

Git, software de control de versions

Octubre, 2020 - Antoni Juanico Soler

tjuanico@gmail.com

Contingut del curs

- ▶ Mòdul 1. Desenvolupament no lineal, gestió de branques, mesclat de diferents versions. 19/10/2020 - 1 hora i 30 min.
- ▶ Mòdul 2. Navegació i visualització de l'historial de desenvolupament no lineal. 19/10/2020 - 1 hora i 30 min.
- ▶ Mòdul 3. Gestió distribuïda. 19/10/2020 - 1 hora.
- ▶ Mòdul 4. Magatzems d'informació. 21/10/2020 - 1 hora
- ▶ Mòdul 5. Notificacions de canvis en fitxers. 21/10/2020 - 1 hora
- ▶ Mòdul 6. Renombrat de fitxers. 21/10/2020 - 1 hora
- ▶ Mòdul 7. Reemmagatzament en paquets. 22/10/2020 - 1 hora
- ▶ Mòdul 8. Ordres bàsiques (fetch, merge, pull, commit, push, checkout, etc.). 22/10/2020 - 1 hora
- ▶ Mòdul 9. Fluxos de feina i tipus de branques a Git: Master, development, features, hotfix, release 22/10/2020 - 1 hora

Bibliografia

Si durant el curs creus que utilitzo material sense dir el seu autor pots enviar-me la correcció al correu electrònic tjuanico@gmail.com

- ▶ **Version Control with Git**, Jon Loeliger & Matthew McCullough, O'Reilly, 2nd edition.
- ▶ **Git**, Ry's Git Tutorial, Ryan Hodson
- ▶ <https://es.wikipedia.org/wiki/Git>
- ▶ <https://www.atlassian.com/es/git/tutorials/comparing-workflows>
- ▶ <https://medium.com/@patrickporto/4-branching-workflows-for-git-30d0aaee7bf>
- ▶ <https://www.valenciatech.com/formacion/online/git/puntero-head-git/>
- ▶ <https://git-scm.com/book/es/v2/Git-en-entornos-distribuidos-Flujos-de-trabajo-distribuidos>
- ▶ <https://www.datacamp.com/community/tutorials/git-push-pull>
- ▶ <https://git-scm.com/docs/git-gc>

Introducció a Git

- ▶ És un software **distribuït** de control de versions (CVS) dissenyat pel senyor Linus Torvald
- ▶ El llançà l'any 2005
- ▶ Escrit en llenguatge C, Bourne Shell, Perl
- ▶ És multiplataforma, sota llicència GNU GPL v2
- ▶ Característiques principals:
 - ▶ Cerca ajudar als desenvolupaments no lineals. Per tant ens donarà eines per gestionar “branques” i mesclar diferents versions. Proporcionarà eines per navegar i visualitzar l'historial de desenvolupament no lineal. (mòdul 1)
 - ▶ Gestió distribuïda (mòdul 3). Git ens dóna una còpia local de l'historial de desenvolupament sencer. Així els canvis s'importen com a “branques” addicionals i es poden annexar a la branca “local”.

Introducció a Git

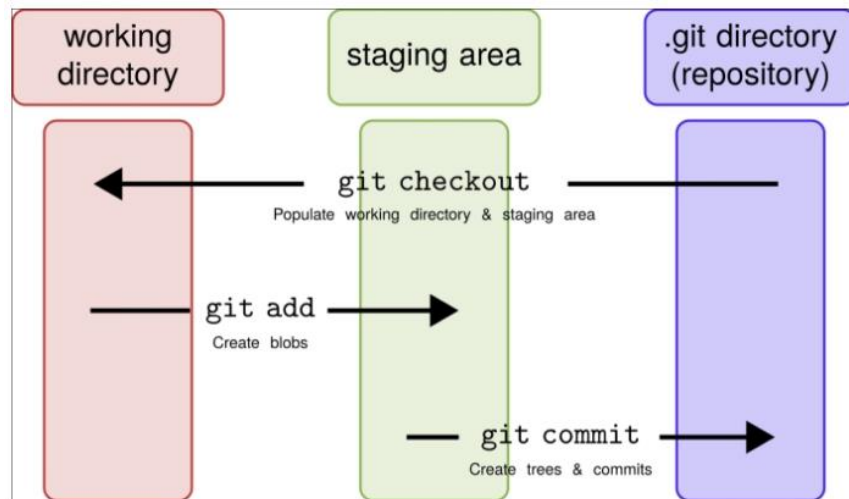
▶ Característiques principals:

- ▶ Els magatzems d'informació es poden publicar per HTTP, FTP, rsync o mitjançant un protocol natiu ja sigui amb una connexió TCP/IP o de manera xifrada SSH.
- ▶ Git també pot emular servidors CVS la qual cosa permet l'ús de clients CVS ja existents i els seus IDE's. Els repositori *subversion* també es poden usar directament (git-svn).
- ▶ Gestió eficient de projectes grans. Gestió ràpida de diferències entre fitxers.

