Empezando con Git y GitHub

1. Introducción

Un **sistema de control de versiones** (VCS) es un programa o conjunto de programas que realiza un seguimiento de los cambios en una colección de archivos. Con un VCS, puedes:

- Ver todos los cambios realizados en el proyecto, fecha y responsable.
- Recuperar versiones anteriores del proyecto completo o de archivos individuales.
- Incluir un mensaje explicativo con cada cambio.
- Crear *ramas*, donde los cambios se pueden hacer experimentalmente. Esta característica permite que se trabaje en varios conjuntos de cambios diferentes al mismo tiempo, posiblemente por personas distintas, sin que ello afecte a la rama principal. Más adelante se pueden fusionar los cambios que se quieren mantener en la rama principal.

Git es un VCS de código abierto rápido, versátil, muy escalable y gratuito. Su autor principal es Linux Torvalds, creador de Linux.

Git es un **sistema distribuido**, lo que significa que el historial completo de un proyecto se almacena en el cliente y en el servidor. Se pueden editar archivos sin conexión de red, protegerlos localmente y sincronizarlos con el servidor cuando una conexión esté disponible. Si un servidor deja de funcionar, todavía tendrá una copia local del proyecto. Técnicamente, ni siquiera es necesario tener un servidor. Los cambios pueden pasarse por correo electrónico o compartirse mediante medios extraíbles, pero, en la práctica, nadie usa Git así.

Por su parte, GitHub es un servicio de alojamiento de tus repositorios en internet. No es el único, por supuesto, existen otros como GitLab o Bitbucket, pero GitHub es el más conocido y utilizado, por lo que es el que utilizaremos en este documento. Si no tienes cuenta, deberías crearla ya.

1.1. Terminología de Git

Para entender cómo se trabaja con Git hay que comprender su terminología. Esta es una breve lista de algunos términos que se usan frecuentemente con Git. De momento no te preocupes por los detalles, todos estos términos te irán resultando familiares con el tiempo.

- **Árbol de trabajo**: Conjunto de directorios y archivos anidados que contienen el proyecto en el que se trabaja.
- **Repositorio**: Directorio, situado en el nivel superior de un árbol de trabajo, donde Git mantiene todo el historial y los metadatos de un proyecto.
- **Hash**: Número generado por una función *hash* que representa el contenido de un archivo u otro objeto como un número de dígitos fijo. Git usa hashes de 160 bits de longitud. Una ventaja de usar códigos hash es que Git puede indicar si un archivo ha cambiado aplicando un algoritmo hash a su contenido y comparando el resultado con el hash anterior. Si se cambia la marca de fecha y hora del archivo, pero el hash del archivo no, Git sabe que el contenido del archivo no se ha modificado.

- **Objeto**: Un repositorio de Git contiene cuatro tipos de *objetos*, cada uno identificado de forma única por un hash SHA-1[1].
  - Un objeto *blob* contiene un archivo normal.
  - Un objeto *árbol* representa un directorio. Contiene nombres, hashes y permisos.
  - Un objeto de *commit* representa una versión específica del árbol de trabajo.
  - Una *etiqueta* es un nombre asociado a una confirmación.
- **Confirmación (*commit*)**: Cuando se usa como verbo, *commit* significa crear un objeto de confirmación. Esta acción toma su nombre de las confirmaciones en una base de datos. Significa que se confirman los cambios realizados.
- **Rama**: Es una colección de confirmaciones vinculadas. La rama predeterminada que se crea al inicializar un repositorio se denomina `main` (antiguamente `master`). El nivel superior de la rama actual se denomina `HEAD`. Las ramas son una característica increíblemente útil de Git porque permiten a los desarrolladores trabajar de forma independiente (o conjunta) en ramas y luego combinar los cambios en la rama principal.
- **Remoto**: Referencia con nombre a otro repositorio de Git. Cuando se crea un repositorio local, Git puede crear uno remoto, denominado `origin`, que es el predeterminado para las operaciones de envío e incorporación de cambios.
- **Comandos, subcomandos y opciones**: Las operaciones de Git se realizan mediante comandos, como `git push`. Aquí, `git` es el comando y `push` es el subcomando. El subcomando especifica la operación que quiere que Git realice. Los comandos suelen ir acompañados de opciones, que usan guiones (-) o guiones dobles (--). Por ejemplo, `git reset --hard`.

1.2. Herramientas de línea de comandos de Git

Hay varias interfaces gráficas diferentes disponibles para Git, como GitHub Desktop. Muchos editores, como Visual Studio Code, también tienen una interfaz gráfica para Git. Todas estas GUIs, aunque funcionan de manera diferente y tienen distintas limitaciones, facilitan el trabajo con Git. Sin embargo, ninguna de ellas implementa la funcionalidad de Git al completo.

La interfaz de línea de comandos (o *CLI*) de Git funciona igual independientemente del sistema operativo donde se use y permite aprovechar toda la capacidad de Git. Si aprendes a usar Git solo a través de una GUI quizá puedas hacer todo lo que necesitas en un proyecto, pero es posible que recibas algún mensaje de error que no sepas resolver.

La hoja de ruta recomendada es aprender a manejar Git primero con la línea de comandos y, después, usar una GUI. Más adelante veremos su integración en Visual Studio Code (VSC, en adelante), de momento, utilizaremos la línea de comandos.

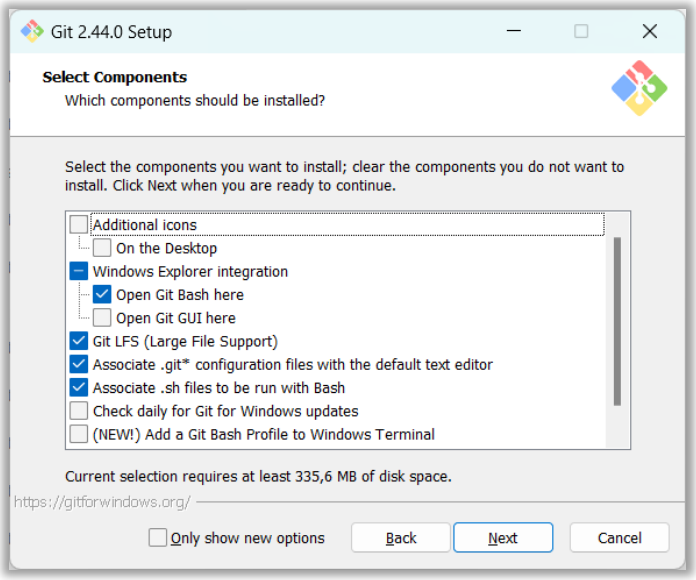
[1] SHA-1 (*Secure Hash Algorithm 1*) es una función de hash criptográfica que produce un resumen de 160 bits (20 bytes) a partir de una entrada de datos de cualquier tamaño. Ampliamente utilizada en el pasado, se considera insegura hoy en día, por lo que el equipo de Git está trabajando para migrar a SHA-256 (actualizado en mayo de 2024).

1.3. Instalación y configuración de Git

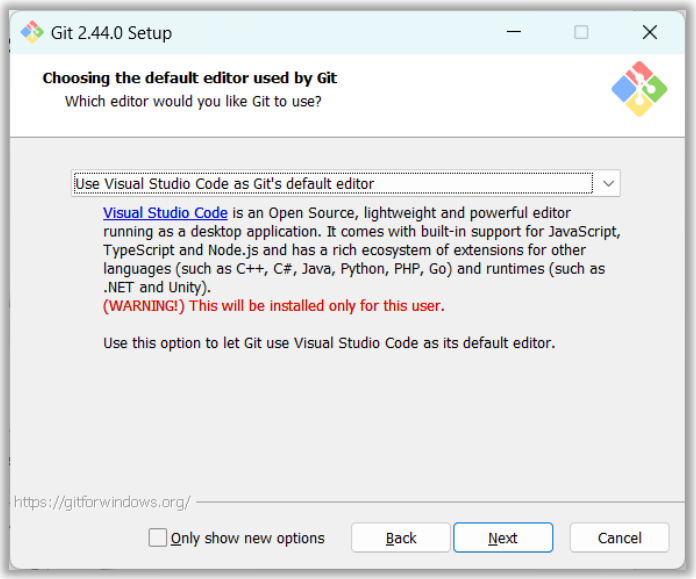
Quizá ya tienes instalado Git y no lo sepas. Para ello, ve a la terminal de tu sistema operativo y teclea:

```
$ git --version
```

Si no lo tienes instalado, puedes hacerlo siguiendo los pasos descritos aquí. Si vas a usar Git bajo Windows, es conveniente que dejes la configuración por defecto:



La siguiente pantalla te pedirá que elijas el editor de texto por defecto que utilizará Git cuando necesite que introduzcas un mensaje. Puedes usar el que prefieras, en este caso usaremos uno de los más usados, si no el que más, Visual Studio Code.



Otra opción es hacerlo una vez instalado desde la terminal con el siguiente comando:

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

1.3.1. Línea de comandos

Como hemos comentado antes, vamos a aprender a utilizar Git primero con la línea de comandos. Tanto macOS como cualquier distribución de Linux incluyen terminales de líneas de comandos Unix integrados. Sin embargo, Windows utiliza el símbolo del sistema, que no es Unix, por lo que los comandos de Git no funcionarán. Para solucionarlo, con la instalación de Git bajo Windows normalmente se incluye **Git Bash**, una interfaz de línea de comandos que emula una terminal de Unix.

Bash es un acrónimo de ***B**ourne **A**gain **SH**ell*. Una *shell* es un programa que proporciona una interfaz entre el usuario y el sistema operativo, permitiendo a los usuarios interactuar con el sistema operativo mediante la introducción de comandos. Bash es una interpretación de la shell Bourne, que fue la shell estándar en muchos sistemas Unix, y se acabó convirtiendo en la shell predeterminada en muchas distribuciones de Linux.

1.3.2. Configuración inicial

Lo primero que deberás hacer cuando instales Git es establecer tu nombre de usuario y dirección de correo electrónico. Esto es **importante** porque los commits de Git usan esta información para identificar al autor de cada commit:

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

Sólo necesitarás hacer esto una vez si especificas la opción `--global`, ya que Git siempre usará esta información para todo lo que hagas en ese sistema. Si quieres sobrescribir esta información con otro nombre o dirección de correo para proyectos específicos, ejecuta el comando sin la opción `--global` cuando estés en ese proyecto.

Ejecuta el siguiente comando para comprobar que los cambios han funcionado:

```
$ git config --list
```

1.4. Ayuda desde Git

Git, al igual que la mayoría de las herramientas de línea de comandos, tiene una función de ayuda integrada que se puede usar para buscar comandos y palabras clave.

Usa el siguiente comando para obtener ayuda sobre lo que puede hacer con Git:

```
$ git help
```

Observa las distintas opciones disponibles con Git. Cada comando incluye su propia página de ayuda. Para obtener ayuda específica sobre un comando, puedes utilizar cualquiera de estas dos opciones:

```
$ git help <comando>
$ git <comando> –help
```

Por ejemplo, para obtener ayuda sobre el comando `commit` sería:

```
$ git help commit
$ git commit –help
```

1.5. Estados de un archivo en Git

Presta atención porque es **muy importante** que tengas esto presente en todo momento cuando trabajes con Git. Cada archivo dentro de un proyecto de Git estará en alguno de los siguientes estados:

- **Rastreado (*tracked*).** Los archivos rastreados son todos aquellos que estaban en el último `commit` del repositorio. Por tanto, en un repositorio recién inicializado no hay ningún archivo rastreado. A su vez, estos archivos pueden estar:
  - **Confirmado (*committed*):** Significa que la última versión del archivo que tiene el repositorio es exactamente la misma que la que tenemos nosotros en nuestro directorio local.
  - **Modificado (*modified*):** Significa que el archivo ha sido modificado desde la última confirmación, pero aún no lo has confirmado ni has indicado a Git que quieres que vaya en la próxima confirmación.
  - **Preparado (*staged*):** Significa que el archivo ha sido modificado y lo has marcado en su versión actual para que vaya en el próximo commit.
- **No rastreado (*untracked*).** Los archivos sin rastrear son todos los demás, es decir, cualquier otro archivo en tu directorio de trabajo que no estaba en tu último commit y que no está en el área de preparación.

Esto nos lleva a las **tres secciones principales de un proyecto de Git**: el directorio de Git (*Git directory*), el directorio de trabajo (*working directory*) y el área de preparación (*staging area* o *index*).

- El **directorio de Git** es donde se almacenan los metadatos y la base de datos de objetos para tu proyecto. Es la parte más importante de Git y es lo que se copia cuando clonas un repositorio desde otra computadora.
- El **directorio de trabajo** (*working directory*) es una copia de una versión del proyecto. Estos archivos se sacan de la base de datos comprimida en el directorio de Git y se colocan en disco para que los puedas ver o modificar.
- El **área de preparación** (también llamada *index* o *staged area*) es un archivo, generalmente contenido en tu directorio de Git, que almacena información acerca de lo que se va a incluir en tu próximo commit.

Por tanto, el **flujo de trabajo** básico en Git es algo así:

1. Modificas una serie de archivos en tu directorio de trabajo.
2. Añades tus archivos al área de preparación.
3. Confirmas los cambios. Esto implica tomar los archivos tal y como están en el área de preparación y almacenar esa copia instantánea de manera permanente en tu directorio de Git.

2. Manejo básico de Git

2.1. Crear un repositorio local: `git init`

Vamos a crear nuestro primer repositorio Git en un directorio llamado *repo-pruebas*. Crea la carpeta desde el explorador de archivos de tu sistema operativo o desde una terminal de línea de comandos:

```
$ mkdir repo-pruebas
$ cd repo-pruebas
```

Para inicializar Git en este directorio usamos el comando `git init`:

```
$ git init
```

Esto crea un subdirectorio nuevo llamado `.git`, que es donde se almacenarán todos los archivos de Git de este repositorio. Si te fijas, verás que en la terminal aparece ahora la indicación de la rama principal de Git, llamada *main*.

```
~/Desktop/repo-pruebas (main)
$ ls -la
total 8
drwxr-xr-x 1 isaac 197609 0 Sep  6 19:47 ./
drwxr-xr-x 1 isaac 197609 0 Sep  6 19:47 ../
drwxr-xr-x 1 isaac 197609 0 Sep  6 19:47 .git/
```

Un comando que utilizarás con mucha frecuencia es `git status`, dado que muestra mucha y muy valiosa información.

```
$ git status
On branch main

No commits yet

nothing to commit (create/copy files and use "git add" to track)
```

En nuestro caso nos está diciendo que estamos en la rama *main*, que no hay ningún commit aún y que no hay nada con lo que hacer un commit, que creemos o copiemos algún archivo y usemos `git add`. Pues eso vamos a hacer.

2.2. Añadir modificaciones al *staged area*: `git add`

Crearemos dos ficheros vacíos, *archivo1* y *archivo2*. En sistemas basados en Linux puedes usar el comando `touch` para crear los archivos vacíos.

```
$ touch archivo1
$ touch archivo2
```

Si ahora ejecutamos `git status`:

```
$ git status
On branch main

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        archivo1
        archivo2

nothing added to commit but untracked files present (use "git add" to track)
```

Nos indica que tenemos dos archivos no rastreados (*untracked*) y que no hemos añadido nada al área de preparación (*staged area*). Un archivo sin rastrear es aquel que no tenías en el commit anterior. **Git no incluirá los archivos no rastreados en tu próximo commit a menos que se lo indiques explícitamente**. Se comporta así para evitar incluir accidentalmente archivos binarios o cualquier otro archivo que no quieras incluir.

Tendremos que indicar qué archivos queremos añadir al *staged area* usando el comando `git add`.

Vamos a añadir *archivo1* al *staged area*:

```
$ git add archivo1
```

Veamos la situación con `git status`:

```
$ git status
On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   archivo1

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        archivo2
```

El comando `git status` nos está indicando que hemos añadido *archivo1* al *staged area* (está *preparado*) y que tenemos otro archivo no rastreado, *archivo2*.

Imagina que nos hemos equivocado añadiendo *archivo1*, podemos usar el comando `git rm -cached` para sacarlo del *staged area*. Esto no elimina ni modifica el archivo, simplemente le indica a Git que no queremos que esté en el *staged area*.

```
$ git rm --cached archivo1
```

Ahora `git status` detecta los dos archivos, pero ninguno de ellos está *trackeado*:

```
$ git status
On branch main

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        archivo1
        archivo2

nothing added to commit but untracked files present (use "git add" to track)
```

Si queremos añadir varios archivos los separamos con un espacio con `git add`:

```
$ git add archivo1 archivo2
```

Y si queremos añadirlos todos simplemente usamos un punto ('.'):

```
$ git add .
```

Ahora, `git status` nos indica que están los dos ficheros añadidos:

```
$ git status
On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   archivo1
        new file:   archivo2
```

Hemos utilizado `git add .` para añadir los dos archivos al *staged area*. Sin embargo, hay otras opciones con `git add` que tenemos que considerar:

- `git add .` añade todos los archivos nuevos y modificados en el directorio actual y sus subdirectorios, pero no añade archivos eliminados.
- `git add -u` añade todos los archivos modificados y eliminados, pero no incluye archivos nuevos.
- `git add -A` añade todos los archivos nuevos, modificados y eliminados en el directorio actual y sus subdirectorios.

Por tanto, para añadir todas las posibles modificaciones, incluidos archivos nuevos, modificados y eliminados, **el comando más completo es** `git add -A`.

2.3. Confirmar cambios: `git commit`

El término `commit` es un verbo y un sustantivo. Básicamente tiene el mismo significado que cuando se confirma un cambio en una base de datos. Como verbo, *hacer un* ***`commit`*** significa confirmar los cambios y colocar una copia en el repositorio de aquello que hayamos confirmado como una nueva versión. Como sustantivo, un `commit` es el pequeño fragmento de datos que asigna una identidad única a los cambios que se confirman. Los datos que se guardan en un `commit` incluyen el nombre del autor y la dirección de correo electrónico, la fecha, los comentarios sobre lo que se ha hecho, una firma digital opcional y el identificador único del `commit` anterior.

Recordemos la situación, `git status` nos indicó que tenemos todos los cambios en el *staged area*:

```
$ git status
On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   archivo1
        new file:   archivo2
```

Vamos a hacer nuestro primer commit para confirmar los cambios:

```
$ git commit
```

Al ejecutar este comando, se nos abrirá el editor de texto predeterminado para que insertemos un mensaje de commit. Si no hemos configurado aún VSC como nuestro editor predeterminado, podemos hacerlo con el siguiente comando:

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

Para elegir otros editores, haz clic aquí.

Si has elegido VSC como editor, al ejecutar `git commit` se abrirá el programa con un fichero más o menos como este:

```


# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# On branch main
#
# Initial commit
#
# Changes to be committed:
# new file:   archivo1
# new file:   archivo2
#
```

Insertar un mensaje en cada commit es **obligatorio**, si no insertamos nada y cerramos el fichero, el commit se abortará:

```
Aborting commit due to empty commit message.
```

Elegir el mensaje de commit no es irrelevante, hay algunas buenas prácticas que conviene que conozcas. Revísalas en el apartado Buenas prácticas al escribir un mensaje de commit.

Para añadir un mensaje de una línea rápidamente podemos usar el parámetro `-m`:

```
$ git commit -m 'commit inicial'
```

2.4. El extraño comportamiento de `git add`

En nuestro repositorio de pruebas hemos ejecutado hasta el momento las siguientes instrucciones:

```
git init
touch archivo1
touch archivo2
git add -A
git commit -m 'commit inicial'
```

Ambos archivos están vacíos de momento. Vamos a añadir una línea a *archivo1*:

```
$ echo "Nueva línea" >> archivo1
```

Ahora el fichero *file1* sólo tiene esa línea:

```
$ cat archivo1
Nueva línea
```

Si ejecutamos `git status`:

```
$ git status
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   archivo1

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

El archivo *archivo1* aparece en una sección llamada "*Changes not staged for commit*", lo que significa que existe un archivo rastreado que ha sido modificado en el directorio de trabajo pero que aún no está preparado. Pues vamos a prepararlo con `git add`:

```
$ git add archivo1
```

Ahora `git status` nos indica que el archivo está listo para ser añadido en el próximo commit:

```
$ git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   archivo1
```

Supongamos que ahora recuerdas que debes hacer un pequeño cambio en *archivo1* antes de confirmarlo. Abres el archivo, lo cambias y ahora estás listo para confirmar.

```
$ echo "Otra nueva línea" >> archivo1
```

Ejecutemos `git status` una vez más:

```
$ git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   archivo1

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   archivo1
```

¡WTF! Ahora *archivo1* aparece dos veces, como preparado y como no preparado al mismo tiempo. ¿Cómo es posible? Pues porque Git prepara un archivo de acuerdo al estado que tenía cuando ejecutas el comando `git add`. Si confirmas ahora, se confirmará la versión de *archivo1* que tenías la última vez que ejecutaste `git add`, no la versión que ves ahora en tu directorio de trabajo al ejecutar.

Así, es **importante** recordar que, si modificas un archivo después de ejecutar `git add`, deberás ejecutar `git add` de nuevo para preparar la última versión del archivo.

```
$ git add archivo1
```

Veamos ahora el estado de nuestros ficheros:

```
$ git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   archivo1
```

2.5. Eliminar archivos: `git rm`

Vamos a partir de la siguiente situación:

```
git init
touch archivo1
touch archivo2
git add -A
git commit -m "Commit inicial"
```

Ahora queremos eliminar *archivo1*. Existen dos maneras de eliminar un archivo trabajando con Git:

1. Usando el comando `git rm archivo1`.

```
$ git rm archivo1
```

Con este comando eliminamos el archivo y, además, avisamos a Git de la eliminación, por lo que añadirá automáticamente ese cambio al *staged area*. Lo vemos con `git status`:

```
$ git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        deleted:    archivo1
```

Hay que tener **cuidado** al leer la salida de `git status`. La frase *use "git restore --staged <file>..." to unstage* no nos está indicando cómo recuperar el archivo, sino cómo quitar ese cambio del área de preparación.

2. Usando el comando `rm archivo1` o eliminando el archivo desde el explorador de archivos de nuestro sistema operativo.

```
$ rm archivo1
```

En este caso no avisamos a Git de la eliminación, por lo que no se habrá añadido ese cambio al *staged area*. Observa la salida de `git status`, que nos informa de que este cambio no está en el área de preparación (*Changes not staged for commit*).

```
$ git status
On branch main
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        deleted:    archivo1

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

Si hemos usado esta manera de eliminar un archivo, simplemente ejecutando `git add archivo1` llegamos al mismo punto que en el primer caso.

```
$ git add archivo1

$ git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        deleted:    archivo1
```

2.6. Cambiar el nombre de un archivo: `git mv`

Git no rastrea explícitamente los cambios de nombre en archivos. Si renombras un archivo en un repositorio Git, no se guardará ningún metadato que indique que renombraste el archivo. Es por esto que resulta confuso que Git tenga un comando `git mv`. Si quieres renombrar un archivo en Git, puedes ejecutar:

```
$ git mv archivo1 file1

$ git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        renamed:    archivo1 -> file1
```

2.7. Ignorar archivos (archivo *.gitignore*)

A veces tendrás algún tipo de archivo que no quieras que Git añada automáticamente con `git add -A`. O, más aún, que ni siquiera quieras que aparezca como no rastreado, simplemente quieres que Git lo ignore por completo. Este suele ser el caso de archivos generados automáticamente, archivos creados por el sistema de compilación, archivos temporales, etc. En estos casos, puedes usar un archivo llamado "*.gitignore*", donde se pueden listar patrones de archivos a ignorar. Este es un ejemplo de un archivo ".gitignore":

```
# Ignora los archivos terminados en .a
*.a
# Deja de ignorar lib.a, aun cuando había ignorado los archivos terminados en .a en
la línea anterior
!lib.a
# Ignora unicamente el archivo TODO de la raiz, no subdir/TODO
/TODO
# ignora todos los archivos del directorio build/
build/
# ignora doc/notes.txt, pero no doc/server/arch.txt
doc/*.txt
# ignora todos los archivos .txt del directorio doc/
doc/**/*.txt
```

GitHub mantiene una extensa lista de archivos ".gitignore" adecuados a docenas de proyectos y lenguajes en https://github.com/github/gitignore.

