GIT- Colaboración

Índice- Colaboración con GIT

- Comandos vinculados con las sincronización de repositorios
  - git clone
  - git remote
  - git fetch
  - git pull
- Comandos vinculados con el uso de ramas
  - git branch
  - git checkout
  - git merge
  - git rebase [-i]
  - git push
- Merging vs Rebase
  - La regla de oro de Rebasing
- Tag
- Referencias relativas a commit
- gitignore
- Buenas prácticas

Índice- Colaboración con GIT

En Git cada desarrollador tiene su propia copia del repositorio completo con su propia historia y las ramas de la estructura local. Los usuarios necesitan compartir una serie de “commit” en lugar de un único conjunto de cambios.

Git te permite compartir ramas enteras entre repositorios.

Los comandos para sincronizar los repositorios:

- git remote
- git fecth
- git pull
- git push

Nos permiten:

- Manejar las conexiones con otros repositorios [varios]
- Publicar la historia local "push" de las ramas a otros repositorios.
- Ver las contribuciones de otros "pull" o “fetch”.

Comando Git: git clone

El comando **git clone** genera un repositorio local a partir repositorio Git existente.

Como ventaja:

la clonación crea automáticamente una conexión remota llamada origen que señala al repositorio original. Esto hace que sea muy fácil de interactuar con un repositorio central.

Uso:

```
$ git clone <repo>
```

Clona el repositorio localizado en repo sobre la máquina local.

Comando Git: git clone

El repositorio original puede estar localizado en un sistema de ficheros o sobre una máquina remota accesible vía SSH o vía HTTP

Otro uso del comando

$ git clone <repo> <directorio>

Clona el repositorio localizado en <repo> en el directorio <directorio> sobre la máquina local.

Comando Git: git clone

Si un proyecto ya está en un repositorio central, el comando git clone es la forma más común que los usuarios obtengan una copia de desarrollo.

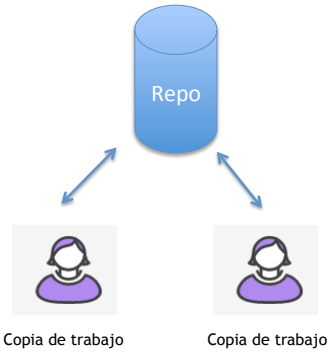
Al igual que `git init`, la clonación es generalmente una sola vez.

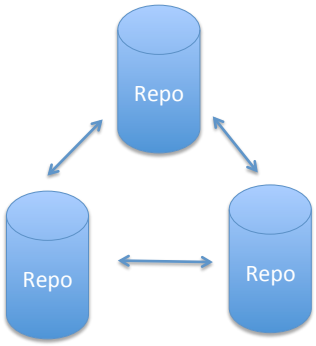
De una única operación el desarrollador ha obtenido una copia de trabajo del repositorio central.

Todas las operaciones de control de versiones y colaboraciones se gestionan a través de su repositorio local.

Comando Git: git clone

El modelo de colaboración de Git se basa en la colaboración **de repositorio-a-repositorio.**





Comando Git: git remote

El este comando permite crear, ver y eliminar las conexiones con otros repositorios.

Las conexiones remotas pueden ser consideradas más como alias que como enlaces directos a otros repositorios.

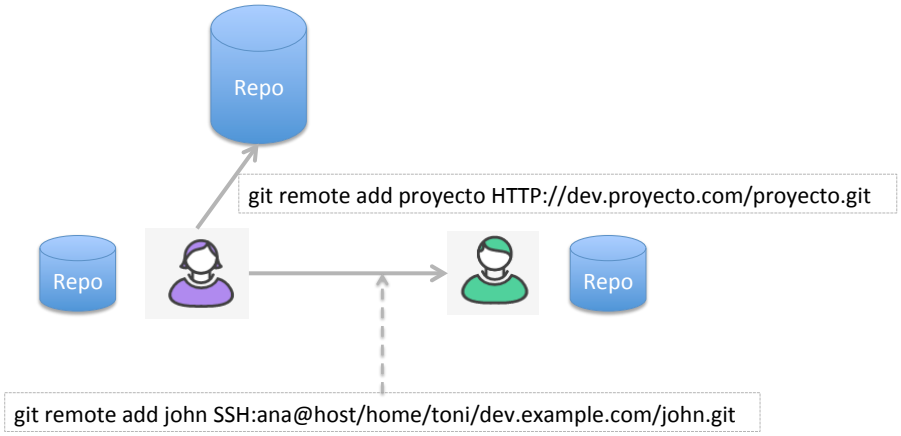
No proporcionan enlaces en tiempo real. Sirven como nombres que pueden utilizarse para hacer referencia a una URL|SSH. Ya que siempre será más complejo utilizar la URL completa o la conexión SSH.

Este comando lo que hace es generar un nombre más sencillo para utilizarlo en aquellos comando que hemos de utilizar una URL (de un repositorio).

Git no actualiza automáticamente los repositorios. Para actualizar un repositorio local con las ultimas actualización hay que “pull” del repositorio remoto y “push” para lo contrario.

Comando Git: git remote

En el siguiente diagrama muestra dos conexiones remotas desde un repositorio: una a el repositorio central y otra al repositorio de otro desarrollador.



Comando Git: git remote

Usos:

COMANDO	DESCRIPCIÓN
`$ git remote`	Muestra las conexiones remotas con otros repositorios.
`$ git remote -v`	Muestra las conexiones remotas que tiene con otros repositorios incluyendo la URL de cada conexión.
`$ git remote add <nombre> <url>`	Crea una nueva conexión con un repositorio remoto. El nombre será usado como alias de la URL en otros comandos git.
`$ git remote rm <nombre>`	Elimina la conexión con el repositorio remoto llamado <nombre>.
`$ git remote rename <nombre-actual> <nuevo nombre>`	Permite renombrar una conexión remota, solo afecta al nombre asociado de la conexión.

Comando Git: git remote

Cuando clonamos un repositorio con "`git clone`" automáticamente se crea una conexión remota llamada "origin" como punto de retorno para comandos que sincronizan el repositorio. (`pull` o `fetch` y `push`).

Comando Git: git fetch

COMANDO	DESCRIPCIÓN
`$ git fetch <remote>`	Este comando recupera todos los datos del proyecto remoto que no estén en local. Después de ejecutar el comando, se obtienen las referencias a todas las ramas del repositorio remoto, que se pueden unir o inspeccionar en cualquier momento.
`$ git fetch <remote> <branch>`	Lo mismo que el anterior comando pero solo se recupera una rama específica.

El comando fetch sólo recupera la información y la pone en el repositorio local, no la une automáticamente en nuestro repositorio ni modifica ningún archivo del directorio de trabajo.

Si se desea unir se deberá hacer manualmente en otro momento, mediante el comando `git merge` (se verá posteriormente)

Comando Git: git pull

La fusión de los cambios ascendentes en su repositorio local es una tarea común en los flujos de trabajo de colaboración basado en Git.

Ya vimos que se puede ha hacer esto con `git fetch` seguido por `git merge`, pero `git pull` lo hace en un solo comando.

El comando `git pull` actualiza el repositorio local al último commit.

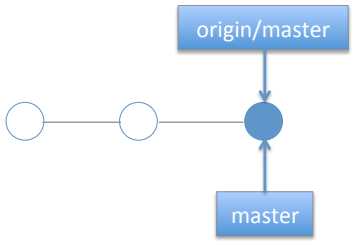
Al ejecutar `git pull` en el directorio de trabajo se bajan y fusionan los cambios remotos.

Comando Git: git pull

COMANDO	DESCRIPCIÓN
`$ git pull <remote>`	Buscar la copia remota especificada de la rama actual e inmediatamente la fusiona con la copia local. Lo que hace de una solo vez: git fetch <remote> y git merge origin /<rama-actual>
`$ git pull --rebase <remote> <branch>`	Lo mismo que el anterior comando, pero en lugar de hacer un merge realiza un rebase git fetch <remote> y git rebase origin /<rama-actual>

Comando Git: git pull

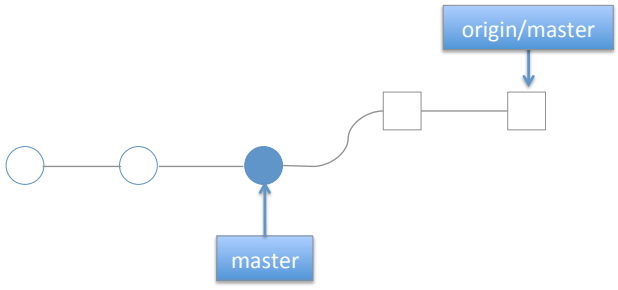
Es una manera fácil de sincronizar el repositorio local con cambios que hay en el repositorio central o en el repositorio de otro desarrollador.



Empezamos con un repositorio sincronizado

Comando Git: git pull

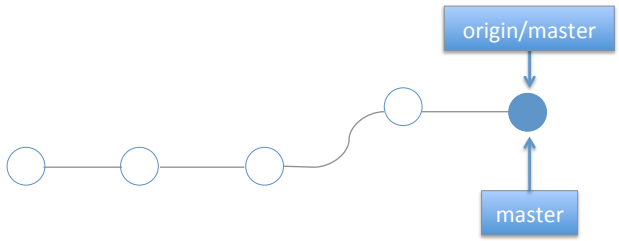
Es una manera fácil de sincronizar el repositorio local con cambios que hay en el repositorio central o en el repositorio de otro desarrollador.



`$ git fetch` revela que la versión de la rama master del repositorio origen ha progresado desde la última vez se examinó.

Comando Git: git pull

Es una manera fácil de sincronizar el repositorio local con cambios que hay en el repositorio central o en el repositorio de otro desarrollador.



`$ git merge` integra el repositorio local con el repositorio remoto.