▶ Elements principals a Git

- ▶ El directori de treball (working directory)
- ▶ La foto de l'etapa (Staged snapshot)
- ▶ El conjunt d'etapes validades (committed snapshots)
- ▶ Branques de desenvolupament (development branches) (mòdul 1)

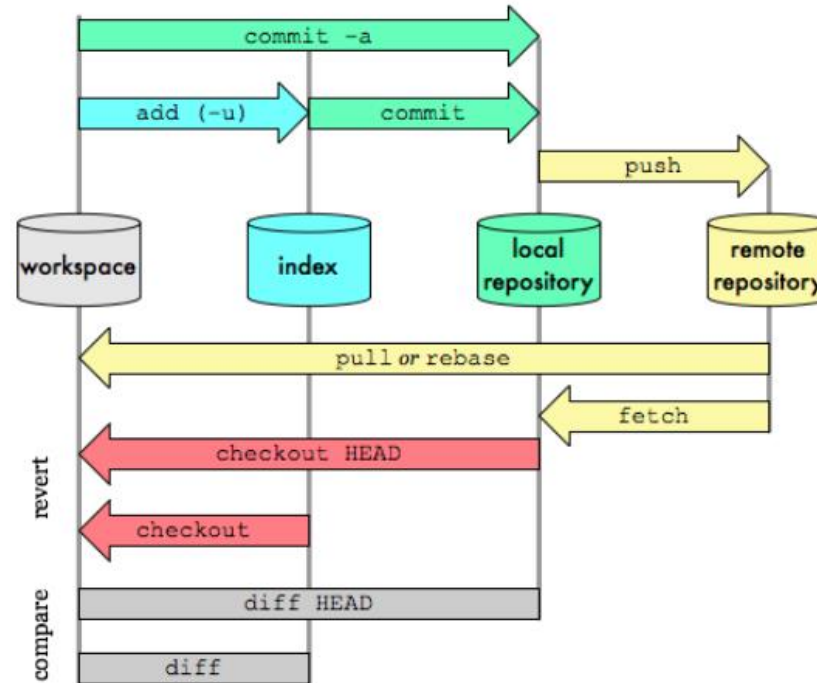
Introducció a Git



1. **Staging.** Telling Git what files to include in the next commit.
2. **Committing.** Recording the staged snapshot with a descriptive message.

Git Data Transport Commands

<http://osteele.com>



Introducció a Git

► Exemple pràctic 1

- Instal·lació git (Windows / Linux) i configuració bàsica
 - `$ git config --global user.name "tjuanico"`
 - `$ git config --global user.email tjuanico@gmail.com`
- Creació de repositori local → `$ git init`
- Veure l'estat del repositori local → `$ git status`
- Afegir fitxers a la branca actual
 - Crear el fitxer `fitxer1.xhtml`
 - Afegir fitxer a l'snapshot → `$ git add fitxer1.xhtml` (podem revisar l'estat)
 - Afegir fitxer al repositori local → `$ git commit | git commit -m "my comment"`

Nota: comentar `$ git commit --amend --reset-author`

Mòdul 1. Desenvolupament no lineal, gestió de branques, mesclat de diferents versions

► Desenvolupament no lineal

Avui en dia el desenvolupament del software no és lineal. Són pocs els projectes que segueixen un desenvolupament en cascada (recordam que quasi és un esquema acadèmic que algunes empreses encara segueixen): Requeriments, Anàlisi, Disseny, Codificació i Proves.

El món és mou a una velocitat on el client té la necessitat de veure i tocar el producte abans de que estigui finalitzat. Les metodologies de desenvolupament ràpid s'imposen: XP, Scrum, Agile, RAD.

► Manteniment no lineal

Un cop tenim el software a l'entorn de producció, durant l'etapa del manteniment el client ràpidament té necessitats diverses: manteniment evolutiu (p. ex. nou navegador, nova versió de base de dades), manteniment correctiu (bugs i errors), manteniment adaptatiu (canvi legislació), manteniment perfectiu (afegir noves funcionalitats)

Mòdul 1. Desenvolupament no lineal, gestió de branques, mesclat de diferents versions

► Gestió de branques

- Què una branca? Dins Git una branca és una línia independent de desenvolupament.
- Permet als desenvolupadors continuar el seu desenvolupament seguint diferents direccions alhora (desenvolupament no lineal). Ens permet aconseguir versions del projecte diferents.
- Una branca es mesclarà i unificarà amb les altres amb l'objectiu de reunificar l'esforç
→ Gestió de branques i mescla de diferents versions per aconseguir una versió madura per ser alliberada (*release version*)
- Git ens permet gestionar quantes branques vulguem sense condicions. La creació de noves branques pot respondre a moltes raons: creació de noves funcionalitats, resolució de *bugs*, canvis cercant millores de rendiment, etc. (mòdul 9).

Mòdul 1. Desenvolupament no lineal, gestió de branques, mesclat de diferents versions

► Gestió de branques, exemple pràctic 2

- Podem veure les branques `$ git branch`
- Podem crear una nova branca amb la seva etiqueta `$ git branch BUG001`
- Ens podem canviar a la nova branca `$ git checkout BUG001`. Podem afegir les modificacions que vulguem. Per exemple un nou fitxer i modificar el primer fitxer
- Podem tornar a la branca master i comprovar l'estat dels fitxers
- Per esborrar una branca `$ git branch -d BUG001`

```
tjuanico@PCTAULA:~/curs/curs-git/projecte2$ git branch
* BUG001
  master
tjuanico@PCTAULA:~/curs/curs-git/projecte2$ ls -l
total 0
-rw-r--r-- 1 tjuanico tjuanico 69 Oct 13 23:10 fitxer1.xhtml
-rw-r--r-- 1 tjuanico tjuanico 101 Oct 13 23:28 fitxer2.xhtml
-rw-r--r-- 1 tjuanico tjuanico 14 Oct 13 23:37 fitxer3.xhtml
```

```
tjuanico@PCTAULA:~/curs/curs-git/projecte2$ git checkout master
Switched to branch 'master'
tjuanico@PCTAULA:~/curs/curs-git/projecte2$ ls -l
total 0
-rw-r--r-- 1 tjuanico tjuanico 69 Oct 13 23:10 fitxer1.xhtml
-rw-r--r-- 1 tjuanico tjuanico 101 Oct 13 23:28 fitxer2.xhtml
```

Mòdul 1. Desenvolupament no lineal, gestió de branques, mesclat de diferents versions

► Mesclat de diferents versions

- Ha d'existir el rol del programador encarregat de la branca màster. Aquest ha de revisar el codi de les altres branques i decidir si podem ser carregades a la seva branca.
- Utilitzem la comana `git merge nom_branca` per a carregar una branca dins la nostra branca actual. Seguint l'exemple anterior, ens posicionam a la branca màster i fem
 - `$ git merge BUG001.`
- Amb aquesta acció podem trobar diferents situacions
 - Carregar o afegir a la màster nous canvis
 - Anar fent canvis a una branca i voler refrescar l'estat de la branca abans d'incorporar-la a la màster.

