

# Puesta en Producción Segura

Unidad 0. Herramientas Necesarias.

Título: Sistemas de Control de  
Versiones: Git



*"Git no olvida... y los secretos que subes, tampoco."*

*"Un buen commit guarda tu trabajo; un buen .gitignore protege tu infraestructura."*

## — Cultura DevSecOps

# Objetivos

- Comprender la función de los Sistemas de Control de Versiones (SCV)
- Diferenciar entre SCV centralizados y distribuidos
- Conocer los conceptos clave: commit, branch, merge, etc.
- Entender el funcionamiento de Git y su papel en el desarrollo colaborativo
- Realizar operaciones básicas en Git de forma práctica
- Aplicar buenas prácticas y evitar riesgos comunes de seguridad en el uso de Git



# Contenidos

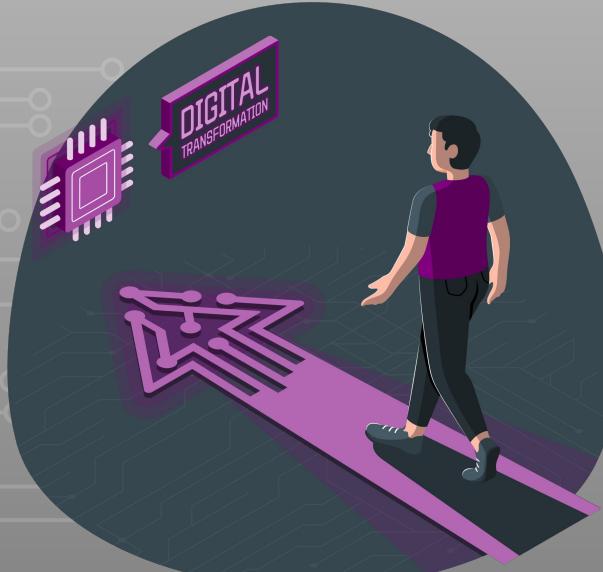
**01** Sistemas de control de versiones

**02** Tipos de VCS

**03** Conceptos básicos

**04** Git

**05** Operaciones básicas en Git



01

# Sistemas de Control de versiones



# ¿Qué es un SCV?

Un Sistema de Control de Versiones (SCV o VCS, por sus siglas en inglés) es una práctica de software que consiste en administrar el código por versiones, haciendo un seguimiento de las revisiones y del historial de cambios para facilitar la revisión y la recuperación del código.



Esta práctica suele implementarse con sistemas de control de versiones, como Git, que permite que varios desarrolladores colaboren para crear código.

# ¿En qué nos puede ayudar un SCV?

- Gestión de ficheros a lo largo del tiempo
  - Evolución del trabajo
- Gestión del versionado de los ficheros
  - Si un archivo se corrompe o hemos cometido un fallo volvemos atrás
  - Mecanismo para compartir ficheros
  - Habitualmente tenemos nuestro propio mecanismo y modelo de trabajo
    - Versionado: Documento.docx, Documento\_v2.docx
    - Herramientas: Dropbox, Adjunto correo.



# ¿Por que usar un SCV?

Las metodologías/mecanismos particulares no escalan para proyectos de desarrollo

Un SCV permite:

- Crear copias de seguridad y restaurarlas
- Sincronizar (mantener al día) a los desarrolladores respecto a la última versión de desarrollo
- Deshacer cambios
  - Tanto problemas puntuales, como problemas introducidos hace tiempo
- Gestionar la autoría del código
- Realizar pruebas (aisladas)
  - Simples o utilizando el mecanismo de branches/merges



# Problemas resueltos por un VCS

"He tocado algo y ahora no funciona"

Miriam



programa-  
proyecto-pot



```
cpu_usage.py
#!/usr/bin/env python3

import psutil

if(psutil.cpu_percent(1) > 80):
    print("Uso de CPU excesivo")
else:
    print("Todo OK")
```

Fuente: Taller de Git y GitHub desde cero de Iván Martínez Ortiz

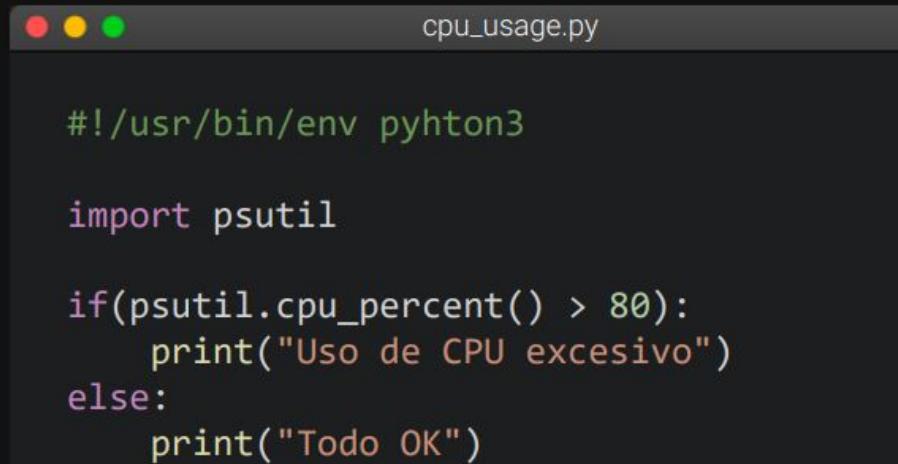
# Problemas resueltos por un VCS

*"He tocado algo y ahora no funciona"*

Miriam



programa-  
proyecto-pot



```
cpu_usage.py

#!/usr/bin/env python3

import psutil

if(psutil.cpu_percent() > 80):
    print("Uso de CPU excesivo")
else:
    print("Todo OK")
```

