebas y ponerlo bajo control de versión con Git. Utilizaremos este directorio lo largo del tutorial.

Crearemos un directorio tutorial en cualquier lugar de nuestro equipo. Después accederemos al directorio y utilizaremos el comando «init» para convertir ese directorio en un repositorio Git local.

```
$ git init
```

Mediante los siguientes comandos crearemos el nuevo directorio tutorial en un repositorio Git.

```
$ mkdir ~/tutorial
$ cd ~/tutorial
$ git init
Initialized empty Git repository in /home/yourname/tutorial/.git/
```

*Hacer commit de un fichero*

En el directorio del tutorial que hemos creado anteriormente, vamos a añadir un nuevo archivo y lo registraremos en el repositorio.

Crea el archivo «sample.txt» en ese directorio con un texto cualquiera, por ejemplo:

```
echo "After three days without programming, life becomes meaningless." > sample.txt
```

Podemos usar el comando «status» para confirmar el estado de nuestro «working tree» e «index» en Git.

```
$ git status
```

Obteniendo el siguiente resultado

```
On branch master

Initial commit

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

        sample.txt

nothing added to commit but untracked files present (use "git add" to track)
```

Como podemos deducir de respuesta de estado, «sample.txt» actualmente no se está bajo seguimiento. Tendremos que añadir «sample.txt» en el «index» primeramente antes de poder registrar y trabajar con sus cambios.

Para hacerlo simplemente utilizaremos el comando «add», seguido por el que queremos añadir al «index». Si queremos añadir varios archivos al «index», podemos hacerlo separándolos con espacios.

```
$ git add <file1> <file2>
```

Si queremos añadir todos los ficheros de un directorio al «index» simplemente utilizaremos «.»

```
$ git add .
```

Ahora, vamos a comprobar que «sample.txt» se ha añadido con éxito al «index».

```
$ git add sample.txt
$ git status
On branch master

Initial commit

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

        new file:   sample.txt
```

Ahora que «sample.txt» se ha añadido al «index», podemos hacer un commit del archivo. Utilizaremos el comando «commit» tal y como se muestra a continuación.

```
$ git commit -m ""
```

Despues de ejecutar el commit comprobaremos el estado.

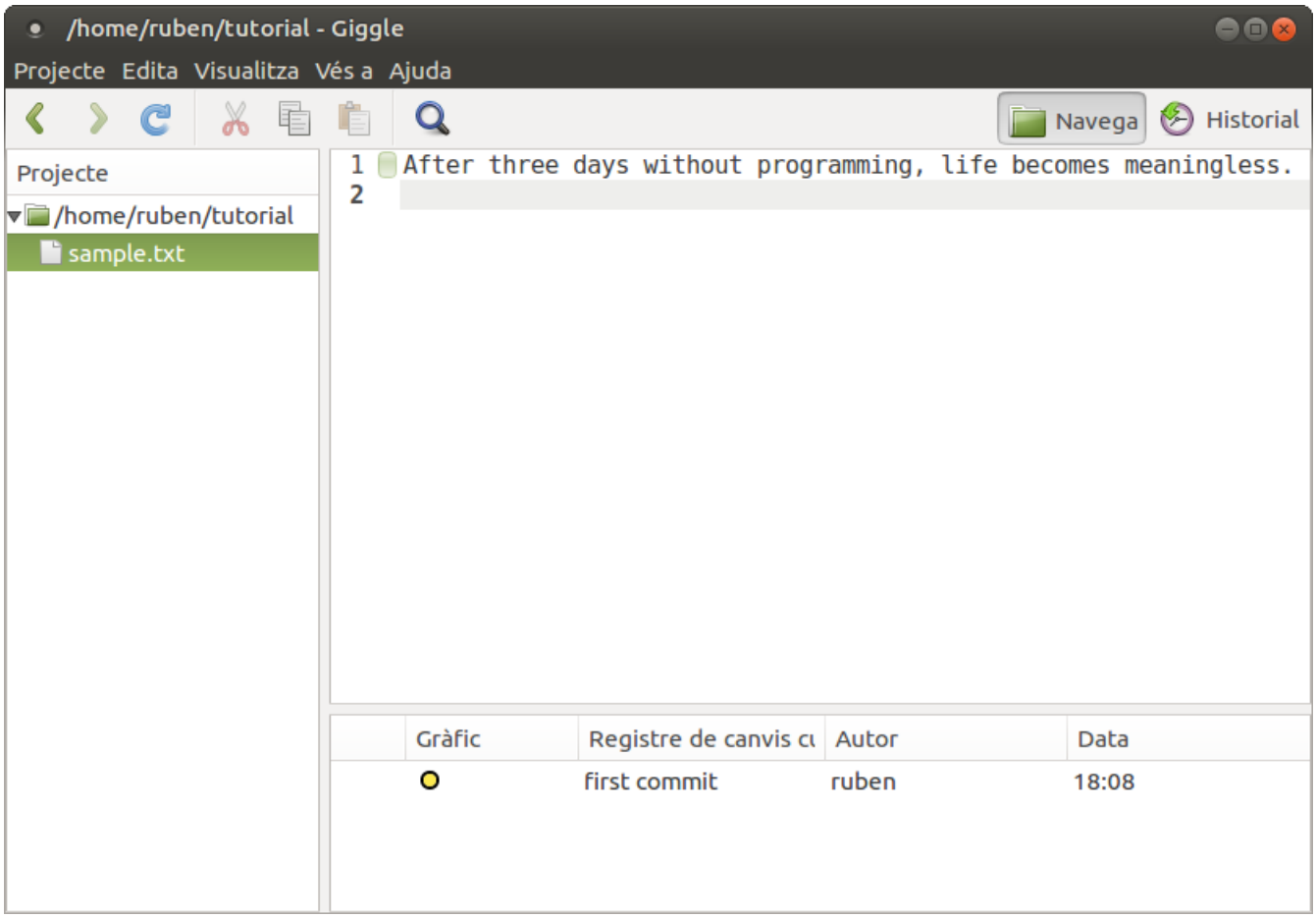
```
$ git commit -m "first commit"
[master (root-commit) 0742011] first commit
 1 file changed, 1 insertion(+)
 create mode 100644 sample.txt

$ git status
On branch master
nothing to commit, working directory clean
```

El resultado del comando «status» nos indica que no hay nuevos cambios a ser comiteados.