Comando Git: git pull

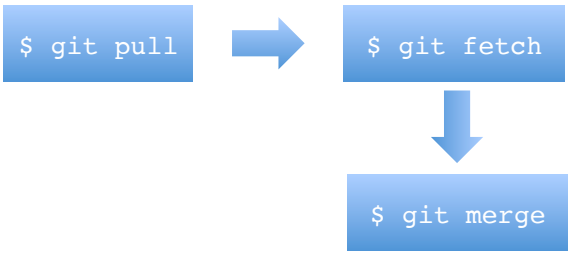
La opción --rebase se puede utilizar para asegurar una historia lineal mediante la prevención de commit innecesario cuando se ejecuta un `git merge.`

Muchos desarrolladores prefieren cambio de base sobre el merge, ya que es como decir: "Quiero poner mis cambios en la parte superior de lo que todo el mundo ha hecho".

De hecho `$ git pull --rebase` es una forma de trabajar tan común que se puede configurar en git de forma que siempre se haga un pull con la opción rebase, para ello hemos de ejecutar configurar git con el siguiente comando:

```
$ git config --global branch.autosetuprebase always
```

Comando Git: git fetch vs git pull



Se puede hacer un `git fetch` en cualquier momento para actualizar las ramas remotas que sigues que se encuentran en refs/remotes/<remote>/.

Esta operación nunca cambia nuestras ramas locales en refs/heads y es seguro de hacerlo ya que no cambia la copia de trabajo.

Un `git pull` hace que tu rama local este actualizada con la versión remota, mientras que también actualiza tus otras ramas remotas que sigues.

Comando Git: git branch

Una rama representa una línea independiente del desarrollo.

Son una abstracción del proceso: edit/stage/commit.

Se puede pensar en ellas como una manera de solicitar:

- un nuevo directorio trabajo
- una nueva área de ensayo
- una nueva historia del proyecto.

Nuevas confirmaciones se registran en la historia de la rama actual, lo que se traduce en una bifurcación en la historia del proyecto.

Comando Git: git branch

El trabajo con ramas forma parte del día a día del procesos de desarrollo utilizando Git como sistema de control de versiones. Cada vez que se quiera añadir una nueva funcionalidad o corregir una incidencia se creará una nueva rama que encapsulará los cambios.

Las ramas facilitan que el código inestable nunca forme parte de la historia oficial del proyecto.

Las ramas dan la oportunidad de limpiar el historial de los commit's antes de fusionar la nueva rama con la rama principal del proyecto.

Comando Git: git branch

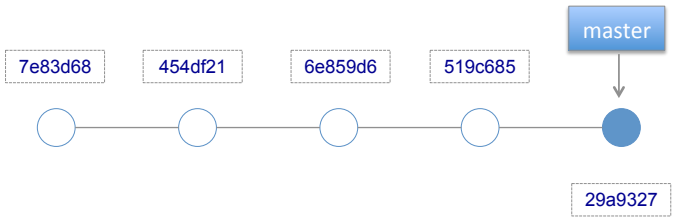
COMANDO	DESCRIPCIÓN
`$ git branch`	Lista todas las ramas de un repositorio.
`$ git branch <nombre>`	Crea una nueva rama llamada "nombre".
`$ git branch –d <nombre>`	Borra la rama que se llama "nombre". La operación es segura, nos informa si vamos a eliminar una rama que todavía no se ha fusionado.
`$ git branch –D <nombre>`	Borra la rama "nombre" aunque no se haya fusionado. Es un comando que se puede usar para borra todos los commit que se hayan hecho de un determinado desarrollo.
`$ git branch –m <nombre>`	Renombra la rama en la que estamos situados.

Comando Git: git checkout

Se usa para ver un estado anterior de un proyecto sin alterar el estado actual. `$ git checkout` de un archivo permite ver una versión antigua de ese archivo, dejando el resto de su directorio de trabajo sin tocar.

El comando git checkout tiene tres funciones:

- checkout files
- checkout commit
- checkout branch



Comando Git: git checkout

Permite navegar entre ramas creadas mediante el comando git branch.

Actualiza los archivos en el directorio de trabajo para que coincida con la versión almacenada en esa rama, y le dice a Git que registre todas las nuevas confirmaciones en esa rama.

Piense en ello como una manera de seleccionar en qué línea de desarrollo está trabajando.

Cuando cambiamos a una rama no solo cambiamos el directorio de trabajo si no que también los nuevos commit se guardan en la historia del proyecto, es decir, que no es una operación de sólo lectura.

Comando Git: git checkout

COMANDO	DESCRIPCIÓN
`$ git checkout <branch>`	Nos posicionamos en la rama. La rama debe de haber sido creada con anterioridad. Hace que "branch" sea la rama actual y modifica el directorio de trabajo.

`$ git checkout -b <branch>`	Crea y se posición en la nueva branch. El flag "–b" le dice a git que haga:
	`$ git branch <branch> &`
	`$ git checkout <branch>`

`$ git checkout -b <new-branch> <existing-branch>`	Igual que la anterior, pero la rama nueva se bifurca de la rama <existing-branch>

Comando Git: git merge

La fusión es la manera de Git de unir nuevamente una bifurcación de la historia de un proyecto.

El comando $ `git merge` permite tomar las líneas independientes de desarrollo creadas por las ramas e integrarlos en una sola rama.

Tenga en cuenta que todos los comandos se presentan a continuación se funden en la rama actual.

La rama actual será actualizada para reflejar la fusión, pero la bifurcación de destino no se verá afectada en absoluto.

`$ git merge` se utiliza a menudo en combinación con:
- `$ git checkout` para seleccionar la rama actual
- `$ git branch -d` para borrar la rama obsoleta.

Comando Git: git merge

COMANDO	DESCRIPCIÓN
`$ git merge <rama>`	Fusiona la rama especificada con la rama actual. Git determinará que algoritmo de fusión utilizará de forma automática.
`$ git merge --no-ff <rama>`	Fusiona la rama especificada con la rama actual, fuerza generar un commit merge independiente del algoritmo de fusión utilizado. Se utiliza para documentar todos los merge que ocurran en el repositorio.

Comando Git: git merge

Una vez terminado el desarrollo de una característica en una rama aislada, hemos de poder generar una nueva versión del código de nuestro proyecto que incluya la nueva rama

Dependiendo de la estructura de su repositorio, Git tiene dos algoritmos distintos para lograr la fusión:

- Fast-Forward merge.
- 3-way merge.

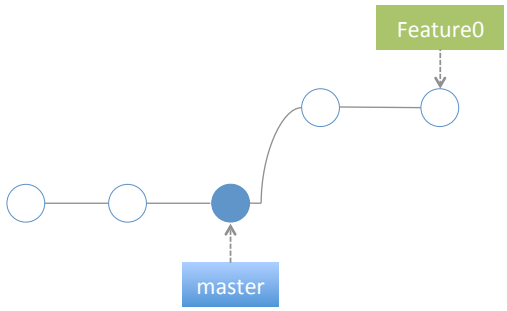
Comando Git: git merge: Fast-Forward

Este método de fusión se utiliza cuando hay una trayectoria lineal entre la rama actual y la bifurcación destino.

En lugar de fusionar las ramas en "realidad" todo lo que Git tiene que hacer para integrar las historias es decir un "avance rápido" y avanzar la rama actual hasta la punta de la rama de destino.

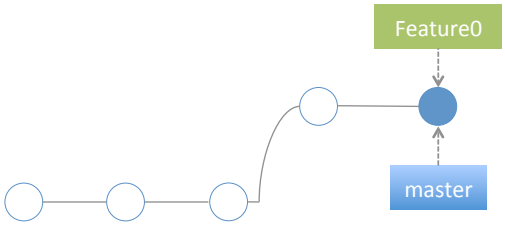
Esto combina con eficacia las historias, ya que todas las confirmaciones accesible desde la bifurcación del destino ya están disponibles a través de la rama actual.

Comando Git: git merge: Fast-Forward

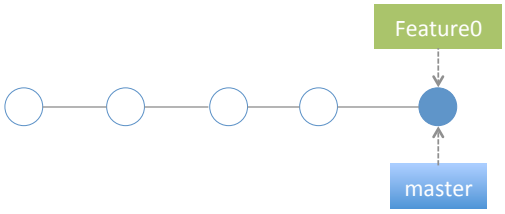



```
$ git merge Feature0
```

Comando Git: git merge: Fast-Forward



Comando Git: git merge: Fast-Forward



Comando Git: git merge: 3-way merge

No siempre es posible fusionar ramas utilizando el algoritmo fast-forward.

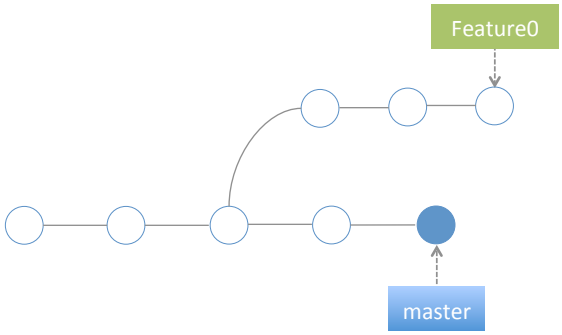
Cuando no hay un camino lineal desde la bifurcación de destino hasta el top de la rama, Git no tiene más remedio que combinarlos a través 3-way-merge.

Este algoritmo de fusión de ramas utiliza un **commit dedicado y único para** unir las dos historias.

