1.3

Gestión de versiones y control de revisiones del software

Es evidente que un programa no se escribe directamente "bien" desde el principio. El proyecto tiene una "historia" en la que se van añadiendo funcionalidades y corrigiendo errores.

Los sistemas de gestión de versiones y revisiones nos ayudan en los siguientes aspectos:

- asignar un número de versión global a nuestro programa, que irá aumentando según pase el tiempo y vaya mejorándo. El número de versión ayuda a identificar las características y capacidades del programa en un instante dado.
- a nivel interno, mantener documentación sobre los cambios que cada fichero que compone el proyecto experimenta, asignando un número interno de versión al fichero y guardando todas todas las versiones del fichero.
- en relación con el punto anterior, gestionar las copias de seguridad del proyecto.
- facilitar el trabajo en equipo en el proyecto, de forma que distintos programadores puedan simultánemente escribir código.

La idea fundamental en que se basan un sistema de control de versiones es en un "repositorio" que funciona como almacén del proyecto, y una o más copias de trabajo del proyecto. Desde una copia de trabajo, se guarda en el repositorio nuevas versiones de los ficheros, cuando el programador lo decide. Aquí hay una lista de sistemas de control de versiones.

1.3.1 git

git, como dice en su página, es un sistema de control de versiones distribuido, de código abierto, para manejar cualquier tipo de proyecto con rapidez y eficiencia.

Vamos a explicar cuáles son los pasos para utilizarlo.

Si no está ya en nuestro sistema, podemos descargar el paquete de instalación desde su página git. Una vez instalado, hay que anotar dónde están los ejecutables (`git`, `git-shell`, `git-receive-pack`, `...`). Podrían estar, por ejemplo, en `/usr/bin` o en `/usr/local/git/bin`.

1.3.1.1 Uso básico de git, localmente

Si no trabajamos en equipo, la forma más sencilla de utilizar git es localmente. Junto con el directorio que contiene nuestro proyecto, se crea siempre un directorio `.git` con una copia de nuestros ficheros gestionada por `git` (no necesitamos entrar dentro de `.git`). Cuando hacemos cambios y queremos "guardarlos" (`commit`) esos cambios se transladan a la copia en `.git`.

Supongamos que en el directorio `miProyecto` tenemos un proyecto que queremos gestionar con `git`. que por el momento contiene `fichero1.txt`

Para empezar la gestión del proyecto hacemos:


```
git init
```

que crea e inicializa el directorio `.git`.

A continuación, hemos de decir qué ficheros queremos que sean gestionados podemos hacer (añadir todos):

Añadir ficheros para gestionar

```
git add -A
```

Para ver cuál es el estado de los ficheros en nuestro directorio de trabajo, a saber, si se han guardado ("commit"), si tienen cambios no guardados, o si no están siendo gestionados ("staged–unstaged" o "tracked–untracked"), haremos

Estado de los ficheros

```
git status
```

que nos muestra por ejemplo:

```
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
```


```
#
#       new file:   fichero1.txt
#
```

avisándonos que hay un nuevo fichero gestionado pero no confirmado ("commit").

Otro ejemplo de `git status` podría ser:

```
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   fichero1.txt
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       fichero2.txt
```

que nos informa que el fichero gestionado `fichero1.txt` tiene cambios no confirmados ("commit", no guardados en `.git`) y que hay un fichero sin gestionar `fichero2.txt`

commit

Guardar instantánea

Ahora vamos a hacer un “commit” para guardar los cambios y hacer una “foto fija” de cómo están las cosas justo en ese momento. El commit recibirá un número para identificarlo y siempre podremos volver a éste punto. El commit se hace así:


```
git commit -a
```

que nos llevará a un editor de texto (por defecto 'vi') para escribir un mensaje:

```
Primer comit del proyecto
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# Committer: Jordi Bataller i Mascarell <Jordi@brahms.gnd.upv.es>
#
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
# new file:   fichero1.txt
```

salimos guardando (ZZ en 'vi') y el commit queda realizado.