2.8. Historial de confirmaciones: `git log`

El comando `git log` muestra los commits del proyecto en orden cronológico inverso, es decir, las confirmaciones más recientes se muestran al principio. Vamos a ver algunos de sus parámetros, aunque puedes ver una descripción detallada en la documentación.

```
$ git log
commit cbe9a83da3b93082c6e18e3e9fcc0284e193f1aa (HEAD -> main)
Author: Isaac-MSI <iexpositol03@educarex.es>
Date:   Sat Sep 7 16:10:24 2024 +0200

    Modifica archivo2

commit f6aa936e4e8b68a1cf2bbe5cfddd481048db98d9
Author: Isaac-MSI <iexpositol03@educarex.es>
Date:   Sat Sep 7 16:10:24 2024 +0200

    Modifica archivo1

commit cb501e72f58ce5f9a79b0169df6cab2829cc4230
Author: Isaac-MSI <iexpositol03@educarex.es>
Date:   Sat Sep 7 16:10:24 2024 +0200

    Commit inicial
```

2.8.1. En una línea: `--oneline`

Con la opción `--oneline` obtenemos una lista más concisa:

```
$ git log --oneline
cbe9a83 (HEAD -> main) Modifica archivo2
f6aa936 Modifica archivo1
cb501e7 Commit inicial
```

2.8.2. Últimos *n* commits: `-n`

Si queremos mostrar únicamente los *X* últimos commits podemos usar la opción `-n X`, donde X es un número de confirmación: '1' para ver solo el último commit, '2' para ver los dos anteriores, y así sucesivamente. Esta opción es compatible con usar `--oneline`. Por ejemplo, para mostrar los dos últimos commits en formato de una línea, sería:

```
$ git log -2 --oneline
cbe9a83 (HEAD -> main) Modifica archivo2
f6aa936 Modifica archivo1
```

Una forma abreviada es juntar la 'n' y el número: Incluso podemos prescindir de la 'n':

```
$ git log -n2
$ git log -2
```

2.8.3. En forma gráfica: `--oneline --decorate --graph --all`

También podemos combinar parámetros. Por ejemplo:

```
$ git log --oneline --decorate --graph --all
```

Cuya salida típica podría ser:

```
* f30ab9c (HEAD -> main) Merge branch 'feature'
|\
| * 1a4d8c7 (feature) Merge branch 'feature-subbranch' into feature
| |\
| | * 9a8f9e1 (feature-subbranch) Add new feature subbranch implementation
| |/
| * 6d7893f Add initial feature implementation
|/
* 92fb1b6 Fix bug in main branch
* 374d1c8 Update README
* a29b8f1 Initial commit
```

- La opción `--decorate` añade información adicional al log sobre las referencias de los commits (como ramas y tags).

- La opción `--graph` dibuja un gráfico ASCII que representa la historia del commit. Las líneas y caracteres adicionales muestran las relaciones entre los commits, incluyendo las fusiones (*merges*) y la estructura de las ramas.

- La opción `--all` muestra los commits de todas las ramas, no solo de la rama actual en la que te encuentras. Esto permite ver el historial completo del repositorio, incluyendo todas las ramas locales y remotas.

2.8.4. Personalizar el formato de salida: `--pretty=format`

Con esta opción podemos personalizar cómo queremos obtener la salida. Una manera común de usar esta opción es la siguiente:

```
$ git log --pretty=format:"%h - %an, %ar : %s"
faf100c - Isaac-MSI, 23 hours ago : Modifica archivo2
64bd0d8 - Isaac-MSI, 23 hours ago : Modifica archivo1
3cf7edc - Isaac-MSI, 23 hours ago : Commit inicial
```

Donde:

- *%h*: Abreviatura del hash del commit.
- *%an*: Nombre del autor.
- *%ar*: Fecha del commit (relativa).
- *%s*: Mensaje del commit.

Otros modificadores para esta opción los puede encontrar en la documentación.

2.8.5. Filtrar por fecha: `--since` y `--until`

Podemos mostrar los commits desde una fecha concreta, hasta una fecha concreta y también entre dos fechas:

```
$ git log --since="2023-01-01"
$ git log --until="2023-06-30"
$ git log --since="2023-01-01" --until="2023-06-30"
```

2.8.6. Filtrar por autor: `--author`

Podemos mostrar únicamente los commits de un autor determinado, útil cuando usamos Git en entornos colaborativos.

```
$ git log --author="nombre_autor"
```

2.8.7. Filtrar por texto en el commit: `--grep`

También podemos buscar los commits que tengan un texto concreto, por ejemplo:

```
$ git log --grep="Modifica" --oneline
faf100c Modifica archivo2
64bd0d8 Modifica archivo1
```

2.8.8. Mostrar el historial de un archivo: `--follow`

Podemos hacer un seguimiento de un archivo en concreto, por ejemplo:

```
$ git log --follow archivo1
commit 64bd0d81bbe6645508f1c29c54989db6a6ee7657
Author: Isaac-MSI <iexpositol03@educarex.es>
Date:   Wed Jun 19 12:14:19 2024 +0200

    Modifica archivo1

commit 3cf7edcea3f817bca3ff810b3ce792ceb7e0b6fc
Author: Isaac-MSI <iexpositol03@educarex.es>
Date:   Wed Jun 19 12:14:12 2024 +0200

    Commit inicial
```

2.8.9. Filtrar por eliminación o inserción de texto: `-S`

Podemos usar esta opción para saber en qué commits se modificó el código añadiendo o eliminando un fragmento de texto:

```
$ git log -S "Hola" --oneline
faf100c Modifica archivo2
64bd0d8 Modifica archivo1
```

2.8.10. Mostrar archivos modificados: `--name-only` y `--name-status`

Podemos mostrar también solo los archivos modificados (añadidos, modificados o eliminados) en cada commit con la opción `--name-only`:

```
$ git log --name-only --oneline
85081a5 (HEAD -> main) Añade .gitkeep para rastrear carpeta1
carpeta1/.gitkeep
faf100c Modifica archivo2
archivo2
64bd0d8 Modifica archivo1
archivo1
3cf7edc Commit inicial
archivo1
archivo2
```

Una variación de esta opción es `--name-status`, que muestra también si el archivo fue añadido (A), modificado (M) o eliminado (D).

```
$ git log --name-status --oneline
85081a5 (HEAD -> main) Añade .gitkeep para rastrear carpeta1
A       carpeta1/.gitkeep
faf100c Modifica archivo2
M       archivo2
64bd0d8 Modifica archivo1
M       archivo1
3cf7edc Commit inicial
A       archivo1
A       archivo2
```

2.9. Etiquetas: `git tag`

El comando `git tag` se utiliza para crear, listar y gestionar etiquetas en un repositorio. Una etiqueta señala un punto específico en el historial del repositorio, normalmente para identificar una versión o un punto concreto del estado del repositorio.

Existen dos tipos de etiquetas:

1. Etiquetas ligeras (*lightweight tags*): Son simples punteros a un commit específico.
2. Etiquetas anotadas (*annotated tags*): Guardan más información, como el nombre del autor, la fecha y un mensaje opcional.

Comando	Descripción
`git tag`	Muestra todas las etiquetas creadas en el repositorio.
`git tag v1.0`	Crea una etiqueta llamada v1.0 que apunta al commit actual.
`git tag -a v1.0 -m "Versión 1.0"`	Crea una etiqueta anotada que apunta al commit actual. La opción `-a` indica que es una etiqueta anotada y `-m` permite agregar un mensaje explicativo.
`git tag v1.0 <commit_hash>`	Crea una etiqueta en un commit anterior
`git show v1.0`	Muestra el commit y los metadatos asociados a la etiqueta (como el mensaje).
`git tag -d v1.0`	Elimina la etiqueta v1.0.

Para ejemplificar algunos de esos comandos vamos a partir del siguiente repositorio:

```
git init
echo "Hola Mundo desde archivo1" >> archivo1
git add -A
git commit -m "C1"

echo "Hola Mundo desde archivo2" >> archivo2
git add -A
git commit -m "C2"
```

Veamos el log:

```
$ git log --oneline
3ffe804 (HEAD -> main) C2
370cd20 C1
```

Vamos a etiquetar el último commit:

```
$ git tag -a v1.0 -m "Versión 1.0"
```

Veamos el log de nuevo:

```
$ git log --oneline
3ffe804 (HEAD -> main, tag: v1.0) C2
370cd20 C1
```

Podemos usar git show para ver la información de ese tag:

```
$ git show v1.0
tag v1.0
Tagger: Isaac-MSI <iexpositol03@educarex.es>
Date:   Sun Sep 8 11:37:05 2024 +0200

Versión 1.0

commit 3ffe804a6d9c12c8d4d58bf300a37204c1719df8 (HEAD -> main, tag: v1.0)
Author: Isaac-MSI <iexpositol03@educarex.es>
Date:   Sun Sep 8 11:36:20 2024 +0200

    C2

diff --git a/archivo2 b/archivo2
new file mode 100644
index 0000000..db9aee1
--- /dev/null
+++ b/archivo2
@@ -0,0 +1 @@
+Hola Mundo desde archivo2
```

Podemos etiquetar cualquier commit haciendo referencia a su hash:

```
$ git tag "C1" 370cd20

$ git log --oneline
3ffe804 (HEAD -> main, tag: v1.0) C2
370cd20 (tag: C1) C1
```

Por último, eliminaremos un tag:

```
$ git tag -d C1
Deleted tag 'C1' (was 370cd20)

$ git log --oneline
3ffe804 (HEAD -> main, tag: v1.0) C2
370cd20 C1
```

3. Deshaciendo cosas

En cualquier momento puede que quieras deshacer algo. Aquí repasaremos algunas herramientas básicas usadas para deshacer cambios que hayas hecho. **Esta es una de las pocas áreas de Git en las que puedes perder parte de tu trabajo si cometes un error**.

3.1. Modificar el último commit: `git commit --amend`

Ante de nada, debes saber que es **muy mala idea** cambiar commits que se han compartido con otro desarrollador o que se han publicado en un repositorio compartido, como GitHub.

Una de las acciones más comunes a deshacer es cuando confirmas un cambio antes de tiempo y olvidas agregar algún archivo, o te equivocas en el mensaje de confirmación. Si quieres rehacer la confirmación, puedes reconfirmar con la opción:

```
$ git commit --amend
```

Este comando utiliza tu área de preparación para la confirmación. Si no has hecho cambios desde tu último commit (por ejemplo, ejecutas este comando justo después de haber hecho commit), entonces lo único que cambiarás será el mensaje del commit.

Por ejemplo, partamos de la siguiente situación:

```
git init
touch archivo1
touch archivo2
git add -A
git commit -m "Commit inicial"
```

Acabamos de hacer un commit. En este punto, `git log` nos devuelve:

```
$ git log
commit 13c1d791bd83ae55340ccfb496db716b18ece967 (HEAD -> main)
Author: Isaac-MSI <iexpositol03@educarex.es>
Date:   Sat Sep 7 19:30:37 2024 +0200

    Commit inicial
```

Supongamos que justo después de ejecutar este commit nos damos cuenta de que hemos olvidado crear e incluir un tercer archivo llamado *archivo3*. Primero creamos y añadimos *archivo3* al área de preparación:

```
$ touch archivo3
$ git add archivo3
```

Ya podemos enmendar el commit anterior:

```
$ git commit --amend
```

En este momento, Git abrirá tu editor de texto predeterminado para que puedas modificar el mensaje del commit si lo deseas. Si no deseas cambiar el mensaje, simplemente guarda y cierra el editor. El comando `git commit --amend` reemplazará el commit anterior con uno nuevo que incluye los cambios adicionales (en este caso, *archivo3*).

La salida de `git log` ahora es:

```
$ git log
commit 3e6b566744ec2b8a25222d1eea521c4cd1e09ea2 (HEAD -> main)
Author: Isaac-MSI <iexpositol03@educarex.es>
Date:   Sat Sep 7 19:30:37 2024 +0200

    Commit inicial enmendado
```

Fíjate que el hash del commit ha cambiado, pero no la hora de su ejecución.

Existen dos opciones interesantes con combinación con `--amend`. Para modificar directamente el mensaje del commit podemos usar la opción `-m`, como con `git commit`:

```
$ git commit --amend -m "Commit inicial modificado"
```

O, si no queremos modificar el mensaje, podemos evitar que Git nos abra el editor de texto con la opción `--no-edit`:

```
git commit --amend --no-edit
```

3.2. Deshacer un archivo modificado: `git restore`

Supongamos que desde que hiciste el último commit has estado trabajando en un archivo. Ahora te das cuenta de que está todo mal y quieres deshacer los cambios. Es decir, quieres que tu archivo vuelva al estado en el que estaba en el último commit.

```
git init
echo "Hola Mundo" > archivo1
git add -A
git commit -m "Commit inicial"
echo "Editando archivo1" >> archivo1
```

El comando `git status` también nos dice cómo hacerlo:

```
$ git status
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   archivo1

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

Para descartar los cambios en *archivo1*, ejecutamos `git restore archivo1`

```
$ cat archivo1
Hola Mundo
Editando archivo1

$ git restore archivo1

$ cat archivo1
Hola Mundo
```

Si quieres descartar todos los cambios en todos los archivos:

```
$ git restore .
```

3.3. Deshacer un archivo preparado: `git restore --staged`

Supongamos que has modificado dos archivos y que quieres confirmarlos como dos cambios separados en dos commits distintos, pero accidentalmente has escrito `git add -A`.

```
git init
touch archivo1
touch archivo2
git add -A
git commit -m "Commit inicial"
echo "Hola Mundo" > archivo1
echo "Adiós Mundo" > archivo2
git add -A
```

El comando `git status` nos recuerda cómo sacar del índice un archivo.

```
$ git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   archivo1
        modified:   archivo2
```

Por tanto, para sacar *archivo2* del área de preparación ejecutaremos:

```
& git restore --staged archivo2
```

Ahora, la salida de `git status` nos dice que *archivo1* está preparado, pero *archivo2* no:

```
$ git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   archivo1

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   archivo2
```

También podemos sacar todos los archivos del área de preparación al mismo tiempo:

```
$ git restore --staged .
```

Si en algún sitio lees que este comando se escribe así:

```
$ git reset HEAD nombre_archivo
```

Debes saber que simplemente esa era la manera antigua, declarada obsoleta.

3.4. Recuperar un archivo eliminado: `git checkout`

Has eliminado un archivo por error, ¿cómo lo recuperas? Aquí hay que diferenciar entre dos situaciones posibles: si hemos hecho algún commit posterior a la eliminación o no.

3.4.1. Si no hay ningún commit posterior a la eliminación del archivo

Estamos en esta situación:

```
git init
touch archivo1
touch archivo2
git add -A
git commit -m "Commit inicial"
git rm archivo1
```

Tras esto, `git status` nos informará de que se ha eliminado un archivo y ese cambio en el repositorio se ha añadido al área de preparación:

```
$ git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        deleted:    archivo1
```

Para recuperar el archivo, primero tenemos que sacar esa modificación del área de preparación. Esto lo hacemos con `git restore --staged`, que ya vimos que es el comando que se utiliza para sacar modificaciones del *staged area*:

```
$ git restore --staged archivo1
```

Y ahora simplemente ejecutamos `git checkout -- archivo1`:

```
$ git checkout -- archivo1
```

El archivo aparecerá de nuevo en nuestro directorio. Ahora, `git status` no detectará ningún cambio, obviamente:

```
$ git status
On branch main
nothing to commit, working tree clean
```

Más adelante veremos que el comando `git checkout` también se usa para otras cosas, como cambiar de ramas.

3.4.2. Si hay al menos un commit posterior a la eliminación del archivo

Partimos de esta situación, donde creamos dos archivos e inicializamos el repositorio:

```
git init
echo "Hola Mundo desde archivo1" > archivo1
echo "Hola Mundo desde archivo2" > archivo2
git add -A
git commit -m "Commit inicial"
```

En este momento eliminamos *archivo1*, modificamos *archivo2* y hacemos un nuevo commit:

```
git rm archivo1
echo "Hola otra vez" >> archivo2
git add -A
git commit -m "Elimina archivo1 y modifica archivo2"
```

Ahora, para complicarlo más, creamos un tercer archivo y realizamos otro commit:

```
echo "Hola Mundo desde archivo3" > archivo3
git add -A
git commit -m "Crea archivo3"
```

Tenemos, por tanto, 3 commits:

```
$ git log --oneline
beb78fb (HEAD -> main) Crea archivo3
5bd830d Elimina archivo1 y modifica archivo2
b9a1481 Commit inicial
```

Además, en nuestro directorio solo deben existir *archivo2* y *archivo3*:

```
$ ls -la
total 54
drwxr-xr-x 1 isaac 197609   0 Jun 25 12:43 ./
drwxr-xr-x 1 isaac 197609   0 Jun 25 12:25 ../
drwxr-xr-x 1 isaac 197609   0 Jun 25 12:43 .git/
-rw-r--r-- 1 isaac 197609  40 Jun 25 12:43 archivo2
-rw-r--r-- 1 isaac 197609  26 Jun 25 12:43 archivo3

$ cat archivo2
Hola Mundo desde archivo2
Hola otra vez

$ cat archivo3
Hola Mundo desde archivo3
```

En este punto nos damos cuenta de que *archivo1* no deberíamos haberlo eliminado. Podríamos volver atrás y deshacer los dos últimos commits, pero no queremos perder esos commits porque contienen cambios importantes. ¿Cómo recuperamos *archivo1*?

Primero, debemos buscar en el historial de cambios el hash del último commit en el que *archivo1* estaba presente. Para ello usamos el comando `git log -- archivo1`.

```
$ git log -- archivo1
commit 1240e717f209a17ab51225d4795f0b4732aa09aa
Author: Isaac-MSI <iexpositol03@educarex.es>
Date:   Tue Jun 25 12:43:57 2024 +0200

    Elimina archivo1 y modifica archivo2

commit e940b295ea104c2f80399797aa710e9de8e95d83
Author: Isaac-MSI <iexpositol03@educarex.es>
Date:   Tue Jun 25 12:43:57 2024 +0200

    Commit inicial
```

Si el historial es largo, podemos usar la opción `--oneline`:

```
$ git log --oneline -- archivo1
1240e71 Elimina archivo1 y modifica archivo2
e940b29 Commit inicial
```

El hash que necesitamos es el del commit inicial, `e940b29`, porque en el otro es donde eliminamos *archivo1*, por lo que el archivo no está en ese commit.

Una vez tenemos el hash del commit correcto, restauramos el archivo con el comando `git checkout`:

```
$ git checkout e940b29 -- archivo1
```

Si todo ha ido bien, deberías ver de nuevo *archivo1* en tu directorio con el contenido que tenía en el commit inicial:

```
$ ls -la
total 55
drwxr-xr-x 1 isaac 197609   0 Jun 25 12:49 ./
drwxr-xr-x 1 isaac 197609   0 Jun 25 12:25 ../
drwxr-xr-x 1 isaac 197609   0 Jun 25 12:49 .git/
-rw-r--r-- 1 isaac 197609  27 Jun 25 12:49 archivo1
-rw-r--r-- 1 isaac 197609  40 Jun 25 12:43 archivo2
-rw-r--r-- 1 isaac 197609  26 Jun 25 12:43 archivo3

$ cat archivo1
Hola Mundo desde archivo1
```

Además, ni *archivo2* ni *archivo3* han visto modificado su contenido:

```
$ cat archivo2
Hola Mundo desde archivo2
Hola otra vez

$ cat archivo3
Hola Mundo desde archivo3
```

3.5. Eliminar commits: `git reset`

La función del comando `git reset` es mover el `HEAD` a un estado específico.

Por ejemplo, estás trabajando en un proyecto añadiendo una nueva funcionalidad. Llegado cierto momento, te das cuenta que tienes un bug e intentas arreglarlo. Trabajas unos días en arreglar ese bug, pero te das cuenta que estás encarando mal la solución, por lo que quieres deshacer todo lo que has hecho desde que te pusiste a solucionarlo. Desde entonces has hecho varios commits que te gustaría eliminar definitivamente del historial. En estas situaciones puede usar el comando `git reset`.

Existen varios modos de uso para `git reset` que puedes ver en la documentación. En esta sección solo veremos los dos modos probablemente más utilizados:

- `--soft`: Este modo mueve el puntero del commit actual (`HEAD`) al commit especificado, pero deja los cambios en la zona de preparación y en el directorio de trabajo. Esto es útil si deseas rehacer el último commit o hacer cambios adicionales antes de volver a hacer un commit.

```
git reset --soft <commit>
```

- `--hard`: Este modo mueve el puntero del commit actual (`HEAD`), el índice la zona de preparación y el directorio de trabajo al commit especificado. Todos los cambios no confirmados y no preparados se pierden. Es importante tener cuidado al usar este modo, ya que **se pueden perder datos**.

```
git reset --hard <commit>
```

Las diferencias principales son las siguientes:

Modo	Staged Area	Working Directory	Efecto
`--soft`	Mantiene los cambios ya añadidos y añade los cambios desde el commit indicado	No cambia	Deshace el commit, pero mantiene los cambios preparados
`--hard`	Restablece la zona de preparación al commit indicado	Restablece el directorio de trabajo al commit indicado	Descarta todos los cambios y restablece el estado del último commit

Vamos a ver en funcionamiento estas dos opciones. Partiremos del siguiente estado en el repositorio:

```
git init
touch archivo1
touch archivo2
git add -A
git commit -m "C1"
git tag C1

echo "Hola Mundo desde archivo1" >> archivo1
git add -A
git commit -m "C2"
git tag C2
```

```
echo "Hola Mundo desde archivo2" >> archivo2
git add -A
git commit -m "C3"
git tag C3

echo "Hola Mundo desde archivo3" >> archivo3
git add -A
```

Vamos a usar tags para hacer más comprensible el proceso. Tenemos 3 commits y 3 archivos. La modificación de archivo3 está añadida al *staged area*, pero aún no hay ningún commit que confirme ese cambio en el historial.

```
$ git log --oneline
a255d08 (HEAD -> main, tag: C3) C3
21a435a (tag: C2) C2
582fc4a (tag: C1) C1

$ git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   archivo3
```

En este momento queremos eliminar los commits C3 y C2. Veamos las opciones.

3.5.1. *Reset* blando: `git reset --soft`

Veamos lo que ocurre al ejecutar un *reset* blando:

```
$ git reset --soft C1
```

Lo primero que observaremos es que la salida de `git log` cambia, ya no muestra los commits C2 y C3. Esto no significa que estos commits se hayan eliminado, un poco más adelante veremos cómo volver a C3.

