

Sistemas de Control de Versiones

1.2 Control de versiones con Git

Elena García-Morato, Felipe Ortega, Enrique Soriano, Gorka Guardiola

GSyC, ETSIT. URJC.

Grupo de Innovación Docente Laboratorio de Ciencia de Datos para la Innovación de la Enseñanza
(DSLAB-TI)

6 de julio, 2023





(cc) 2014-2023 Elena García-Morato, Felipe Ortega
Enrique Soriano, Gorka Guardiola.

Algunos derechos reservados. Este trabajo se entrega bajo la licencia
Creative Commons Reconocimiento - NoComercial - SinObraDerivada
(by-nc-nd). Para obtener la licencia completa, véase
<https://creativecommons.org/licenses/by-nc-nd/3.0/es/>.

Sistemas de Control de Versiones

1.2 Control de Versiones con Git

Contenidos

- 1.2.1 Control de versiones con Git
- 1.2.2 Configuración
- 1.2.3 Registro de cambios
- 1.2.4 Sesión básica
- 1.2.5 Repositorios remotos

1.2.1 Control de versiones con Git

¿Qué es Git?

- SCV (sistema de control de versiones).
- Distribuido.
- Creado por Linus Torvalds (proyecto kernel Linux) en 2005.
- Para desarrollar el kernel de Linux sustituyendo a Bitkeeper.

SCV

- Permite sincronizar trabajo en archivos del proyecto (código fuente y otros).
- Hacer, deshacer, mezclar cambios (*merge*), agruparlos.
- Tener variantes y diferentes líneas de desarrollo (*branches*).
- Preserva la historia del proyecto (grafo de cambios).
- Evitar nombrado manual: `version_definitiva.1`, `version_redefinitiva_1.2...`
- Desarrollo en grupo (por ejemplo, en proyectos de software libre).

SCV

- Importante: agrupar y confirmar un conjunto de cambios (**commit**).
- Cada *commit* como una [transacción](#), de forma atómica.
- Puede haber conflictos (cambios incompatibles) → hay que resolverlos.

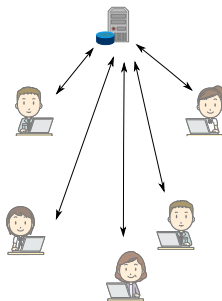
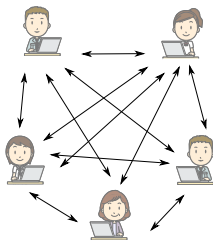
SCV distribuido

- Puedo tener varios árboles de ficheros (peers).
- Cada uno trabaja en local y luego sincronizan.
- Es un conjunto de bosques de árboles de ficheros en diferentes máquinas.

Distribuido

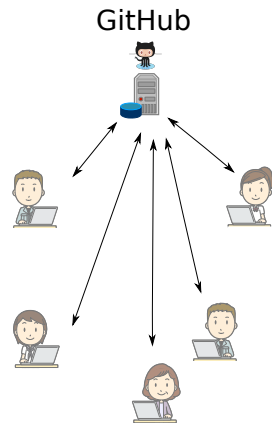
v.s.

Centralizado



SCV Distribuido

- Muchas veces es cómodo tener una copia centralizada (*remote repository*), manteniendo las copias locales.
- Por organización, simplicidad, disponibilidad, como salvaguarda, etc.
- Hay servicios de *hosting* de Git, como [GitHub](#) o [GitLab](#).



¿Qué no es Git?

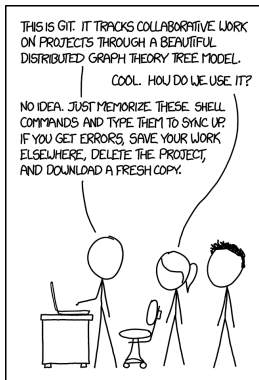
- Git no es un sistema general para hacer copias de seguridad (*backup*).
- Puede servir de *backup* para el fuente (aunque requiere intervención del usuario).
- Git no es bueno gestionando archivos binarios.

¿Qué cosas puedo hacer?

- Añadir y modificar (editar) un fichero o varios.
- Crear y mezclar una *branch* (rama o variante del código fuente) con y sin conflictos.
- Ver la historia/historial de cambios.
- Deshacer un conjunto de cambios confirmados (*commit*).
- Compartir, sincronizar el código con un repositorio remoto o central.