Mòdul 1. Desenvolupament no lineal, gestió de branques, mesclat de diferents versions

► Mesclat de diferents versions

- **Resoldre conflictes.** En algunes ocasions ens podem trobar que un fitxer ha estat modificat a la branca master (amb commit) i també a una branca auxiliar. En intentar fer un *merge* tindrem un conflicte i aquest s'haurà de resoldre.
- Exemple pràctic 3, anem a crear un conflicte
 - A la branca master afegim un fitxer index3.xhtml, add i commit
 - A la branca feature1, afegim un fitxer index3.xhtml, add i commit.
 - Tornem a la branca master i intentam fer el merge de la branca feature1.

```
tjuanico@PCTAULA:~/curs/curs-git/projecte1$ git merge feature1
Auto-merging index3.xhtml
CONFLICT (content): Merge conflict in index3.xhtml
Automatic merge failed; fix conflicts and then commit the result.

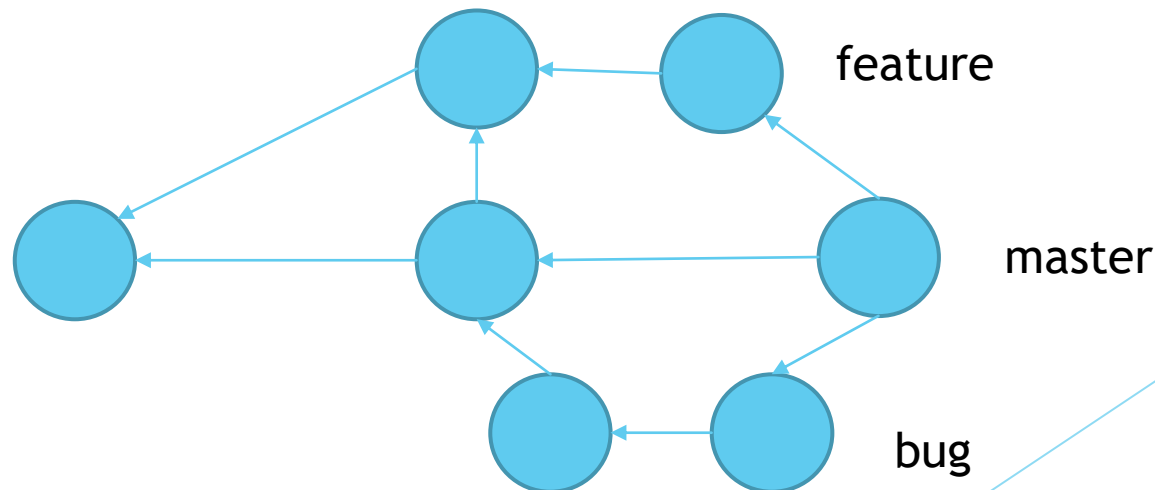
tjuanico@PCTAULA:~/curs/curs-git/projecte1$ git checkout feature1
index3.xhtml: needs merge
error: you need to resolve your current index first
```

Mòdul 1. Desenvolupament no lineal, gestió de branques, mesclat de diferents versions

► Mesclat de diferents versions

► Exemple 3, resolent el conflicte

- Ens informa que hi ha un conflicte i si miram el fitxer veurem que ens ha afegit les línies i que ens indica que hi ha un problema. Hem de resoldre el conflicte i fer un “add i commit” perquè la resolució finalitzi i la branca figura como carregada.
- **Golden rule:** la branca master es sagrada i sempre té versions candidates a release. Abans de carregar branques a la branca master convé sincronitzar la branca **no** master. Només així podrem avançar-nos als conflictes i saber si el tindrem



Mòdul 2. Navegació i visualització de l'historial de desenvolupament no lineal.

- ▶ Poder veure l'historial d'una branca i/o de la branca en la que ens trobem
- ▶ \$git log | git log -oneline | \$ git log master

```
tjuanico@PCTAULA:~/curs/curs-git/projecte1$ git log
commit cccf8e8dbddf4d07cf571e236dea843d0babaea8 (HEAD -> feature1)
Author: Toni Juanico <tjuanico@PCTAULA.localdomain>
Date:   Mon Oct 12 23:31:59 2020 +0200

    Afegim funcionalitat F003

commit 866a887f3f3e815a1efb101934423f00bdc394ac (master)
Author: Toni Juanico <tjuanico@PCTAULA.localdomain>
Date:   Mon Oct 12 23:30:02 2020 +0200

    Segon fitxer que hem introduït

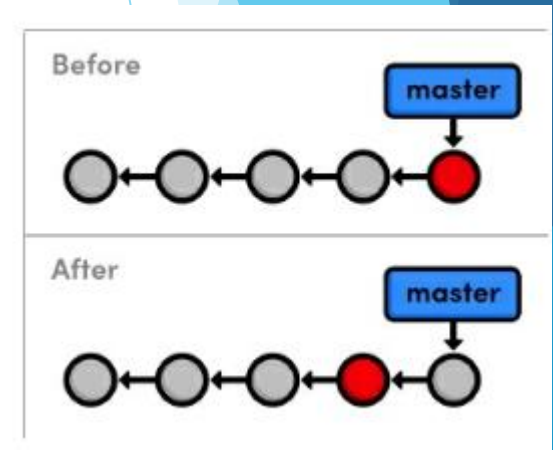
commit 2e3879743a84e4af323c6ce3a5fd591ea816e354
Author: Toni Juanico <tjuanico@PCTAULA.localdomain>
Date:   Mon Oct 12 22:01:07 2020 +0200

    Canvi de millora M001
```

```
tjuanico@PCTAULA:~/curs/curs-git/projecte1$ git log --oneline
cccf8e8 (HEAD -> feature1) Afegim funcionalitat F003
866a887 (master) Segon fitxer que hem introduït
2e38797 Canvi de millora M001
tjuanico@PCTAULA:~/curs/curs-git/projecte1$
```