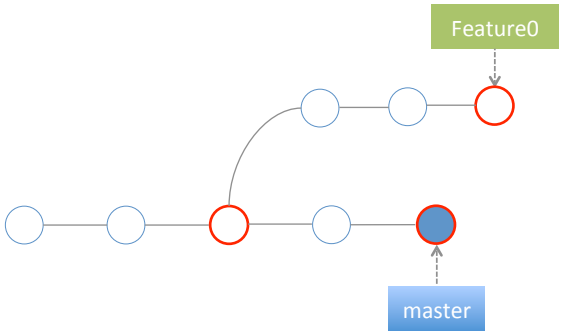
El nombre viene del hecho de que Git utiliza tres commit para generar el commit de la fusión:

- Los commit de los dos extremos de las ramas
- Y el commit del ancestro común.

Comando Git: git merge: 3-way merge

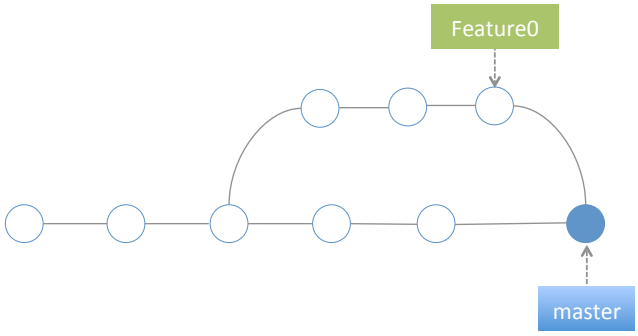


Comando Git: git merge: 3-way merge




```
$ git merge Feature0
```

Comando Git: git merge: 3-way merge



Comando Git: git merge: 3-way o fast-forward merge

Para pequeñas características y corrección de errores (hotfix) muchos programadores prefieren **utilizar fast-forward** con la ayuda git rebase.

Mientras que **3-way merge** lo utilizan para la integración de nuevas características con muchos commit. El commit que se genera sirve como commit simbólico de la unión de las dos ramas.

Comando Git: git merge: Resolviendo conflictos

Si las dos ramas que estamos tratando de fusionar cambiaron el mismo archivo, Git no será capaz de averiguar qué versión utilizar.

Cuando se produce una situación de este tipo, el proceso se detiene justo antes de la fusión de manera que se pueda resolver el conflicto de forma manual.

Para resolver el conflicto hemos de utilizar los comandos: git status/ editar archivos / git add / git commit

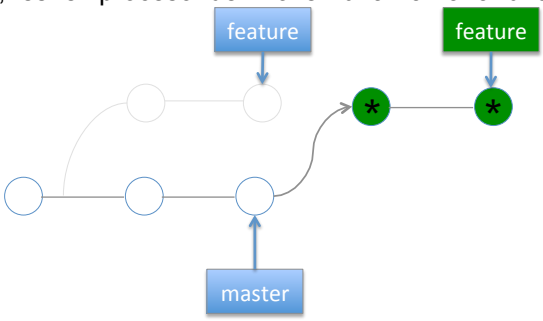
Cuando se encuentra un conflicto en la fusión, se ejecuta el comando git status que muestra en qué ficheros está el conflicto, se han de modificar los ficheros para resolver el conflicto, y posteriormente añadir al staging area y por último ejecutar el git commit.

Comando Git: git merge: Resolviendo conflictos

Tenga en cuenta que conflictos de fusión sólo se producirá en el caso de una fusión del tipo 3-way merge.

Reescribir la historia del repositorio: git rebase

Rebase, es el proceso de mover una rama a una nueva base del commit.



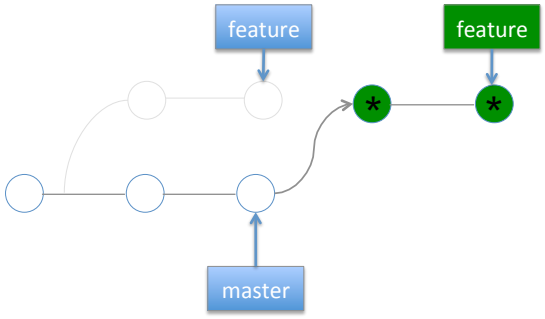
Lo que hace es mover una rama de un commit a otro. Git logra esto mediante la creación de nuevos commit's y su aplicación a la base. Esto es volver a escribir la historia del proyecto.

La rama tiene el mismo aspecto pero se compone de nuevos commit.

Reescribir la historia del repositorio: git rebase

```
git rebase <base>
```

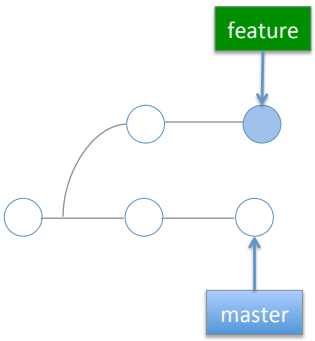
Rebase el actual branch (debemos estar posicionados en él) a <base> y <base> puede ser: cualquier tipo de referencia a un commit por ejemplo: ID commit; rama; tag (etiqueta); una referencia relativa a HEAD.



Reescribir la historia del repositorio: git rebase

La razón principal de cambio de base es mantener un historial de proyectos lineal.

La rama master ha progresado desde que se comenzó a trabaja con la nueva feature.



Tenemos dos alternativas:

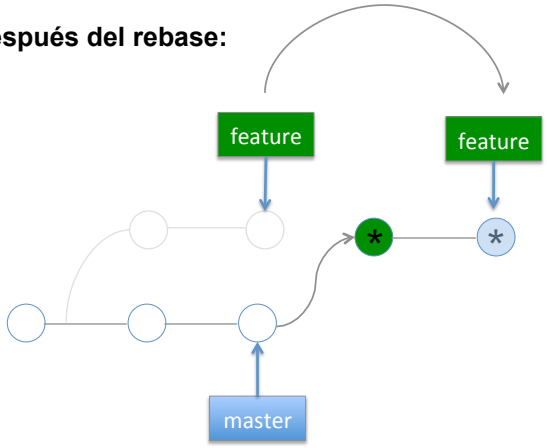
Hacer un merge : 3-way merge.

Hacer un rebase y merge: Genera un fast-forward merge y linealiza la historia de los commit's.

Reescribir la historia del repositorio: git rebase

Hacer rebase con master + merge genera una historia lineal de nuestro repositorio utilizando como algoritmo de fusión: fast-forward.

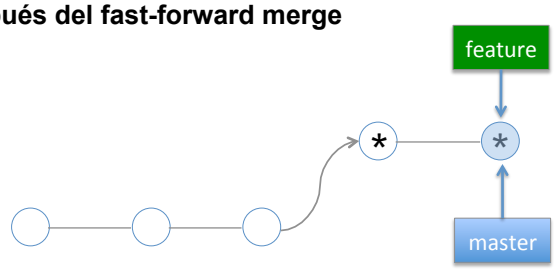
Después del rebase:



Reescribir la historia del repositorio: git rebase

Hacer rebase con master + merge genera una historia lineal de nuestro repositorio utilizando como algoritmo de fusión: fast-forward.

Después del fast-forward merge



Es una forma común para integrar los cambios ascendentes en el repositorio local.
Utilizar git merge genera commit superfluos cada vez que deseamos ver como ha progresado el proyecto.
"Quiero basar mis cambios en lo que todo el mundo ya ha hecho."

Reescribir la historia del repositorio: git rebase i

Al ejecutar el comando `git rebase -i` con el flag –i comienza un sesión interactiva

En lugar de mover ciegamente todas las confirmaciones a la nueva base, la sesión interactiva nos da la oportunidad de alterar commit individuales en el proceso.

Esto permite limpiar la historia mediante la eliminación, la división, y la modificación de una serie confirmaciones existentes.

Reescribir la historia del repositorio: git rebase i

Uso:

```
$ git rebase –i <base>
```

Se realiza el rebase sobre la rama actual. Este comando abre un editor donde se podrá decidir, sobre cada de uno de los commit, que hacer.

Reescribir la historia del repositorio: git rebase i

Este comando da un control completo sobre la historia del proyecto.

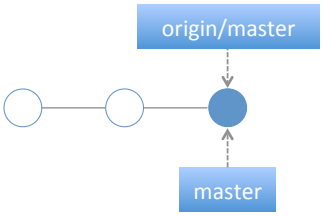
El desarrollador está concentrado en el desarrollo y no en la historia.

Muchos desarrolladores utilizan un rebase interactivo:

Para pulir una **rama** antes de la fusión con el código principal (master).

- Eliminar commit insignificantes.
- Eliminar commit obsoletos.

Uso de BRANCHES

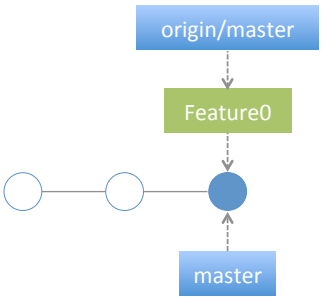


Creamos una nueva rama a la que llamaremos Feature0:

```
$ git branch Feature0
$ git checkout Feature0


$ git checkout –b Feature0
```

Uso de BRANCHES



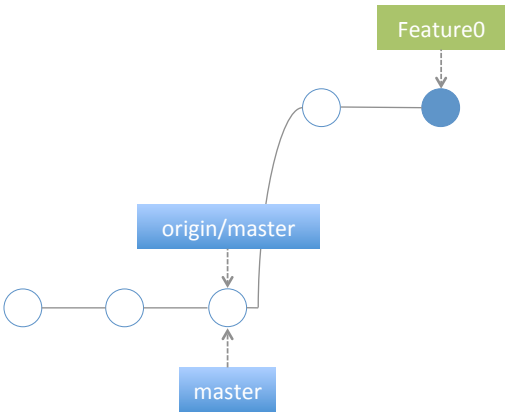
Añadimos un fichero que este en el directorio de trabajo.
Hacemos un commit

```
git commit –m “ADD nuevo fichero a Feature0”
```

Modificamos un fichero existen y hacemos un commit

```
$ git commit –m “UPDATE fichero.txt”
```

