

Git - Introducción

Control de versiones y trabajo en equipo



EUSKO JAURLARITZA
GOBIERNO VASCO

LAN ETA ENPLEGU
SAILA
DEPARTAMENTO DE TRABAJO
Y EMPLEO








EUROPAR BATASUNA
Europako Gizarte Funtzia
EGFk zure etorkizunean inbertitzen du

UNION EUROPEA
Fondo Social Europeo
(El FSE) invierte en tu futuro



Cuántas veces ha pasado esto

| | | |
|---|-----------------------------------|---------------------------------|
|  | proyecto.docx | Última |
| | modificación: 29/10/2021 | |
|  | proyecto-version-final.docx | Última modificación: |
| | 30/10/2021 | |
|  | proyecto-version-final-final.docx | Última modificación: 29/10/2024 |
|  | proyecto-v2.docx | Última |
| | modificación: 08/09/2023 | |
|  | proyecto-v2-final.docx | Última modificación: |
| | 15/02/2024 | |



Cuántas veces ha pasado esto



proyecto.docx

modificación: 29/10/2021



proyecto-version-final.docx

30/10/2021



proyecto-version-final-final.docx



proyecto-v2.docx

modificación: 08/09/2023



proyecto-v2-final.docx

15/02/2024

Última

Última modificación:

Última modificación: 29/10/2024

Última

Última modificación:



Qué es Git

Git es un sistema de control de versiones distribuido, usado para gestionar y seguir el historial de cambios de un proyecto.

Su principal ventaja es que facilita la colaboración y permite revertir los cambios cuando sea necesario.





Sincronizar con repositorios en la nube

GitHub, GitLab y BitBucket son plataformas que alojan repositorios y facilitan la colaboración en proyectos.

Estas plataformas ofrecen herramientas de control de acceso, colaboración e integración y despliegue continuo (CI/CD).

CI/CD = Continuous Integration / Continuous Delivery/Deployment



Git NO es GitHub



Git

VS



GitHub

Lectura
Escritura



Dani



Portátil Dani



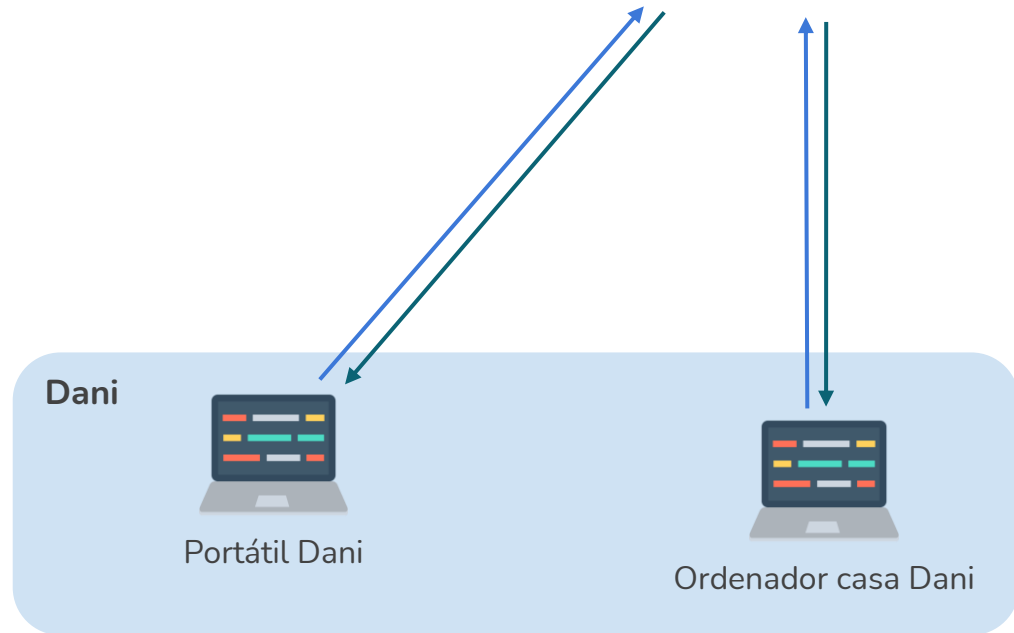
Ordenador casa Dani

Nahikari

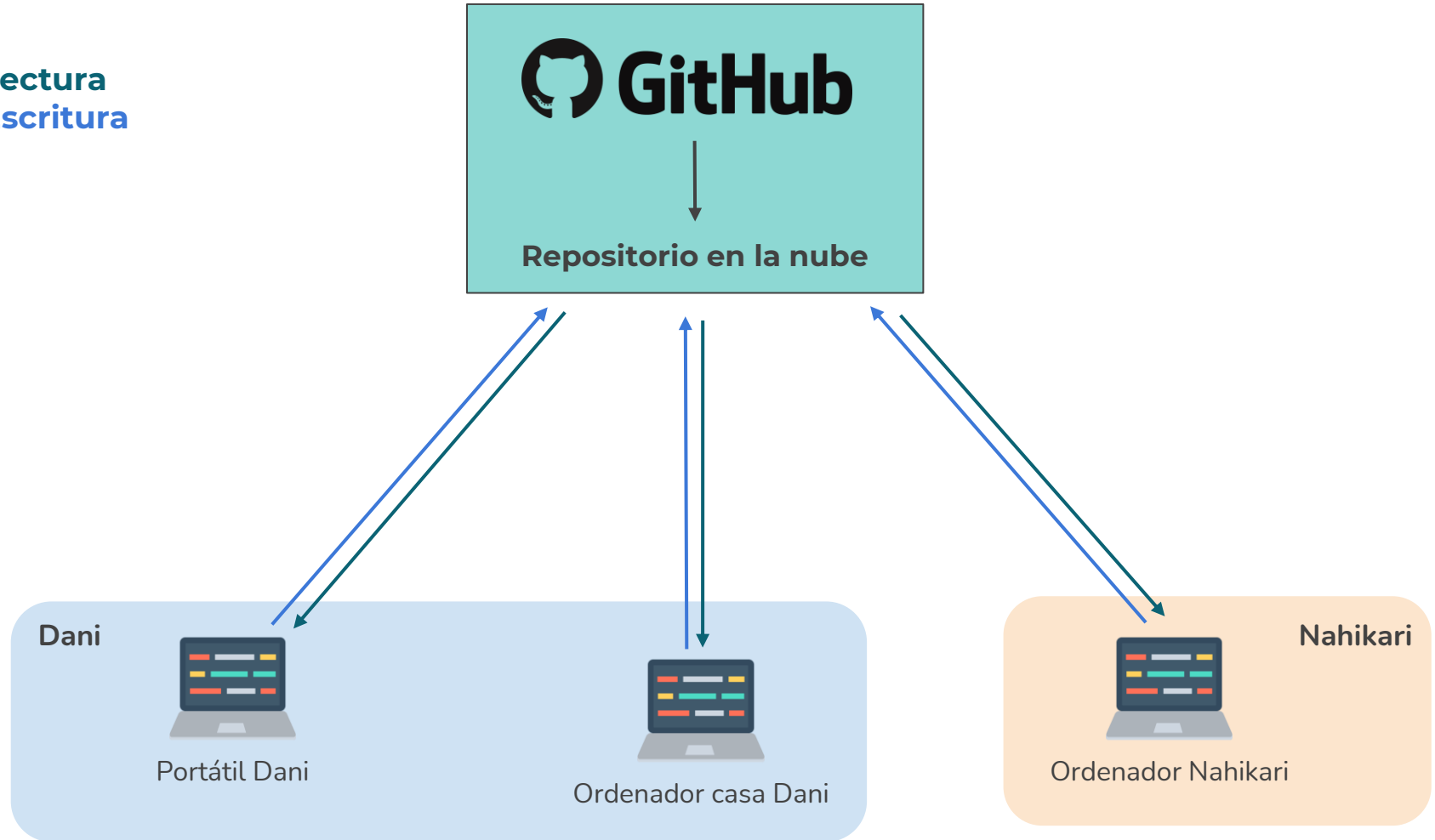


Ordenador Nahikari

Lectura
Escritura



Lectura
Escritura





Por qué Git es **esencial**

- **Proporciona un historial de cambios**, facilitando muchísimo el seguimiento del progreso.
- **Permite una colaboración eficiente**, permitiendo que múltiples desarrolladores trabajen en paralelo.
- **Aporta seguridad y recuperación**, ya que permite restaurar versiones previas en caso de errores.
- **Permite asignar permisos a los miembros** (generalmente lectura y escritura) teniendo un mayor control.