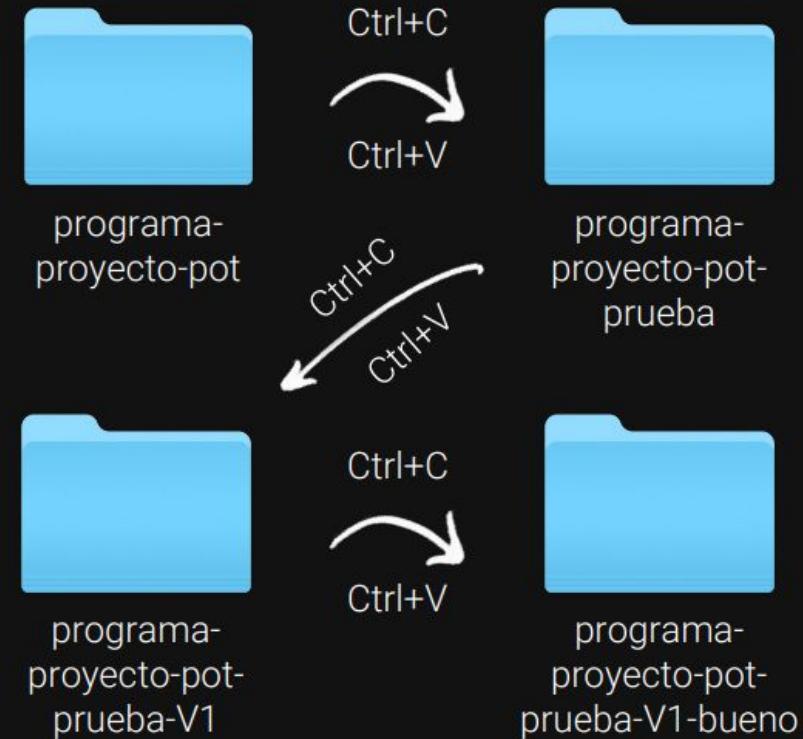
Fuente: Taller de Git y GitHub desde cero de Iván Martínez Ortíz

# Problemas resueltos por un VCS

"Eeh... ¿Cuál era el bueno?"

"Me he quedado sin espacio. ¿Qué elimino?"

Olga



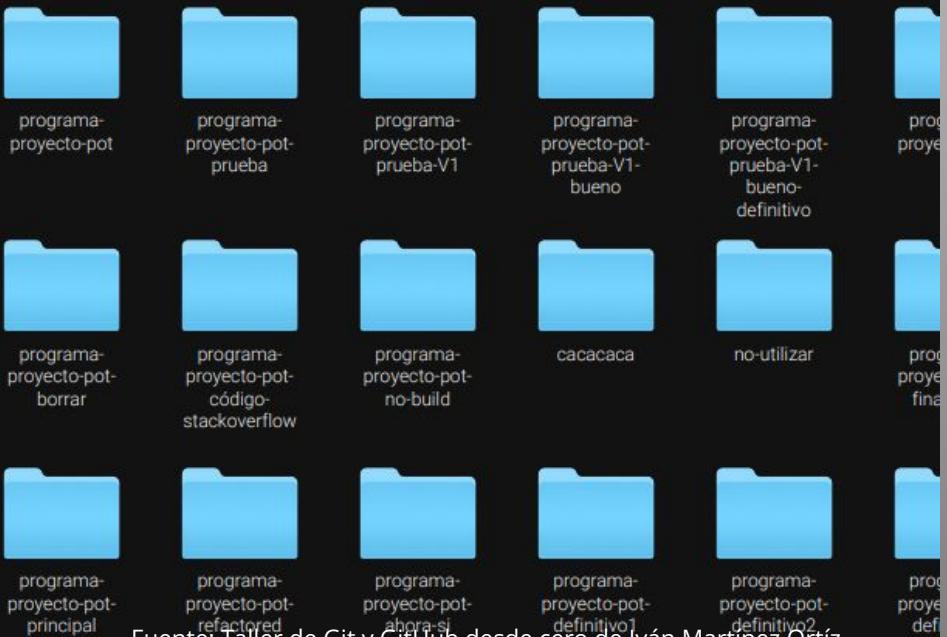
Fuente: Taller de Git y GitHub desde cero de Iván Martínez Ortíz

# Problemas resueltos por un VCS

**2.32 GB**

*"Eeh... ¿Cuál era el bueno?  
"Me he quedado sin espacio. ¿Qué elimino?"*

**Olga**



Fuente: Taller de Git y GitHub desde cero de Iván Martínez Ortíz

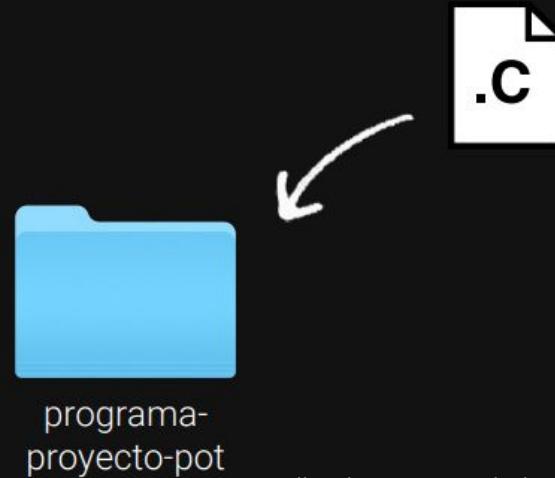
# Problemas resueltos por un VCS

"Fran no hace ni el huevo."

"Es que hasta que no acabes no puedo poner mi parte."



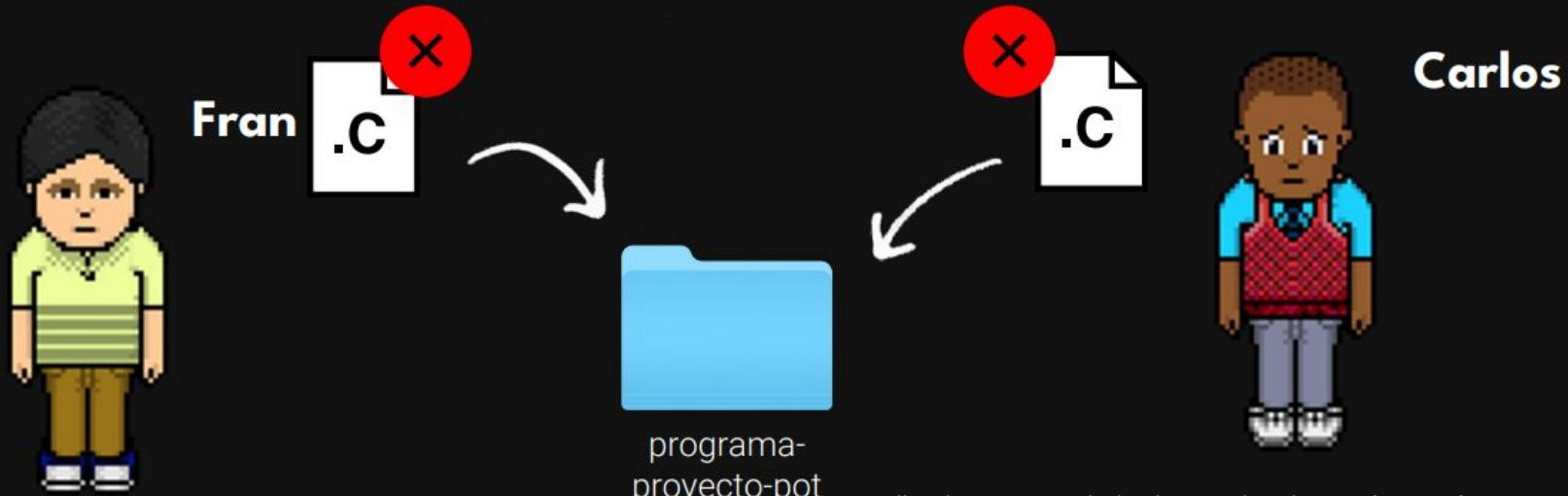
Fran



Carlos

# Problemas resueltos por un VCS

"Al unirlo todo, la hemos liado..."



