**Introduction to Git**

Section 3: Configuration, Clone, and Git Basic Workflow

Configurar git con: **git config --global user.name** (lo mismo para **user.email**)

Se puede ver qué cosas se han configurado mediante **git config --global –list**

El primer paso es clonar el repositorio creado en Github con: **git clone** [**https://url**](https://url)

En la carpeta que estuviéramos, al hacer el clone se va a crear una carpeta con el nombre del repositorio. Entrando en ella estamos en el repositorio.

Con **git status** obtenemos el estado de nuestro repositorio, y nos indica que estamos en la rama principal: master.

Recordemos que hay:

* working directory
* staging area
* repository

Todo lo que estemos haciendo en el **working directory** ha de ser commiteado.

Al commitear pasa a ser contenido gestionado por el repositorio, en otro caso no.

Si creamos un archivo nuevo dentro de la carpeta en la que trabajamos, gestionada por el repositorio, ese archivo primero está **untracked**. Es un archivo en nuestro **working directory** que no ha sido añadido al repo.

Con **git add start.txt** pasamos el archivo de **untracked** a **staged**, de modo que el repo ya es consciente de su existencia. Después de git add starting.txt el repo nos dice que hay cambios sin entregar relativos a ese archivo.

Para commitear el archivo se hace: **git commit -m "Commiting a new file"**.

Para subir lo commiteado en local a la rama remota: **git push origin master**

Section 4: Text Editor Installation

Se puede configurar git para reconocer notepad++ y que trabajen juntos. Hay que añadir a la variable de entorno path la ruta al ejecutable de notepad++

Para crear un **alias** de línea de comandos, crear en **Users** un **.bash\_profile** y una línea con: **alias npp='notepad++ -multiInst -nosession'**

Se puede configurar un programa como editor por defecto de git con **git config --global -e** Podemos editar la configuración global, para lo que abre el editor por defecto, así podemos probar q todo está OK

[user]

name = Daniel Carregal

email = dani\_carregal@hotmail.com

password = dv310917242942

[core]

editor = notepad++ -multiInst -nosession

[alias]

hist = log --decorate --oneline --all --graph

[merge]

tool = p4merge

[mergetool "p4merge"]

path = C:/Program Files/Perforce/p4merge.exe

[mergetool]

prompt = false

[diff]

tool = p4merge

[difftool "p4merge"]

path = C:/Program Files/Perforce/p4merge.exe

[difftool]

prompt = false

Section 5: Basic Git Commands 22. Starting with a Fresh Project (git init)

**git init repoName:** crea una carpeta y un repo local.

**ls** con git bash: para ver los archivos ocultos es con **ls -la,** o bien **ls -al**

Si entramos en la carpeta **.git**, vemos más carpetas, y es donde reside el repositorio en disco. Son carpetas ocultas.

**git status** nos muestra los **untracked** files. Lo **contrario** a git add, sacar algo de la lista de tracked files, seria **git rm --cached hipster.txt**

23. Adding Git to an Existing Project (git init)

**rm -rf** en gitbash borra una carpeta con todo lo que tenga dentro, sin preguntar.

**git add .** -> añade de forma recursiva todo lo que se encuentra en la carpeta actual.

Si se quiere eliminar la naturaleza git (repositorio) de una carpeta donde reside un proyecto, simplemente se borra la carpeta oculta .git

24. Starting on GitHub by Joining an Existing Project (git clone)

Se puede ir a cualquier repositorio público y hacer un **fork**, eso crea una copia del repo en tu cuenta de github. Después de hacer el fork en tu cuenta de github, para tener la copia local del repo hay que hacer un **git clone** de la url del repo forkeado.

25. Basic Git Workflow (**add, commit, pull & push**)

Antes de hacer un push, se recomienta hacer **git pull origin master** (o la rama en la que estemos trabajando). Entonces ya se puede hacer **git push origin master**

26. Tracked Files

Parece que si modificas un fichero tracked (previamente added), en el commit aun así tienes que hacer add con **-am** (no sé por qué)

**git ls-files** lista todo lo que tiene tracked

28. Recursive Add

Se puede crear una ristra de carpetas anidadas con **mkdir -p level1/level2.../leveln**

Si haces **git status** te muestra el primer nivel, o sea, en una estructura de carpetas anidadas muestra la de fuera. Se puede, por suerte, hacer un **git add .** recursivo para incluir todo lo que haya dentro.

