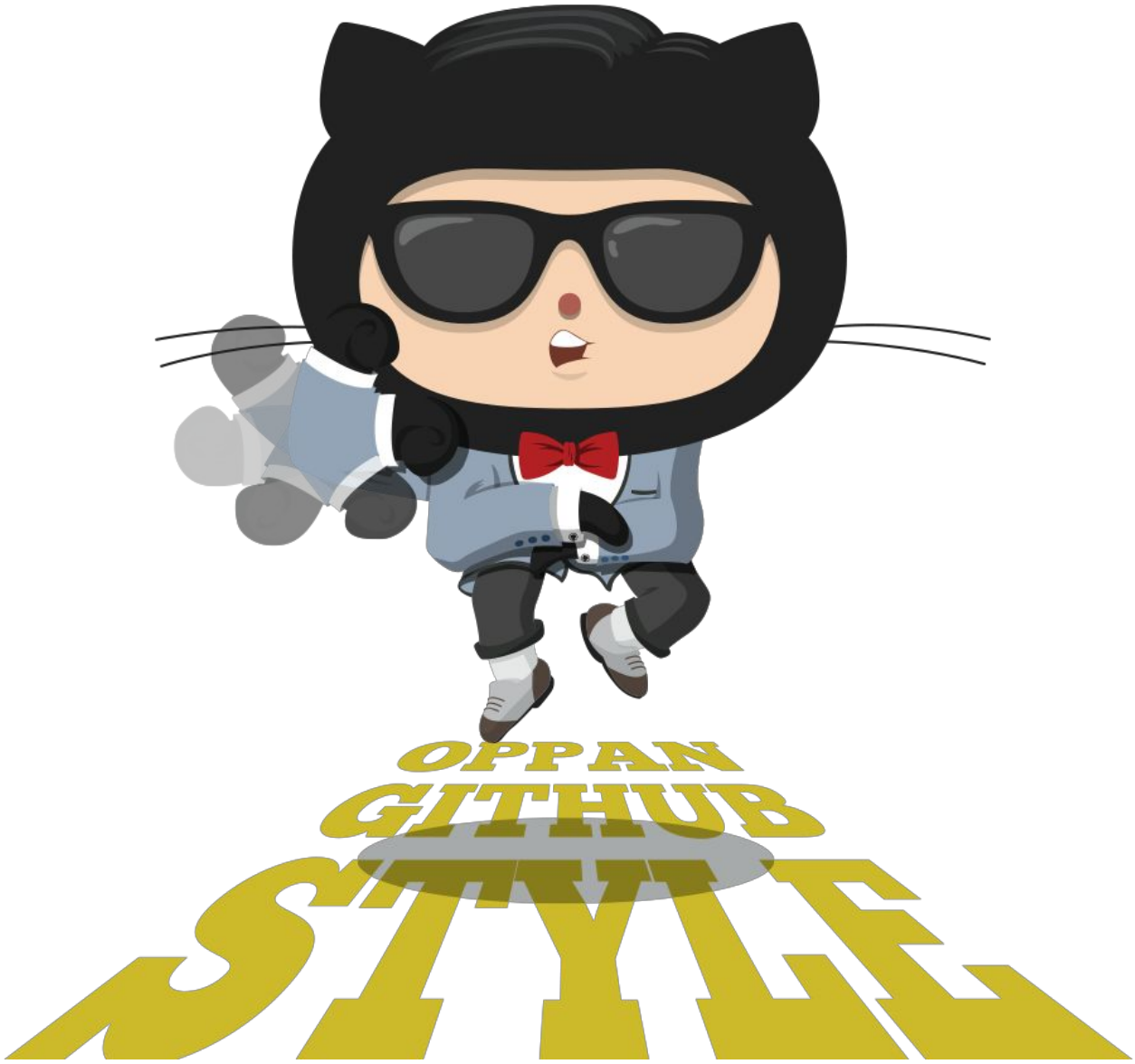
OPPAN
GITHUB
STYLE

Taller de Git

¿Qué es un sistema de control de versiones y cómo afecta directamente en vuestra felicidad?

Los sistemas de control de versiones gestionan los cambios realizados sobre diversos elementos a lo largo del tiempo. En la informática se utilizan habitualmente para gestionar las distintas versiones por las que pasa el código fuente de las aplicaciones, lo que nos permite saber quién realiza qué cambios y poder volver a ellos en un determinado momento. Los más utilizados en este campo a lo largo del tiempo (aunque existen bastante más) son CVS, Subversion y Git.

¿Porqué usar un control de versiones nos hará felices?