Fuente: Taller de Git y GitHub desde cero de Iván Martínez Ortiz

# Tracking de diferencias

Versión anterior



```
#!/usr/bin/env python3

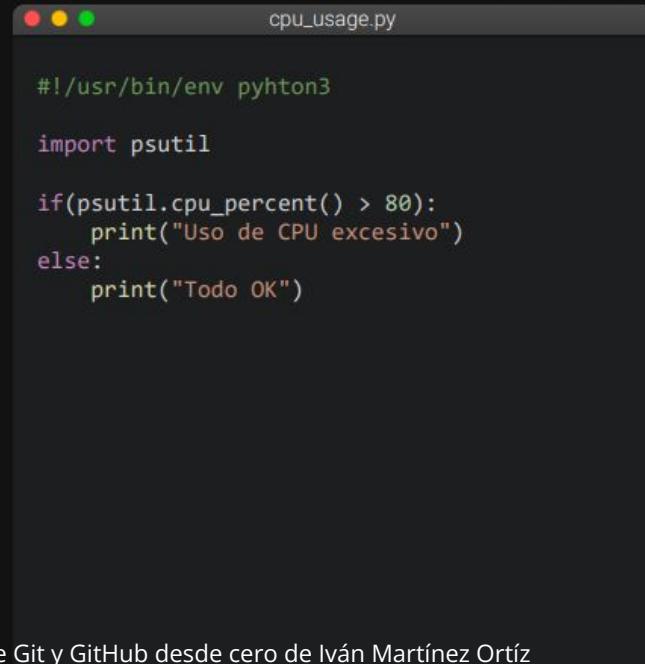
import psutil

if(psutil.cpu_percent(1) > 80):
    print("Uso de CPU excesivo")
else:
    print("Todo OK")
```

Miriam



Versión actual



```
#!/usr/bin/env python3

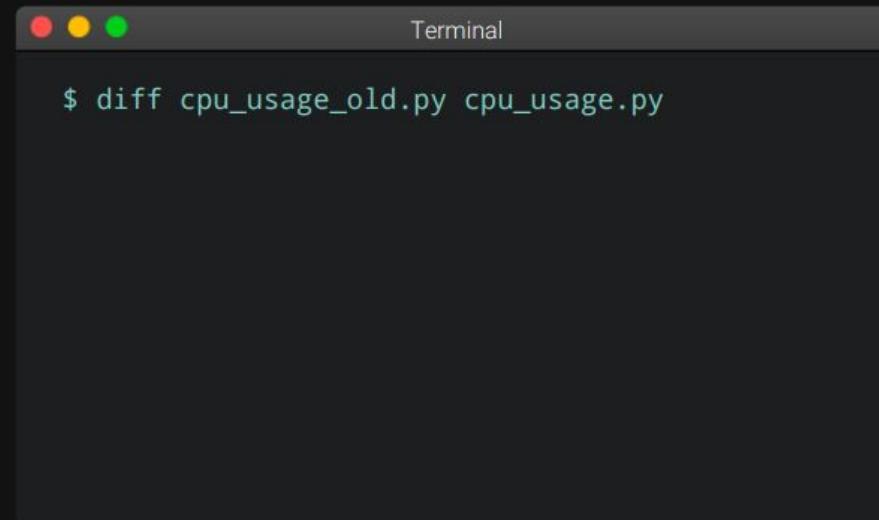
import psutil

if(psutil.cpu_percent() > 80):
    print("Uso de CPU excesivo")
else:
    print("Todo OK")
```

Fuente: Taller de Git y GitHub desde cero de Iván Martínez Ortíz

# Tracking de diferencias

Miriam



A terminal window titled "Terminal" with three red, yellow, and green buttons at the top. The window contains the command "\$ diff cpu\_usage\_old.py cpu\_usage.py".

Fuente: Taller de Git y GitHub desde cero de Iván Martínez Ortíz



02

# Tipos de VCS

# VCS según Arquitectura de almacenamiento de los datos

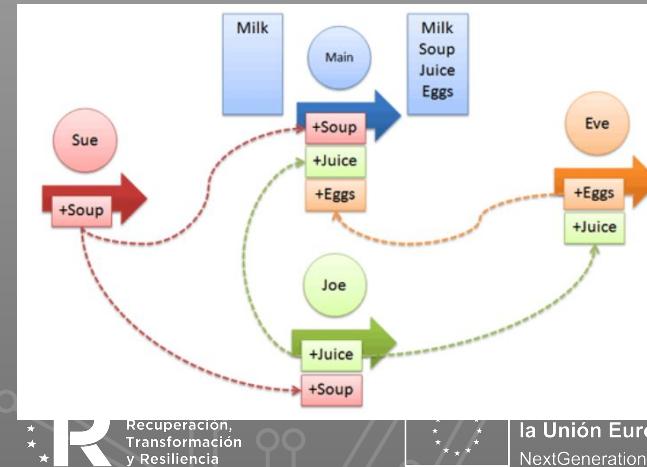


**Según la arquitectura de almacenamiento de datos, podemos encontrarnos Sistemas de Control de versiones:**

- **Distribuidos**
- **Centralizados**

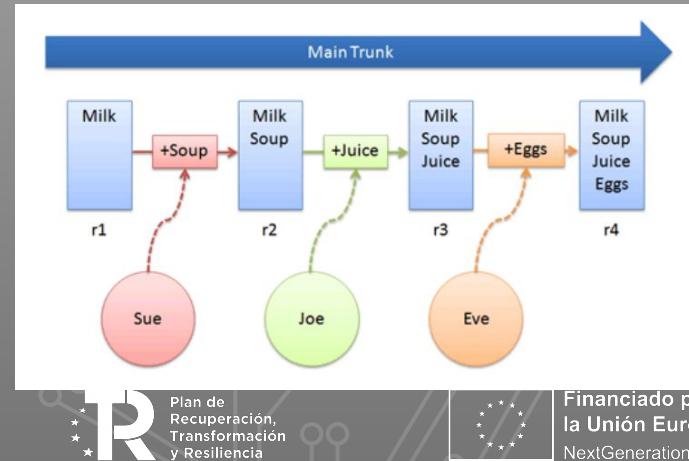
# VCS según Arquitectura de almacenamiento de los datos

**Distribuidos:** cada usuario tiene su propio repositorio. Los distintos repositorios pueden intercambiar y mezclar revisiones entre ellos. Es frecuente el uso de un repositorio, que está normalmente disponible, que sirve de punto de sincronización de los distintos repositorios locales.  
Ejemplos: Git y Mercurial.



