



# Curso Git

---

Arturo Silvelo

Try New Roads

# Introducción

---

# ¿Qué es un control de versiones?

**Version Control System o VCS** es una herramienta esencial en el desarrollo de software y la gestión de proyectos. Permite gestionar los cambios realizados en un archivo o conjunto de archivos a lo largo del tiempo, facilitando el seguimiento de modificaciones, la colaboración entre equipos y la recuperación de versiones anteriores.

## Beneficios clave:

- Colaboración
- Trazabilidad
- Recuperación

## Ejemplos de VCS:

- Git: *Distribuido*
- Subversion (SVN):  
*Centralizado*

# ¿Qué es Git?

**Git** es un sistema distribuido de control de versiones, **gratuito** y de **código abierto**, desarrollado por **Linus Torvalds** en 2005. Está diseñado para mejorar:

- **Rendimiento** de las operaciones.
- Uso eficiente del **espacio de almacenamiento**
- **Distribuido**: Eliminar la necesidad de un servidor central

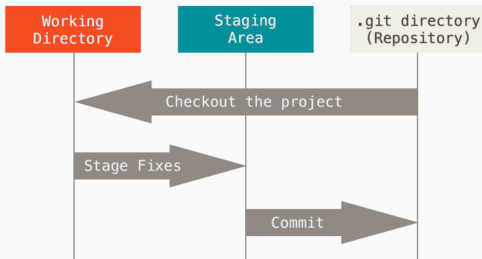
## Fundamentos:

- **Repositorios:** Un repositorio es un espacio donde se almacenan los archivos y su historial de versiones. Pueden ser **locales** (en tu máquina) o **remotos** (en un servidor).
- **Ramas:** Git utiliza un sistema de ramas, donde cada proyecto tiene al menos una rama principal (por defecto llamada **main** o **master**). Las ramas permiten trabajar en paralelo sin interferir con la versión principal.

# Estados de los Archivos

**Estados de los archivos:** En Git, los archivos pasan por diferentes estados en su ciclo de vida:

- **Modificado (modified):** El archivo ha cambiado, pero aún no está preparado para ser confirmado.
- **Preparado (staged):** El archivo está listo para ser confirmado.
- **Confirmado (committed):** El archivo ha sido guardado en el historial del repositorio.



# ¿Qué es una rama?

Una **rama** es una versión de la colección de directorios y archivos del repositorio. Cada vez que se crea una rama, se crea una **copia** de la colección de archivos actual.

Se pueden crear **ramas a partir de otras ramas**. Los cambios realizados en esas ramas pueden ser **integrados en otras ramas**. Este proceso se conoce como **merge** (fusión).

## ¿Para qué sirven las ramas?

Las ramas son útiles en un **entorno de colaboración**, donde diferentes personas están trabajando en el mismo código. Mientras una persona puede estar añadiendo una nueva funcionalidad al código, otra podría estar arreglando un **bug** y otra añadiendo **documentación**.

De esta forma, partiendo del mismo código, se generan diferentes **ramas**. Esto permite **aislar el trabajo** de cada persona, y una vez finalizado, se pueden integrar esos cambios en la rama principal.



# La rama master o la rama main

La rama **master** ha sido tradicionalmente la rama principal de un repositorio, y suele ser creada automáticamente cuando se inicia un nuevo proyecto en Git. Aunque históricamente se ha utilizado el nombre **master** para esta rama, **no es obligatorio** que se llame de esta manera, ya que su elección responde a razones históricas y no tiene implicaciones técnicas.

En la actualidad, muchos proyectos y plataformas, como GitHub, recomiendan cambiar el nombre de la rama principal a **main**. Esta recomendación busca evitar connotaciones negativas y racistas asociadas al término **master**, promoviendo un lenguaje más inclusivo.

# Git, GitHub y GitLab

**Git** es el sistema de control de versiones que permite gestionar proyectos de manera local. Aunque es posible configurar un servidor remoto propio, esto requiere tiempo y recursos, lo que no siempre es rentable.



Aquí es donde entran plataformas como **GitHub** y **GitLab**, que ofrecen alojamiento en la nube basado en Git, con interfaces gráficas amigables y servicios adicionales como **CI/CD** y **gestión de proyectos**, facilitando la colaboración remota y el flujo de trabajo.

# Instalar Git

Comprueba si *git* está instalado, si está mostrará la versión:

```
git --version
```

La instalación de Git varía según el sistema operativo:

- **Windows:** Descargar desde <https://git-scm.com/> y seguir las instrucciones predeterminadas.
- **Linux:** Descargar desde <https://git-scm.com/downloads/linux> o utilizar el siguiente comando:
  - En Debian/Ubuntu: `sudo apt install git`
  - En Fedora: `sudo dnf install git`
- **Mac:** Usar Homebrew: `brew install git` o descargar desde <https://git-scm.com/>.

# Configurar Git

Antes de empezar a usar Git, es recomendable hacer una configuración mínima para asociar tus commits con tu nombre y correo electrónico.

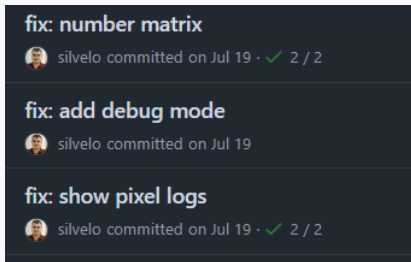
Para que tus commits se asocien correctamente a tu nombre y aparezcan de forma adecuada en plataformas como GitHub, necesitas realizar la siguiente configuración:

```
git config --global user.name "Arturo Silvelo"  
git config --global user.email "arturo.silvelo@gmail.com"
```

# Configurar Git

Si quieres cambiar la configuración para un repositorio en particular, puedes eliminar la opción `--global` y configurarlo directamente en el repositorio:

```
cd <tu-repositorio>  
git config user.name "silvelo"  
git config user.email "silvelo@work.com"
```



Por defecto, Git intenta abrir el editor **vim** para modificar los archivos cuando encuentra conflictos o para escribir el mensaje del commit.

Se puede cambiar esta configuración para que Git abra el editor de texto de tu elección. Por ejemplo, para usar Visual Studio Code, ejecuta el siguiente comando:

```
git config --global core.editor "code"
```

# Configurar Git

La opción **core.autocrlf** controla cómo Git maneja los saltos de línea entre diferentes sistemas operativos.

## Opciones:

- **true**: Convierte los saltos de línea de tipo CRLF (Windows) a LF (Linux/Mac) al hacer commit.
- **input**: No modifica los saltos de línea al hacer commit, pero convierte CRLF a LF al hacer **checkout**.
- **false**: No realiza ninguna conversión.

## Ejemplo de configuración:

```
git config --global core.autocrlf true # En Windows
git config --global core.autocrlf input # En Linux/Mac
```

# Configurar Git

Los alias permiten acortar y personalizar comandos largos de Git, haciendo que tu flujo de trabajo sea más eficiente.

## Ejemplos de alias comunes:

- **st** para **status**
- **co** para **checkout**
- **br** para **branch**

## Ejemplo de configuración de alias:

```
git config --global alias.st status # `git st` en lugar de `git  
↪ status`  
git config --global alias.co checkout # `git co` en lugar de `git  
↪ checkout`
```



# Comprobar Configuración de Git

Para revisar la configuración actual de Git, usa el comando:

```
git config --list
```

Git tiene varios archivos de configuración:

- **local**: Para el repositorio actual.
- **global**: Para todos los repositorios del usuario.
- **system**: Para todos los usuarios del sistema.