29. Backing Out Changes

Al hacer **git status**, nos da indicación de cómo unstage cambios de ficheros added. Para volver a una versión del último commit y descartar los últimos cambios en un archivo: **git checkout --**

30. Renaming and Moving Files

**git mv nombre1 nombre2** para renombar archivos a nivel filesystem y repository. Git este cambio lo stage, pero no lo commit, de modo que se puede revocar.

Se recomienda cambiar el nombre y comprobar si se ha hecho antes de hacer más cambios, para no liarse más tarde.

**git add -A** da cuenta de si ha habido cambios en el SO que no se han hecho con git, como el cambio de nombre, o borrado. Si se hace el renombrado en el so, al hacer **git status** verá como un borrado y como un nuevo file.

31. Deleting Files

**git rm** borrará del repo el file. Esto borra el archivo físicamente, y pone en **staged** el cambio, pero no lo commit. Haciendo **git reset head** se puede unstage el borrado, pero eso no restaura el archivo en local. Para restaurarlo hace falta **git checkout --**

32. History

Para la history, se usa **git log** : saca los commits en orden temporal decreciente, pero se puede sacar abreviado (el hash sha1) con git log --abbrev-commit

**git log --decorate --oneline --graph** se ve con algo de info gráfica de ramas y el HEAD.

Se puede listar un rango de commits poniendo el primero y el ultimo

**git log --since="3 days ago"**

**git log --follow (**con **--follow** sige los renombramientos, sin el NO)

git show muestra info sobre un commit concreto

33. Git Alias

Para crear versiones cortas de comandos: **git config --global alias= "linea de comando larga de git, con parametros, sin git"**

Section 6: Visual Merge/Diff Tool Installation

P4Merge for Windows Git Configuration

Instalación de **p4merge** para que git lo use como herramienta standard para la resolución de conflictos de código.

**git config --glogal merge.tool p4merge**

También hay que ejecutar **git config --global mergetool.p4merge.path "C:/Program Files/Perforce/p4merge.exe"**

ponemos también **mergetool.prompt false**, para que no se ejecute automáticamente siempre que haya que resolver un conflicto. Hacemos la misma config. pero en vez de **merge.tool**, con **diff.tool**

Section 7: Comparisons Git Repository Setup (for comparing examples)

**git remote add origin URL** para crearle una url al remote de un repo.

**git remote set-url "https://github.com/danicarregal/blahblah"** para set la url si ya la hubiese seteado antes

45. Comparing Working Directory and the Staging Area

Ahora vamos a comarar el working directory con la staging Area

Git status no dice qué se ha modidicado en el archivo, aunque avise de que tiene cambios. Para revisar esos cambios: **git diff**

Si se hace sin parámetros, **git diff** compara la **staged area** con el **working dir**.

Si haces **git difftool** y tienes una herramienta configurada, se abren las versiones y se muestra la comparación en esa herramienta, p4merge.

Lo dicho, sin parámetros adicionales, compara la staging (izq) con el working dir (der)

46. Comparing Working Directory and Git Repository (Last Commit)

Comparar el staging area (cosas added y cosas en control) con el último commit también se puede, claro. Usando el parámetro **HEAD** se compara el working directory con el HEAD del repo, o sea, lo último commited. En las comparaciones, el working directory siempre sale en la parte derecha cuando se usa la difftool.

47. Comparing between the Staging Area and the Git Repository (Last Commit)

**git diff --staged HEAD** para comparar lo que esta stageado con el ultimo commit

48. Limiting Comparisons to one File (or path)

En el caso genérico, un **diff** tiene varios archivos, y a veces tan solo nos interesa ver los cambios de uno. Para limitar a un solo archivo el check de diferencias, **git diff -- Readme.md**

Si haces **git difftool** y son varios archivos, p4merge abre la comparativa uno a uno, o sea, cuando cierras uno automáticamente abre el siguiente.

49. Comparing Between Commits

**git diff ref1 ref2** siendo las refs hashes de commits, o en caso del último, podemos poner HEAD. Y para comparar HEAD con el commit anterior al actual, se usa **git diff HEAD HEAD^**

Cuando se hace **git diff ref1 ref2**, se muestra una lista de diferencias entre archivos concretos, par a par. Se ponen primero los de ref1

50. Comparing Between Local and Remote Master Branches

**git diff** para comparar ramas funciona como con commits, right side compare vs lefts side

Section 8: Branching and Merging 52. Branching Basics

**git branch -a** lista las ramas, locales y remotas.

Se pueden crear branches con **git branch nombrerama**.