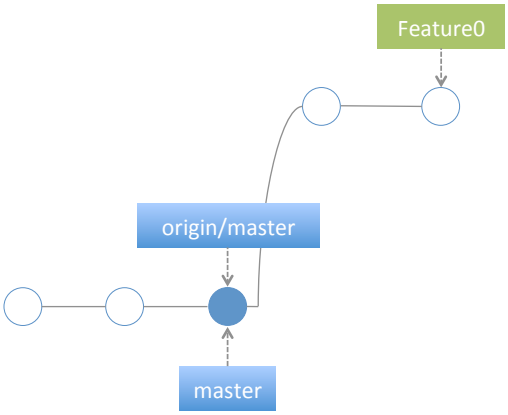
Uso de BRANCHES



Hay una incidencia en producción que tenemos que atender. Hemos de parar el desarrollo de Feature0 y nos ponemos a resolver la incidencia en producción. Para ello volvemos a master.

```
$ git checkout master
```

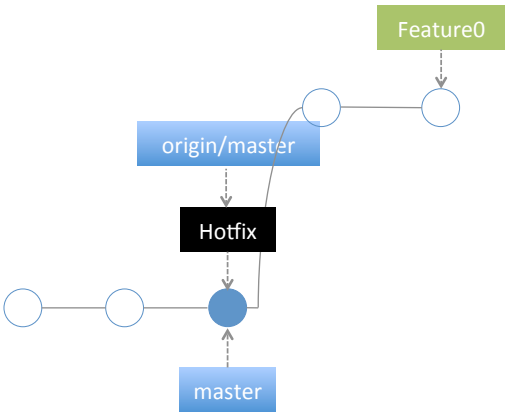
Uso de BRANCHES



Creamos una rama nueva en master que se llama hotfix y cambiamos a ella:

```
$ git checkout –b Hotfix
```

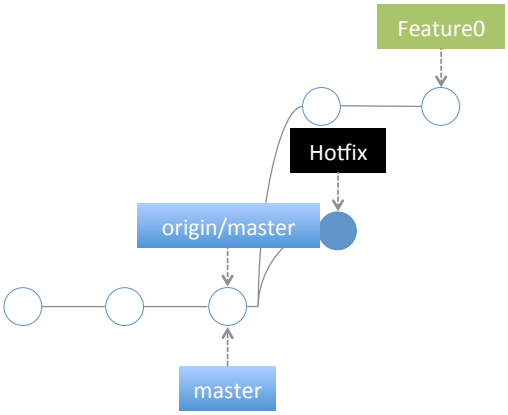
Uso de BRANCHES



Modifico le/los ficheros que generan la incidencia.
Realizo un commit de los cambios realizados.

```
$ git commit –m “fixed error en el split de email”
```

Uso de BRANCHES



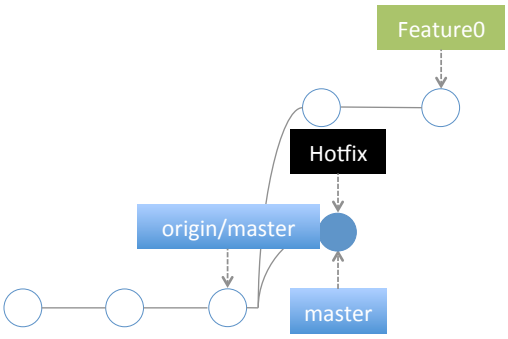
Verificamos que la corrección soluciona el Hotfix y hemos de incorporar los cambios a la rama master para ponerlo en producción. Para ello utilizamos el comando merge:
Nos situamos en la rama master:

```
$ git checkout master
```

Ejecutamos el merge

```
$ git merge Hotfix // fast-forward.
```

Uso de BRANCHES



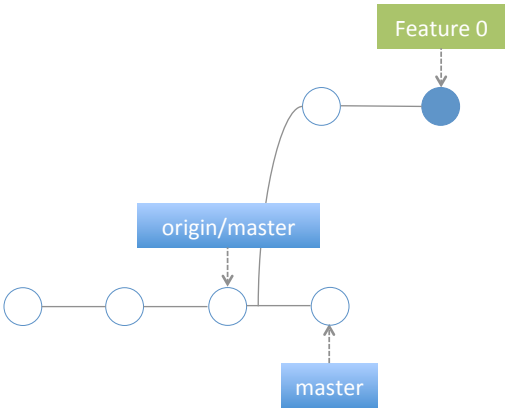
Ahora podemos seguir trabajando en Feature0, pero antes hemos de borrar la rama Hotfix.

```
$ git branch –d Hotfix
```

Cambiamos a la rama Feature0 para seguir trabajando

```
$ git checkout Feature0
```

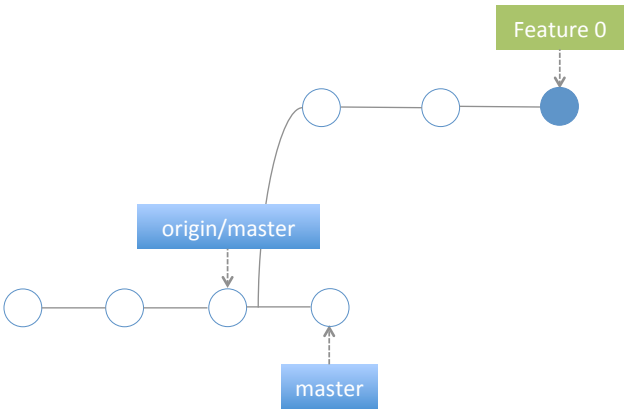
Uso de BRANCHES



Añadimos un nuevo fichero y hacemos un commit.

```
$ git commit –m “ADD newFichero.txt incluye nuevo objeto”
```

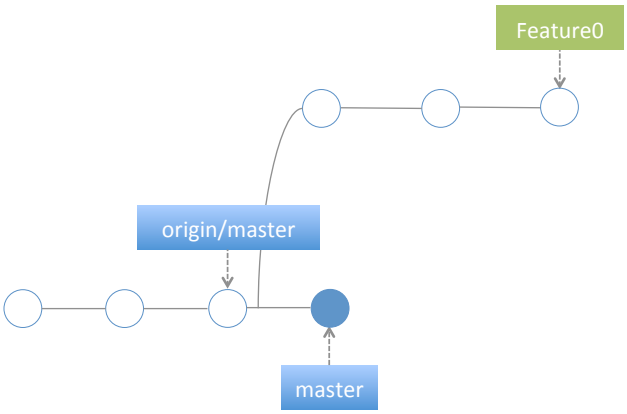
Uso de BRANCHES



Nos piden que dejemos lo que estamos haciendo y que desarrollemos una nueva funcionalidad Feature1 que debe de estar en producción antes del próximo lunes. Nos situamos en master.

```
$ git checkout master
```

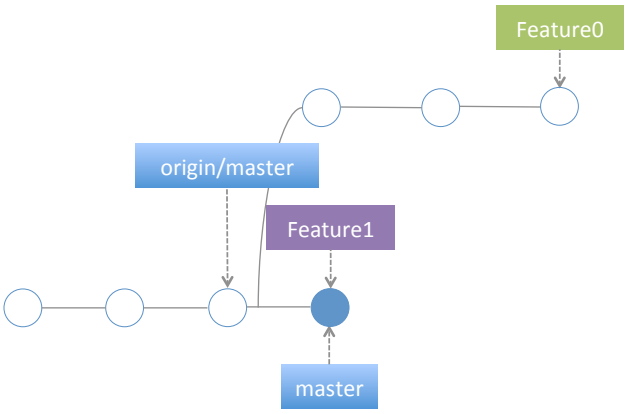
Uso de BRANCHES



Creamos una nueva rama Feature1.

```
$ git checkout –b Feature1
```

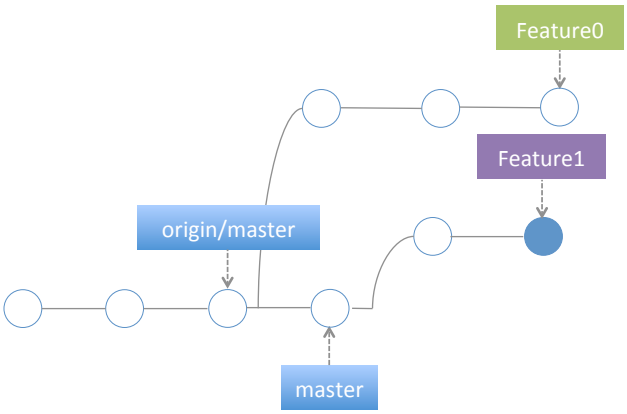
Uso de BRANCHES



Implementamos la nueva funcionalidad, haciendo un par de commit.

```
$ git Commit –m “UPDATE fichero.txt”
$ git Commit –m “ADD nuevo fichero.txt”
```

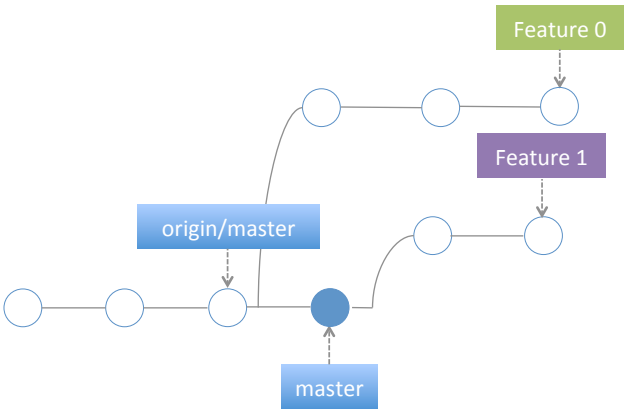
Uso de BRANCHES



Se decide liberar las dos Features. Para ello primero fusionamos Feature 1 con master. Podemos hacer un fast-forward ya que todo está en la misma rama hacemos un merge para ello primero nos situamos en la rama master:

```
$ git checkout master
```