Podemos ver lo que se acaba de agregar mediante commit en el registro histórico del repositorio con el comando «log».

```
$ git log
commit 0742011aaf70e0e3a611fb22500c73d633f755c1
Author: someone <someone@gmail.com>
Date:   Fri Jan 30 18:08:42 2015 +0100

    first commit
```

Como referencia diremos que hay varios entorno gráficos que permiten realizar estas tareas. Por ejemplo tenemos Giggle




que podemos instalar fácilmente mediante los siguientes comando:

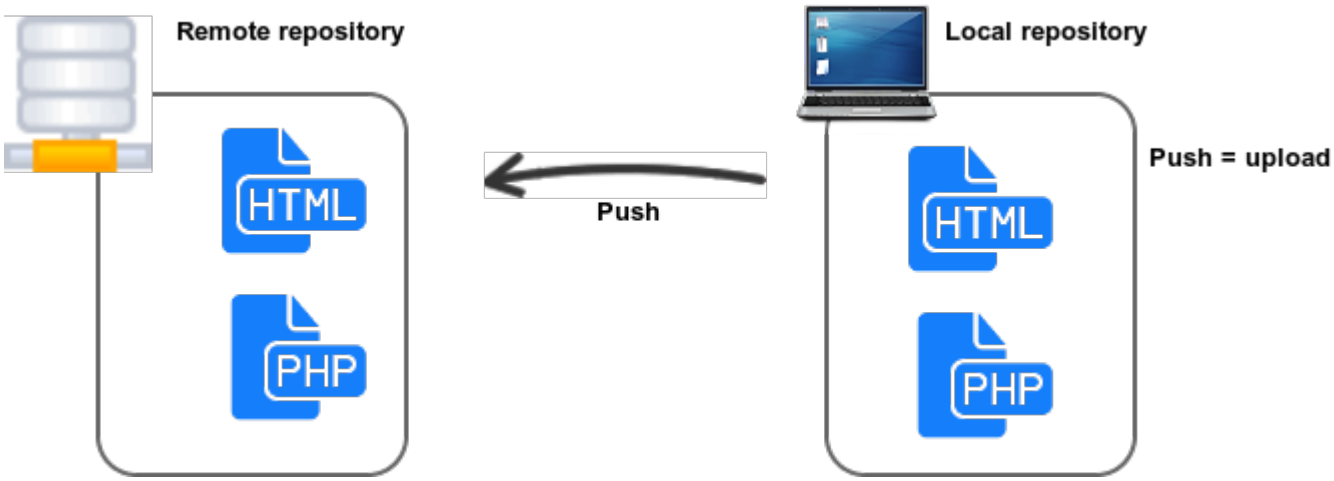
```
sudo apt-get install giggle giggle-terminal-view-plugin giggle-personal-details-plugin
```

## Trabajar con un repositorio remoto

*Hacer push a un repositorio remoto*

Hasta ahora, sólo hemos trabajado con repositorios locales. A partir de ahora trabajaremos con repositorios remotos que nos permitirán que compartir nuestros cambios con otros miembros del equipo.

Para empezar a compartir nuestros cambios, tenemos que subirlos a un repositorio remoto con el comando «push». Esto hará que el repositorio remoto se actualice y sincronice con nuestro repositorio local.




*Hacer clone de un repositorio remoto*

Si ya existe un repositorio remoto, se puede recuperar una copia y guardarla en tu máquina local para y empezar a trabajar.

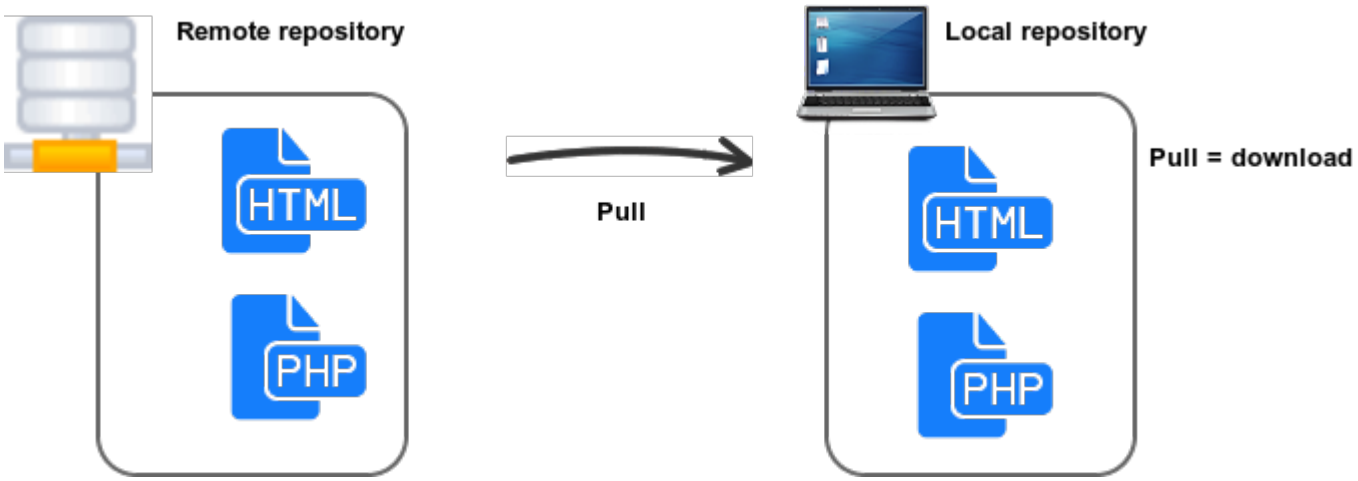
Se utiliza el comando «clon» para copiar un repositorio remoto. Por defecto, el comando «clon» sería configurar automáticamente una rama master en local a partir de la rama master remota des de la cual se ha realizado el clonado.

Un repositorio clonado tendrá el mismo registro histórico que el repositorio remoto. Se puede consultar y revertir cualquier de los commits en el repositorio local.

*Hacer pull de un repositorio remoto*

Cuando se realiza un «push» de los cambios realizados al repositorio remoto compartido, el repositorio local se queda desincronizado respecto a la última versión remota. Mediante Git, es bastante fácil de sincronizar el repositorio local con el repositorio remoto que ha sido actualizado.

Mediante una operación «pull» se pueden recuperar los cambios que se encuentran en el repositorio remotol. Cuando se ejecuta un «pull», la última revisión se descarga desde el repositorio remoto y se importa al repositorio local.




**Observaciones**

En este segundo post hemos visto como crear un repositorio trabajar con el, hacer un commit… Todo ello en local. Hemos introducido los conceptos necesarios para trabajar con un repositorio remoto, en el próximo post mostraremos paso a paso como hacerlo.

Ruben.

# Tutorial básico de GIT – Introducción

**Introducción**

Bien este va a ser el primero de un seguido de posts sobre GIT. A través de ellos aprenderemos el uso de Git e instalación.

Este extendido sistema de versiones nos permite trabajar conjuntamente en equipo sobre un mismo programa y guardar revisiones sobre nuestro código, siendo después fácilmente recuperables. Git fue creado por Linus Torvalds (el creador del Kernel de Linux)

En este primer post trataremos:

- ¿Qué es Git?
- Principios de Git
- Beneficios de Git
- Diferencias con otros sistemas de control de versiones
- Repositorios
- Working Tree i Index

## ¿Qué es Git?

Git es un sistema de control de versiones distribuido (Version Control System») o  herramienta de gestión de código («Code Management tool «). Creado por Lnuis Torval y actualmente utilizado por el equipo de desarrollo del Kernel de GNU\Linux.

Utilizando Git fácilmente puedes revisar el histórico de los códigos fuentes  de tu aplicativo y revertir cambios volviendo  una versión anterior o comprobar diferencias entre versiones de tus ficheros.

Si la última versión de un archivo se encuentra en un repositorio compartido, Git evitará una sobrescritura no intencionada por lo que mantienen una versión anterior del archivo.

## Principios de Git

El código fuente se guarda en un «working directory», donde tienes ficheros «tracked» ( ficheros que estan controlados por versiones) y ficheros «untracked» (exluidos del control de versiones, por ejemplo ficheros class generados a partir del código fuente que no son necesarios).