Instalación y primeros comandos

- Descargar e instalar **Git** en nuestro ordenador: git-scm.com

```
laptop@laptop-Creator-M16-A12UC:~$ git
uso: git [-v | --version] [-h | --help] [-C <path>] [-c <name>=<value>]
      [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
      [-p | --paginate | -P | --no-pager] [--no-replace-objects] [--bare]
      [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
      [--config-env=<name>=<envvar>] <command> [<args>]
```



Instalación y primeros comandos

- Comandos más comunes

Estos son comandos comunes de Git usados en varias situaciones:

comenzar un área de trabajo (mira también: `git help tutorial`)

`clone` Clonar un repositorio dentro de un nuevo directorio

`init` Crear un repositorio de Git vacío o reinicia el que ya existe

trabajar en los cambios actuales (mira también: `git help everyday`)

`add` Agregar contenido de archivos al índice

`mv` Mover o cambiar el nombre a archivos, directorios o enlaces simbólicos

`restore` Restaurar archivos del árbol de trabajo

`rm` Borrar archivos del árbol de trabajo y del índice

examinar el historial y el estado (mira también: `git help revisions`)

`bisect` Usar la búsqueda binaria para encontrar el commit que introdujo el bug

`diff` Mostrar los cambios entre commits, commit y árbol de trabajo, etc

`grep` Imprimir las líneas que concuerden con el patrón

`log` Mostrar los logs de los commits

`show` Mostrar varios tipos de objetos

`status` Mostrar el estado del árbol de trabajo



Instalación y primeros comandos

- Comandos más comunes

crecer, marcar y ajustar tu historial común

| | |
|--------|--|
| branch | Listar, crear, o borrar ramas |
| commit | Grabar los cambios al repositorio |
| merge | Juntar dos o más historiales de desarrollo juntos |
| rebase | Volver a aplicar commits en la punta de otra rama |
| reset | Reiniciar el HEAD actual a un estado específico |
| switch | Cambiar de branch |
| tag | Crear, listar, borrar o verificar un objeto de tag firmado con GPG |

colaborar (mira también: `git help workflows`)

| | |
|-------|--|
| fetch | Descargar objetos y referencias de otro repositorio |
| pull | Realizar un fetch e integra con otro repositorio o rama local |
| push | Actualizar referencias remotas junto con sus objetos asociados |

'`git help -a`' y '`git help -g`' listan los subcomandos disponibles y algunas guías de concepto. Consulta '`git help <command>`' o '`git help <concepto>`' para leer sobre un subcomando o concepto específico.
Mira '`git help git`' para una vista general del sistema.



Instalación y primeros comandos

- Descargar e instalar **Git** en nuestro ordenador: git-scm.com
- Después de instalar, abrimos una consola y ejecutamos los siguientes comandos:
 - `git config --global user.name "Tu Nombre"`
 - `git config --global user.email "tuemail@ejemplo.com"`

Con estos comandos estaremos configurando “la firma” que se va a aplicar a cada conjunto de cambios que añadamos.

Podremos consultar el valor introducido

```
git config --get user.name  
git config --get user.email
```

- Vamos a utilizar **GitHub**, y para ello necesitamos una cuenta. Accedemos GitHub y nos registramos: <https://github.com/>

Git - Básico

Comandos básicos y nuestro primer repositorio



EUSKO JAURLARITZA
GOBIERNO VASCO

LAN ETA ENPLEGU
SAILA
DEPARTAMENTO DE TRABAJO
Y EMPLEO



EUROPAR BATASUNA
Europako Gizarte Funtzia
EGFk zure etorkizunean inbertitzen du

UNION EUROPEA
Fondo Social Europeo
(El FSE) invierte en tu futuro



Conceptos básicos

- Un **conjunto de cambios** es un grupo de modificaciones listos para guardarse en el historial (hacer un commit del conjunto de cambios).
- Un **repositorio** es la carpeta contenedora del proyecto y su historial de cambios.
- Las **ramas** representan líneas independientes de desarrollo dentro de un repositorio.
- **Remote y origin.**
Remote se refiere a una versión alojada en la nube, como la que se encuentra en GitHub, nos permite sincronizar nuestro trabajo local con ese repositorio en la nube.

Por defecto git le asigna al remote principal (puedes tener varios) el alias **origin**, esto nos permitirá trabajar con el remote a través del alias sin tener que especificar toda la URL.

```
git push https://github.com/danieltamargo/aprendiendo-juntos.git  
git push origin
```

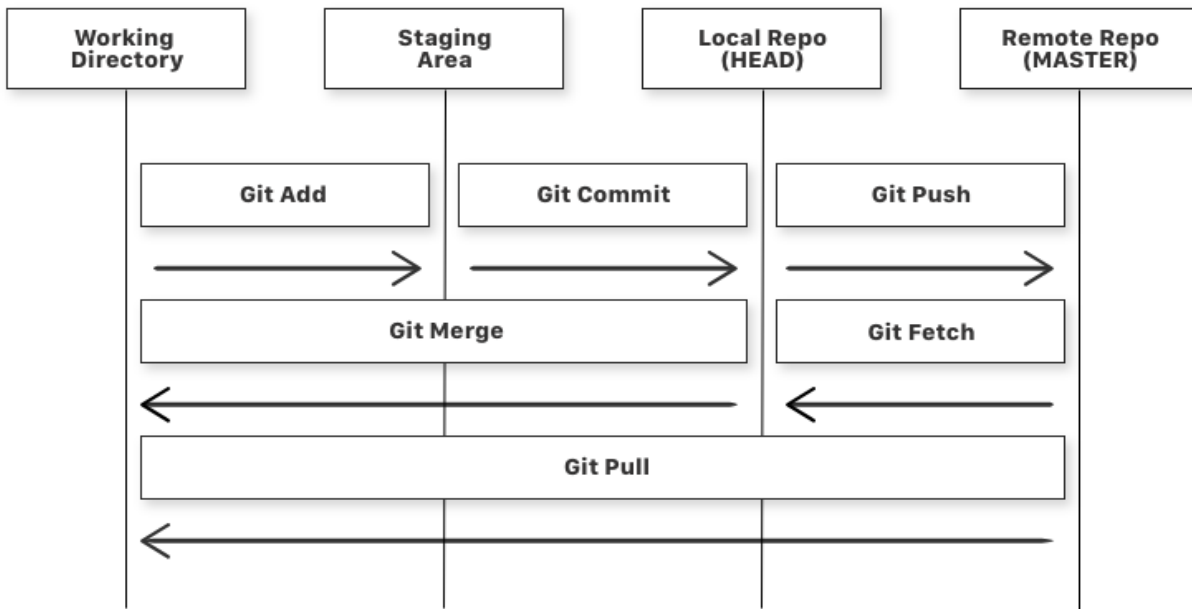



Conceptos básicos

- **Working Directory:** Donde editas archivos. Cambios aún no guardados en Git.
- **Staging Area:** Zona de preparación. Con `git add`, preparas cambios para el commit.
- **Local Repository:** Guarda los cambios en tu historial local con `git commit`.
- **Remote:** El repositorio en la nube. Subes los cambios con `git push origin`.
- **HEAD:** Es el puntero que señala al commit actual en el que estás trabajando. Cambia de posición cuando te mueves entre ramas o haces nuevos commits.