El histórico de commits **es el log**, accesible como **git log --oneline –decorate**

Branches are labels, just pointers. En el caso de crear una rama nueva con **git branch nombre** y hacer un log, el último contiene labels.

**git branch -m** sirve para **renombrar**, por ejemplo: **git branch -m dev develop** renombraría la rama dev a nombre develop.

**git branch -d** para **borrar** una rama, pero no puedes estar en ella

53. Happy Path / Fast Forward Merges

**git branch** lista las ramas locales, **git branch -a** lista las locales y las remotas.

**git checkout -b nombreRama** crea **una rama nueva** de nombre nombreRama y hace checkout a esa rama, todo happy path claro.

Para pasar un cambio commiteado en una rama a otra rama, hay que hacer un merge de la segunda en la primera. Antes de hacer un merge es bueno comprobar las diferencias con **git diff o difftool ref1** **ref2**

**git merge sourceBranch**: o sea se mezcla la sourcebranch a la actual.

**fast forward merge** (poner todos los commits de la rama source en la rama tree) solo es posible cuando en la rama tree no hay otros commits.

**git log --oneline --graph**; se ven los commits de forma gráfica, y las ramas con sus rutas. Se ven los mensajes de commit

Una vez hemos hecho merge de la rama develop en la master, ya podemos borrar la rama develop

54. Happy Path / Disable Fast Forward Merges

**git checkout -b nuevaRama**, la crea y nos pasamos a ella. En el **git merge** podemos decir que no lo haga *fast forward* con **--no-ff**: **git merge copyright --no-ff**

Si luego borramos la rama, en el git log simplemente no aparecera más el label de la rama borrada, solo eso

55. Automatic Merges

Si abrimos una rama, hacemos un commit, volvemos a master, hacemos otro commit y luego hacemos un merge, se hace un commit de la mezcla: **merge commit**, se llama.

Automatic merge que resulta en automatic commit.

efectivamente se hace un commit automático tras el merge, con el mensaje -m, y dice: **merge recursive**

56. Conflicting Merges and Resolution

A veces al hacer merges se producen conflictos. Son de utilidad **git diff** o **difftool**

Vamos a forzar un conflicto creando una rama, haciendo un cambio, haciendo commit, cambiando a master y haciendo un cambio en el mismo punto.

Cuando intentamos hacer el merge, no lo hace y se ‘queja’ de que hay cambios conflictivos, hay que arreglarlos antes. En este caso no estamos con el **working directory** en un **clean stage**, sino que estamos en un estado intermedio. Si abrimos el archivo con conflictos, las zonas del archivo que tienen conflicto lo tendrán marcado con **<<>>>>>RAMA2**, para que lo edites y elijas.

Se pueden modificar esos archivos manualmente, pero antes hemos hablado de **mergetools**:

**git mergetool** es la solución. Esto abre la herramienta configurada para ese fin, en nuestro caso **p4merge**

Una vez resuelto de qué versión prevalece cada cosa, hay que hacer **save,** cerrar el p4merge y entonces hacer commit de los cambios. La rama vuelve a ser MASTER, en vez de MASTER|MERGING

Git en estos casos crea un fichero con el archivo original y los conflictos, por si fuese necesario revocar los cambios. Hay que untrackearlo.

Añadir al repositorio un **.gitignore**, y dentro los **nombres de archivos o bien las expresiones** con comodines para excluir los archivos o extensiones

Section 9: Rebasing. Simple Rebase Example

Un rebase se entiende en el siguiente contexto: Se crea una rama para una nueva característica, pero antes de terminar el desarrollo surge la necesidad de cambios menores en la rama principal, que se hacen y se hace el commit. Luego se va a buscar que la rama de la nueva característica contenga también esos cambios**. Llevar a la rama nueva los cambios de la rama principal, que se hicieron a posteriori de la creación de la rama, es lo que se llama rebase**. Tiene sentido: **re-establecer la base** o punto de partida.

Para exponer lo que es rebasing hemos creado una rama nueva y hecho un commit de un cambio en un archivo. Después volvemos a master. Rebase es como hacer un merge en la rama nueva de los nuevos cambios de master.