Mòdul 2. Navegació i visualització de l'historial de desenvolupament no lineal

- ▶ **Concepte HEAD a Git**, és un punter que utilitza internament Git per indicar l'snapshot que actualment es troba “checked out”.
- ▶ Sempre apunta al darrer commit realitzat de la branca a on ens trobem.
- ▶ Git crea el HEAD quan realitzam el primer commit.
- ▶ Ens podem moure pels diferents nodes de la branca (commit) i el punter (HEAD) es mou
 - ▶ Si fem un `$ git checkout 866a887` el HEAD canvia a ‘detached HEAD’ state.
 - ▶ En l'estat ‘detached HEAD’ podem mirar, experimentar i fer commit. Això ens crearà un snapshot “orfe”. Ara bé si volem que aquests siguin persistents haurem de crear una nova branca fent un nou checkout → `git checkout -b <nova_branca>`. Si no ho fem i no tenim apuntat l'id del canvi el podem perdre!!!. Alertau!.
 - ▶ Exemple d'aplicació ens demanen afegir logs per a trobar un error en producció (recorda la Golden rule)
 - ▶ Per tornar al head → `$ git checkout <nom_branca>`



Mòdul 2. Navegació i visualització de l'historial de desenvolupament no lineal


```
tjuanico@PCTAULA:~/curs/curs-git/projecte1$ git status
On branch feature1
nothing to commit, working tree clean
tjuanico@PCTAULA:~/curs/curs-git/projecte1$ git log --oneline
cccfbe8 (HEAD -> feature1, tag: FUNC01) Afegim funcionalitat F003
866a887 (tag: FUNC02, BRANCA02) Segon fitxer que hem introduït
2e38797 Canvi de millora M001
tjuanico@PCTAULA:~/curs/curs-git/projecte1$ git checkout 866a887
Note: checking out '866a887'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

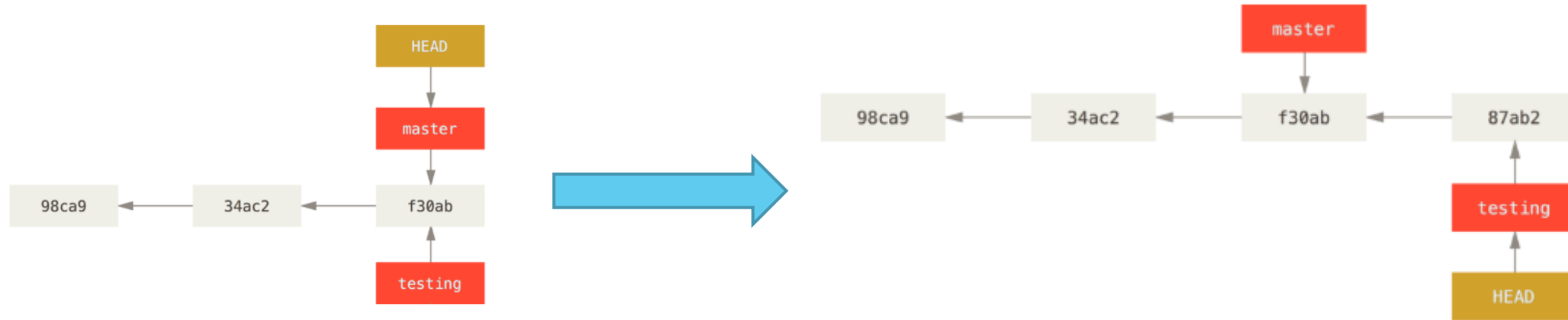
If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

    git checkout -b <new-branch-name>

HEAD is now at 866a887 Segon fitxer que hem introduït
tjuanico@PCTAULA:~/curs/curs-git/projecte1$
```



Mòdul 2. Navegació i visualització de l'historial de desenvolupament no lineal

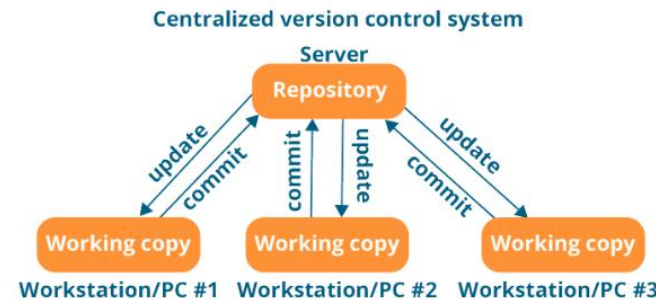


Fuente de las imágenes y más información : <https://bit.ly/2PpqyNM>

Mòdul 3. Gestió distribuïda

► Gestió distribuïda vs gestió centralitzada

- Un model bastant habitual de desenvolupament usant control de versions és el **centralitzat**. Hi ha un repositori central que conté totes les versions a partir de les quals els desenvolupadors obtenen una còpia en local, la modifiquen i la integren. El primer desenvolupador no tindrà problemes però el segon tindrà que resoldre conflictes i per tant fer un “merge” local abans de pujar-ho al servidor centralitzat.
- Aquest model és vàlid a Git, CVS, Subversion. I és l'utilitzat fins ara al curs. Si el vostre entorn de desenvolupament (empresa, equip, organització) ho té assumit així es pot fer feina d'aquesta manera. Ara bé, si el servidor centralitzat cau cap membre de l'equip pot seguir fent feina. S'han de tenir backups i s'ha d'anar amb compte.