Cómo NO usar Git

Para evitar



Documentación

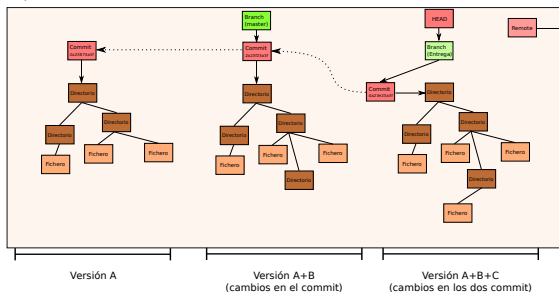
- <https://git-scm.com/book/en/v2>.
- <https://juristr.com/blog/2013/04/git-explained/>.
- <https://codewords.recurse.com/issues/two/git-from-the-inside-out>.
- Los subcomandos de git son `git XXX` entonces: `man git-XXX`.
- Por ejemplo para mirar como hacer `git clone` mirar: `man git-clone`.
- Donde XXX (`clone` en el ejemplo anterior) se suelen llamar verbos (*verbs* en la terminología de Git). Definen acciones que podemos realizar sobre el repositorio de código.

Terminología básica

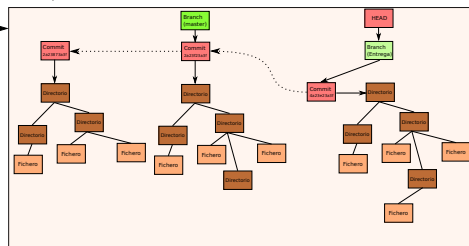
- **clone**: copia un repositorio diferente (puede ser remoto) a un sitio local
- **commit**: conjunto de cambios que **se aplican agrupados** al repositorio local.
- **fetch** o **pull**: traen cambios de un repositorio diferente (o remoto); **fetch** trae, **pull** hace además *merge*.
- **push**: manda cambios a un repositorio diferente, que puede ser remoto.
- **head**: referencia al **commit** sobre el que estamos trabajando.
- **branch** : etiqueta de una rama de código (línea separada de cambios).
- **main**: *branch* principal del repositorio (de trabajo o integración). También se usa **master** (se puede configurar).

Repositorio de ejemplo: vista del usuario

Repositorio



Otro repositorio



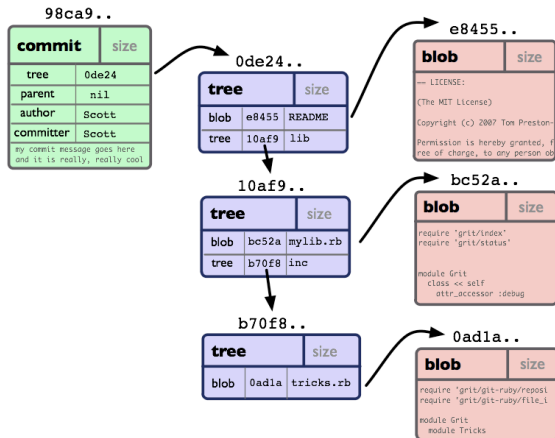
No están dibujadas las ramas remotas (las veremos más adelante).

Funcionamiento interno

- Ficheros y directorios se representan como objetos binarios (*blobs*) y árboles (*trees*).
- Se direccionan por *hash* de su contenido.
 - Un *hash* es una función matemática (en este caso SHA-1) que genera un código hexadecimal que resume el contenido.
- Un directorio se representa como un **tree**, una lista de SHAs con nombres de ficheros y directorios que contiene.
- Un fichero se registra como un **blob** de bytes a partir de su contenido.
- Se mantienen más metadatos, como HEAD o etiquetas de *branch*.
- Los metadatos de gestión están dentro del directorio `.git`
- Se maneja con comandos de alto nivel: `git clone`, `git pull`, etc.

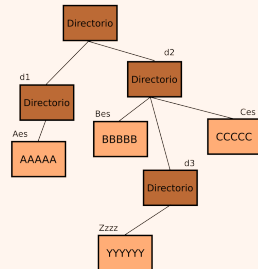
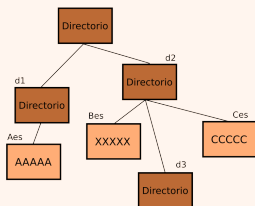
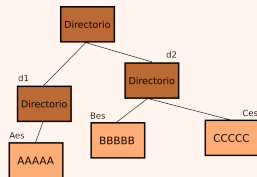
Representación interna de cambios