# VCS según Arquitectura de almacenamiento de los datos

**Centralizados:** existe un repositorio centralizado de todo el código, del cual es responsable un único usuario (o conjunto de ellos). Se facilitan las tareas administrativas a cambio de reducir flexibilidad, pues todas las decisiones fuertes (como crear una nueva rama) necesitan la aprobación del responsable. Algunos ejemplos son CVS, Subversion o Team Foundation Server.



# Según la forma de colaborar

**De forma exclusiva:** en este esquema para poder realizar un cambio es necesario comunicar al repositorio el elemento que se desea modificar y el sistema se encargará de impedir que otro usuario pueda modificar dicho elemento. Una vez hecha la modificación, esta se comparte con el resto de colaboradores. Si se ha terminado de modificar un elemento entonces se libera ese elemento para que otros lo puedan modificar. P.e. SourceSafe, SubVersión lo incorpora.

**De forma colaborativa:** en este esquema cada usuario modifica la copia local y cuando el usuario decide compartir los cambios el sistema automáticamente intenta combinar las diversas modificaciones. El principal problema es la posible aparición de conflictos que deban ser solucionados manualmente o las posibles inconsistencias que surjan al modificar el mismo fichero por varias personas no coordinadas. además, esta semántica no es apropiada para ficheros binarios.P.e. subversion o Git permiten implementar este modo de funcionamiento.

# Principales herramientas CVS

Algunas de las herramientas CVS son , Subversion, Git, Mercurial, Perforce, Team Foundation Server, Clearcase, BitBucket, etc..





03

# Conceptos Básicos



# Conceptos básicos: Elementos básicos



## - Repositorio

- Almacén de que guarda toda la información del proyecto.
- Habitualmente tiene estructura de árbol.

## - Servidor

- Máquina donde está alojado el Repositorio.

## - Working Copy/Working Set (Copia de trabajo)

- Copia local donde el desarrollador trabaja.

## - Trunk/Main/master (Rama principal):

- Localización dentro del repositorio que contiene la rama principal de desarrollo.

# Conceptos básicos: Operaciones básicas

- **Add**  
Añade un archivo para que sea rastreado por el SCV.
- **Revisión**  
Versión de un archivo/directorio dentro del SCV
- **Head**  
Última versión del repositorio (completo o de una rama)
- **Check out**  
Creación de una copia de trabajo que rastrea un repositorio
- **Check in / Commits**  
Envío de cambios locales al repositorio  
Como resultado cambia la versión del archivo(s)/repositorio



# Conceptos básicos: Operaciones básicas

- **Mensaje de Check in/log**

Todo Check in tiene asociado un mensaje que describe la finalidad del cambio



Puede estar asociado al un sistema de gestión de incidencias

- **Log (Historia)**

Permite visualizar/revisar la lista de cambios de un archivo/repositorio

- **Update/Syncronize/fetch&pull (Actualizar)**

Sincroniza la copia de trabajo con la última versión que existe en el repositorio.

- **Revert/Reset (Deshacer)**

Permite deshacer los cambios realizados en la copia de trabajo y dejar el archivo/recurso en el último estado conocido del repositorio.

# Conceptos básicos: Operaciones avanzadas

- **Branching** (ramas)

Permite crear una copia de un archivo/carpeta rastreada

Permite desarrollar en paralelo en otra “rama” pero dejando constancia de la relación que existe con la rama original.

- **Diff/Change/Delta/** (Cambio)

Permite encontrar las diferencias entre dos versiones del repositorio.

Se puede generar un parche que permitiría pasar de una versión a otra.

- **Merge/Patch**

Aplica los cambios de un archivo a otro

Utilizado habitualmente para mezclar branches

- **Conflict** (Conflicto)

Problema que surge cuando varios desarrolladores modifican el mismo recurso y los cambios se solapan.



# Política de control de versiones



Para aprovechar los SCV es necesario

- Establecer una política para el control de versiones para los proyectos:
  - Estructura del repositorio
  - Política para la rama principal
- Documentar el desarrollo
- Utilizando alguna herramienta de gestión de seguimiento: Trac
- Es interesante adoptar un modelo de trabajo que sea adecuado para el equipo de desarrollo
- El modelo de trabajo del equipo de desarrollo puede influir en la elección del SCV a utilizar.

# Metodología básica de trabajo

## La primera vez

1. Creación del repositorio del proyecto (Opcional)
2. Importación inicial del código del proyecto (Opcional)
3. Crear una copia de trabajo del repositorio
4. Modificar la copia de trabajo
5. Envío de cambios al repositorio



## Siguientes ocasiones

1. Actualizar el repositorio
2. Modificar la copia de trabajo
3. Envío de cambios al repositorio



JUNTA DE EXTREMADURA



04

# Git



Financiado por  
la Unión Europea  
NextGenerationEU



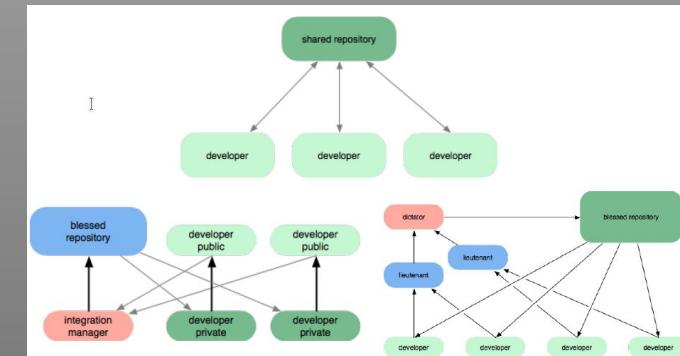
# ¿Por qué Git?

- Todo es local
  - Operaciones más rápidas
  - Puedes trabajar sin red
  - Todos los repositorios de los desarrolladores son iguales
  - En caso de emergencia puede servir de backup
- Git es rápido
  - Comparado con otras herramientas
- Git es pequeño
  - Pese a que es una copia de todo el repositorio
  - En algunos casos incluso comparándolo con svn
    - Si dos archivos son iguales sólo se guarda el contenido de 1.
    - El contenido de los archivos se guarda comprimido
    - Periódicamente se compactan los archivos
    - Se generan deltas entre las diferentes versiones de los archivos.



