



# Sistemas de control de versiones distribuidos

Controla las versiones de tu trabajo con GIT

Nacho Álvarez

🐦 @neonigmacdb

✉ neonigma@gmail.com

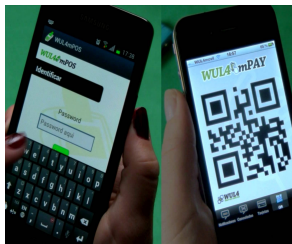


11 de diciembre de 2013

- 1 Acerca de mí
- 2 Definiciones
- 3 ¿Por qué GIT?
- 4 Arquitectura SCV
- 5 Flujo de trabajo en GIT
- 6 Demo
- 7 Gestión de conflictos
- 8 Tags
- 9 Más funcionalidades útiles
- 10 Algo más avanzado
- 11 Problemas comunes
- 12 Enlaces de interés

# Who?

- ▶ **Trayectoria profesional:** soporte UCO, desarrollador Web, desarrollador / integrador distribuciones GNU/Linux.
- ▶ **Actualmente:** WUL4 Córdoba (mobile + backend developer)
- ▶ **Involucrado en:**



# Definiciones

- ▶ **Control de versiones:** gestión de los diversos cambios que se realizan sobre los elementos de algún producto
- ▶ 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
- ▶ Los **sistemas** de control de versiones (SCV) vienen a automatizar parcialmente la gestión de este control de cambios
- ▶ Existen SCV **centralizados** (repositorio único remoto) y SCV **distribuidos** (cada usuario su repositorio local + remoto)
- ▶ Los más famosos y de más trayectoria son: SVN, GIT, Mercurial, Bazaar, ClearCase, Perforce...


# Ventajas SCV centralizados

- ▶ En los sistemas distribuidos hay un **mayor bloqueo** del estado final del proyecto que en los sistemas centralizados.
- ▶ En los sistemas centralizados las versiones vienen identificadas por un **número de versión**. En lugar de eso cada versión tiene un identificador (hash) al que se le puede asociar una etiqueta (tag).
- ▶ La **curva de aprendizaje** es sensiblemente menor que en los sistemas distribuidos
- ▶ Requiere **menor intervención** del mantenedor

# Ventajas SCV distribuidos

- ▶ Necesita **menos operaciones en red** => mayor autonomía y una mayor rapidez.
- ▶ Aunque se **caiga** el **repositorio remoto** la gente puede seguir trabajando
- ▶ Alta probabilidad de **reconstrucción** en caso de falla debido a su arquitectura distribuida
- ▶ Permite mantener **repositorios centrales más limpios**, el mantenedor decide
- ▶ El **servidor remoto** requiere **menos recursos** que los que necesitaría un servidor centralizado ya que gran parte del trabajo lo realizan los **repositorios locales**.
- ▶ Al ser los sistemas distribuidos **más recientes** que los sistemas centralizados, y al tener **más flexibilidad** por tener un repositorio local y otro/s remotos, estos sistemas han sido diseñados para hacer fácil el uso de **ramas locales y remotas** (creación, evolución y fusión) y poder aprovechar al máximo su potencial.

# ¿Por qué GIT?

A man with sunglasses and a white t-shirt stands on a sandy beach with turquoise water in the background. A large black speech bubble is overlaid on the right side of the image, containing text.

**A MI ME SOBRA CON SVN  
Y HAGO TODO  
LO QUE NECESITO**

# ¿Por qué GIT?



PostgreSQL





# En números



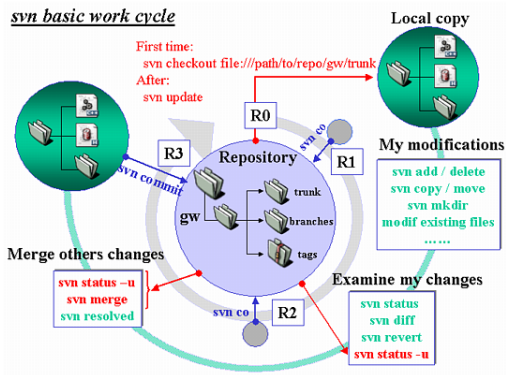
- ▶ Enfocado a **código privado** (Mailchimp, Opera...)
- ▶ Más de **1 millón** de usuarios registrados
- ▶ Integración del resto del ecosistema software: Bamboo (CI), Confluence (Doc), Jira (project tracking), SourceTree (GUI)...
- ▶ Se cobra por **número de integrantes de equipo**: 0-5 (gratis), 6-10 (\$10 month), 11-25 (\$25 month), 26-50 (\$50 month)...