- Proporciona copias de seguridad automáticas de los ficheros.
- Permite volver a un estado anterior de nuestros ficheros.
- Ayuda a trabajar de una forma más organizada.
- Permite trabajar de forma local, sin conexión con servidor. (en distribuidos).
- Permite que varias personas trabajen en los mismos ficheros.
- Permiten trabajar en varias funcionalidades en paralelo separado (ramas).

Conceptos generales

Antes de nada, vamos explicar conceptos generales de los sistemas de control de versiones:

Branches

Branches o ramas, son herramientas proporcionadas por los sistemas de control de versiones para poder trabajar paralelamente en distintas versiones del código sin interferir entre sí. Suelen utilizarse ramas de largo recorrido para las principales versiones de la aplicación (*master* para la aplicación en producción, *staging* para aplicación en pruebas...) y ramas puntuales para el desarrollo independiente de distintas funcionalidades.

Commits

Los commits son los puntos de guardado que se van realizando sobre las ramas en los que fijamos los cambios realizados en el código. Podremos volver a ellos para recuperar el estado del código en un determinado momento.

Merge

Un merge es la acciones para juntar una rama sobre otra y unificar los cambios. Por ejemplo para añadir una funcionalidad sobre el proyecto principal. Estas acciones son especialmente críticas cuando trabajan varias personas a la vez, y dependerá de la calidad del sistema de control de versiones que surjan más o menos conflictos a resolver para unificar el código.

Sincronización (Pull/Push)

Son los procesos para sincronizar el estado de nuestros sistemas locales con el servidor de código principal, tanto para subir los cambios locales como para descargar las últimas actualizaciones.

CVS

Cvs (Concurrent Versions System) es uno de los primeros sistemas de control de versiones. Tiene una arquitectura cliente-servidor, en la que el código está almacenado en un servidor central y con el software cliente podemos hacer una copia del código local para hacer cambios y posteriormente volver a subirla al servidor.

Permite el trabajo concurrente entre distintos programadores, pero el servidor solo acepta actualizaciones de los ficheros que estén en la última versión, de forma que los propios usuarios deben actualizar sus copias locales antes de subirlas al servidor. Soporta además el trabajo en distintas ramas.

Se hizo bastante popular entre la comunidad de software libre por ser lanzado bajo la licencia GNU.

SVN

El proyecto de Subversion surgió en el año 2000 con el objetivo crear un sistema de control de versiones con la misma filosofía que CVS, pero arreglando problemas y cubriendo carencias del mismo, como por ejemplo:

- commits atómicos, en CVS un commit interrumpido puede dejar datos inconsistentes.
- La creación de ramas es más eficiente, con complejidad constante a diferencia de CVS que es lineal (aumenta con el número de ramas).
- Manejo de archivos binarios como tal, CVS los trata como archivos de texto.
- Envía los incrementos de los ficheros en la comunicación cliente-servidor, en lugar de los ficheros completos como CVS.

La estructura más habitual en un repositorio SVN:

- **Trunk** con la versión del código principal.
- **Tags** para almacenar las distintas versiones de una aplicación que no volverán a

modificarse.
- **Branches** para gestionar las distintas versiones de código que se desarrollan paralelas al **trunk**.

Al ser también software libre, ha sido adoptado por multitud de proyectos, incluso en grandes corporaciones, convirtiéndose en un referente.

Git

En 2005, Linus Torvalds inició el desarrollo de Git como alternativa a BitKeeper, el sistema de control de versiones que se usó hasta el momento para desarrollar el kernel de Linux. BitKeeper había pasado a ser software propietario con necesidad de licencia y ninguno de los sistemas gratuitos del momento cumplía las necesidades que Linus quería.

Por esto, Git nace con las principales necesidades de ser rápido, eficiente y distribuido:

- Rápido: combinando el trabajo sobre el repositorio local y la posterior distribución remota.
- Eficiente: pensado para manejar grandes proyectos con muchos ficheros (Linux) y no se vuelve lento a medida que la historia del proyecto crece.
- Distribuido: fundamental para el trabajo concurrente y en remoto de muchos usuarios. Cada usuario tiene una copia local en la que trabajar, que posteriormente sincroniza con el resto de usuarios. Es también especialmente efectivo a la hora de mezclar los cambios hechos por distintos usuarios.