Podemos comprobarlo haciendo:

```
git status
```

y veremos:


```
# On branch master
nothing to commit (working directory clean)
```


A partir de ahora podremos ver los “commits” (instantáneas que hayamos hecho) con el comando


```
git log
```

que nos mostrará algo como

```
commit 1365b46e9931ac4078b92e12b72e5bf02be9af59
Author: Jordi Bataller i Mascarell <Jordi@example.com>
Date:   Wed May 22 13:45:13 2013 +0200

    Primer comit del proyecto
```

Mientras trabajamos modificaremos ficheros, los renombraremos, o borraremos algunos. Por ejemplo `git status` podría mostrarnos algo como

```
# On branch master
# Changed but not updated:
#   (use "git add/rm <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   fichero1.txt
#       deleted:    fichero2.txt
```


```
#       deleted:    fichero3.txt
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       fichero4.txt
#       fichero5.txt
```

Que nos informa que de los ficheros gestionados `fichero1, 2 y 3.txt` hay uno modificado y dos borrados. Además hay 2 nuevos no gestionado, el 4 y 5.

Deshacer cosas

Si queremos **borrar** cualquier fichero o directorio **no gestionado** (en el ejemplo, los ficheros 4 y 5) haremos:

```
git clean -fdx
```

Si queremos **deshacer** todos cambios en los ficheros **gestionados** y dejarlo todo como estaba justo después del último "commit" haremos

```
git reset --hard
```

Incorporar cambios

Sin embargo si aceptamos los nuevos ficheros, para añadirlos a las gestión haremos:

```
git add -A
```

Ahora `git status` nos informa que


```
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   fichero1.txt
#       deleted:    fichero2.txt
#       renamed:    fichero3.txt -> fichero4.txt
#       new file:   fichero5.txt
#
```

no hay fichero sin gestionar, pero los cambios pendientes de "commit" son la modificación del fichero 1, que se ha borrado el 2, que el 3 se ha cambiado de nombre a 4, y que hay uno nuevo: el 5.

Si estamos de acuerdo hacemos "commit"

```
git commit -a
```

Volver atrás

Cuando necesitemos consultar el estado de los ficheros en un "commit" pasado podemos volver a ese estado haciendo `checkout` usando el número de "commit" (se averigua con `git log`)

```
git checkout 0cb3edc496bedca09e578946b79edeb6fe62a7ac
```

Volver al final

Para volver al último "commit" (a la versión más reciente) hacemos

```
git checkout master
```

Etiquetas. tag

Resulta útil etiquetar los commits para movernos entre ellos en caso de necesidad.

El commit en el que estamos actualmente se etiqueta con:


```
git tag versionInteresantePorAlgo
```

Podemos ver qué "tags" tenemos con

```
git tag
```

Movernos a un tag

y cambiarnos a su estado fácilmente haciendo

```
git checkout versionInteresantePorAlgo
```

Tags en log

Si queremos que `log` nos informe también de los tags hay que escribir

```
git log --decorate
```

Ramas ("branch" y "merge")

Trabajando en un proyecto será muy probable que necesitamos tener dos líneas de desarrollo.

Por ejemplo, tenemos un programa desarrollado y operativo que vamos a distribuir, con la versión `v1.0`. Si en su utilización se descubren "bugs" tendremos que sacar nuevas versiones menores con las correcciones: `v-1.1, v-1.2`. Pero, además sobre la versión`v1.0`, queremos añadir funcionalidad que no sabemos si finalmente acabaremos incorporando al programa.

Ante esta situación necesitamos hacer una "bifurcación". En la rama principal "master", partiendo de `v1.0` realizaremos únicamente las posibles correcciones que se puedan dar. Por otro lado, en una rama "test", que también parte de `v1.0` comenzaremos a añadir nuevas funciones.

Crear ramas (branch)

A partir del "commit" en el que estemos (incluso no él último) podemos crear una rama haciendo


```
git branch test
```

y movernos a esa rama

```
git checkout test
```

o volver a la principal

```
git checkout master
```

Con el siguiente comando veremos qué ramas tiene nuestro proyecto

```
git branch -v
```

Si trabajamos con ramas, resultará muy útil mostrar el árbol de versiones, con los "commit" realizados, las ramas, las etiquetas usadas y averiguar en qué en qué punto estamos trabajando. Este comando nos mostrará:

```
git log --graph --oneline --all --decorate
```

algo como esto

```
* bb412e6 (HEAD, master) septimo commit
* fb74b4a sexto commit
* f1fd07a Quinto commit
* 8a2f24d cuarto commit
* 4fc9390 tercer commit
| * 08f8d20 (test) cambios en el segundo commit, para ver si queda el 3 y 4
```


```
|/
* 0cb3edc (tag: versionAntiguaInteresante) segunda edicion, la guardo
* 1365b46 Primer comit del proyecto
```

donde sabremos que:

- hay dos ramas: `master` y `ramaSeparada`
- las ramas parten del "commit" cuyo número empieza por 0cb3edc y que está etiquetado como `versionAntiguaInteresante`
- ahora mismo estamos trabajando (ver `HEAD`) a partir del último "commit" de la rama principal, el bb412e6.

merge

merge

Si el desarrollo realizado en la rama "test" es satisfactorio y decidimos incorporarlo a la versión principal "master" de forma que se unan las posibles correcciones de "bugs" que hayamos realizado en la rama principal con las nuevas funciones realizaas en la rama de test, deberemos hacer un `merge`, estando en la rama principal:

```
git checkout master
git merge test
```

Conflictos

Si los ficheros sólo tienen modificaciones en una de las ramas, no habrá conflictos y el `merge` será automático y limpio. De no ser así, el comando nos dirá qué ficheros han sido modificados en ambas ramas para que manualmente solucionemos el problema:


```
Auto-merging fichero1.txt
CONFLICT (content): Merge conflict in fichero1.txt
Automatic merge failed; fix conflicts and then commit the result.
```

Con `git status` podemos consultar los ficheros con problemas:

```
# On branch master
# Unmerged paths:
#   (use "git add/rm <file>..." as appropriate to mark resolution)
#
#       both modified:      fichero1.txt
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   fichero5.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
```

Para resolver los conflictos editaremos manualmente cada fichero problemático y en el encontraremos líneas


```
<<<<<<< HEAD
 una cosa
 (version en la rama master)
=======
 otra cosa diferente
 (version en la rama test)
>>>>>>> test
```

que nos marcan puntos no compatibles. Decidiremos qué dejar, y luego eliminaremos las 3 lineas de separación (`<HEAD, ====, > test`)

Una vez resueltos los conflicos debemos hacer

```
git add -A
```

(o individualmente `git add fichero1.txt`) para indicar a `git` que ya están resueltos los conflictos. Y finalmente haremos:

```
git commit
```

Finalmente, veremos la historia de commits en forma de árbol:

```
git log --graph --oneline --all --decorate
```

```
*   22771ea (HEAD, master) Merge branch 'test'
|\
| * 08f8d20 (test) cambios en el segundo commit, para ver si queda el 3 y 4
```


```
* | bb412e6 septimo commit
* | fb74b4a sexto commit
* | f1fd07a Quinto commit
* | 8a2f24d cuarto commit
* | 4fc9390 tercer commit
|/
* 0cb3edc (tag: versionAntiguaInteresante) segunda edicion, la guardo
* 1365b46 Primer comit del proyecto
```

1.3.1.2 Preparación de un repositorio remoto en un servidor (accesible mediante ssh)

Si estamos trabajando en equipo, necesitaremos disponer de un repositorio en un ordenador que guarde centralizadamente el proyecto, mientras cada programador tiene en su equipo una copia de trabajo del mismo.

Así pues, desde nuestro ordenador de trabajo sacaremos del repositorio (“pull” o “clone”) el código para trabajar, y cuando lo consideremos oportuno, volcaremos (“push”) al repositorio las actualizaciones realizadas.

En el servidor

Para crear el repositorio centralizado, en el ordenador donde queremos que esté, deberemos realizar los siguientes pasos (ver servidor git):

1. Instalar un servidor ssh que permita conectarse al servidor desde otros ordenadores. En los sistemas tipo Unix suele estar ya instalado o es muy fácil de instalar y configurar.
2. Crear un usuario llamado `git`[1] en el servidor.
   1. Hay que conseguir que el directorio donde están los ejecutables de `git` estén en la variable de entorno `PATH`. Una forma de hacerlo es añadir al fichero `~/.bashrc` (o equivalente) una línea como:

      ```
      export PATH=$PATH:/usr/local/git/bin
      ```

   2. La contraseña del usuario `git` va a ser conocida por todos los participantes en el proyecto o, en cualquier caso, van a poder conectarse al ordenador.
      Por este motivo, conviene cambiar el shell del usuario a otro restringido que se proporciona con la distribución. Este shell restringido sólo permitirá realizar acciones relacionadas con git. El nombre del shell es `git-shell` y hay que saber dónde está, (`/usr/local/git/bin/git-shell` por ejemplo) para hacer la modificación en `/etc/passwd` o donde corresponda[2].
3. Dar permiso de acceso en el servidor a los programadores.