```
$ git log --oneline
582fc4a (HEAD -> main, tag: C1) C1
```

Lo siguiente que veremos es que `git status` marca ahora *archivo1* y *archivo2* como archivos añadidos al índice:

```
$ git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   archivo1
        modified:   archivo2
        new file:   archivo3
```

Por supuesto, los 3 archivos mantienen el contenido que indicamos al principio:

```
$ cat archivo1
Hola Mundo desde archivo1

$ cat archivo2
Hola Mundo desde archivo2

$ cat archivo3
Hola Mundo desde archivo3
```

En este punto podemos hacer varias cosas.

1. Crear un commit nuevo, de forma que estaríamos uniendo en un solo commit los cambios que antes teníamos en C2 y C3:

```
$ git commit -m "Nuevo commit agrupando cambios de C2 y C3"
```

2. Deshacer algún cambio hecho antes o introducir algún cambio nuevo que no hubiéramos hecho en los commits anteriores antes de hacer un nuevo commit:

```
$ echo "Modificación adicional" >> archivo1
$ git add archivo1
$ git commit -m "Nuevo commit con cambios adicionales"
```

3. Restaurar el índice al estado de C1, pero manteniendo los cambios en el directorio de trabajo, es decir, sacar los cambios del índice:

```
$ git restore --staged .

$ git status
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   archivo1
        modified:   archivo2

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        archivo3

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

Fíjate en la presencia del punto '.' tras `--staged` para indicar todos los archivos.

4. Volver al estado de C3 para deshacer el `reset --soft`:

```
$ git reset --soft C3
```

Aquí hemos usado el tag C3 para indicar al commit al que queremos volver, pero si tu último commit no tiene tag o no te acuerdas, no puedes usar `git log` para averiguarlo:

```
$ git log --oneline
582fc4a (HEAD -> main, tag: C1) C1
```

Para ver el historial completo debemos usar el comando `git reflog`. Tienes una explicación más detallada sobre este comando en la documentación.

```
$ git reflog
582fc4a (HEAD -> main, tag: C1) HEAD@{0}: reset: moving to C1
a255d08 (tag: C3) HEAD@{1}: commit: C3
21a435a (tag: C2) HEAD@{2}: commit: C2
582fc4a (HEAD -> main, tag: C1) HEAD@{3}: commit (initial): C1
```

Resumidamente, este comando muestra el historial de movimientos del puntero `HEAD`. Su utilidad radica en que permite recuperar commits, ya que registra todos los cambios de `HEAD`, incluyendo aquellos que no aparecen en el historial de commits normal.

Viendo la salida del comando, vemos que el último commit es `a255d08`. Utilizamos ese hash para volver a C3:

```
$ git reset --soft a255d08
```

Al ejecutar este comando volvemos a la situación inicial:

```
$ git log --oneline
a255d08 (HEAD -> main, tag: C3) C3
21a435a (tag: C2) C2
582fc4a (tag: C1) C1

$ git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   archivo3
```

5. Descartar definitivamente todos los cambios posteriores a C1. Para esto usamos la opción `--hard`, que veremos a continuación.

3.5.2. *Reset* duro: `reset --hard`

Veamos qué diferencia hay entre las opciones `--soft` y `--hard`. Si ejecutamos el reset duro, volveremos al commit C1:

```
$ git reset --hard C1
HEAD is now at 331a9f9 C1

$ git log --oneline
331a9f9 (HEAD -> main, tag: C1) C1
```

Nuestro índice ahora estará exactamente como estaba en C1, no mantiene ni los cambios añadidos tras C3 ni los cambios introducidos por C2 y C3.

```
$ git status
On branch main
nothing to commit, working tree clean
```

Y el estado de los archivos es el mismo de C1, con *archivo1* y *archivo2* vacíos. Además, *archivo3* no existirá ya que se crea después de C1.

```
$ ls -l
total 4
-rw-r--r-- 1 isaac 197609   0 Jul  1 20:24 archivo1
-rw-r--r-- 1 isaac 197609   0 Jul  1 20:24 archivo2

$ cat archivo1

$ cat archivo2
```

La situación del directorio de trabajo y del índice son distintas que en el caso de *reset* blando. Cuando hacemos un *reset* duro, tanto el directorio de trabajo como el índice pasan a estar exactamente como estaban en el commit destino, C1 en este caso. Sin embargo, con el *reset* blando el directorio de trabajo se mantiene tal y como estaba antes del commit; mientras que en el área de preparación se mantienen los cambios añadidos antes del *reset* y se añaden los cambios de los commits que se han deshecho, en este caso C2 y C3.

Por supuesto, podemos volver a último commit igual que hacíamos con el *reset* blando. Podemos usar `git reflog` para averiguar el último commit del historial:

```
$ git reflog
f394e9a (HEAD -> main, tag: C1) HEAD@{0}: reset: moving to C1
8139659 (tag: C3) HEAD@{1}: commit: C3
61cd2c2 (tag: C2) HEAD@{2}: commit: C2
f394e9a (HEAD -> main, tag: C1) HEAD@{3}: commit (initial): C1

$ git reset --hard C3
HEAD is now at 8139659 C3

$ git log --oneline
8139659 (HEAD -> main, tag: C3) C3
61cd2c2 (tag: C2) C2
f394e9a (tag: C1) C1
```

Sin embargo, a diferencia del *reset* blando, fíjate en el estado del directorio de trabajo y del área de preparación ahora mismo:

```
$ ls -l
total 6
-rw-r--r-- 1 isaac 197609  27 Jul  1 21:07 archivo1
-rw-r--r-- 1 isaac 197609  27 Jul  1 21:07 archivo2

$ cat archivo1
Hola Mundo desde archivo1

$ cat archivo2
Hola Mundo desde archivo2

$ git status
On branch main
nothing to commit, working tree clean
```

Ahora *archivo3* ya no existe. Con el *reset* duro perdemos todos los cambios que hayamos hecho después del último commit.

Esto es **importante**. Ejecutar `reset --hard` es **peligroso** porque es **uno de los pocos casos en que Git destruye datos sin posibilidad de recuperación**. Cualquier otra opción de *reset* puede deshacerse fácilmente, pero no con la opción `--hard`, dado que sobrescribe los archivos en el directorio de trabajo

Puedes acceder a una explicación más detallada en el apartado Reset Demystified de la documentación (también disponible en español).

3.6. Deshacer cambios en el historial de commits: `git revert`

En tu empresa recientemente se ha desplegado a producción una nueva versión de una aplicación. Poco después se descubre que un commit específico introdujo un bug crítico. Necesitas deshacer ese commit para corregir el problema en producción lo antes posible, pero manteniendo un historial claro de todos los cambios realizados posteriores a ese commit. En este tipo de situaciones usaremos el comando `git revert`.

Vamos a simular la situación que acabamos de describir:

```
git init
touch archivo1
touch archivo2
git add -A
git commit -m "C1 - Commit inicial"
echo "Hola Mundo desde archivo1" >> archivo1
git add -A
git commit -m "C2 - Inicialización de archivo1"
echo "Hola Mundo desde archivo2" >> archivo2
git add -A
git commit -m "C3 - Inicialización de archivo2"
echo "Nueva línea en archivo1" >> archivo1
git add -A
git commit -m "C4 - Modifica archivo1"
```

Veamos la salida de `git log`:

```
$ git log --oneline
2e47ce6 (HEAD -> main) C4 - Modifica archivo1
a278d82 C3 - Inicialización de archivo2
5b375d2 C2 - Inicialización de archivo1
dc6e8d5 C1 - Commit inicial
```

En este punto nos damos cuenta de que en el commit C3 introdujimos un error crítico y queremos revertir ese commit. Sólo queremos eliminar los cambios que introdujo ese commit, ninguno más. Para esto tenemos el comando `git revert`:

```
$ git revert a278d82
```

Git abrirá el editor por defecto para que introduzcamos un mensaje de commit. El mensaje que añade por defecto debe ser parecido al siguiente:

```
Revert "C3 - Inicialización de archivo2"

This reverts commit a278d82ad4f0aac81f485806ab8954af403314d0.

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# On branch main
# Changes to be committed:
# modified:   archivo2
```

Cuando cerremos el archivo Git habrá creado otro commit, revirtiendo C3:

```
$ git log --oneline
ec9e213 (HEAD -> main) Revert "C3 - Inicialización de archivo2"
2e47ce6 C4 - Modifica archivo1
a278d82 C3 - Inicialización de archivo2
5b375d2 C2 - Inicialización de archivo1
dc6e8d5 C1 - Commit inicial
```

Si recuerdas, en ese commit C3 lo único que hacíamos es escribir *"Hola Mundo desde archivo2"* en *archivo2*. Ese es el cambio que hemos revertido, por lo que ahora el archivo debería estar vacío, mientras que *archivo1* debería estar intacto:

```
$ cat archivo2

$ cat archivo1
Hola Mundo desde archivo1
Nueva línea en archivo1
```

Por supuesto, si hubiera habido más cambios en otros archivos en ese commit hubieran sido eliminados también.

Al utilizar este comando es frecuente que se produzcan **conflictos** que tendremos que resolver. Utilizando el mismo ejemplo con el que estábamos trabajando, imagina que el error crítico que queremos deshacer está en el commit C2, no en el C3. Partimos de este punto (recuerda inicializar de nuevo el repositorio):

```
git init
touch archivo1
touch archivo2
git add -A
git commit -m "C1 - Commit inicial"

echo "Hola Mundo desde archivo1" >> archivo1
git add -A
git commit -m "C2 - Inicialización de archivo1"

echo "Hola Mundo desde archivo2" >> archivo2
git add -A
git commit -m "C3 - Inicialización de archivo2"

echo "Nueva línea en archivo1" >> archivo1
git add -A
git commit -m "C4 - Modifica archivo1"
```

Si te fijas, en el commit C2 se introducen cambios en *archivo1*, pero en el commit C4, que es posterior, también. Así, *archivo1* luce ahora mismo de la siguiente manera:

```
Hola Mundo desde archivo1   <-- Línea introducida en C2
Nueva línea en archivo1     <-- Línea introducida en C4
```

Cuando revirtamos el commit C2 se va a producir un conflicto y Git nos pedirá que lo resolvamos. Veamos:

```
$ git log --oneline
ee86d5d (HEAD -> main) C4
33d0b68 C3
798d65d C2
61dcb73 C1

$ git revert 798d65d
Auto-merging archivo1
CONFLICT (content): Merge conflict in archivo1
error: could not revert 798d65d... C2 - Inicializaci├│n de archivo1
hint: After resolving the conflicts, mark them with
hint: "git add/rm <pathspec>", then run
hint: "git revert --continue".
hint: You can instead skip this commit with "git revert --skip".
hint: To abort and get back to the state before "git revert",
hint: run "git revert --abort".
```

Git nos está diciendo con esta salida es que hay un conflicto en *archivo1* y nos da tres posibles caminos a seguir (*hints*). Hay que tener en cuenta que se pueden revertir más de un commit al mismo tiempo, que veremos un poco más adelante, por eso nos ofrece estas posibilidades.

1. **Resolver el conflicto y continuar.** Después de resolver los conflictos manualmente, tendremos que marcar los archivos afectados (en este caso, *archivo1*) como resueltos usando `git add archivo1` y continuar con el proceso usando `git revert --continue`. Si no hay más commits por revertir, el proceso acaba aquí.

```
hint: After resolving the conflicts, mark them with
hint: "git add/rm <pathspec>", then run
hint: "git revert --continue".
```

2. **Saltar este commit.** Si no quieres resolver el conflicto, puedes omitir el commit problemático usando el comando `git revert --skip`. Si no hay más commits por revertir, el proceso acaba aquí.

```
hint: You can instead skip this commit with "git revert --skip".
```

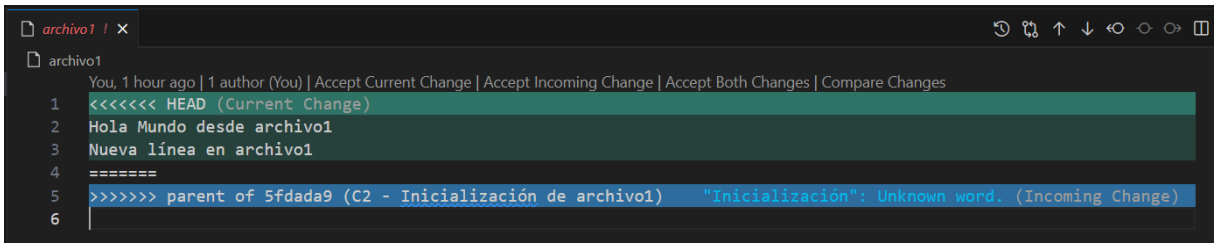
3. **Abortar el proceso completamente.** Si decides abortar el proceso de revertir y volver al estado anterior, usa `git revert --abort`. No importa si quedan más commits pro revertir o no.

```
hint: To abort and get back to the state before "git revert",
hint: run "git revert --abort".
```

Vamos a completar el paso 1. El conflicto en *archivo1* es el que ya comentamos un poco más arriba:

```
Hola Mundo desde archivo1   <-- Línea introducida en C2
Nueva línea en archivo1     <-- Línea introducida en C4
```

Cuando abrimos el archivo los conflictos estarán marcados con indicadores de conflicto (‘<<<<<<<’, ‘=======’, ‘>>>>>>>’). En el caso de Visual Studio Code tendrá la siguiente apariencia:



Para resolver el conflicto, simplemente vamos a eliminar la frase que introdujimos en *archivo1* en el commit C1, “Hola Mundo desde archivo1”.

Una vez resueltos los conflictos, guarda el archivo, ciérralo y marca el conflicto como resuelto para concluir con el proceso:

```
$ git add archivo1
$ git revert --continue
```

Veamos ahora el historial y el contenido de los archivos:

```
$ git log --oneline
a2b3c6a (HEAD -> main) Revert "C2 - Inicialización de archivo1"
91efd07 C4
e69029b C3
5fdada9 C2
401ca3c C1
```

```
$ cat archivo1
Nueva línea en archivo1

$ cat archivo2
Hola Mundo desde archivo2
```

La frase que introdujimos en el commit que hemos revertido, “Hola Mundo desde archivo1”, ya no existe. Todo lo demás está igual. Misión cumplida.

Para acabar este apartado nos queda por comentar la opción de revertir múltiples commits. Si necesitas revertir varios commits puedes hacerlo como hemos hecho hasta ahora, uno a uno, en cuyo caso crearás tantos commits de *revert* como commits hayas revertido.

Una opción interesante si vas a revertir varios commits es usar la opción `-n`, que crea un solo commit al final. Por ejemplo, partimos de la situación inicial (reinicia de nuevo el repositorio) y pongamos que ahora queremos revertir C2 y C3:

```
$ git log --oneline
cf62fa1 (HEAD -> main) C4
80f8783 C3
49e8ed7 C2
24636b4 C1
```

Cuando revertimos varios commits es importante hacerlo en el orden inverso al que se aplicaron. Esto significa que debes revertir primero el commit más reciente (C3) y luego el anterior (C2). Esto es porque cada *revert* crea un nuevo commit que deshace los cambios del commit específico, y hacerlo en orden inverso asegura que cada reversión se aplique correctamente sin conflictos adicionales debidos a la interdependencia de los commits.

Por tanto, comenzamos revirtiendo C3 y luego C2.

```
$ git revert -n 80f8783
```

Revertir este commit no genera ningún conflicto, podemos continuar con C2:

```
$ git revert -n 49e8ed7
```

Aquí tenemos que resolver el mismo conflicto que resolvimos en el ejemplo anterior. Una vez resuelto (eliminamos la línea “Hola Mundo desde archivo1”), podemos hacer el commit de *revert*:

```
$ git commit -m "Revierte C2 y C3"
[main e7c207c] Revierte C2 y C3
 2 files changed, 1 insertion(+), 3 deletions(-)
```

Ahora, *archivo2* debería estar vacío y *archivo1* debería contener “Nueva línea en archivo1”:

```
$ cat archivo1
Nueva línea en archivo1

$ cat archivo2

```

4. Ramas

Una rama en Git es simplemente un puntero móvil apuntando a un commit. La rama por defecto de Git es la rama *main* (antiguamente *master*). Con el primer commit que hagamos se creará esta rama principal apuntando a dicha confirmación. Con cada commit que realicemos la rama irá avanzando automáticamente.

La rama *main* en Git no es una rama especial, es como cualquier otra. La única razón por la cual aparece en casi todos los repositorios es porque es la que crea por defecto el comando `git init`.

4.1. Crear una rama

Digamos que tenemos un par de commits en nuestro proyecto:

```
git init

echo "file1" > file1
git add .
git commit -m "C1"

echo "file2" > file2
git add .
git commit -m "C2"
```

La situación es la siguiente:

```
$ git log --oneline
5cad13c (HEAD -> main) C2
64e95ff C1
```

Vamos a crear una rama. Cuando creas una nueva rama simplemente se crea un nuevo apuntador para que lo puedas mover libremente. Por ejemplo, crearemos una rama nueva denominada *testing*. Tenemos 3 opciones:

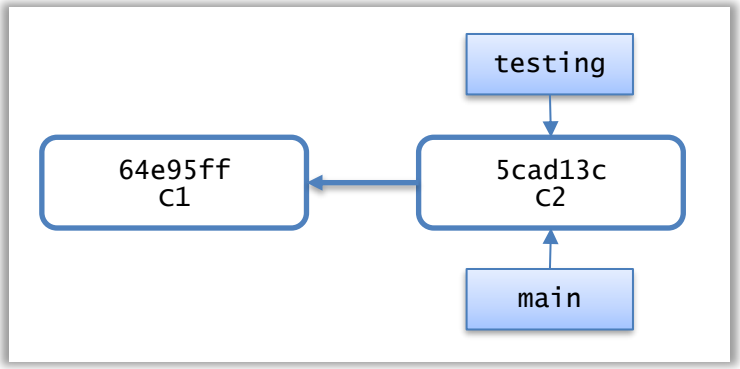
Comando	Descripción
`git branch testing`	Solo crea la rama `testing`, pero no cambia a ella.
`git checkout -b testing`	Crea y cambia a la rama `testing`. Método más antiguo.
`git switch -c testing`	Crea y cambia a la rama `testing`. Disponible desde Git 2.23.

Vamos a crear la rama, pero sin cambiarnos a ella:

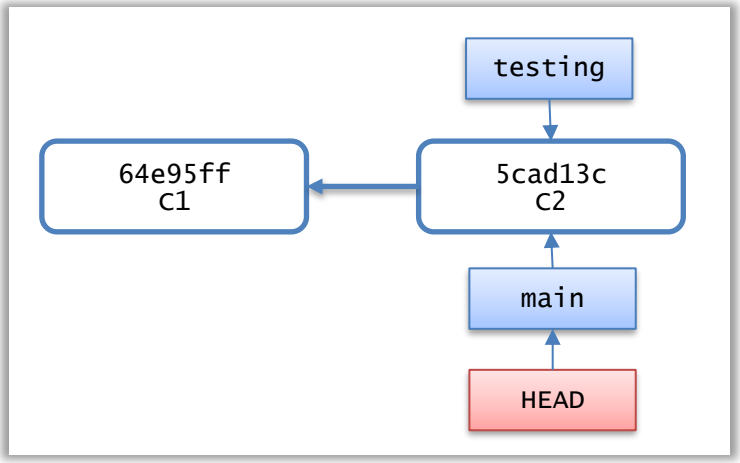
```
$ git branch testing
```

Una vez creada, en la terminal aparece al lado de *main* la nueva rama:

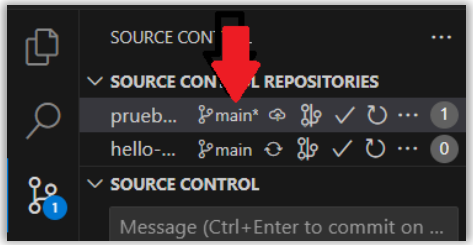
```
$ git log --oneline
5cad13c (HEAD -> main, testing) C2
64e95ff C1
```

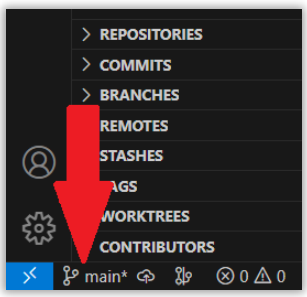


Git sabe en qué rama estás en este momento mediante el puntero ***HEAD***, que apunta a la rama local en la que estés. Fíjate de nuevo en la salida de `git log`, ¿dónde apunta *HEAD*? A la rama *main*, dado que cuando creamos la rama no nos cambiamos a ella.

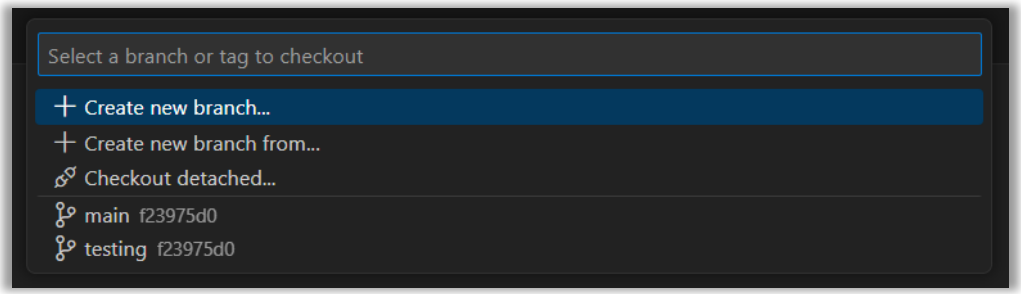


En VSC, crear una rama se puede hacer desde dos puntos, al menos:





En ambos casos, VSC nos pide que seleccionemos alguna de las opciones disponibles para trabajar con ramas:



4.2. Cambiar de rama

Para cambiar de rama e ir a una rama ya existente podemos hacerlo con el comando `git checkout rama` (forma antigua) o `git switch rama` (desde Git 2.23). Ambos comandos realizan dos acciones:

- Mover el apuntador *HEAD* a la rama indicada.
- Modificar el directorio de trabajo, dejándolo tal y como estaba en la última instantánea confirmada de la rama a la que te has movido.