- ▶ Enfocado a **código abierto** (bootstrap, nodejs, jquery...)
- ▶ Más de **4 millones** de usuarios registrados y **8 millones** de **repositorios** creados
- ▶ Se cobra por **repositorios privados**: 5 (\$7 month), 10 (\$12 month), 20 (\$22 month), 50 (\$50 month)

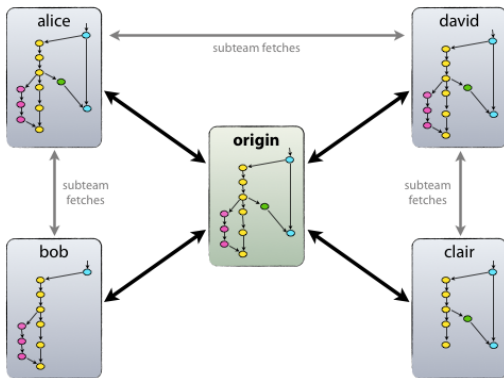
# Arquitectura SCV centralizado

- Todo el mundo actualiza un mismo repositorio central remoto

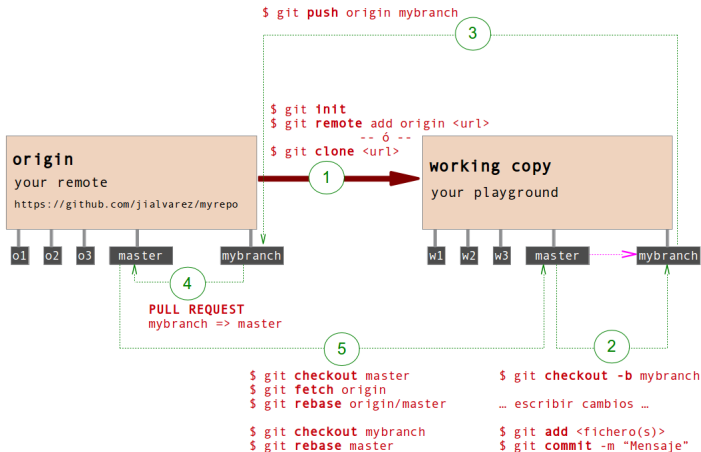


# Arquitectura SCV distribuido

- ▶ Todo el mundo mantiene su copia del proyecto
- ▶ Cada integrante del equipo trabaja en sus propias funcionalidades en su repositorio local particular
- ▶ El mantenedor del repositorio acepta o no las modificaciones de los integrantes



# Flujo de trabajo en GIT



# Punteros: HEAD, origin...

# Etapas de un fichero (I)

# **Archivos sin seguimiento:**

# (use git add <archivo>... para incluir lo que se ha de ejecutar)

#

# fichero.tex

- ▶ Son archivos que aún no forman parte del repositorio local ni del remoto
- ▶ Al añadirlos pasan a la etapa III

## Etapas de un fichero (II)

```
# Cambios no preparados para el commit:  
# (use git add <archivo>... para actualizar lo que se ejecutará)  
# (use git checkout -- <archivo>... para descartar cambios en el  
# directorio de trabajo)  
#  
# modificado: advanced.tex  
# modificado: principal.pdf
```

- ▶ Son ficheros ya añadidos previamente pero que aún no han sido *marcados* para comitear
- ▶ Al añadirlos pasan a la etapa III
- ▶ Si le aplicamos un *checkout* vuelven a la etapa I