Conceptos básicos



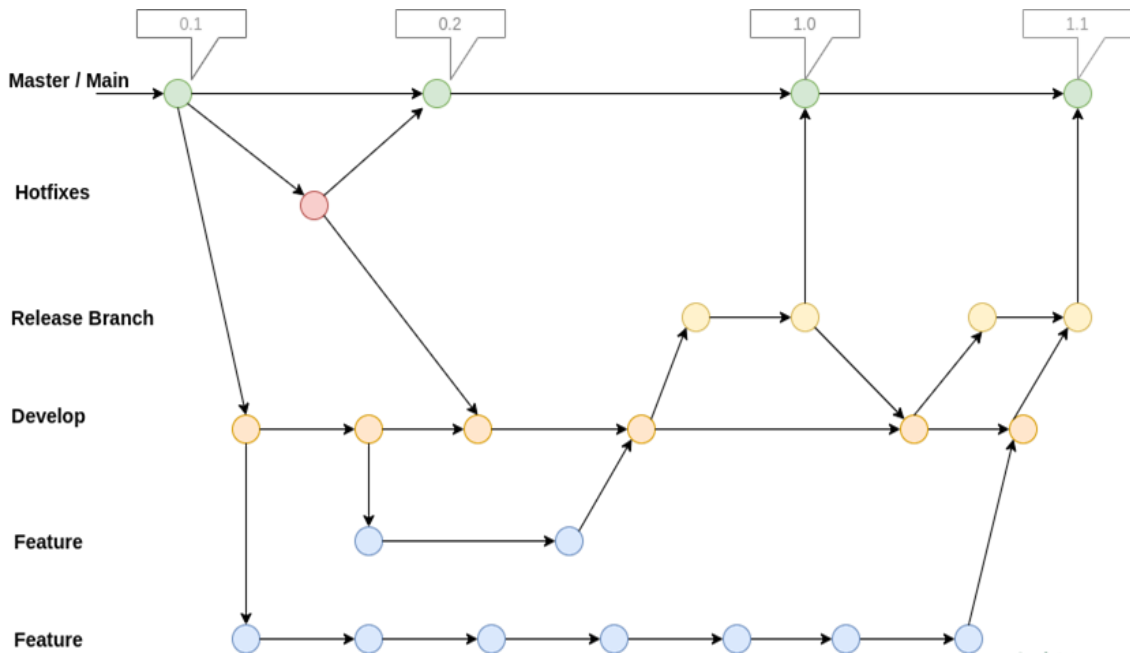


Visualizando las ramas

Las **ramas** nos permiten trabajar en líneas independientes, las cuales luego podremos juntar.

A partir de un commit concreto de una rama podremos crear nuevas ramas, donde trabajaremos para desarrollar funcionalidades (features) para luego volver a unirlos.

Una metodología de trabajo con git muy común y conocida es la llamada **git flow**, que da un enfoque ágil al desarrollo.

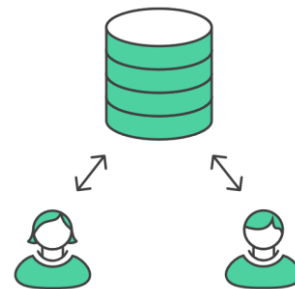




Comandos básicos I

Configuración de un repositorio

- **git init** → inicializa un nuevo repositorio
- **git clone** → clonar un repositorio
 - `git clone <url_repositorio> <nombre_carpeta>`
- **git config** → aplicar configuraciones de git por repositorio o globales
 - `git config --global user.name "Tu Nombre"`
 - `git config --global ui.color auto`
 - `git config ui.color auto`



Cuando hablamos de globales nos referimos a configuraciones globales en nuestra máquina, es decir, que afectarán a todos los repositorios donde no se haya especificado esa configuración.



Comandos básicos II

Guardado y visualización de cambios

- **git add** → añadir archivos al conjunto de cambios:
 - `git add <archivo>` # Añadir sólo los cambios de un fichero
 - `git add .` # Añadir todos los cambios de la ruta actual
- **git commit** → guardar el conjunto de cambios en el historial del proyecto:
 - `git commit -m "Descripción del cambio"`
 - `git commit --amend "Nueva descripción"` # Modificar la descripción del último commit
- **git diff** → ver lista de cambios pendientes de añadir
- **git status** → muestra el estado del repositorio (si hay ficheros modificados, o añadidos pero no commiteados, etc)
- **git log** → ver historial de commits
 - `git log`
 - `git log --graph --oneline --decorate`

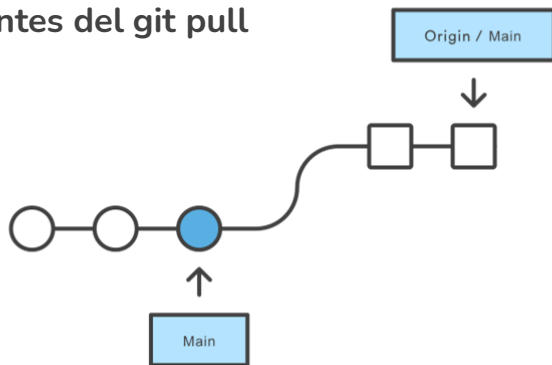
Comandos básicos III

Sincronización con un remoto

- **git fetch** → descarga el historial de commits del repositorio remoto a tu repositorio local, pero sin mezclar esos cambios con tu trabajo actual. Básicamente es como mirar si hay actualizaciones en la nube que todavía no estén en tu proyecto local.
 - **git fetch <remote>** # Recupera todas las ramas del remoto
específico
 - **git fetch <remote> <branch>** # Recupera la rama específica del remoto
específico
 - **git fetch --all** # Recupera todos los
repositorios remotos y sus ramas
 - **git fetch --dry-run** # Demo del comando sin aplicarlo, como una
preview
- **git pull** → descarga la lista de los cambios pendientes del remoto (fetch) y los une (merge) en la rama local
 - **git pull** # Descarga los cambios del
remote principal y la rama en la que estés
 - **git pull <remote> <branch>** # Especificar de qué remote y qué rama
descargar los cambios
- **git push** → cargar (subir) el contenido del repositorio local a un repositorio remoto
 - **git push <remote> <branch>**
 - **git push <remote> <branch> --force** # ¡Cuidado! ⚠ Este comando fuerza el sobrescribir el remote

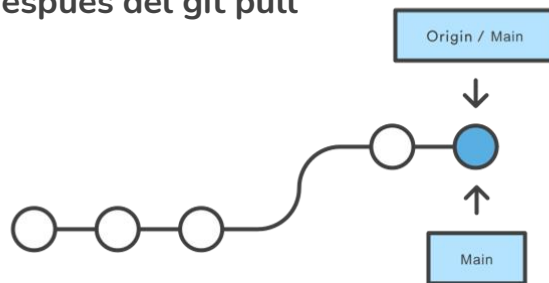
Visualizando git pull

Antes del git pull



Hay cambios en el remoto que no tenemos en local, por lo que queremos obtenerlos para trabajar sobre la última versión, evitando posibles conflictos innecesarios.