# ¿Por qué Git?

- Es distribuido
  - Todos los desarrolladores tienen una copia completa del repositorio
    - Pueden ser usadas como backups de emergencia
- – No es (demasiado) lento comparado con SVN
  - Teniendo en cuenta que con SVN sólo trabajamos con una rama a la vez.
- Permite múltiples flujos de trabajo
- Git es el nuevo estándar
  - En una gran cantidad de proyectos Open Source: Android, Apache (algunos), Debian, Drupal, ....



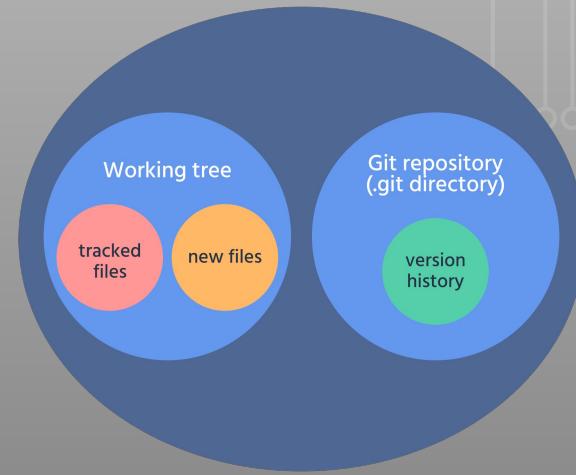
# Creando un proyecto en Github.com

- El comando git no permite crear un proyecto en [github.com](https://docs.github.com/en/get-started/importing-your-projects-to-github/importing-source-code-to-github/adding-locally-hosted-code-to-github)  
<https://docs.github.com/en/get-started/importing-your-projects-to-github/importing-source-code-to-github/adding-locally-hosted-code-to-github>
- Alternativas  
Usar el modo gráfico de [github.com](#)  
Github.com tiene su propia interfaz de línea de comandos llamada "gh" (sudo snap install gh)  
gh repo create



# Funcionamiento de Git

- Git funciona con repositorios **locales** y **remotos**
- Dentro del equipo local, trabajamos en un directorio (**working directory**)
- Dentro de dicho directorio de trabajo, no todos los ficheros están siendo gestionados (**tracked**) por Git, sólo aquellos que se meten en el escenario (**stage**)

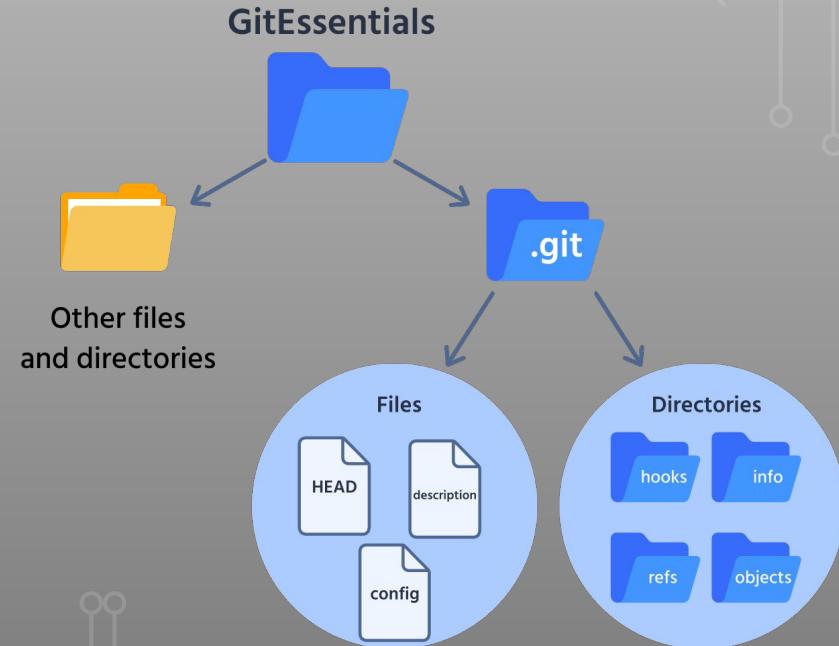


Fuente de las imágenes:  
<https://codefinity.com/courses/git-essentials>

# Directorios y archivos en repositorio local

En nuestro **Working directory** nos encontraremos:

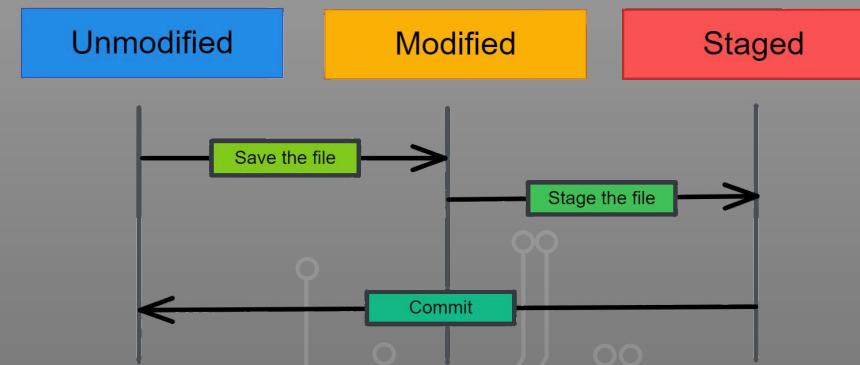
- Los directorio y archivos que creamos y añadimos al repositorio.
- Directorio .git con archivos y metadatos de todo el repositorio.