Fíjate en la diferencia de la salida de `git log` antes y después de cambiar de rama:

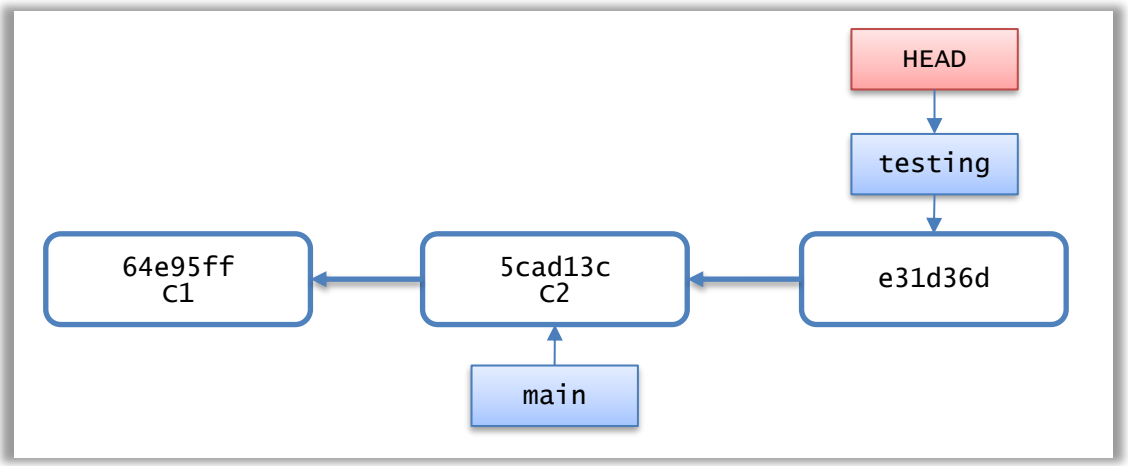
```
~/Desktop/repo-pruebas (main)
$ git log --oneline
5cad13c (HEAD -> main, testing) C2
64e95ff C1

~/Desktop/repo-pruebas (main)
$ git switch testing
Switched to branch 'testing'

~/Desktop/repo-pruebas (testing)
$ git log --oneline
5cad13c (HEAD -> testing, main) C2
64e95ff C1
```

Ahora el puntero HEAD apunta a la rama *testing*. En esta situación vamos a crear un nuevo archivo en nuestro directorio de trabajo y confirmar los cambios en esta nueva rama. Veremos que la rama *testing* avanza mientras *main* se queda atrás:

```
~/Desktop/repo-pruebas (testing)
$ echo "File3" > file3

~/Desktop/repo-pruebas (testing)
$ git add -A
warning: in the working copy of 'file3', LF will be replaced by CRLF the next
time Git touches it
```

```
~/Desktop/repo-pruebas (testing)
$ git commit -m "Commit en la rama testing"
[testing e31d36d] Commit en la rama testing
 1 file changed, 1 insertion(+)
 create mode 100644 file3

~/Desktop/repo-pruebas (testing)
$ git log --oneline
e31d36d (HEAD -> testing) Commit en la rama testing
5cad13c (main) C2
64e95ff C1
```

Ahora la rama *testing* ha avanzado al nuevo commit y el puntero HEAD con ella:



Revisemos los ficheros que tenemos en nuestro directorio de trabajo:

```
~/Desktop/repo-pruebas (testing)
$ ls
file1  file2  file3
```

Ahora volvamos a saltar a la rama *main*:

```
~/Desktop/repo-pruebas (testing)
$ git switch main
Switched to branch 'main'

~/Desktop/repo-pruebas (main)
$ git log --oneline
5cad13c (HEAD -> main) C2
64e95ff C1
```

Ahora el puntero HEAD apunta a *main*. Además, de nuestro directorio de trabajo ha desaparecido el fichero *file3*.

```
~/Desktop/repo-pruebas (main)
$ ls
file1  file2
```

Esta es la segunda consecuencia de cambiar de rama, el directorio de trabajo se traslada al último commit de la rama de destino.

4.3. Visualizar ramas gráficamente

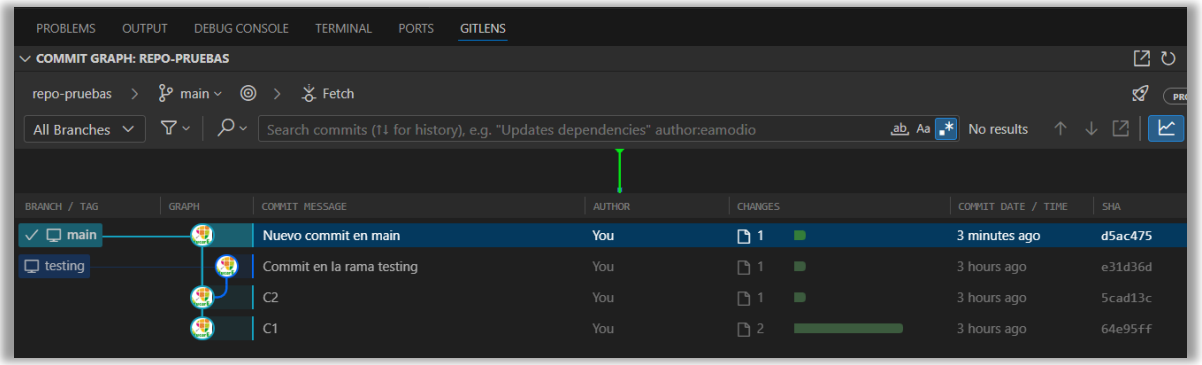
El comando `git log` permite visualizar gráficamente el log de Git aplicando las algunas opciones. Vamos a crear un nuevo archivo y un nuevo commit en la rama *main*:

```
~/Desktop/repo-pruebas (main)
$ echo "file 4" > file4
$ git add -A
$ git commit -m "Nuevo commit en main"
```

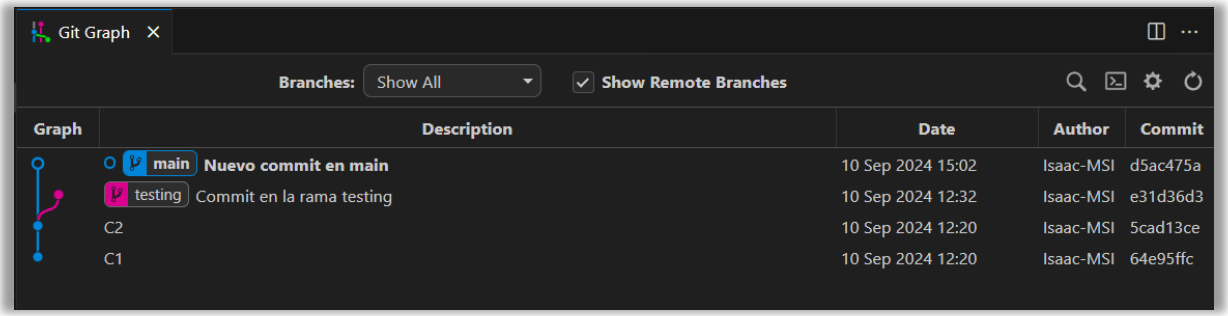
Fíjate ahora en el comando `git log` con las siguientes opciones:

```
~/Desktop/repo-pruebas (main)
$ git log --oneline --decorate --graph --all
* d5ac475 (HEAD -> main) Nuevo commit en main
| * e31d36d (testing) Commit en la rama testing
|/
* 5cad13c C2
* 64e95ff C1
```

En VSC existen al menos un par de formas de visualizar ramas. Una es con GitLens:



Y otra con Git Graph:



4.4. Procedimientos básicos para ramificar

Vamos a presentar un ejemplo simple de ramificar y de fusionar con un flujo de trabajo que se podría presentar en la realidad. Imagina que sigues los siguientes pasos:

- Trabajas en un sitio web.
- Creas una rama para un nuevo tema sobre el que quieres trabajar.
- Realizas algo de trabajo en esa rama.

En este momento, recibes una llamada avisándote de un problema crítico que has de resolver. Y sigues los siguientes pasos:

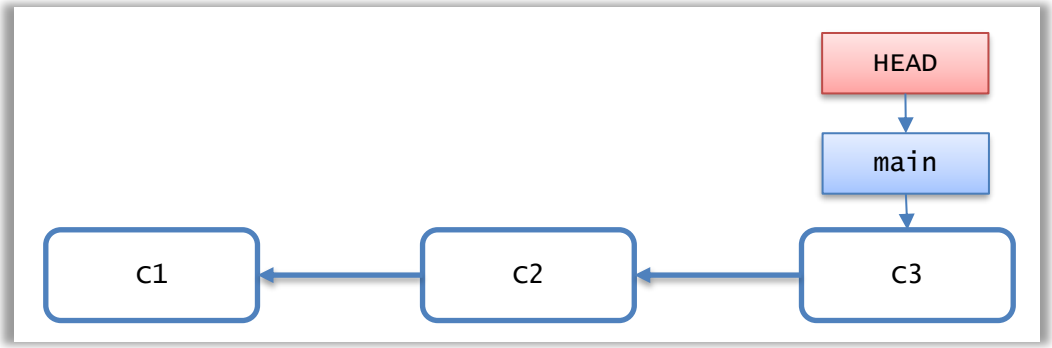
- Vuelves a la rama de producción original.
- Creas una nueva rama para el problema crítico y lo resuelves trabajando en ella.
- Tras las pertinentes pruebas, fusionas (*merge*) esa rama y la envías (*push*) a la rama de producción.
- Vuelves a la rama del tema en que andabas antes de la llamada y continúas tu trabajo.

Vamos a simular este escenario. Estás trabajando en el proyecto y tienes ya 3 commits realizados. Puedes simularlos con los siguientes comandos (usaremos tags en cada commit para identificarlos mejor):

```
git init
echo "Primer commit" > file1
git add -A
git commit -m "C1"
git tag "C1"

echo "Modificación y segundo commit" >> file1
git add -A
git commit -m "C2"
git tag "C2"

echo "Tercer commit" >> file1
git add -A
git commit -m "C3"
git tag "C3"
```

Por lo que estamos en este punto:

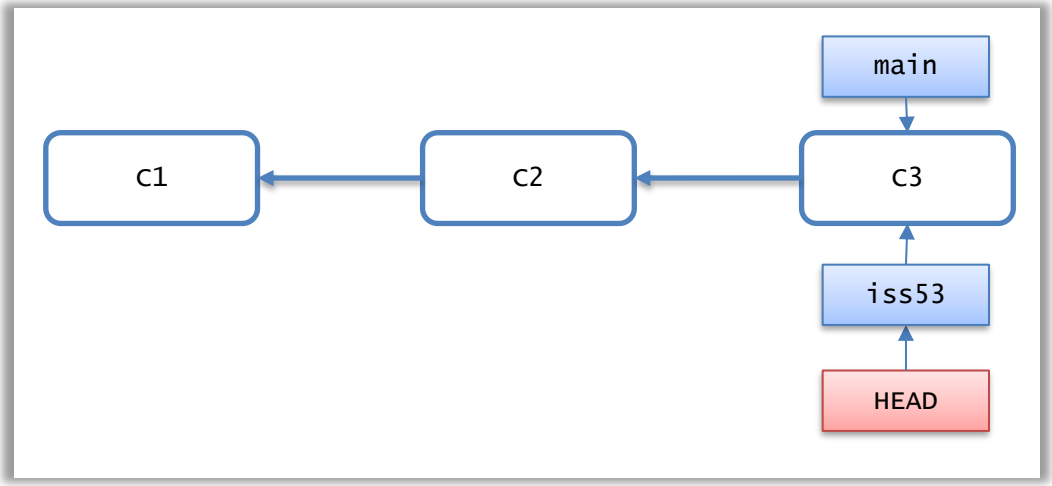
```
$ git log --oneline
* 2d9c526 (HEAD -> main, tag: C3) C3
* 97e9eab (tag: C2) C2
* be28841 (tag: C1) C1
```



Tu jefe te pide que trabajes para corregir el problema *#53*, según el sistema que tu empresa utiliza para llevar el seguimiento de los problemas. Para crear una nueva rama y saltar a ella ya sabemos lo que tenemos que hacer:

```
~/Desktop/repo-pruebas (main)
$ git switch -c iss53
Switched to a new branch 'iss53'

~/Desktop/repo-pruebas (iss53)
$ git log --oneline
2d9c526 (HEAD -> iss53, tag: C3, main) C3
97e9eab (tag: C2) C2
be28841 (tag: C1) C1
```

Ahora, nuestro puntero HEAD apunta a la rama *iss53*, no a *main*, aunque ambas ramas comparten último commit, de momento:



Trabajas en el proyecto y haces algunos commits. Con ello avanzas la rama *iss53*, que es en la que estás en este momento (es decir, a la que apunta HEAD).

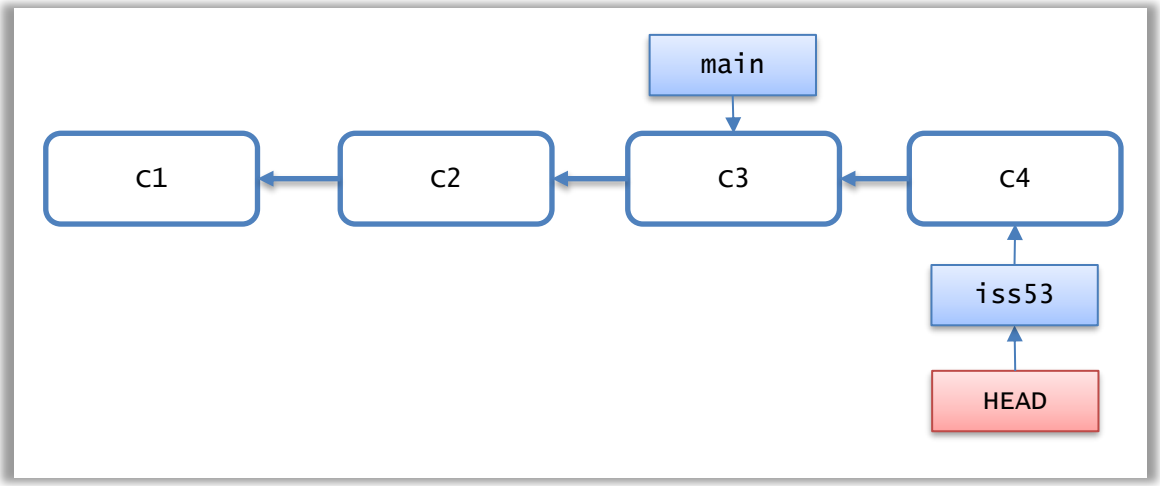
Simularemos todo esto con un solo commit:

```
~/Desktop/repo-pruebas (iss53)
$ echo "Modifico file1 en la rama iss53" >> file1
$ git add -A
$ git commit -m 'Modifica file1 en iss53'
$ git tag "C4"
```

Esta es la situación actual:

```
~/Desktop/repo-pruebas (iss53)
$ git log --oneline
26a0a81 (HEAD -> iss53, tag: C4) Modifica file1 en iss53
2d9c526 (tag: C3, main) C3
97e9eab (tag: C2) C2
be28841 (tag: C1) C1
```

Gráficamente:



Hace varios minutos que hiciste el último commit y has avanzado cosas en tu rama iss53. Simularemos esto añadiendo una línea a *file1* y creando *file2*:

```
~/Desktop/repo-pruebas (iss53)
$ echo "Sigo editando file1 en iss53" >> file1
$ echo "Creando file2 desde la rama iss53 " > file2
```

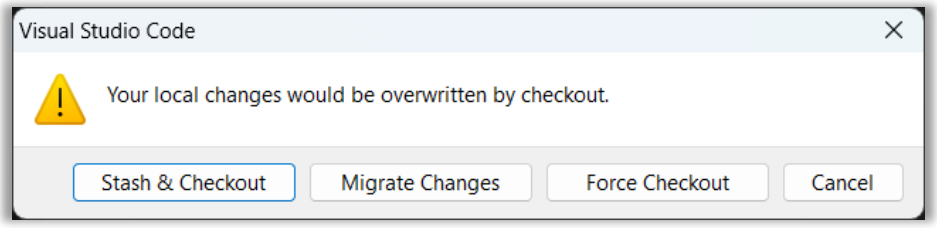
De repente, tu jefe te llama avisándote de otro problema urgente en el sitio web que debes resolver inmediatamente. Con Git no necesitas mezclar el nuevo problema con los cambios que ya habías realizado sobre el problema *#53* ni tampoco perder tiempo revirtiendo esos cambios para poder trabajar sobre el contenido que está en producción. Basta con saltar a la rama *main* y continuar trabajando desde allí.

Pero no puedes saltar a la rama *main* directamente. Como tienes archivos modificados desde que hiciste tu ultimo commit, Git no te dejará. **Si tienes cambios aún no confirmados en el directorio de trabajo o en el área de preparación, Git no te permitirá saltar a otra rama con la que podrías tener conflictos**.

Si intentamos saltar a la rama *main*, Git no nos lo permitirá:

```
$ git switch main
error: Your local changes to the following files would be overwritten by
checkout:
        file1
Please commit your changes or stash them before you switch branches.
Aborting
```

En VCS, al intentar cambiar de rama, nos dice:



Estas opciones de las que nos habla VSC son posibles soluciones que se escapan del objeto de este documento. Por el momento, quédate con que **lo mejor es tener siempre un estado de trabajo limpio y despejado antes de saltar entre ramas**. Así pues, haz otro commit para confirmar tus últimos cambios:

```
~/Desktop/repo-pruebas (iss53)
$ git add -A
$ git commit -m "Modifica file1 y añade file2 en iss53"
$ git tag "C5"
```

Quédate con cómo está tu *working directory* en estos momentos:

```
$ ls
file1  file2

$ cat file1
Primer commit
Modificación y segundo commit
Tercer commit
Modifico file1 en la rama iss53
Sigo editando file1 en iss53

$ cat file2
Creando file2 desde la rama iss53
```

Ahora sí, como tenemos confirmados todos los cambios podemos saltar a la rama *main* sin problemas:

```
~/Desktop/repo-pruebas (iss53)
$ git switch main
Switched to branch 'main'
```

Fíjate cómo ha cambiado tu *working directory*:

```
~/Desktop/repo-pruebas (main)
$ ls
file1

~/Desktop/repo-pruebas (main)
$ cat file1
Primer commit
Modificación y segundo commit
Tercer commit
```

Ahora tu *file1* es distinto y *file2* ni siquiera existe.

Es importante recordar que Git revierte el directorio de trabajo exactamente al estado en que estaba en el último commit de la rama a la que saltamos. **Git añade, quita y modifica archivos automáticamente para asegurar que tu copia de trabajo está exactamente como estaba la rama en el último commit realizado sobre ella**.

A continuación, es momento de resolver el problema urgente. Vamos a crear una nueva rama *hotfix*, sobre la que trabajar hasta resolverlo.

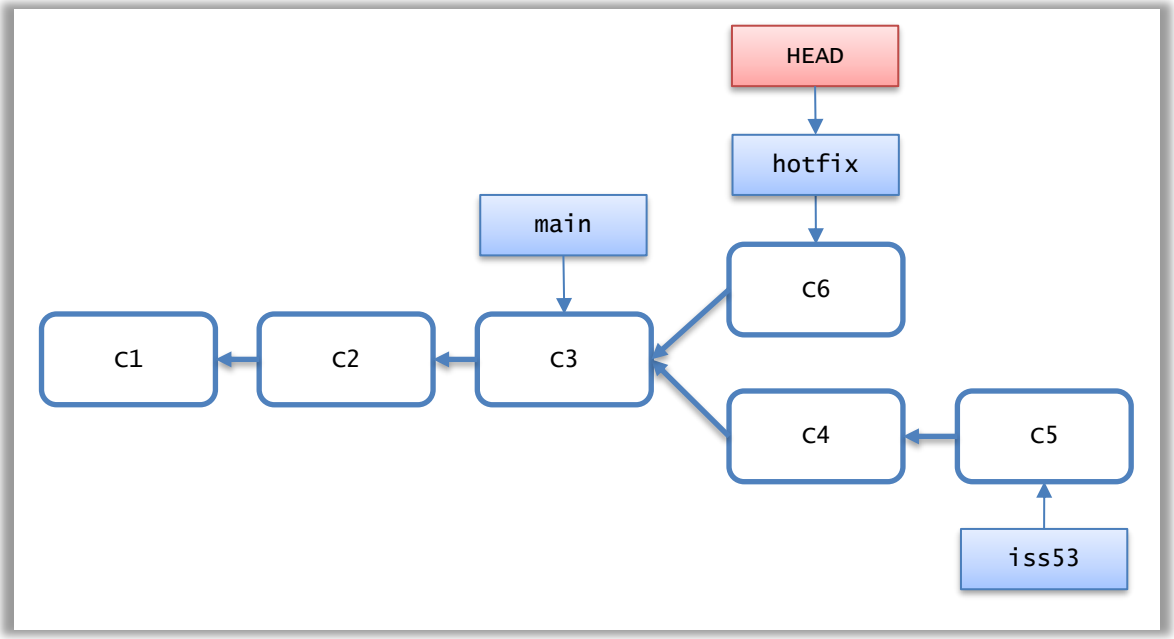
```
$ git switch -c hotfix
Switched to a new branch 'hotfix'
```

Ahora, en la recién creada rama, trabajamos para resolver el problema.

```
~/Desktop/repo-pruebas (hotfix)
$ echo "Modifica file1 en la rama hotfix" >> file1
$ git add -A
$ git commit -m 'Modifica file1 en la rama hotfix'
$ git tag 'C6'
```

Ahora la situación es:

```
~/Desktop/repo-pruebas (hotfix)
$ git log --oneline --graph --all
* 28df5fc (HEAD -> hotfix, tag: C6) Modifica file1 en la rama hotfix
| * a125580 (tag: C5, iss53) Modifica file1 y añade file2 en iss53
| * 40ad2b4 (tag: C4) Modifica file1 en iss53
|/
* b228c9f (tag: C3, main) C3
* 62a3609 (tag: C2) C2
* 5f27461 (tag: C1) C1
```



Una vez arreglado el problema en la rama *hotfix*, podemos fusionarla con *main*.

4.5. Procedimientos básicos para fusionar

Vamos a fusionar *hotfix* en *main* sin *fast-forward*. Hablaremos de la opción `--no-ff` y de qué es *Fast-Forwarding* más adelante en esta sección.