La línea de tiempo de desarrollo se compone de las revisiones («commits» en terminología de Git) del código fuente. Cada commit tiene conocimiento de sus predecesores. Mediante la comparación de un commit con su padre (es decir, la comparación de una revisión a la anterior), uno puede ver los cambios introducidos por el commit hijo.

Un commit contiene los nombres y contenidos de los archivos bajo control de versión, información sobre el autor y el committer (quien realizo el commit y que no tiene por qué ser el autor,), el momento en que se hizo el commit y un mensaje en el se deben indicar cuáles son las razones para los cambios

realizados.

Para hacer un commit, primero se debe indicar a Git qué cambios deben formar parte del nuevo commit mediante un comando add y luego de realizar el commit en sí, seguidamente abrirá un editor en el que podremos escribir un mensaje para indicar la razón por la cual se hicieron dichos cambios.

Beneficios de Git

Git no sólo permite realizar un histórico de tus cambios, permite realizar un seguimiento fácil del desarrollo de otros programadores e integrar esos cambios («merge» en terminología Git).

Aunque nos encontremos en un punto en que nuestro programa ha dejado de funcionar por completo mediante Git podemos saber fácilmente que cambios hemos hecho respecto al commit en el que trabajamos.

Otras ventajas:

- Estudiar el historico de un proyecto para entender no sólo lo que hicieron los desarrolladores, sino también por qué al leer sus mensajes en los commits (es decir, el registro de los cambios).
- Recuperar cualquier revisión anterior (es decir, «regression»), por ejemplo, cuando la versión más reciente contiene bugs que no están presentes en una versión anterior.
- Encontrar fácilmente el commit que introdujo los errores para realizar la regresión.
- Asegurarse de que el código y su histórico nunca se perderán, incluso si nuestro disco duro muere. Cualquier persona con un «clon» del repositorio tiene una copia de toda la historia.
- El trabajo en múltiples funcionalidades o corrección de

errores, organizándolas fácilmente y cambiando entre ellas utilizando branches.

El único aspecto negativo de Git es el tiempo de aprendizaje. Una vez se llega a dominar, Git es una gran herramienta para cualquier proyecto de desarrollo y su equipo.

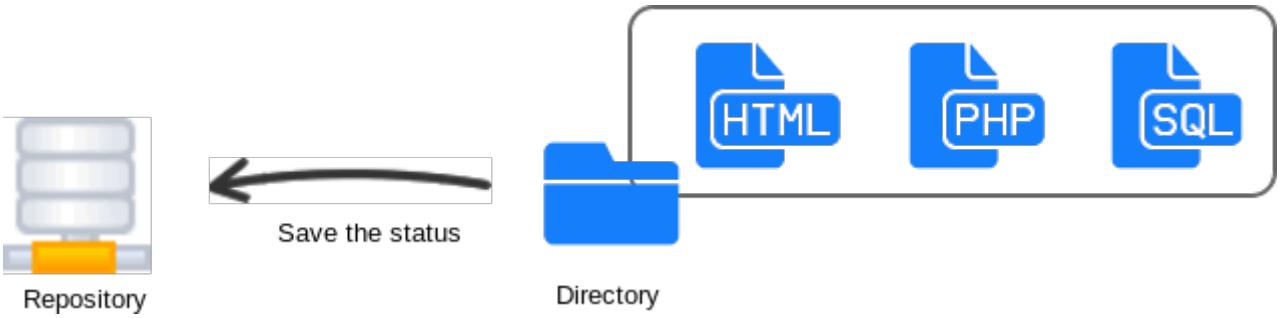
## Diferencias con otros sistemas de control de versiones

En comparación con otros sistemas, como CVS o Subversion, nos encontramos con que:

- En Git, cada repositorio es local. Para publicar cambios, es necesario tener un repositorio remoto, también, y hacer un «push» para guardar los cambios allí.
- En Git, ramas son de fáciles de usar y rápidas.
- En Git, imbécil agregar contenido, no los archivos. En otras palabras, cuando el fichero README que ya se realiza un seguimiento, git add README dirá Git que desea que los cambios a los ficheros de ser parte de la próxima confirmación.
- En Git, nunca, nunca intenta integrar los cambios remotos en un estado comprometido. En otras palabras, si usted tiene cambios no confirmados, siempre cometerlos antes de llamar git fetch origen; git merge origin / master.

## Repositorios

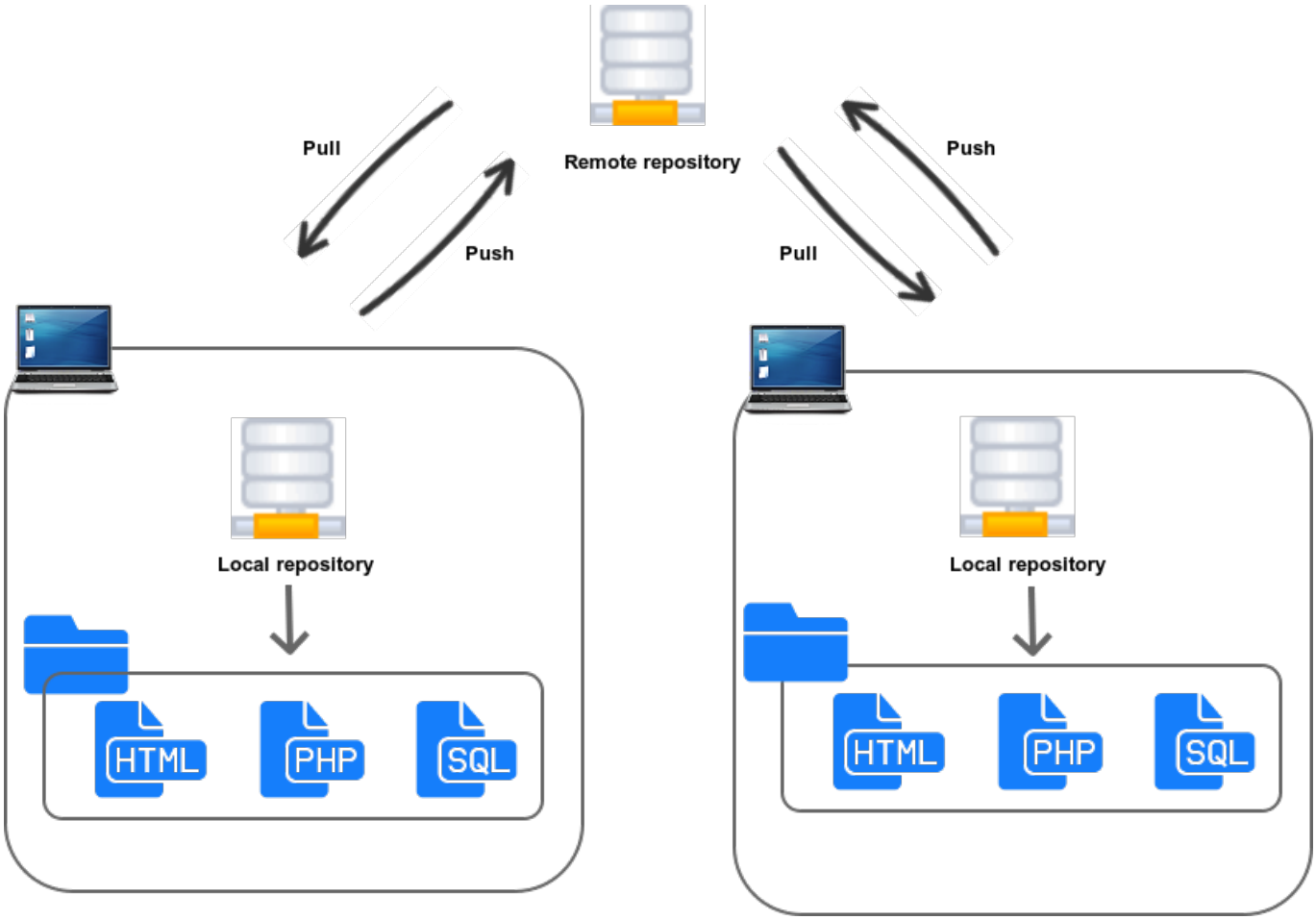
Cuando secrea un repositorio de Git con directorios y ficheros, se puedee guardar sus cambios e el historico y revisar sus estados y versiones.




