Flujos de trabajo Git

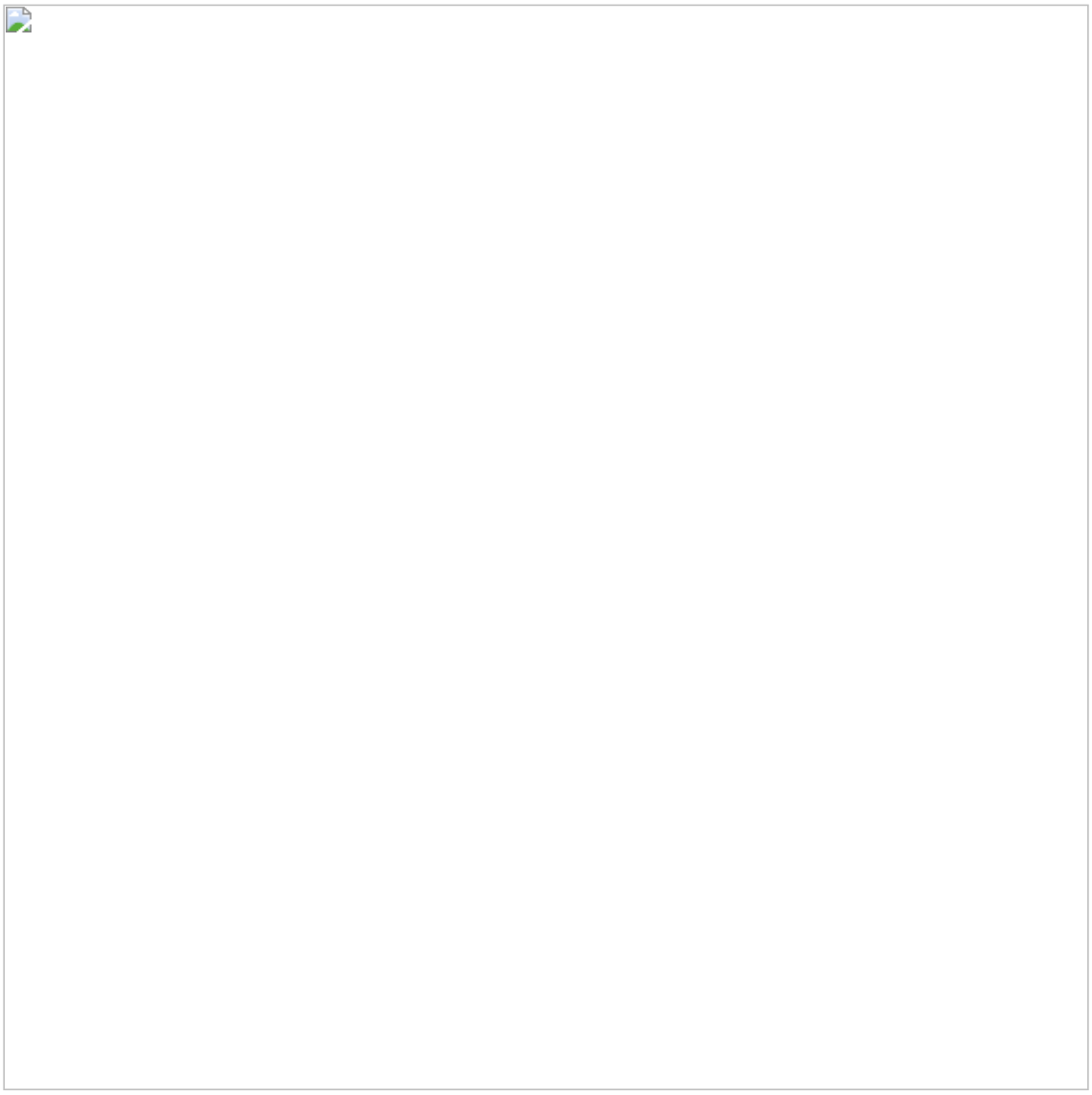
En este tema vamos a hacer un breve repaso de los conceptos fundamentales de Git y presentar los flujos de trabajo más habituales que se utilizan para gestionar el trabajo de equipos de desarrollo usando esta herramienta de control de versiones.

> En todas las imágenes y en el texto aparece `master` como la rama principal, aunque la tendencia en la actualidad es usar `main`.
>
> Términos como *master-slave* tienen connotaciones racistas en sociedades con un pasado esclavista como la de Estados Unidos y están siendo sustituidos por otros. En Git se usa el término *master* de forma individual, sin el término *slave*, y esto lo aleja de estas connotaciones racistas. Sin embargo, GitHub ha decidido modificar el nombre por defecto de la rama principal a *main* (puedes leer aquí el anuncio de la noticia).
>
> Es muy sencillo renombrar la rama principal a `main`. Lo único que tienes que hacer en local es:
>
> ```
> $ git branch -M master main
> $ git push -u origin main
> ```
>
> Y en GitHub hay que cambiar la rama por defecto:
>
> - Ir a repository > Settings > Branches
> - Seleccionar `main` como rama principal
> - Ir a Code > Branches y borrar la rama `master`

Git

Lo primero que nos dicen siempre de Git es que es un sistema de control de versiones (VCS en inglés, *Version Control System*) distribuido. ¿Qué significa esto?

Todos conocemos la utilidad de los sistemas de control de versiones. Es un sistema que registra los cambios a lo largo del tiempo en un fichero o un conjunto de ficheros, de forma que es posible recuperar más tarde versiones específicas.



Cuando trabajamos en equipo en un proyecto software, el sistema de control de versiones nos va a permitir cosas como:

- Revertir un conjunto de ficheros a un estado previo.
- Comparar cambios a lo largo del tiempo.
- Consultar quién ha sido el último que ha modificado algo en algún fichero que está causando problemas.
- Abrir una rama independiente del tronco principal en la que realizar un desarrollo que se integra después cuando esté terminado y probado.
- Recuperar un estado anterior estable si hemos roto algo en los ficheros que estás tocando.

Un VCS es el elemento fundamental de prácticas de desarrollo más avanzadas. Cualquier sistema o metodología de desarrollo en equipo tiene como prerrequisito la utilización de un sistema de control de versiones.

Ejemplos de utilización del VCS en el desarrollo de software:

- Distribución de tareas entre miembros del equipo e integración posterior del código.
- Revisión de código.
- Sistema de integración continua, como el que se muestra en la imagen.
- Mantenimiento de distintas versiones del producto desarrollado.
- Compartir código con desarrolladores externos.

Historia de Git

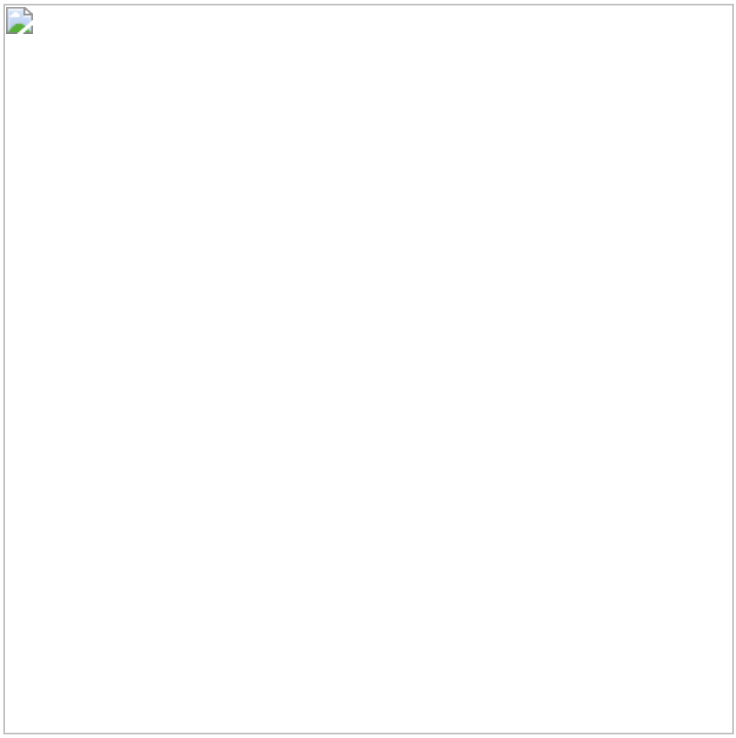
Git nace para gestionar el desarrollo del kernel de Linux en 2005. La comunidad de Linux, y en especial su creador Linus Torvalds, lo desarrolla en esa fecha para sustituir un software de control de versiones propietario que no cubría las necesidades del equipo.

Entre los objetivos que se persiguen:

- Velocidad
- Diseño simple
- Soporte para el desarrollo de miles de ramas simultáneas
- Completamente distribuido
- Capacidad de manejar proyectos grandes (en tamaño y tiempo) como el kernel de Linux

Desde su nacimiento en 2005, Git ha evolucionado y madurado, convirtiéndose en un sistema increíblemente rápido, muy eficiente con proyectos grandes y con una enorme facilidad de gestionar cientos de ramas simultáneamente.

Sistemas de control de versiones distribuidos



Los sistemas de control de versiones modernos como Git y Mercurial son distribuidos. Cada desarrollador tiene su propio repositorio y su copia de trabajo en su máquina.

Después de hacer un commit de tus cambios, los demás no tienen acceso a ellos hasta que los subes (push) al repositorio remoto central. Para obtener los cambios del repositorio central hay que bajarlos (fetch) al repositorio local y actualizar la copia de trabajo.

El ciclo de trabajo básico es:

- Haces uno o varios commits de tus cambios en tu repositorio local.
- Cuando quieres que los compañeros vean todos los cambios haces un push al repositorio central.
- Los compañeros hacen un fetch para actualizar su repositorio local.
- Después actualizan su copia de trabajo.

Hacer notar que la confirmación (commit) solo mueve los cambios entre el directorio de trabajo y el repositorio local. A diferencia de otros sistemas de control de versiones centralizados, es necesario otro comando (git push) para mover el cambio al repositorio central.

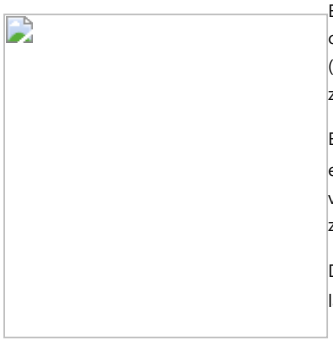
Repositorios y copias de trabajo

Un sistema de control de versiones usa un repositorio (una recopilación de todos los cambios) donde se almacena la historia de cambios cambios que se han realizado. Este repositorio se guarda en el directorio `.git` del proyecto.

Mientras que trabajas en el directorio actual no se modifica nada en el repositorio Git. Cuando has realizado un conjunto de cambios con una cierta entidad, confirmas (commit en inglés) esos cambios y se guardan en el repositorio con un nombre, añadiéndose un nuevo commit a la historia del proyecto.

Zona de stage

Podemos entender mejor el funcionamiento de los comandos git add y git commit introduciendo el concepto de zona de stage.



En Git no es obligatorio introducir en el siguiente commit todos los cambios que hayamos hecho, podemos seleccionar una parte de ellos (o todos, si así nos interesa). Para ello se define en Git la denominada zona de *stage*.

El comando `git add` sirve para añadir nuevos ficheros que antes no estaban en seguimiento y para marcar los cambios que queremos que vayan en el siguiente commit. Todos esos cambios se guardan en la zona de stage.

Después, el comando `git commit` añade estos cambios que están en la zona de stage al repositorio.

Borrado y renombrado

Existen dos formas posibles de borrar y renombrar un fichero: usando los comandos propios del sistema operativo o usando los comandos equivalentes de Git.

- En el primer caso, podemos usar los comandos `rm` para borrar un fichero o `mv` para cambiar el nombre de un fichero. Una vez realizado el cambio en el directorio de trabajo, si hacemos un `git status` veremos que los cambios no están en stage (están en rojo).

  Debemos entonces hacer un `git add .` para confirmar el borrado o el cambio de nombre. Si hacemos un `git status` veremos que ya están en verde, con lo que los cambios se confirmarán en el siguiente commit.

  Por ejemplo:

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

  Changes not staged for commit:
    (use "git add/rm <file>..." to update what will be committed)
    (use "git checkout -- <file>..." to discard changes in working directory)

      deleted:    imagen.png (EN ROJO)

  no changes added to commit (use "git add" and/or "git commit -a")
  $ git add .
  $ git status
  On branch main
  Your branch is up to date with 'origin/main'.

  Changes to be committed:
      (use "git reset HEAD <file>..." to unstage)

      deleted:    imagen.png (EN VERDE)
  $ git commit -m "Borrado fichero imagen"
  ```

  Puede resultar curioso que para borrar un fichero tengamos que hacer un `git add`, pero hay que entender que lo que estamos haciendo es añadir en el stage el cambio (un borrado).

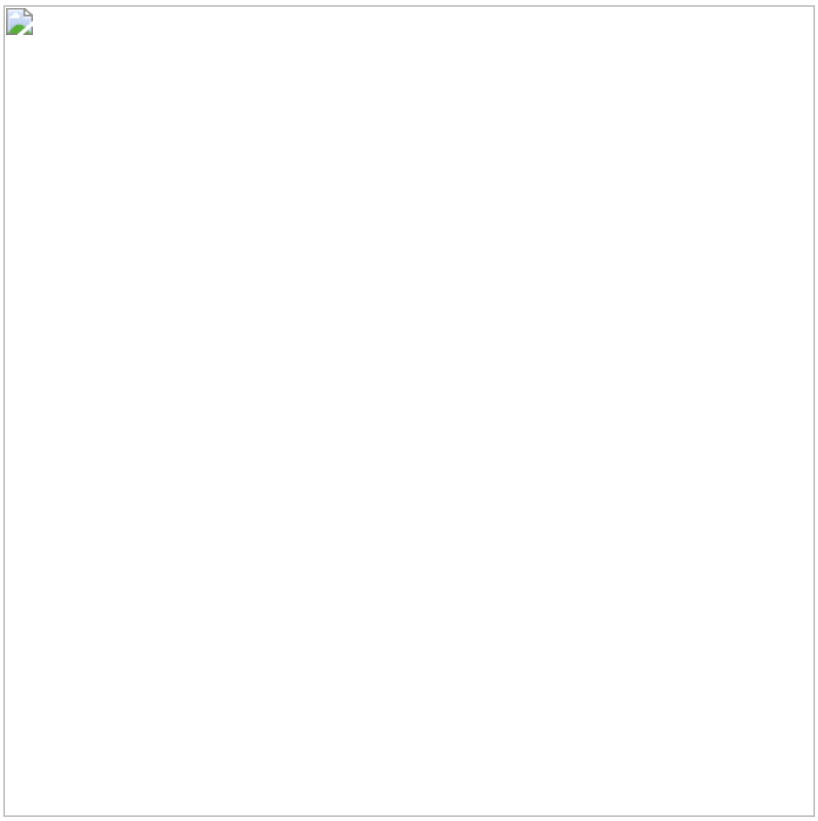
- La otra forma es usando los comandos equivalentes de Git: `git rm` y `git mv`. Estos comandos realizan el cambio en el directorio de trabajo y lo añaden al stage. No hace falta usar `git add`:

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

  Changes to be committed:
      (use "git reset HEAD <file>..." to unstage)

      deleted:    repository.png (EN VERDE)
  ```

Grafo de commits e índices



La figura muestra un grafo de commits de un repositorio. Los commits tienen identificadores únicos.