http://shafiulazam.com/gitbook/1_the_git_object_model.html



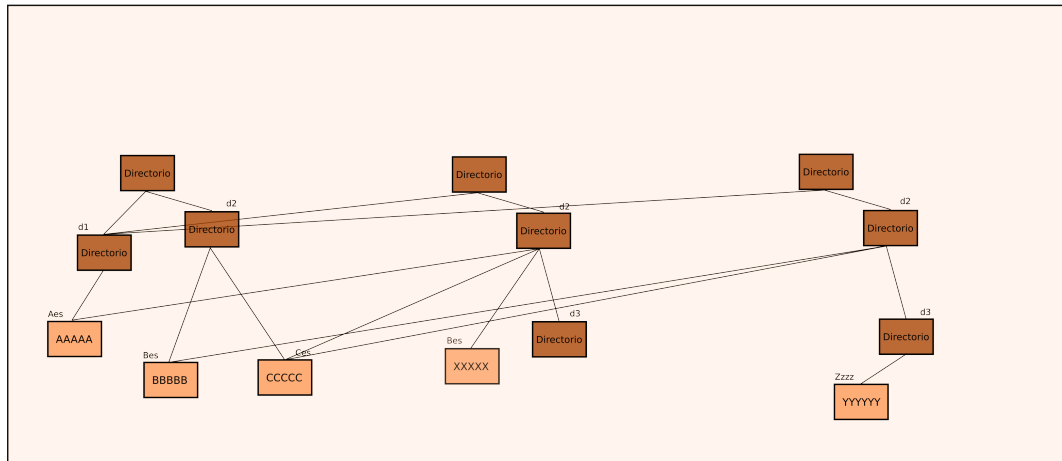
Árbol de cambios (vista interna)

Repositorio

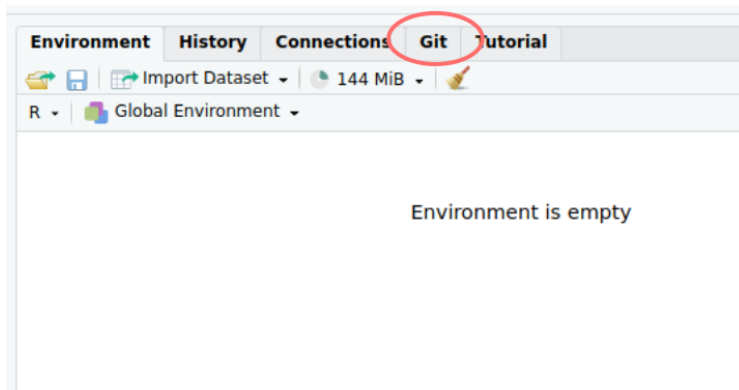


Árbol de cambios (vista interna)

Por dentro



Soporte para Git en RStudio



1.2.2 Configuración

Identificación de usuario

- El primer paso es configurar los datos básicos del autor de los cambios.
 - Se identifica por un nombre completo y un correo electrónico.
- El siguiente ejemplo configura tus datos de identificación para todos los repositorios Git del usuario en la máquina.

```
$ git config --global user.name "Felipe Ortega"
```

```
$ git config --global user.email "bigdatalab@felipeortega.net"
```

Identificación de usuario

- Hay tres posibles niveles de configuración:
 - A nivel de **proyecto**. Se almacena en `.git/config`, dentro del directorio del proyecto.
`git config user.name "John Doe"`
 - A nivel **global** (todos los proyectos de un usuario). Se almacena en el archivo `$HOME/.gitconfig`.
`git config --global user.name "John Doe"`
 - A nivel **system** (todos los proyectos de esta máquina). Se almacena en el archivo `/etc/gitconfig`.
`git config --system user.name "John Doe"`
 - Más información: `man git-config`

Alias de comandos

- Podemos definir comandos más cortos que faciliten el trabajo

```
$ git config --global alias.adog "log --all --decorate --oneline --graph"
$ git config --global alias.alog "log --all --decorate --oneline --graph "\
"--date=short '--format=%C(yellow)%h %C(red)%ae%C(reset) %ad %s'"
$ git config --global alias.aLog "log --all --decorate --oneline --graph "\
"--date=short '--format=%h %ae %ad %s'"
```

Otras configuraciones

- Podemos configurar el editor de texto.

```
$ git config --global core.editor vim
```

- Podemos definir la herramienta para mezclar cambios (*merge*).

```
$ git config --global merge.tool emerge
```

- Podemos definir el nombre de la rama principal. Por defecto es master, pero otras alternativas son main, trunk y development.

```
$ git config --global init.defaultBranch <nombre>
```

1.2.3 Registro de cambios

Inicialización de un repositorio

- Git guarda toda la información sobre su estado en el subdirectorio `.git` dentro del directorio raíz de nuestro proyecto.
- Ahí está todo: HEAD, los árboles, etc.
- El directorio donde se encuentra la carpeta `.git` es nuestro directorio de trabajo.
- Para poner un directorio bajo control de Git, creando la carpeta `.git` y todo lo necesario:

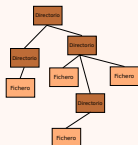
```
$ git init gitrepo
```

```
Inicializado repositorio Git vacío en /home/jfelipe/LSIS-class/gitrepo/.git/
```