Existen dos tipos de repositorios:

- Remote repository: Repositorio que reside en un servidor remoto, el cual se comparte con varios miembros del equipo.
- Local repository: Repositorio que reside en un equipo local para uso individual.

Se pueden usar todas las funcionalidades de control de versiones de Git en tu repositorio local (revertir cambios, seguimiento de los cambios, etc.). Sin embargo, cuando se trata de compartir cambios o obtener cambios de miembros de del equipo de desarrollo un repositorio remoto viene muy bien.

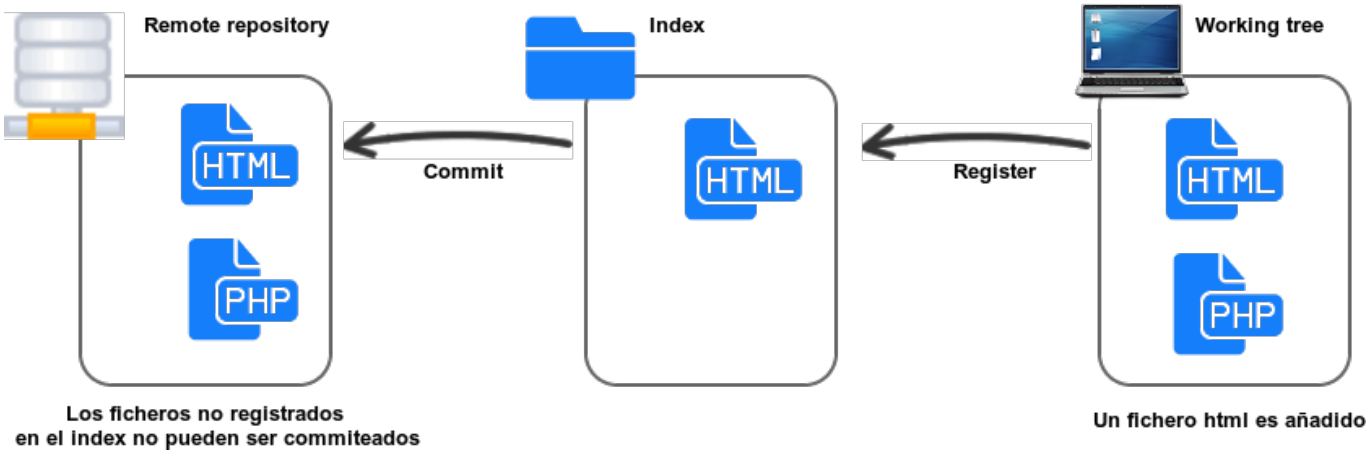



Hay dos formas de crear un repositorio local:

- Puede crear un nuevo repositorio desde cero
- Mediante la clonación obtener un repositorio remoto existente en en nuestro equipo local.

## Working Tree y Index

Un «working tree» es un conjunto de archivos con los cuales estas trabajando. Un «index» es un área de ensayo donde se preparan nuevos commits. Actúa como interfaz entre un repositorio y un «working tree».




Los cambios realizados en el «working tree» no serán commiteados directamente al repositorio. Necesitan ser movidos al «index» primero. Todos los cambios que esten en el «index» serán los que realmente se commitearan al repositorio.

Un «index» permite un mejor control sobre qué archivos deben ser incluidos y permite hacer un commit de una parte especifica de un archivo que se encuentra en el «index» al repositorio.

**Observaciones**

Con está introducción podemos tener un acercamiento a la importancia de los sistemas de versiones y a la versatilidad de Git en concreto. Comenzamos una serie de posts para profundizar en la utilización y explotación de este sistema para mostrar también como puede ayudarnos a trabajar colaborativamente en un proyecto.

Ruben.

# Ubuntu 12.04 Nvidia driver problema tras actualizar

Tras perder varias horas intentando solucionar un problema con la tarjeta 6150se NVIDIA encontré esta solución :

1) Eliminar los drivers instalados .

sudo apt-get purge nvidia* && sudo apt-get autoremove

2) Reiniciar

3) Si arranca el entorno gráfico probar las aplicaciones 3D

————CONTINUAR SI NO SE HA SOLUCIONADO EL PROBLEMA————

4) Si no funciona el entorno 3D probar la opción de controladores restringidos o adicionales e instalar el recomendado .

————CONTINUAR SI NO SE HA SOLUCIONADO EL PROBLEMA————

5) Si no funciona correctamente probar a instalar
(recordad que hemos hecho un purge para eliminar ficheros de configuración residuales)

sudo apt-get install nvidia-current

Ante todo agradecer a la fuente el librarme del problema.

Saludos

# Foremost – Recuperando archivos php borrados en servidor

**Introducción**

Bien en este post vamos a mostrar como recuperar archivos php borrados accidentalmente (situación a la que no se debe llegar, puesto que se tiene que contar con un sistema de backup). Para ello utilizaremos Foremost. Lo mas importante llegados a este punto es no usar dicho almacenamiento!!!! Los datos continúan estando en el disco a menos que grabemos un nuevo dato en el mismo sector, en ese caso no habrá posible recuperación. Tampoco podemos tomar esto como la solución a todos nuestros males, el uso de esta herramienta no es garantía de recuperar nuestros datos.

Foremost

Es un programa para hacer carving (rescate selectivo de ficheros).

¿Cómo funciona? Foremost trabaja con imágenes generadas con dd o particiones directamente, y se basa en el análisis de encabezados y footers de los archivos para 'extraer' lo que se pueda salvar. Esto se realiza mediante el encabezado hexadecimal de un fichero, por ejemplo:

- jpg
- gif
- png
- bmp
- avi
- exe
- mpg
- wav

- riff
- wmv
- mov
- pdf
- ole (PowerPoint, Word, Excel, Access y StarWriter)
- doc
- zip
- rar
- htm
- cpp

Instalando Foremost

sudo apt-get install foremost

Configurando Foremost para recuperar archivos php

Vamos a explicar poco la estructura del archivo de configuración. Este consta de líneas en las que se especifican búsquedas

Cada linea tiene la siguiente estructura:

- Extensión del archivo (php, cpp) que se quiera buscar
- Definir si se debe hacerse búsqueda case-sensitive «y» o no «n»
- Tamaño máximo del archivo.
- Encabezado: cadena de texto a buscar en los encabezados de los archivos; puede ser especificado en texto o hexadecimal.
- Footer: cadena de texto a buscar al final de los archivos; puede ser especificado en texto o hexadecimal.