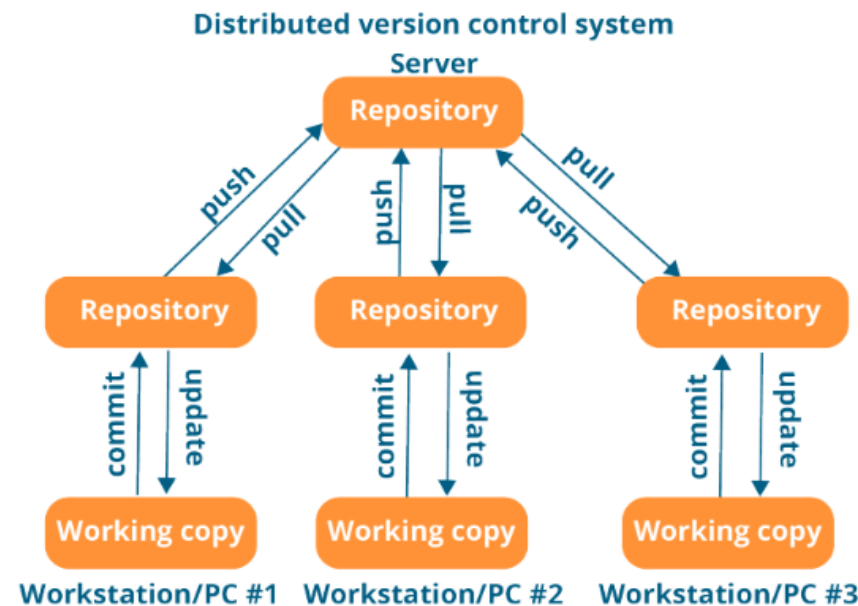


Mòdul 3. Gestió distribuïda

- ▶ En els repositoris distribuïts no existeix aquesta idea centralitzadora. Es a dir, cada desenvolupador té una còpia en local amb tota la informació: històrics, branques, etiquetes, etc. Per tant, no necessiten estar connectats tota l'estona per a realitzar operacions contra el repositori.
- ▶ Exemple d'aplicació
 - ▶ Una comunitat autònoma desenvolupa un software força bé, té èxit. Aquesta comunitat autònoma té el seu repositori Git (A). La llei 39/2015 i la llei 40/2015 preveuen col·laboració entre administracions. El software és públic i qualsevol administració es pot beneficiar d'ell. L'adapta a la seva comunitat autònoma però aquesta adaptació com a tal es fa a un repositori nou (B). Dins aquest marc de col·laboració la comunitat autònoma fa canvis adaptats propis i resol bugs que ha trobat i ho notifica a la comunitat autònoma (A). Les branques de resolució de bugs del repositori B s'afegeixen al repositori A. Totes les comunitats es veuen beneficiades.

Mòdul 3. Gestió distribuïda

- ▶ Es tenen moltes còpies del repositori “original”. En cas de desastre es pot obtenir “usar” el repositori de qualsevol dels desenvolupadors locals. Parlem de “clons”.
- ▶ Apareixen dues noves accions “pull” i “push”
- ▶ Problema, sincronització i resolució de conflictes entre repositori.



Cas pràctic mòduls 1, 2 i 3

1. Tenim el repositori remot de githut següent: <https://github.com/tjuanico/curs-git>
2. Clonau el repositori remot a un repositori vostre en local, podeu seguir les instruccions de: <https://docs.github.com/es/free-pro-team@latest/github/creating-cloning-and-archiving-repositories/cloning-a-repository>. El podeu clonar amb https: <https://github.com/tjuanico/curs-git.git> [1]
3. Afegiu un fitxer nou al projecte. Per evitar col·lisions creau un fitxer amb els 3 darrers nombres del vostre dni (per exemple 513.xhtml). Dueu a terme les operacions “add, commit (comentari el vostre nom sense llinatges)”.
4. Realitzau una petició per afegir-ho al repositori remot. (cercau per internet, pista: push origin)
5. Quan tothom hagi apujat el seu fitxer vos avisaré perquè actualitzeu el vostre repositori local. (cercau per internet, pista: pull) [2]
6. Discussió [1] git clone REMOTEURL tag vs git remote add tag REMOTEURL
7. Discussió [2] pull vs fetch

Mòdul 4. Magatzems d'informació

- ▶ Git és un sistema d'arxiu orientat a contingut. El *core* de Git és un simple magatzem de claus i valors.
- ▶ Amb Git, qualsevol inserció en aquest magatzem d'informació ens retornarà una clau que podrem utilitzar en qualsevol moment per recuperar el contingut inserit prèviament.
- ▶ Cal conèixer quins tipus d'objectes es guarden al magatzem de Git i com podem interactuar amb ells en cas de necessitat.