**El último valor prevalece.**

Además, puedes usar los siguientes filtros para comprobar la configuración:

- **--local**, **--global**, **--system** para filtrar.
- **--show-scope** para saber de dónde proviene cada valor.

## Trabajando Local

---

# Git de forma local

Trabajando con Git de forma local



Directorio  
de Trabajo



Área temporal  
transitoria  
(Stage Area)



Repositorio  
Local



Repositorio  
Remoto

# Iniciar un nuevo proyecto

Para crear un nuevo proyecto en Git, usa el comando:

```
git init <nombre del proyecto>
```

Si ya tienes un directorio creado y deseas convertirlo en un repositorio de Git, navega a él con:

```
cd <directorio>  
git init
```

En ambos casos, Git crea una rama principal por defecto y el directorio **.git** se genera para almacenar toda la información del proyecto.

Para comprobar si tu proyecto tiene un repositorio inicializado, puedes usar el comando:

```
git status
```

# Directorio de trabajo

El **directorio de trabajo** es la carpeta donde tienes todos los archivos y en la que has iniciado tu repositorio.

Creamos un nuevo archivo con el comando:

```
touch index.html
```

Luego, revisamos el estado del repositorio con:

```
git status
```

Esto mostrará que el archivo **index.html** ha sido añadido y está en estado **modificado**.

Para obtener una vista más simplificada, puedes usar:

```
git status -s
```

## Deshacer un archivo modificado (Usando `git restore`)

Si modificamos un archivo y queremos volver al estado inicial, podemos usar el comando:

```
git restore index.html  
git restore .  
git restore '*.js'
```

**Nota:** Este comando hará que los cambios se pierdan. Si el archivo no está guardado en un commit previo, Git nos dará un error.

## Deshacer un archivo modificado (Usando `git checkout`)

El comando `git restore` es relativamente nuevo, y puede que no esté disponible en versiones antiguas de Git. En ese caso, podemos usar como alternativa:

```
git checkout -- index.html  
git checkout -- '*.md'  
git checkout .
```

Estos comandos tienen la misma función que `git restore`, restaurando el archivo o conjunto de archivos al estado anterior.

# Eliminar archivos no rastreados (Usando `git clean`)

Si queremos eliminar archivos no rastreados del directorio de trabajo, podemos usar el comando `git clean`.

```
touch index2.html  
git clean
```

Opciones principales de `git clean`:

- `-n`: Muestra qué se eliminaría sin ejecutar la acción.
- `-f`: Fuerza la eliminación de los archivos.
- `-d`: Permite eliminar directorios no rastreados.
- `-i`: Activa el modo interactivo para confirmar cada acción.

```
git clean -n    # Muestra qué se eliminaría  
git clean -f    # Fuerza la limpieza
```



El área de **staging** es una zona temporal donde preparamos los archivos modificados antes de confirmarlos con un **commit**.

```
git add index.html          # Prepara el archivo index.html
git add archivo1.js archivo2.js  # Prepara varios archivos
git add *.js                # Prepara todos los archivos .js
git add -A                  # Prepara todos los cambios (incluyendo
↪ eliminaciones)
git add .                    # Prepara todos los cambios en el directorio
↪ actual
git add resources/          # Prepara todos los archivos en el directorio
↪ resources
```

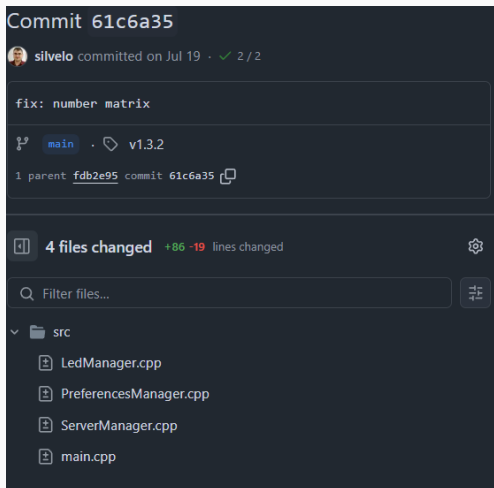
# Sacar archivos de Staging

Podemos eliminar los ficheros del área de **staging** y devolverlos al estado de modificados con el comando **git reset**.

```
git reset index.html          # Elimina index.html del área de staging
git reset archivo1.js archivo2.js  # Elimina varios archivos del área de
↳ staging
git reset *.js                # Elimina todos los archivos .js del área
↳ de staging
git reset -A                  # Elimina todos los archivos del área de
↳ staging
git reset .                    # Elimina todos los cambios del directorio
↳ actual del área de staging
git reset resources/          # Elimina todos los archivos del directorio
↳ resources del área de staging
```

# ¿Qué es un commit?

Los commits sirven para registrar los cambios que se han producido en el repositorio. Cada commit muestra el estado de todos los archivos del repositorio, el autor, la fecha y otra información útil.



# ¿Cómo hacer un commit?

Para guardar los ficheros del área de **staging**, se utiliza el comando:

```
git commit    # Este comando creará una referencia al commit
```

Este comando abrirá el editor para que puedas poner un mensaje de commit. Si quieres añadir el mensaje directamente en el comando, puedes usar la opción **-m**:

```
git commit -m 'new feature'    # Realiza un commit con el mensaje 'new feature'
```

# Commit sin staging

También es posible evitar añadir directamente los archivos modificados al área de **staging**. Para realizar esta operación se utiliza el comando:

```
git commit -a    # Realiza un commit de todos los archivos modificados sin  
↪ necesidad de añadirlos a staging
```

Este comando realizará un commit directamente de los archivos modificados. Además, se puede añadir la opción **-m** para incluir el mensaje de commit directamente:

```
git commit -am 'new feature'    # Realiza un commit con el mensaje 'new feature'
```

**Nota:** Esto sólo funciona para archivos modificados. Los archivos nuevos o eliminados necesitan ser añadidos a staging primero.

Cada **commit** se graba con un hash único que puede ser complicado de utilizar como referencia rápida. Para esto existe **HEAD**, que normalmente apunta al último **commit** de la rama activa.

```
# Mostrar la rama a la que apunta HEAD  
git symbolic-ref HEAD
```

```
# Mostrar el hash del commit al que apunta HEAD  
git rev-parse HEAD
```

# Deshacer Cambios

Si necesitamos deshacer el último **commit** porque nos hemos equivocado o faltan archivos, podemos hacerlo de dos maneras:

```
# Mantener los cambios  
git reset --soft HEAD~1
```

```
# No mantener los cambios  
git reset --hard HEAD~1
```

El **HEAD~1** indica que queremos movernos a la versión inmediatamente anterior a la actual.

**Nota:** Esto solo funcionará si los cambios no se han subido al repositorio remoto.

# Arreglar Commit

Si lo único que necesitamos es corregir el último `commit`:

```
# Editar el mensaje  
git commit --amend -m 'Nuevo mensaje'
```

```
# Añadir archivos y modificar el commit  
git add archivo3.js  
git commit --amend -m 'Nuevo mensaje'
```

El comando `amend` no crea un nuevo `commit`, sino que actualiza el anterior.