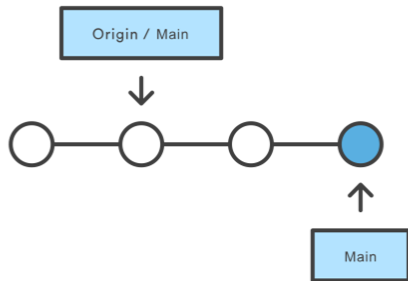
Después del git pull



Ya estamos sobre la última versión, podremos trabajar sin preocuparnos.

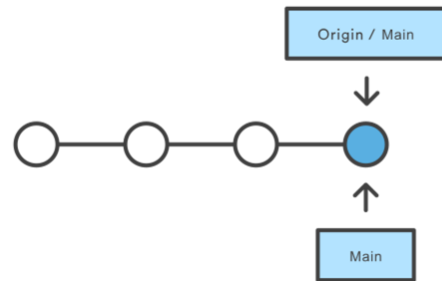
Visualizando git push

Antes del git push



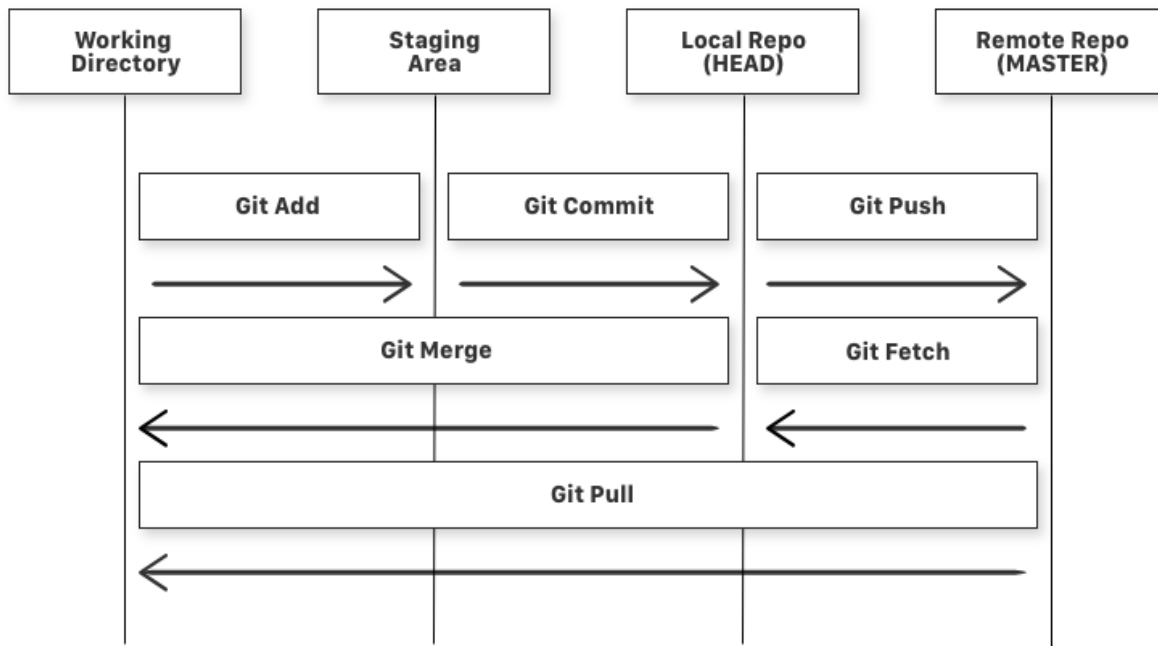
Nuestra rama **main** en **local** tiene una serie de **commits** que no están sincronizados con el repositorio **remoto**.

Después del git push



Los cambios se han **subido** y la **versión** actual del repositorio remoto en la rama main ya tiene nuestros últimos conjuntos de cambios.

Volvemos a ver el esquema





Creando alias para los comandos

A veces tendemos a ejecutar ciertos comandos de manera habitual y puede ser tedioso escribirlos completos una y otra vez.

Git nos permite crear alias para crear nuestros propios comandos.

Por ejemplo, creamos el alias **log-bonito**:

```
git config --global alias.log-bonito "log --graph --oneline --decorate"
```

Y así en vez de ejecutar el comando completo, podríamos utilizar directamente **git log-bonito**

Git - Intermedio

Trabajar con ramas y deshacer cambios



EUSKO JAURLARITZA
GOBIERNO VASCO

LAN ETA ENPLEGU
SAILA
DEPARTAMENTO DE TRABAJO
Y EMPLEO



EUROPA BATASUNA
Europako Gizarte Funtzia
EGFk zure etorkizunean inbertitzen du

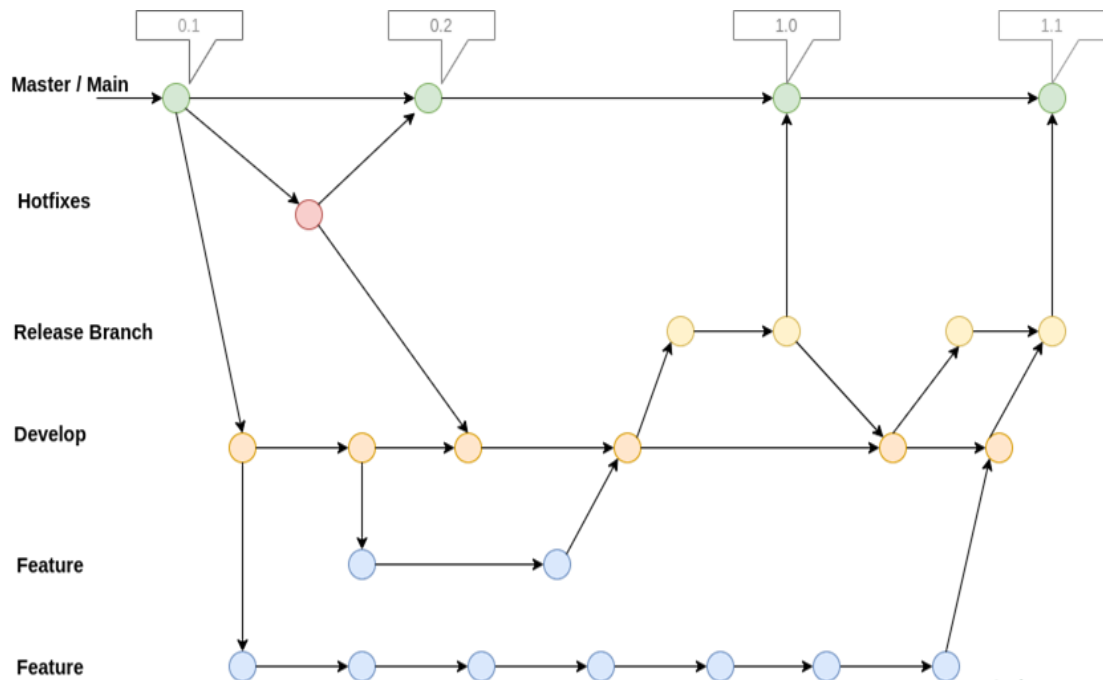
UNION EUROPEA
Fondo Social Europeo
(El FSE) invierte en tu futuro

Visualizando las ramas (de nuevo)