Mòdul 4. Magatzems d'informació

► Tipus d'objectes a Git

- El *core* del repositori Git té un magatzem d'objectes. Aquests objectes poden ser de quatre tipus: blobs (binary large object), trees (arbres), commits i tags (etiquetes)
- **Blobs**, cada versió d'un fitxer s'emmagatzema dins un blob. No conté cap metadada ni tan sols el seu nom.
- **Tree**, l'objecte arbre representa la informació d'un directori a un nivell donat. Desa els identificadors dels blobs, el path names i metadades dels fitxers que es troben a un directori.
- **Commits**, contenen metadades per a cada canvi introduït al repositori, l'autor, la data, el comentari. Cada commit apunta a un objecte de tipus tree.
- **Tags**, nom en format humà d'un objecte del repositori (normalment un commit. Per exemple el commit 9da581d910c9c4ac93557 pot tenir el tag versió-1.5-alpha)

Mòdul 4. Magatzems d'informació

- ▶ Per exemple, examinar un objecte COMMIT
 - ▶ \$ git cat-file commit HEAD, ens permet veure la informació de l'objecte de tipus "commit" que té l'etiqueta HEAD.

```
tjuanico@PCTAULA:~/curs/curs-git/projecte1$ git cat-file commit HEAD
tree d0025c5b07321121fdc9793ef57b0d6a91f1cc63
parent 3328849466d9d48dcc47cb753d28a4e0176ef8aa
parent 016e3d0391e4f05f6fd69649dd0c0611aa78e9ef
author tjuanico <tjuanico@gmail.com> 1602962603 +0200
committer tjuanico <tjuanico@gmail.com> 1602962603 +0200

Merge branch 'feature2'
```


Mòdul 4. Magatzems d'informació

- ▶ Per exemple, examinar un objecte TREE
 - ▶ \$ git ls-tree HEAD,

```
tjuanico@PCTAULA:~/curs/curs-git/projecte1$ git ls-tree HEAD
100644 blob 6760892daaee4a9837ca982b02c5c263a5322d37    index.xhtmll
100644 blob 6760892daaee4a9837ca982b02c5c263a5322d37    index2.xhtmll
100644 blob 5be86d6372cea940bbaabd6371e410f92d35ed71    index3.xhtmll
100644 blob 549245ae6df8ed49942411315919b80a689daef4    index4.xhtmll
100644 blob fe43814f278f122c5429991308fdbd627e9d4227    index5.xhtmll
```

- ▶ Notau que no hem utilitzat la comana anterior (cat-file). El motiu és que els objectes de tipus tree dins el magatzem es guarden com a dades binaries i per tant la visualització (cat) del fitxer no es en format “humà”.

Mòdul 4. Magatzems d'informació

- ▶ Per exemple, examinar un objecte BLOB
 - ▶ `$ git cat-file blob <id_objecte>`

```
tjuanico@PCTAULA:~/curs/curs-git/projecte1$ git ls-tree 3328849
100644 blob 6760892daaee4a9837ca982b02c5c263a5322d37    index.html
100644 blob 6760892daaee4a9837ca982b02c5c263a5322d37    index2.html
100644 blob 70201e18dd08f36857c33df19d94b38e600cd1aa    index3.html
100644 blob 549245ae6df8ed49942411315919b80a689daef4    index4.html
100644 blob fe43814f278f122c5429991308fdbd627e9d4227    index5.html
tjuanico@PCTAULA:~/curs/curs-git/projecte1$ git cat-file blob fe43814f278f122c5429991308fdbd627e9d4227
<html>
<====<<<<<<< HEAD
      <body></body>
      <head></head>
>>>>>>> feature1
</html>
```

- ▶ Això ens mostra el contingut ja que els objectes blob contenen text pla.

Mòdul 4. Magatzems d'informació

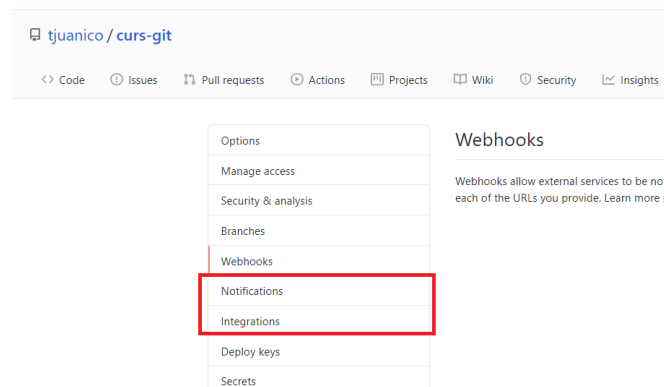
- ▶ Per exemple, examinar un objecte TAG
 - ▶ `$ git cat-file tag v2.0`
 - ▶ Hem fet poca feina amb els tags,
 - ▶ Crear tag: `$ git tag <tag_name> HEAD`
 - ▶ Visualitzar tags creats: `$ git tag -n`

```
tjuanico@PCTAULA:~/curs/curs-git/projecte1$ git tag -n
FUNC01          FUNC01
FUNC02          Això és el meu tag
tjuanico@PCTAULA:~/curs/curs-git/projecte1$ git cat-file tag FUNC01
object cccfbe8dbddf4d07cf571e236dea843d0babaea8
type commit
tag FUNC01
tagger tjuanico <tjuanico@gmail.com> 1602957840 +0200

FUNC01
```

Mòdul 5. Notificacions de canvis de fitxers

- ▶ Dins un entorn col·laboratiu i distribuït com Git ens interessant que tots els que fan ús d'un determinat repositori tinguin coneixement dels canvis introduïts (alguns poden afectar directament a la seva feina).
- ▶ Les notificacions de canvis de fitxers es poden fer de dues maneres:
 - ▶ Webhook: notificació automatitzada a una url
 - ▶ Notificacions a correus, sempre millor una llista de correu

