Acceso a git

Ing. Alejandro Furfaro
Profesor Titular

April 15, 2017

Abstract

Git es una herramienta de desarrollo para control de versiones ampliamente utilizada en la industria del software inicialmente pero que se puede extender a la gestión de versiones de cualquier tipo de proyecto cuyo material de trabajo se mantiene en archivos. Una de sus novedades respecto de sus predecesores es su carácter distribuido. Se instala como paquete en numerosas distribuciones de Linux / UNIX y se han desarrollado para su mejor administración herramientas centralizadas, que a modo de front end permiten habilitar el acceso a los diferentes usuarios con diferentes perfiles, administrar proyectos y muchas otras funcionalidades que permiten ampliar el enorme potencial que de por si tiene *Git*. El objeto de este artículo es obrar como una guía para quienes nunca hayan utilizado esta herramienta describiendo las cuestiones mínimas necesarias para comenzar.

1 Introducción a los versionadores

Un sistema de control de versiones es una herramienta de desarrollo ampliamente utilizada que mantiene la historia de una colección de archivos que componen un proyecto. Esta colección de archivos generalmente es llamada "código fuente". Cualquier sistema de control de versiones debe contemplar la posibilidad efectiva de revertir la colección de archivos a una versión diferente. En términos generales una versión diferente puede referirse a una colección diferente de archivos, o al contenido diferente de los archivos de la colección actual.
Así podemos volver cuando lo necesitemos a la versión de varios días atrás, o cambiar entre los estados para características experimentales y problemas de producción.
Git es un sistema de control de versiones distribuido, y por ello todos tienen una copia completa del código fuente (incluyendo la historia completa del código fuente) y pueden realizar operaciones referidas al control de versiones mediante esa copia local. El uso de un sistema de control de versiones distribuido no requiere un repositorio central.
Git mantiene todas las versiones del código fuente de modo que permite revertir a cualquier punto en la historia del código fuente.
Los cambios que hacemos en el código fuente, los agregamos al repositorio (operación que se denomina ***commit***). *Git* realiza los ***commit*** al repositorio local del usuario, y es posible sincronizar ese repositorio con otros (ya sean estos remotos o locales).

Git permite clonar un repositorio, esto es, crear una copia exacta de un repositorio incluyendo la historia completa del código fuente (***clone***). Los dueños de los repositorios pueden sincronizar los cambios con ***push*** (transfiere los cambios al repositorio remoto) o ***pull*** (obtiene los cambios desde un repositorio remoto).

2 Instalación

No tiene grandes secretos. Para versiones de Linux basadas en administración de paquetes Debian like (como Ubuntu por ejemplo) el comando es:

```
sudo apt-get install git-core
```

Recordar que la clave que se debe ingresar a sudo es la del usuario con el que iniciamos la sesión.

3 Administración

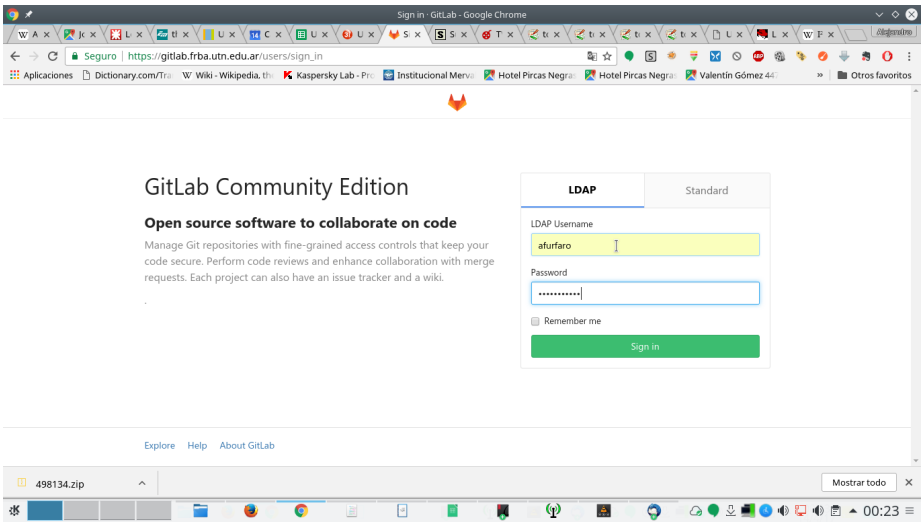
3.1 Front-Ends de administración

Si bien no se requiere un repositorio centralizado, con el uso creciente de *Git* se hizo necesaria una herramienta de administración que permitiese por ejemplo administrar el acceso de los diferentes usuarios a los diferentes proyectos, gestionar perfiles de usuarios, importar y exportar proyectos, etc.
Así es como comenzaron a aparecer algunas herramientas de administración. La primera fue GitHub. Pero al tiempo una empresa llamada Gitlab desarrolló una herramienta equivalente en funcionalidad pero que podemos instalar en un server de nuestra organización en forma gratuita.
Así es que como muchísimas instituciones, la FRBA instaló en un servidor una licencia libre de Gitlab.
Entonces, primer concepto:

> Una cosa es Gitlab, que está instalado en un server de la Facultad, y sirve para administrar usuarios perfiles y proyectos (entre muchas otras cosas mas), y otra es *Git*, que es el administrador de versiones que estará instalado en cada una de nuestras computadoras con el comando detallado en la sección 2

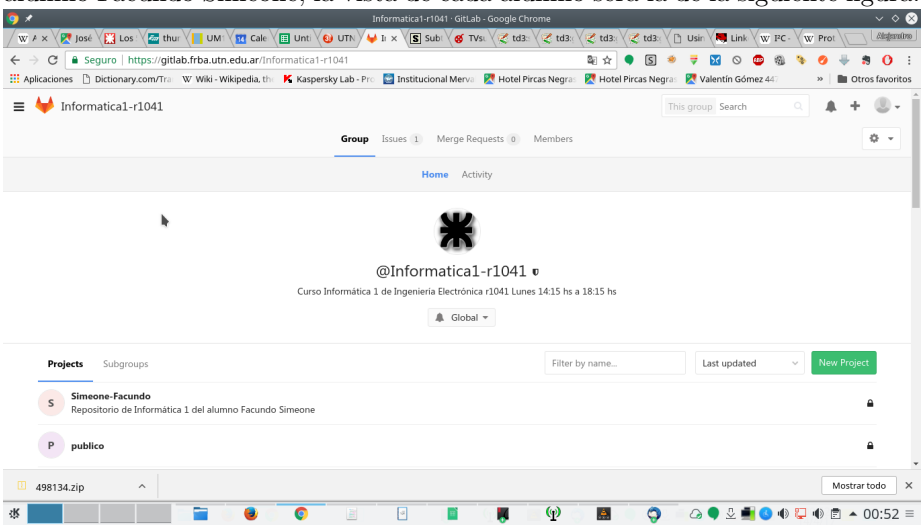
3.2 Ingresando a Gitlab FRBA

Para ingresar a Gitlab, se necesita solamente un navegador para acceder a la url en la cual nos solicita el usuario SINap y nuestra clave de acceso para iniciar sesión. La dirección que necesitamos escribir en la barra de dirección del navegador es gitlab.frba.utn.edu.ar.
La pantalla de ingreso es la siguiente:



El usuario se ingresa en la solapa LDAP para que se busque en la estructura de directorio de usuarios LDAP (una especie de Base de datos) que utiliza SINap para identificar a los usuarios y validar su acceso. En el ejemplo está mi usuario ***afurfaro***, y mi clave.

Una vez ingresado el usuario SINap y la clave de acceso, simplemente pulsar Enter o clickear en el botón "Sign in". El resultado será la vista de los proyectos en los que está incluido un alumno, en nuestro caso considerando el ejemplo del alumno Facundo Simeone, la vista de cada alumno será la de la siguiente figura:



3.3 Grupos, Proyectos, perfiles

Los docentes de cada curso han creado un grupo llamado genéricamente ***Informatica1-r10xx***, donde ***xx***, corresponde a cada curso. A este grupo pertenece todo el cuerpo docente del curso (profesor y ayudante(s). Todos los proyectos que se creen dentro de este grupo serán visibles para los docentes. Se creará un proyecto para cada alumno al cual se incluirá solo a ese alumno, y un proyecto publico

que tendrá incluidos a todos los alumnos de modo que allí los docentes puedan subir su material.
De este modo, cada alumno verá dos proyectos:

- ***publico***: Este proyecto contendrá slides de clase, material de lectura, ejemplos de código y demás recursos que los docentes de cada curso pondrán a disposición de los alumnos. Por lo tanto es accesible para todos los alumnos del curso. El perfil de cada alumno en este proyecto es Reporter ya que el alumno debe acceder a este proyecto con el único fin de ejecutar ***pull***. No necesitan acceso para ejecutar ***commit***: ni para ***push*** en este grupo de modo que el contenido de este proyecto, sea administrado exclusivamente por los docentes (ver sección 4 para detalles sobre estos comandos).