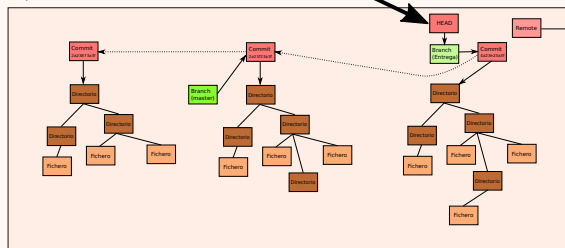
Áreas y estados

Repositorio

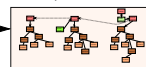
Working directory (copia de trabajo)



Repositorio



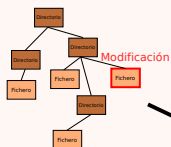
Otro repositorio



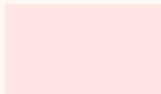
Modifico fichero(s): **git add** ficheros; editor ficheros

Repositorio

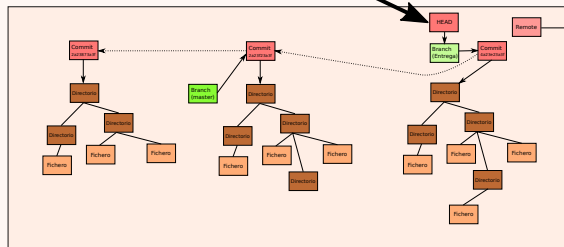
Working directory (copia de trabajo)



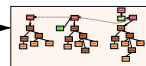
Staging area



Repositorio



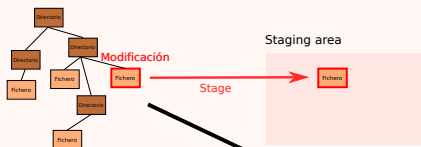
Otro repositorio



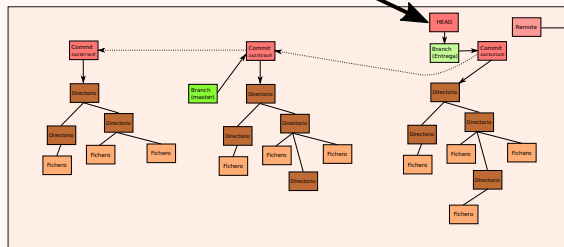
Paso a stage: **git add** ficheros

Repositorio

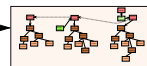
Working directory (copia de trabajo)



Repositorio



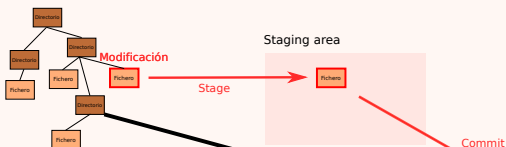
Otro repositorio



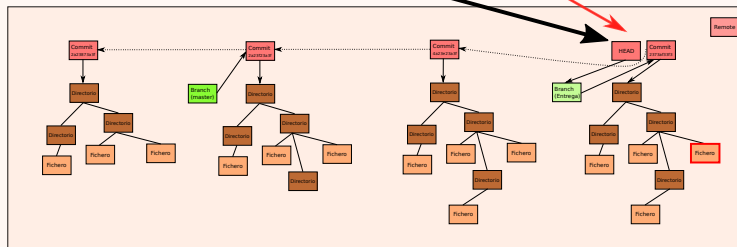
Hago commit al repo local: **git commit**

Repositorio

Working directory (copia de trabajo)



Repositorio



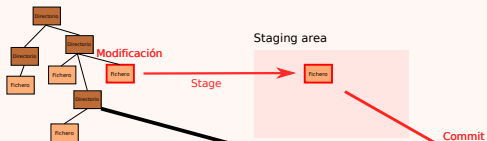
Otro repositorio



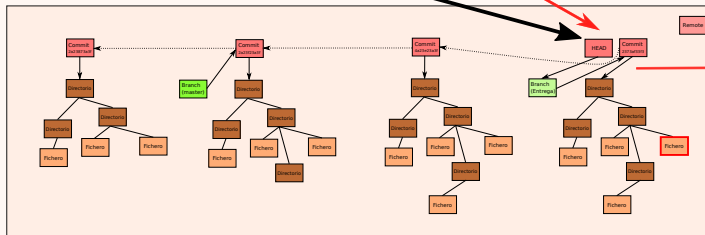
Hago push al repo remoto: **git push**

Repositorio

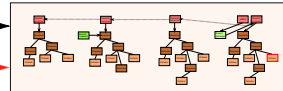
Working directory (copia de trabajo)