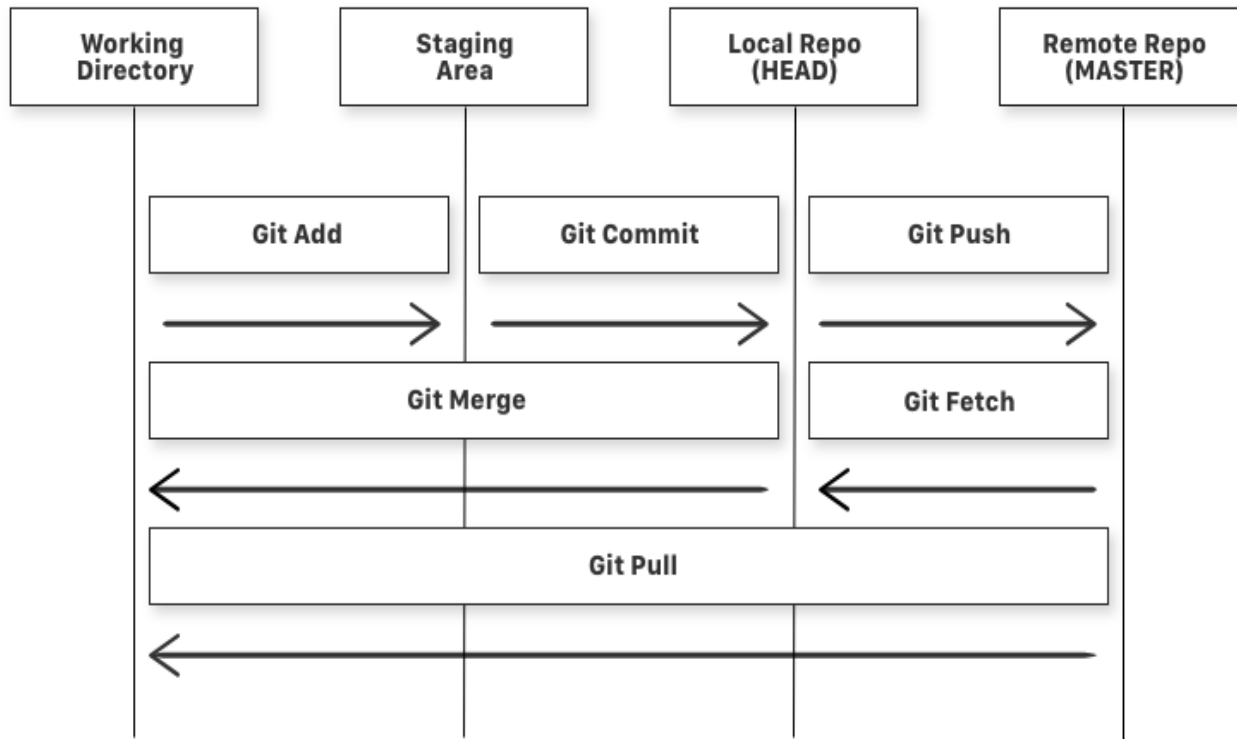
Las **ramas** nos permiten trabajar en líneas independientes, las cuales luego podremos juntar.

A partir de un commit concreto de una rama podremos crear nuevas ramas, donde trabajaremos para desarrollar funcionalidades (features) para luego volver a unir las.

Una metodología de trabajo con git muy común y conocida es la llamada **git flow**, que da un enfoque ágil al desarrollo.



Volvemos a ver el esquema



Comandos intermedios I

Crear ramas y cambiar de una a otra

- **git branch <nombre>** → crear una rama (si ya existe mostrará mensaje de error y no pasará nada)
- **git checkout** → moverse a un commit o rama
 - **git checkout <nombre_rama>**
 - **git checkout <commit_id>**
 - **git checkout -b <nombre_nueva_rama>**
- **git switch** → moverse entre ramas (similar a checkout pero más enfocado en las ramas)
 - **git switch <nombre_rama>**
 - **git switch -c <nombre_nueva_rama>**

Comandos intermedios II

Listar, eliminar y unir ramas

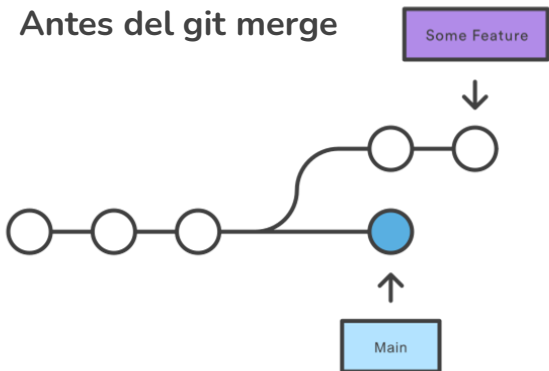
- **git branch** → listar ramas
 - **git branch** # Listar ramas locales
 - **git branch -r** # Listar ramas remotas
 - **git branch -a** # Listar todas las ramas (locales y remotas)
- **git branch -d <nombre-rama>** → eliminar una rama
 - **git branch -d <nombre_rama>** # Eliminar rama, dará error si no se han fusionado sus cambios
 - **git branch -D <nombre_rama>** # Eliminar rama y descartar los cambios sin fusionar
- **git merge** → unir ramas
 - **git merge <nombre-origen>** # Unir la rama en la que estás con la rama que indicas
 - **git merge --abort** # Cancelar el merge si ha dado conflictos y no queremos resolver
 - **git reset --merge** # Otra forma de intentar cancelar el merge si la primera falla

En main no han habido cambios desde que se creó la rama `some-feature`.

A diagram illustrating a linked list structure. It consists of a sequence of five circular nodes connected by a horizontal line. The first three nodes are white, and the last two are blue. A purple box labeled "Some Feature" has a downward arrow pointing to the first blue node. A light blue box labeled "Main" has an upward arrow pointing to the same first blue node.

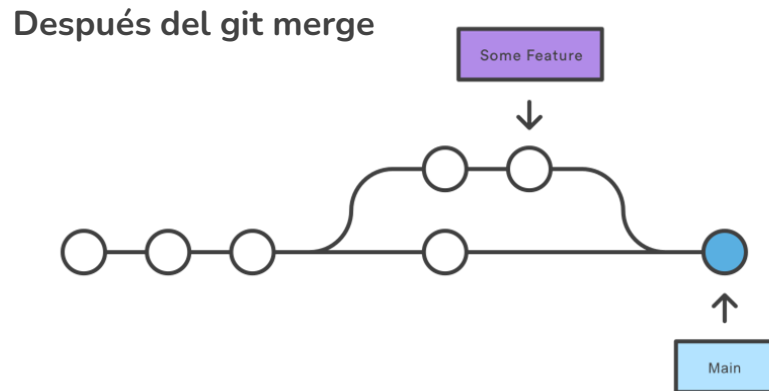
Puesto que no hay dos estados distintos que no puedan coexistir, no surgirán conflictos. A este tipo de merges se les conoce como **fast-forward merge** (merge de avance rápido)

Visualizando git merge



Tenemos una **rama** llamada **some-feature** con dos conjuntos cambios que no están presentes en **main**.

A su vez en **main** tenemos un conjunto de cambios que no está en la rama **some-feature**.