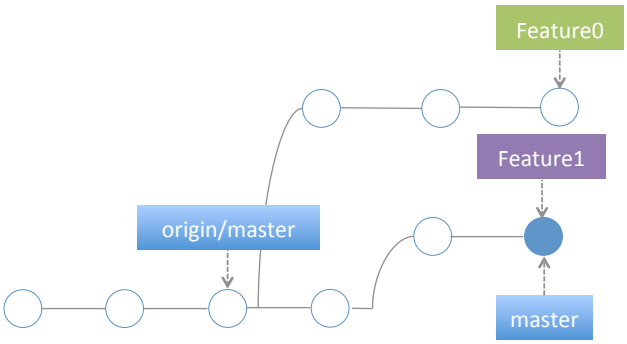
Uso de BRANCHES



Ejecutamos merge Feature1

```
$ git merge Feature1
```

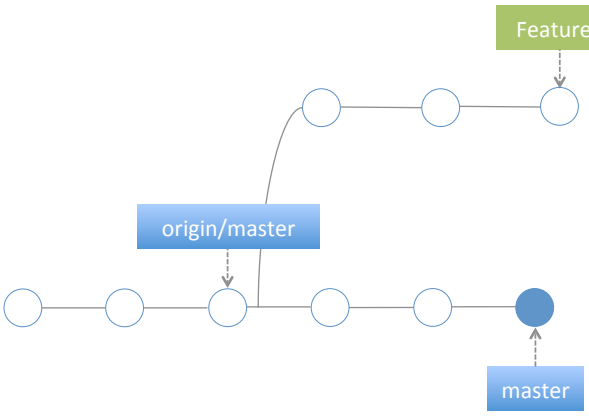
Uso de BRANCHES



Ahora podemos eliminar la rama Feature1, para ello ejecutamos

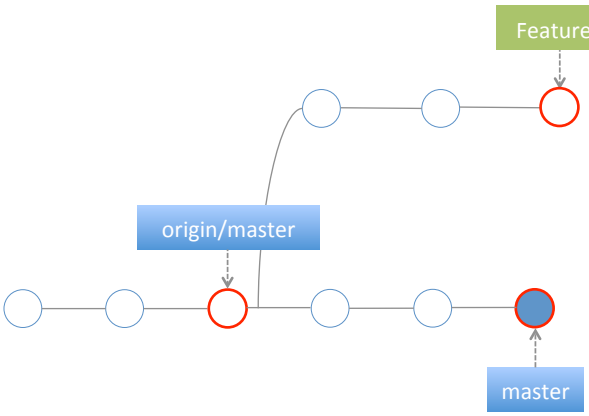
```
$ git branch –d Feature1
```

Uso de BRANCHES



Ahora nos queda fusionar Feature0 con master para luego liberar la versión del software. No podremos hacer merge fast-forward, sino hemos de mezclar mediante el algoritmo 3-way merge.

Uso de BRANCHES



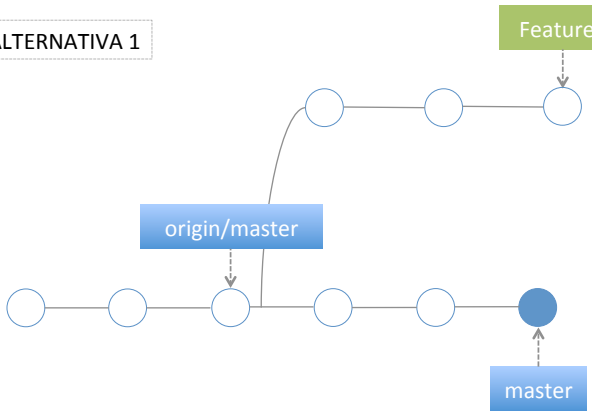
Tenemos dos alternativas:

Alternativa 1: $ git rebase + git merge

Alternativa 2: $ git merge.

Uso de BRANCHES

ALTERNATIVA 1

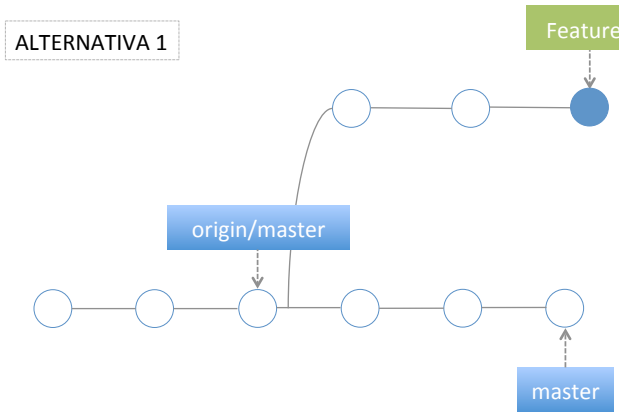


Nos situamos en Feature0, para ello ejecutamos:

```
$ git checkout Feature0
```

Uso de BRANCHES

ALTERNATIVA 1

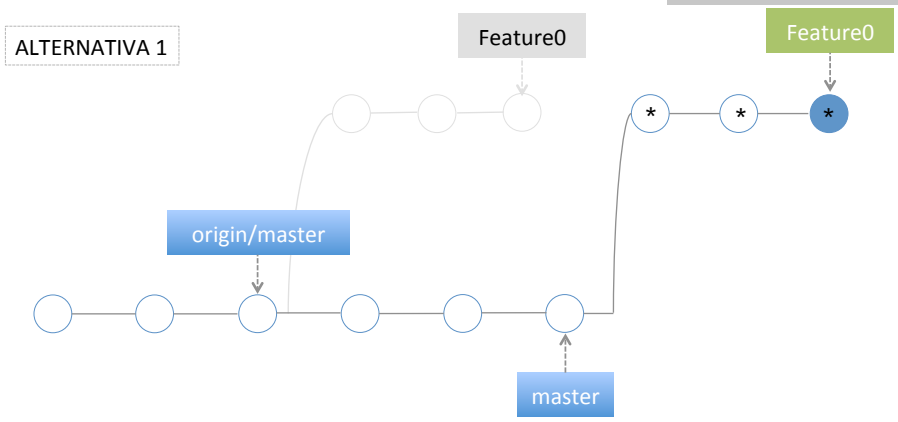


A continuación re-escribimos la historio con el rebase master:

```
$ git rebase master
```

Uso de BRANCHES

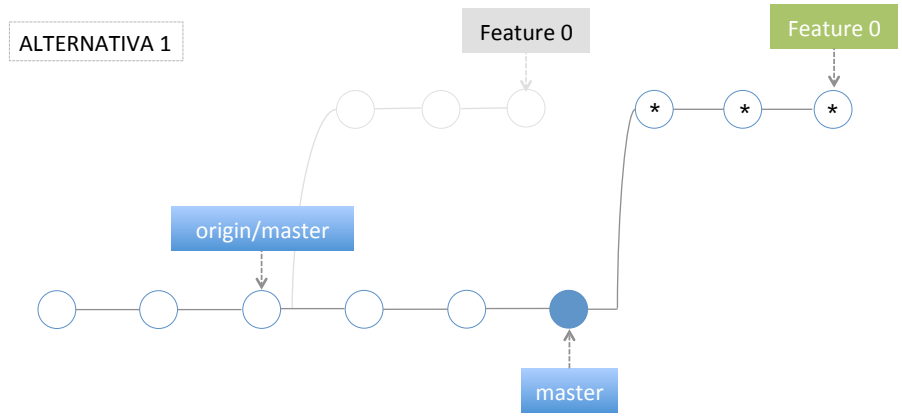
ALTERNATIVA 1



Ahora es lineal la secuencia de commit en la rama master y podemos hacer un merge fast-forward, para ello cambiamos a la rama master:

```
$ git checkout master
```

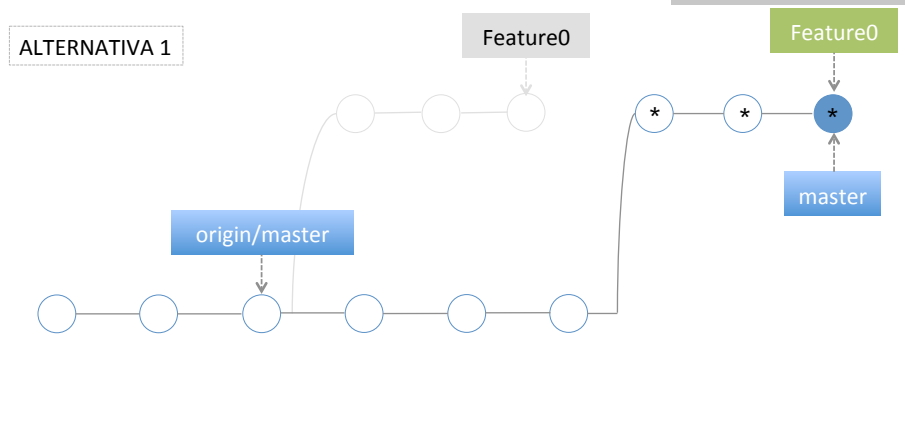
Uso de BRANCHES



Ahora podemos hacer el merge [fast-forward]

```
$ git merge Feature0
```

Uso de BRANCHES

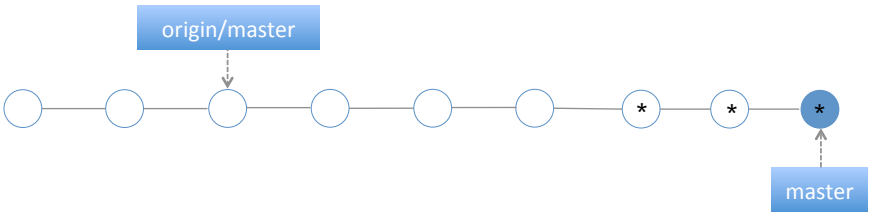


Borramos el branch Feature0

```
$ git branch –d Feature0
```

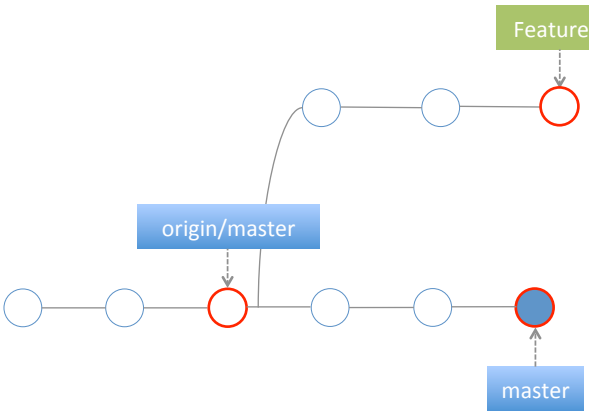
Uso de BRANCHES

ALTERNATIVA 1



Y ya tenemos nuestra historia toda integrada en nuestra rama master en el repositorio local.