Para fusionar una rama dentro de otra, primero debemos movernos a la rama destino, en nuestro caso, *main*:

```
$ git switch main
Switched to branch 'main'
```

Ahora ejecutamos el merge:

```
~/Desktop/repo-pruebas (main)
$ git merge --no-ff hotfix
```

Se abrirá el editor predeterminado para incluir un mensaje, dado que vamos a crear un *commit de merge*.

```
Merge branch 'hotfix'
# Please enter a commit message to explain why this merge is necessary,
# especially if it merges an updated upstream into a topic branch.
#
# Lines starting with '#' will be ignored, and an empty message aborts
# the commit.
```

Cuando guardemos y cerremos el editor, se habrá creado el *commit de merge*. Vamos a ponerle un tag:

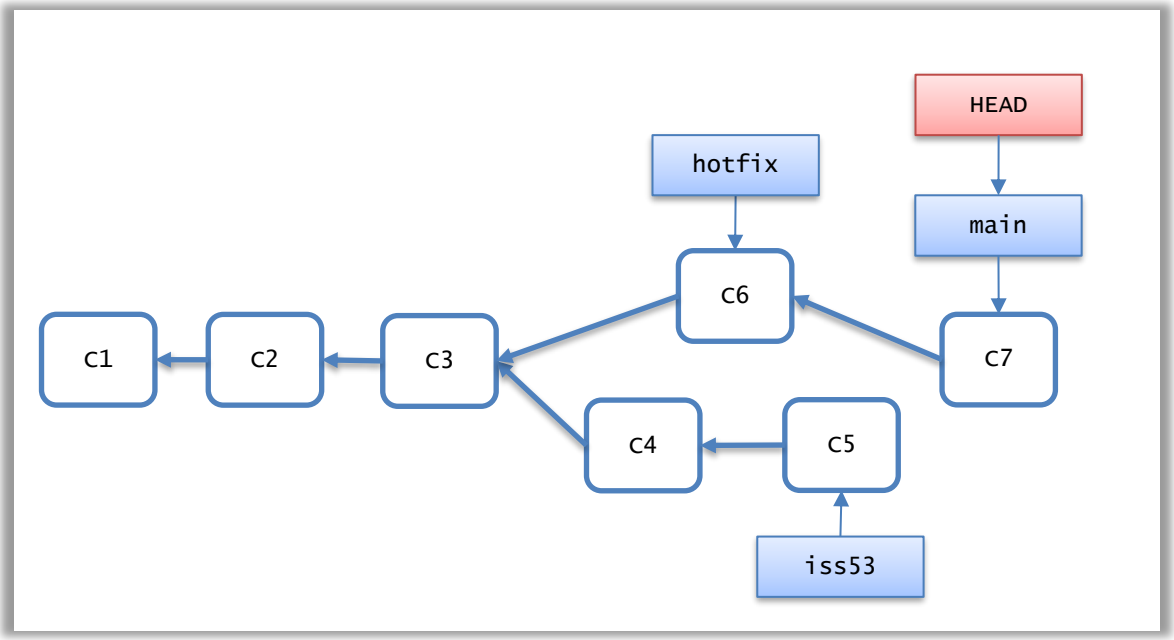
```
$ git tag "C7"
```

Fíjate ahora en la salida de `git log`:

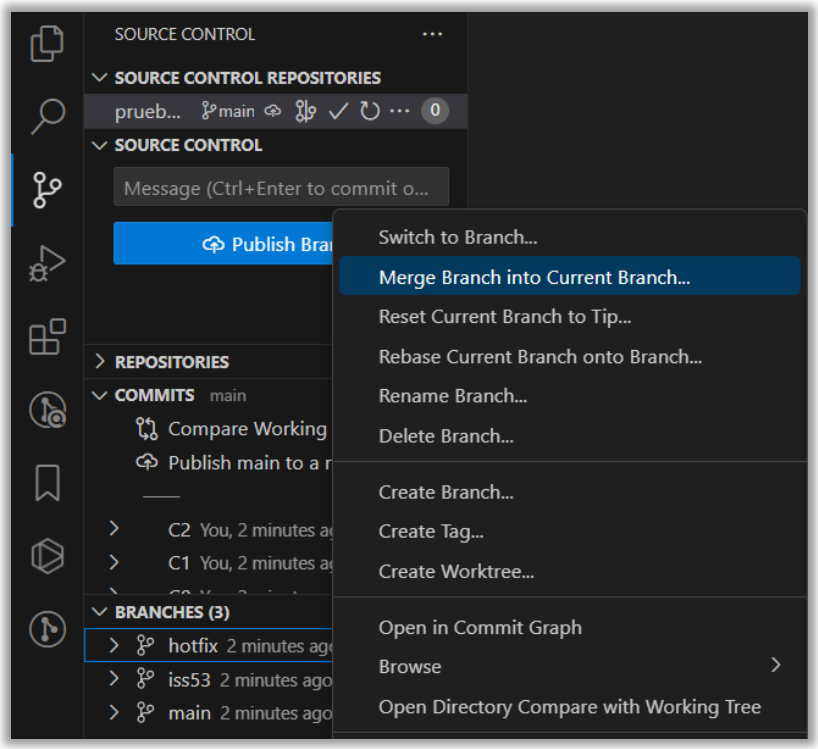
```
$ git log --oneline --graph --all
*   f279b4a (HEAD -> main, tag: C7) Merge branch 'hotfix'
|\
| * 28df5fc (tag: C6, hotfix) Modifica file1 en la rama hotfix
|/
| * a125580 (tag: C5, iss53) Modifica file1 y añade file2 en iss53
| * 40ad2b4 (tag: C4) Modifica file1 en iss53
|/
* b228c9f (tag: C3) C3
* 62a3609 (tag: C2) C2
* 5f27461 (tag: C1) C1
```

La fusión habrá creado un commit nuevo en *main*. A este tipo de commits se les suele denominar *commit de merge*.

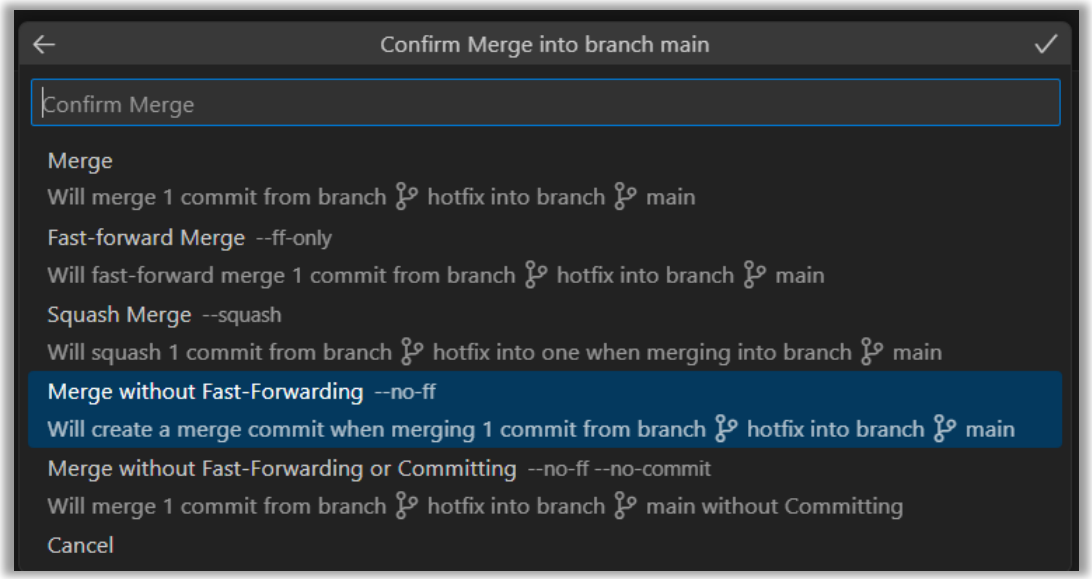
Gráficamente, la situación es la siguiente:



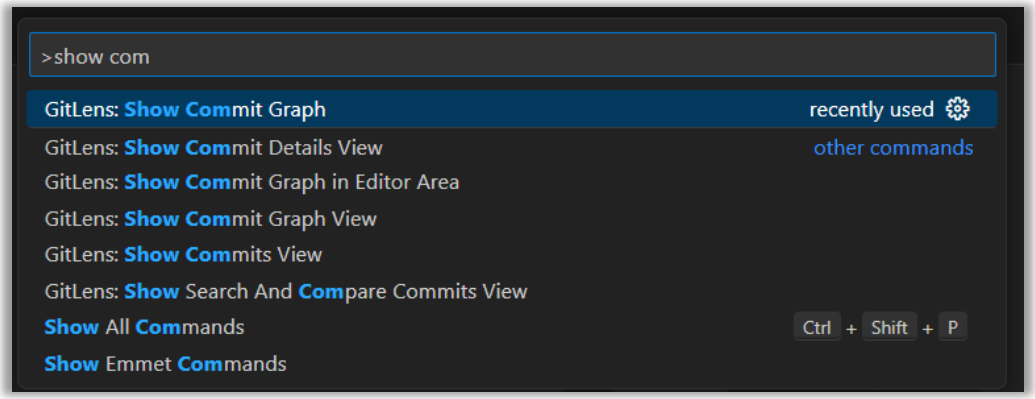
Para hacer este procedimiento en VSC, despliega la sección "BRANCHES", haz clic derecho sobre la rama que quieres fusionar y selecciona "*Merge branch into current branch*".



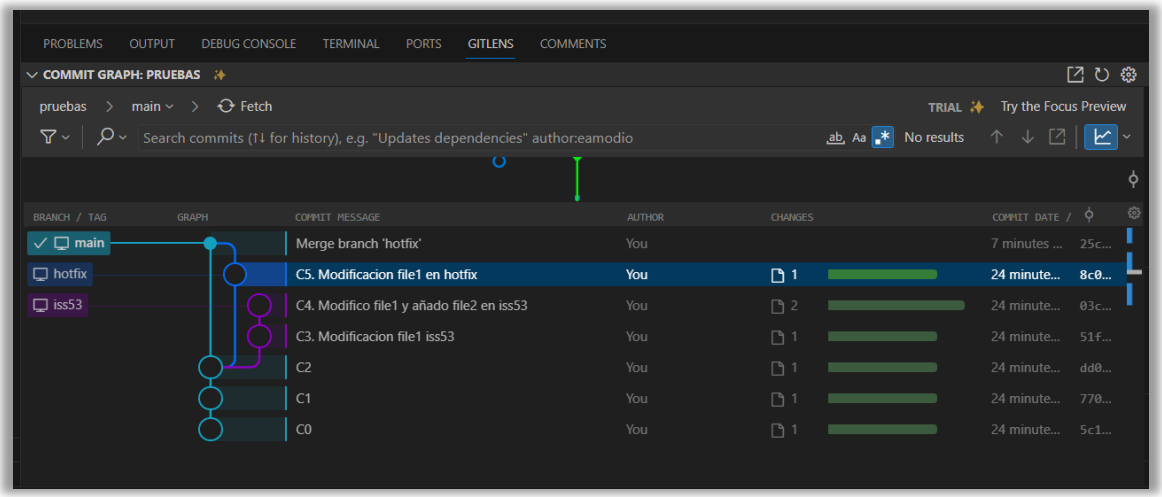
De entre las opciones de fusión, selecciona "*Merge without Fast-Forwarding*":



En VSC podemos acceder a la opción gráfica de GitLens pulsando *Ctrl+Shift+P* y buscando la opción "*Show Commit Graph*":



Y veremos gráficamente nuestro historial de commits, las ramas abiertas y las ramas cerradas:



4.6. Borrar ramas

Tras haber resuelto el problema urgente que había interrumpido tu trabajo, puedes volver a donde estabas. Pero antes quizá quieras borrar la rama *hotfix*, ya que no la vas a necesitar más. Esto lo puedes hacer con la opción `-d` del comando `git branch`.

Ahora tenemos 3 ramas:

```
~/Desktop/repo-pruebas (main)
$ git branch
  hotfix
  iss53
* main
```

Eliminamos *hotfix*:

```
~/Desktop/repo-pruebas (main)
$ git branch -d hotfix
Deleted branch hotfix (was 7ad9c67).
```

Y ya solo nos quedan 2:

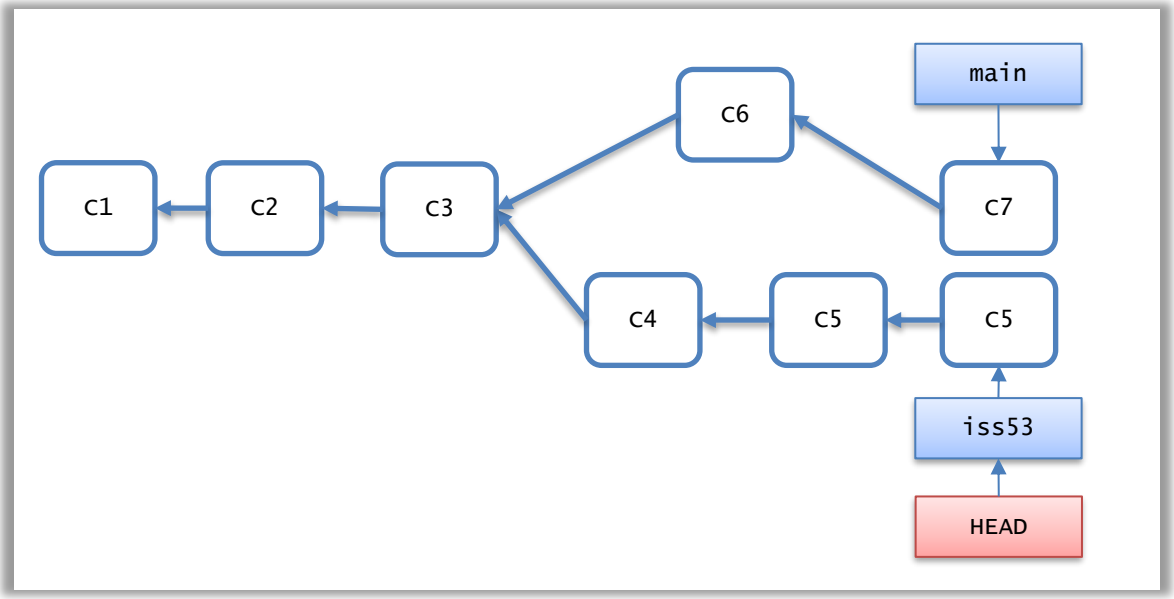
```
~/Desktop/repo-pruebas (main)
$ git branch
  iss53
* main
```

Con esto, ya estás listo para regresar al trabajo sobre el problema *#53*. Volvemos a la rama *iss53* y creamos otro commit:

```
~/Desktop/repo-pruebas (main)
$ git switch iss53

~/Desktop/repo-pruebas (iss53)
$ echo "Modificación 2 por iss53 en file1" >> file1
$ git add -A
$ git commit -m 'Modifica file1 en la rama iss53'
$ git tag "C8"
```

Ahora la situación es la siguiente:

```
$ git log --oneline --graph --all
* 1de4a98 (HEAD -> iss53, tag: C8) Modifica file1 en la rama iss53
* a125580 (tag: C5) Modifica file1 y añade file2 en iss53
* 40ad2b4 (tag: C4) Modifica file1 en iss53
| * f279b4a (tag: C7, main) Merge branch 'hotfix'
|/|
| * 28df5fc (tag: C6) Modifica file1 en la rama hotfix
|/
* b228c9f (tag: C3) C3
* 62a3609 (tag: C2) C2
* 5f27461 (tag: C1) C1
```



Cabe destacar que todo el trabajo realizado en la rama *hotfix* no está en los archivos de la rama *iss53*. Si fuera necesario agregarlos, puedes fusionar (*merge*) la rama *main* sobre la rama *iss53* o puedes esperar hasta que decidas fusionar la rama *iss53* sobre la rama *main*.

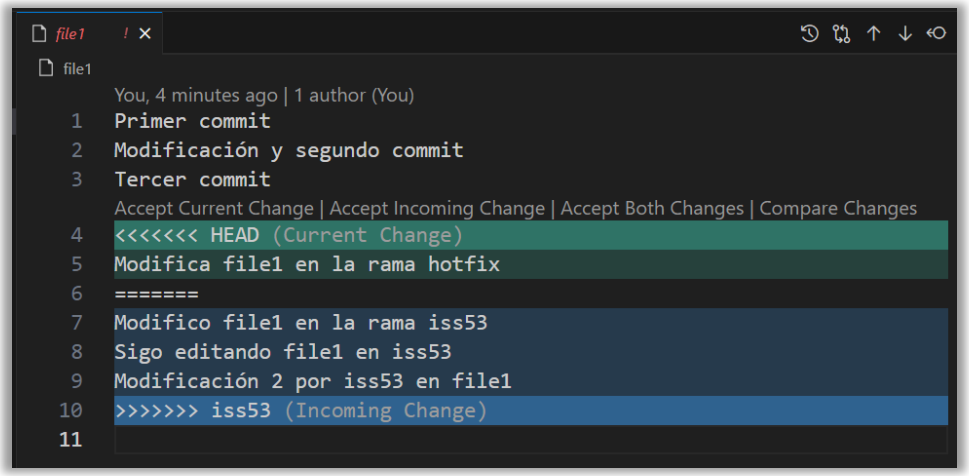
Supongamos que has terminado de arreglar el *issue #53*. Estás en disposición de fusionar la rama iss53 sobre *main*. Ya sabemos el proceso, primero nos cambiamos a la rama destino y luego hacemos la fusión:

```
~/Desktop/repo-pruebas (iss53)
$ git switch main
Switched to branch 'main'

~/Desktop/repo-pruebas (main)
$ git merge --no-ff iss53
Auto-merging file1
CONFLICT (content): Merge conflict in file1
Automatic merge failed; fix conflicts and then commit the result.
```

El mensaje que nos está dando Git es que hay conflictos en *file1*, es decir, que las versiones del archivo en *main* y en iss53 no son compatibles. Tenemos que resolver el conflicto manualmente editando el archivo.

Para resolver el conflicto, tendremos que abrir el archivo, buscar las secciones con conflictos, editarlas para decidir qué partes del código conservar, y luego realizar un commit con los cambios resueltos.

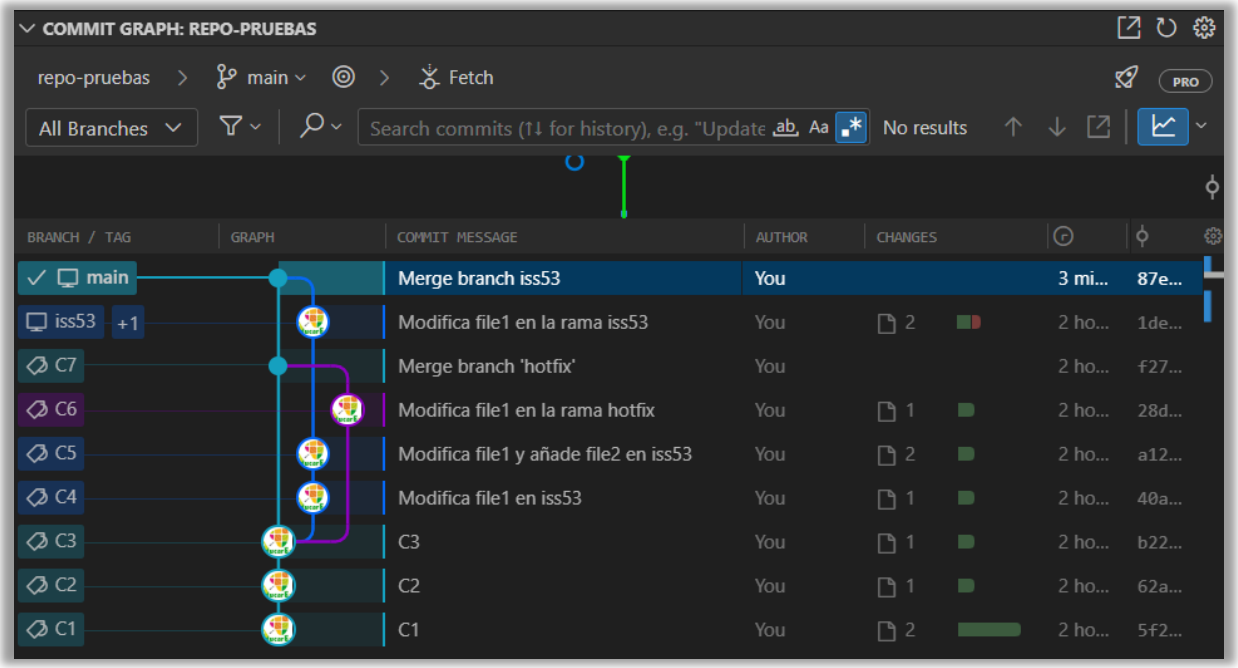
Si abres el archivo con VSC observarás lo siguiente:

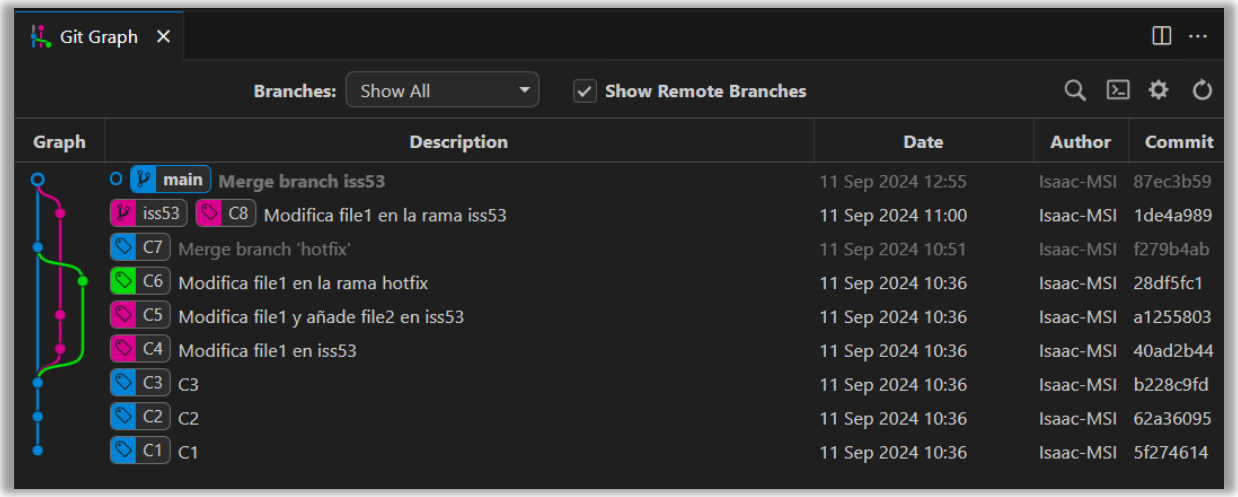


Quizá parezca un poco extraño, pero simplemente nos está diciendo qué líneas se añadieron y cuándo. Una vez resueltos los conflictos, guardamos el archivo, añadimos todo al índice y ejecutamos el commit.

```
~/Desktop/repo-pruebas (main|MERGING)
$ git add -A

~/Desktop/repo-pruebas (main|MERGING)
$ git commit -m "Merge branch iss53"
[main 87ec3b5] Merge branch iss53

~/Desktop/repo-pruebas (main)
$ git log --oneline --graph --all
*   87ec3b5 (HEAD -> main) Merge branch iss53
|\
| * 1de4a98 (tag: C8, iss53) Modifica file1 en la rama iss53
| * a125580 (tag: C5) Modifica file1 y añade file2 en iss53
| * 40ad2b4 (tag: C4) Modifica file1 en iss53
* |   f279b4a (tag: C7) Merge branch 'hotfix'
|\ \
| |/
|/|
| * 28df5fc (tag: C6) Modifica file1 en la rama hotfix
|/
* b228c9f (tag: C3) C3
```

Podemos observar cómo ha quedado el repositorio con Git Lens o Git Graph.





Para ver un ejemplo más detallado, revisa la sección ejemplo de resolución de conflictos.

5. Ejemplo de resolución de conflictos

Vamos a ejemplificar cómo resolver conflictos al fusionar ramas. Para ello, vamos a suponer que 3 personas trabajan sobre una página web, cada una con un cometido y en una rama distinta.

5.1. Arranque del proyecto

El proyecto arranca con dos ficheros: index.html y estilos.css. El fichero CSS está vacío, mientras que el fichero index.html contiene una estructura HTML base:

```
<!DOCTYPE html>
<html lang="es">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Ejemplo resolución de conflictos en Git</title>
  <link rel="stylesheet" href="estilos.css">
</head>

<body>
</body>

</html>
```

En este punto hacemos un commit inicial:

```
$ git log --oneline
4202ec0 (HEAD -> main) Añade estructura inicial index.html
```

Ahora separamos el proyecto en 3 ramas, en las que cada uno trabajará por separado durante un tiempo.

Programador	Rama	Tarea
Pepe	*navbar-branch*	Desarrollar la barra de navegación.
María	*main-branch*	Desarrollar la parte central de la página (main).
Juan	*footer-branch*	Desarrollar el pie de la página (*footer*).