- ***apellido-nombre***: Este proyecto es privado de cada alumno. Por eso su nombre se compone del Apellido y Nombre del alumno, de modo de poder identificarlo claramente. El perfil de cada alumno en este proyecto es Master ya que se requiere que lo administre como corresponde.

4 Comandos mínimos indispensables

Si bien es posible descargar los archivos desde la interfaz web, el uso real de *Git* es desde cada nodo interconectado que tenga instalado el paquete git-core de acuerdo con lo indicado en el ítem 2.
El siguiente es el conjunto mínimo de operaciones a ejecutar en *Git* como para movernos inicialmente:

4.1 Antes que nada

Lo primero que debemos hacer es crear un directorio que será el raíz de nuestro repositorio local. Esto es. Un directorio por debajo del cual se desplegarán los diferentes proyectos que manejaremos con *Git*.
Lo que completa esta sub sección no es mas que una serie de recomendaciones. El lector organizará su disco de la manera que le resulte mas adecuado a su estilo usos y costumbres.
Lo único que recomendamos muy fuertemente es armar el repositorio local en algún punto que derive del `HOME directory` del usuario.
En mi caso tengo un directorio `work` dependiendo de mi `HOME` y dentro de este se despliega a esta altura de mi actividad una cantidad insana de directorios y dentro de estos otros directorios, y así ...
De modo que para crear mi repositorio local he tenido un rapto de imaginación que me llevó a nombrar el directorio raíz de mi repositorio como `git`.
El comando será (suponiendo que recién abro mi consola y por lo tanto el prompt me ubica en mi `HOME directory`):

```
alejandro@DarkSideOfTheMoon:~$
alejandro@DarkSideOfTheMoon:~/work/facu/InfoI$mkdir git
alejandro@DarkSideOfTheMoon:~/work/facu/InfoI$cd git
alejandro@DarkSideOfTheMoon:~/work/facu/InfoI/git$
```

4.2 Configuración Global

Git utiliza una firma d cada usuario cuando se establece un ***commit***. Por eso la primer vez que se va a trabajar con alguno de los proyectos del repositorio es necesario configurar nuestro usuario. En mi caso los dos comandos que tuve que ejecutar desde el directorio raíz de mi repositorio fueron los dos siguientes:

```
git config --global user.name "Alejandro Furfaro"
git config --global user.email "afurfaro@frba.utn.edu.ar"
```

El lector deberá reemplazar mi nombre por el suyo y mi dirección de correo electrónico por la suya asociada a SINap.

4.3 Creación de los repositorios

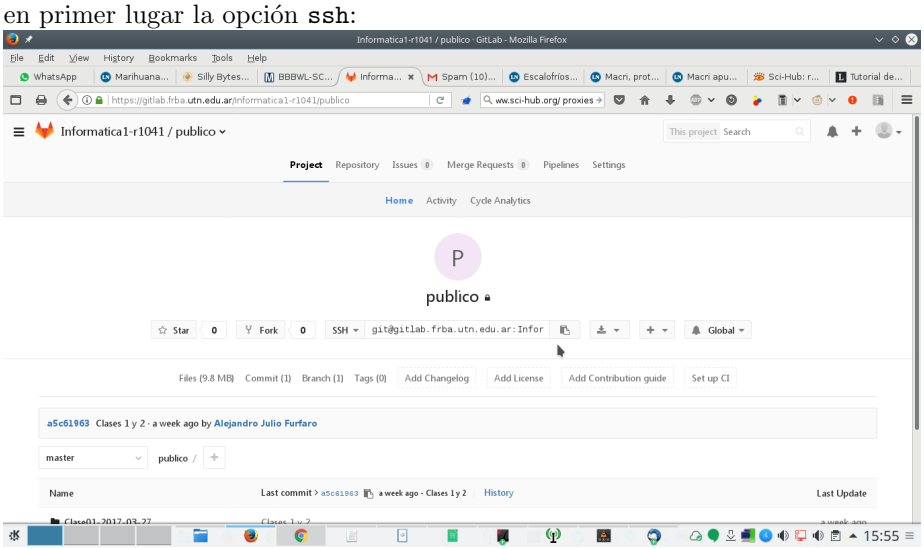
Sin necesidad de realizar ninguna operación previa, podemos clonar directamente el repositorio que se creó en el server. Para ello si el protocolo elegido es `ssh`, el comando es:

```
git clone git@gitlab.frba.utn.edu.ar:Informatica1-r1041/
    publico.git
```