# Ignorar archivos

No todos los archivos en nuestro directorio de trabajo deben ser controlados por **git**. A veces, existen archivos o directorios que no queremos incluir en el repositorio porque son de configuración, temporales, o no aportan valor al historial de cambios.

Para que **git** los ignore, debemos especificarlos en el archivo **.gitignore**. Ejemplo:

```
# Ignorar carpeta de módulos
node_modules
# Ignorar fichero con variables de entorno
.env
# Ignorar fichero de sistema
.DS_Store
# Ignorar carpeta generada
build/
```

# Eliminar ficheros

Si queremos borrar un fichero y registrar el cambio, podemos hacerlo de dos formas:

```
# Forma manual
```

```
rm config.js          # Elimina el fichero
git add config.js     # Marca el fichero como eliminado en staging
git commit -m 'Remove config'
```

```
# Usando git directamente
```

```
git rm config.js     # Elimina el fichero y lo marca para el commit
git commit -m 'Remove config'
```

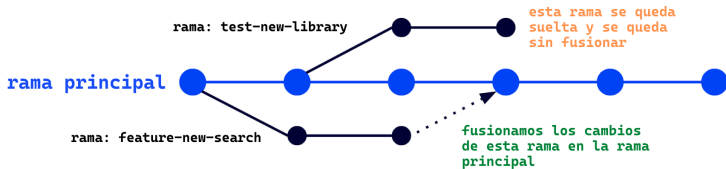
Si queremos borrar el fichero del repositorio, pero conservarlo en el directorio local:

```
git rm --cached <nombre-de-archivo>
```

**Nota:** Para eliminar carpetas, utiliza la opción recursiva:

```
git rm -r <nombre-de-carpeta>
```

## Creación de ramas



# Crear una rama

Para crear una nueva rama y cambiarte a ella, puedes usar los siguientes comandos:

```
# Para crear una nueva rama  
git branch first-branch
```

```
# Para cambiar a la nueva rama  
git switch first-branch
```

Si quieres realizar ambos pasos en un solo comando, puedes usar:

```
# Crea la rama y te cambia a ella  
git switch -c first-branch
```

También existe la opción más antigua de realizarlo con:

```
# Crea la rama y te cambia a ella  
git checkout -b first-branch
```

**Nota:** A partir de Git 2.23, se introdujo `git switch` como una alternativa más sencilla a `git checkout` para cambiar de ramas, pero ambas opciones siguen siendo válidas.

# Listando ramas

Para mostrar todas las ramas disponibles localmente, puedes usar el comando:

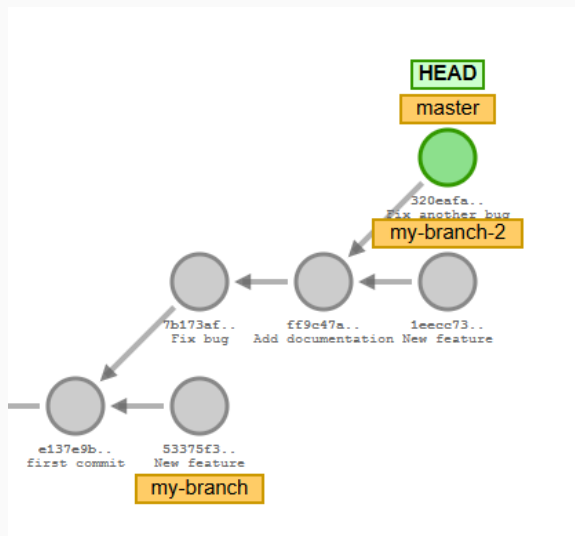
```
git branch
```

La rama actual tendrá un asterisco al inicio.

También puedes usar el mismo comando para ver la rama en la que te encuentras. Si deseas ordenar las ramas según la última fecha de modificación, puedes hacerlo con:

```
git branch --sort=-committerdate
```

# Trabajando con ramas



# Trabajando con ramas

Estando en nuestro repositorio y en la rama principal:

```
git branch --show-current
# Grabamos un commit en la rama master
git commit -am "First commit"
```

Vamos añadir una nueva características a nuestro programa:

```
# Creamos nuestra primera rama
git switch -c my-branch
# Verificamos la rama
git branch --show-current
# Creamos el archivo y commiteamos
code index.js
git commit -am "New feature"
```

Nos han reportado un error en la aplicación principal:

```
# Volvemos a la rama
git switch master
# Editamos el fichero y commiteamos
code bug.js
git commit -am 'fix bug'
code readme.md
git commit -am 'Add Documentation'
```

Un compañero se pone a trabajar en otra característica:

```
# Creamos una nueva rama
git switch -c my-branch-2
code feature.js
git commit -am 'New Feature'
```

Nos reportan otro error:

```
# Volvemos a la rama
git switch master
code bug_2.js
git commit -am 'Fix another bug'
```



# Fusionando ramas

Con fusión nos referimos a que los cambios realizados en una rama se integren en otra, de forma que el código generado en la nueva rama se asimile en la rama destino.

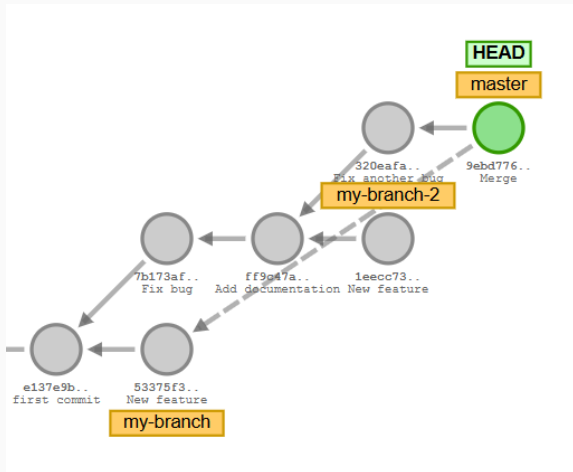
Para fusionar ramas, podemos usar el comando `git merge`.

Si continuamos con el ejemplo anterior y queremos fusionar una rama con otra, debemos colocarnos en la rama destino y usar el comando:

```
# Fusiona la rama "my-branch" con la rama actual  
git merge my-branch
```

Podemos verificar en los logs que se ha creado un nuevo commit de merge. Este commit incluye todos los cambios realizados en la rama `my-branch`.

# Fusionando ramas



# Merge fast-forward

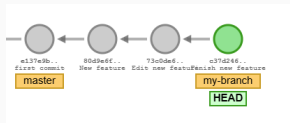
Un *merge fast-forward* ocurre cuando la rama destino no ha tenido ningún cambio adicional desde que se creó la rama que queremos fusionar.

En este caso, Git simplemente mueve el puntero de la rama destino al último *commit* de la rama que estamos fusionando, sin crear un *commit de merge*.

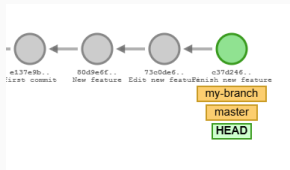
```
# Realiza un merge fast-forward, sin crear un commit de merge  
git merge --ff-only my-branch
```

# Merge fast-forward

Creamos la nueva rama y hacemos las modificaciones necesarias:



Una vez finalizada la característica hacemos el *merge* en la rama master



```
git checkout -b my-branch
# Añadimos un fichero
git commit -m 'New feature'
# Modificamos el fichero
git commit -m 'Edit new
↪ feature'
# Modificamos de nuevo
git commit -m 'Finish new
↪ feature'
# Volvemos a la rama destino
git checkout master
# Hacemos el merge
git merge --ff-only
↪ my-branch
# Comprobamos el gráfico
git log --oneline --graph
```

## Merge no fast-forward

Un *merge no fast-forward* ocurre cuando la rama destino ha tenido cambios adicionales desde que se creó la rama que queremos fusionar.