En verde aparecen los índices que se utilizan en Git para definir ramas (`master` y `origin/master`), etiquetas (`v1.0`) y el commit actual en el que nos encontramos (`HEAD`). Los índices de las ramas y tags apuntan a commits concretos. El índice `HEAD` apunta a una rama, para indicar que estamos en esa rama, o a un commit concreto cuando hacemos un `checkout` (ver el siguiente apartado).

En la figura, las ramas `master` y `origin/master` (rama remota copiada en local) se encuentran en el mismo commit, el último commit añadido. Estamos en la rama `master` (`HEAD` apunta a esa rama). Y el segundo commit está etiquetado con `v1.0` .

Para etiquetar un commit, nos movemos a él y hacemos un `git tag` :

```
$ git tag v1.0
```

Trabajar con la historia en Git

A la hora de trabajar con Git es importante tener en cuenta que una rama es realmente un índice que apunta al último commit añadido. Además el índice `HEAD` indica nuestra posición en el repositorio.

Si hacemos un `checkout` a un commit pasado el índice `HEAD` se moverá a ese commit y todos los ficheros de nuestro directorio de trabajo cambiarán automáticamente y se mostrarán tal y como estaban en ese commit. Podremos examinarlos, compilarlos y, por ejemplo, buscar el origen de algún bug justo en el momento en que se originó.

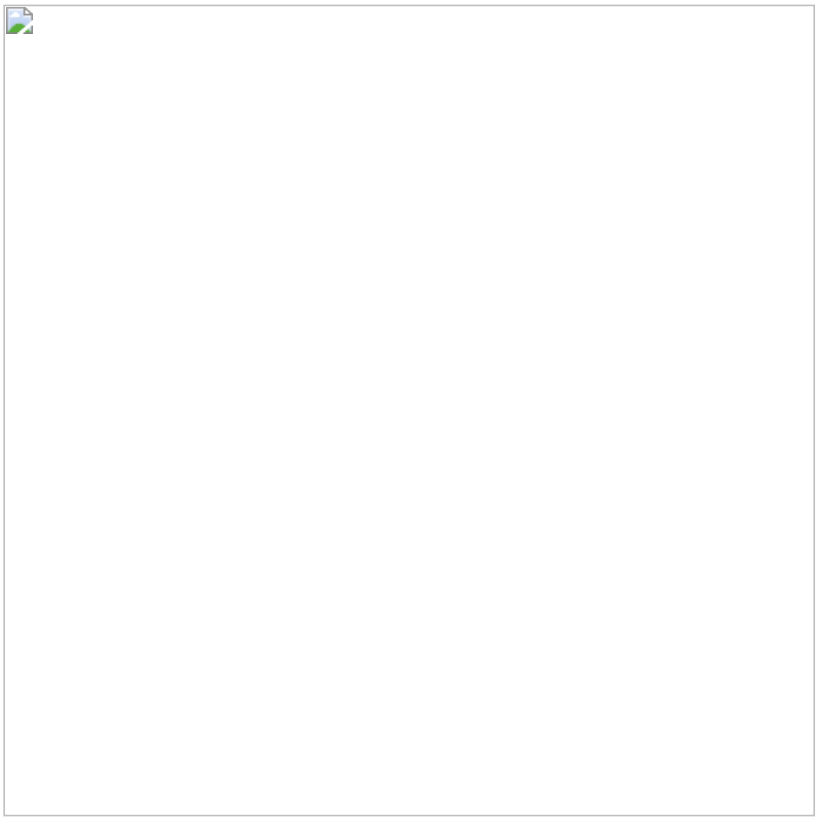
En la imagen anterior, podemos hacer un checkout al commit `0bf82cd` :

```
$ git checkout 0bf82cd
```

El comando anterior sería equivalente a hacer un checkout al tag `v1.0` ;

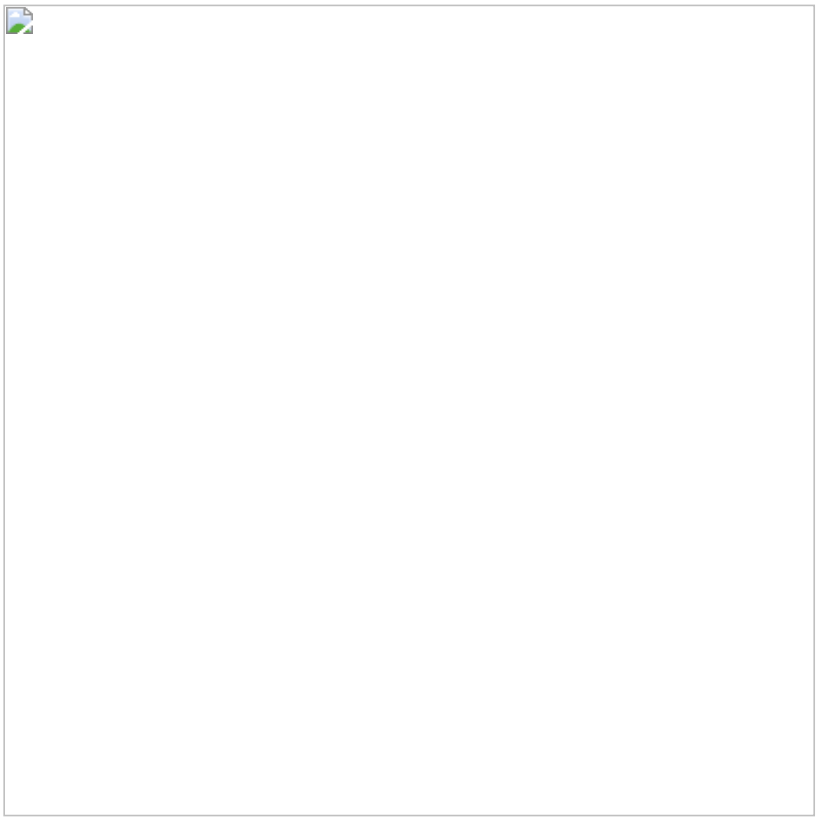
```
$ git checkout v1.0
```

El resultado será que moveremos `HEAD` a ese commit y que todos los ficheros en nuestro directorio de trabajo se cambiarán automáticamente a como estaban en aquel momento.



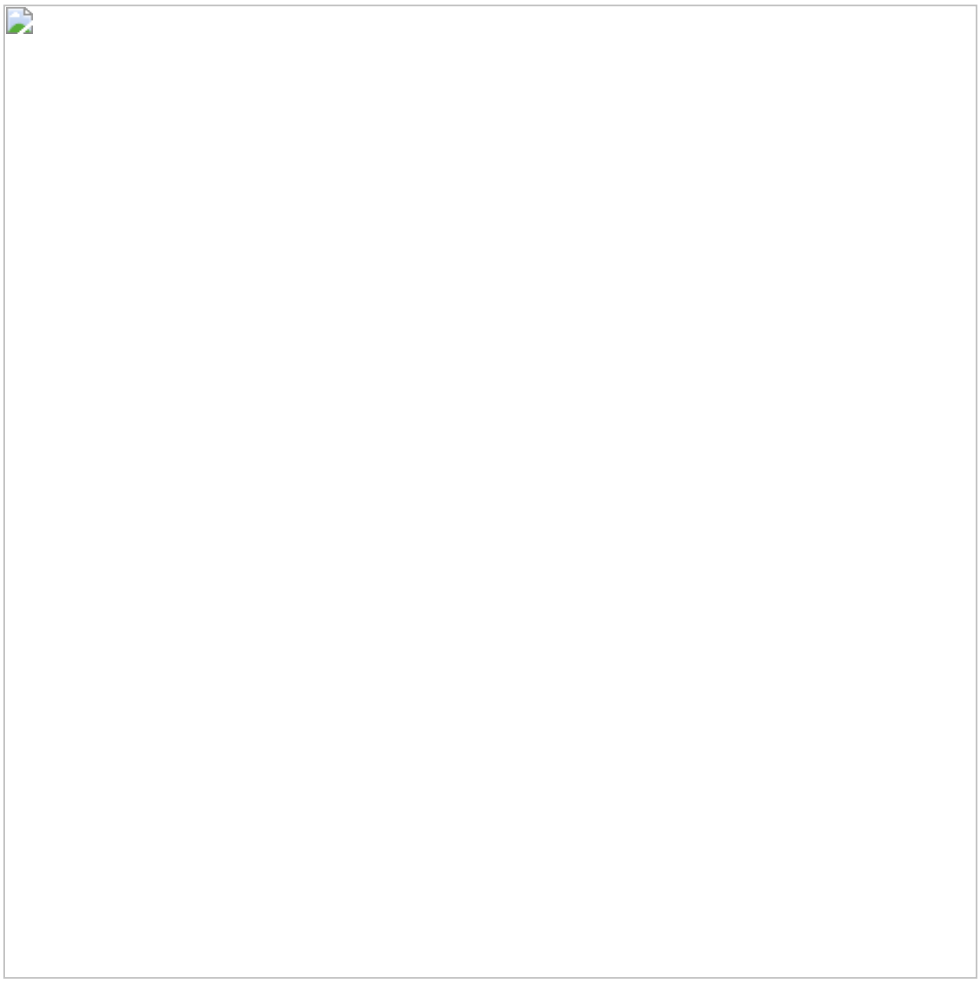
Podemos hacer ahora arreglar el bug con un nuevo commit:

```
# Hacemos cambios
$ git add .
$ git commit -m "Cambios sobre un commit pasado"
```



Y, por último, podríamos crear una rama, movernos a `master` y mergear el fix:

```
$ git branch fix
$ git checkout master
$ git merge fix
```



Hablaremos más adelante más de ramas y merges y quedará más claro todo esto.

Cambiar el último commit en local

Si todavía no hemos subido el commit al repositorio remoto tenemos varias opciones para cambiar el último commit.

Si queremos cambiar sólo el mensaje del commit:

```
$ git commit --amend -m "<nuevo mensaje>"
```

Si queremos deshacer el commit, pero no los cambios introducidos en él:

```
$ git reset HEAD^
```

El espacio de trabajo no cambia, pero el commit se ha desecho.

`HEAD^` significa "el commit anterior a HEAD". Es equivalente a poner el número de commit anterior al actual:

```
$ git reset <commit-anterior>
```

Si queremos eliminar los cambios del último commit del espacio de trabajo y empezar de nuevo:

```
$ git reset --hard HEAD^
```

Deshacer el último commit cuando se ha publicado

Si ya hemos publicado el commit en el repositorio remoto otras personas pueden habérselo descargado, con lo que el commit ya no es solo nuestro, sino que es parte de la historia pública del proyecto.

Es posible revertir los cambios del commit que queremos eliminar con el comando git revert .

El comando introduce un commit con exactamente los cambios contrarios al commit indicado.

Por ejemplo, el siguiente comando crea un commit que revierte el último commit

```
$ git revert HEAD
```

Para revertir los cambios realizados hace 3 commits:

```
$ git revert HEAD~3
```

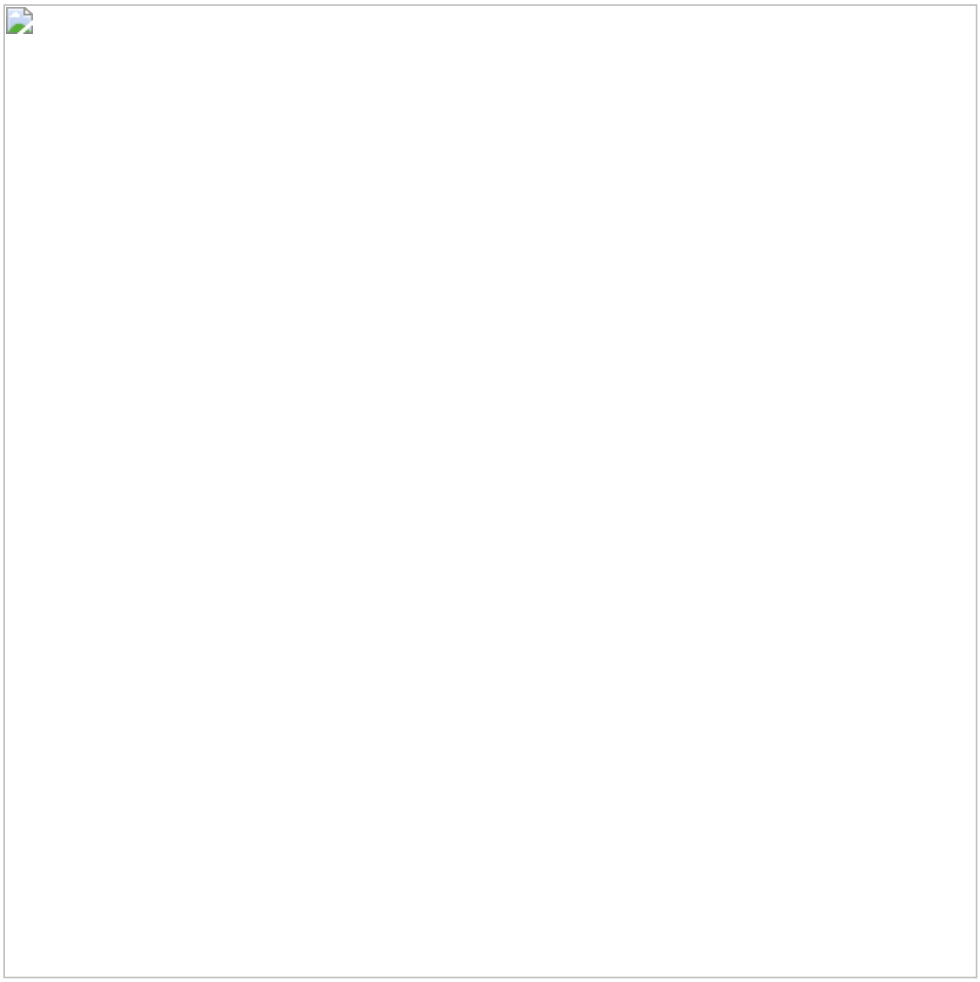
Para revertir sin commitear los cambios realizados por el quinto último commit en master (incluido) hasta el tercer último commit en master (incluido), usando la opción -n:

```
$ git revert -n master~5..master~2
```

Deshacer un conjunto de commits (git reset)

El comando `git reset` permite mover el índice de la rama a un commit anterior, eliminando los commits que intermedios.