Uso de BRANCHES



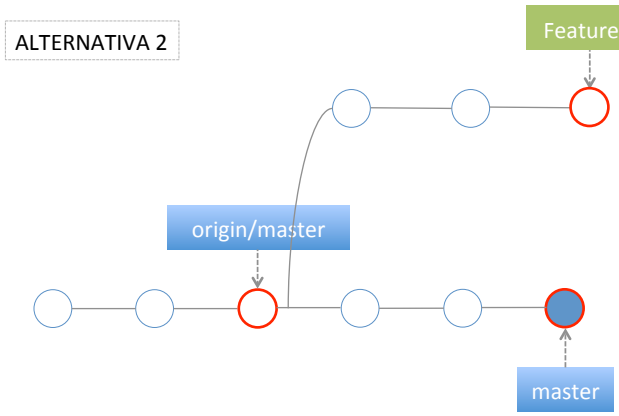
Tenemos dos alternativas:

Alternativa 1: $ git rebase + git merge

Alternativa 2: $ git merge.

Uso de BRANCHES

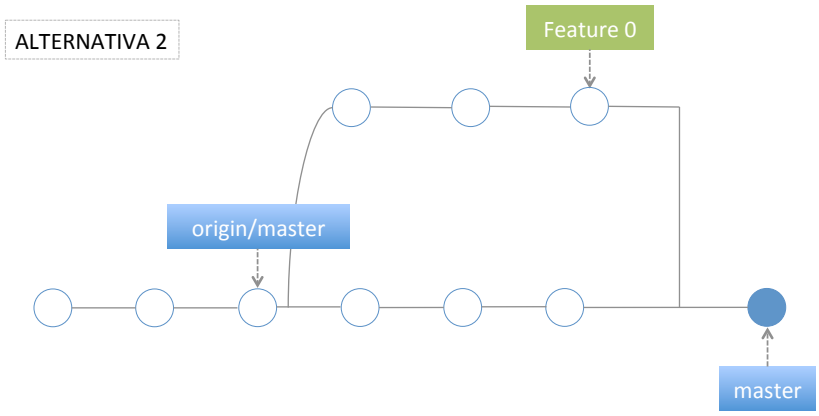
ALTERNATIVA 2



Esta alternativa es realizar un merge [3 way-merge], y para ello ejecutamos:
$ git merge Feature0

Uso de BRANCHES

ALTERNATIVA 2

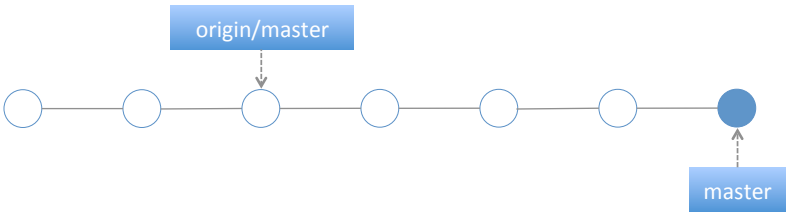


Se crea un nuevo shapshot resultante de la fusión a tres bandas de los commit, En está fusión el nuevo commit tiene más de un padre. Es git quien determina el mejor padre para hacer la fusión. Ahora podemos eliminar la rama (branch) Feature0 ejecutando:

```
$git branch –d Feature0
```

Uso de BRANCHES

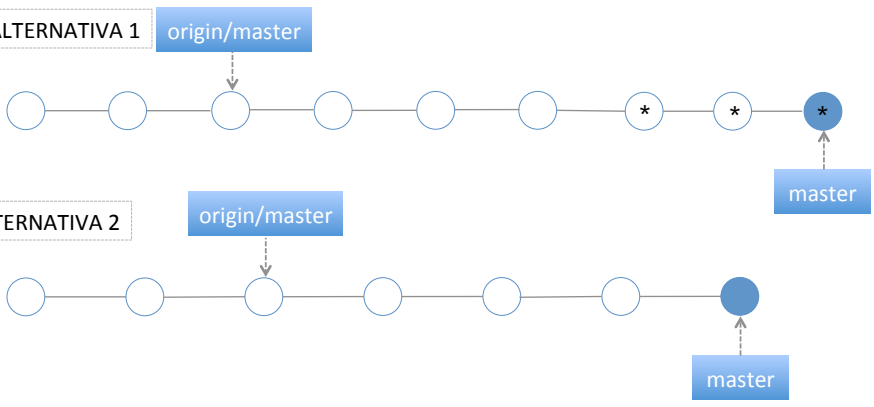
ALTERNATIVA 2



En este caso tenemos nuestra rama master lista para poner en producción.

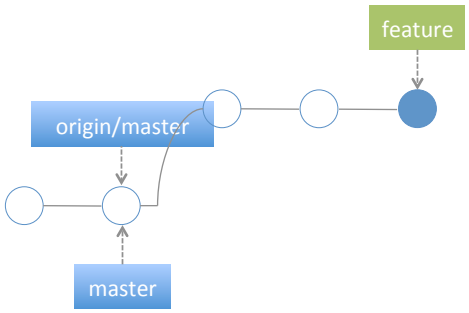
Uso de BRANCHES

ALTERNATIVA 1



Para liberar la versión del software queda realizar un “push” al repositorio del proyecto. Para que otros miembros del equipo puedan hacer un “pull” de la rama master del proyecto.

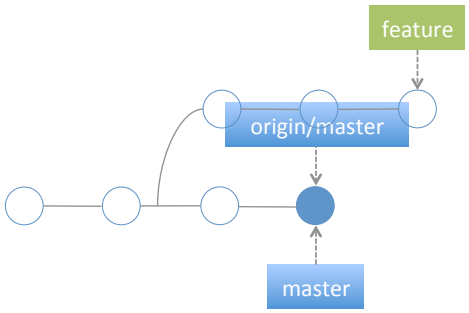
Merging versus Rebase




```
$ git checkout master
$ git pull origin
```

Merging versus Rebase

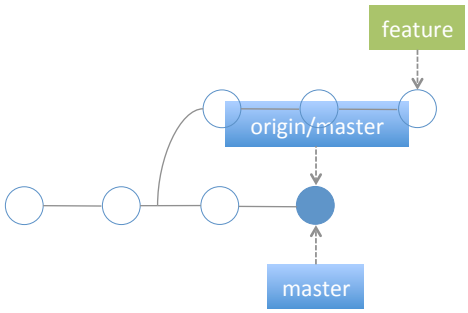
Los cambios en la historia de master son relevantes para nuestro desarrollo. Hemos de incorporar los commit en master a nuestra característica. **Tenemos dos opciones merging o rebasing**.




```
$
```

Merging versus Rebase

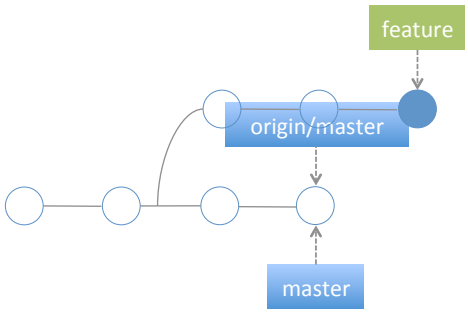
MERGE




```
$ git checkout feature
```

Merging versus Rebase

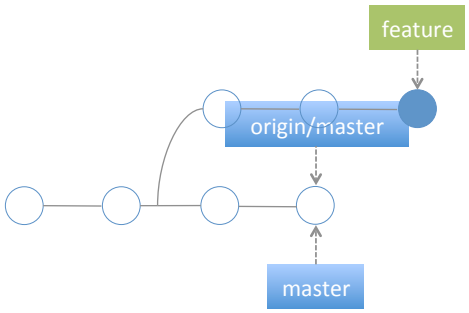
MERGE




```
$ git checkout feature
```

Merging versus Rebase

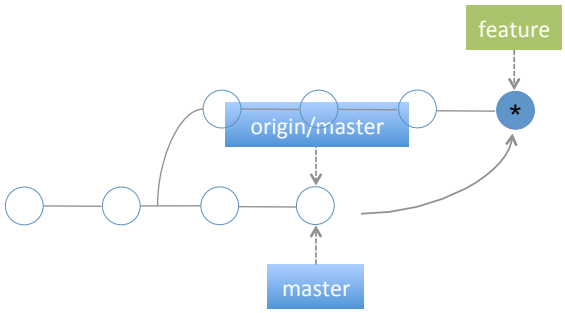
MERGE




```
$ git merge master
```

Merging versus Rebase

MERGE

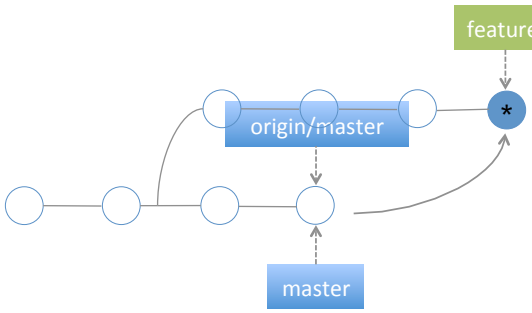



```
$ git merge master
```

Merging versus Rebase

MERGE

- No es una operación destructiva.
- Las existentes no cambian.
- Evita potenciales errores por el cambio de base.