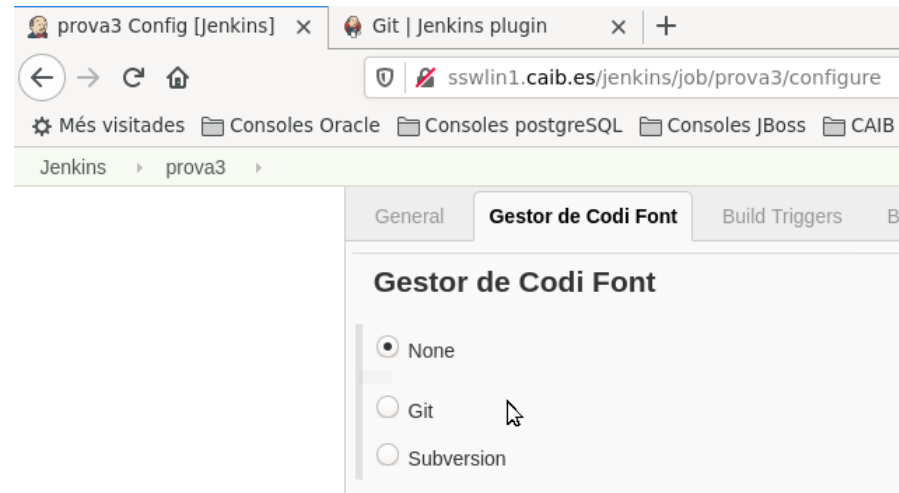



Mòdul 5. Notificacions de canvis de fitxers

- ▶ Un exemple d'aplicació del WebHook és el de poder disparar un nou desplegament al nostre Jenkins quan s'incorpora un commit a la branca màster.
- ▶ La pròpia documentació de Jenkins ens diu com ho hem de fer: <https://plugins.jenkins.io/git> (Pushing notification from repository)
- ▶ Al mateix temps, un cop Jenkins té “l'ear” construït podríem utilitzar els WebHook de Jenkins per generar automàticament el cai a la DGTIC.





Mòdul 5. Notificacions de canvis de fitxers





 **Jenkins**


Jenkins > concsv-2 >


 Retornar al Panell de Control


 Estat


 Canvis


 Espai de treball

 Construir Ara









 Suprimeix Maven project

 Configura

 **Modules**

 Rename

Modules

S	W	Name ↓	Darrer muntatge correcte
		concsv	1 yr 3 mo - #65
		concsv-ear	1 yr 3 mo - #65
		concsv-ejb	1 yr 3 mo - #65
		concsv-front	1 yr 3 mo - #65

Icona: [S](#) [M](#) [L](#)

Mòdul 6. Renombrat de fitxers

- ▶ Git vigília els canvis realitzats als fitxers del directori de treball del nostre repositori.
- ▶ Ho fa utilitzant com a guia el nom del fitxer (o directori).
- ▶ Si nosaltres renombrem un fitxer, als ulls de Git tenim:
 - ▶ S'ha creat un nou fitxer (amb el nom nou)
 - ▶ S'ha esborrat un fitxer
- ▶ Per tant, per Git, renombrar o moure un fitxer són operacions idèntiques.
- ▶ Per renombrar fitxers o directoris dins Git hem d'utilitzar la comana:
 - ▶ `$ git mv <nom_fitxer> <nou_nom_fitxer>`
 - ▶ Haurem de fer el corresponent commit

Mòdul 6. Renombrat de fitxers

- ▶ Se'ns presenta un problema. Si utilitzem IDEs (eclipse, per exemple) i renombram o movem un fitxer sense tenir en compte el control que fa Git podem perdre informació.
- ▶ Suposam que tenim un fitxer (index.xhtml) que té un llarg historial de canvis (més de 150 anotacions). Si amb un IDE el renombram cara a Git l'esborram (i amb ell tot el seu historial) i en donam d'alta un de nou sense historial.
- ▶ Es per aquest motiu que utilitzam la comana `$ git mv | git mv -f | git mv -force`
- ▶ Exercici 10, observeu el comportament si no utilitzam `git mv`

Mòdul 6. Renombrat de fitxers

- ▶ Exercici 10, observeu el comportament si no utilitzam git mv
 - ▶ En el repositori no s'esborra el fitxer i tenim dos fitxers de versions diferents. És un error molt comú dels programadors que no coneixem Git!!

```
tjuanico@PCTAULA:~/curs/curs-git-remote/curs-git$ mv alta_new2.xhtml index3.xhtml
tjuanico@PCTAULA:~/curs/curs-git-remote/curs-git$ git status
On branch development
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        deleted:    alta_new2.xhtml

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        index3.xhtml

no changes added to commit (use "git add" and/or "git commit -a")
```

Mòdul 7. Reemmagatzament en paquets

- ▶ Recordar que al llarg del curs hem dit que Git no basa el seu funcionament en desar els canvis d'un fitxer (delta) sinó en guardar una imatge sencera de cada snapshot. Així el seu funcionament es molt ràpid, però pot ser ineficient pel que comentarem a continuació.
- ▶ De manera ordinària, Git comprimeix el contingut dels fitxers del nostre repositori utilitzant zlib. D'aquesta manera estalviem molt d'espai a disc.
- ▶ Ara bé, suposam que en el nostre repositori desam un arxiu d'un megabyte (pes considerable).
- ▶ Si tan sols afegim una línia al fitxer i fem un commit obtindrem un blob completament nou, els fitxers s'han duplicat i per tant el nostre repositori pot créixer desmesuradament