5.2. Trabajo en las ramas

5.2.1. navbar-branch

Pepe es el primero que termina su parte, la rama navbar-branch.

index.html en la rama navbar-branch:

```
<!DOCTYPE html>
<html lang="es">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
  <link rel="stylesheet" href="estilos.css">
</head>

<body>
  <nav>Cabecera</nav>
</body>

</html>
```

estilos.css en la rama navbar-branch:

```
nav {
  padding: 20px;
  font-size: 1.25rem;
  background-color: aqua;
}
```

Pepe hace un commit en su rama:

```
~/Desktop/repo-pruebas (navbar-branch)
$ git log --oneline
f19548b (HEAD -> navbar-branch) Commit navbar-branch
4202ec0 (main-branch, main, footer-branch) Añade estructura inicial
index.html
```

5.2.2. footer-branch

Juan, trabajando en su rama footer-branch, también termina. Este es el estado de los dos archivos:

index.html en la rama footer-branch:

```
<!DOCTYPE html>
<html lang="es">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Ejemplo resolución de conflictos en Git</title>
  <link rel="stylesheet" href="estilos.css">
</head>

<body>
  <footer>
    <div>Práctica 1</div>
    <div>Autores: Pepe, María y Juan</div>
  </footer>
</body>

</html>
```

estilos.css en la rama footer-branch:

```
footer {
  display: flex;
  justify-content: space-between;
  background-color: black;
  color: white;
  font-size: 1.5rem;
  padding: 10px 20px;
}
```

Juan hace un commit en su rama:

```
~/Desktop/repo-pruebas (footer-branch)
$ git log --oneline
28133fb (HEAD -> footer-branch) Commit footer-branch
4202ec0 (main-branch, main) Añade estructura inicial index.html
```

5.2.3. main-branch

Por último, María termina su sección en main-branch:

index.html en la rama main-branch:

```
<!DOCTYPE html>
<html lang="es">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Ejemplo resolución de conflictos en Git</title>
  <link rel="stylesheet" href="estilos.css">
</head>

<body>
  <div class="wrapper">
    <main>
      <h1>Lorem ipsum</h1>
      <p>Lorem ipsum, dolor sit amet consectetur adipisicing elit. Pariatur voluptas, consequuntur quo animi at, magnam, ducimus iste tempore atque distinctio modi harum ut impedit dolor totam deserunt consequatur delectus aperiam?</p>
      <p>Debitis rem velit laborum. Hic autem sequi fugit enim suscipit odio aliquid iste, repellendus non, possimus voluptatem labore nihil minima libero natus illum quasi maiores tempora quis, ab esse nisi.</p>
      <p>Voluptatibus fugiat animi non deserunt! Eos nobis voluptatibus dolorem fugiat id tenetur alias quae non, sapiente voluptates exercitationem at sed quasi, maiores officia excepturi iusto error, rerum reiciendis aperiam. Maxime!</p>
    </main>
    <aside>
      <p>Soy un gatito</p>
      <img src="gato.jpeg" alt="Imagen de un gato">
    </aside>
  </div>
</body>

</html>
```

estilos.css en la rama footer-branch:

```
* {
  box-sizing: border-box;
}

.wrapper {
  width: 992px;
  margin: 0 auto;
  display: flex;
  flex-direction: row;
  align-items: center;
}
```

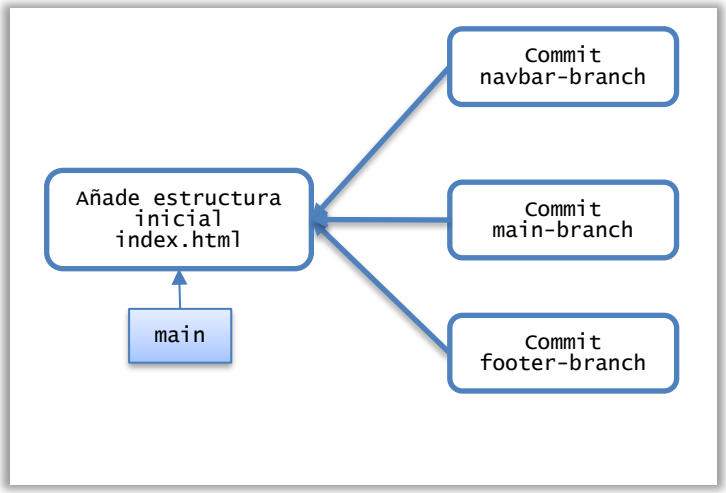
```
main {
  padding: 0px 20px;
  width: 70%;
}

aside {
  width: 30%;
}
```

Además, María añade al proyecto una imagen llamada “gato.jpg”.

María hace un commit en su rama:

```
~/Desktop/repo-pruebas (main-branch)
$ git log --oneline
8fc5885 (HEAD -> main-branch) Commit main-branch
4202ec0 (main) Añade estructura inicial index.html
```

Por tanto, nuestro proyecto tiene el siguiente aspecto ahora mismo:



5.3. Fusionando ramas

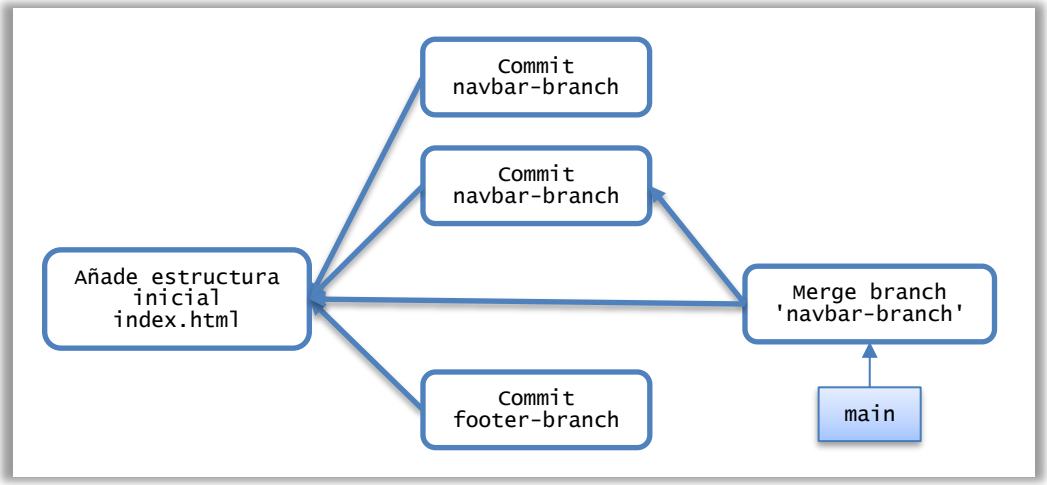
5.3.1. Fusión de navbar-branch en main

Como Pepe fue el que primero terminó su rama, él es el primero en fusionar su rama con la rama main (recuerda hacer la fusión sin la opción *fast-forward*). Como no hay conflictos, Git no nos lo indica.

Si miramos el log en la rama *main* veremos el commit que Pepe realizó en la rama navbar-branch:

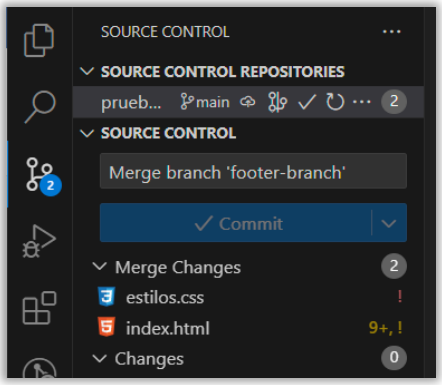
```
$ git log --oneline
0a09b86 (HEAD -> main) Merge branch 'navbar-branch'
f19548b (navbar-branch) Commit navbar-branch
4202ec0 Añade estructura inicial index.html
```

La situación del proyecto ahora es:

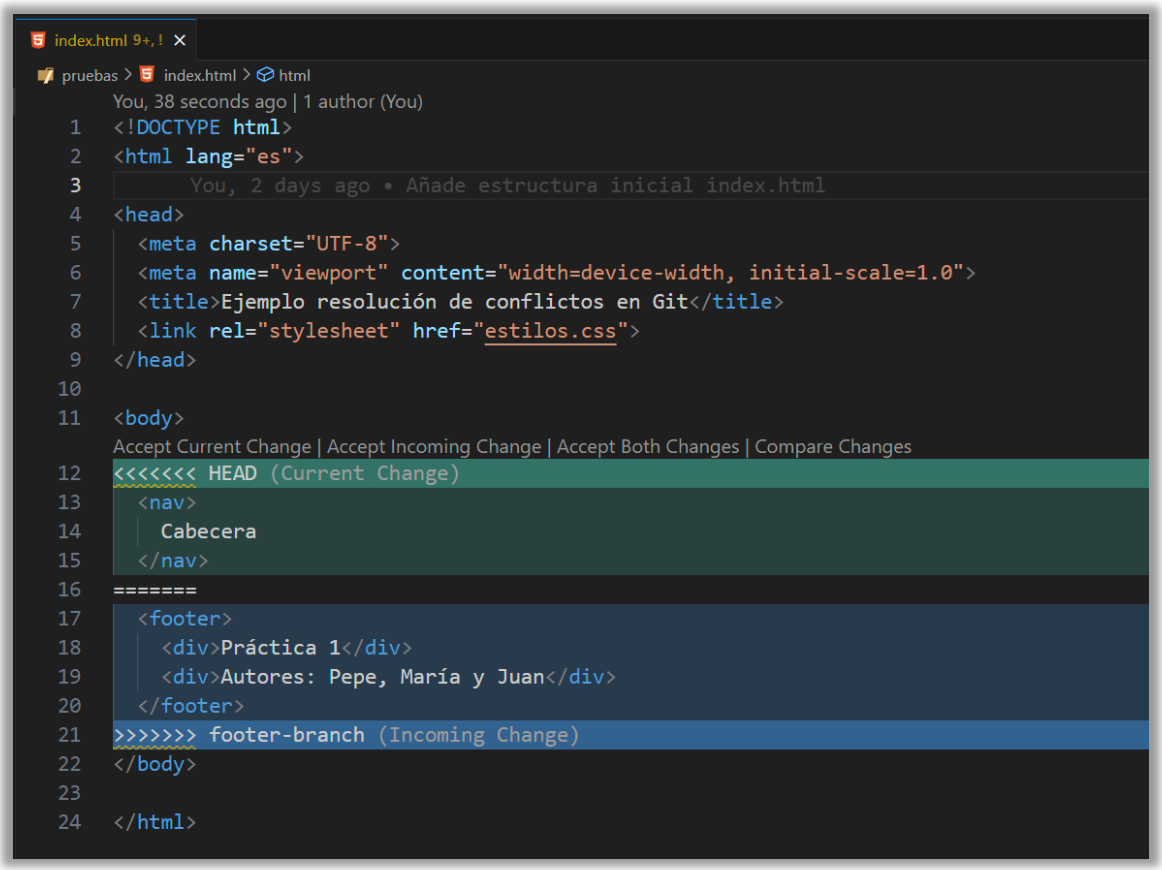


5.3.2. Fusión de footer-branch en main. Primeros conflictos

Juan es el segundo en terminar su trabajo, por lo que es el siguiente en fusionar su rama con main. Git en esta ocasión sí detectará conflictos (recuerda hacer la fusión sin la opción *fast-forward*) y no nos dejará ejecutar el commit hasta que resolvamos los conflictos (fíjate en los iconos en los archivos):

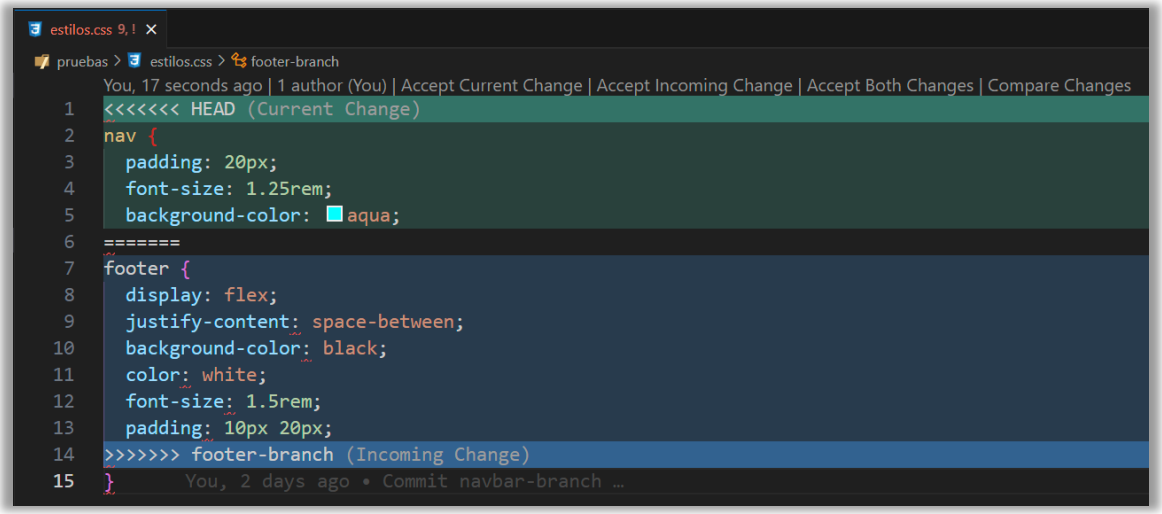


Si abrimos los ficheros, VSC nos informa de los conflictos.



En este caso, aceptaremos los dos cambios (*accept both changes*).

Haremos lo mismo con el archivo css:

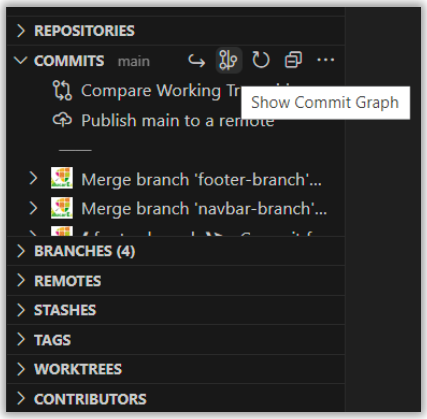


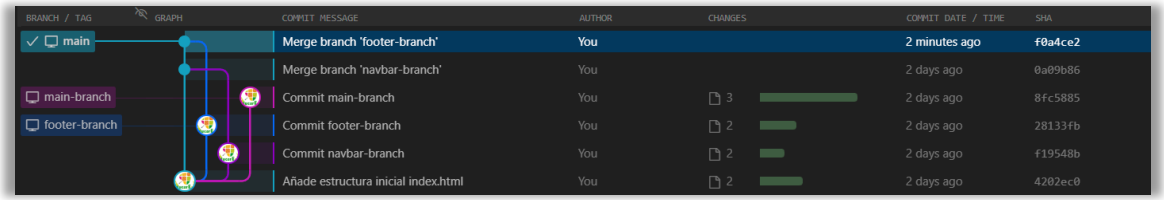
Una vez resueltos los conflictos, simplemente tendremos que añadir los dos archivos al *staged area* y completar el commit.

Ahora tenemos en nuestra rama main los commits de las ramas fusionadas:

```
$ git log --oneline
f0a4ce2 (HEAD -> main) Merge branch 'footer-branch'
0a09b86 Merge branch 'navbar-branch'
28133fb (footer-branch) Commit footer-branch
f19548b (navbar-branch) Commit navbar-branch
4202ec0 Añade estructura inicial index.html
```

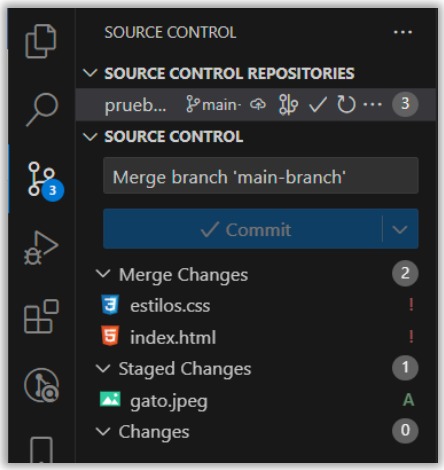
Podemos ver esto mejor en VSC con GitLens haciendo clic en *Show Commit Graph* en la pestaña “Commits”:



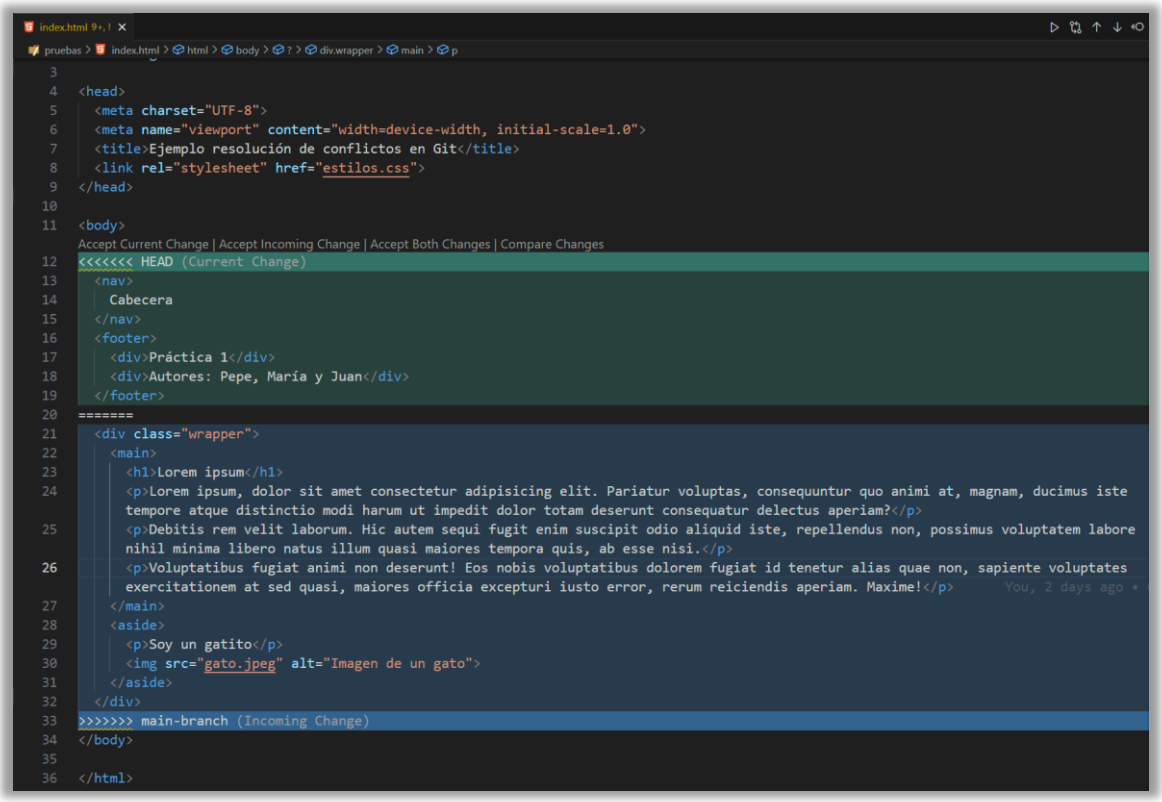


5.3.3. Fusión de main-branch en main. Más conflictos

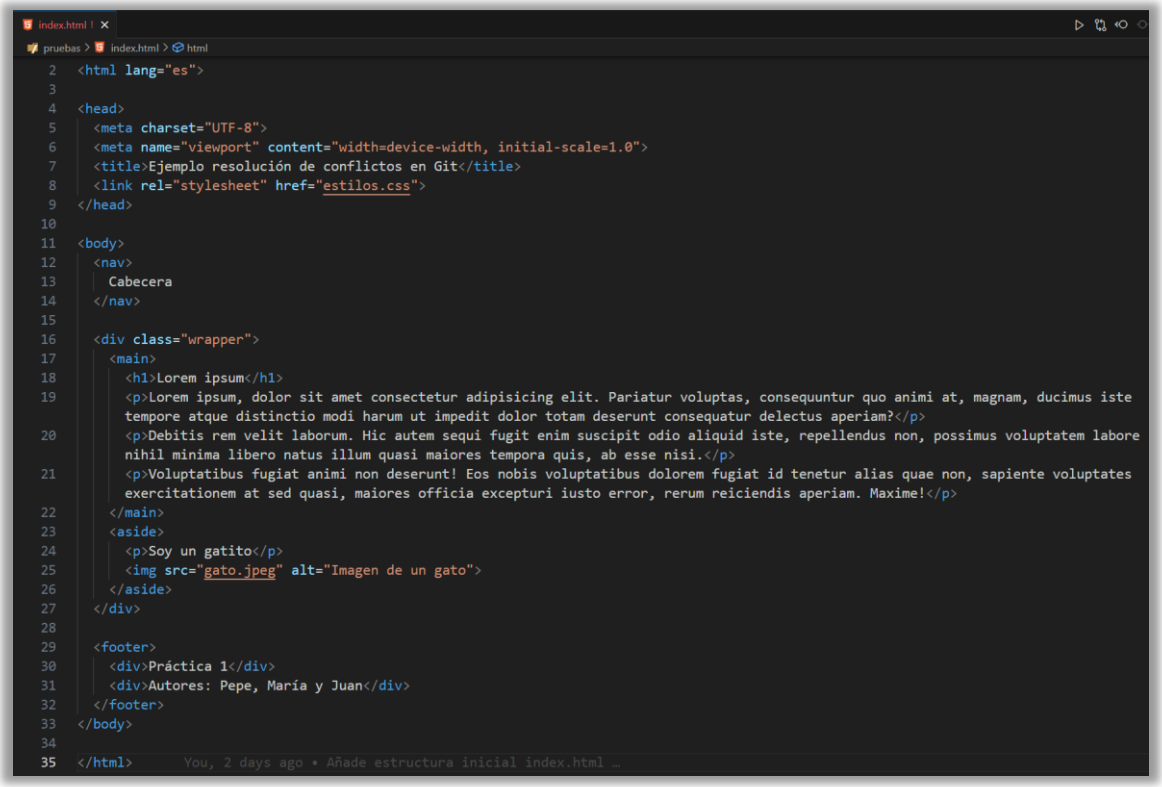
Ya solo nos queda fusionar la rama main-branch, que fue la última en terminar. Volviendo a hacer el mismo proceso de fusión (sin *fast-forward*), Git nos avisará de que hay conflictos de nuevo sin resolver y un archivo a añadir (gato.jpeg).



En el archivo estilos.css podemos aceptar los cambios de las dos ramas (*accept both changes*) sin problema, pero en index.html no.



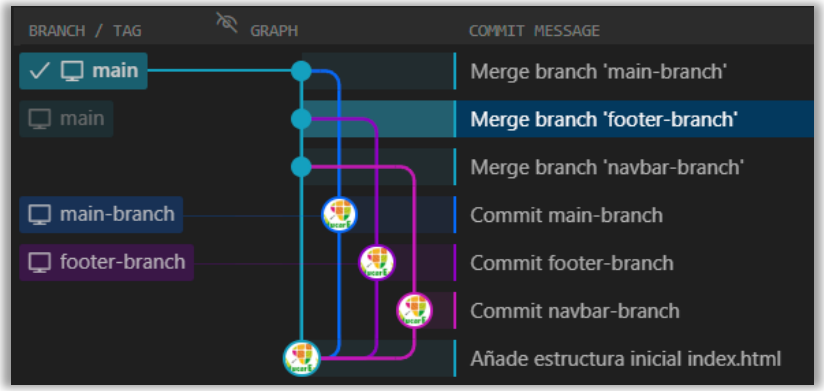
Si aceptásemos los cambios tal cual, tendríamos el footer antes del main, lo que es absurdo. Tenemos que reordenar nuestro código antes de guardarlo:



Ahora ya sí, guardamos el fichero, añadimos todo al *staged area* y finalizamos el commit. Fíjate en el log:

```
git log --oneline
40db76d (HEAD -> main) Merge branch 'main-branch'
f0a4ce2 Merge branch 'footer-branch'
0a09b86 Merge branch 'navbar-branch'
8fc5885 (main-branch) Commit main-branch
28133fb (footer-branch) Commit footer-branch
f19548b (navbar-branch) Commit navbar-branch
4202ec0 Añade estructura inicial index.html
```

O en el gráfico de GitLens:



Ya tenemos nuestro proyecto finalizado.