---

1 No es estrictamente necesario que se llame así.

2 Por ejemplo en Mac OSX: Preferencias del sistema - cuentas - boton derecho sobre la cuenta git - opciones avanzadas - shell.

Para poder descargar o actualizar el proyecto en que se está trabajando, los programadores debe poder acceder al servidor como usuario `git`. La manera más rápida (y menos segura) es darles la contraseña del usuario (al menos habremos restringido el shell, para que no puedan realizar un login normal).

Una mejor forma de permitir el acceso es mediante una clave pública:

1. En su ordenador, el programador ejecuta

   ```
   ssh-keygen
   ```

   que generará en el directorio `~/.ssh` dos ficheros, la pareja de claves pública y privada: `id_rsa.pub` y `id_rsa`.
2. El programador ha de hacer llegar al administrador del servidor el fichero `id_rsa.pub`.
3. En el servidor, el administrador ha de añadir, literalmente copiada la clave pública al fichero `~/.ssh/authorized_keys`. Por ejemplo:

   ```
   cat /tmp/id_rsa.pub >> ~/.ssh/authorized_keys
   ```
4. Una vez hecho esto, un programdor podrá realizar acciones con git utilizando `ssh` sin necesidad de conocer la contraseña del usuario `git`.

4. Ahora hay que elegir un directorio donde estará el repositorio para proyectos `git`. Elegimos por ejemplo `/opt/git`. Lo creamos y damos la propiedad al usuario `git`

   ```
   cd /opt
   mkdir git
   chown git git
   ```

A continuación, creamos nuestro primer proyecto de prueba `prueba.git`:


```
cd /opt/git

mkdir prueba.git
```

y en él activamos los ficheros de control

```
cd /opt/git/prueba.git

git --bare init

                initialized empyt Git repository in /opt/git/prueba.git
```

En realidad, cualquier usuario del ordenador servidor puede configurar un directorio como repositorio `git` y utilizar su usuario personal con los comandos `git` remotos (ver siguiente sección). Además, este usuario puede dar permisos de sólo lectura (`chmod a+rx-w directorio`) a cualquier proyecto a otros usuarios del ordenador o al usuario `git` para que mediante git pueda obtener copias de lectura del proyecto.

En el cliente

Ahora, un programador tendrá que crear y dejar en el servidor `git` la estructura inicial del proyecto desde su ordenador de trabajo.

1. Supongamos que en el directorio `prueba`, un programador ha generado un proyecto con maven o con cualquier otra herramienta. En dicho directorio (limpio, sin ficheros re-generables ni transitorios), tendrá que ejecutar los siguientes comandos, para crear los ficheros de control de `git` y indicarle que queremos que se guarden. La orden `commit` especifica que queremos guardar los cambios y en ella se ha de dar un comentario:

```
cd prueba

git init
```


```
git add *
git commit -m 'creación del proyecto'
```

A continuación indicamos a `git` que queremos guardar remotamente este proyecto, indicando ese lugar:

```
git remote add origin git@servidor.gnd.upv.es:/opt/git/prueba.git
```

Fijémonos en que damos el usuario, `git`, el servidor, `servidor.gnd.upv.es`, y el directorio que habíamos preparado allí para el proyecto. Si nos equivocamos en este paso podemos hacer

```
git remote rm origin
```

para borrar la asociación `origin` con usuario-servidor-directorio.
Finalmente, con la siguiente orden, enviamos ("push") el proyecto al servidor asociado a `origin` como rama principal ("master") del proyecto.

```
git push origin master
```

2. Una vez hecho lo anterior, cualquier programador del proyecto puede:
   - Descargar por por primera vez el proyecto:

     ```
     git clone git@servidor.gnd.upv.es:/opt/git/prueba.git
     ```

     Nota: si un usuario nos ha dado permiso de lectura (por ejemplo al usuario `git`) de uno de sus proyectos (`/opt/git.jordi/otroProyecto.git`), podemos descargarlo haciendo

     ```
     git clone git@servidor.gnd.upv.es:/opt/git.jordi/otroProyecto.git
     ```

     siendo en ese ejemplo `jordi` el propietario del directorio con el proyecto.

Después del trabajo que hay realizado el programador, y tras hacer “commit” podrá enviar los cambios al repositorio con:


```
git push origin master
```

– Cualquier programador puede actualizar su copia de trabajo del proyecto desde el servidor con:

```
git pull
```