Git, como el propio Torvalds dice, es un control de versiones estúpido. Su sencillez es lo que lo diferencia de otros sistemas de control de versiones y lo que le hace verdaderamente potente. La mayoría de sistemas del momento almacenaban los cambios (deltas) en los ficheros entre distintas versiones y requerían de complejos algoritmos de aplicación de esos deltas para recuperar un determinado estado del repositorio. Git en cambio es una sencilla base de datos clave-valor, en la que cada versión apunta a un árbol de nodos que representa la estructura de directorios y contenidos comprimidos de los ficheros. Esto le permite recuperar estados únicamente apuntando a un determinado árbol y facilita las tareas de distribución y mezclado de contenidos.

Además, al ser distribuido y romper con la arquitectura cliente-servidor, cada usuario dispone de un repositorio completo de forma local. Esto permite trabajar sin necesidad de conexión a un servidor para realizar las distintas acciones, pudiendo más tarde sincronizar nuestros datos con el servidor.

Otra de las particularidades de Git respecto a otros sistemas de control de versiones es el área de *stage*. La gran mayoría de sistemas almacenan la información en dos sitios, la copia local de ficheros y directorios del usuario, y el almacenamiento de versiones, ya sean *deltas* en

ficheros o árboles como veíamos antes. Git proporciona una tercera opción con el área de *stage*. Consiste en un área intermedia entre las otras dos, donde ir colocando los cambios de la copia local con las que queremos hacer un nuevo commit. De esta forma se consigue mucha más versatilidad y control a la hora de ir guardando versiones del código.

Presentaciones

- Introducción: http://rsierra.github.io/git/intro/
- Workflow: http://rsierra.github.io/git/workflow/

Instalando git en windows

Bajar e instalar el cliente de windows. http://git-scm.com/download/. En las opciones de instalación, usar el bash (mejor no usar el símbolo de comandos de Windows) y hacer la conversions de finales de línea siempre, para poder trabajar sin problemas con todos los sistemas.

Configurar nombre y correo para los commits de git:

```
$ git config --global user.name ‘Nombre’
$ git config --global user.email ‘usuario@dominio.com’
```

Configuramos las opciones por defecto para pull y merge.

```
$ git config --global branch.autosetuprebase always
$ git config --global merge.ff false
```

Resumen de comandos básicos

Iniciar un repositorio de git en una carpeta:
```
$ git init
```

Descargar repositorio de un servidor:
```
$ git clone <URL>[directory]
```

Comprobar el estado de los ficheros: nuevos, con cambios o pendientes de commit (stage):
```
$ git status
```

Añadir un fichero o todos (.) al área de stage:
```
$ git add <file>
$ git add .
```

Add interactivo, con un menú para ir haciendo distintas acciones, entre ellas patch, para ir añadiendo sólo partes específicas de cada fichero (así podemos meter unas líneas en un commit y otras en el siguiente por ejemplo):
```
$ git add -i
```

Guardar los cambios en el repositorio y dejar descripción:
```
$ git commit -m ‘<message>’
```

Guarda los cambios pendientes en el ‘limbo’ de los commits. Se usa cuando tienes cosas pendientes de commitear, pero necesitas cambiarte a otra rama:
```
$ git stash
```

Saca el ultimo stash de la pila
```
$ git stash pop
```

```
También podemos asignarle un nombre a cualquier stash para después
recuperarlo mediante el mismo:
$ git stash save wadus
$ git stash apply --wadus

Además podemos listar los diferentes stash disponibles:
$ git stash list
```

Log de los commits realizados para ver su información (y SHA):
```
$ git log
```

Detalle de un commit:
```
$ git show <SHA>
```

Comparar las diferencias de los cambios pendientes de commit con los del último commit:
```
$ git diff
```

Identifica quién y cuando ha sido el último en cambiar cada línea en un fichero.
```
$ git blame <file>
```

Ver ramas:
```
$ git branch # locales
$ git branch -r # remotas
$ git branch -a # todas, locales y remotas
```

Crear una rama:
```
$ git branch <branch>
```

Cambiar de rama:
```
$ git checkout <branch>
```

Crear y cambiar a una rama:
```
$ git checkout -b <branch>
```

Visor gráfico por defecto con git:
```
$ gitk
$ gitk --all
```

Mezclar los commits de una rama con la que estamos:
```
$ git merge [--no-ff] <branch>
# Con las configuraciones que recomendamos (git config --global merge.ff
false), el no-ff se hace por defecto
```

Mezcla una rama sobre la que estamos, reescribiendo los commits de la rama en la que estamos después del resto (peligroso):
```
$ git rebase <branch>
```
Normalmente usado para ponerse al día con otro rama, siempre que los commits no se hayan compartido ya:
```
$ git rebase master
```

Conflictos

Cuando hay conflictos, los ficheros con conflicto quedan fuera del stage. Buscamos los las partes con conflicto:

<<<<<
...
=====
...
>>>>>

y dejamos el código oportuno, puede ser el de la rama actual, el de la rama que traemos, la mezcla…

Finalmente añadimos los ficheros al stage (git add) y hacemos ‘git commit’ sin mensaje, apra que el merge continue.