El código del ejemplo lo tienes disponible en el siguiente repositorio de GitHub:

https://github.com/isaac-exposito/DIW-Git-Ej-fusion-ramas

6. Moviéndonos entre commits: `git checkout`

El comando `git checkout` era muy versátil, se usaba tanto para cambiar de ramas como para restaurar archivos o moverse a un commit específico. Esto generaba confusión, ya que el mismo comando hacía varias cosas con diferentes parámetros. Con Git 2.23 (lanzada en 2019), los desarrolladores decidieron descomponer estas funcionalidades en dos comandos más específicos: `git switch` para cambiar entre ramas y `git restore` para restaurar archivos o el estado del repositorio.

En esta sección vamos a ver cómo utilizar `git checkout` para movernos entre commits del repositorio y poder observar cómo estaba el proyecto en un momento concreto del historial.

Para ejemplificarlo, ejecuta los siguientes comandos para crear un repositorio:

```
git init
echo "Hola Mundo desde archivo1" >> archivo1
git add -A
git commit -m "C1"
git tag C1

echo "Hola Mundo desde archivo2" >> archivo2
git add -A
git commit -m "C2"
git tag C2

echo "Hola Mundo desde archivo3" >> archivo3
git add -A
```

Hemos creado 3 archivos, pero solo hemos confirmado 2, dado que el tercero lo tenemos en el índice. La situación es esta:

```
$ git log --oneline
66e53ff (HEAD -> main, tag: C2) C2
883c39a (tag: C1) C1

$ git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   archivo3

$ ls -la
total 55
drwxr-xr-x 1 isaac 197609   0 Sep  8 19:00 ./
drwxr-xr-x 1 isaac 197609   0 Sep  8 17:42 ../
drwxr-xr-x 1 isaac 197609   0 Sep  8 19:01 .git/
-rw-r--r-- 1 isaac 197609  26 Sep  8 19:00 archivo1
-rw-r--r-- 1 isaac 197609  26 Sep  8 19:00 archivo2
-rw-r--r-- 1 isaac 197609  26 Sep  8 19:00 archivo3
```

Si ahora quiero ver el espacio de trabajo tal y como estaba en el commit C1 tengo que usar `git checkout` junto con el hash o el tag del commit:

```
$ git checkout C1
A       archivo3
Note: switching to 'C1'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by switching back to a branch.

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

  git switch -c <new-branch-name>

Or undo this operation with:

  git switch -

Turn off this advice by setting config variable advice.detachedHead to false

HEAD is now at 883c39a C1
```

La salida del comando nos indica que estamos en **modo *detached HEAD*** (algo así como *HEAD desconectado*), esto significa que el puntero `HEAD` no está en ninguna rama, sino en un commit específico. Fíjate cómo ha cambiado la cabecera de la terminal:

```
~/Desktop/repo-pruebas (main)
~/Desktop/repo-pruebas ((C1))
```

En este estado podemos mirar, hacer cambios y crear commits, pero cualquier commit que realicemos no estará vinculado a ninguna rama existente. Esto es importante porque los cambios que no confirmemos en este estado se perderán si no los asociamos a una nueva rama, ya que cuando volvamos a una rama Git no recordará esos commits automáticamente.

Fíjate en el estado del repositorio:

```
$ git status
HEAD detached at C1
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   archivo3

$ ls -la
total 54
drwxr-xr-x 1 isaac 197609   0 Sep  8 19:05 ./
drwxr-xr-x 1 isaac 197609   0 Sep  8 17:42 ../
drwxr-xr-x 1 isaac 197609   0 Sep  8 19:09 .git/
-rw-r--r-- 1 isaac 197609  26 Sep  8 19:00 archivo1
-rw-r--r-- 1 isaac 197609  26 Sep  8 19:00 archivo3

$ git log --oneline
883c39a (HEAD, tag: C1) C1
```

La salida del comando `git status` nos está avisando (en rojo) que estamos en modo *detached HEAD* y que tenemos un cambio pendiente, aquel *archivo3* que creamos, pero no incluimos en ningún commit al inicializar el repositorio.

Por otro lado, la salida de `ls -la` nos indica que *archivo2* ya no está. Ese archivo lo creábamos en el commit C2, que ahora ya no existe, como vemos en el log.

Si lo que queremos en este punto es descartar todos los commits posteriores, lo que deberíamos hacer es volver al último commit de la rama y utilizar `git reset`, como vimos en este apartado.

Imagina que en este estado creamos un nuevo archivo y lo añadimos al índice.

```
$ echo "Hola Mundo desde archivo4" >> archivo4

$ git add -A
```

Ahora tenemos dos archivos preparados, pero seguimos en modo *detached HEAD*.

```
$ git status
HEAD detached at C1
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   archivo3
        new file:   archivo4

$ git log --oneline
883c39a (HEAD, tag: C1) C1
```

Vamos a hacer commit.

```
$ git commit -m "commit en modo detached HEAD"
[detached HEAD 4b069c6] commit en modo detached HEAD
 2 files changed, 2 insertions(+)
 create mode 100644 archivo3
 create mode 100644 archivo4
```

De nuevo, la cabecera de la terminal cambia para indicar donde estamos:

```
~/Desktop/repo-pruebas ((C1))
~/Desktop/repo-pruebas ((4b069c6...))
```

Y el log es:

```
$ git log --oneline
4b069c6 (HEAD) commit en modo detached HEAD
883c39a (tag: C1) C1
```

Si queremos conservar estos commits de manera permanente cuando volvamos a la rama *main*, debemos crear una nueva rama que apunte al estado actual del repositorio (que incluye los nuevos commits):

```
$ git checkout -b nueva-rama
```

Ahora podemos volver a cualquier rama que tengamos. Volvamos a *main*:

```
$ git checkout main
```

Veamos el log y el estado del directorio de trabajo:

```
$ git log --oneline
66e53ff (HEAD -> main, tag: C2) C2
883c39a (tag: C1) C1

$ ls -la
total 54
drwxr-xr-x 1 isaac 197609   0 Sep  9 11:05 ./
drwxr-xr-x 1 isaac 197609   0 Sep  9 10:50 ../
drwxr-xr-x 1 isaac 197609   0 Sep  9 11:05 .git/
-rw-r--r-- 1 isaac 197609  26 Sep  8 19:00 archivo1
-rw-r--r-- 1 isaac 197609  27 Sep  9 11:05 archivo2
```

Estamos exactamente en la situación inicial del repositorio, la que dejamos con el commit C2, con la salvedad de que *archivo3* ha desaparecido porque entonces lo teníamos añadido al índice, pero no estaba confirmado.

¿Dónde está el commit que hicimos en modo *detached HEAD*? Tenemos que cambiarnos a la rama que creamos en aquel momento, *nueva-rama*:

```
$ git checkout nueva-rama
Switched to branch 'nueva-rama'

$ git log --oneline
4b069c6 (HEAD -> nueva-rama) commit en modo detached HEAD
883c39a (tag: C1) C1

$ ls -la
total 55
drwxr-xr-x 1 isaac 197609   0 Sep  9 11:06 ./
drwxr-xr-x 1 isaac 197609   0 Sep  9 10:50 ../
drwxr-xr-x 1 isaac 197609   0 Sep  9 11:06 .git/
-rw-r--r-- 1 isaac 197609  26 Sep  8 19:00 archivo1
-rw-r--r-- 1 isaac 197609  27 Sep  9 11:06 archivo3
-rw-r--r-- 1 isaac 197609  27 Sep  9 11:06 archivo4
```

Ahí tenemos nuestro commit y el directorio de trabajo actualizado a la situación que dejamos cuando hicimos aquel commit. Ahora *archivo2* no existe porque ese archivo se creaba en el commit C2, que está en la rama *main*, pero no en *nueva-rama*.

6.1.1. En Visual Studio Code

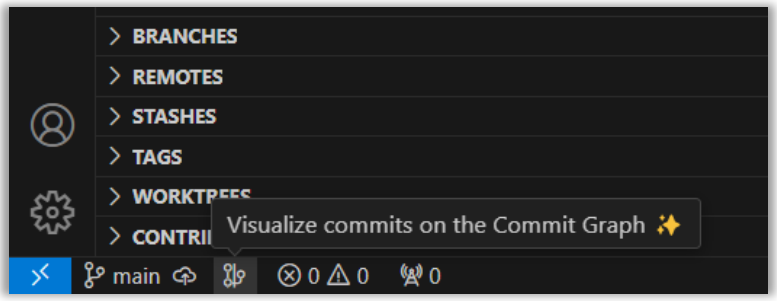
Para movernos entre commits utilizando VSC vamos a usar el mismo ejemplo de repositorio que antes:

```
git init
echo "Hola Mundo desde archivo1" >> archivo1
git add -A
git commit -m "C1"
git tag C1

echo "Hola Mundo desde archivo2" >> archivo2
git add -A
git commit -m "C2"
git tag C2

echo "Hola Mundo desde archivo3" >> archivo3
git add -A
```

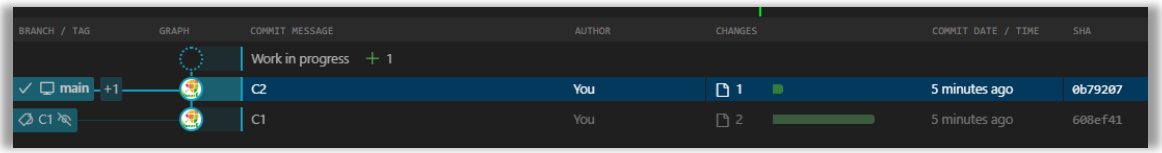
Tenemos:

```
$ git log --oneline
0b79207 (HEAD -> main, tag: C2) C2
608ef41 (tag: C1) C1
```

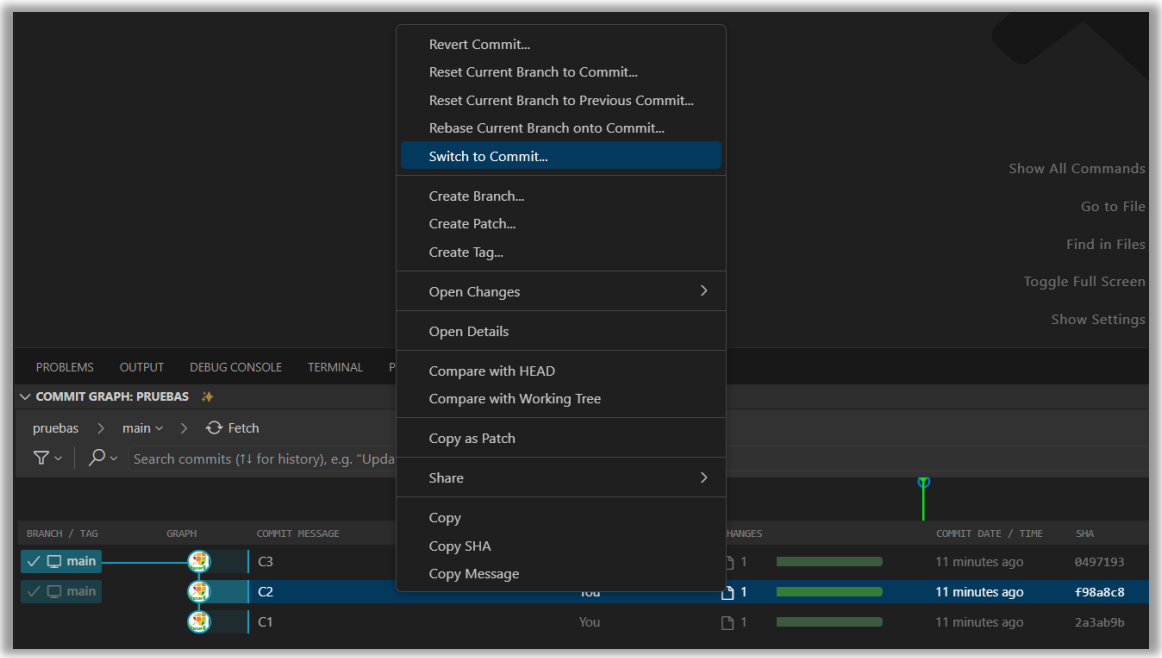
En VSC se puede ver la gráfica con Git Lens:



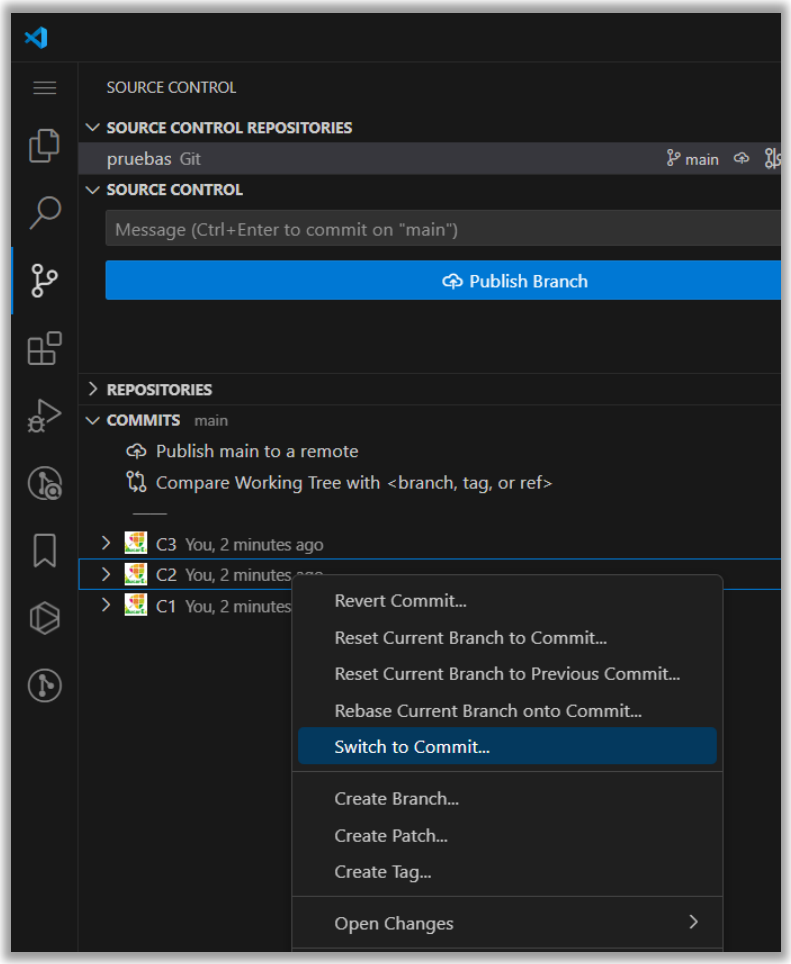
Lo que nos muestra:



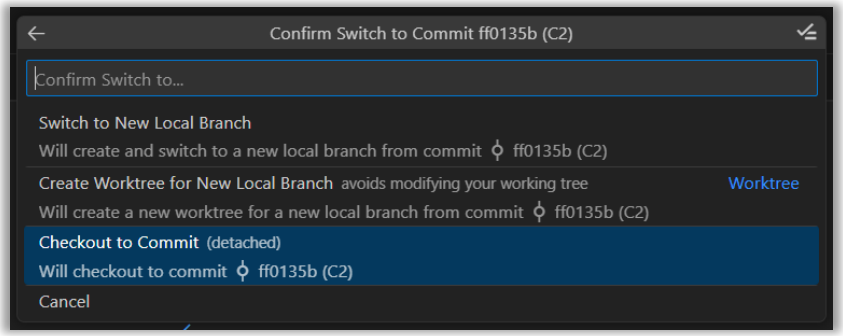
Si ahora nos queremos mover al commit C2 sin usar la línea de comandos podemos hacerlo desde esta misma ventana, haciendo clic derecho en el commit y seleccionando *Switch to Commit…* en el menú contextual:



O desde la pestaña Source Control, seleccionando la misma opción:



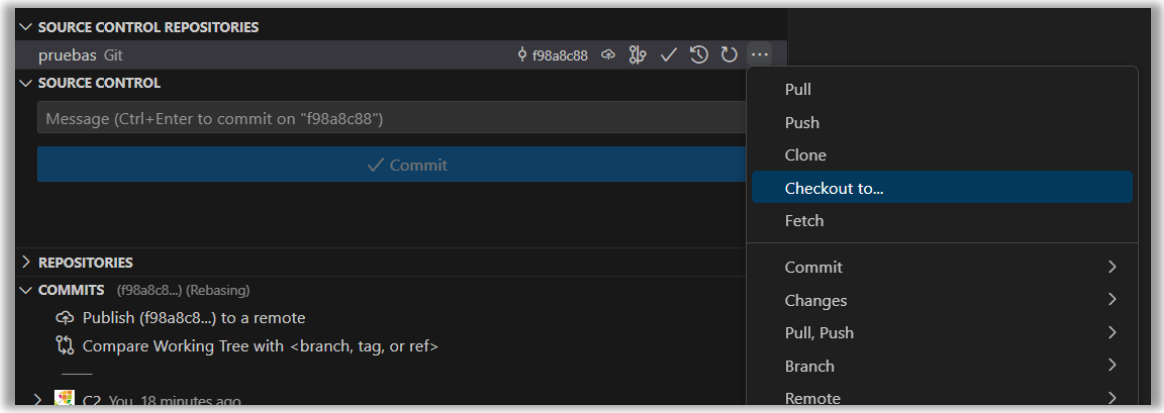
En ambos casos VSC nos ofrecerá varias opciones y seleccionaremos *Checkout to Commit (detached)*.



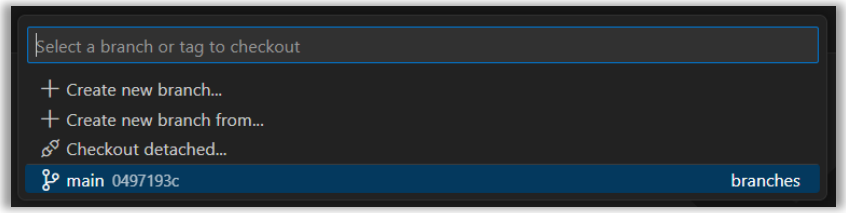
Ahora habremos movido el HEAD a ese commit y se verá reflejado en nuestro espacio de trabajo.

```
$ git log --oneline
608ef41 (HEAD, tag: C1) C1
```

Si ahora queremos volver al último commit, pulsando sobre los 3 puntos en el repositorio en la pestaña *Source Control* seleccionamos *Checkout to…*:



Y seleccionamos la rama a la que queremos movernos:



Ya estamos de nuevo con el HEAD apuntando al último commit:

```
$ git log --oneline
0b79207 (HEAD -> main, tag: C2) C2
608ef41 (tag: C1) C1
```

7. Tips

7.1. ¿Cómo probar los ejemplos de este documento?

Este documento está redactado de manera que se puedan probar todos los ejemplos de manera sencilla y rápida. En la mayoría de apartados se crea un repositorio nuevo para probar el comando o comandos que se van a explicar, muchas veces creando o modificando archivos. Esto implicaría tener que copiar varios comandos solo para empezar a ver el ejemplo en cuestión, lo que resulta en una tarea que acaba siendo tediosa. Existen varias maneras de hacer esto de manera más rápida, aquí simplemente explicaré la que me parece más sencilla.

Lo primero que haremos es crear una carpeta de pruebas, por ejemplo, *repo-pruebas*. Dentro de ella crearemos un fichero con extensión *sh*. Los archivos con esta extensión son scripts de shell, utilizados en sistemas operativos basados en Unix/Linux, que contienen una serie de comandos que se ejecutan en secuencia por el intérprete de comandos del sistema. Estos archivos se emplean para automatizar tareas y simplificar la ejecución de comandos repetitivos, que es exactamente lo que pretendemos. Si utilizas Windows necesitarás Git Bash o cualquier otra interfaz de línea de comandos que emule una terminal de Unix, como ya explicamos en el apartado dedicado a Git Bash.

Llamaremos a este archivo *ejecutar_pruebas.sh*. Lo abrimos y copiamos dentro:

```
#!/bin/bash

find . -mindepth 1 ! -name "$(basename "$0")" -delete

# Ejecutar comandos de Git
echo "Hola Mundo"
```

La primera línea, `#!/bin/bash`, indica al sistema que el script debe ser ejecutado usando el intérprete de comandos Bash. La siguiente línea, la que empieza con el comando `find`, busca y elimina todos los archivos y subdirectorios en el directorio actual, excepto el script que se está ejecutando. Finalmente, `echo "Hola Mundo"` simplemente muestra por pantalla “Hola Mundo”, está ahí sólo provisionalmente, para comprobar que todo funciona.

Ahora vamos a dar permisos de ejecución a este archivo, necesario en sistemas basados en Unix. Para ello abrimos una terminar, nos situamos en nuestra carpeta y escribimos:

```
chmod +x ejecutar_pruebas.sh
```

Finalmente, para ejecutar el comando, escribiremos:

```
./ ejecutar_pruebas.sh
```

Si todo ha ido bien, el script se ejecutará, vaciará la carpeta de archivos y subcarpetas, si las hubiera, y mostrará “Hola Mundo”. Pues ya está, nuestro script funciona.

¡Atención! Este script elimina archivos y carpetas, por lo que sólo deberías ejecutarlo en una carpeta de pruebas, en ningún otro sitio.

Ahora bien, ¿cómo usamos este script para simplificar el proceso de pruebas?

Supongamos que tenemos un ejemplo en el que inicializamos un repositorio, creamos dos archivos y hacemos un commit:

```
git init
touch archivo1
touch archivo2
git add -A
git commit -m "Commit inicial"
```

Solo tendremos que copiar estas líneas al final del script:

```
#!/bin/bash

find . -mindepth 1 ! -name "$(basename "$0")" -delete

# Ejecutar comandos de Git
git init
touch archivo1
touch archivo2
git add -A
git commit -m "Commit inicial"
```

Guardamos los cambios y ejecutamos el script, como ya hemos visto:

```
./ejecutar_pruebas.sh
```

El script habrá limpiado el directorio y habrá creado un repositorio nuevo. Este proceso lo podemos hacer tantas veces como queramos, dado que el script siempre limpia el directorio antes de ejecutar ningún comando.

7.2. Fusiones de ramas sin *fast-forward*

Por defecto, cuando se realiza un *merge* entre ramas, Git crea un nuevo commit que combina los cambios de ambas ramas y los aplica al historial del proyecto. Sin embargo, si la rama que se está fusionando está directamente por delante de la rama actual, es decir, **si la rama actual no tiene cambios que la rama a fusionar no tenga, entonces Git puede simplemente mover la rama actual hacia adelante hasta la posición de la rama a fusionar**. A esto se le llama "*fast-forward*".