Bien deberemos editar el archivo de configuración

sudo vi /etc/foremost.conf

y añadir la siguiente línea

php y 100000 \x3C\x3F\x70\x68\x70/

Bien para empezar con la recuperación ejecutaremos el siguiente comando:

foremost -t php -i /dev/sda1 -o /dev/sda2/recover/

Teniendo en cuenta que:

- -t especifica el tipo de archivo a buscar (si no se usa por defecto busca todos los que esten configurados en foremost.conf)
- Deberemos substituir /dev/sda1 por la ruta al disco del cual se quiera recuperar datos
- Deberemos modificar /dev/sda2/recover/ por la ruta donde queremos guardar los archivos recuperados (esta carpeta debe estar vacía)
- Los archivos recuperados no conservaran el nombre original. Nos encontraremos con archivos tipo 6856758.php
- Se debe tener mucha paciencia este proceso no durará segundos

**Observaciones**
Espero que esto pueda servir de ayuda. Normalmente esto ocurre cuando se no se tiene un sistema de backups. Se debe tener cuidado, hacer periódicamente backups incrementales puede evitar encontrarnos en la tediosa situación de utilizar este tipo de programas. Que si, deben estar,  se agradecen y mucho, pero no son garantía de recuperación y entramos en un terreno delicado nunca es bueno.

Ruben.

# XMPP – OpenFire

## Introducción


OpenFire es un servidor de XMPP programadfo en java (multiplataforma) y con un como WebApp para administrarlo. Se encuentra bajo licencia GPL y con el puedes administrar a tus usuarios, compartir archivos, auditar mensajes, mensajes offline, mensajes broadcast, grupos, etc y además contiene plugins para ampliar sus funcionalidades.

## Características

Openfire implementa las siguientes características:

- Panel de administración web
- Interfaz para agregar plugins
- SSL/TLS
- Conferencias
- Interacción con MSN, Google Talk, Yahoo messenger, AIM, ICQ
- Estadísticas del Servidor, mensajes, paquetes, etc.
- Cluster con múltiples servidores
- Transferencia de Archivos
- Compresión de datos
- Tarjetas personales con Avatar
- Mensajes offline
- Favoritos
- Autenticación vía Certificados, Kerbeos, LDAP, PAM y Radius
- Almacenamiento en Active Directory, LDAP, MS SQL, MySQL, Oracle y PostgreSQL
- SASL: ANONYMOUS, DIGEST-MD5 y Plain

## Instalación

Tan simple como descargar OpenFire Downloads (en nuestro manual un deb):

```
# wget http://www.igniterealtime.org/downloads/download-landing.jsp?file=openfire/openfire_3.6.4_all.deb
```

Luego podemos instalarlo fácilmente:

```
# sudo dpkg -i openfire_3.6.4_all.deb
```

**Configuración**

En primera instancia debemos seguir el intuitivo instalador des de la siguiente url:

http://localhost:9090

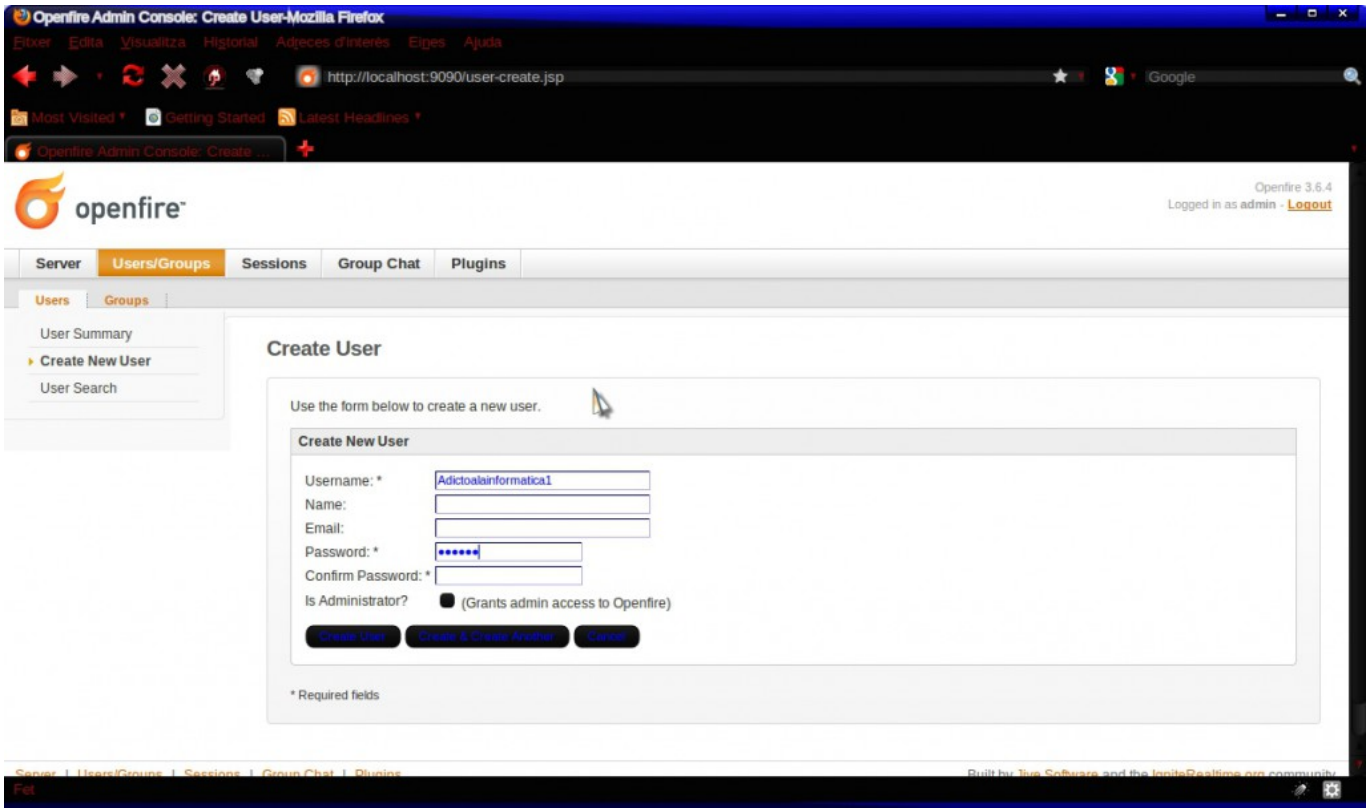
Por último deberemos realizar un restart del servidor para poder acceder al WebApp con el nuevo usuario administrador:

```
# /etc/init.d/openfire restart
```