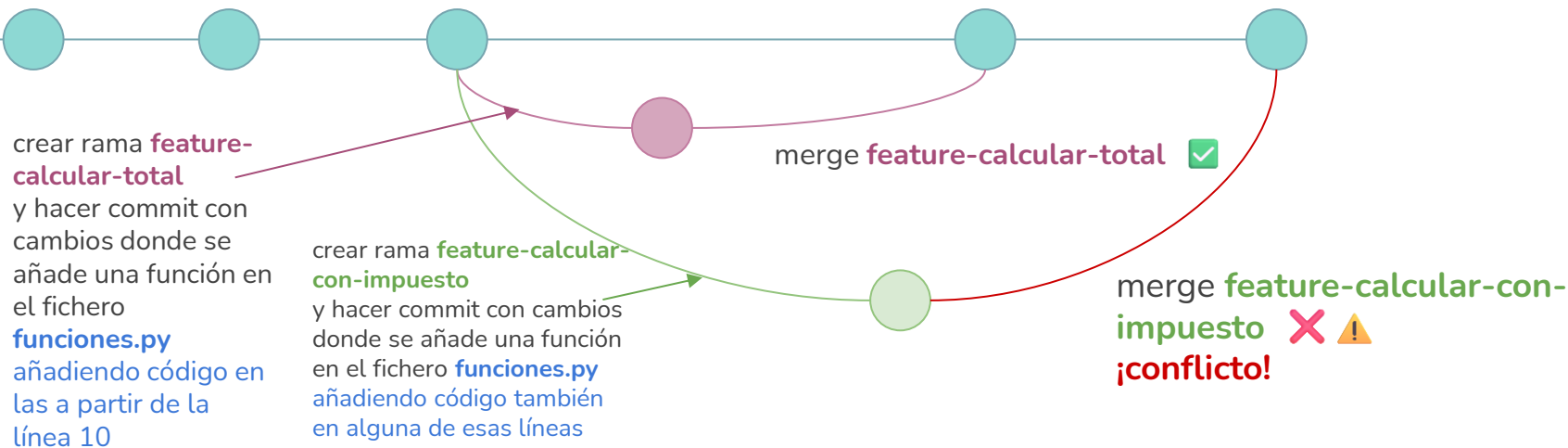
Nos situamos en main y le decimos que **fusiona los cambios de la rama some-feature** y ya tendríamos todos los conjuntos de cambios en la rama **main**.

Como han habido cambios en las dos ramas, **es posible que al unir surja algún conflicto** que deba ser resuelto.

Cuándo ocurren los conflictos

Cuando intentamos unir una rama y uno o varios conjuntos de cambios están modificando los mismos ficheros y las mismas líneas que otros cambios que han surgido desde la creación de la rama que estamos uniendo.

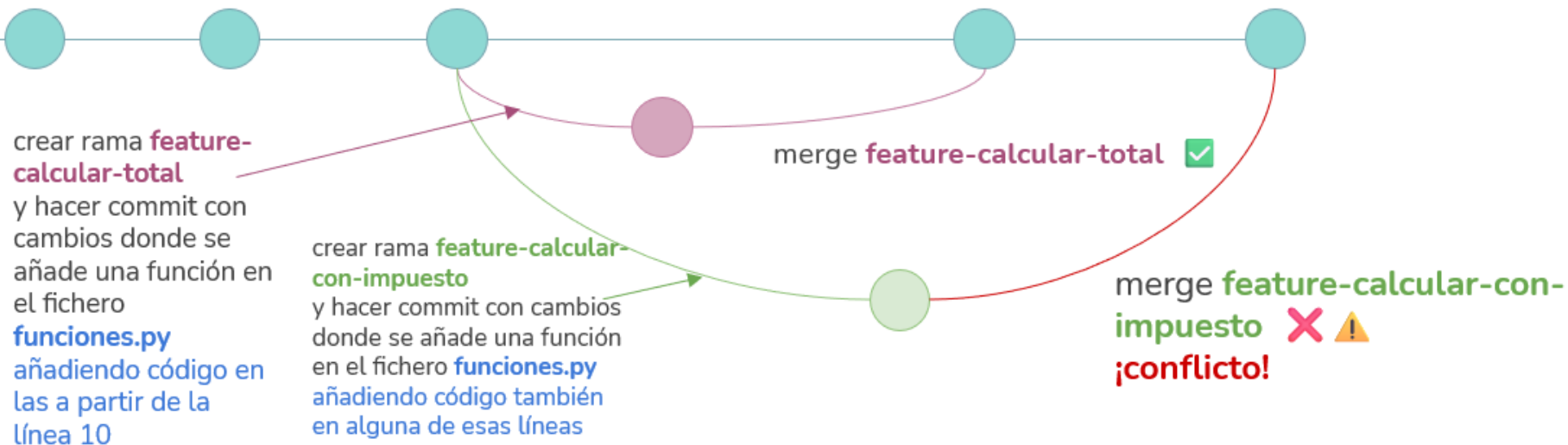
main



Cuándo ocurren los conflictos

Cuando intentamos unir una rama y uno o varios conjuntos de cambios están modificando los mismos archivos y las mismas líneas que otros cambios que han surgido desde la creación de la rama que estamos uniendo.

main



Git - Avanzado

Forks, pull requests, etc



EUSKO JAURLARITZA
GOBIERNO VASCO

LAN ETA ENPLEGU
SAILA
DEPARTAMENTO DE TRABAJO
Y EMPLEO



EUROPAR BATASUNA
Europako Gizarte Funtzia
EGFk zure etorkizunean inbertitzen du

UNION EUROPEA
Fondo Social Europeo
El FSE invierte en tu futuro

Conceptos avanzados



- Un **fork** es una copia completa de un repositorio en tu cuenta GitHub (u otra plataforma). Te permite desarrollar de forma independiente sin afectar el repositorio original. Generalmente los forks se usan para contribuir a proyectos de código abierto.
- Un **pull request** es una solicitud para fusionar los cambios de tu rama (o **fork**) con una rama del repositorio original. Es una forma de proponer una serie de conjunto de cambios y los mantenedores del proyecto podrán revisarlo, discutirlo y aceptarlos e integrarlos (o rechazarlos).
- **Stash** es una función que guarda temporalmente cambios que aún están en el working directory sin hacer un commit, permitiéndote cambiar de rama o hacer otras tareas sin perder tu trabajo. Es como sacar los cambios y guardarlos en un almacén temporal.

Git stash

- **git stash** → utilizar un almacén temporal donde guardar los cambios que aún no queremos confirmar
 - **git stash push -m "WIP: Añadir funcionalidad de filtrado"**
 - **git stash list** # Mostrar listado de stashes guardados
 - **git stash apply** # Aplicar el último stash **sin eliminarlo**
 - **git stash pop** # Aplicar **y eliminar** el último stash
 - **git stash apply stash@{2}** # Aplicar un stash específico usando su identificador (se ve con git stash list)
 - **git stash drop stash@{1}** # Eliminar un stash específico sin aplicarlo
 - **git stash clear** # Limpiar todos los stashes guardados



Fork y pull request

github.com/danieltamargo/tabler-icons

danieltamargo / tabler-icons

Code Pull requests Actions Projects Security Insights Settings

tabler-icons Public

Pin Watch 0 Fork 0

This branch is 1 commit ahead of, 341 commits behind tabler/tabler-icons:main

Contribute Sync fork

danieltamargo Added strokeWidth property to icons-vue SVGProps i... 96d9850 · last year 1,462 Commits