Por ejemplo, creamos un commit en la rama *main*, creamos una rama nueva y hacemos otro commit en esa nueva rama:

```
git init
echo "file1" > file1
git add -A
git commit -m "C1"

git switch -c feat1
echo "file2" > file2
git add -A
git commit -m "C2"
```

Si ahora volvemos a la rama *main* y hacemos *merge* con *feat1*:

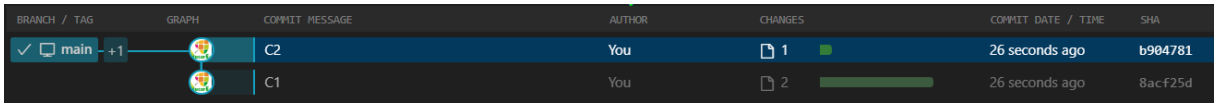
```
$ git switch main
$ git merge feat1
```

Git simplemente movería el commit C2, creado en la rama *feat1*, a la rama *main*. Esto es **fast-forward**. Dado la rama que se está fusionando, *feat1*, está por delante de la rama *main* (*main* no tiene cambios que *feat1* no tenga) Git puede simplemente mover la rama *main* hacia adelante hasta la posición de la rama a feat1.

En el log lo vemos así:

```
$ git log --oneline --decorate --graph --all
* 32c9f41 (HEAD -> main, feat1) C2
* 8474463 C1
```

Y con Git Graph:



No se aprecia la fusión de las ramas gráficamente porque la opción *fast-forward* está habilitada de forma predeterminada en `git merge`. Para forzar la creación de un nuevo commit de *merge* se utiliza la opción `--no-ff`:

Volvamos a la situación justo antes de hacer el *merge*:

```
git init
echo "file1" > file1
git add -A
git commit -m "C1"

git switch -c feat1
echo "file2" > file2
git add -A
git commit -m "C2"

git switch main
```

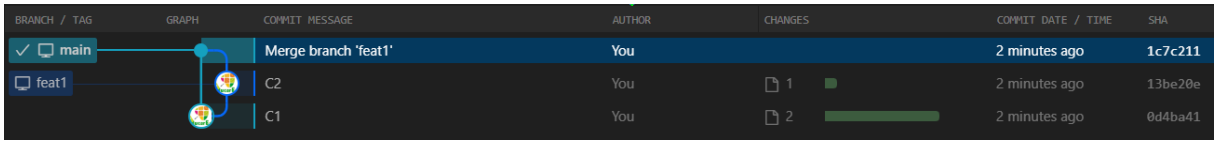
Ahora, usaremos la opción `--no-ff` al fusionar las ramas:

```
$ git merge --no-ff feat1
```

Ahora en el log vemos la fusión de forma gráfica:

```
$ git log --oneline --decorate --graph --all
*   1c7c211 (HEAD -> main) Merge branch 'feat1'
|\
| * 13be20e (feat1) C2
|/
* 0d4ba41 C1
```

Y en Git Graph:



La ventaja de utilizar la opción *fast-forward* es que el historial del proyecto se mantiene más limpio y fácil de seguir, ya que no se crea un nuevo commit de *merge* para cada fusión que se realiza. Sin embargo, si se desea mantener un registro completo y detallado de todas las fusiones, se puede desactivar la opción *fast-forward* y forzar la creación de un nuevo commit de *merge* para cada fusión que se realice.

7.3. Eliminar una rama y todos sus commits

Hemos creado una rama y al pasar un rato nos hemos dado cuenta de que no sirve para nada y queremos volver a la rama original y eliminar lo que hemos hecho.

En un repositorio nuevo de Git ejecutamos los siguientes comandos:

```
git init
echo "fichero1" > file1
git add -A
git commit -m "C1"
echo "fichero2" > file2
git add -A
git commit -m "C2"
echo "fichero3" > file3
git add -A
git commit -m "C3"

git switch -c experiment

echo "fichero4" > file4
git add -A
git commit -m "C4"
echo "fichero5" > file5
git add -A
git commit -m "C5"
```

En este momento me doy cuenta de que la rama *experiment* ya no me sirve y quiero volver al último commit de la rama *main*.

```
~/Desktop/repo-pruebas (experiment)
$ git switch main
Switched to branch 'main'
```

La situación es la siguiente:

```
~/Desktop/repo-pruebas (main)
$ git log --oneline --graph --all
* a493edc (experiment) C5
* 3de56d0 C4
* 779ee87 (HEAD -> main) C3
* e9446c7 C2
* a7eb799 C1
```

Ahora eliminamos la rama *experiment*:

```
~/Desktop/repo-pruebas (main)
$ git branch -D experiment
Deleted branch experiment (was a493edc).
```

Esto eliminará la rama *experiment* y todos los commits que hayas hecho en ella, ya que no están presentes en la rama *main* ni en cualquier otra rama que hayas mantenido.

```
~/Desktop/repo-pruebas (main)
$ git log --oneline --graph --all
* 779ee87 (HEAD -> main) C3
* e9446c7 C2
* a7eb799 C1
```

7.4. Buenas prácticas al escribir un mensaje de commit

Elegir un buen comentario para un commit es fundamental para mantener un historial de commits claro, comprensible y útil. Algunas buenas prácticas son:

1. Escribe un mensaje corto y descriptivo.
   - La línea principal del mensaje del commit debe ser breve (generalmente menos de 50 caracteres).
   - Debe resumir el cambio de manera clara y concisa.
2. Usa el tiempo presente imperativo.
   - Es una convención común escribir los mensajes de commit en tiempo presente imperativo. Por ejemplo, "Añade validación de entrada" en lugar de "Añadido validación de entrada".
3. Incluye más detalles si es necesario.
   - Si el cambio necesita más explicación, añade una línea en blanco después de la primera línea y luego proporciona una descripción más detallada.
   - Explica por qué se hicieron los cambios y cualquier detalle adicional que pueda ser relevante.

4. Haz el mensaje relevante.
   - El mensaje del commit debe estar directamente relacionado con los cambios realizados.
   - No incluyas información irrelevante o genérica.
5. Referencia a problemas y tareas.
   - Si tu commit resuelve un problema o tarea específica, referencia el identificador del problema (por ejemplo, "#123") en el mensaje del commit. Esto ayuda a rastrear cambios y entender el contexto.
6. Sé consistente.
   - Sigue un formato consistente para los mensajes de commit dentro de tu equipo o proyecto.
   - Si estás contribuyendo a un proyecto existente, revisa el historial de commits para ver cómo están formateados los mensajes.

Veamos algunos ejemplos de buenos mensajes de commit:

Mensaje corto:

```
Corrige error en la función de autenticación
```

Mensaje detallado:

```
Añade validación de entrada en el formulario de registro

- Se añadió una función para validar los campos de entrada del formulario de registro.
- Se corrigieron errores menores en el manejo de excepciones.
- Referencia al problema #45.

Este cambio mejora la seguridad al asegurarse de que todos los datos de entrada sean válidos antes de ser procesados.
```

Mensaje con referencia a un problema:

```
Corrige UserService al obtener usuarios (#67)
```

Siguiendo estas prácticas, tus mensajes de commit serán más útiles tanto para ti como para otros desarrolladores que trabajen en el proyecto.

8. Visual Studio Code

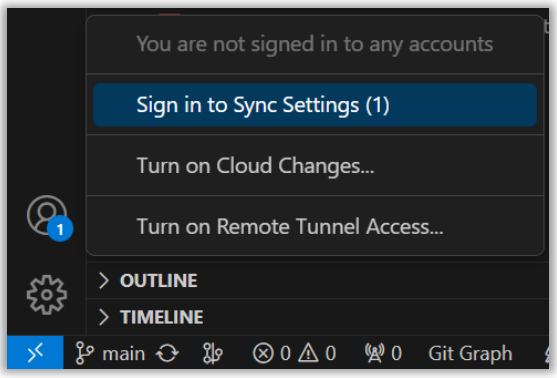
8.1. Configuración y extensiones de Visual Studio Code

No hay que hacer nada especialmente importante para utilizar VSC junto con Git. Sin embargo, sí hay algunas extensiones interesantes que podemos instalar y que mejorarán nuestra productividad usando Git.

1. GitLens. Proporciona una integración profunda de Git, mostrando información detallada en el editor sobre cambios, autores, etc.
2. Git Graph. Ofrece una visualización interactiva del historial de Git y las ramificaciones en un gráfico fácil de entender.
3. Git History. Permite explorar fácilmente el historial de cambios de un repositorio.
4. Git Project Manager. Facilita la gestión y navegación entre proyectos Git almacenados localmente.
5. GitHub Pull Requests. Facilita la revisión y gestión de *pull requests* directamente desde Visual Studio Code.
6. GitHub Repositories. Proporciona acceso rápido a los repositorios de GitHub, permitiendo la clonación, navegación y otras operaciones.
7. Git File History: Permite explorar el historial de cambios de un archivo específico en un repositorio Git, mostrando detalles como las confirmaciones relacionadas y los cambios realizados.

Además, vamos a enlazar la cuenta de GitHub con nuestro VSC local, de forma que podamos comunicarnos con nuestra cuenta de GitHub desde VSC.

Clic sobre el icono *Accounts* y seleccionamos la opción *Sign in to Sync Settings*:



Se nos abrirá una ventana del navegador donde deberemos autentificarnos con nuestra cuenta de GitHub. Una vez hecho esto satisfactoriamente, ya tenemos enlazado nuestro VSC con GitHub.

8.2. Abrir un proyecto directamente en VSC

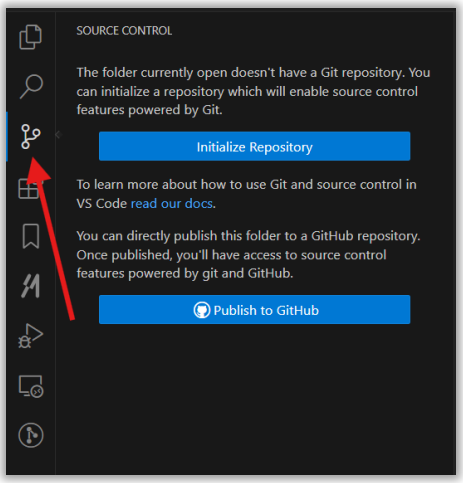
Una característica que mucha gente desconoce de VSC es la capacidad de abrir desde la terminal de comandos el directorio actual como un proyecto usando el comando `"code ."`.

Cuando instalas VSC se agrega el comando `code` al PATH del sistema, lo que permite ejecutar `code` desde una terminal de comandos. Si ejecutas `code` sin ningún argumento se abrirá Visual Studio Code con una ventana vacía, pero al añadir el punto se abrirá el directorio actual como un proyecto.

Tienes más información en esta sección de la documentación de VSC.

8.3. Crear un repositorio

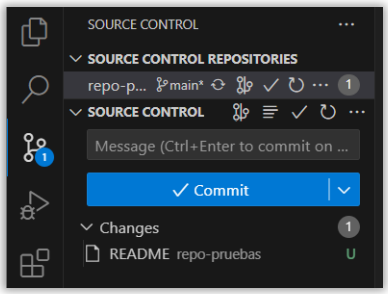
Crear un repositorio en VSC es muy fácil. Solo tienes que abrir un proyecto con VSC (como vimos en el apartado anterior, por ejemplo), ir a la pestaña de código fuente y hacer clic en el botón *Initialize Repository*.



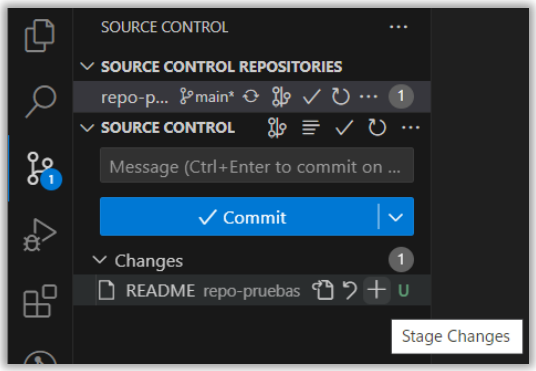
Si eliges la opción *Publish to GitHub*, además de crear el proyecto git en local lo subirás directamente a tu cuenta de GitHub, si es que lo tienes configurado.

8.4. Revisar el estado de los archivos: `git status`

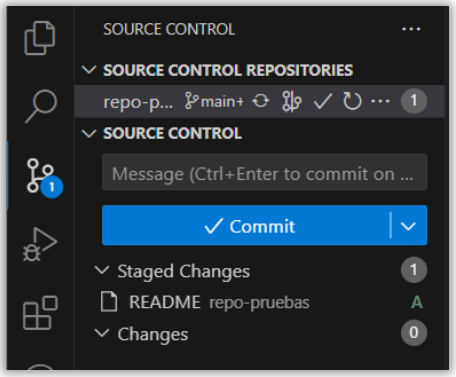
En VSC, podemos ver el estado de los archivos en la pestaña *Source Control* del panel lateral izquierdo:



La 'U' verde a la izquierda del nombre de archivo indica que está *untracked*. Para incluir el archivo debemos comenzar a rastrearlo. Con la terminal sería con el comando `git add`. Con VSC simplemente tenemos que pulsar el icono del '+' justo al lado de la 'U'.



Ahora, el archivo pasará al estado de *staged* (*preparado*). La 'A' indica que el archivo ha sido añadido (*added*):



La salida de `git status` indica lo mismo:

```
git status
On branch main
Your branch is up to date with 'origin/main'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   README
```

8.5. Clonar un repositorio

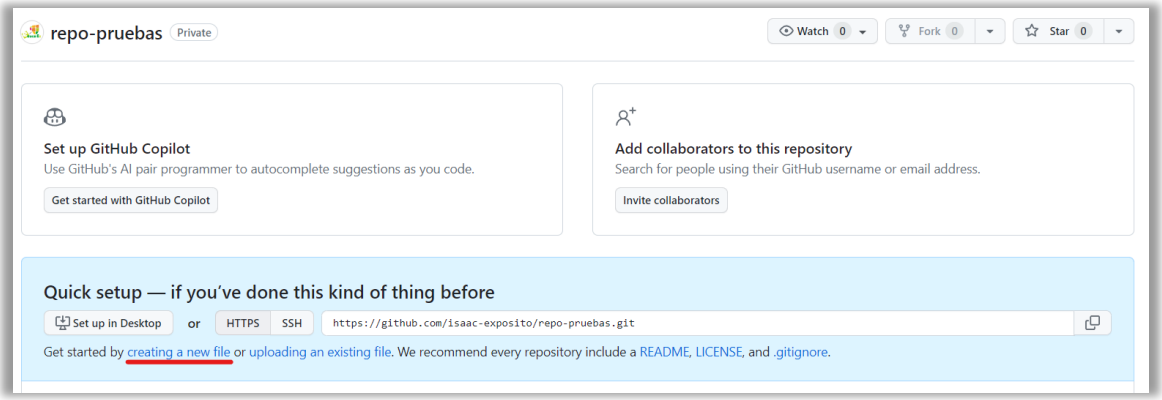
Llamamos *clonar un repositorio* a la acción de crear una copia de un repositorio remoto en tu propio ordenador. Esto te permite trabajar con los archivos del repositorio de manera local, realizar cambios, y luego sincronizar esos cambios con el repositorio remoto.

No es necesario que el repositorio que vayas a clonar sea tuyo, podrás clonar cualquier repositorio alojado en GitHub siempre que sea público. Si es privado necesitarás que el dueño del repositorio te ha dado privilegios de colaborador.

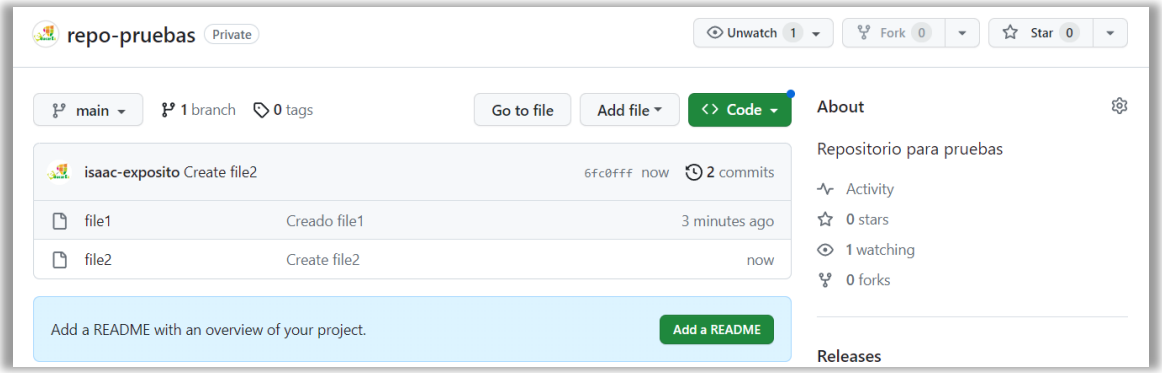
8.5.1. Usando línea de comandos

Para traer desde tu cuenta de GitHub un repositorio que no tienes en el ordenador en el que estás trabajando necesitarás usar el comando `git clone`.

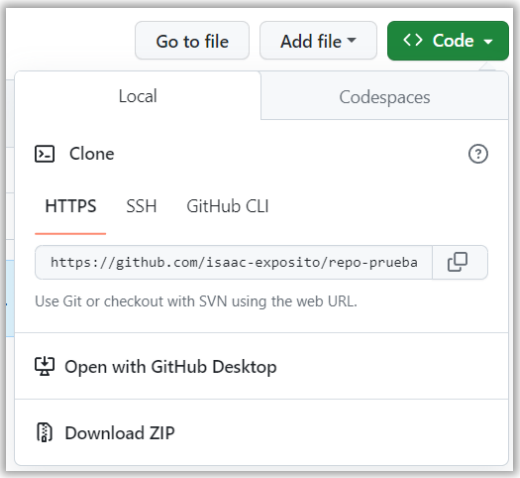
Hagamos una prueba. Vamos a crear un repositorio nuevo en nuestra cuenta de GitHub llamado *repo-pruebas*. Añadiremos un par de archivos también:



Una vez creados los dos archivos, nuestro repositorio de pruebas lucirá así:



Si pinchas sobre el botón verde "Code" verás que aparecen algunas opciones.



Lo primero que vemos son 3 pestañas que nos indican 3 direcciones para clonar el repositorio: “HTTPS”, “SSH” y “GitHub CLI”.

- **HTTPS:** Puedes usar `git clone [url]`, con la *url* que se indica aquí para clonar el repositorio en local. Se te pedirá una autentificación en el navegador. Una vez hecho, tendrás tu repositorio clonado.

```
git clone https://github.com/isaac-exposito/repo-pruebas.git
Cloning into 'repo-pruebas'...
info: please complete authentication in your browser...
remote: Enumerating objects: 6, done.
remote: Counting objects: 100% (6/6), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 6 (delta 0), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (6/6), done.
```

- **SSH:** SSH son las siglas de *Secure SHell*, un protocolo de red destinado a la conexión con máquinas a las que accedemos por línea de comandos. Con las claves SSH puedes conectarte a GitHub sin necesidad de proporcionar el nombre de usuario y el *personal access token* cada vez. Más información sobre esta manera de autentificación en la página de GitHub y un tutorial de cómo configurarlo aquí.

- **GitHub CLI:** Es una herramienta propia de GitHub que incluye diversas características de GitHub incorporadas. No lo veremos en este documento. Si quieres más información, puedes consultar la documentación de GitHub.

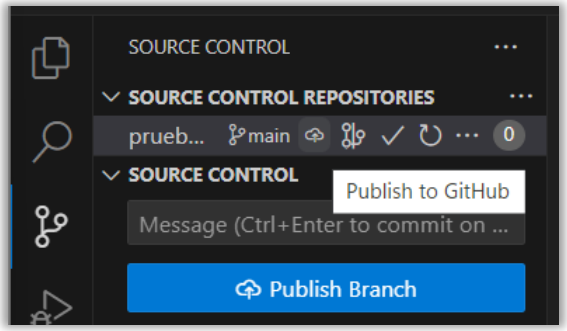
Las otras dos opciones que tenemos son clonar y abrir el repositorio con el software de escritorio GitHub Desktop o descargar el repositorio en formato zip.

8.5.2. Usando Visual Studio Code

Una vez hemos enlazado nuestra cuenta de GitHub con VSC, clonar un repositorio es fácil. Puedes seguir los pasos descritos en la documentación de VSC.

8.6. Publicar un repositorio local desde VSC a GitHub

VSC tiene la opción de publicar en GitHub un repositorio creado localmente sin necesidad de tener creado el repositorio en GitHub previamente, simplemente tenemos que pinchar en *Publish Branch* o en el icono de la nube:



8.7. Ver los cambios realizados en archivos

Si quieres ver exactamente qué ha cambiado en un archivo, el comando que necesitas es `git diff`. Puedes aprender a usar `git diff` mirando la documentación de Git, aquí nos centraremos en ver cómo hacerlo en VSC.

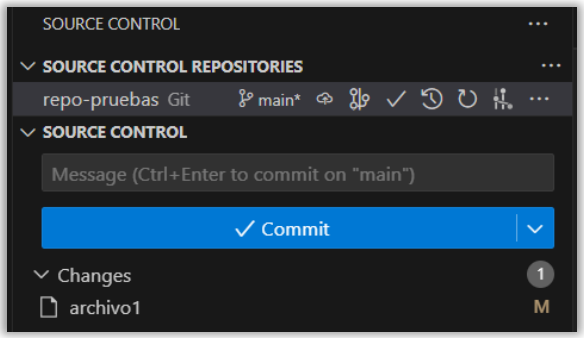
Veamos un ejemplo. Ejecuta las siguientes instrucciones para crear un repositorio Git con dos ficheros:

```
git init
echo "Soy el archivo1" > archivo1
echo "Soy el archivo2" > archivo2
git add -A
git commit -m "Primer commit"
```

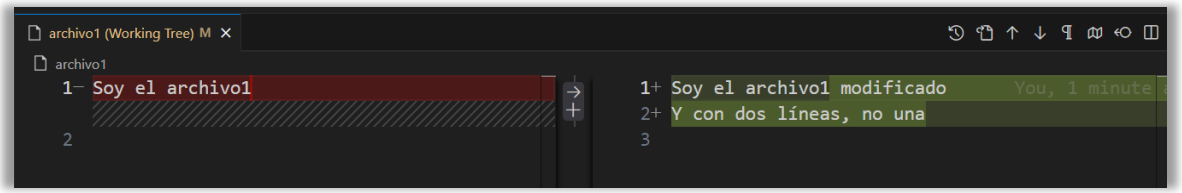
Modifica *archivo1* de manera que quede:

```
Soy el archivo1 modificado
Y con dos líneas, no una
```

VSC detectará que tienes un archivo modificado:



Si haces doble clic sobre *archivo1* verás lo siguiente:



Esta es la manera que tiene VSC de mostrarte los cambios que ha habido en un fichero.

9. Webgrafía

- Documentación de Git: https://git-scm.com/doc
- Pro Git book (español)
- GitHub Cheat Sheet (español)
- Using Git source control in VS Code
- Using GitHub Codespaces in Visual Studio Code
- [Youtube] Using Git with Visual Studio Code (Official Beginner Tutorial)
- https://learngitbranching.js.org/?locale=es_ES
- https://davidinformatico.com/apuntes-git