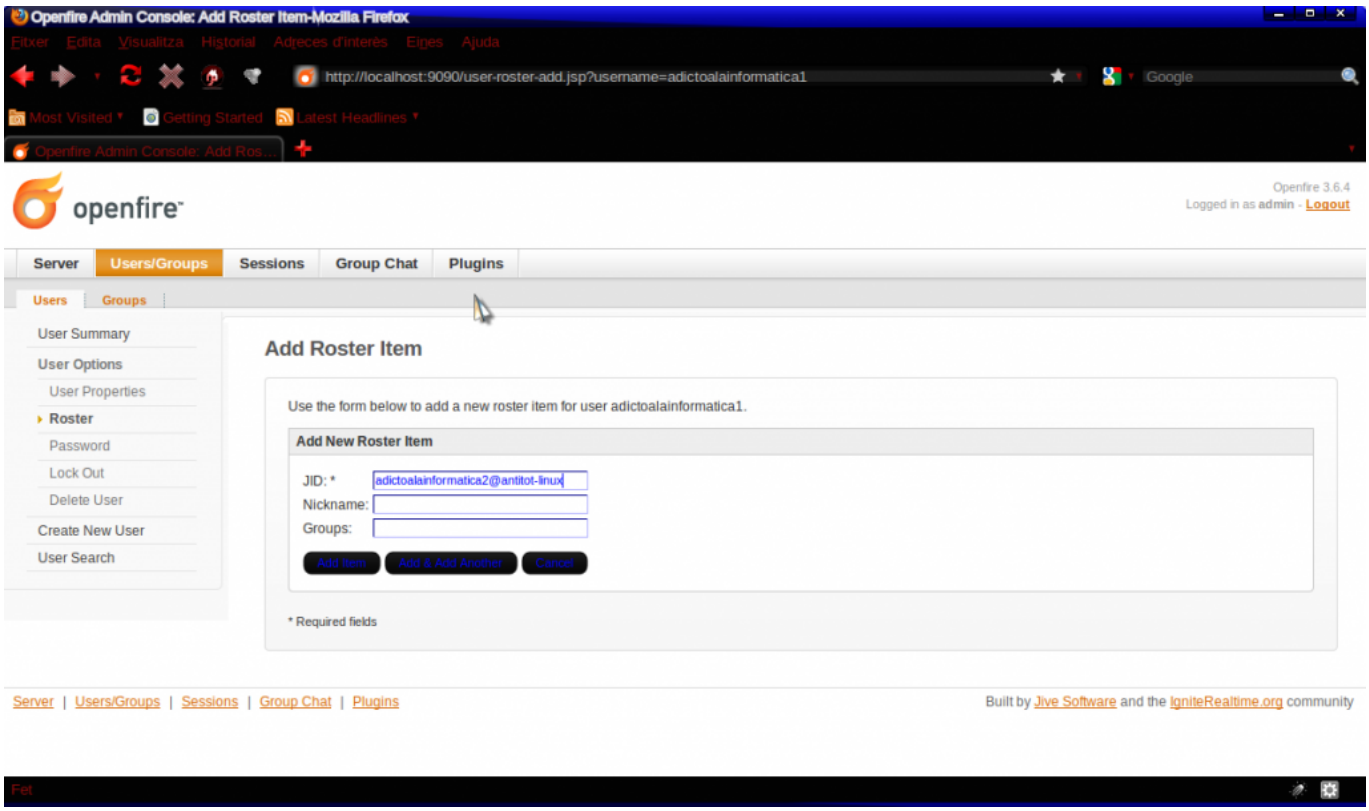
**Creación de usuarios**

Cuando creamos dos usuarios y queremos que estos puedan verse y enviar mensajes debemos añadirlos al roster de cada uno. Es decir, el usuario u1 debe tener en su roster al usuario u2 y u2 debe tener en su roster a u1. De esta manera podrán verse. Luego los dos usuarios deberán tener subscription both, es decir, deben estar subscritos a from y to para poder enviar y recibir mensajes de los usuarios que tienen en su roster.

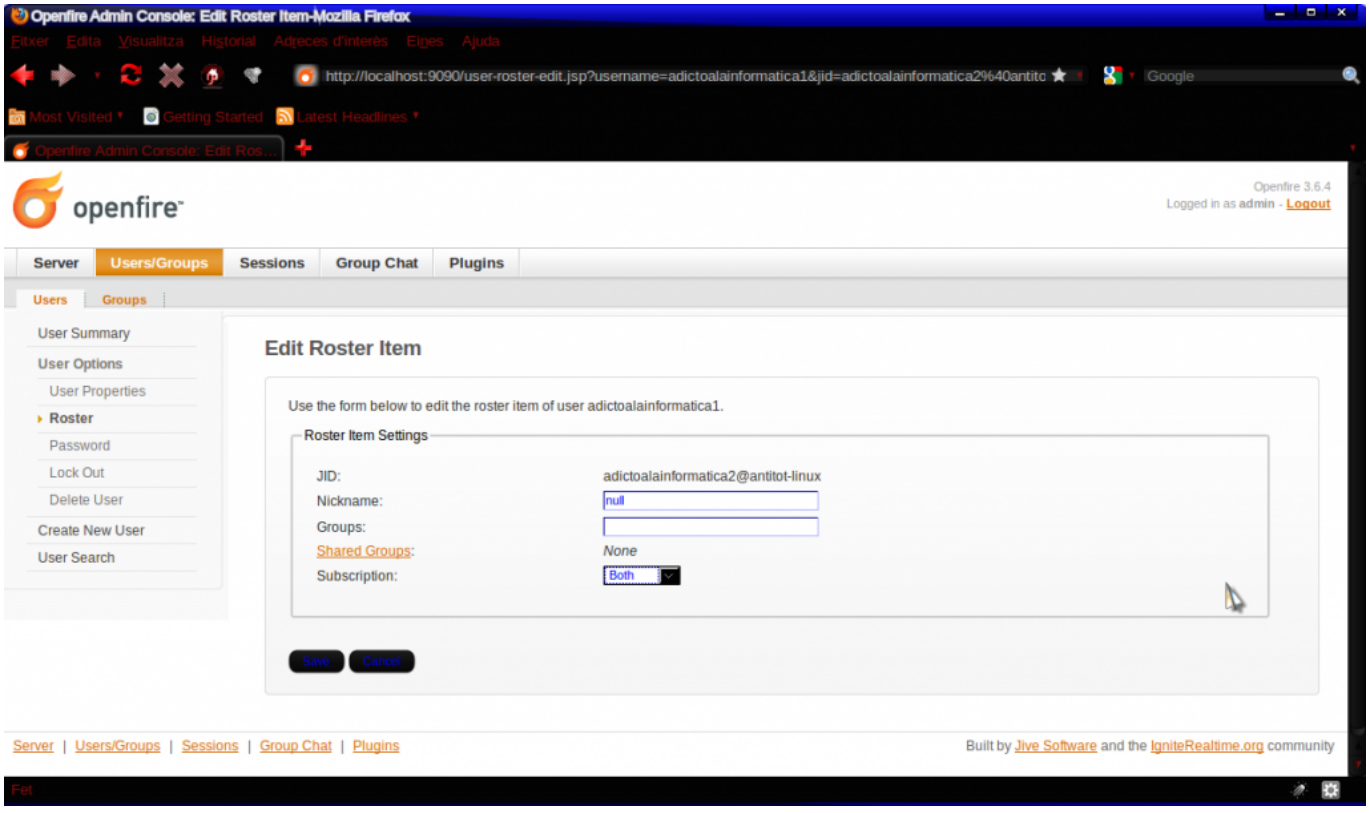
Para acceder a él tan solo debemos acceder mediante navegador web, después acceder a user/group y por último create user:




Crearemos dos usuarios adictoalainformatica1 y adictoalainformatica2 (originalidad antetodo). Despues deberemos entrar en la sección de cada usuario, en roster y por último clicar en Add Item:



Como podemos observar al usuario adictoalainformatica1 le añadimos al roster al usuario adictoalainformatica2 y debemos hacer lo respectivo para el usuario adictoalainformatica2 con adictoalainformatica1. Una vez echo esto deberemos cambiar el subscription para que los usuarios tengan completa «visibilidad» entre ellos. Debemos entrar en la sección del usuario adictoalainformatica1, luego en su roster donde veremos al usuario adictoalainformatica2. Bien, por último deberemos modificarlo y como vemos en la imagen debemos cambiar el subscription a both, en este caso los estamos haciendo para el usuario adictoalainformatica2 con respecto al usuario adictoalainformatica1 y deberemos hacerlo también a la inversa.