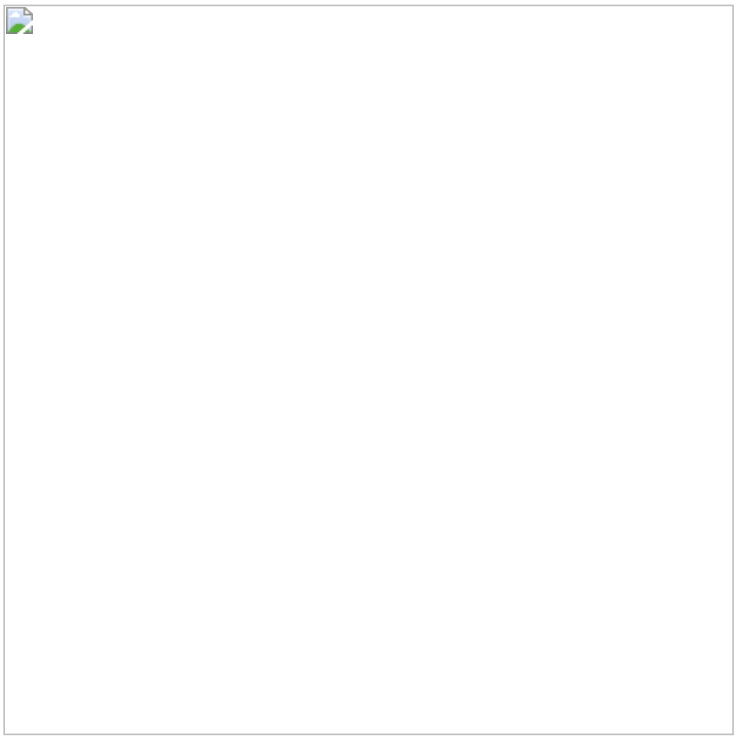
Por ejemplo, supongamos la siguiente historia de commits, en la que se muestran los cambios introducidos en los últimos dos commits:



Si hacemos:

```
$ git reset C2
```

moveremos el índice de la rama actual a `C2` y todos los cambios de los commits intermedios (`C3` y `C4`) se mantienen en el directorio de trabajo, pero sin estar anotados en ningún commit:

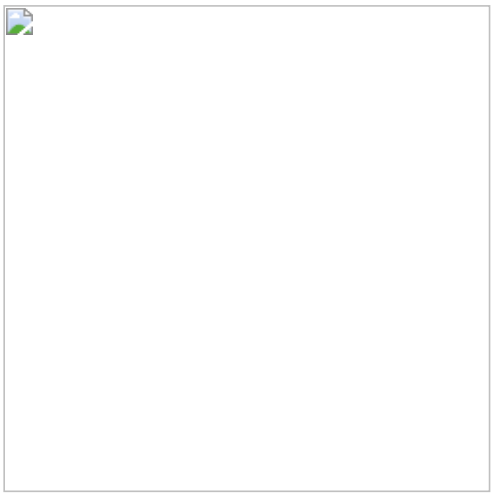


El comando anterior es equivalente a hacer un `git reset --soft`. Es un comando seguro que no borra nada del directorio de trabajo.

Sin embargo, si hacemos

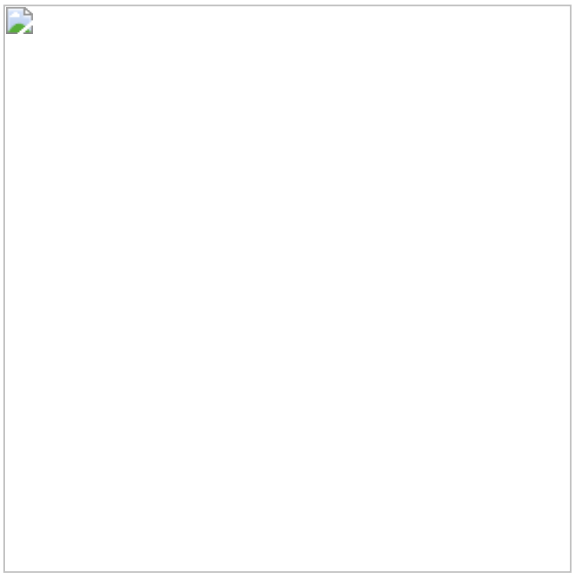
```
$ git reset --hard C2
```

eliminamos todos los cambios y toda la historia. Es equivalente a un `checkout` moviendo el índice de la rama.



Trabajo con ramas en Git

Supongamos que comenzamos con la siguiente historia (por simplificar, en las imágenes no mostramos la etiqueta `head`).



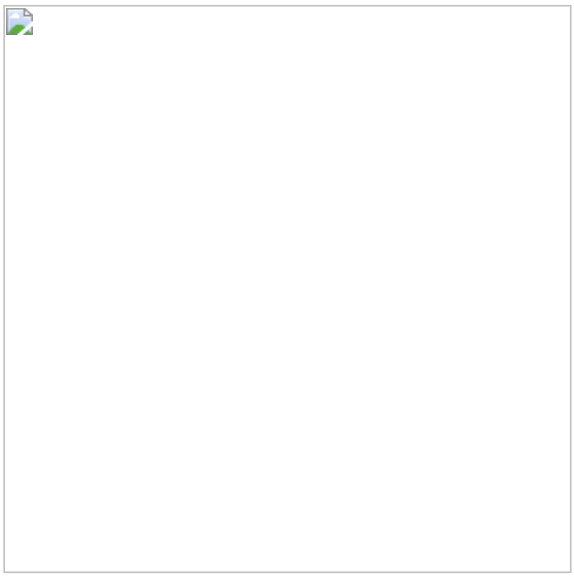
Vamos a crear una rama. Lo podemos hacer con los comandos:

```
$ git branch iss53
$ git checkout iss53
```

Los comandos anteriores son equivalentes al más usual:

```
$ git checkout -b iss53
```

El resultado es el siguiente:



Ahora escribimos algo de código en la rama y hacemos un commit.

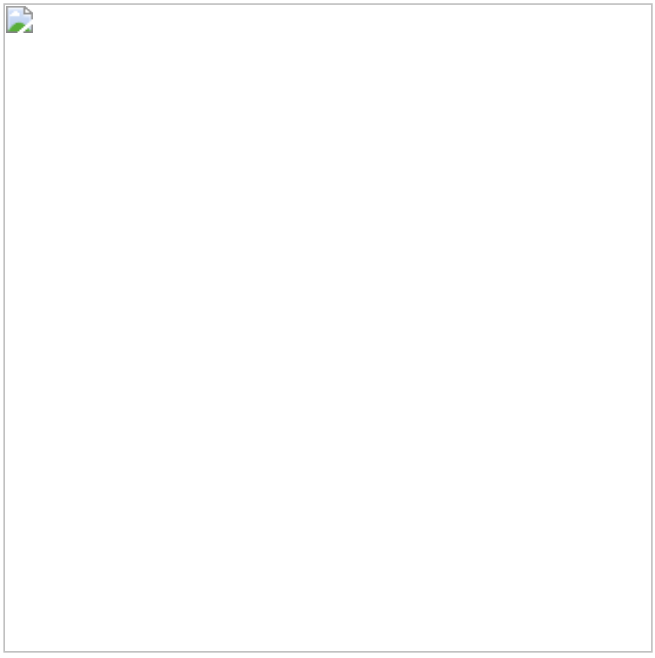
```
# escribimos código
$ git commit -am "Añadido nuevo font"
```

El resultado (`head` estaría apuntando a `iss53`):

Ahora nos vamos otra vez a `master` y creamos allí otra rama en la que hacemos un hotfix:

```
$ git checkout master
$ git checkout -b hotfix
# corregimos el error
$ git commit -am "Corregido el enlace erróneo"
```

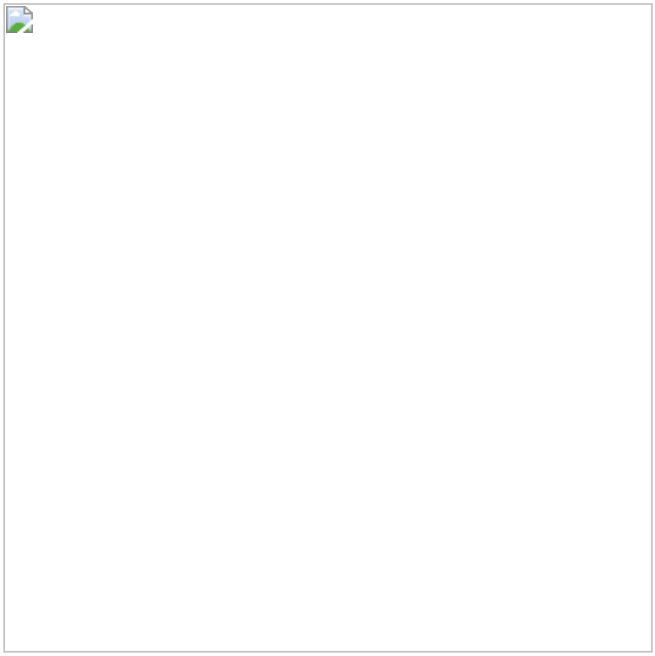
El resultado (`head` estaría apuntando a `hotfix`):



Ahora integramos el hotfix en `master` :

```
$ git checkout master
$ git merge hotfix
```

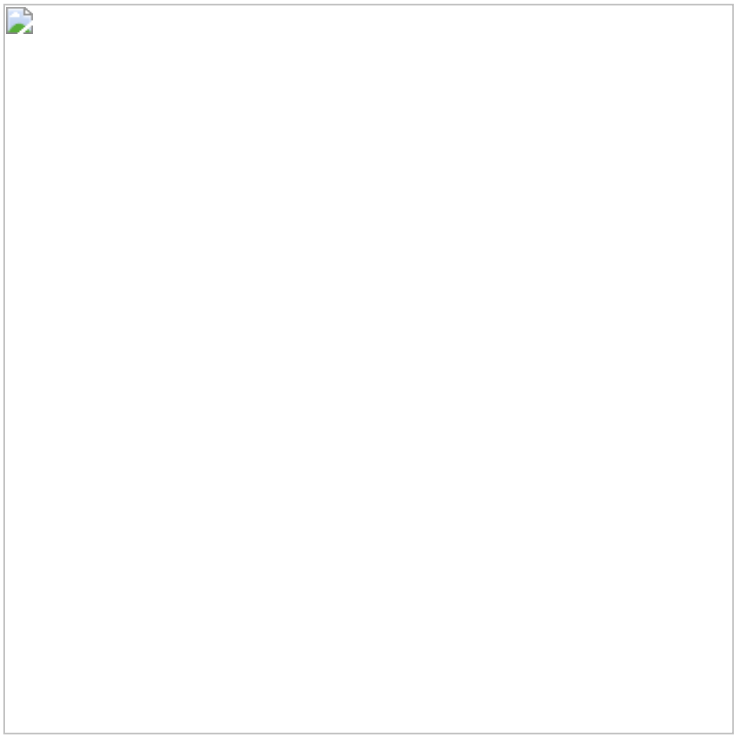
Al estar `master` y `hotfix` en la misma línea de commits, git no tiene que crear ningún commit de merge, sino que hace un *fast forward* y adelanta la rama de master para que apunte a `C4` (`head` estaría apuntando a `master`):



Ahora borramos la rama `hotfix` y hacemos un nuevo commit en `iss53` :

```
$ git branch -d hotfix
$ git checkout iss53
# Cambiamos más cosas
$ git commit -am "Finalizado el nuevo aspecto"
```

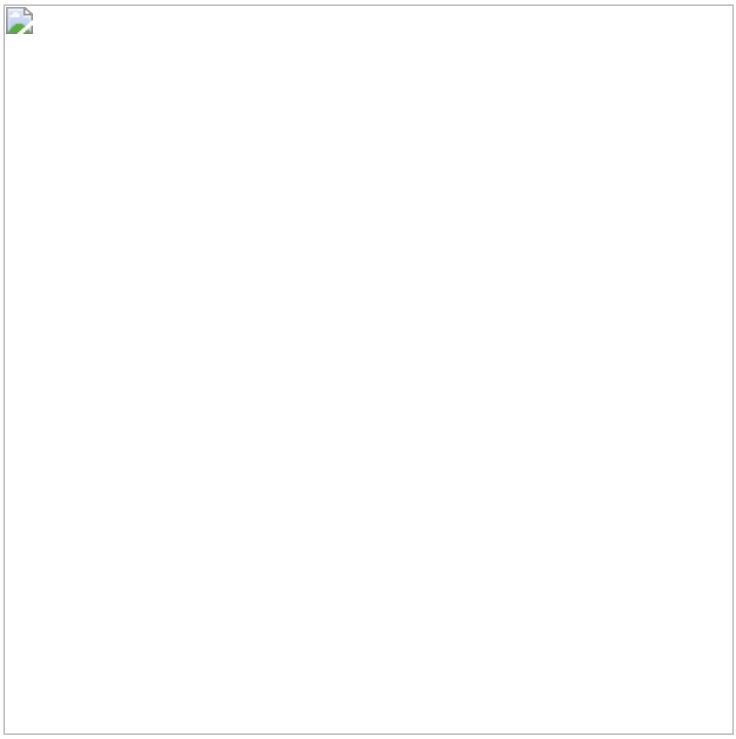
El resultado es el siguiente (`head` estaría apuntando a `iss53`):