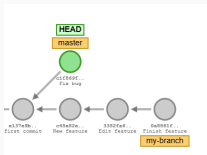
En este caso, Git no puede simplemente mover el puntero de la rama destino al último commit de la rama que estamos fusionando. Por ello, crea un commit de merge que combina los cambios de ambas ramas.

```
# Fusiona la rama y crea un commit de merge  
git merge my-branch --no-ff
```

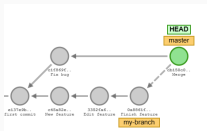
Este tipo de merge es útil para preservar el historial completo de los cambios realizados en cada rama.

# Merge fast-forward

Creamos la nueva rama y hacemos las modificaciones necesarias:



Una vez finalizada la característica hacemos el *merge* en la rama master



```
git checkout -b my-branch
# Añadimos un fichero
git commit -m 'New feature'
# Modificamos el fichero
git commit -m 'Edit new
↪ feature'
# Modificamos de nuevo
git commit -m 'Finish new
↪ feature'
# Volvemos a la rama destino
git checkout master
# Creamos modificaciones
git commit -m 'Fix Bug'
# Hacemos el merge
git merge my-branch --no-ff
# Comprobamos el gráfico
git log --oneline --graph
```

# Merge Squash

Un *merge squash* permite combinar todos los commits de una rama en un solo commit al fusionarla con otra rama. Este método es útil para simplificar el historial de cambios, especialmente cuando hay múltiples commits pequeños o intermedios en la rama que estamos fusionando.

```
# Realiza un merge squash
```

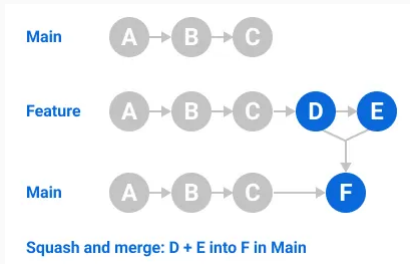
```
git merge --squash my-branch
```

```
# Después, crea un commit manualmente
```

```
git commit -m "Resumen de cambios de my-branch"
```

El *merge squash* no crea un commit de merge automáticamente y no mueve el puntero de la rama destino, sino que combina los cambios de la rama especificada en el área de staging.

# Merge Squash



```
git checkout -b my-branch
# Añadimos un fichero
git commit -m 'New feature'
# Modificamos el fichero
git commit -m 'Edit new
↳ feature'
# Modificamos de nuevo
git commit -m 'Finish new
↳ feature'
# Volvemos a la rama destino
git checkout master
# Creamos modificaciones
git commit -m 'Fix Bug'
# Hacemos el merge
git merge my-branch --squash
git commit -m "Merge changes
↳ squash"
# Comprobamos el gráfico
git log --oneline --graph
```



# Modificar el mensaje de Merge

Cuando realizamos una fusión de ramas usando `git merge`, Git crea automáticamente un mensaje de commit para registrar la fusión. Sin embargo, podemos personalizar este mensaje antes de completar el commit. Para hacerlo, existen dos opciones:

```
# Editar el mensaje del merge  
git merge --edit
```

```
# Fusionar sin hacer commit automáticamente  
git merge --no-commit
```

# Merge con Conflictos

Un conflicto de merge ocurre cuando Git no puede determinar automáticamente qué cambios deben prevalecer durante una fusión. Esto suele suceder cuando diferentes ramas han modificado las mismas líneas de un archivo o si uno ha eliminado un archivo que el otro ha editado.

Los conflictos no son algo malo, sino una parte normal del trabajo en equipo, especialmente en proyectos grandes donde múltiples desarrolladores trabajan en los mismos archivos.

## Pasos típicos para manejar conflictos:

1. Ejecuta `git status` para identificar los archivos en conflicto.
2. Edita manualmente los archivos afectados para resolver el conflicto.
3. Añade los archivos resueltos al área de staging con `git add`.
4. Completa la fusión con `git commit`.

# Creando un conflicto

```
git init
# Creamos un fichero
↪ index.html y lo
↪ añadimos
git add .
git commit -m 'Init
↪ commit'
# Nos cambiamos a otra
↪ rama
git switch -c changes
# Editar el fichero
↪ index.html
git commit -am 'Edit
↪ index.html'
# Volvemos a la rama
↪ principal
git switch main
#Editamos el fichero
git commit -am 'Edit on
↪ main index.html'
# Fusionamos las ramas
git merge changes
```

```
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
G:\Users\Arturo\repo\course_7 [master|MERGING +0 -0 |1 | +0 -0 -0 |1 |] git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Unmerged paths:
  (use "git add <file>..." to mark resolution)
    both modified:   index.html

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

# Resolviendo el Conflicto

Cuando ocurre un conflicto, tienes dos opciones iniciales:

1. Ejecutar `git merge --abort` para deshacer el merge y volver al estado previo.
2. Resolver el conflicto manualmente.

Para entender qué ha sucedido, puedes usar `git diff` para examinar las diferencias.

## Anotaciones en los conflictos:

- El contenido de la rama principal aparece entre `<<<<<< HEAD` y `=====`.
- El contenido de la rama que queremos fusionar está entre `=====` y `>>>>>> branch`.

# Opciones para resolver un Conflicto

## Opciones para resolver un conflicto:

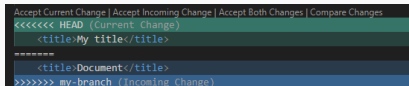
```
# Usar el cambio de la rama principal  
git checkout --ours archivo-conflicto
```

```
# Usar el cambio de la rama que estamos fusionando  
git checkout --theirs archivo-conflicto
```

**Personalización:** Modifica manualmente el archivo y guarda los cambios deseados.

Una vez resuelto, utiliza:

```
git add archivo-conflicto  
git commit
```



```
Accept Current Change | Accept Incoming Change | Accept Both Changes | Compare Changes  
<<<<<< HEAD (Current Change)  
  <title>My title</title>  
=====
```

Las ramas fusionadas no se eliminan automáticamente, por lo que debemos hacerlo manualmente.