Ahora podríamos configurar dos clientes de xmpp (por ejemplo gajim y pidgin) con los dos usuarios y comprobar que pueden enviarse mensajes.

## Observaciones

Ya hemos viso un servidor de xmpp, no es el que ofrece más

opcionalidades pero para un desarrollo de nivel medio es más que suficiente. En grandes desarrollos donde tengamos una increíble cantidad de usuarios sería recomendable usar Ejabberd. Pero de todos modos con OpenFire tendremos un gran abanico de posibilidades y es mucho mas cómodo de administrar que Ejabberd. Un servidor XMPP puede ser útil tanto como para crear una red de mensajería a nivel interno de empresa como para montar un sistema de distribución de comandos. En el próximo post, ya cerrando con XMPP, mostraremos como crear un cliente con python capaz de conectarse, recibir y enviar estados y enviar y recibir mensajes.

**Fuentes**

- Wikipedia – Openfire

Ruben

---

# XMPP

**Introducción**

XMPP(Extensible Messaging and Presence Protocol) es un protocolo estándar y abierto que se basa en el intercambio de mensajes XML. Inicialment va ser concebut per implementar xarxes de missatgeria instantànea . Inicialmente fue concebido para implementar redes de mensajería instantánea. Quizás las herramientas más conocidas que usan el protocolo XMPP son Jabber, GTalk y las funciones de videoconferencia y audioconferencia de Google.

**Funcionalidades**

- Redundancia.
- Escalabilidad.

- No hacen falta VPNs para compartir recursos dentro de una NAT.
- Soporte **SSL** y Certificados
- Backends donde se guardan los usuarios: MySQL, LDAP…
- Extensible (usa lo que se llaman **XEP** – *XMPP Extension Protocols*)
- **BOSH** permite usar XMPP sobre **HTTP**, lo que por diseño del protocolo XMPP sería un problema.

**Protocolo**

Vamos a desglosar, identificar y explicar las principales opciones que nos ofrece XMPP como protocolo.

- **JID:** los nodos de una red XMPP identifican a través de este identificador, que es de la forma: user @ domain / resource (ejemplo: ruben@adictosalainformatica.com/linux ). Tratamiento de los JID:
  - user@example.com – conocido como JID
  - user@example.com/desktop – conocido como JID u **full JID**
- **Stanza:** los mensajes **XML** que se intercambian entre un servidor XMPP y un cliente se llaman Stanzas. Hay tres tipos de Stanzas:
  - **Messages:** transportan información entre nodos, los mensajes se pueden organizar en *threads.* Los hay de diferentes tipos:
    - normal
    - chat
    - groupchat
    - headline
    - error
  - **Presence:** sirven para informar de la dipsonibilitat de un recurso (online / offline):
    - away
    - do not disturb
    - extended away
    - free for chat

        - **IQ Stanza** (info query): similar a un HTTP GET / POST / PUT, sirve para pedir informaciones concretas a un nodo. Ideales para extender el protocolo. Por ejemplo, las IQ usan para saber qué recursos (usuarios) están conectados a un canal de chat. Los hay de tres tipos:
            - **get:** piden información (HTTP GET)
            - **set:** proveen información (HTTP POST / PUT)
            - **result:** devuelven información requerida o confirman que se ha aceptado un pedido 'set'.
    - **Extensibility:** para que sea simple extender el protocolo, las **Stanzas** soportan **namespaces** y cualquier elemento **XML** de una **stanza** se puede usar como un **payload,** para transportar: **XHTML** tags, Atom feeds, **XML-RPCs**, etc.
    - **Roster:** lista de personas que participan en un evento.
    - **Presence Subscription:** los recursos de una red (a menudo los usuarios) pueden subscribirse a otros recursos (otros usuarios) para saber si están o no disponibles en cada momento.
    - **Asincronismo:** la gracia del XMPP respecto a otros protocolos como HTTP es que se trata de un protocolo asíncrono, o sea, que las conexiones se establecen durante mucho tiempo y en cualquier momento el servidor y / o el cliente pueden enviar y recibir **Stanzas** a través de este canal. Los protocolos HTTP establecen conexiones relativamente cortas donde a menudo sólo hay una petición y una respuesta después se cierra la conexión.

**Jingle(add-on)**

**Jingle** es una extensión al protocolo XMPP que permite la transferencia de información p2p. Este protocolo permite transmitir datos multimedia, habilitando servicios de VideoConferencia y de VoIP. Este protocolo fue diseñado inicialmente por Google junto con la XMPP Standards Foundation

y liberado (bajo licencia similar a la de BSD).

**Observaciones**

Bien puede entenderse que este protocolo solo és útil para redes de mensajería instantànea, pero pensemos en sistemas de distribución de comandos. Resolvemos muchas incertidumbres facilmente, tenemos identificados los clientes, podemos enviar comandos a ciertos clientes y saber si estos clientes estan disponibles o no, por lo cual no habra un intento de envio esperando una respuesta de error simplemente no se envia, o si han recibido el comando correctamente, luego mediante respuesta del cliente podemos saber como ha ido la ejecución de cada comando por cada cliente.

**Fuentes**

- http://en.wikipedia.org/wiki/XMPP
- http://en.wikipedia.org/wiki/Jingle_(protocol)

Ruben

---

# Android sdk en Ubuntu unknown device ??????????

Tras instalar android sdk , eclipse , actualizar etc , nos podemos encontrar con que el emulador funciona,
pero no podemos debugar en el dispositivo.

Debugar en el dispositivo , nos ayuda a crear aplicaciones para los sensores , y nos permite probarlo al mismo tiempo , como los sensores de nivel , gps , etc , sin el dispositivo sería muy difícil .

Problema en DDMS , dispositivo desconocido ???????????
No funciona el debug con el dispositivo.