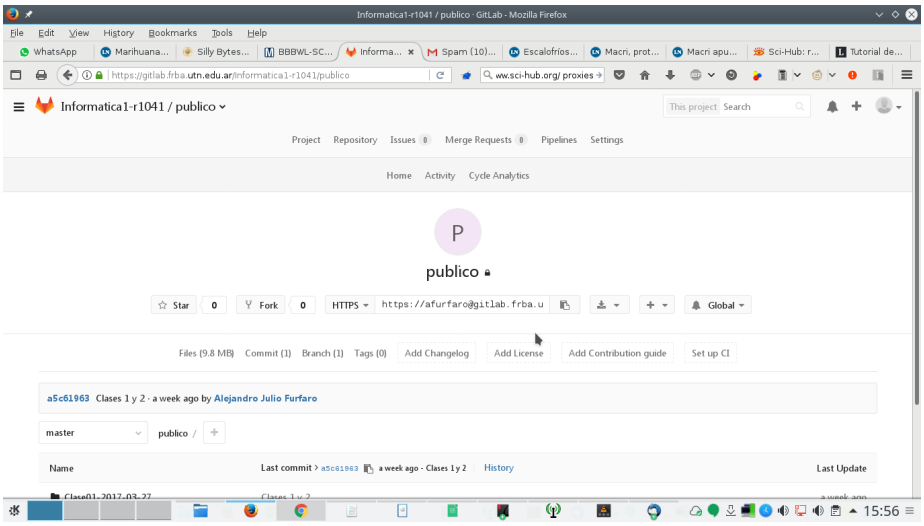
o bien si se accede por `https`:

```
git clone https://afurfaro@gitlab.frba.utn.edu.ar/
    Informatica1-r1041/publico.git
```

Estas opciones se puede ver claramente en Gitlab. Si nos logueamos como se indicó anteriormente y clickeamos uno de los proyectos que nos muestra, veremos en primer lugar la opción `ssh`:



o bien desplegando el botón que dice `ssh`, y seleccionando la otra opción (`https`):



Como se puede apreciar en el segundo caso dentro de la dirección del repositorio va inserto nuestro usuario (el lector deberá reemplazar mi usuario, **afurfaro**, por el suyo para que funcione).
Esto nos evitará ingresar el usuario cada vez que realicemos una actualización en el repositorio, pero si en cambio nos requerirá la contraseña de nuestro usuario SINap.
En cambio, si se accede por **ssh** hay dos opciones:

- Ingresar cada vez usuario y clave para cada operación, o
- Generar una clave pública, subirla a Gitlab, y acceder sin necesidad de identificarnos ya que la clave pública lo hará por nosotros

Lo mas práctico y por cierto con un margen de seguridad aceptable es generar las claves pública y privada para acceder a nuestros repositorios vía **ssh** sin necesidad de por cada operación ingresar usuario SINap y clave.
En consecuencia, hacemos un pequeño paréntesis en **git** para enfocarnos en generar nuestro par de claves pública y privada.
En primer lugar conviene verificar si ya no tenemos una clave generada (es poco probable si el lector recién instala su Linux, pero la verificación no está de mas).
Para verificar si ya existe una clave pública generada, el comando es el siguiente:

```
cat ~/.ssh/id_rsa.pub
```

En caso que esté generada el comando **cat** (**man cat** por si el lector no sabe lo que hace y no lo recuerda de lo visto en clase) presentará una cadena de texto que indefectiblemente comienza con *ssh-rsa* continúa (al menos en caso de mi clave) con 373 caracteres y finaliza con el correo electrónico.
Si la salida del comando **cat** no es la indicada, no hay clave pública generada.
En tal caso para generarla ejecutamos:

```
ssh-keygen -t rsa -C "su.email@est.frba.utn.edu.ar" -b 4096
```

Cabe aclarar que el mail va como comentario al final de la clave (la opción -C significa Comment), y -b 4096 especifica el tamaño en bits de la clave (2048 bits

ya se considera suficiente, de modo que 4096 es mas que suficiente por ahora para asegurar que no pueda ser desencriptada por ningún algoritmo de fuerza bruta.
Una vez generada la clave vamos a copiarla al portapapeles para pegarla en la pantalla de Gitlab, como vamos a ver.

```
xclip -sel clip < ~/.ssh/id_rsa.pub
```