```
# Eliminar una rama fusionada  
git branch --delete my-branch  
git branch -d my-branch
```

Si la rama no ha sido fusionada, forzaremos su eliminación con:

```
# Eliminar una rama no fusionada  
git branch -D my-branch
```

El rebase es una operación que permite reescribir el historial de commits de una rama, permitiendo **añadir, mover, ordenar y eliminar commits**.

Para realizar un rebase:

1. Nos posicionamos en la rama donde queremos añadir los commits:  
`git switch main`
2. Ejecutamos el rebase con la rama de la que queremos añadir commits:  
`git rebase my-branch`
3. Resolvemos los conflictos en caso de que surjan.

Cuando se realiza un rebase, pueden surgir conflictos que necesitan ser resueltos. Tienes tres opciones para manejar estos conflictos:

1. Continuar con el rebase después de resolver los conflictos:

```
git rebase --continue
```

2. Omitir el commit con conflicto y continuar el rebase:

```
git rebase --skip
```

3. Abortar el rebase y volver al estado anterior:

```
git rebase --abort
```



# Rebase Interactivo

El rebase interactivo permite reescribir el historial de una rama de manera más detallada. Se utiliza el siguiente comando:

```
git rebase --interactive
```

Dentro del rebase interactivo, puedes realizar varias operaciones sobre los commits:

- **Cambiar el mensaje de un commit (pick):** Permite modificar el mensaje de un commit.
- **Reordenar commits:** Puedes cambiar el orden de los commits en el historial.
- **Borrar un commit (drop):** Elimina un commit específico.
- **Fusionar commits (squash):** Combina dos o más commits en uno solo.
- **Separar un commit en dos o más (edit):** Divide un commit en múltiples commits.

El rebase, aunque útil, tiene algunos riesgos y desventajas:

- **Pérdida de trabajo:** Si no se maneja correctamente, se pueden eliminar commits importantes.
- **Conflictos silenciosos:** Los conflictos pueden surgir sin ser detectados si no se resuelven adecuadamente.
- **Historia artificial:** Reescribir la historia de los commits puede llevar a una secuencia de cambios que no refleja la realidad del desarrollo.

# Ejemplo Rebase

```
# Inicialización del repositorio
git init
git add .
git commit -m "init commit"
# Crear la rama 'feature' y comenzar
↳ a trabajar en ella
git switch -c feature
# Cambios iniciales en feature.js
git add feature.js
git commit -m "add feature"
git commit -am "Edit feature"
git commit -am "Edit feature"
# Añadir y modificar feature_lib.js
git add feature_lib.js
git commit -m "add feature lib"
git commit -am "edit feature lib"
git commit -am "remove commit"
↳ feature lib"
```

```
# Cambios en feature_dep.js
git add feature_dep.js
git commit -m "add feature_dep"
git commit -am "edit feature_dep"
git commit -am "new edit"
↳ feature_dep"
git commit -am "other edit"
↳ feature_dep"
git commit -am "more edit"
↳ feature_dep"

git rebase -i HEAD~12
# Marcamos las operaciones en el
↳ fichero y realizamos las
↳ operaciones una vez finalizado
git rebase --continue
```

# Ejemplo Rebase

```
GNU nano 6.3 C:/Users/ArturoSilvelo/base_repo_2/.git/rebase-merge/git-rebase-todo
pick ca2e2df add feature
pick 1c0bd32 edit feature
pick fdb6cbf edit feature
pick fa2f35d add feature lib
pick c8eeefd edit feature lib
pick fe99d3f remove commit feature lib
pick 2bb8a33 add feature dep
pick 9ddb81d edit feature dep
pick ab18d8e more edit feature dep
pick 39d9de2 other edit feature dep
pick 41ed5c3 more edit
pick ecee351 add graph image

# Rebase e90e6da..ecee351 onto e90e6da (12 commands)
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
# f, fixup [-C | -c] <commit> = like "squash" but keep only the previous
#                          commit's log message, unless -C is used, in which case
#                          keep only this commit's message; -c is same as -C but
#                          opens the editor
# x, exec <command> = run command (the rest of the line) using shell
# ! <command> = run command (the rest of the line) using shell
```

Figure 1: Fichero de configuración del rebase

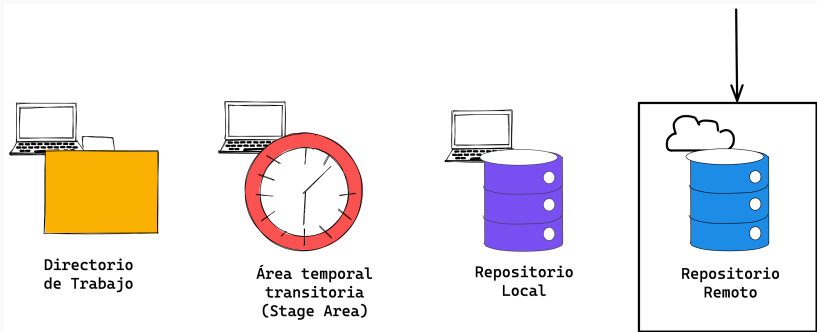
# Rebase vs Merge

Aspecto	Rebase	Merge
<b>Ventajas</b>	<ul style="list-style-type: none"><li>- Historial más limpio y lineal.</li><li>- Menos commits de fusión.</li></ul>	<ul style="list-style-type: none"><li>- Conserva la historia completa.</li><li>- Fácil de usar, no requiere manipular el historial.</li></ul>
<b>Desventajas</b>	<ul style="list-style-type: none"><li>- Reescribe el historial, lo que puede causar problemas si ya se ha compartido la rama.</li><li>- Riesgo de pérdida de trabajo si no se hace correctamente.</li></ul>	<ul style="list-style-type: none"><li>- Historial más complicado, con muchos commits de fusión.</li><li>- Conflictos de fusión pueden ser más complejos de resolver.</li></ul>
<b>Cuándo usarlo</b>	<ul style="list-style-type: none"><li>- Cuando quieres un historial más limpio y lineal.</li></ul>	<ul style="list-style-type: none"><li>- Cuando deseas conservar el historial completo de la rama y no te importa tener commits de fusión.</li></ul>
<b>Cuándo evitarlo</b>	<ul style="list-style-type: none"><li>- Si ya has compartido la rama con otros colaboradores.</li></ul>	<ul style="list-style-type: none"><li>- Cuando prefieres mantener la historia tal cual ocurrió.</li></ul>

## Trabajando de forma remota

---

# Trabajando de forma remota



# Creando un repositorio

## Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)


Required fields are marked with an asterisk (\*).


Owner \*      Repository name \*

 silvelo /

Great repository names are short and memorable. Need inspiration? How about [potential-dollop](#)?

Description (optional)

☒  **Public**  
Anyone on the internet can see this repository. You choose who can commit.

☐  **Private**  
You choose who can see and commit to this repository.

Initialize this repository with:

☐ **Add a README file**  
This is where you can write a long description for your project. [Learn more about READMEs.](#)

Add .gitignore

.gitignore template: **None**

Choose which files not to track from a list of templates. [Learn more about ignoring files.](#)

Choose a license


License: **None**

A license tells others what they can and can't do with your code. [Learn more about licenses.](#)

Grant your Marketplace apps access to this repository

You are subscribed to 1 Marketplace app.

☐  **Travis CI**  
Test and deploy with confidence

 You are creating a public repository in your personal account.

Create repository

- Dueño del repositorio
- Nombre del repositorio
- Descripción corta
- Visibilidad del repositorio
- Añadir README, gitignore y licencia



# Configurando conexión SSH

SSH es un protocolo de comunicación segura que permite a los usuarios de una red conectar a un servidor remoto. Esto nos permite trabajar con repositorios remotos sin necesidad de usar usuario y contraseña cada vez que hagamos una acción

```
ssh-keygen -t ed25519 -C "your_email@example.com"
```

# Clonando un repositorio ya creado

Para clonar un repositorio que ya ha sido creado, puedes utilizar el siguiente comando:

```
git clone <URL-del-repositorio>
```

Existen dos formas principales de clonar un repositorio:

- **Usando SSH:** Requiere una clave SSH configurada en tu equipo.  
`git clone git@github.com:user/repository.git`
- **Usando HTTPS:** Requiere ingresar tus credenciales (usuario y contraseña) cada vez que interactúas con el repositorio.  
`git clone https://github.com/user/repository.git`

Al clonar un repositorio, Git crea automáticamente una carpeta con el mismo nombre que el repositorio remoto.

Si deseas especificar un nombre diferente para la carpeta local, puedes hacerlo de la siguiente manera:

```
git clone <URL-del-repositorio> <nombre-de-carpeta>
```

# Enlazar repositorio local con uno remoto

Para enlazar un repositorio local con un repositorio remoto, puedes usar el siguiente comando:

```
git remote add <alias> <direccion>
```

Donde:

- **alias** es el nombre que se le asignará al repositorio remoto, por ejemplo 'origin'.
- **direccion** es la URL del repositorio remoto (puede ser SSH o HTTPS).

Para comprobar los repositorios remotos configurados, usa:

```
git remote -v
```

Este comando te mostrará las direcciones URL de los remotos configurados para el repositorio local.

# Traer cambios remotos al repositorio local

Para traer cambios desde un repositorio remoto, usa los siguientes comandos:

- **git fetch:** Descarga los cambios, pero no los integra en tu rama actual.  
`git fetch <alias>`
- **git pull:** Descarga y fusiona los cambios del remoto en tu rama actual.  
`git pull <alias> <branch>`

Usa **git fetch** para revisar cambios antes de integrarlos, y **git pull** para aplicarlos directamente.

# Escribiendo en el repositorio remoto

Para subir tus cambios locales al repositorio remoto, utiliza el siguiente comando:

```
git push <alias> <branch>
```

- **<alias>**: Es el nombre del repositorio remoto (por ejemplo, origin).
- **<branch>**: Es la rama local que deseas subir al repositorio remoto.

A veces, puedes encontrar problemas al intentar hacer 'git push' si tu rama local está por detrás de la rama remota. En este caso, necesitarás hacer un 'git pull' primero para actualizar tu rama local antes de poder hacer el 'push'.

# Crear ramas en remoto

Para crear una rama en remoto, primero debes crearla en tu entorno local y luego enviarla al repositorio remoto:

- Crea la rama en local con:  
`git switch -c my-branch`
- Envía la rama al remoto con:  
`git push origin my-branch`

Esto crea la rama en el repositorio remoto y la vincula con tu rama local.

Para fusionar ramas, tenemos dos opciones:

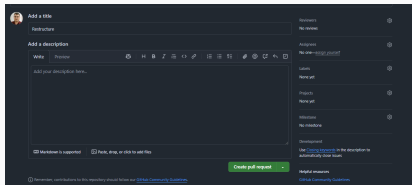
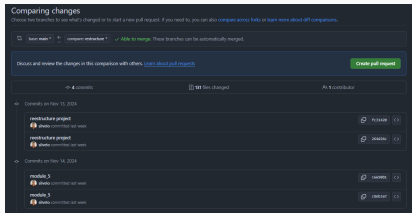
- **De forma local:**

```
# Primero, cambia a la rama de destino  
git switch main  
# Luego, fusiona los cambios de la rama de origen  
git merge my-branch  
# Por último, sube los cambios al repositorio remoto:  
git push origin main
```

- **Mediante una Pull Request:** Esta opción es útil cuando trabajamos en equipos de desarrollo, ya que otra persona revisará los cambios antes de integrarlos. Para ello, primero subimos nuestra rama al repositorio remoto:

```
git push origin my-branch
```

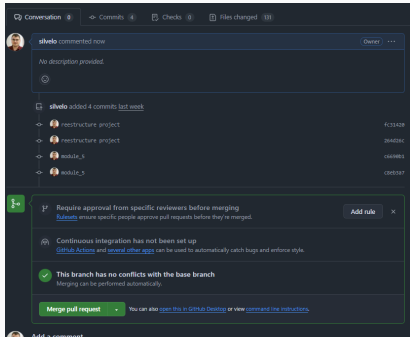
# Como crear una PR



1. Asegúrate de haber subido tu rama al repositorio remoto con el comando:  
`git push origin my-branch`
2. Accede repositorio en GitHub.
3. En la pestaña "Pull Requests", haz clic en "New Pull Request".
4. Selecciona la rama base y la rama con los cambios que quieres fusionar.
5. Agrega una descripción y, si es necesario, asigna revisores.
6. Haz clic en "Create Pull Request" para finalizar.



# Merge PR



Una vez creada y este preparada para ser integrada, la persona encargada de revisar los cambios puede confirma el merge.

fork

---

Un ***fork*** es una copia personal de un repositorio que se encuentra en un repositorio remoto. Se utiliza principalmente cuando deseas realizar cambios en un proyecto de otra persona sin afectar el repositorio original.

Algunos motivos para usar un fork son:

- El proyecto original ha sido abandonado.
- El proyecto original no acepta nuestros cambios.
- No tenemos permisos para enviar código al proyecto original.

# Sincronizar cambios

Para sincronizar tu fork con el proyecto original (upstream), primero debes agregar el repositorio original como un "remote" adicional.

Usa el siguiente comando para añadir el repositorio original como un remoto llamado 'upstream':

```
git remote add upstream <url_proyecto>
```

Ahora tendrás dos remotos configurados:

- **origin:** Apunta a tu fork, donde realizas tus cambios.
- **upstream:** Apunta al repositorio original, desde donde puedes obtener las actualizaciones.

Para sincronizar con el repositorio original, utiliza:

```
git pull upstream main
```

Para sincronizar con tu fork, utiliza:

```
git pull origin main
```

## Buenas Prácticas

---

# Flujos de trabajo

Todas las estrategias de flujo de trabajo en Git se basan esencialmente en cómo se crean y fusionan las ramas con la principal. No hay una estrategia universalmente mejor, simplemente cada proyecto define la que mejor se adapta a sus necesidades.

## Principales flujos de trabajo:

- **Git Flow:** Estructura ramificada con ramas de desarrollo, producción y soporte.
- **GitHub Flow:** Enfoque simple y lineal adecuado para despliegues continuos.
- **Trunk Based:** Ramas cortas que se fusionan rápidamente a la principal.
- **Ship/Show/Ask:** Flujo basado en revisiones y ciclos de retroalimentación antes de fusionar.

**Git Flow** es una estrategia de ramificación que organiza el trabajo en dos ramas principales y ramas de apoyo temporales.

## Ramas principales:

- **main**: Contiene el código en producción.
- **develop**: Almacena los últimos cambios en desarrollo. Cuando está lista y estable, sus cambios se fusionan en **main**.

**Ramas de apoyo**: Estas ramas son temporales y se eliminan una vez fusionadas:

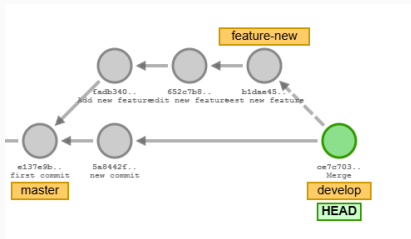
- **Feature**: Para desarrollar nuevas funcionalidades.
- **Release**: Para preparar versiones listas para producción.
- **Hotfix**: Para corregir errores en producción.

# Ramas de apoyo en Git Flow

Rama	Desde dónde se crea	A dónde se fusiona	Convención de nombre
Feature	develop	develop	feature-*
Release	develop	main y develop	release-*
Hotfix	main	main y develop	hotfix-*

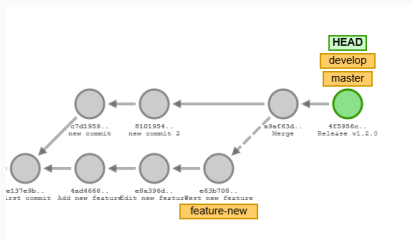


# Feature



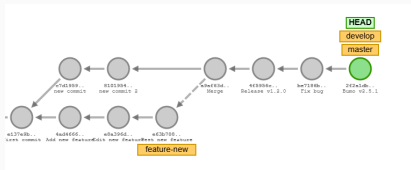
```
# Creamos la nueva rama desde  
↳ develop  
git switch -c feature-new develop  
# Hacemos los cambios necesarios  
# Volvemos a develop y fusionamos  
↳ las ramas  
git switch develop  
git merge --no-ff feature-new  
# Eliminamos la rama  
git branch -d feature-new
```

El flag `--no-ff` es opcional, pero genera siempre un commit al fusionar. Esto deja un commit en el historial que contiene todos los cambios de la rama, haciendo más sencillo seguir el flujo de trabajo.