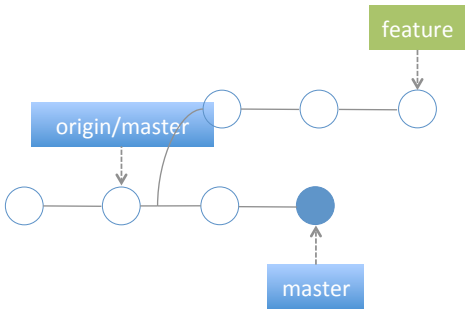


- Nuestra rama feature tendrá extraños commit cada vez que actualicemos
- Si master es muy activa podría contaminar la rama feature.
- Podría ser duro a terceras personas seguir la historia de feature.

```
$
```

Merging versus Rebase

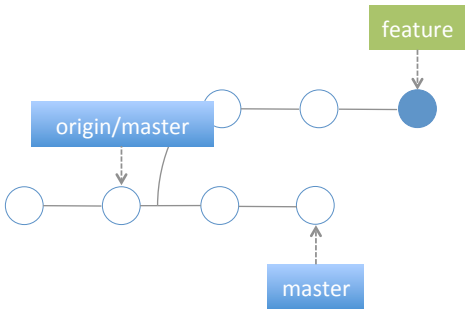
REBASE




```
$ git checkout feature
```

Merging versus Rebase

REBASE

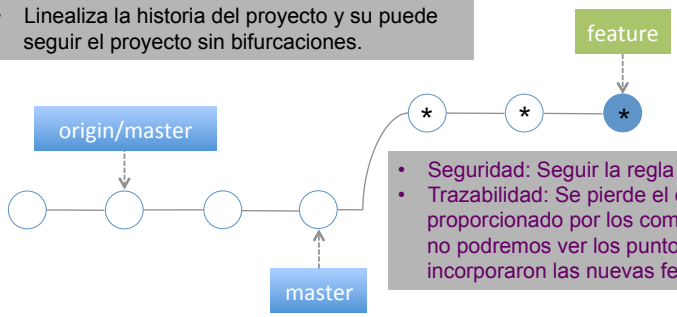



```
$ git rebase master
```

Merging versus Rebase

REBASE

- Se obtiene una historia del proyecto más limpia.
- Elimina commit de fusiones adicionales.
- Linealiza la historia del proyecto y su puede seguir el proyecto sin bifurcaciones.

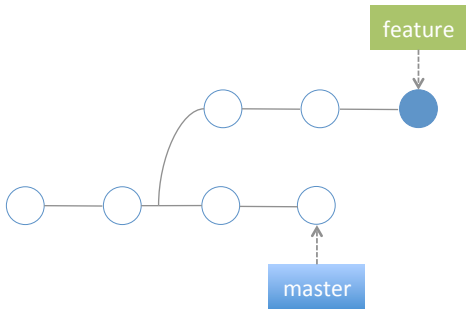


- Seguridad: Seguir la regla de oro del rebasing
- Trazabilidad: Se pierde el contexto proporcionado por los commit de los merge, no podremos ver los puntos donde se incorporaron las nuevas feature.

```
$
```

Merging versus Rebase

Los cambios en la historio de master son relevantes para nuestro desarrollo. Hemos de incorporar los commit en master a nuestra característica. **Tenemos dos opciones merging o rebasing**.

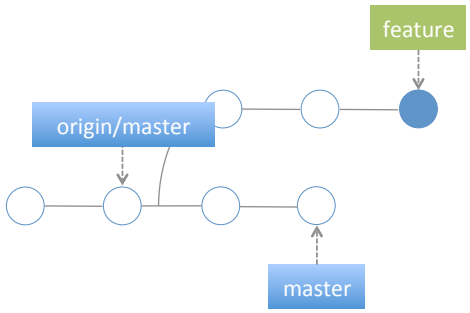



```
$
```

Comando rebase: La Regla de Oro

Lo más importan de saber con respecto a git rebase es tener claro cuando no utilizar este comando.

La regla de oro: Nunca utilizar este comando en ramas públicas.

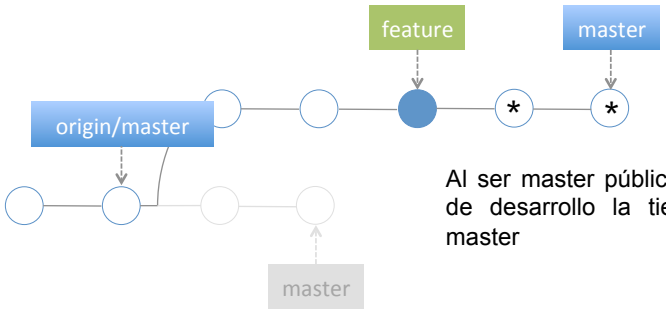



```
$ git rebase master
```

Comando rebase: La Regla de Oro

Lo más importan de saber con respecto a git rebase es tener claro cuando no utilizar este comando.

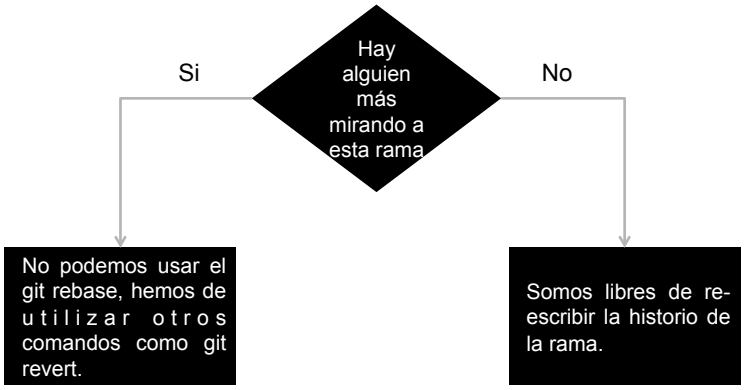
La regla de oro: Nunca utilizar este comando en ramas públicas.



Al ser master público todo el equipo de desarrollo la tiene como rama master

Comando rebase: La Regla de Oro

Antes de usar el comando git rebase hemos de hacernos la siguiente pregunta:

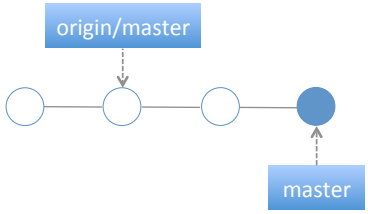


Comando Git: git push

Transfiere los commit del repositorio local al repositorio remoto.

Este comando tiene el potencial de sobre-escribir los cambios.

El caso de uso más común es publicar sus cambios locales a un repositorio central.



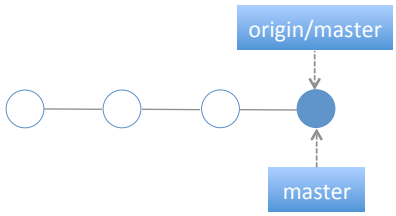

```
$ git push origin master
```

Comando Git: git push

Transfiere los commit del repositorio local al repositorio remoto.

Este comando tiene el potencial de sobre escribir los cambios.

El caso de uso más común es publicar sus cambios locales a un repositorio central.



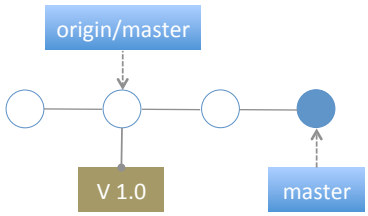
Comando Git: git push. Usos

COMANDO	DESCRIPCIÓN
$ git push <remote> <branch>	Trasfiere la rama especificada al repositorio remoto, junto con todas las confirmaciones necesarias y los objetos internos. Crea una rama local en el repositorio remoto. Git no permitirá la transferencia al directorio remoto cuando el resultado no sea fast-forward.
$ git push <remote> --force	Fuerza la trasferencia del repositorio aunque sea no-fast-forward. No usar --force si no estamos absolutamente seguros de lo que estamos haciendo.
$ git push <remote> --all	Trasfiere todas las ramas locales a repositorio remoto.
$ git push <remote> --tags	Los tags [etiquetas] no son trasferidos con la opción --all, por lo que hay que hacerlo expresamente.

Comando Git: git tag

Git tiene la funcionalidad de **etiquetar (tag)** puntos específicos en la historia de un proyecto como relevantes.

Normalmente se usa esta funcionalidad para marcar puntos donde se ha liberado alguna versión (v1.0, y así sucesivamente).



Comando Git: git tag

Tipos de tag:

Ligeros: Parecidas a ramas que no cambian. Es un puntero a un commit específico.

Anotados: son almacenadas como objetos completos en la base de datos de Git:

- Tienen hash [SHA-1]
- Contienen el nombre del etiquetador
- Correo electrónico
- Fecha
- Tienen mensaje de etiquetado

Comando Git: git tag

COMANDO	DESCRIPCIÓN
`$ git tag`	Listar las etiquetas disponibles en orden alfabético.
`$ git tag –l "texto"`	Buscar etiquetas de acuerdo a un patrón en particular.
`$ git tag <etiqueta> -a –m "texto"`	Crear una etiqueta anotada, equivalente git commit
`$ git tag <etiqueta> -s –m "texto"`	Crear una etiqueta anotada, equivalente git commit, firmada digitalmente. Requisitos gpg, generar claves, config, …
`$ git show <etiqueta>`	Visualiza la información de la etiqueta junto con la información del commit.
`$ git tag <etiqueta>`	Crea una etiqueta ligera.