Mòdul 7. Reemmagatzament en paquets

► Exemple

```
tjuanico@PCTAULA:~/curs/curs-git-remote/curs-git$ git cat-file -p development^{tree}
100644 blob fd98069f20b50bc5480fe19e4ad13d527c8243d1    alta.xhtmll
100644 blob 8e4711e333ea53ce367441fc3fdb1455c33edbc3    index.xhtmll
100644 blob 591e59e6a785672ab31c7603f14114c5d91c24a2    llistat.xhtmll
100644 blob 8621c2fe7faa55b4c7ab319eeb7bba81a048d0fb    llistat2.xhtmll
tjuanico@PCTAULA:~/curs/curs-git-remote/curs-git$ vim llistat.xhtmll
tjuanico@PCTAULA:~/curs/curs-git-remote/curs-git$ git add llistat.xhtmll
tjuanico@PCTAULA:~/curs/curs-git-remote/curs-git$ git commit -m "una altre canvi"
[development 80af39d] una altre canvi
 1 file changed, 2 insertions(+)
tjuanico@PCTAULA:~/curs/curs-git-remote/curs-git$ git push origin development
```

```
tjuanico@PCTAULA:~/curs/curs-git-remote/curs-git$ git cat-file -p development^{tree}
100644 blob fd98069f20b50bc5480fe19e4ad13d527c8243d1    alta.xhtmll
100644 blob 8e4711e333ea53ce367441fc3fdb1455c33edbc3    index.xhtmll
100644 blob 6b5aa08835545d91faa2c82132a3d9e08225bf57    llistat.xhtmll
100644 blob 8621c2fe7faa55b4c7ab319eeb7bba81a048d0fb    llistat2.xhtmll
```

Mòdul 7. Reemmagatzament en paquets

- ▶ En aquest escenari si que podria ser interessant que Git adoptés un comportament de l'estil només desar el fitxer original (mida gran) i desar un petit conjunt de canvis (delta).
- ▶ Git té una funcionalitat semblant. Inicialment Git desa els objectes en el disc amb un format que ell anomena “loose” (loose object).
- ▶ Ocasionalment Git empaqueta un conjunt de “loose” objects dins un objecte binari que anomena “packfile”. D'aquesta manera implementa un mecanisme eficient per resoldre les situacions com les que acabam de descriure
- ▶ Podem forçar aquesta funcionalitat amb `$ git gc`

```
tjuanico@PCTAULA:~/curs/curs-git-remote/curs-git$ git gc
Counting objects: 14, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (14/14), done.
Writing objects: 100% (14/14), done.
Total 14 (delta 5), reused 0 (delta 0)
```

Mòdul 8. Ordres bàsiques

- ▶ `git init`: converteix el directori en un repositori Git
- ▶ `git status`: visualitza l'estat del repositori.
- ▶ `git add <nom_fitxer>`: afegeix el fitxer a l'snapshot pel proper staging
- ▶ `git fetch`:
- ▶ `git merge <nom_branca>`
- ▶ `git pull`
- ▶ `git commit -m "<missatge>"`
- ▶ `git push origin <nombre_branca>`
- ▶ `git status`: mostre l'estat actual de la branca i els canvis pendents

Mòdul 8. Ordres bàsiques

- ▶ `git add <nom_fitxer>`
- ▶ `git checkout -b <nom_branca_nova>`: crea una nova branca a partir de l'actual i apunta a la nova branca
- ▶ `git checkout -t origin/<nom_branca>`:
- ▶ `git branch`: llista totes les branques
- ▶ `git branch -a`:
- ▶ `git branch -d <nom_branca>`: esborra una branca
- ▶ `git push origin <nom_branca>`:
- ▶ `git remote prune origin`:
- ▶ `git reset -hard HEAD`:
- ▶ `git revert <hash_commit>`:

Mòdul 8. Ordres bàsiques

- ▶ `git log` | `git log -oneline`
- ▶ `git merge <nom_branca>`: Carrega la branca dins la branca actual (checked-out)
- ▶ `git rm <fitxer>`: esborra un fitxer del directori de treball (si es possible) i atura el seu seguiment (stop tracking the file)
- ▶ `Git mv <nom_fitxer> <nou_nom_fitxer>`: canvia el nom a un fitxer

Mòdul 9. Fluxos de feina i tipus de branques a Git

- ▶ Un flux de treball a Git és una fórmula o una recomanació envers l'ús de Git per a realitzar un treball de manera uniforme i productiva.
- ▶ Git es centra en la flexibilitat i per tant no existeix un procés estandaritzat de com interactuar amb ell.
- ▶ Per això és necessari que l'equip de treball (desenvolupadors) estableixin quin és el flux de feina més convenient per la seva dinàmica i que tot l'equip el conegui i el segueixi.
- ▶ Existeixen diferents fluxos de feina (workflows) publicats i recomanats: Git-Flow, GitHub-flow, GitLab Flow o One Flow

Mòdul 9. Fluxos de feina i tipus de branques a Git

► Git-Flow

- Vicent Driessen el proposà l'any 2010 i es basa en dues branques de temps de vida il·limitat i la resta són branques de recolçament:
- **Master:** és la branca principal. Conté el repositori de codi font que actualment es troba a producció, per la qual cosa **aquesta branca sempre s'ha de mantenir estable.**
- **Development.** És una branca obtinguda a partir del Master. És la branca d'integració, totes les noves funcionalitats s'han d'integrar en aquesta branca. Després de que es realitzi la integració i es corregeixin els errors (si se'n detecta algun), es el moment en que es pot fer un merge damunt la branca Master.

Mòdul 9. Fluxos de feina i tipus de branques a Git

► Git-Flow

- **Features:** cada nova funcionalitat s'ha de realitzar damunt una branca nova. Aquestes s'han de treure de la branca *Development*. Un cop que la funcionalitat s'ha desenvolupat, es fa un merge damunt la branca *Development*, a on s'integrarà amb les altres funcionalitats.
- **Hotfix:** Són errors de software que sorgeixen en producció, per la qual cosa s'han d'arreglar i publicar de forma urgent. És per això, que són branques generades a partir del *Master*. Un cop corregit l'error, s'ha de fer una unificació de la branca damunt el *Master*. Finalment, perquè la branca *Development* no quedi des actualitzada, s'ha de realitzar la seva unificació (de *Master* damunt *Development*).
- **Release:** Les branques release recolzen la preparació de noves versions de producció. S'arreglen molts d'errors menors i es preparen correctament les metadades.