Durante un conflicto, también recoger TODOS (Ojocuidado, hay que estar seguro de mirar los conflictos bien) los cambios de la rama SOBRE la que se mergea y descarta los de la que se mergea:
$ git checkout --ours <path>

Lo mismo pero cambiando las ramas:
$ git checkout --theirs <path>

Comandos trabajando en remoto

Añadir un repositorio remoto (si hemos usado clone, el repositorio remoto ***origin*** será por defecto el repositorio clonado):
`$ git remote add <remote_name> <URL>`

Subir una rama al repositorio remoto:
`$ git push -u <remote_name> <branch>`
Ejemplo típico:
`$ git push -u origin master`
Con -u, recordará la rama y el remoto, la siguiente vez puede hacerse *git push* a secas

Recoger una rama del repositorio, por defecto hace un merge --no-ff para mezclar los cambios:
`$ git pull`

Cuando todo se rompe

Arreglar el último commit:
`$ git commit --amend -m '<new_message>'`

Dejar un fichero como en el último commit:
`$ git checkout <file>`

Quitar un fichero de stage:
`$ git reset HEAD <file>`

`Si rompemos una rama pero queremos conservar los cambios que hemos ido realizando sobre ella, podemos copiar un commit a otra rama:`
`$ git cherry-pick <SHA>`

Revertir un commit (aplicar un nuevo commit inverso a otro):
`$ git revert <SHA>`

Volver a un estado anterior:
`$ git reset <SHA|branch>`
Por defecto (--mixed) mantiene los cambios de los commits eliminados, pero sin aplicar y fuera del stage

`$ git reset --soft <SHA|branch>`
Mantiene los cambios de los commits eliminados, pero sin aplicar y dentro del stage

`$ git reset --hard <SHA|branch>`
Elimina todos los cambios desde el commit indicado

Guía para resolver conflictos:
http://sethrobertson.github.io/GitFixUm/fixup.html

Buenas prácticas: la policía de git.

- Ignorar ficheros para que git no los tenga en cuenta y no se suban al repositorio. Por ejemplo configuraciones personales al entorno de cada usuario, ficheros con datos sensibles como contraseñas, *API keys* de servicios externos… Para ignorar basta con añadir los paths de los ficheros en un fichero *.gitignore* en el raiz del repositorio y commitearlo.
- Hacer merge por defecto con --no-ff. Ejecutar:
  `$ git config --global branch.autosetuprebase always`

  o modificar el fichero .gitconfig, en el directorio raíz del usuario:

  ```
  [merge]
      ff = false
  ```

- Hacer pull siempre con rebase. Ejecutar:
  `$ git config --global merge.ff false`

  o modificar el fichero .gitconfig, en el directorio raíz del usuario:

  ```
  [branch]
      autosetuprebase = always
  ```

- **NO** reescribir la historia ya publicada, es decir no usar rebase sobre commits que ya estén en un remoto.
- Trabajar siempre sobre una rama puntual (features, fixes...), nunca sobre las de largo recorrido. ¡Son gratis!
- Crear commits progresivamente que engloben partes lógicas y claras o cambios concretos (no usar un commit enorme para toda una funcionalidad) ¡También son gratis!
- Hacer siempre un pull en master siempre antes de mezclar una rama sobre master.

Cuando se trabaja con más gente:

- Establecer una misma guía de estilo para escribir el código y usar las mismas configuraciones del editor de texto, eliminación de espacios al final de líneas, salto de line al final de fichero, anidaciones... se evitarán modificaciones innecesarias.
- Hacer commits descriptivos y estructurados:
  - Una primera línea con una descripción general de menos de 50 caracteres.
  - Si es necesario, dejar una línea en blanco y escribir una descripción más detallada, máximo 72 caracteres.
  - Todo esto ayuda a una mejor legibilidad y a posteriores tareas como por ejemplo

escribir un log de cambios o notas de lanzamiento de versión.

Recursos

Tutoriales

- Guía sencilla: http://rogerdudler.github.com/git-guide/index.es.html
- Tutorial interactivo: http://try.github.com/

Libros

- Pro Git: http://git-scm.com/book

Referencia

- Git: http://git-scm.com/docs
- GitHub: https://help.github.com/

Vídeos

- http://git-scm.com/videos
- http://gitcasts.com/

Hosting

- GitHub: https://github.com/
- BitBucket: https://bitbucket.org/