github.com/tabler/tabler-icons

tabler / tabler-icons

Code Issues 340 Pull requests 12 Discussions Actions Security Insights

tabler-icons Public

Sponsor Edit Pins Watch 105

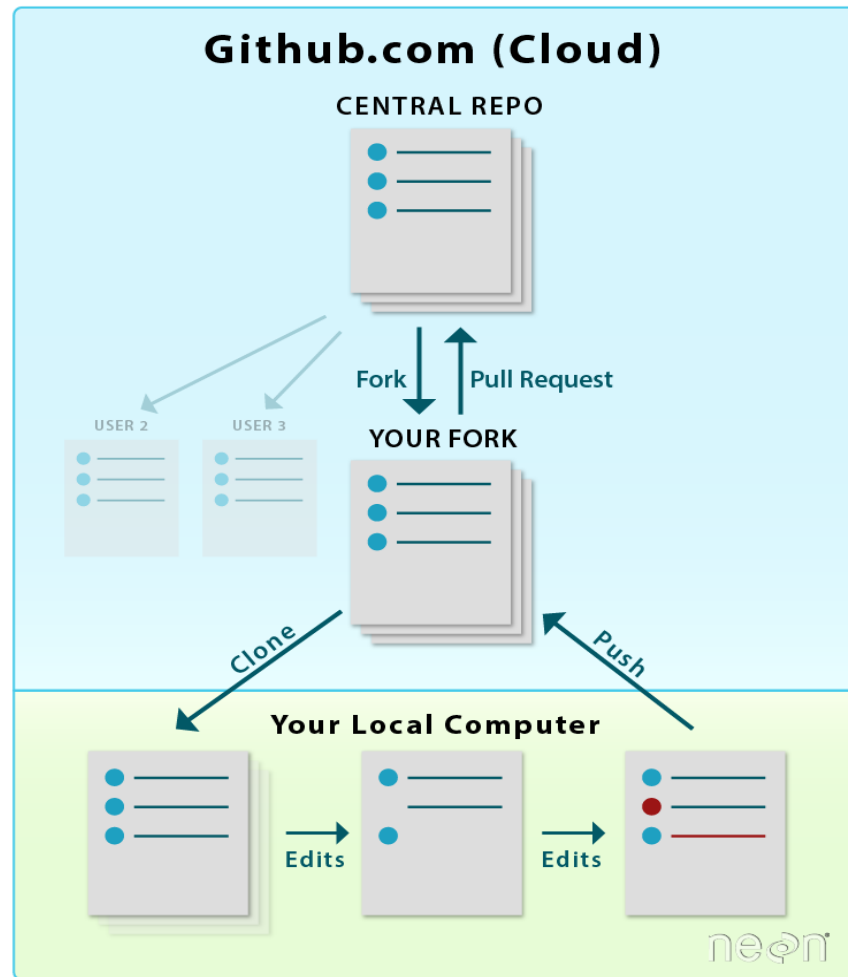
main 11 Branches 229 Tags

Go to file

codecalm Release 3.21.0 ✓ 6975214 · yesterday 1,802 Commits

| | | |
|------------|-------------------------------------|--------------|
| .build | build fix | last week |
| .github | Release 3.21.0 | yesterday |
| .vscode | Tabler Icons 3.0 (#993) | 8 months ago |
| docs/icons | Fix stylesheet webfont path (#1093) | 6 months ago |

Fork y pull request



Fichero .gitignore



El archivo **.gitignore** es un fichero especial que indica a Git qué archivos o directorios debe **ignorar** y no incluir en el control de versiones. Esto es útil para excluir archivos temporales, configuraciones locales, o datos sensibles que no quieres subir al repositorio.

```
# Ignorar archivos de configuración local
config/settings.json

# Ignorar todos los archivos .log
*.log

# Ignorar la carpeta de compilación
/build/
```

Podemos utilizar herramientas para preparar un fichero .gitignore muy útil, por ejemplo:

<https://www.toptal.com/developers/gitignore>



Otros conceptos que podrían ser interesantes

- **git hooks** (scripts que se ejecutan automáticamente en respuesta a ciertos eventos de Git)
- **cherry-pick** (copiar un commit específico de una rama y pegarlo en otra)
- **rebase** (juntar varios commits para tener un historial de commits más limpio)
- **remote upstream** (repositorio original del que hiciste un fork)
- **reflog** (histórico de todas las acciones realizadas en el repositorio, permite recuperar commits que ya no están accesibles desde ninguna referencia, puede usarse para recuperar un commit eliminado accidentalmente, guarda en el repositorio local un histórico durante 30 días)
- **clean** (elimina ficheros que no están rastreados ni se han añadido al staging area, es decir, archivos locales sin seguimiento en Git)
- **rm** (elimina ficheros que ya están rastreados o en el staging area, marcándolos para que se incluyan en el próximo commit y se eliminan también del repositorio al hacer push)
- **tag** (crear etiquetas, es decir, referencias, a puntos concretos en el historial de git)

Comandos intermedios III

Deshacer cambios

- `git reset --soft <commit>` → deshacer cambios manteniéndolos en el **staging area**
- `git reset --mixed <commit>` → deshacer cambios manteniéndolos en el **working**
directory
- `git reset --hard <commit>` → deshacer y descartar los cambios, **eliminandolos**
- `git reset <nombre_fichero>` → sacar los cambios de un fichero del **staging area** al
working directory
- `git reset` → sacar todos los cambios del **staging area** al
working directory

Ejemplos:

- `git reset --soft <commit_id>`
- `git reset --soft HEAD~1`
- `git reset --mixed HEAD~2`
- `git reset --soft <commit_id>`
- `git reset --hard <commit_id>`
- `git reset --hard`
irreversible

--mixed es el valor predeterminado

¡¡Cuidado!! **Es irreversible**

Elimina lo del staging area, ¡¡cuidado!! **Es**

Git reset puede ser confuso

Reset para los **commits**

- `git reset --soft <commit>` → deshacer cambios manteniéndolos en el **staging area**
- `git reset --mixed <commit>` → deshacer cambios manteniéndolos en el **working directory**
- `git reset --hard <commit>` → deshacer y descartar los cambios, **eliminándolos**

Reset para el contenido del **staging area**

- `git reset <nombre_fichero>` → sacar los cambios de un fichero del **staging area** al **working directory**
- `git reset` → sacar todos los cambios del **staging area** al **working directory**

Comandos intermedios IV

Revertir cambios

Mientras que con `git reset <commit>` estamos deshaciendo los cambios y modificando el historial de commits, con `git revert <commit>` estamos revirtiendo los cambios de un commit específico con un nuevo commit, es como crear un nuevo commit con la contraparte de esos cambios, haciendo el negativo de todo lo que hemos avanzado desde entonces .

- `git revert <commit>` → revertir los cambios creando un nuevo commit, sin modificar el historial

Ejemplo, teniendo los siguientes commits:

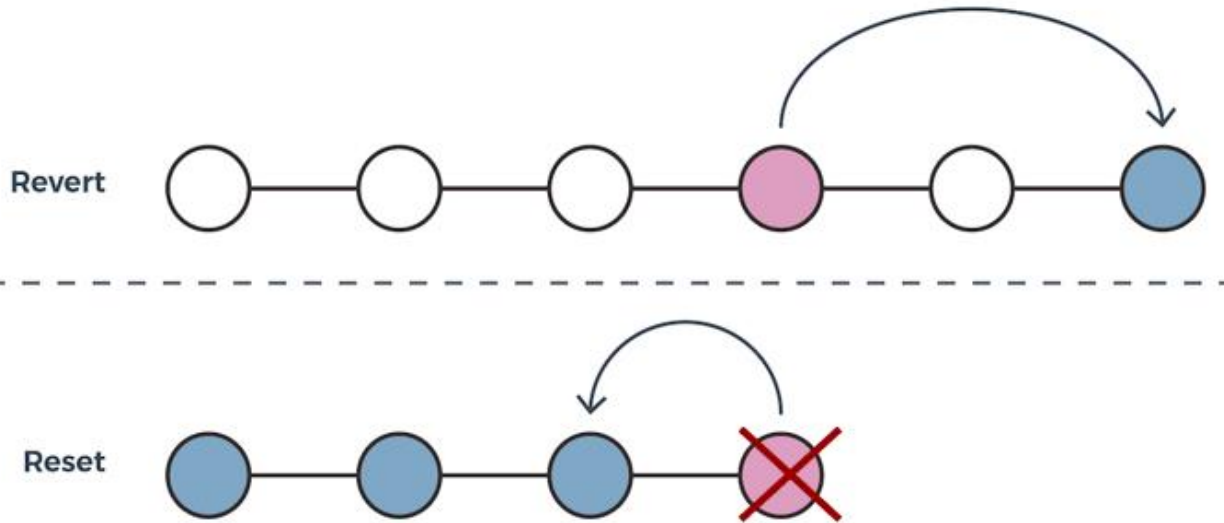


Ejecutamos: `git revert B`, y como resultado obtendremos:



Los cambios de **E** serán el negativo de los cambios de **B**

git reset vs git revert (visual)



Git - Avanzado

Forks, pull requests, etc



EUSKO JAURLARITZA
GOBIERNO VASCO

LAN ETA ENPLEGU
SAILA
DEPARTAMENTO DE TRABAJO
Y EMPLEO



EUROPAR BATASUNA
Europako Gizarte Funtzia
EGFk zure etorkizuneari inbertitzen du

UNION EUROPEA
Fondo Social Europeo
(El FSE) invierte en tu futuro

Conceptos avanzados



- Un **fork** es una copia completa de un repositorio en tu cuenta GitHub (u otra plataforma). Te permite desarrollar de forma independiente sin afectar el repositorio original. Generalmente los forks se usan para contribuir a proyectos de código abierto.
- Un **pull request** es una solicitud para fusionar los cambios de tu rama (o **fork**) con una rama del repositorio original. Es una forma de proponer una serie de conjunto de cambios y los mantenedores del proyecto podrán revisarlo, discutirlo y aceptarlos e integrarlos (o rechazarlos).
- **Stash** es una función que guarda temporalmente cambios que aún están en el working directory sin hacer un commit, permitiéndote cambiar de rama o hacer otras tareas sin perder tu trabajo. Es como sacar los cambios y guardarlos en un almacén temporal.

Git stash



- **git stash** → utilizar un almacén temporal donde guardar los cambios que aún no queremos confirmar
 - **git stash push -m "WIP: Añadir funcionalidad de filtrado"**
 - **git stash list** # Mostrar listado de stashes guardados
 - **git stash apply** # Aplicar el último stash **sin eliminarlo**
 - **git stash pop** # Aplicar **y eliminar** el último stash
 - **git stash apply stash@{2}** # Aplicar un stash específico usando su identificador (se ve con git stash list)
 - **git stash drop stash@{1}** # Eliminar un stash específico sin aplicarlo
 - **git stash clear** # Limpiar todos los stashes guardados

Fork y pull request

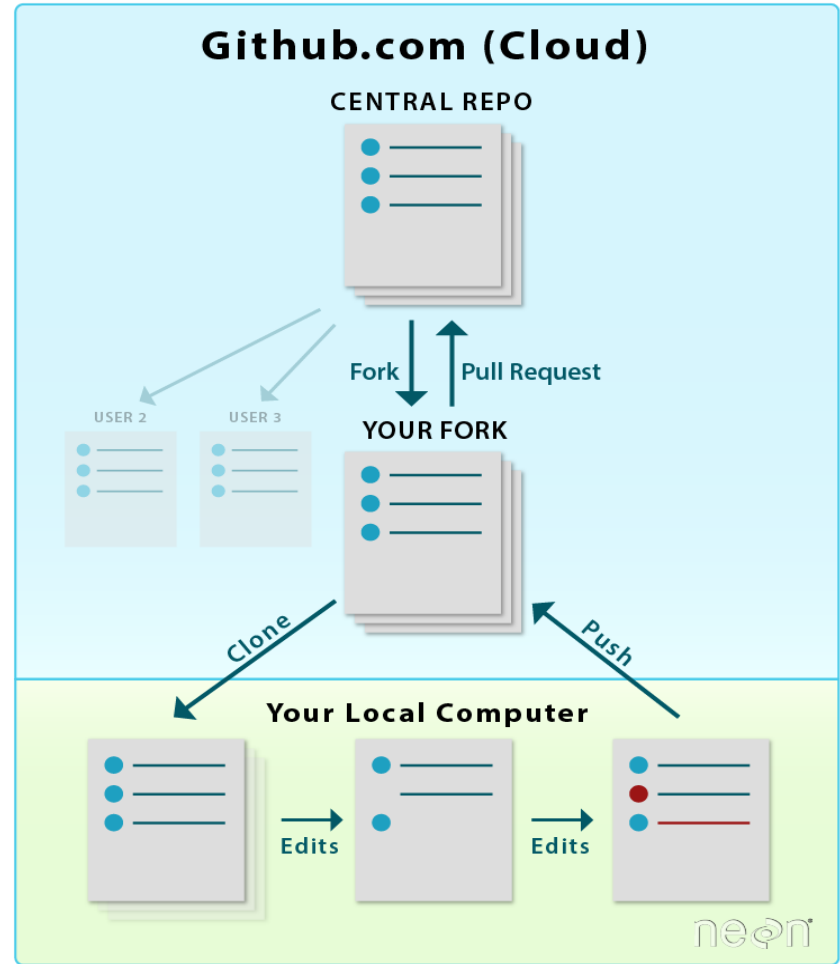
The diagram illustrates the workflow for forking a repository and creating a pull request on GitHub. It consists of two screenshots connected by a red arrow.

Top Screenshot (Original Repository): This screenshot shows the GitHub page for the `tabler / tabler-icons` repository. The repository is public and has 340 issues and 12 pull requests. The repository is owned by `codecalm` and has 11 branches and 229 tags. The repository is a fork of `tabler / tabler-icons`. The repository is public and has 105 watchers.

Bottom Screenshot (Forked Repository): This screenshot shows the GitHub page for the `danieltamargo / tabler-icons` repository, which is a fork of the original repository. The repository is public and has 0 issues and 0 pull requests. The repository is owned by `danieltamargo` and has 1 branch and 0 tags. The repository is a fork of `tabler / tabler-icons`. The repository is public and has 0 watchers. The repository is 1 commit ahead of, 341 commits behind the original repository's `main` branch.

The red arrow indicates the flow from the original repository to the forked repository, representing the process of forking the repository.

Fork y pull request



Fichero .gitignore

El archivo **.gitignore** es un fichero especial que indica a Git qué archivos o directorios debe **ignorar** y no incluir en el control de versiones. Esto es útil para excluir archivos temporales, configuraciones locales, o datos sensibles que no quieres subir al repositorio.

```
# Ignorar archivos de configuración local
config/settings.json

# Ignorar todos los archivos .log
*.log

# Ignorar la carpeta de compilación
/build/
```

Podemos utilizar herramientas para preparar un fichero .gitignore muy útil, por ejemplo:
<https://www.toptal.com/developers/gitignore>

Otros conceptos que podrían ser interesantes

- **git hooks** (scripts que se ejecutan automáticamente en respuesta a ciertos eventos de Git)
- **cherry-pick** (copiar un commit específico de una rama y pegarlo en otra)
- **rebase** (juntar varios commits para tener un historial de commits más limpio)
- **remote upstream** (repositorio original del que hiciste un fork)
- **reflog** (histórico de todas las acciones realizadas en el repositorio, permite recuperar commits que ya no están accesibles desde ninguna referencia, puede usarse para recuperar un commit eliminado accidentalmente, guarda en el repositorio local un histórico durante 30 días)
- **clean** (elimina ficheros que no están rastreados ni se han añadido al staging area, es decir, archivos locales sin seguimiento en Git)
- **rm** (elimina ficheros que ya están rastreados o en el staging area, marcándolos para que se incluyan en el próximo commit y se eliminen también del repositorio al hacer push)
- **tag** (crear etiquetas, es decir, referencias, a puntos concretos en el historial de git)