Fuente de las imágenes:  
<https://codefinity.com/courses/git-essentials>

# Confirmaciones

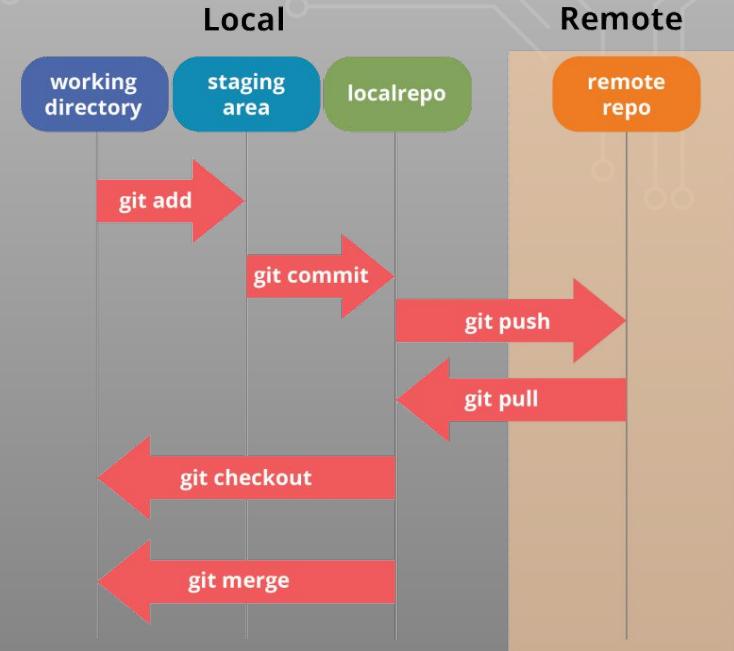
Hay tres estados de archivo en Git: **modificado**, en **staging** y **confirmado**. Cuando realizas cambios en un archivo, los cambios se guardan en el directorio local. No forman parte del historial de desarrollo de Git. Para crear una confirmación, primero tienes que poner en staging los archivos cambiados. Puedes añadir o eliminar cambios en el área de staging y luego empaquetar estos cambios como confirmación con un mensaje que describa los cambios.



# Repositorio local y remoto

Una vez metidos los ficheros en el escenario, se van haciendo versiones en un repositorio local, y, si se desea, en el remoto.

- Commit para crear una versión en local.
- Push para crear una versión en remoto.



Fuente de la imagen:  
<https://www.edureka.co/blog/git-tutorial/>

# commits

Un commit es una imagen de los cambios hechos al proyecto en un cierto momento.

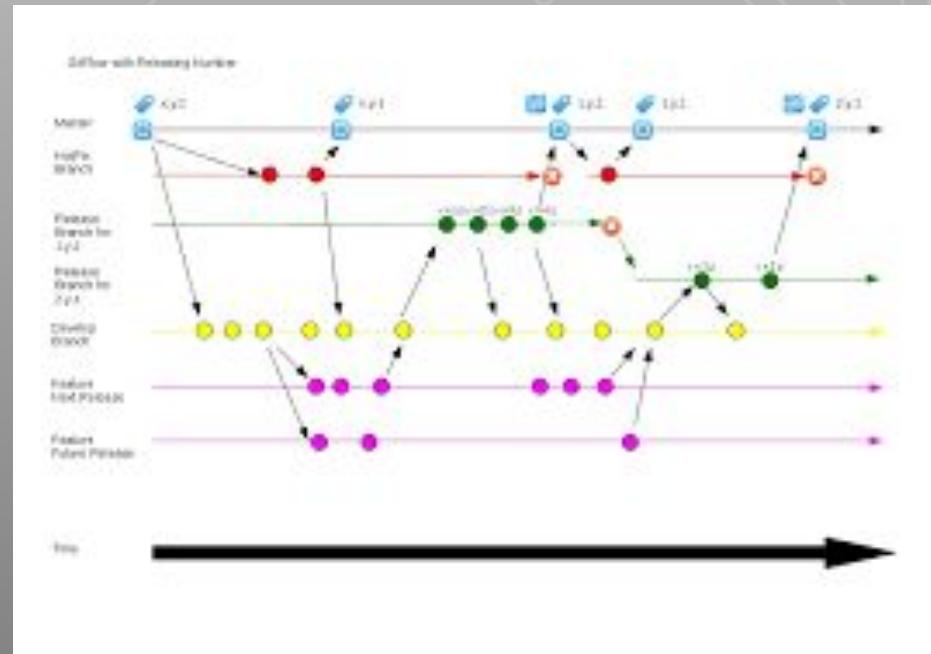
Estas copias no se modificarán a menos que se haga explícitamente.



Fuente de la imagen:  
<https://www.edureka.co/blog/git-tutorial/>

# Branchs

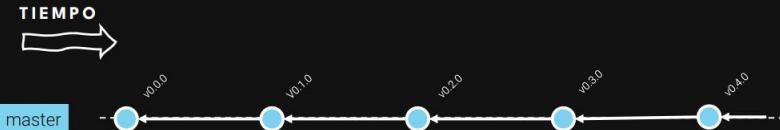
En un momento dado, se pueden crear distintas **ramas (branch)** de un commit, y más tarde se pueden mezclar (**merge**) o colocar una tras otra (**rebase**).



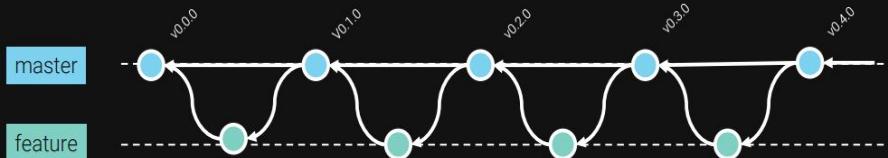
fuente de la imagen: <https://www.flickr.com/photos/miguelpd/>

# Flujos de git

## Master-only flow



## GitHub flow



## Git flow





05

# Operaciones básicas en Git



# Comandos principales

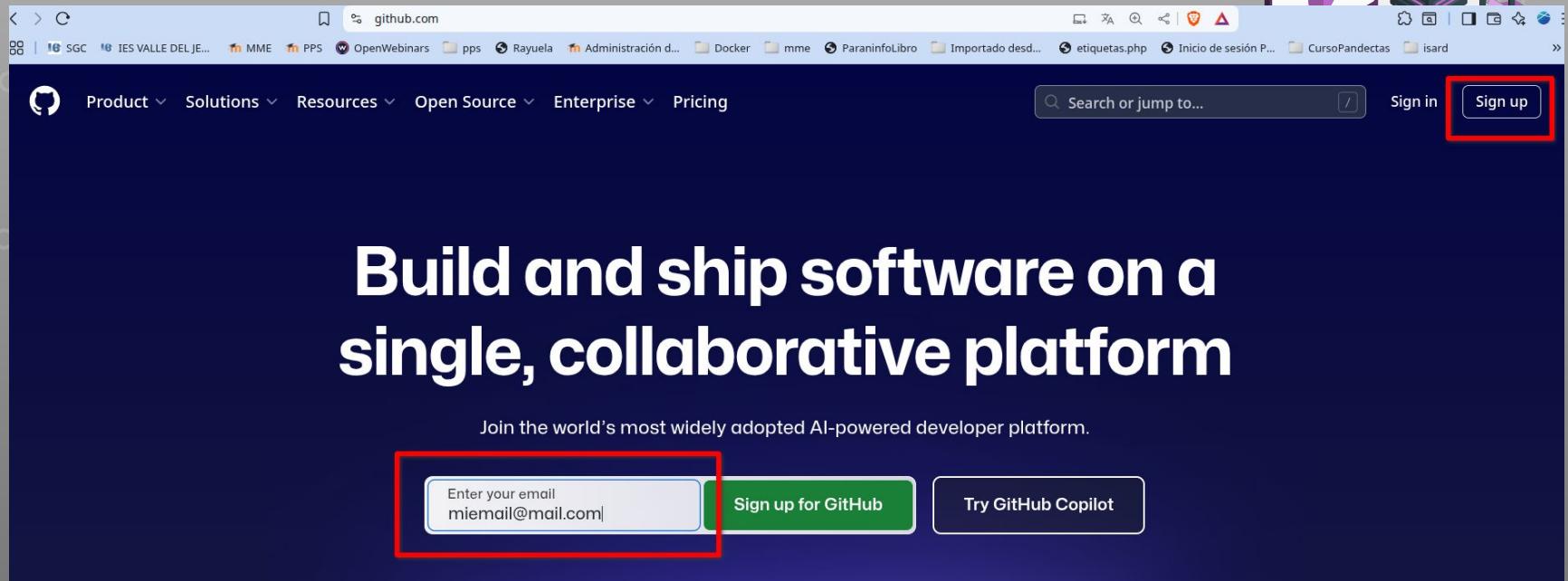
- init
- clone
- add
- status
- commit
- log
- push
- pull
- checkout
- merge