Hacer rebase de master en nuestra myfeature Branch: aplicar los cambios de master en la rama nueva, posteriores a la creación de la rama.

Se hace como **git rebase master**, en la rama myfeature (la nueva). Seria como aplicar todos los commits nuevos de master antes de la creación de la nueva rama. El instructor dice que con esto se aplana el grafico y que facilita un **merge fast forward** en master una vez terminada **myfeature**.

Hacer un rebase en la rama de la nueva característica facilita hacer un merge fast forward una vez terminado el desarrollo, que es como aplicar todos los commit de la nueva rama sobre el punto de partida de la rama principal.

Si hacemos un nuevo cambio en un fichero en la rama nueva y hacemos commit, HEAD está en **myfeature**, que esta algunos commit por delante de master. Momento de volver a master y traerse lo de la rama nueva. Gracias a haber hecho el **rebase** podemos terminar con un **merge de tipo fast forward**

59. Setup for rebasing conflict

¿Y qué pasa si al hacer un rebase tenemos conflictos?

60. Abort a Rebase

Probando a hacer el rebase se obtiene un mensaje de error informando de los conflictos. El mensaje indica que se puede ABORTAR el rebase.

**git rebase --abort** Con esto estamos fuera del rebase completamente

haciendo **git log --oneline --decorate --all --graph** para ver el historial vemos que tras el abort nos hemos quedado como estábamos antes

61. Rebase Conflict and Resolution

Cuando se hace un rebase con conflicto, nos quedamos en un estado intermedio como cuando falla un merge. En este caso se llama RAMA|REBASE 1/1 y podemos hacer uso de la mergetool.

A las operaciones de merge con mergetool durante un rebase hay que hacerles un **git add** antes de 'continuar' con el rebase

Si hemos resuelto los conflictos hay que hacer **git rebase --continue**

Ahora hacemos un cambio más en la rama nueva y volvemos a master. Si hacemos un merge, sera de tipo ff

62. Pull with Rebase (GitHub)

Ahora haremos la prueba con el repositorio remoto. Hacemos pull como buena práctica, y después hacemos push de todo lo nuevo. ¿Como hacer rebase de cualquier conjunto de cambios del remoto en el repositorio local? Lo vemos en este ejemplo.

Primero se hace commit en local de un cambio que hagamos.

Así, tenemos un commit en local, y ahora vamos a github y hacemos un cambio allí, en el mismo archivo. Luego podemos modificar otro archivo, pero directamente en github, y después hacerle commit allí.

Como hay un commit del que no somos conscientes en github, necesitamos hacer git fetch para renovar las referencias. Haciéndolo se lee: **Your branch and 'origin/master' have diverged, and have 1 and 1 different commits each, respectively.   (use "git pull" to merge the remote branch into yours).** Git fetch no hace ningun cambio en los archivos del repositorio, pero actualiza las referencias

Existe la opción de pull con rebase, de modo que los cambios que vienen del repositorio se ponen por delante de los que tenemos nosotros sin subir (sin push). Se consigue pasando un parámetro adicional al comando pull: **git pull --rebase origin master**. Con esto se aplican primero los commits que hubiera en master y luego los que tuviéramos nosotros sin subir al repositorio.

Section 10: Stashing 64. Simple Stash Example

Para ver qué es un stash, vamos a modificar unos cuantos archivos en el proyecto starter-web. Modificamos **simple.html**. Pero no lo hemos *added*. Sin embargo, ahora necesitamos cambiar otro archivo. Y lo de **simple.html** es un *work in progress*, así que no queremos un commit. Podemos hacer un **git stash** para guardar esos cambios 'para luego'. Después de hacer **git stash**, **git status** nos dice que estamos up-to-date con la rama. Si abrimos el archivo del stash, veremos que está igual que antes de la modificación que hemos guardado ‘para luego'.

Si modificamos el README y hacemos commit, estaremos up-to-date con la rama. Pero ¿cómo recuperamos el archivo del stash de antes? **git stash apply**, trae todos los cambios stashed. Podemos seguir con los cambios. El stash hecho sigue guardado, solo se ha aplicado. Para eliminarlo, hacemos **git stash drop**.