```
# Crear una nueva rama de release
↳ desde develop
git checkout -b release-1.2.0
↳ develop
# Actualizar archivos para reflejar
↳ el cambio de versión
git commit -am 'Release v1.2.0'
# Cambiar a la rama main para
↳ fusionar los cambios
git switch main
# Fusionar la rama de release con un
↳ commit dedicado
git merge --no-ff release-1.2.0
# Cambiar a la rama develop para
↳ sincronizarla
git switch develop
# Fusionar los cambios de la rama de
↳ release en develop
git merge --no-ff release-1.2.0
# Eliminar la rama de release porque
↳ ya no es necesaria
git branch -d release-1.2.0
```

# Hotfix



```
# Crear una nueva rama hotfix desde
↳ main
git switch -c hotfix-2.5.1 main
# Corregir el bug y registrar los
↳ cambios en un commit
git commit -am 'Fix bug'
# Actualizar archivos para reflejar
↳ el cambio de versión
git commit -am 'bump version 2.5.1'
# Cambiar a la rama main para
↳ fusionar los cambios del hotfix
git switch main
# Fusionar la rama hotfix en main
↳ con un commit dedicado
git merge --no-ff hotfix-2.5.1
# Hacer lo mismo en develop
git switch develop
git merge --no-ff hotfix-2.5.1
# Borrar la rama
git branch -d hotfix-2.5.1
```

GitHub Flow se basa en la creación de **Pull Requests**, que serán revisadas y discutidas antes de ser integradas en la rama principal (**main**). Es una estrategia ideal para proyectos de código abierto, ya que cualquier persona puede proponer cambios que serán aceptados o rechazados tras revisión.

## Tipos de ramas:

- **main**: La rama principal, estable y lista para producción.
- Otras ramas: Creadas temporalmente para implementar cambios y luego integrarse en **main**.

Trunk Based Development se basa en el trabajo directo sobre la rama principal (**main** o **trunk**). Los desarrolladores crean ramas temporales de corta duración, que integran frecuentemente en la rama principal para evitar conflictos grandes.

## Características principales:

- **main** o **trunk**: La rama principal y siempre estable.
- Las ramas de características son de corta duración, idealmente no más de un día o unas pocas horas.
- Integración continua: Se integran cambios frecuentemente en la rama principal.

Ship/Show/Ask es un enfoque de desarrollo ágil que se enfoca en lanzar cambios de manera continua y rápida para obtener retroalimentación temprana de los usuarios.

## Pasos del flujo de trabajo:

- **Ship:** Desarrollar y lanzar el cambio rápidamente.
- **Show:** Usamos pull requests para integrar cambios, pero no esperamos revisiones manuales. En lugar de eso, esperamos a que los tests automatizados pasen las pruebas para garantizar que los cambios no rompan el sistema.
- **Ask:** Solicitar retroalimentación para evaluar el impacto y mejorar el producto.

Este flujo de trabajo es ideal para equipos que necesitan entregar características rápidamente y ajustar según el feedback del usuario.

**Conventional Commits** es una especificación para escribir mensajes de commit que sigan un formato consistente y semántico, facilitando la comprensión y automatización en los flujos de trabajo.

## Formato básico:

`<tipo>(<área opcional>): <descripción breve>`

`<cuerpo opcional>`

`<footer opcional>`

# Hooks en Git

Los **Git Hooks** son scripts que se ejecutan automáticamente en respuesta a eventos específicos de Git, como:

- **pre-commit**: Antes de que se cree un commit.
- **pre-push**: Antes de enviar cambios al repositorio remoto.
- **commit-msg**: Al escribir el mensaje del commit.

## ¿Por qué usar hooks?

- Garantizar la calidad del código mediante linters o formateadores como **Prettier**.
- Ejecutar tests automáticamente para evitar errores en el código.
- Asegurar convenciones de estilo, como **Conventional Commits**.

Los hooks permiten automatizar tareas clave y mejorar la colaboración en equipos.



## Tipos más comunes:

- **feat**: Introducción de una nueva funcionalidad.
- **fix**: Corrección de un bug.
- **docs**: Cambios en la documentación.
- **style**: Cambios de formato (espacios, comas, etc.).
- **refactor**: Cambios en el código que no corrigen bugs ni agregan funcionalidades.
- **test**: Adición o modificación de tests.
- **chore**: Cambios en tareas de construcción o herramientas.

Este enfoque ayuda a generar automáticamente changelogs y facilita el entendimiento del historial del proyecto.

---

# Hooks en Git

Los **Git Hooks** son scripts que se ejecutan automáticamente en respuesta a eventos específicos de Git, como:

- **pre-commit**: Antes de que se cree un commit.
- **pre-push**: Antes de enviar cambios al repositorio remoto.
- **commit-msg**: Al escribir el mensaje del commit.

## ¿Por qué usar hooks?

- Garantizar la calidad del código mediante linters o formateadores como **Prettier**.
- Ejecutar tests automáticamente para evitar errores en el código.
- Asegurar convenciones de estilo, como **Conventional Commits**.

Los hooks permiten automatizar tareas clave y mejorar la colaboración en equipos.

# Usando Husky para gestionar hooks

Husky es una herramienta que simplifica la configuración y gestión de hooks en proyectos Git.

## Configuración básica de Husky:

```
# Instalar Husky
npm install husky --save-dev

# Habilitar hooks en el proyecto
npx husky install

# Crear un hook para ejecutar linters antes de un commit
npx husky add .husky/pre-commit "npm run lint"

# Crear un hook para validar mensajes de commits
npx husky add .husky/commit-msg "npm run commitlint --edit $1"
```

Con Husky, puedes garantizar que los linters, tests y convenciones de commits se ejecuten automáticamente.

# Configurando nuestro repositorio

Para empezar a configurar un nuevo proyecto en **Node.js** y establecer buenas prácticas de desarrollo, sigue estos pasos:

- Inicializa un nuevo proyecto con **npm**:

```
npm init
```

- Instala **ESLint** para analizar el código y asegurarte de que sigue las mejores prácticas:

```
npm install eslint --save-dev
```

- Instala **Prettier** para formatear el código de forma consistente:

```
npm install prettier --save-dev
```

- Configura **Commitlint** para asegurar que los mensajes de los commits siguen un estándar:

```
npm install @commitlint/{config-conventional,cli} --save-dev
```

Para proyectos en **Python**, considera usar **pycodestyle** como linter y **autopep8** como formatter.

# Configurando nuestro repositorio (Continuación)

- Instala **lint-staged** para ejecutar linters en los archivos modificados antes de hacer un commit:

```
npm install --save-dev lint-staged
```

- Instala **Husky** para añadir *git hooks* y automatizar tareas, como ejecutar linters antes de cada commit:

```
npm install husky --save-dev
```

- Instala **semantic-release** para gestionar el versionado semántico de tu proyecto:

```
npm install --save-dev semantic-release
npm install --save-dev @semantic-release/changelog
↪ @semantic-release/commit-analyzer @semantic-release/git
↪ @semantic-release/github @semantic-release/npm
↪ @semantic-release/release-notes-generator
```

Estos pasos ayudan a mantener un código limpio y fácil de mantener.

# Configurando las herramientas

Una vez instaladas las dependencias, es necesario configurarlas para integrarlas correctamente en nuestro proyecto:

- Configurar **ESLint**:

```
npx eslint --init
```

Esto generará un archivo `.eslintrc.json` con la configuración básica.

- Configurar **Prettier**:

```
echo {} > .prettierrc
```

- Configurar **lint-staged**:

```
echo {} > .lintstagedrc
```

- Configurar **Husky** para usar los hooks:

```
npx husky init
```

# Configurando las herramientas

- Crea un archivo de configuración `commitlint.config.js` en la raíz del proyecto:

```
echo "export default { extends:  
  ↳ ['@commitlint/config-conventional'] };" >  
  ↳ commitlint.config.js
```

- Integra **Commitlint** con **Husky** para que valide automáticamente los mensajes de commit:

```
npx husky add .husky/commit-msg 'npx --no-install commitlint  
  ↳ --edit "$1"'
```

# Configurando las herramientas

```
C:\Users\ArturoSilvelo\cid [main ↑1] > git commit -m 'Commit sin convetional commits'
→ No staged files match any configured task.
✖ input: Commit sin convetional commits
✖ subject may not be empty [subject-empty]
✖ type may not be empty [type-empty]

✖ found 2 problems, 0 warnings
💡 Get help: https://github.com/conventional-changelog/commitlint/#what-is-commitlint

husky - commit-msg script failed (code 1)
C:\Users\ArturoSilvelo\cid [main ↑1 +0 ~1 -0 ~] > █
```



# Configurando las herramientas

- Para personalizar el comportamiento de **standard-version**, crea un archivo **.releaserc.json** en la raíz de tu proyecto.
- Aquí tienes un ejemplo básico de configuración:

```
{  
  "branches": ["main"],  
  "plugins": [  
    "@semantic-release/commit-analyzer",  
    "@semantic-release/release-notes-generator",  
    "@semantic-release/changelog",  
    "@semantic-release/git",  
    "@semantic-release/github"  
  ]  
}
```

- En este archivo, puedes configurar:
  - **branches**: las ramas en las que se va a gestionar el versionado.
  - **plugins**: los plugins que manejarán el análisis de commits, generación de changelog, gestión de tags, etc.

# Configurando acciones en GitHub

- Las **GitHub Actions** te permiten automatizar tareas como la ejecución de tests, linters y despliegues dentro de tu flujo de trabajo en GitHub.
- Para configurarlas, crea un archivo `.github/workflows/ci.yml` en tu repositorio.
- Aquí tienes un ejemplo básico de configuración para un flujo de trabajo que ejecuta linting y tests en cada push a la rama **main**:
- Este archivo define un flujo de trabajo que se ejecuta cuando hay un **push** a la rama **main**, instala las dependencias del proyecto, ejecuta el linter y luego corre los tests.

# Configurando acciones en GitHub

```
name: CI
on:
  push:
    branches:
      - main
jobs:
  lint-and-test:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v2
      - name: Set up Node.js
        uses: actions/setup-node@v2
        with:
          node-version: '14'
      - name: Install dependencies
        run: npm install
      - name: Lint code
        run: npm run lint
      - name: Run tests
        run: npm test
```

# Trucos

---

# Git Cherry-pick

**Git Cherry-pick** permite aplicar un commit específico de otra rama a la rama actual, sin necesidad de fusionar toda la rama. Es útil para incorporar cambios puntuales sin mezclar todas las modificaciones de una rama.

## Uso básico:

- Identifica el hash del commit que quieres aplicar (usualmente con `git log`).
- Ejecuta el comando:  
`git cherry-pick <commit-hash>`

**Ejemplo:** Si quieres aplicar el commit con el hash **abc1234** de una rama **feature** a tu rama actual, usarías:

```
git cherry-pick abc1234
```

**Nota:** Si hay conflictos durante el cherry-pick, tendrás que resolverlos manualmente antes de continuar con el proceso.

**Git Bisect** es una herramienta que permite encontrar un commit específico que introdujo un error en el código utilizando una búsqueda binaria.

## Uso básico:

- Inicia el bisecting con el comando:  
`git bisect start`
- Marca un commit bueno y uno malo:  
`git bisect good <commit-hash>`  
`git bisect bad <commit-hash>`

Git hará una búsqueda binaria y te pedirá que marques cada commit como bueno o malo hasta encontrar el commit defectuoso.

**Ejemplo de uso:** Si sabes que el último commit estaba funcionando y el actual no, puedes usar:

```
git bisect start  
git bisect good <last-good-commit>  
git bisect bad <current-commit>
```

**Finaliza el bisecting:** Para salir del modo bisecting y volver al estado original:

```
git bisect reset
```

## Volver a la rama previa

En Git, para volver a la rama en la que estuviste trabajando antes de la rama actual, puedes utilizar dos métodos:

- Usar el comando **git switch -**:

```
git switch -
```

- Usar **@-1** para la misma funcionalidad:

```
git switch @{-1}
```

Ambos comandos permiten cambiar de vuelta a la rama donde estabas antes de hacer el último cambio, sin necesidad de recordar su nombre.



En Git, puedes habilitar una funcionalidad de auto-corrección para evitar errores tipográficos en los comandos. Esto se puede lograr con el siguiente comando:

- **Habilitar el auto-corrector:**

```
git config --global help.autocorrect 20
```

El valor **20** indica el número de décimas de segundo que Git esperará antes de intentar corregir automáticamente un error de comando. Si se introduce un comando incorrecto, Git lo corregirá automáticamente después de este tiempo. Puedes ajustar el valor según tus necesidades.

## Bibliografía

---

Para profundizar en el uso de Git, recomendamos los siguientes recursos:

- **Aprendiendo Git: ¡Domina y comprende Git de una vez por todas!** Un recurso práctico y completo para entender y dominar Git. Disponible en:  
<https://leanpub.com/aprendiendo-git>
- **Pro Git (oficial):** Libro oficial de Git, disponible de forma gratuita en múltiples idiomas. <https://git-scm.com/book/en/v2>