Repositorio



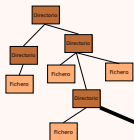
Otro repositorio



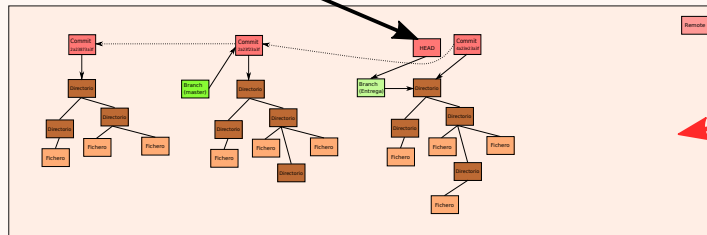
Hago pull para traer cambios del repo remoto: **git pull**

Repositorio

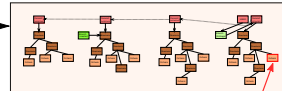
Working directory (copia de trabajo)



Repositorio



Otro repositorio



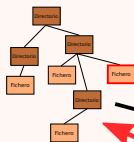
Modificado

Pull

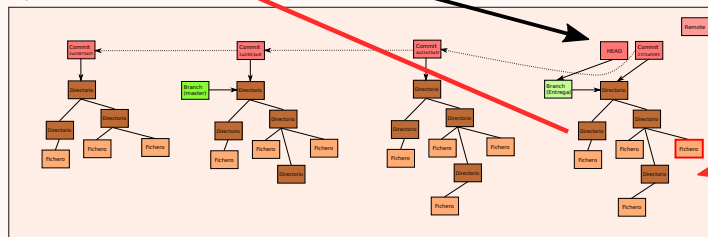
Hago pull para traer cambios del repo remoto

Repositorio

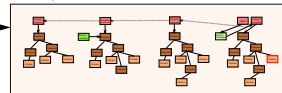
Working directory (copia de trabajo)