- init
- clone
- add
- status
- commit
- log
- push
- pull
- checkout
- merge



# Crear cuenta en github

Acceder a github y crear cuenta



The screenshot shows the GitHub homepage with a dark blue background. At the top, there is a navigation bar with links for Product, Solutions, Resources, Open Source, Enterprise, and Pricing. To the right of the navigation bar is a search bar labeled "Search or jump to..." and a "Sign in" button. A prominent red box highlights the "Sign up" button on the far right of the header. Below the header, the main slogan "Build and ship software on a single, collaborative platform" is displayed in large white text. Further down, a sub-slogan reads "Join the world's most widely adopted AI-powered developer platform." At the bottom of the page, there is a form with a red border containing an input field for "Enter your email" with the placeholder "miemail@mail.com" and a green "Sign up for GitHub" button. To the right of this form are two other buttons: "Try GitHub Copilot" and another "Sign up for GitHub" button.

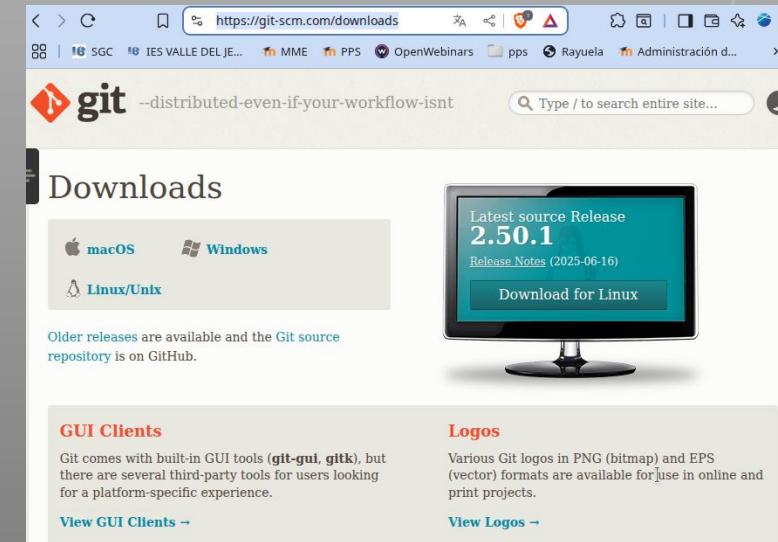
# Crear cuenta en github

Descargar la aplicación de cliente  
desde <https://git-scm.com/downloads>

En linux:

```
# apt-get install  
git
```

Después ya podremos hacer las  
operaciones desde el **terminal de  
Bash o Windows PowerShell**



# git init <repositorio>

Inicializa/crea un repositorio git.

- git init  
/home/usuario/proyecto

Terminal



```
$ git init /home/usuario/proyecto  
Initialized empty Git repository in *
```

# Configurar cuenta git

**git config:** permite configurar algunas variables de entorno de git

- git config --list
- git config --global user.name "usuario"
- git config --global user.email "ejemplo@test.com"
- git config --global core.editor emacs



# Añadir claves ssh o token

Para realizar operaciones con nuestro repositorio remoto necesitamos mecanismos de seguridad:

Claves SSH:

<https://docs.github.com/es/authentication/connecting-to-github-with-ssh/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent>

Tokens:

<https://docs.github.com/es/authentication/keeping-your-account-and-data-secure/managing-your-personal-access-tokens>

# git clone <url>

Descarga un repositorio git en un servidor remoto. Por defecto, el nombre de ese remoto es '**origin**'.

- git clone  
<https://github.com/jmmedinac03vjp/PuestaProducionSegura.git>

Terminal



```
$ git clone <url>
```

Cloning into "..."

```
remote: Enumerating objects: 101, done. remote:  
Counting objects: 100% (101/101), done. remote:  
Compressing objects: 100% (75/75), done.  
remote: Total 101 (delta 37), reused 78 (delta 18),  
pack-reused 0 Receiving objects: 100% (101/101),  
2.60 MiB | 11.43 MiB/s,  
done.  
Resolving deltas: 100% (37/37),  
done.
```



# git add [-A | file]

Añade uno o más archivos al área de stage para, posteriormente, poder registrar sus cambios.

- git add fichero
- git add \*
- git add -A

Terminal



```
$ git add <file>
```

# git rm [file]

Añade uno o más archivos al área de stage para, posteriormente, poder registrar sus cambios.

- git rm Clase.java

Terminal



```
$ git rm <file>  
rm <file>git
```

# git status

Indica los archivos que han sido modificados y cuyos cambios se han incorporado en el área de stage o no.

- git status

Terminal



```
$ git status
```

```
On branch master
```

```
No commits yet
```

```
Changes to be committed:
```

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

```
new file: README.md
```

```
Untracked files:
```

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

```
LICENSE
```

# git commit -m <mensaje>

Crea un commit y con -a  
añade  
ficheros nuevos y  
modificados  
- git commit -m "Primer  
commit"  
- git commit -a

Terminal



```
$ git commit [-a] [-m <mensaje>]  
[master (root-commit) cb47011] Añadida  
información del proyecto 1 file changed, 0  
insertions(+), 0 deletions(-)  
create mode 100644 README.md
```

# git diff

muestra las diferencias entre dos grupos de ficheros

- los actuales y los del entorno:
  - git diff
  - los del entorno y los que se ha hecho commit:
- git diff --staged
- los de dos ramas distintas:
  - git diff rama\_1 rama\_2

Terminal



```
$ git diff  
diff --git a/prueba b/prueba  
deleted file mode 100644  
index e69de29..0000000
```

# git reset [FICHERO | [--hard] COMMIT]

quita un fichero del entorno, vuelve a un commit anterior, etc.