65. Stashing Untracked Files and Using Pop

Git stash solo hace su operación sobre archivos de los que tiene pista git, los que se dicen tracked files. Para saber la lista de tracked files**: git ls-files**. Este es el comportamiento por defecto. Lo podemos comprobar modificando un fichero tracked y luego creando uno nuevo. Si hacemos **git stash**, y luego **git status**, el fichero nuevo sigue apareciendo como untracked y el status no es el de up-to-date, o sea que NO lo guardó. Para hacer esto podemos recurrir a hacer ADD, pero igual tampoco queremos. Entonces, lo que hay que hacer es: **git stash -u** (incluye los untracked). Después de hacer lo que necesitemos y de hacer commit, podemos hacer en una sola operación el **apply** y el **drop**: **git stash pop**

Siguiente prueba, hacer un numero de stashes distintos. Para tener varios stash hay que hacer: **git stash save "mensaje del stash"**. Hemos hecho 3 stash, y al listarlos, los índices están en orden inverso: el primero **{0}** es el más reciente.

Para ver una simulación de la aplicación de un stash concreto, ver qué archivos se verían afectados, podemos hacer: **git stash show stash@{x}** siendo x el índice elegido. Esa referencia se usa luego para los **apply** o **drop**, o **pop**. Después de aplicar un stash lo podemos sacar de la lista con **drop** o con **pop**. Para borrar **todos** los stash de la lista se puede hacer: **git stash clear**.

67. Stashing into a Branch

Para la siguiente prueba, que se trata de hacer stash a una rama, hemos visto que tenemos un cambio sin hacer add. Hacemos dos cambios más y hacemos add de uno de los tres cambios, de modo que con **git status** vemos que tenemos dos cambios no staged para commit, y uno que sí. Creamos un archivo nuevo. Así tenemos archivos en tres *buckets* de git distintos: cambios para commit, cambios no declarados para commit, y archivo no trackeado.

Podemos pasar todo esto, incluyendo el archivo recién creado, pero no trackeado, a una nueva rama que crearemos sobre la marcha: primero hacemos el **stash de todo** incluyendo el archivo nuevo con: **git stash -u**. Luego pasamos ese stash a la creación de una **rama nueva**:

**git stash branch newBranch**.

Esta última línea hace tres cosas: crea la rama, aplica el stash y finalmente lo borra.

Section 11: Tagging 70. Simple Tag Example / Lightweight Tags

Cuando hacemos git log podemos ver tags en el gráfico: **HEAD**, **origin/master**, **master**... HEAD es un **puntero**, los otros son nombres de ramas. Se pueden marcar **puntos relevantes** en la historia de un repositorio mediante etiquetas. Hay una forma de crear etiquetas light, para un commit particular:

**git tag myTag**.

En ese caso en el git log aparecen las etiquetas habituales y otra marcada como tag: myTag. Se asocia al último commit, de modo que se puede usar para ver los cambios de ese commit con:

**git show myTag**. Para eliminar una etiqueta se puede hacer con **git tag --delete myTag**.

71. Annotated Tags

Existen los tags anotados, que tienen un mensaje como en el caso de un commit, y se suelen utilizar para vincularlos a versiones: **git tag -a v-1.0**

***major milestones or version numbers in source code***. Al hacer esto sale el editor por defecto para añadir **mensaje** al tag. **git tag --list** para verlos.

Si hacemos git show con un tag anotado tenemos una info algo distinta al caso del light tag. Por ejemplo, vemos el mensaje introducido en la creación del tag. Sale el creador del tag, la fecha de creación del tag también. Luego el id del commit y el resto de info.

72. Comparing Tags

Con **git commit --amend** te deja modificar el mensaje del commit. Se abre el editor por defecto y puedes modificar el mensaje.

Se puede crear un tag anotado con la siguiente sintaxis tambien:

**git tag v-1.2 -m "Creando la release 1.2"**

**git diff v-1.0 v-1.1** sirve para comparar commits por tags.

Si mirando atrás vemos que hubo algo que debimos taggear y lo queremos taggear ahora: los commit tienen un **hash** asociado, entonces se puede usar para ese fin.

**git tag -a v-0.9beta 324dsfd23scxc23233**

74. Updating Tags

A veces pones un tag en el commit equivocado, este es el enunciado de la lección. Cómo actualizar tags. Tenemos la opcion de borrar el tag y volver a crearlo, o hacerlo todo de una vez: git tag -a v-0.8-alpha --f 'hashCommit'

75. Using Tags with GitHub

Creando tags en el repositorio remoto. Es sencillo haciendo **push a origin del tag.** **Esto crea en Github un .zip y un .tar** con todo el código de ese commit.