Repositorio



Otro repositorio

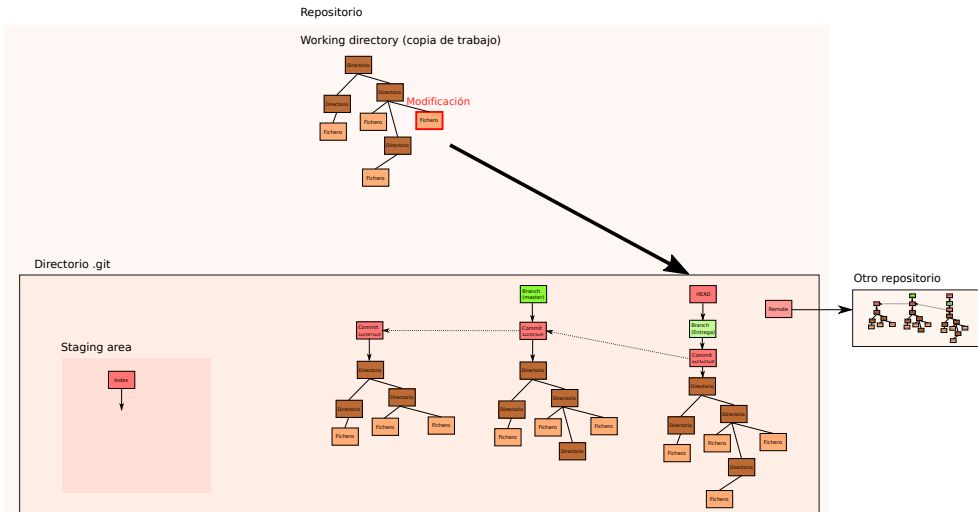


Pull

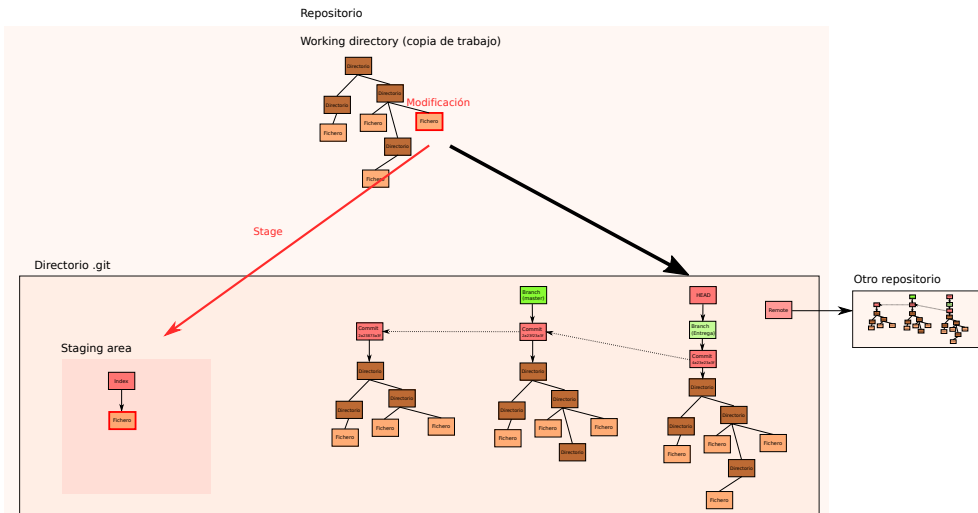
En realidad

- *Staging area* es parte del repositorio, pero no permanente (en `.git`).
- También se llama índice o caché.
- Hay un fichero `index` con los ficheros a añadir.
- Los blobs están en el repo, pero no forman parte de un *commit* (temporal).
- *Commit* los hace parte de un *commit*.
- El *staging* lo deja todo preparado (reserva todos los recursos).

Stage area (realidad)



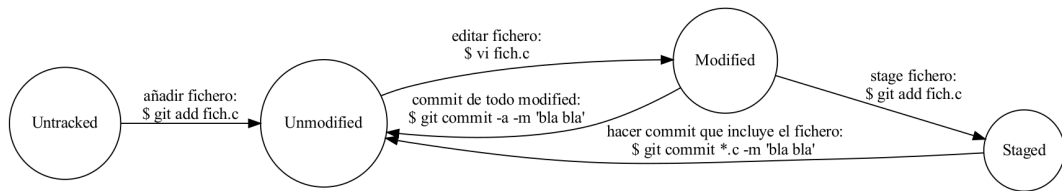
Stage area (realidad)



Estados de un fichero

- En mi directorio de trabajo
- *untracked, unmodified, modified, staged*
- Si Git no lo sigue: *untracked*
- Si Git controla sus cambios:
 - *unmodified*: está en el último *commit*, al día
 - *modified*: ha cambiado desde el último *commit*
 - *staged*: listo para hacer *commit*

Estados de un fichero



Borrar un fichero

- Los ficheros se borran del repositorio con `git rm fichero`.
- Hace que no estén en el siguiente *commit*.

1.2.4 Sesión básica

Sesión

- Creo un repositorio

```
$ mkdir myproject
```

```
$ cd myproject
```

```
$ git init
```

```
Initialized empty Git repository in /home/paurea/myproject/.git/
```

Sesión

- Añado un fichero

```
$ touch x.c
```

```
$ git add x.c
```

```
$ git status
```

```
On branch master
```

```
No commits yet
```

```
Changes to be committed:
```

```
(use "git rm --cached <file>..." to unstage)
```

```
new file:   x.c
```

Sesión

- Añado un fichero, paso a untracked

```
$ git commit -m 'creo el primer fichero'
```

```
[master (root-commit) 9657128] creo el primer fichero
```

```
1 file changed, 0 insertions(+), 0 deletions(-)
```

```
create mode 100644 x.c
```

```
$ git status
```

```
On branch master
```

```
nothing to commit, working tree clean
```

Sesión

- Paso a tracked
- No dice nada porque no hay cambios

```
$ git add x.c
```

```
$ git status
```

On branch master

Your branch is up to date with 'origin/master'.

nothing to commit, working tree clean

Sesión

- Modifico, paso a modified

```
$ vi x.c
```

```
$ git status
```

On branch master

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: x.c

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

Sesión

- Paso a staged

```
$ git add x.c
```

```
$ git status
```

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

modified: x.c

```
$ git commit -m 'modifico el primer fichero'
```

```
[master 540b592] modifico el primer fichero
```

```
1 file changed, 2 insertions(+)
```

```
$ git status
```

On branch master

nothing to commit, working tree clean

Sesión

- Paso a staged

```
$ git log
```

```
commit 540b592ecd4229ff59c0c1f85cc0c68c3fd843e4 (HEAD -> master)
```

```
Author: paurea <paurea@gmail.com>
```

```
Date: Thu Nov 15 13:25:20 2018 +0100
```

```
modifico el primer fichero
```

```
commit 9657128cccb82061afa8d626af9f9eee8f69a3bd
```

```
Author: paurea <paurea@gmail.com>
```

```
Date: Thu Nov 15 13:24:37 2018 +0100
```

```
creo el primer fichero
```

Fichero .gitignore

- Fichero .gitignore en el raíz del proyecto
- # **Sólo al principio**, comenta la línea
- Ficheros que git no considera (ni siquiera para untracked)
- Usa globbing, dos tipos de patrones, path, glob en general
- Si no contiene / (salvo al final) es un patrón de globbing en cualquier sitio
- El raíz es el del proyecto (/ es el directorio principal)
 - Los directorios (si queremos sólo dir) se indican con / al final
 - (! hace que el patrón se deje de ignorar i.e. lo niega)
 - Línea en blanco no hace nada
 - La barra invertida \ escapa
 - Si contiene / es un path desde el raíz (con globbing)
 - ** encaja con cualquier subpath, incluyendo las /