/android-sdk/platform-tools$ ./adb devices devuelve :

List of devices attached
??????????? ??
unknown device ???????????
[Fuente de la solución]
En esta zona podemos ver la solución:

> *14:25 W/ddms: Unable to get frame buffer: device (???????????) request*
> *rejected: insufficient permissions for device*
> ==
>
> *I then checked*
>
> ==
> *$ ./adb devices*
> *List of devices attached*
> *??????????? no permissions*
> ==
>
> *There appears to be a permission problem. I then wrote a*
> *`51-android.rules' file (with chmod a+r) in /etc/udev/rules.d/, such that*
>
> ==
> *# cat 51-android.rules*
> *SUBSYSTEM==»usb», ATTRS{idVendor}==»0bb4″, ATTRS{idProduct}==»0c87″,*
> *MODE=»0666″*
> ==
>
> *(the idVendor is the HTC one). I then restarted udev, but to no avail:*
> *same problems. Any idea?*

> 
> 
> 

_ 
*Merciadri Luca*
*See http://www.student.montefiore.ulg.ac.be/~merciadri/*
*I use PGP. If there is an incompatibility problem with your mail*
*client, please contact me.*

*Old 08-13-2010, 02:33 PM*
*Merciadri Luca*

*Default Android phone is not recognized by Android SDK, despite udev conf*
*Problem solved: restarting udev from CLI was not sufficient. I needed to*
*restart.*

Resumiendo:
1:)-> crear el fichero /etc/udev/rules.d/51-android.rules :
$ sudo gedit /etc/udev/rules.d/51-android.rules
con el siguiente contenido:
SUBSYSTEM=="usb", ATTRS{idVendor}=="0bb4", ATTRS{idProduct}=="0c87",
MODE="0666"

2:)-> darle permisos $ sudo chmod a+r /etc/udev/rules.d/51-android.rules
3:)-> reiniciar

Bueno , ya podemos utilizar el dispositivo , en la carpeta donde esté instalado adb ,
./adb devices , devuelve en mi caso

List of devices attached
HT0C4RX21596 device

ya podemos debugar , probamos creando un proyecto Android en eclipse , pulsamos el botón de debug y ya tenemos nuestro hello world …. que tanto deseábamos.

---

# NoSql – CouchDB

**Introducción**

CouchDB es una BBDD NoSql (si tienes dudas sobre NoSql es recomendable visitar el siguiente post Bases de Datos NoSql) de documentos en formato JSON. Dichos documentos se referencian por una clave única _id a los cuales se puede acceder por HTTP y gestionar mediante javascript. Esta programado Erlang, un lenguaje de programación muy robusto, fiable y multiplataforma, que se ejecuta sobre una máquina virtual de alto rendimiento.

**Claves y revisiones**

Ambos campos tienen que ser únicos, no pueden estar duplicados entre otros documentos en la misma base de datos.

- _id (clave): es el identificador único e inequivoco del documento (DocID)
- _rev(revisión): es un identificador del documento para tratar su historico. Es decir, de cada documento se guardan revisiones para poder acceder a anteriores versiones de dicho documento.

Revisiones

Cada vez que modificamos un documento en CouchDB se crea una nueva revisión de este. En principo esto nos permite acceder a anteriores versiones del documento, pero puede llegar a ser un

lastre. Por esa razón podemos compactar el documento, al hacerlo tansolo quedará disponible la última revisión.

En CouchDB podemos hacer consultas directas por identificador y por revisión, podemos obtener todas las revisiones activas de un documento (si existieran) y podríamos consultar información sobre las mismas.

A partir de la versión v 0.11 se puede especificar el número de revisiones:

```
curl -X PUT -d "200" http://localhost:5984/test/_revs_limit
```

Para compactar la base de datos y así eliminar las revisiones antiguas:

```
curl -X POST http://localhost:5984/nombre_base_de_datos/_compact
```

**Vistas**

CouchDB permite la creación de vistas, que son el mecanismo que permite la combinación de documentos para retornar valores de varios documentos, es decir, CouchDB permite la realización de las operaciones JOIN típicas de SQL.

**Replicación**

La replicación entrebases de datos nos permite duplicar datos con la seguridad de no perderlos. Hay que tener en cuenta que al guardar revisiones de los documentos siempre dispondremos de todas las revisiones. Es decir, pongamos que en tenemos una base de datos bbdd1 y otra bbdd2 y cambiamos un documento común doc1, después de la replicación tendremos las revisiones de los documentos anteriores y las revisiones de los nuevos documentos en las dos bases de datos. Es nos asegura consistente persistencia de datos.

Para habilitar la replicación activa entre dos bases de datos (cada vez que hay un cambio automaticamente las bases de datos se replican) tansolo debemos ejecutar el siguiente comando:

curl -X POST http://localhost:5984/_replicate -d '{"source":"db", "target":"db-replica", "continuous":true}'

Resolución de conflictos

En CouchDB la detección y resolución de conflictos esta automatizada. En el caso de que un documento se intente actualizar en distintos nodos se guardaran las distintas versiones y poniendo como la versión más reciente la ganadora. Para solventar conflictos, CouchDB permite gestionar las versiones, pudiendo borrar veriones erroneas o reordenarlas por fecha.

**Instalación**

Tan simple como ejecutar en la consola:

sudo apt-get install couchdb

Luego podemos comprobar en un navegador si CouchDB ha arrancado correctamentte accediendo a la siguiente

url http://localhost:5984/

, en caso de que todo funcione correctamente recibiremos una respuesta web similar a esta dependiendo de la versión de CouchDB:

{"couchdb":"Welcome","version":"0.10.0"}

**Comandos básicos mediante curl**

En primera instancia deberemos instalar curl:

sudo apt-get install curl

Para visualizar todas las bases de datos:

curl http://localhost:5984/_all_dbs

Crear una base de datos:

```
curl -X PUT http://localhost:5984/test
```

Crear un nuevo documento:

```
curl -X PUT http://localhost:5984/test/8765rftded6c29495e54cc05947f18c8af -d '{"SO":"Linux","Distro":"Debian"}'
```

Borrar la base de datos:

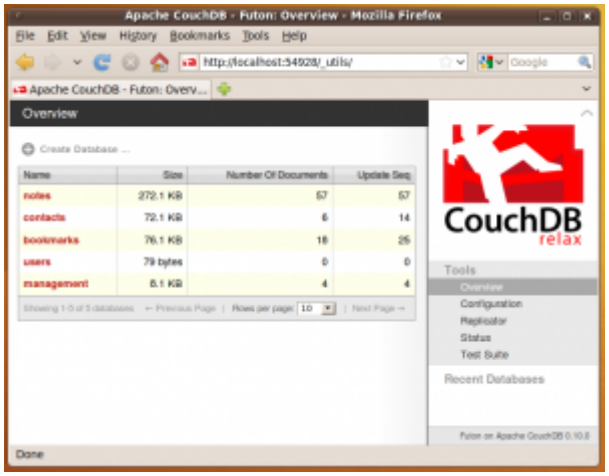
```
curl -X DELETE http://localhost:5984/test
```

## Futon

Futon es una aplicativo web muy intuitivo y fácil de usar, por lo que no explicaré como realizar las operaciones antes descritas con curl , para gestionar bases de datos y sus documentos en CouchDB.

Para acceder a él tansolo debemos acceder mediante navegador web  a la siguiente url:

```
http://localhost:5984/_utils/
```




## Observaciones

Llegados a este punto ya tenemos un concepto básico sobre las bases de datos NoSql y la utilización de un gestor, CouchDB. El próximo post sobre este tema estará en el apartado de programación, donde se expondrá un ejemplo de acceso a CouchDB mediante PHP.

**Fuentes**

- http://www.oriolrius.cat
- http://en.wikipedia.org/wiki/CouchDB

Ruben