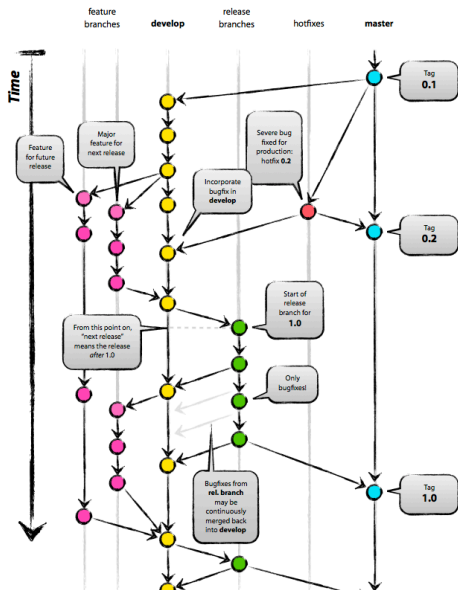
## Etapas de un fichero (III)

```
# Cambios para hacer commit:  
# (use git reset HEAD <archivo>...para eliminar stage)  
#  
# modificado: workflow.tex
```

- ▶ Son ficheros ya añadidos previamente y marcados para comitear
- ▶ Si le aplicamos un *reset HEAD* vuelven a la etapa II
- ▶ En esta etapa es donde se puede **comitear** y hacer **push**

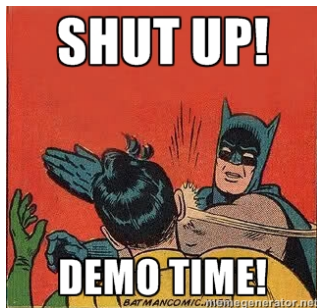


# Git branching



- Reglas propuestas por Vincent Driessen
- Ramas **master** (commits producción) y **develop** (siguiente versión planificada)
- Ramas **feature** (funcionalidad concreta). Se originan a partir de **develop** y vuelven a **develop**
- Ramas **release** (siguiente versión en producción). Se originan de **develop** y pasan a **master** o **develop**.
- Ramas **hotfix** (bugs en producción). Se originan a partir de **master** y vuelven a **master** o **develop**.

# Demo



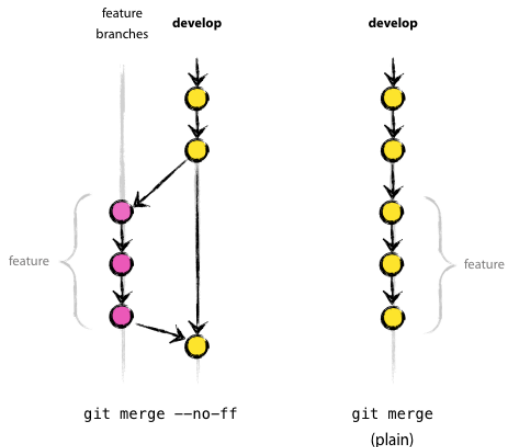
- ▶ Crearemos nuestro propio repositorio Git en Bitbucket
- ▶ Crearemos nuestra rama de trabajo y subiremos cambios
- ▶ Propondremos la integración de estos cambios haciendo *pull request*
- ▶ Simularemos la descarga del repositorio por parte de un compañero y subiremos nuevos cambios
- ▶ Actualizaremos el repo haciendo *rebase* y subiremos más cambios
- ▶ Mostraremos cómo funciona el *rebase* de manera gráfica

# Merges en caso de conflicto

```
$ git checkout master
```

```
Switched to branch 'master'
```

```
$ git merge --no-ff mybranch -m 'Merged pull request #7'
```