Fichero .gitignore

```
# una línea de comentario
# ignorar los ficheros que acaben en .a
*.a
# no ignorar lib.a, a pesar de la regla anterior
!lib.a
# ignorar sólo el fichero /TODO en el raíz
/TODO
# ignorar todo en build/ en cualquier sitio
build/
# ignorar doc/n.txt, pero no doc/server/a.txt
doc/*.txt
# ignorar, encaja con a/x/b y con a/x/c/d/b
a/**/b
# ignorar el contenido de bla y subdirectorios
bla/**
# ignorar, encaja con /a/d/z/b y con /e/z/b
**/z/b
```

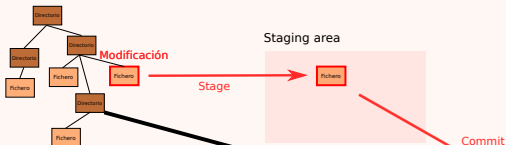
git diff

- Antes de aplicar cambios
- Tres tipos
 - Entre work y stage (`git diff`)
 - Entre stage y repo (`git diff --staged`)
 - Entre dos objetos (`git diff sha1 sha2`, ahora no lo vamos a ver)

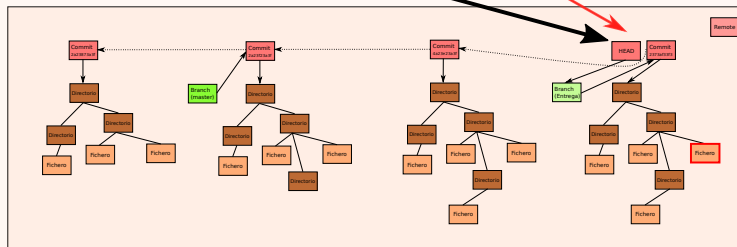
Viendo en un repositorio...

Repositorio

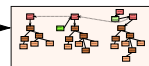
Working directory (copia de trabajo)



Repositorio



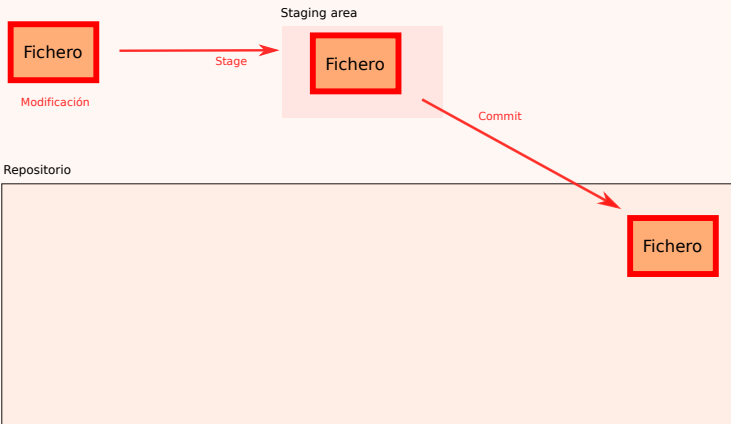
Otro repositorio



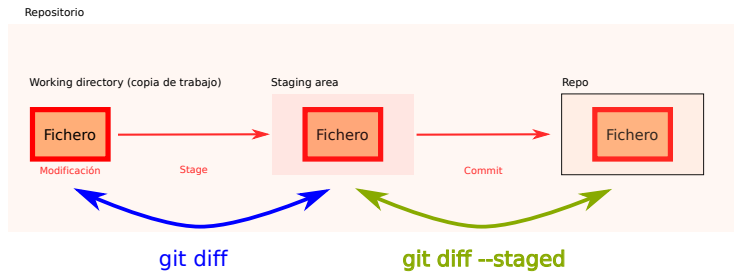
...lo que le pasa a un fichero

Repositorio

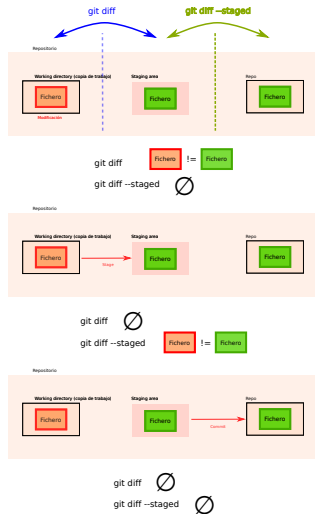
Working directory (copia de trabajo)