- git reset Clase.java
- git reset --hard  
b01557d80d5f53...20d8b8c16

Terminal



```
$ git reset  
291d617808682149d0a455cdfbf8973  
dc862f7b7
```

# git show [commit]

Muestra los metadatos y contenidos de un cierto commit.

- git show  
b01557d80d5f53...20d8b8c16

Terminal



```
$ git log
commit
03e3790be4d5c5c7c483d82f690785c45d60b8c6
(HEAD -> master)
Author: Albert Álvarez Carulla
<albert.alvarez.carulla@gmail.com>
Date: Tue Mar 23 07:39:23 2021 +0100
Commit en blanco de ejemplo
commit
cb470112f942994d34d2f19176d1067ca5698174
Author: Albert Álvarez Carulla
<albert.alvarez.carulla@gmail.com>
Date: Tue Mar 23 07:38:14 2021 +0100
Añadida información del proyecto
```

# git log [--follow FICHERO]

Muestra el historial de versiones de la rama actual, o de un cierto fichero.

- git log
- git log --follow Clase.java

Terminal



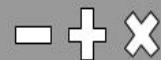
```
$ git log
commit
03e3790be4d5c5c7c483d82f690785c45d60b8c6
(HEAD -> master)
Author: Albert Álvarez Carulla
<albert.alvarez.carulla@gmail.com>
Date: Tue Mar 23 07:39:23 2021 +0100
Commit en blanco de ejemplo
commit
cb470112f942994d34d2f19176d1067ca5698174
Author: Albert Álvarez Carulla
<albert.alvarez.carulla@gmail.com>
Date: Tue Mar 23 07:38:14 2021 +0100
Añadida información del proyecto
```

# git pull URL\_REPO\_REMOTO

Copia el repositorio remoto a local y mezcla los cambios.

```
- git pull  
https://github.com/jmmedin  
ac03vjp/PuestaProduccionS  
egura.git
```

Terminal



```
$ git pull
```

```
  Eremote: Enumerating objects: 5, done.  
  remote: Counting objects: 100% (5/5), done.  
  remote: Total 3 (delta 0), reused 0 (delta 0),  
  pack-reused 0  
Unpacking objects: 100% (3/3), 628 bytes | 1024  
bytes/s, done.  
From https://github.com/Albert-Alvarez/demo  
03e3790..2a62f26 master ->  
origin/master  
Updating 03e3790..2a62f26  
Fast-forward  
 README.md | 1 +  
 1 file changed, 1 insertion(+)
```

# git push [-all] [NOMBRE\_RAMA] [NOMBRE\_RAMA]

Sube los cambios de nuestro repositorio local al remoto o origin.

- git push origin master
- git push --all origin

Terminal



```
$ git push
```

```
Enumerating objects: 4, done.
```

```
Counting objects: 100% (4/4), done.
```

```
Delta compression using up to 8 threads
```

```
Compressing objects: 100% (2/2), done.
```

```
Writing objects: 100% (4/4), 344 bytes | 344.00  
KiB/s, done.
```

```
Total 4 (delta 1), reused 0 (delta 0), packreused 0  
remote: Resolving deltas: 100% (1/1), done.
```

```
To https://github.com/Albert-Alvarez/demo.git
```

```
* [new branch] master -> master
```

```
Branch 'master' set up to track remote branch  
'master' from 'origin'.
```

# git branch [[-d] BRANCH\_ID]

Lista, crea y borra ramas  
(branches) de un proyecto

- git branch
- git branch rama\_1
- git branch -d rama\_1

Terminal



```
$ git branch  
* main
```

# git checkout [-b] [branch | commit\_id]

Salta entre las diferentes versiones del repositorio modificando los contenidos de nuestro working tree. Se debe de indicar o el nombre de la rama o el commit id (6 primeros dígitos). Podemos crear una rama nueva con el parámetro '-b'. Ver también 'git switch'

- git checkout rama\_2
- git checkout -b rama\_2

# git merge <branch>

Fusión de dos ramas.

Incorpora a nuestra rama actual los cambios de la rama que se indica en el comando.

- git merge rama\_2

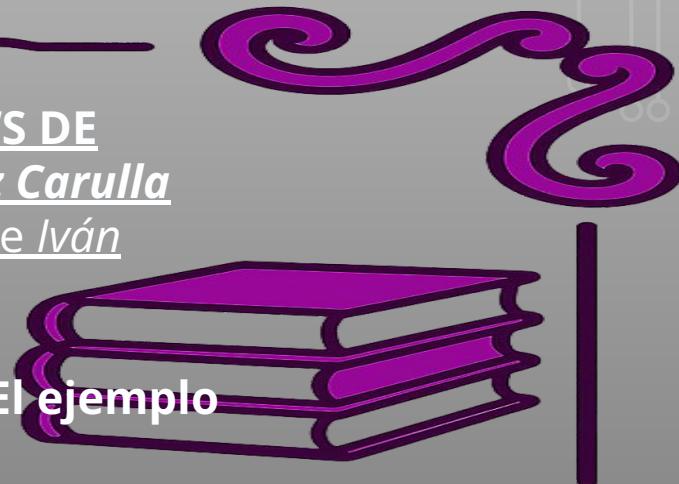
Terminal



```
$ git merge <branch>
Updating 2a62f26..836afbb
Fast-forward
test.yml | 0
 1 file changed, 0 insertions(+), 0 deletions(-)
)
create mode 100644 test.yml
```

# Bibliografía y Webgrafía

- **INTRODUCCIÓN A LOS WORKFLOWS DE DESARROLLO VCS** de *Albert Álvarez Carulla*
- **Taller de Git y GitHub desde cero** de *Iván Martínez Ortiz*
- **Git** de *Victor Ponz*
- **Sistemas de control de versiones: El ejemplo de Git** de *Rafael López García*
- **<https://codefinity.com/courses/git-essentials>**



# Gracias!

¿Alguna pregunta?



[informatica.iesvalledeljerteplasencia.es](http://informatica.iesvalledeljerteplasencia.es)



[coordinacion.cenfp@iesvp.es](mailto:coordinacion.cenfp@iesvp.es)



C/ Pedro y Francisco González, s/n  
10600, Plasencia (Cáceres)



927 01 77 74