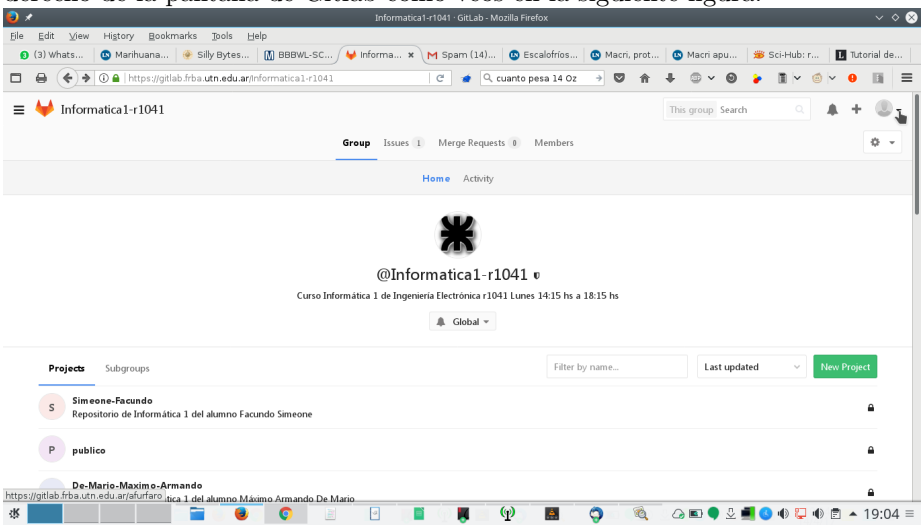
Si el comando arroja la siguiente salida

```
bash: xclip: no se encontr  la orden
```

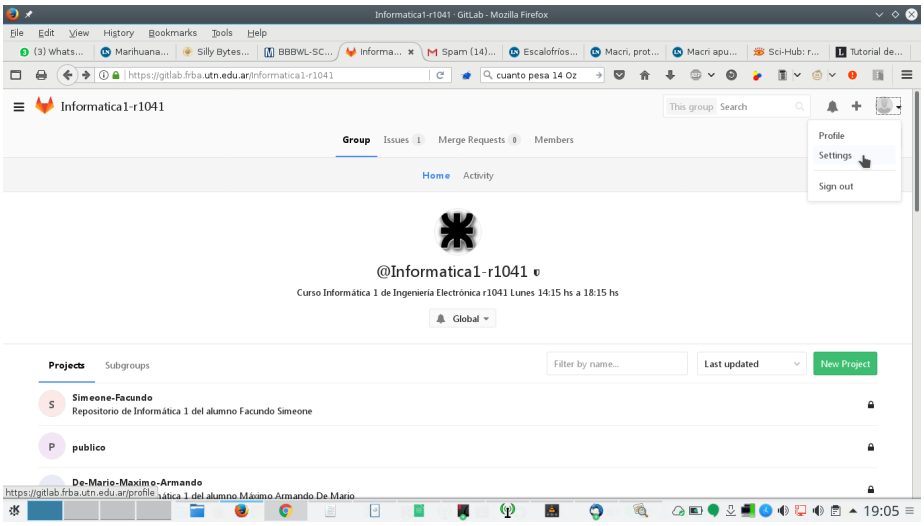
significa que el paquete xclip no está instalado, en cuyo caso repetir el comando anterior luego de ejecutar lo siguiente:

```
sudo apt-get install xclip
```

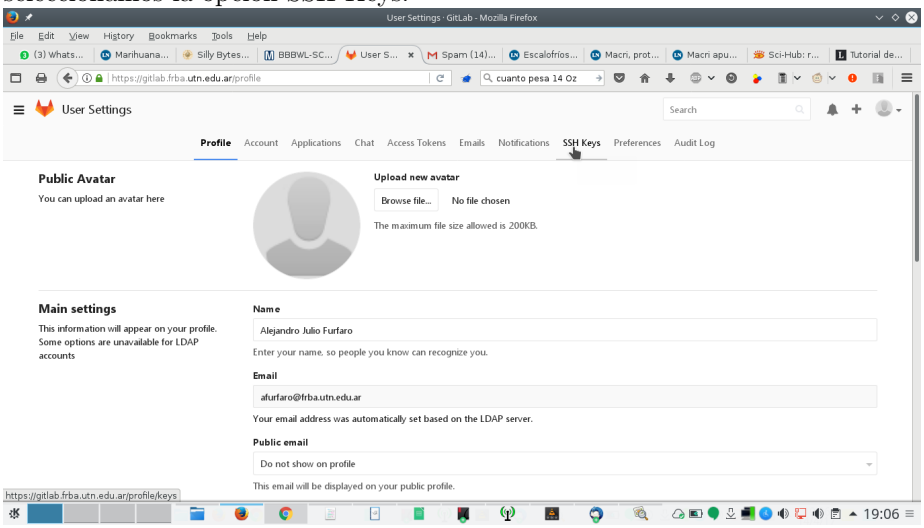
Recordar que la clave que se debe ingresar a sudo es la del usuario con el que iniciamos la sesión.
Una vez copiada la clave al portapapeles, hay que ingresarla a Gitlab. Para ello hay que ir ala configuración de nuestro usuario. Se accede en el borde superior derecho de la pantalla de Gitlab como veos en la siguiente figura:



Al clickear selecionamos settings,



Una vez cargada la pantalla de configuración en la barra superior de la página seleccionamos la opción SSH Keys.



Seleccionando esta poción llegamos a la pantalla en la que disponemos de un text box para pegar allí el texto de la clave. En el text box siguiente podríamos ponerle un título si lo deseamos (no es mandatorio), y finalmente con un click en el botón “Add Key”, agregamos la clave.
Es indistinto generar la clave y luego clonar el repositorio o realizarlo tal como lo describimos anteriormente. La única diferencia es que si primero generamos la clave, la operación de clonado por `ssh` ya no nos va a pedir usuario y clave. La tomará directamente desde las clave publica y privada.

4.4 operación del repositorio

Cada repositorio contiene un archivo ***.git*** en su directorio principal que contiene todas las características del repositorio y su enlace con la configuración local y del usuario. Podemos incluir un archivo ***.gitignore*** para configurarle patrones

en los nombres de los archivos que serán ignorados por la operación `add`, `status`, `commit` y `push` Una vez clonado el repositorio cada vez que nos conectemos y antes de enviar modificaciones es conveniente actualizar los cambios que puedan haberse producido en los demás repositorios locales, y recién entonces generar nuestras actualizaciones.

```
git status
```

Este comando chequea el estado de actualización del repositorio, e informa si hay cambios para actualizar en nuestro repositorio local y/o si hay cambios en nuestro repositorio que es necesario enviar.

```
git pull
```

Este comando actualiza el repositorio local con los cambios producidos en los repositorios remotos siempre que éstos hayan sido publicados con el comando `git push`.
Una vez que hemos trabajado en nuevos archivos en nuestro proyecto, para poderlos publicar es necesario agregarlos al repositorio. EL comando es:

```
git add archivo1 archivo2 ....
```

Agregamos los archivos de a uno o en forma general podríamos utilizar wildcards (caracteres comodín como el '*', por ejemplo para simplificar el comando. Por ejemplo:

```
git add *
```

agregará al repositorio todos los archivos nuevos que se hayan generado y aun no se hayan agregado. Si existe un archivo `.gitignore` en el directorio raíz del repositorio local, `git` no va a tomar en cuenta aquellos archivos cuyo nombre figure en `.gitignore`, o cuyo nombre sea coincida con un wildcard definido en `.gitignore`.
Por ejemplo si queremos que en nuestro proyecto solo suban los fuentes, y no se agreguen los objetos, y además convenimos en llamar siempre al ejecutable "demo", agregando estas líneas en el archivo `..gitignore` los mismos nunca serán agregados añ repositorio con el comando `add *`

```
*.o
demo
```

Además si los archivos se han generado en subdirectorios que tampoco hasta el momento se agregan al repositorio, `Git` se agregaran con la ruta al subdirectorio incluida. Esta es claramente una ventaja respecto de svn donde había que agregar al subdirectorio primero y luego los archivos que se crearon en este subdirectorio.
Para actualizar el repositorio con los cambios realizados, en primer lugar se actualiza el repositorio local, mediante:

```
git commit -m "mensaje que tenga sentido para la
    actualizaci n"
```

Este comando deja actualizado el repositorio local. Para enviar los cambios del resto, el cando es:

```
git push
```

5 Conclusiones

`Git` es un sistema muy potente y altamente utilizado en la industria. Hemos repasado lo básico para empezar a utilizarlo. Como herramienta de desarrollo `Git` es muchísimo mas amplia. Sin embargo a nuestros efectos en esta etapa es suficiente.