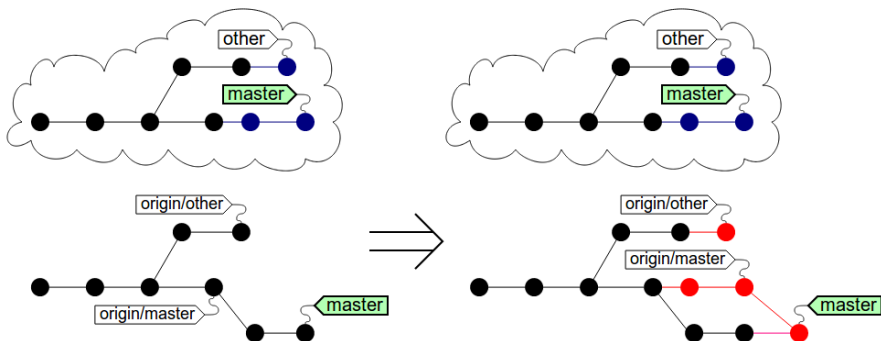
# Tags

- ▶ En SVN, comúnmente se congela una versión copiando y pegando el *trunk* a la carpeta *tags* y subiendo esta nueva carpeta
- ▶ En Git, un tag no es más que una etiqueta en un commit concreto
- ▶ Con este simple acto, Git puede recuperar una versión en cualquier momento utilizando la etiqueta del tag
- ▶ Dos tipos:
  - **Lightweight tags:** `git tag <etiqueta>`
  - **Annotated tags:** `git tag -a <etiqueta> -m <mensaje>`
- ▶ Para recuperar un tag, basta con hacer `git checkout <tag>`

## pull

## ► pull

Utilizándolo sin ningún parámetro adicional **-git pull origin master** funciona igual que un *update* en Subversion. Es decir, se trae los cambios y hace *merge*. Si le pasamos el parámetro **--rebase**, funciona como un *rebase*.



# ignore

- ▶ **gitignore**

Este fichero se suele colocar en la raíz del proyecto y el contenido suele ser un listado de elementos que no queremos que sean reconocidos como ficheros del repositorio.

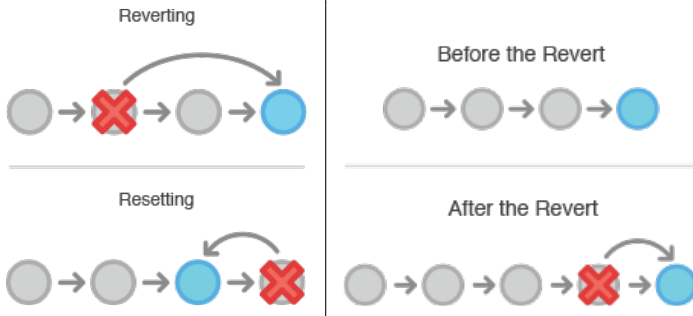
- ▶ **update-index --assume-unchanged**

Este comando se utiliza para ficheros que accidentalmente se han *comiteado* (y probablemente *pusheado*) y no queremos tener en cuenta los cambios producidos en estos ficheros, ya que lo veremos como modificados en la etapa II (listo para *comitear*).

# revert

## ► revert

Este comando deshace un único commit aplicando el parche con la diferencia como un nuevo commit. Ejemplo: `git revert HEAD`



Este comando no destruye la historia ni diverge las ramas *master*. Ejemplo pack commits: `git revert master~2..master`

# soft reset

## ► soft reset

El comando `git reset --soft <hash>` mueve el puntero de la cabeza al hash del commit que le indiquemos.

### BEFORE SOFT RESET



### AFTER SOFT RESET

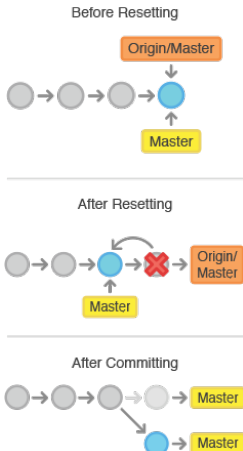




# hard reset

## ► hard reset

El comando `git reset --hard <hash>` mueve el puntero de la cabeza al hash del commit que le indiquemos y **destruye** toda la historia hasta dicho hash.



# stash

## ▶ **stash**

Se utiliza para *aparcar* temporalmente los cambios actuales antes de ser comiteados. El comando se escribe únicamente **git stash**.

```
$ git stash list  
stash@{0}: WIP on master: 049d078 added the index file  
stash@{1}: WIP on master: c264051... Revert added file_size  
stash@{2}: WIP on master: 21d80a5... added number to log
```

Recuperamos los cambios con el comando **git stash apply <id>**

# format-patch

## ► format-patch

Se utiliza para generar parches que se entregarán a mantenedores de repositorios que no aceptan *pull requests*.

Un parche se genera con el comando git **format-patch** --stdout *<hash o rango de hashes>*. Esto genera tantos ficheros con parches como commits se hayan introducido como parámetro.