**git push origin v-1.1**

Para subir todos los tags locales al repositorio de una vez, hay que ejecutar:

**git push origin master --tags**

En Github, las releases se organizan por tags. Haciendo click en los puntos suspensivos podemos ver el mensaje asociado al tag.

Para eliminar la subida de una release o tag a Github, tenemos que ejecutar localmente el comando:

**git push origin :v-0.9beta**

con esto se dice sube 'nada' asociado a ese tag. Lo que **produce es un borrado remoto de ese tag** (de esa release, que **desaparece de la lista de releases** en github)

Section 12: Bonus: Office Hour Sessions 77. Reset and Reflog (Office Hours, Session 1)

**git log** muestra la historia del repositorio, pero **git reflog** muestra los comandos que hemos ejecutado.

Podemos mover el HEAD del repo haciendo **git reset HEAD^**, o **git reset HEAD^2** para ir moviendo el puntero de la cabeza del repo (a qué commit apunta). Podemos comprobar que se mueve haciendo **git log --decorate --oneline --grap --all**, viendo dónde se sitúa el tag **HEAD**.

En esta sección vemos como recorrer la rama **temporalmente** moviendo el HEAD, para lo que usamos **git reset HEAD^** o **git reset HEAD^^**. Con el primero movemos la cabeza una posición hacia el commit anterior, con doble *carat* lo movemos dos veces. En caso de no existir suficientes commits para los movimientos que queremos realizar, se produce un **error**.

Para mover HEAD hacia delante la técnica es otra. Se hace **git reset 'hashCommit'**. Porque uno se mueva hacia delante o atrás no se pierda nada de la historia, aunque lo pueda parecer.

Tipos de **git reset**, hay 3. Local changes (working directory), Staging area and commit history.

Si se usa el mixed reseteará lo que hay en la staging area y pondrá la cabeza donde le digamos. Otra opción es llamada **'dura'** y es más extrema y destructiva. Si no tienes las cosas 'subidas' y te mueves de commit, se **pierden:** resetea la working directory y la staging area. Y por último el **soft**, que solo mueve el puntero, pero no afecta a la staging area ni al working directory.

78. Compare Branches via GitHub (Office Hours, Session 1)

Antes de hacer un pull puedes comparar en Github. Una forma de mergear ramas en Github es crear una **pull request**. Esto lleva a una pantalla en la que poder ver las diferencias entre las ramas elegidas para la pull request (hacer pull de una rama en otra segunda, es como si en la segunda estuviéramos haciendo merge de la primera).

Antes de hacer, por ejemplo, un merge en master, parece más recomendable hacer una **pull request** en Github para ver los cambios antes de aplicarlos sin más. Muchas veces incluso puede usarse solo para comparar.

79. Compare Branches via Command Line (Office Hours, Session 1)

Hacer **git pull**(estando en la rama X) es por defecto hacer **git pull origin X.**

Si conocemos la existencia de una rama en origin de la que no tenemos referencia en local al hacer **git branch -a**, podemos actualizar esas referencias haciento **git fetch origin nuevarama**. De esta forma, al repetir **git branch -a**podremos ver la rama en la lista de ramas remotas.

Para comparar una rama remota con una local, podemos crear una local con **git branch nombreLocal origin/nombreRemoto**, y luego hacer un **git diff master nombreLocal.**

**git branch test-branch origin/test-fetch**

(Branch test-branch set up to track remote branch test-fetch from origin).

Podemos hacer un merge a master o descartar los cambios y borrar la rama creada en local.

81. Gitting Help (Office Hours, Session 1)

Siempre podemos ver la documentación en línea de git sobre un comando cualquiera. Por ejemplo: **git help stash**

82. Cherry Pick (Office Hours, Session 1)

Imaginamos que tenemos dos ramas, **master** y una **featureBranch**, y que en un momento dado tenemos que aplicar un hotfix en master. Después vamos a querer aplicar ese hotfix también en la rama extra**. Se trata, pues, de coger un commit específico para aplicarlo**. Pueden producirse conflictos, claro, pero para eso tenemos la mergetool.

**git cherry-pick hashcommit**