Podemos usar los dos diff



Resultado de los dos diff



Creo el fichero

```
$ echo aaa > file
```

```
$ git add file
```

```
$ git commit -m 'initial file'
```

```
[master f6647e8] initial file
```

```
2 files changed, 2 insertions(+), 1 deletion(-)
```

```
create mode 100644 file
```

```
$ git status
```

```
On branch master
```

```
nothing to commit, working tree clean
```

Modifico

```
$ echo bbb > file
```

```
$ git status
```

On branch master

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: file

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

Modifico

```
$ git diff
diff --git a/file b/file
index 72943a1..f761ec1 100644
--- a/file
+++ b/file
@@ -1,1 @@
-aaa
+bbb
$ git diff --staged
$
```

Stage

```
$ git add file
```

```
$ git status
```

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

modified: file

```
$ git diff
```

```
$ git diff --staged
```

```
diff --git a/file b/file
```

```
index 72943a1..f761ec1 100644
```

```
--- a/file
```

```
+++ b/file
```

```
@@ -1,1 @@
```

```
-aaa
```

```
+bbb
```

Commit

```
$ git commit -m 'changes'
```

```
[master f227358] changes
```

```
1 file changed, 1 insertion(+), 1 deletion(-)
```

```
$ git status
```

```
On branch master
```

```
nothing to commit, working tree clean
```

```
$ git diff file
```

```
$ git diff --staged file
```

1.2.5 Repositorios remotos

Servicios de repositorios remotos

- En la práctica, casi siempre vamos a utilizar un servicio centralizado ([GitHub](#), [GitLab](#)) como repositorio remoto.
- En caso de trabajar directamente en el repo remoto, lo hacemos a través de una interfaz web (solo para cambios muy sencillos o directos).
- El repo remoto sirve para publicar los cambios y que otras personas puedan hacer *pull*.
 - `git push origin main` # *Subir mis cambios a rama main del repo remoto*
- También permite hacer un *fork* de un repo, replicándolo en nuestra cuenta de usuario para hacer cambios en la copia a partir del mismo grafo.
- Ventaja *fork*: si hago *commits* en mi copia que arreglan cosas del repo original (*upstream*) puedo enviar la petición a sus gestores para que los integren → [pull request](#).

Creación de un repo en GitHub

- Primero tienes que *crear una cuenta de usuario/a*.
- Después de autenticarte, en tu cuenta, sigue las [instrucciones para crear un nuevo repo](#).
- También puedes añadir un proyecto Git local a un nuevo repo (vacío) recién creado en GitHub [[📺 Vídeo tutorial](#)].
 - Es muy recomendable añadir un README.md con información básica sobre el proyecto.
 - Recuerda que es importante elegir una licencia e incluir una descripción del proyecto.
 - También es recomendable añadir documentación. Puedes usar la *wiki* incorporada en el propio repo de GitHub o bien utilizar un servicio externo (por ejemplo [Read the Docs](#)).

Creación de un repo en GitLab

- Primero tienes que *crear una cuenta de usuario/a*.
 - En la EIF tenemos un GitLab para alumnos/as y profes:
<https://labs.eif.urjc.es/index.php/tutoriales/gitlab/>.
- Después de autenticarte, en tu cuenta, sigue las [instrucciones para crear un repo y empezar a trabajar con él](#).

Integración con IDEs

- La mayoría de IDEs para desarrollo de software ya incluyen *plug-ins* o soporte directo que permiten integrar un SCV en el trabajo con un proyecto.
 - Inicializar un nuevo repositorio local.
 - Configurar uno o varios repositorios remotos.
 - Operaciones con el repositorio: hacer *commits*, crear ramas (*branches*), integrar cambios (*merge*), comparar cambios (*diff*), etc.
- Las IDEs de JetBrains (PyCharm, IntelliJ IDEA, GoLand), VSCode y muchas otras incluyen estas características.

Lecturas sugeridas

Para saber más

- La referencia fundamental sobre Git accesible de forma pública es el libro [Pro Git](#), disponible en inglés y castellano [1].
- Los libros de la serie *Head First* de O'Reilly son muy conocidos por su nivel muy accesible y su enfoque didáctico. Para este tema, se ha publicado en enero de 2022 *Head First Git* [2].
- Otra referencia muy conocida de la misma editorial es [3].

Referencias I

- [1] S. Chacon y B. Straub. *Pro Git. The Expert's Voice*. Apress, 2014.
- [2] Raju Gandhi. *Head First Git. A Learner's Guide to Understand Git from the Inside Out*. Head First. O'Reilly Media, 2022.
- [3] P.K. Ponuthorai y J. Loeliger. *Version Control with Git*. 3ª ed. O'Reilly Media, 2022.