Para aplicar los parches, se utiliza el comando git **am** --signoff *<file.patch>*

# squash

## ► squash

Se utiliza dentro de lo que se llama el *rebase interactivo*, para unir varios commits en uno solo antes de entregarlos al repositorio remoto. El proceso sería:

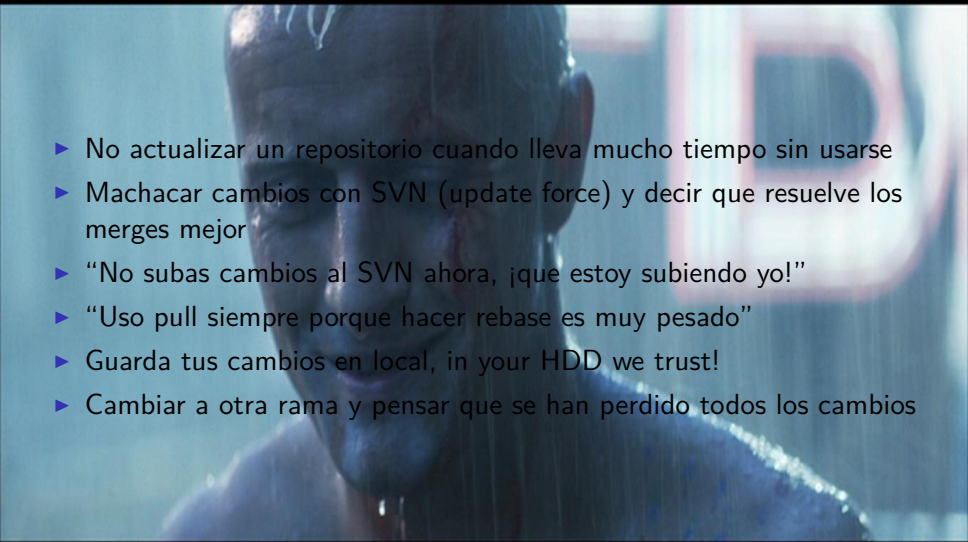
- Comiteamos tantas veces como queramos: `git commit -m "mensaje"`
- Si por ejemplo hemos hecho 2 commits, ejecutamos el rebase interactivo con: `git rebase -i HEAD~2`

```
pick 4ca2acc commit file1
pick 7b36971 commit file2
# rebase 41a72e6..7b36971 onto 41a72e6
#
# commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
# x, exec = run command (the rest of the line) using shell
```

# squash

- ▶ cherry-pick
- ▶ reflog
- ▶ submodules
- ▶ lolcommits

# He visto cosas que no creeríais...

- 
- ▶ No actualizar un repositorio cuando lleva mucho tiempo sin usarse
  - ▶ Machacar cambios con SVN (update force) y decir que resuelve los merges mejor
  - ▶ “No subas cambios al SVN ahora, ¡que estoy subiendo yo!”
  - ▶ “Uso pull siempre porque hacer rebase es muy pesado”
  - ▶ Guarda tus cambios en local, in your HDD we trust!
  - ▶ Cambiar a otra rama y pensar que se han perdido todos los cambios

# Herramientas

- ▶ **IntelliJ** (Windows, Mac, Linux)  
<http://www.jetbrains.com/idea/>
- ▶ **Eclipse** (Windows, Mac, Linux)  
<http://www.eclipse.org/>
- ▶ **SourceTree** (Windows, Mac)  
<http://www.sourcetreeapp.com/>
- ▶ **SmartGit** (Windows, Mac, Linux)  
<http://www.syntevo.com/smartgit/>
- ▶ **Git Tower** (Mac)  
<http://www.git-tower.com/>
- ▶ **TortoiseGit** (Windows)  
<https://code.google.com/p/tortoisegit/>

# Enlaces de interés

- ▶ **Git con calcetines**

<http://danielkummer.github.io/git-flow-cheatsheet/>

<https://www.atlassian.com/git/tutorial/git-basics>

<https://www.atlassian.com/git/workflows>

- ▶ **Successful Git Branching model**

<http://nvie.com/posts/a-successful-git-branching-model/>

- ▶ **Visual Git learning**

<http://pcottle.github.io/learnGitBranching/>

- ▶ **Github vs Bitbucket**

<http://www.infoworld.com/d/application-development/>

[bitbucket-vs-github-which-project-host-has-the-most-22706](#)

- ▶ **Git Android**

<https://github.com/android>

<https://android.googlesource.com/>