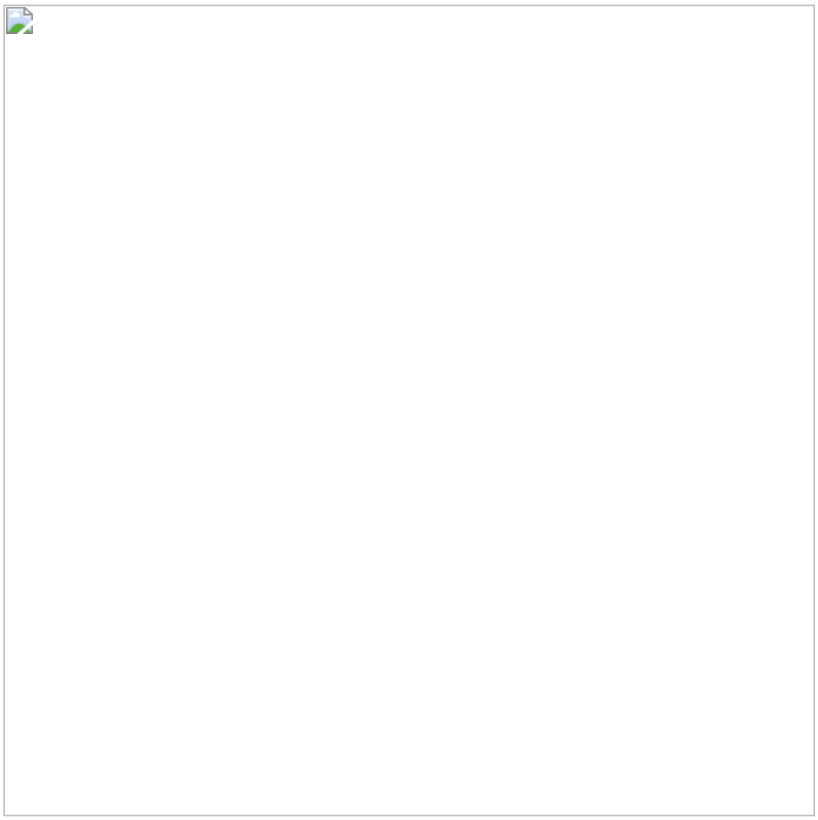
Y ahora mezclamos la rama con `master`:

```
$ git checkout master
$ git merge iss54
```

Para hacer el merge, git utiliza una estrategia denominada *recursiva*. Consiste en buscar el commit antecesor común a ambas ramas (`C2`), calcular los cambios desde ese commit hasta el commit que se quiere mezclar (cambios desde `C2` hasta `C5`) y hacer un *patch* de esos cambios en el commit en el que se quiere hacer la mezcla (`C4`):



Si no hay ningún conflicto, el resultado de la mezcla se consolida automáticamente en un nuevo commit y se avanza la rama `master`:

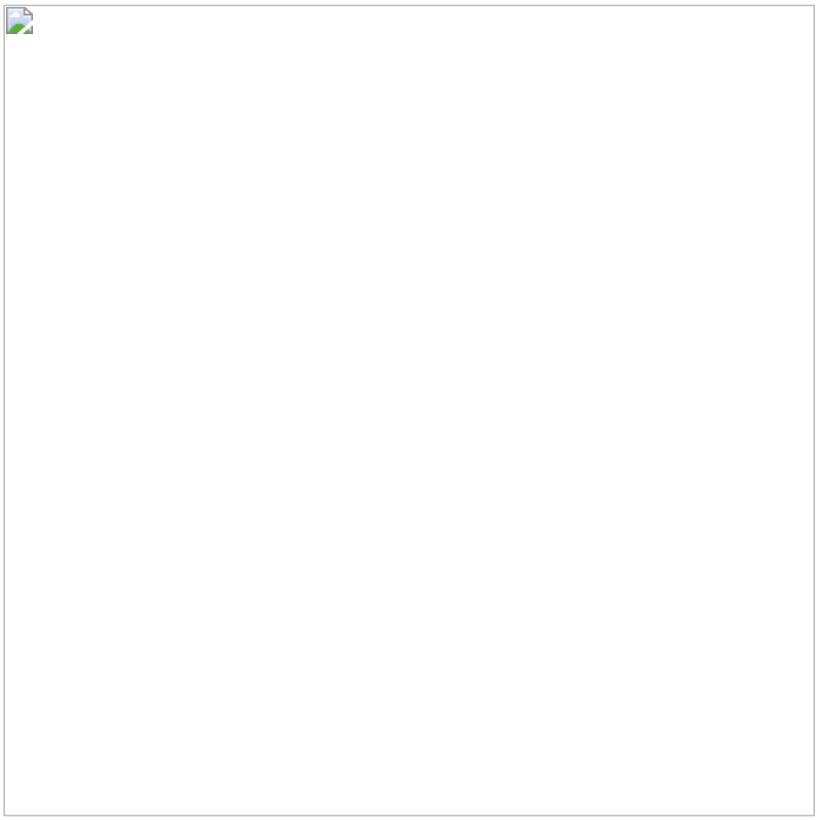


Conflictos en el merge

En el caso en que hubiera conflicto, el merge no se realiza y se queda abierto, hasta que se confirmen cuáles son los cambios correctos o se deshaga el merge.

Los conflictos suceden cuando hay cambios incompatibles entre la rama que se quiere mezclar y la rama en la que se está mezclando. En este caso, podría deberse a haber tocado las mismas líneas de un mismo fichero en los commits `C4` y los commits `C3` y `C5`. En el caso en que los cambios toquen distintas líneas del mismo fichero, git identifica los cambios como compatibles y no lo marca como un conflicto.

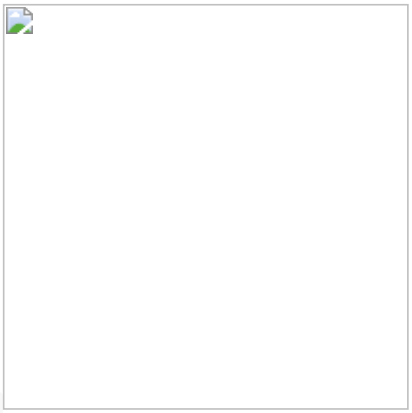
Por ejemplo, en la siguiente figura, en una rama se elimina las últimas líneas de un fichero y en otra se modifican algunas líneas del principio. En este caso no habría un conflicto.



Estrictamente, no hay conflicto en un merge cuando es posible aplicar los cambios de la mezcla en cualquier orden, obteniendo el mismo fichero resultado. En el ejemplo anterior, cualquier orden de ejecución de los cambios tendría como resultado el mismo fichero. Si primero eliminamos las últimas líneas y después se modifican las primeras o si primero se modifican las primeras líneas y después se eliminan las últimas. En ambos casos se obtendría el mismo fichero resultante.

Si aparece un conflicto, Git paraliza el merge e identifica los ficheros en los que existe el conflicto.

```
$ git merge iss56
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```



Git modifica el contenido de los ficheros en los que ha detectado el conflicto, de forma que se marquen los posibles cambios que se introducen en una y otra rama. En la figura de la derecha podemos ver un ejemplo de un fichero que ha sido modificado por Git al detectar un conflicto.

Git marca la zona en la que existe el conflicto y muestra en la parte de arriba el resultado de los cambios de una rama y en la de abajo el resultado de los cambios de otra.

Tendremos que editar los ficheros y dejar el código como queramos. Hay editores que tienen una interfaz especial que permite seleccionar uno de los cambios o los dos. Yo prefiero hacerlo con un editor normal y borrar o modificar las líneas de código que me interesan.

Una vez modificados todos los ficheros en los que hay un error hacer un `add` y al hacer el commit confirmamos el merge:

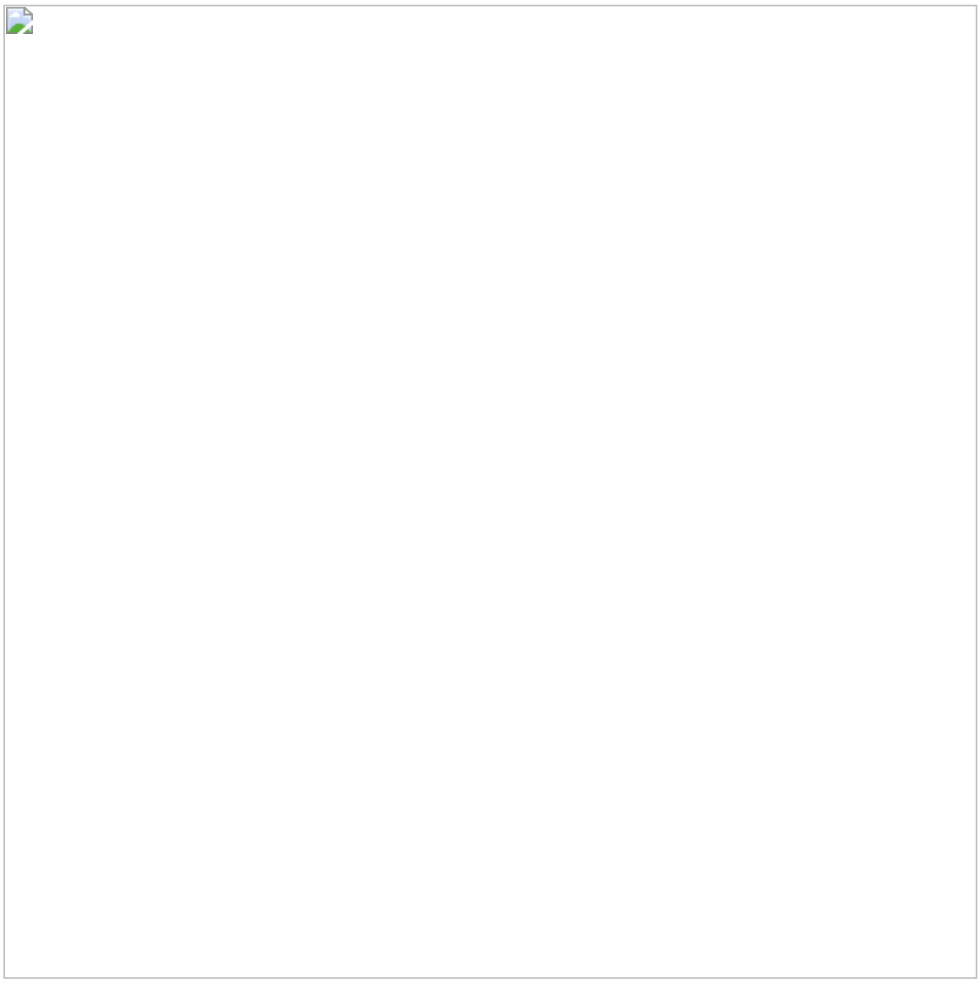
```
$ git add .
$ git status
On branch master
All conflicts fixed but you are still merging.
  (use "git commit" to conclude merge)

Changes to be committed:
```

```
    modified:   index.html
$ git commit -m "Merge branch 'iss56' y resueltos conflictos"
[master 2046b52] Merge branch 'iss56' y resueltos conflictos
```

Ramas remotas

Cuando hacemos un `fetch` del repositorio remoto, Git descarga los cambios en los commits que hay en remoto y los coloca en ramas locales con nombres `<servidor>/<rama>`, para que podamos examinar los cambios antes de hacer la integración.



Por ejemplo, en la imagen anterior alguien ha subido a `master` en el servidor `origin` los commits mostrados en verde. Nosotros hemos añadido dos commits en la rama `master` en nuestra.

La parte inferior de la imagen muestra el resultado de hacer un `git fetch`. Git ha creado una rama local con el nombre `origin/master` en la que se han copiado los commits introducidos en la rama remota.

La rama `origin/master` es una rama local como otra cualquiera. La única diferencia es que es una rama de solo lectura. `HEAD` no puede apuntar a ella (no podemos hacer un checkout y añadir commits como en una rama normal).

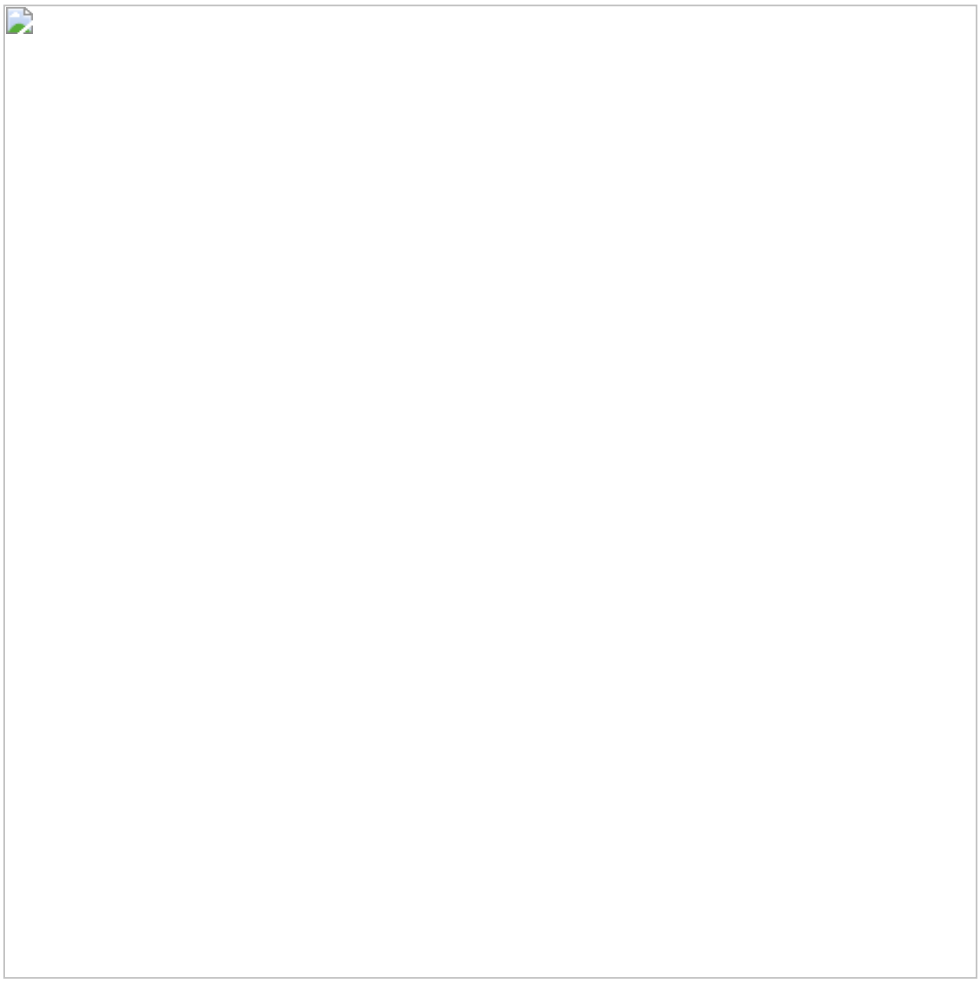
Mezclamos los cambios descargados en master local haciendo un merge.

Una vez en local podríamos comprobar los cambios y después hacer un merge:

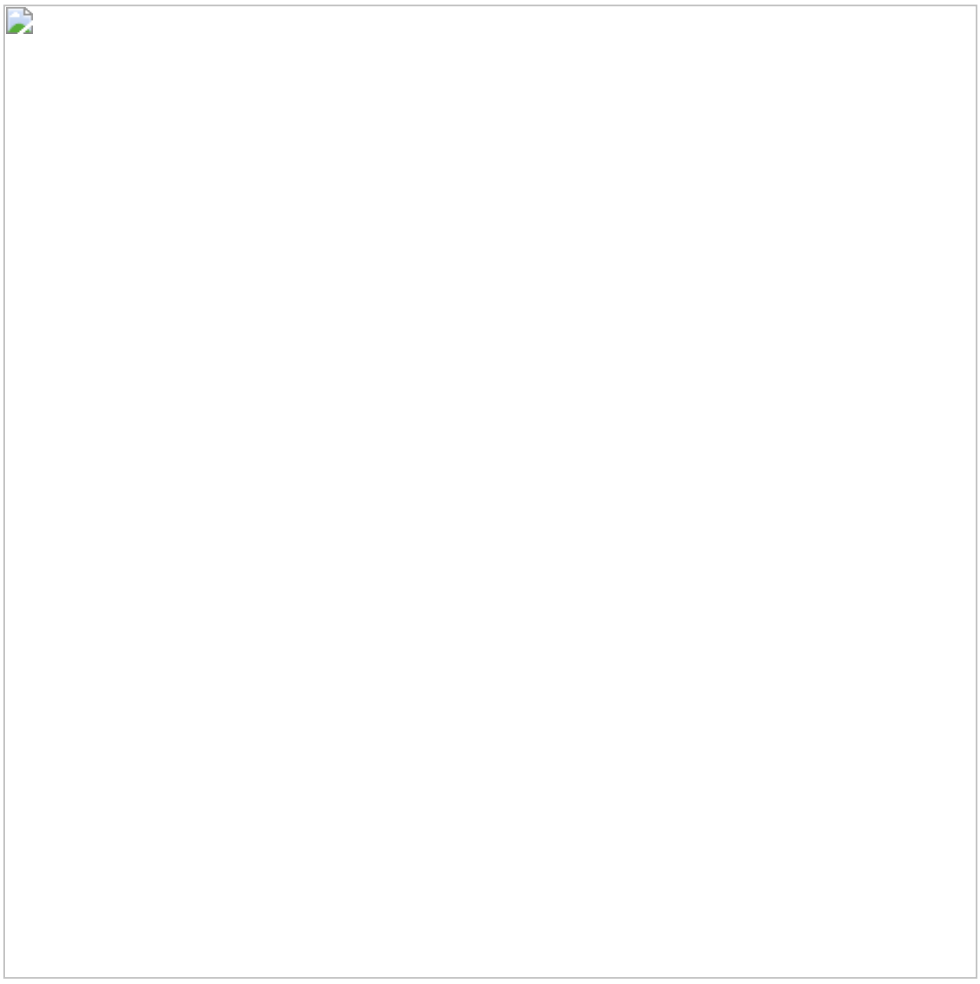
```
$ git diff origin/master
$ git merge origin/master
```

El comando `git pull` hace automáticamente un `git fetch` + `git merge`.

El resultado del merge sería el siguiente:



Y ahora podríamos hacer un `git push` para subir los cambios a `origin` :



El comando `git branch -vva` nos da la información de las ramas locales y las ramas remotas con las que están conectadas:

```
$ git branch -vva
* iss59                 2090ea1 Segundo cambio [iss59]
  master                7c4205e [origin/master] Merge remote-tracking branch 'origin/master'
  remotes/origin/HEAD   -> origin/master
  remotes/origin/master 7c4205e Merge remote-tracking branch 'origin/master'
```

Por último, si en el repositorio remoto algún compañero ha subido alguna nueva rama, al hacer `fetch` nos la bajaremos:

```
$ git fetch
...
From https://github.com/domingogallardo/curso-git-repo1
   a4fe1f4..7c4205e  master     -> origin/master
 * [new branch]      iss59      -> origin/iss59
```

Esa rama ahora una rama remota local, a la que no podemos cambiar. Al hacer un `checkout` se crea una rama local conectada a esa rama remota:

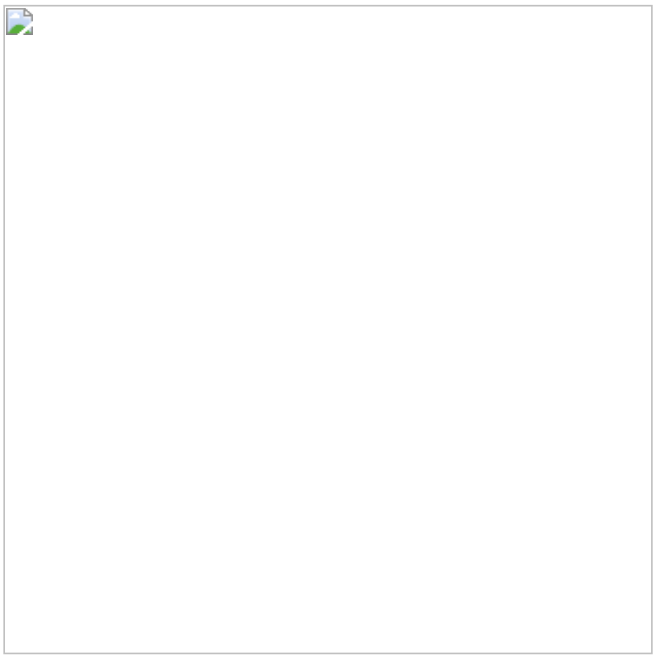
```
$ git checkout iss59
Branch iss59 set up to track remote branch iss59 from origin.
Switched to a new branch 'iss59'
```

Podemos trabajar ahora con esa rama de forma similar a cuando trabajamos con `master`: haciendo `push` y `pull`.

git merge --no-ff

La opción `--no-ff` (no fast forward) en un merge siempre se incluye un nodo de merge, aunque no fuera necesario por estar la rama y `master` en la misma línea de commits.

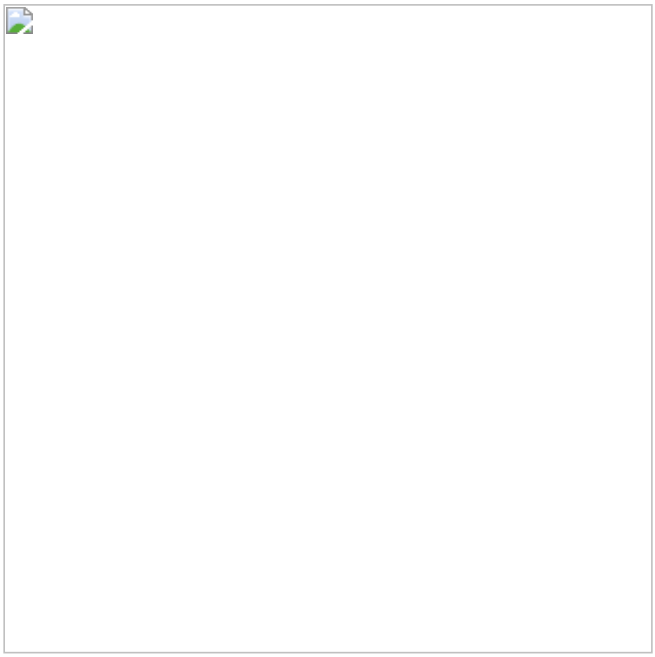
Por ejemplo, en la siguiente figura hemos creado la rama `iss54` y añadido los commits `c4` y `c5` :



Si hiciéramos un `git merge` normal con los siguientes comandos:

```
$ git checkout master
$ git merge iss54
```

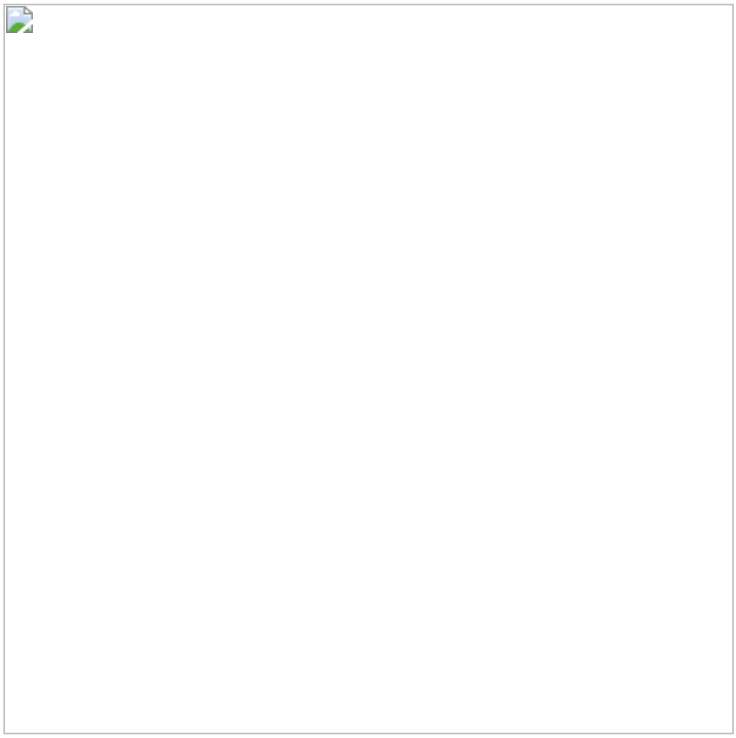
el grafo de commits resultante sería el siguiente:



Sin embargo, si usáramos la opción `--no-ff` :

```
$ git checkout master
$ git merge --no-ff iss54
```

El grafo de commits resultantes es el siguiente:



Esta opción es muy útil para dejar constancia de las mezclas en el grafo con la historia de commits. Es la opción que usa GitHub cuando hace un merge en un pull request.

Cherry-pick

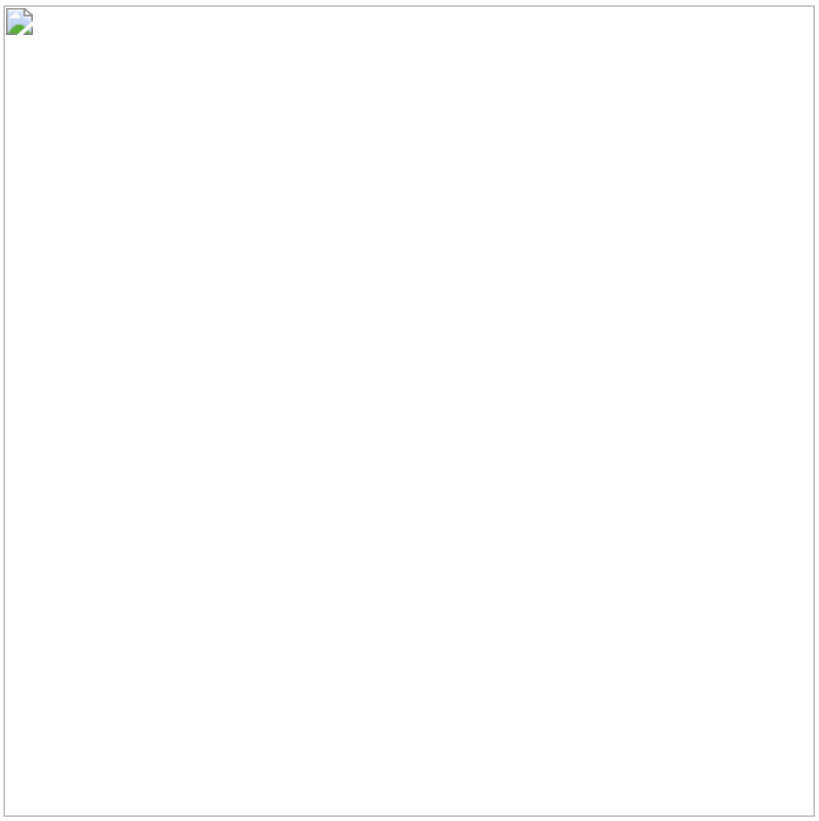
El comando `git cherry-pick` permite mover un commit específico a la rama actual.

Por ejemplo, supongamos el siguiente grafo:

Suponemos que estamos en `master` y queremos añadir los cambios del commit `c4` a la rama actual. Podemos hacer lo siguiente:

```
$ git cherry-pick 2fa6489
```

El grafo resultante será:



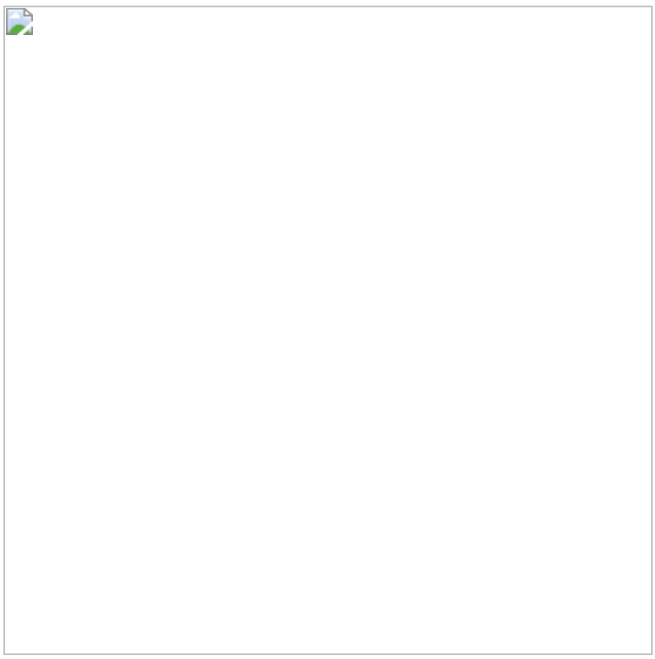
Los cambios del commit `c4` se habrán incorporado a la rama `master`. El commit añadido (`c4'`)contiene los mismos cambios, pero tiene un identificador distinto.

La posterior mezcla de la rama `iss54` con `master` no dará ningún problema, lo único que hemos hecho es adelantar la incorporación de los cambios de un commit a la rama.

Actualizar una rama con los cambios en master

Si estamos trabajando en una rama y hay cambios que se han subido a `master` podemos querer probar que esos cambios no rompen lo que estamos haciendo en la rama, o incorporar alguna funcionalidad que necesitamos para los cambios en los que estamos trabajando.

Por ejemplo, supongamos el siguiente grafo de commits:



Los commits `C8` y `C9` en color verde no están en la rama `featureA` que estamos desarrollando.

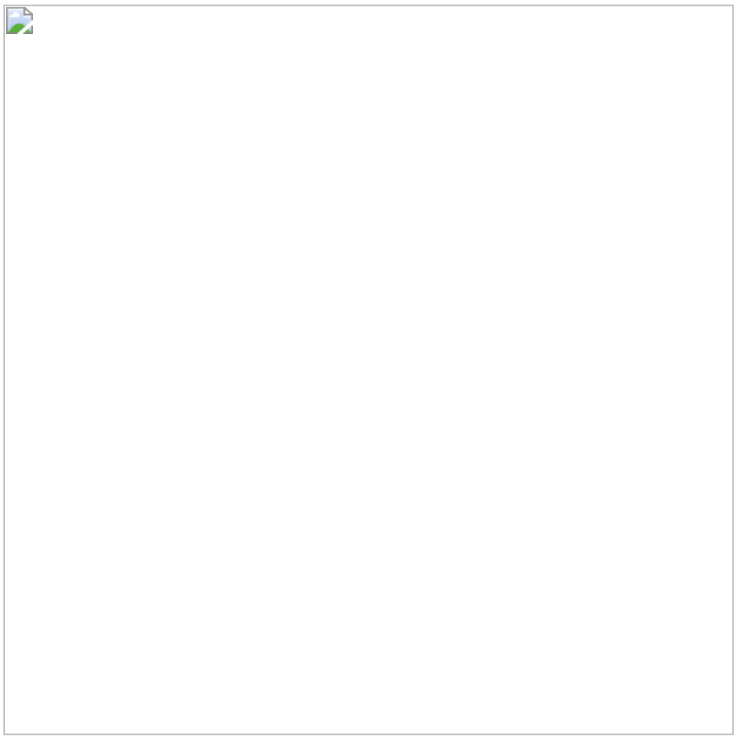
Hay dos formas de incorporar esos commits a `featureA`, haciendo un merge de `master` en la rama o hacer un rebase de la rama en `master`.

Merge de master en otra rama

Si queremos incorporar los nuevos commits de `master` en la rama `featureA` basta con hacer un merge de `master` en `featureA`:

```
$ git checkout featureA
$ git merge master
```

El resultado será el siguiente:



El comando diff sigue funcionando correctamente y muestra sólo los cambios de la rama:

```
$ git diff master...featureA
# Muestra los cambios de los commits C6 y C7
```

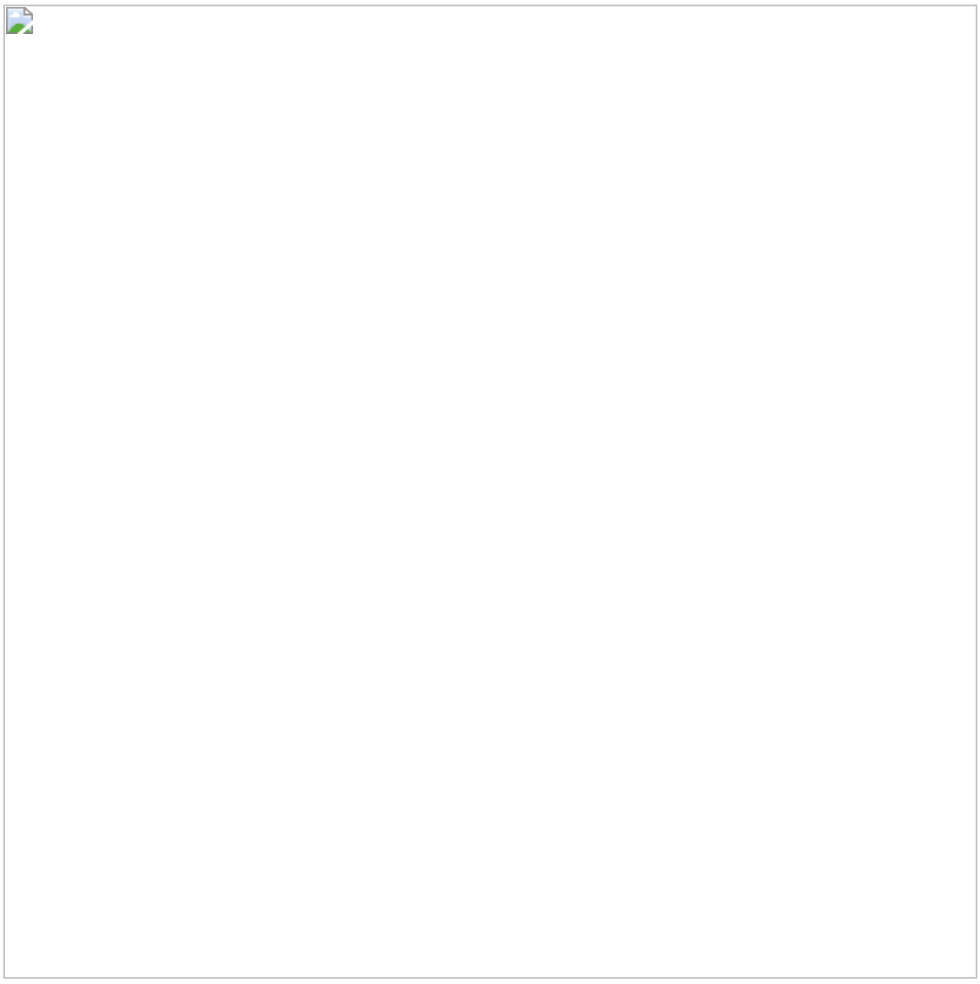
Podemos seguir trabajando en la rama, subirla a repositorio remoto y los compañeros pueden seguir trabajando en ella.

Rebase de la rama sobre master

La otra opción es hacer un rebase y mover el origen de la rama al último commit de master:

```
$ git checkout featureA
$ git rebase master
```

El resultado es el siguiente:

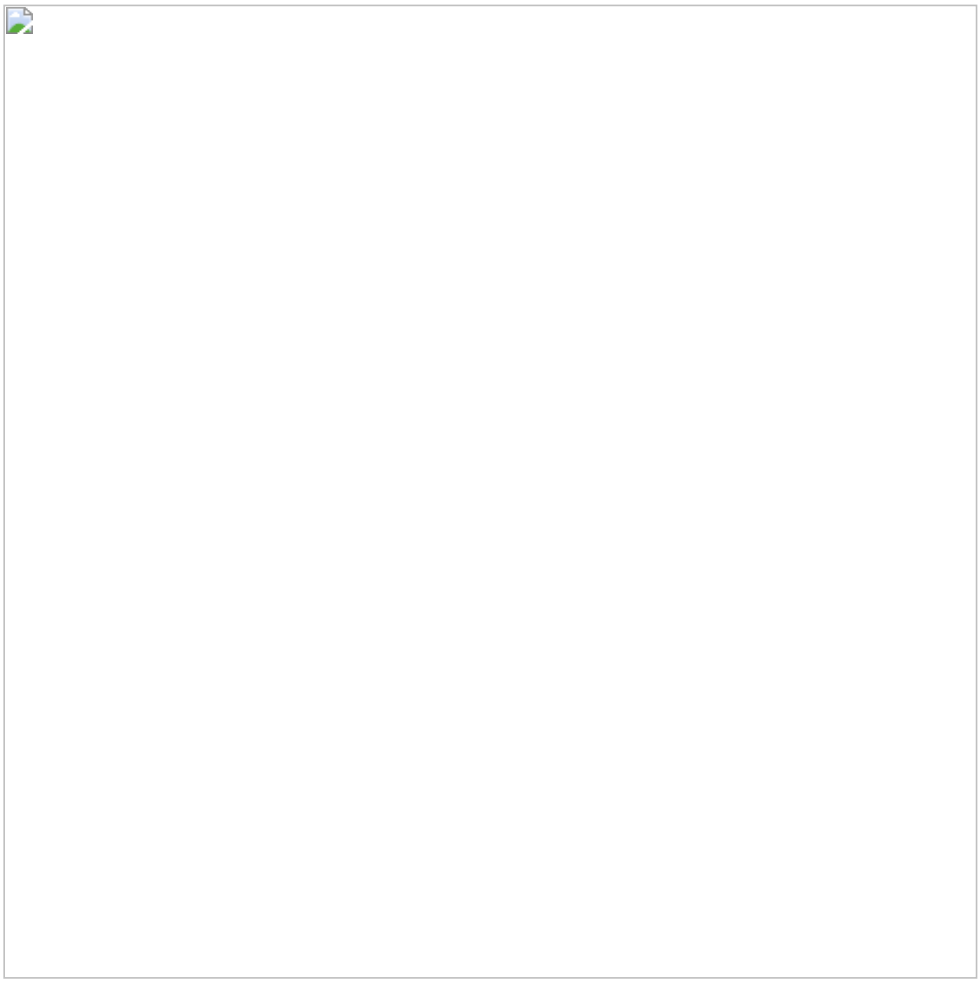


El rebase crea commits nuevos en cabeza de `master` (similar a como lo haría cherry-pick) y mueve el puntero de la rama al último de esos commits.

El problema principal del rebase es que modifica el grafo de historia de commits. Si la rama estaba subida al repositorio, no podremos hacer un `push`, sino que tendremos que hacer un `push --force`. Esto causará problemas a los compañeros que se hayan descargado la rama y estén trabajando en ella.

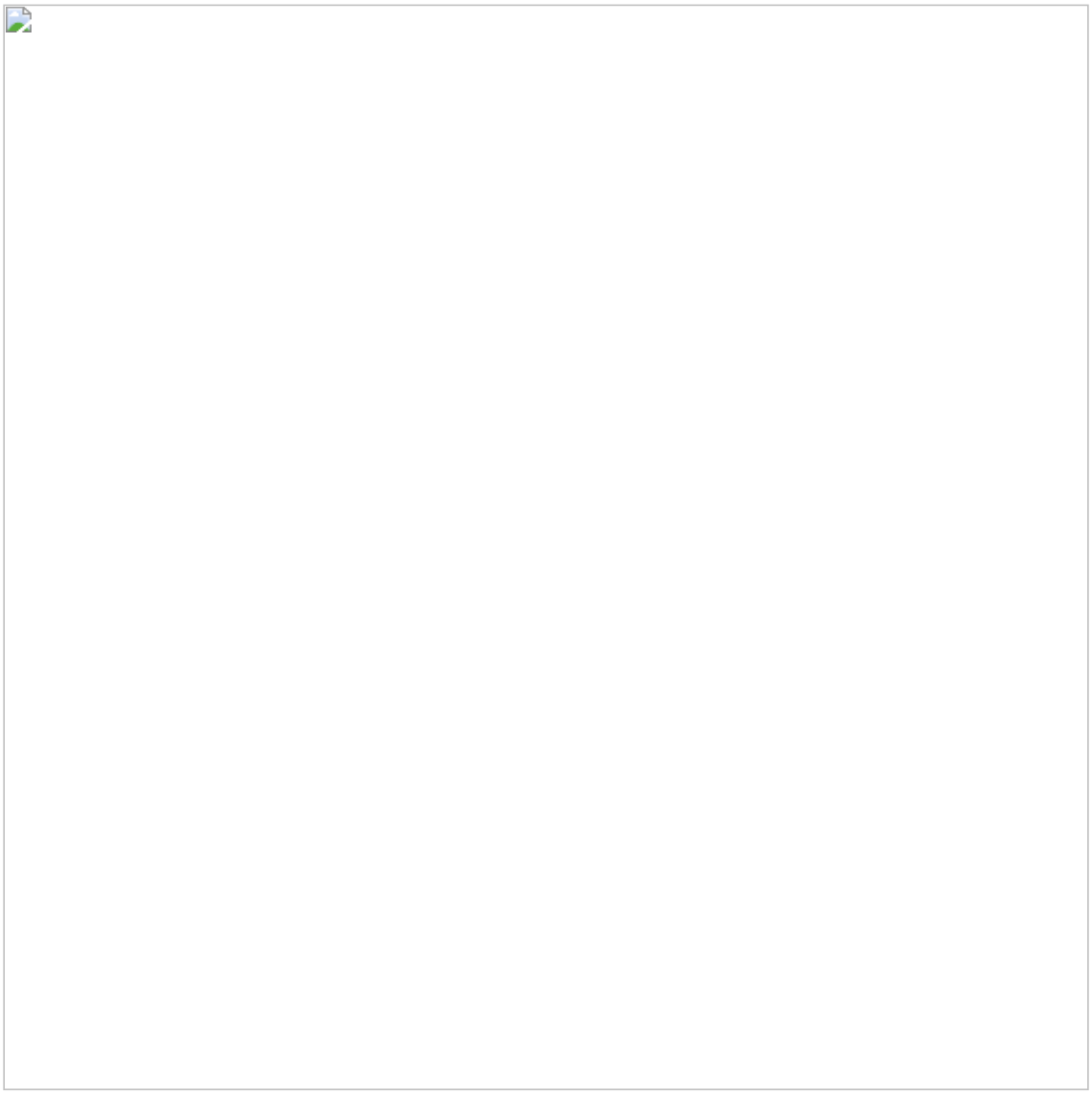
Además, si hay conflictos, pueden ser más complicados de resolver que con el merge.

Pull requests



La idea del `pull request` tiene su origen en el desarrollo de código abierto, para gestionar la posibilidad de que desarrolladores externos puedan contribuir con cambios. El nombre es un poco confuso, yo hablaría mejor de `merge request`. Lo que se pretende con la funcionalidad es, de hecho, solicitar el merge de una rama que se ha subido al repositorio remoto. Esta rama incluso puede provenir de otro repositorio creado a partir de un fork del repositorio original.

Esta funcionalidad no está implementada en el propio Git, sino que se facilita por el servicio que hace el hosting del repositorio (GitHub, GitLab, Bitbucket, etc.). Por ejemplo, en la siguiente imagen se puede ver la interfaz de GitHub para gestionar un pull request.



Como se ve en la ilustración, la interfaz web permite revisar el código que se quiere integrar, abrir conversaciones, aceptar el merge, etc. El servicio de pull request comprueba automáticamente si hay algún conflicto entre la rama que se está subiendo y la rama en la que se va a mezclar. Si lo hubiera, lo marca y no deja realizar el merge.

Es posible seguir subiendo commits a la rama remota mientras el pull request está abierto. La página se actualiza automáticamente con el nuevo commit y se vuelve a comprobar si hay conflictos.

También es posible integrar el pull request con algún servicio de integración continua (como Travis o el propio GitHub Actions). En este caso, no sólo se comprueba si Git marca conflictos con la rama principal sino que también el servicio de integración continua realiza el merge, lanza los tests y comprueba si los tests pasan correctamente.

Una vez aceptado el merge en la interfaz web, la rama remota se integra en la rama principal (o sobre la que estemos haciendo el PR). Podemos borrar la rama remota con el interfaz web.

La próxima vez que cualquier miembro del equipo haga un `pull` se descargará los nuevos commits mergeados.

En concreto, para actualizar en local el resultado del pull request:

```
$ git checkout master
$ git pull
$ git branch -d iss54
$ git remote prune origin
```

Los últimos dos comandos borran la rama mezclada y limpian las referencias a ramas remotas borradas.

Solución de conflictos en pull requests

Como hemos dicho, la página de pull request nos informa si hay conflictos entre los cambios que se van a mezclar y los cambios que se hayan hecho en master.

En la práctica 3 haremos un ejercicio para resolver conflictos en pull requests.

La forma de resolver los conflictos es muy sencilla:

- Hacer un merge de `master` en la rama del PR (en local) y resolver los conflictos.
- Subir el commit de merge a la rama remota y al PR. El PR dejará de mostrar el conflicto.
- Hacer el merge en GitHub.

En GitHub sólo aparecerán en el PR los cambios de la rama que estamos mezclando, no aparecerá código de `master`.

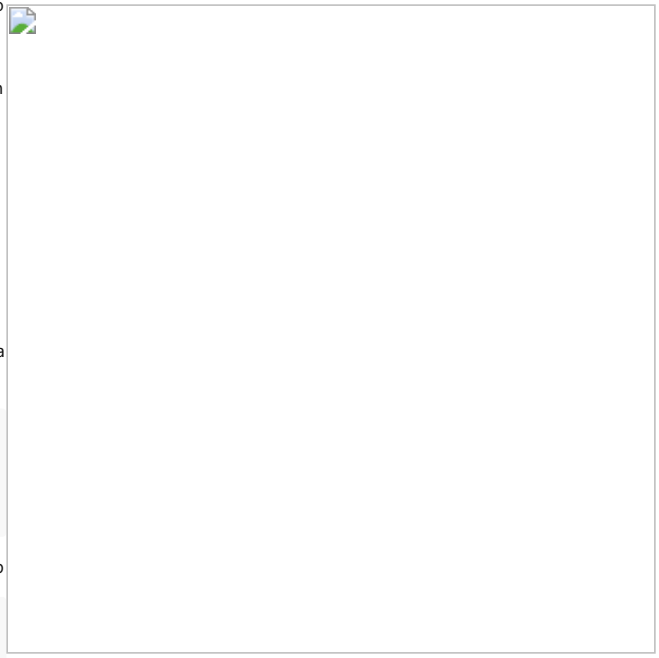
Flujos de trabajo en Git

Dada la cantidad de comandos y la flexibilidad de Git es lógico que no exista una única forma de utilizarlo para trabajar en equipo. Veremos en esta apartado los flujos de trabajo más comunes que se utilizan.

Una idea importante a la hora de analizar los distintos flujos de trabajo es el tiempo de vida de las ramas. Podemos diferenciar entre ramas de corta duración (*short-lived branch*) y ramas de larga duración (*long-lived branch*).

- Una rama de corta duración es aquella que surge de otra, dura lo necesario para hacer un trabajo independiente de la rama original y se termina integrando en ella.
- Un rama de larga duración es una rama que se mantiene para siempre en el repositorio remoto. Podemos tener varias ramas de larga duración para mantener distintas versiones del código.

Desarrollo sobre una rama



Vamos a comenzar con el flujo de trabajo para que un equipo desarrolle sobre una rama, sin hacer ramas adicionales ni pull requests. Los desarrolladores actualizan su versión local de la rama, trabajan sobre esa única rama en local y, cuando tienen el código listo para integrarlo, hacen un push y suben a esa rama los commits que han añadido.

El flujo de trabajo se puede aplicar a la rama `main` o a una rama secundaria en la que un par de desarrolladores están desarrollando una funcionalidad. Vamos a suponer este último caso y supongamos que la rama que están desarrollando se llama `vista-equipos`.

1. En primer lugar, ambos desarrolladores se bajan la rama y se mueven a ella:

```
(Alberto) $ git fetch
(Alberto) $ git checkout vista-equipos
(Ana) $ git fetch
(Ana) $ git checkout vista-equipos
```

2. Desarrollan en la rama y Ana sube los cambios a GitHub

```
(Alberto) # Hace cambios
(Alberto) $ git commit -m "Añadido
botón"
(Alberto) # Cambia más cosas
(Alberto) $ git commit -m "Cambiado tipo
de letra"
(Ana) $ # También hace cambios
(Ana) $ git commit -m "Añadida función
JavaScript"
(Ana) $ git pull (se baja e integra los
posibles cambios del repo remoto)
(Ana) $ git push
```

3. Alberto sube también sus cambios:

```
(Alberto) $ git push (intenta subir sus cambios)
          error: failed to push some refs to
(Alberto) $ git pull (se baja e integra los cambios del reopo remoto)
(Alberto) $ git push
```

4. Ana se descarga los cambios de Alberto:

```
(Ana) $ git pull
(Ana) $ git push
```

Puede suceder que en algún momento haya un conflicto entre los cambios de Ana y Alberto. El primero que sube los commits no tendría que hacer nada y sería el otro el que tendría que bajar los commits, habría un conflicto, y debería solucionarlo y subir la resolución.

GitHub flow

El flujo de trabajo GitHub flow, también denominado de ramas de features es el que hemos estado utilizando en las prácticas de la asignatura hasta ahora.

Es un flujo en el que hay una única rama de larga duración en el repositorio. De esta rama principal se sacan ramas de características en las que se trabaja de forma independiente. En una característica puede trabajar más de una persona usando el flujo de trabajo anterior de desarrollo sobre una rama. Cuando la característica está terminada se integra en la rama principal mediante un pull request.

El pull request se puede usar para revisar el código y para lanzar los tests automáticos mediante un sistema de integración continua como GitHub Actions o Travis.

Los comandos más importantes del flujo de trabajo son los siguientes:

1. Uno de los miembros del equipo crea la rama en la que se va a desarrollar la característica. Se puede subir al repositorio remoto para que algún compañero colabore en el desarrollo:

```
$ git checkout master
$ git checkout -b rama-feature
$ git push -u origin rama-feature
```

2. Se trabaja en la rama, realizando commits que se suben al repositorio remoto. Uno o más compañeros pueden colaborar en el desarrollo usando el flujo de trabajo anterior:

```
$ git commit -m "Cambios"
$ git push
```

3. Cuando se ha terminado el desarrollo se crea un pull request en GitHub. Allí se revisa el código y se termina integrando. Si hubiera algún conflicto se resuelve de la forma que hemos comentado en el apartado sobre pull requests. Se borra en GitHub la rama mezclada.

4. Se actualiza el repositorio local con la mezcla realizada por el pull request y se borra la rama de característica y la referencia a la rama remota:

```
$ git checkout master
$ git pull
$ git branch -d rama-feature
$ git remote prune origin
```

Es posible

Trunk-based development

En el *trunk-based development* todo el equipo utiliza una única rama de larga duración en la que está la versión actual del proyecto. Todos los desarrolladores deben subir los cambios diariamente a esta rama. Es posible trabajar en ramas, pero deben ser ramas que se integren rápidamente, como muy tarde a los dos o tres días.

De esta forma se quiere combatir el problema de que una rama con una característica tarde demasiado en integrarse y la rama principal termine divergiendo demasiado, con los consecuentes problemas en la futura integración.

Al integrarse pronto la rama con el cambio en la rama principal, todos los desarrolladores se verán obligados a tratar con esos cambios pronto y el código evolucionará teniendo en cuenta la presencia de estos cambios.

Esta forma de trabajar es la propia de desarrollos en los que el resultado es una aplicación web en la que existe una única versión en producción y no tienen que mantener versiones anteriores independientes (como sucedería con una app con distintas versiones y que veremos en el

siguiente apartado).

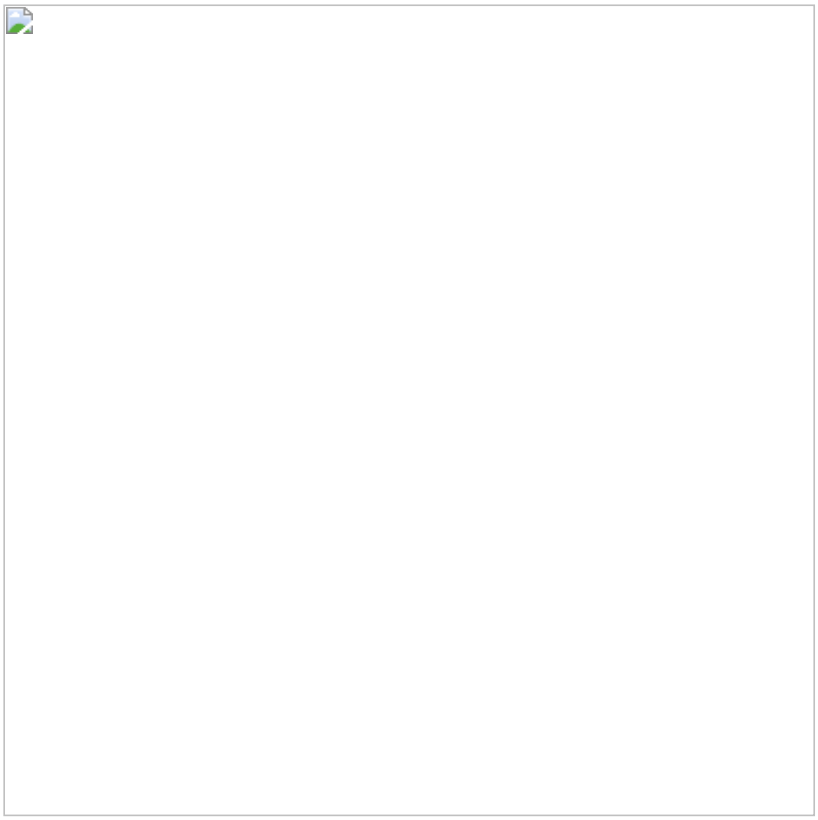
Cuando hablemos de Continuous Delivery comprobaremos que se usa este flujo de trabajo.

Ramas de versiones

En muchas aplicaciones y sistemas software (sistemas operativos, librerías, lenguajes de programación, etc.) es necesario mantener distintas versiones vivas, actualizando y modificando todas ellas de forma simultánea.

Por ejemplo, puedes poner a la venta la versión 3.0 de una app y dar acceso a todos los mantenimientos 3.x. Y al mismo tiempo puede ser que hagas un upgrade con nuevas funcionalidades y vendas la versión 4.0. Habrá usuarios que se muevan a la nueva versión (pagando una pequeña cantidad), pero algunos se quedarán en la 3.x. Deberás mantener ambas versiones, solucionando bugs y realizando pequeños arreglos en las dos.

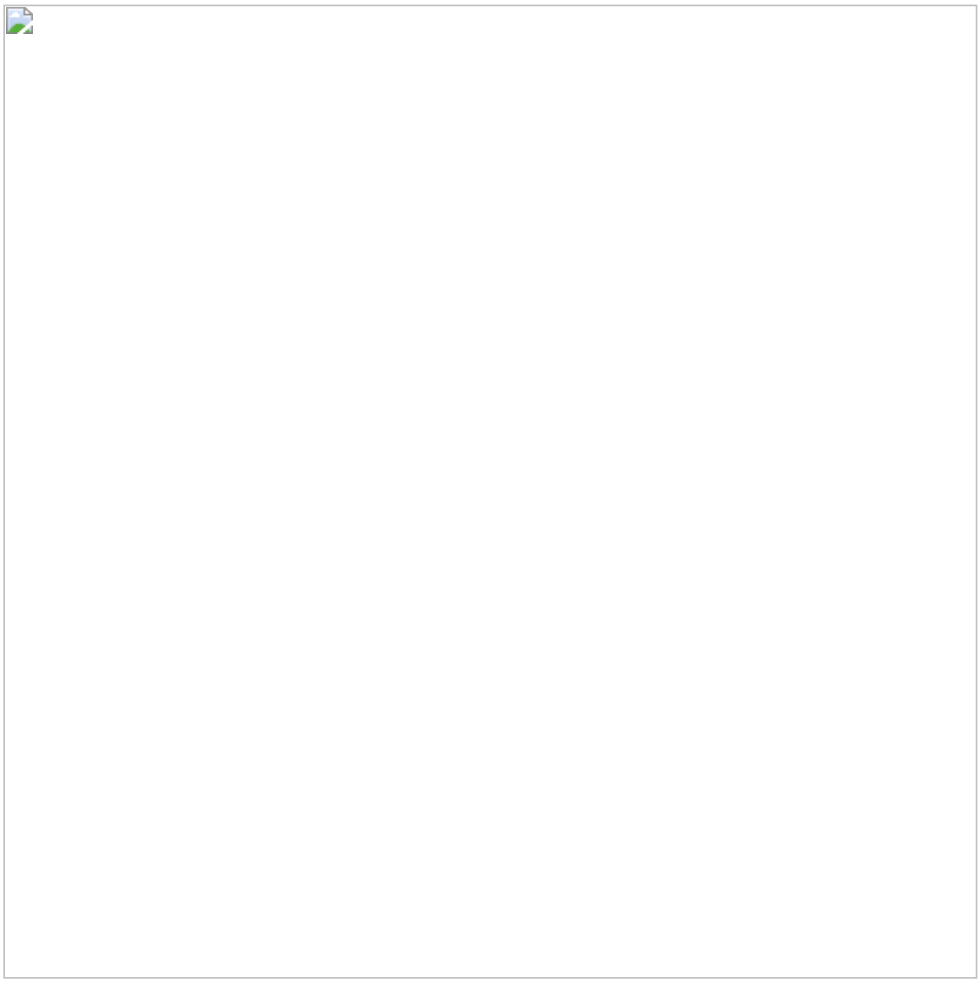
Una forma sencilla de gestionar esto es usando una rama de larga duración por cada una de las versiones (veremos que GitFlow es otra alternativa). Por ejemplo, en la siguiente imagen vemos el repositorio de Spring Boot en GitHub. Puedes visitar la página para estudiar con detalle el flujo de trabajo que usan.



El desarrollo de la versión actual se hace en la rama `master`. Allí abrimos ramas de features y mantenemos la siguiente versión no lanzada.

Cuando lanzamos una nueva versión (mayor o menor, siguiendo la terminología del versionado semántico) creamos una rama nueva y cambiamos la versión de la rama master a la próxima. Por ejemplo, las versiones 3.1 y 3.2 irían en ramas distintas.

Las versiones patch que solucionan bugs (3.1.1 o 3.1.2) se identifican con tags en la rama de la versión menor.



Antes de abrir una rama de versión se puede crear una rama de limpieza (*polishing branch*) en la que estabilizar el código, revisarlo con más detalle y hacer unos últimos arreglos de bugs.

Los cambios se integran después también en master y se abre a partir de ahí la rama con el número de la versión. La rama de limpieza se puede borrar.

Los fixes en la rama de versión se integran también en `master` mezclando la rama de versión en `master` o haciendo un cherry-pick.

GitFlow

El flujo de trabajo GitFlow fue propuesto en 2010 por Vincent Driessen en un artículo que publicó en la web titulado A sucessful Git branching model.

El flujo se hizo muy popular y fue adoptado por muchas empresas. También surgieron muchos comentarios y algunas críticas en las que se proponía flujos de trabajo alternativos:

- GitFlow considered harmful
- Follow-up to 'GitFlow considered harmful'
- A succesful Git branching model considered harmful

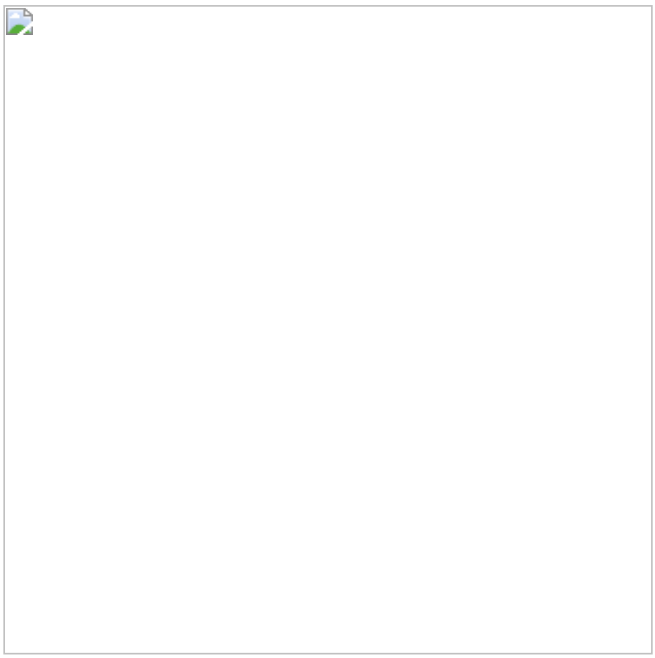
El propio Driessen, en una nota añadida este mismo año en la página, avisa de que su propuesta no es válida para gestionar el flujo de Git de aplicaciones web que son entregadas continuamente y en las que no hay mantener múltiples versiones.

Pero, sin embargo, sí que es un flujo muy útil a la hora de trabajar con aplicaciones en las que se mantiene una única versión antigua que vamos manteniendo con bug fixes y una versión de desarrollo en la que estamos trabajando con la nueva release. Veremos que integra algunos de los flujos ya vistos anteriormente, como el trabajo con ramas de características o el de las ramas de versiones.

El flujo de trabajo contiene dos ramas de larga duración:

- `master`, donde van los commits de release.
- `develop`, la rama de desarrollo, donde se desarrolla la siguiente versión del proyecto.

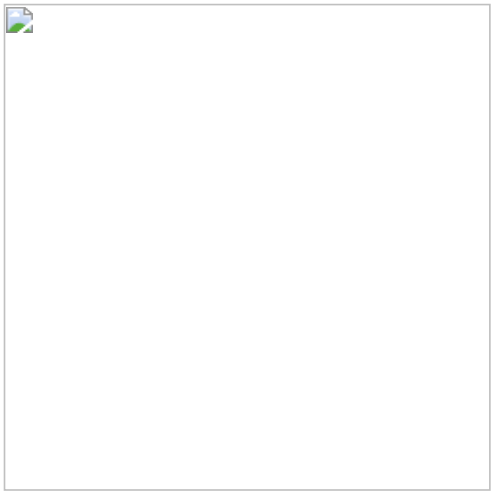
Se muestran en la siguiente imagen:



Se definen distintas ramas de corta duración:

- Ramas de *feature*, en las que se desarrollan las distintas funcionalidades de la nueva versión.
- Ramas de *release*, en las que se prepara el lanzamiento de una nueva versión.
- Ramas de *hotfix*, en las que se corrigen bugs de versiones ya lanzadas.

Ramas de feature



Las ramas de feature salen de `develop` y se mezclan en `develop`. Contienen cambios que añaden funcionalidades a la versión actual en desarrollo.

Pueden recibir cualquier nombre, salvo `master`, `develop`, `release-*` y `hotfix-*`, que son nombres reservados para los otros tipos de rama.

El trabajo con estas ramas es similar al del flujo de trabajo con ramas de características. Pueden subirse al servidor y compartirse con otros desarrolladores para hacer trabajo en común. Y pueden integrarse de nuevo en `develop` haciendo un merge o un pull request.

Cuando Driessen propuso el flujo de trabajo, no se había desarrollado todavía la idea de los pull requests, por lo que él sólo habla de hacer merges. Nosotros en la práctica utilizaremos el pull request como forma de añadir la funcionalidad a la rama principal de desarrollo.

Ramas de release

Las ramas de release son ramas de corta duración que salen de `develop` y se mezclan en `master` y en `develop`. Deben llamarse `release-<numero de release>`.

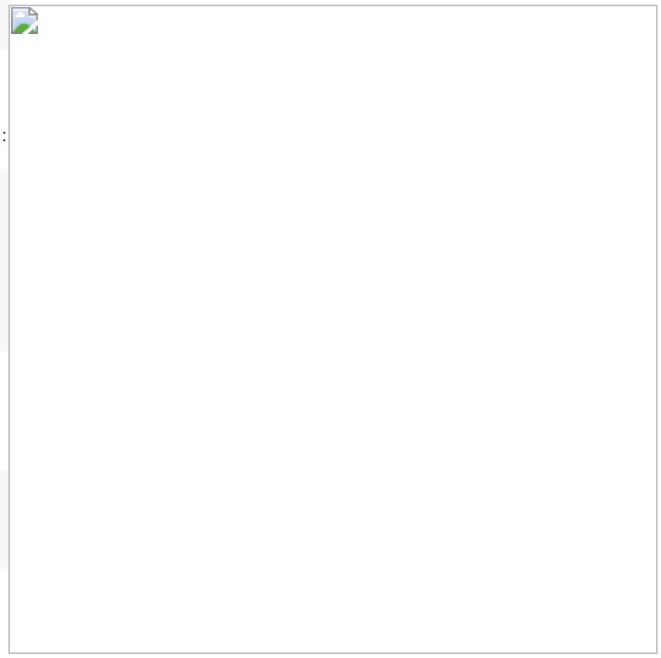
Contienen un commit en el que se modifica el número de la versión y otros en los que se corrigen y añaden cuestiones de última hora.

Cuando todo está listo para lanzar la versión, se mezcla con `master` y se pone una etiqueta en el commit de mezcla. Esa etiqueta marca la versión lanzada. También se mezcla con `develop` para integrar los últimos cambios que se han añadido a la versión.

Por ejemplo, supongamos que la última versión en producción es la `1.1.5` y que hemos añadido en `develop` nuevas características para una nueva versión, que llamaremos `1.2`. Abrimos entonces una rama de release desde `develop` y cambiamos el número de versión (con un script que llamamos `bump-version.sh`:

```
$ git checkout develop
$ git checkout -b release-1.2
```

```
$ ./bump-version.sh 1.2
$ git commit -am "Cambiada la versión a 1.2"
```



Una vez que hemos creado esta rama, hacemos una última revisión y corregimos los últimos bugs que detectamos. Podemos mezclar estas correcciones también con `develop`:

```
$ git checkout release-1.2
# Corregimos un bug
$ git add .
$ git commit -m "Bug corregido"
$ git checkout develop
$ git merge release-1.2
```

Por último, cuando ya queremos hacer el release, mezclamos la rama con `master` y etiquetamos el commit con el número de versión:

```
$ git checkout master
$ git merge release-1.2
$ git tag -a 1.2
```

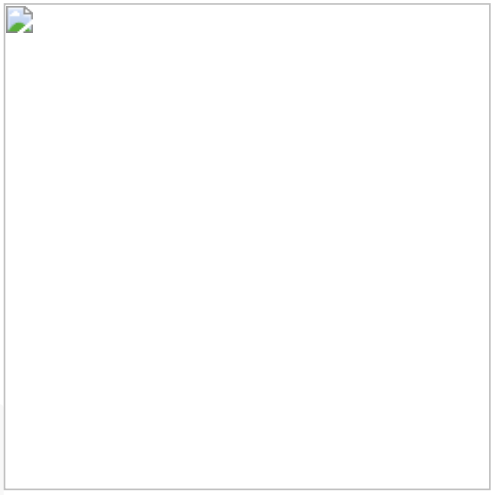
Por último, mezclamos en `develop` para asegurarnos de que todos los cambios en la versión también se pasan a la rama de desarrollo. Y podemos borrar la rama:

```
$ git checkout develop
$ git merge release-1.2
$ git branch -d release-1.2
```

Todos los merges anteriores (o los que contengan código importante que deba ser revisado) pueden hacerse mediante pull requests.

Ramas de hotfix

Las ramas de hotfix son ramas de corta duración que salen de `master` y se mezclan en `master` y `develop`. Su nombre es `hotfix-<numero de versión>`.



En la rama se guaradan commits que reparan bugs encontrados en alguna release y también se modifica el número de la versión.

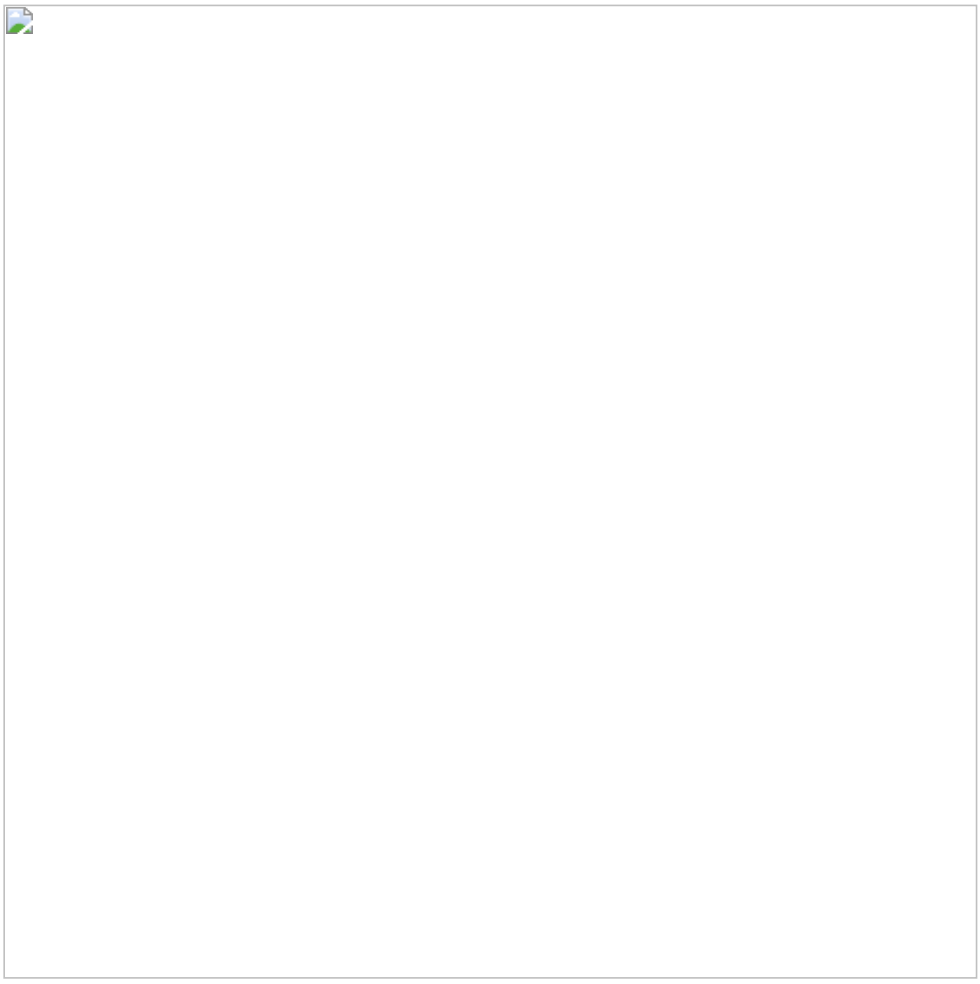
Por ejemplo, supongamos que hemos encontrado unos bugs que queremos reparar en la versión lanzada en `master`, la `1.2`. Deberemos ir a `master`, sacar de allí una rama de hotfix y arreglar los bugs en ella. También cambiaremos el número de versión.

Una vez arreglados lo bugs, crearemos una nueva versión mezclando con `master`. Mezclaremos también con `develop` para copiar también allí las correcciones. Y, por último, borraremos la rama de hotfix.

```
$ git checkout master
$ git checkout -b hotfix-1.2.1
$ ./bump-version.sh 1.2.1
$ git commit -am "Cambiada la versión a 1.2.1"
$ # Corregimos los bugs
$ git add .
$ git commit -m "Corregidos los bugs"
$ git checkout master
$ git merge hotfix-1.2.1
$ git tag -a 1.2.1
$ git checkout develop
$ git merge hotfix-1.2.1
$ git branch -d hotfix-1.2.1
```

Igual que antes, los merges anteriores pueden realizarse mediante pull requests.

La siguiente ilustración muestra el flujo completo, con las distintas opciones y ramas que hemos explicado:



Hay que hacer notar que GitFlow sólo permite corregir los bugs de la release más reciente. En la figura, por ejemplo, no podríamos corregir algún bug que encontráramos en la versión `0.2` siguiendo estrictamente las indicaciones del flujo. En el flujo se dice que para corregir un bug debemos ir a `master` y corregirlo allí. Entonces estaríamos corrigiendo un bug detectado en la `0.2` en la versión más reciente `1.1`. Si alguien quiere arreglar ese bug debería descargarse la versión más reciente y no podría continuar con la versión `0.2`.

Para solucionar este problema podríamos arreglar el bug en una rama nueva que abrimos a partir del commit `0.2`, pero entonces ya estaríamos utilizando otro flujo de trabajo (el de ramas de versiones).

Tal y como dice Driessen, lo que tenemos que hacer es conocer bien las distintas técnicas y posibilidades y configurar un flujo de trabajo adaptado a nuestras necesidades (y no seamos *haters*).

> *To conclude, always remember that panaceas don't exist. Consider your own context. Don't be hating. Decide for yourself.*
>
> *Vincent Driessen*

Referencias

- Scott Chacon, Ben Straub (2014), *Pro Git*
- Atlassian *Comparing workflows*
- Vincent Driessen (2010), Git Flow