Comando Git: git tag

Recordemos que por defecto, el comando `git push`, no transfiere etiquetas a los repositorios remotos. Hay que enviarlas explícitamente a un servidor compartido después de haberlas creado.

```
# comparte una sola etiqueta
$ git push origin <etiqueta>


# si queremos compartir muchas etiqueta a la vez
$ git push origin --tags
```

Cuando clonamos se descargan los **tags**.

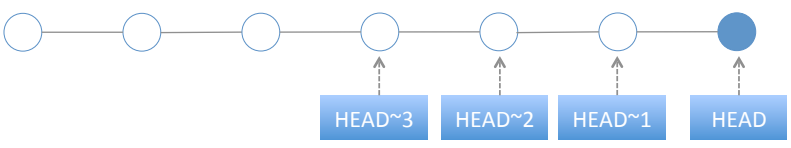
Referencias Relativas

Podemos hacer referencias a commit relativas a otros commit. El carácter ~ [virgulilla] nos permite llegar al commit padre.

Por ejemplo

```
$ git HEAD~2
```

Si nuestra historia es lineal cada nodo tiene un único padre y es fácil navegar en los ancestros de una referencia como HEAD.

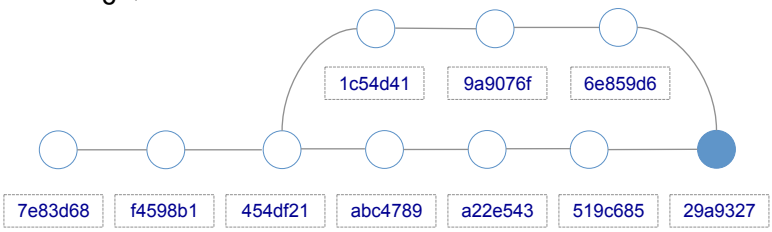


Referencias Relativas

Sin embargo cuando tenemos merge de ramas como en este caso el commit HEAD tiene dos padres.

El commit **29a9327** tiene dos padres: **519c685** y **6e859d6**

HEAD~1 ¿Qué commit es?



3-way merge: el primer padre es de la rama en la que estaba situado cuando se realiza el `git merge`, y el segundo padre es de la rama que pasa al comando `git merge` (con la que se fusiona).

Referencias Relativas

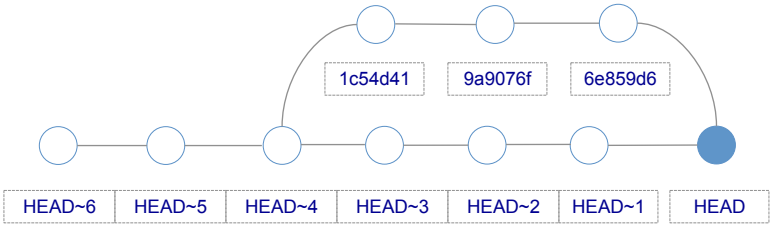
El carácter ~ [virgulilla] siempre se refiere al primer padre del commit, Si queremos referenciar al segundo padre de un commit tenemos que especificarlo con el carácter ^ (acento circunflejo).

Si queremos posicionarnos en el segundo padre de HEAD el comando que debemos ejecutar el HEAD^2.

Podemos utilizar más de un carácter ^ para movernos más de una generación.

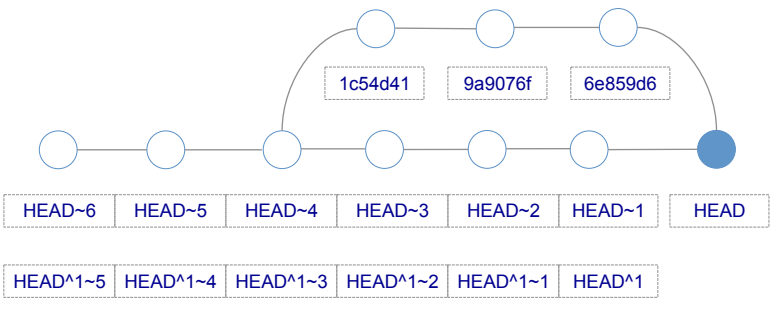
Referencias Relativas

En algunos casos podemos utilizar múltiples referencias para referenciar a commit específico.



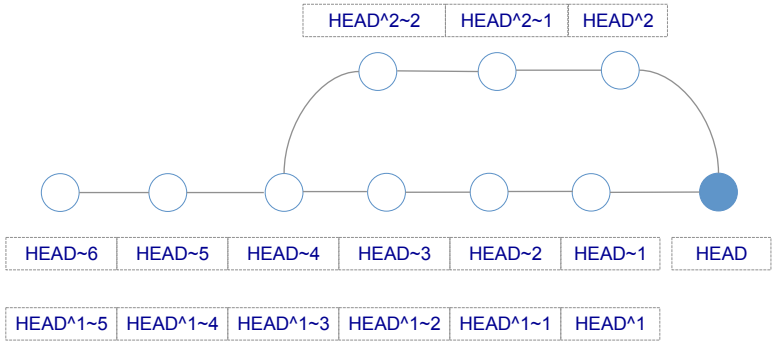
Referencias Relativas

En algunos casos podemos utilizar múltiples referencias para referenciar a commit específico.



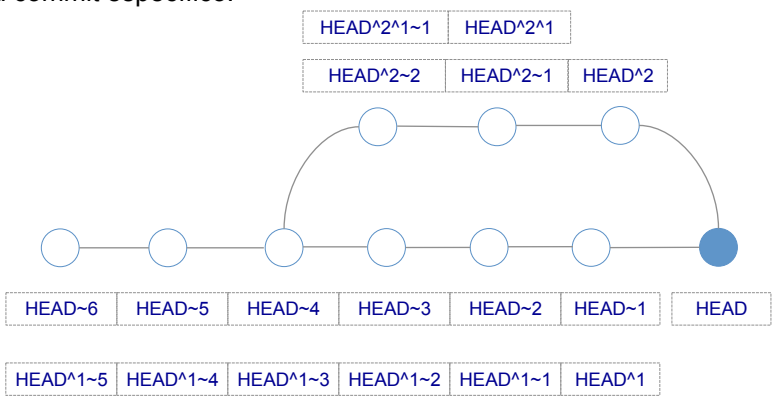
Referencias Relativas

En algunos casos podemos utilizar múltiples referencias para referenciar a commit específico.



Referencias Relativas

En algunos casos podemos utilizar múltiples referencias para referenciar a commit específico.



Excluir ficheros del repositorio. gitignore

En cualquier proyecto, existen ficheros que no deben incluirse en el repositorio. Por nombrar algunos de ellos:

- Ficheros de log
- Cachés
- Ficheros que contienen claves privadas (App keys y App secrets) para acceder a las API o servicios web
- Ficheros de configuración con contraseñas de bases de datos
- Ficheros binarios compilados
- Ficheros de configuración del entorno de desarrollo

Excluir ficheros del repositorio. gitignore

- Hay ciertos comandos, como por ejemplo git ls-files o git read-tree, permiten excluir ficheros a través de argumentos. Estos patrones tendrán la máxima prioridad.
- Crear un fichero .gitignore en el repositorio. Este fichero hay que subirlo al repositorio.
- Crear un fichero .git/info/exclude para poner las reglas locales.
- En el fichero definido, con el comando git config --global core.excludesfiles

> Todos los ficheros mencionados arriba incluyen en su interior un conjunto de patrones de ficheros que se van a ignorar. Cada patrón va en una línea.

Excluir ficheros del repositorio. gitignore

Se pueden tener cumplimentados tres archivos con patrones a excluir. Cuál es el orden de preferencia de aplicación. ¿Qué sucede si hay patrones contradictorios? ¿Cómo se resuelven los conflictos?

Excluir ficheros del repositorio. gitignore

El último archivo de ellos que contenga un patrón que explícitamente ignore o no un fichero dado, es el que aplica.

¿Qué pasa si dentro de un mismo fichero existen reglas contradictorias? Imagínate un fichero .gitignore como el siguiente:

```
*.exe
*.png
*.jpg
!*.exe
```

Igual, se aplica **la última regla**.

Excluir ficheros del repositorio. gitignore

¿Qué patrones hemos de incluir en cada archivo?

.gitignore	Pondremos aquellos patrones que queremos que sean iguales en todos los clones que existen del repositorio. Todos los miembros del equipo los comparten. Es por ello que este fichero debe incluirse en el repositorio. Este es un buen lugar para ignorar ficheros de log y cache, configuraciones, claves privadas, etc.
El fichero exclude	Se incluyen patrones que se van a aplicar sólo a nuestra copia local y no se van a distribuir al resto de miembros del equipo. Este fichero es útil cuando por ejemplo, creas una rama local para desarrollar una determinada característica de la aplicación, y necesitas ignorar ciertos ficheros sin que el resto del equipo se vea afectado por ello.
El fichero indicado por core.excludesfile	Incluimos aquellos patrones que queremos ignorar de forma global en todos los repositorios. Este es un lugar para incluir los ficheros del proyecto generados por nuestro entorno de desarrollo.

Excluir ficheros del repositorio. gitignore

Los patrones y reglas que git utiliza para interpretar los archivos .gitignore.

Comentarios y líneas en blanco

- Cualquier línea que comience por # es un comentario
- Una línea en blanco no se considera un patrón por parte de git, por lo que puede ser usada como un separador

Negación

Utilizando el carácter *!* podemos invertir un patrón
Si ponemos:
!*.exe # queremos decir que se incluyan los archivos que acaba en *.exe

Ignorando carpetas enteras

Si un patrón acaba con una barra / git ignorará cualquier fichero dentro de esa carpeta o en cualquier subcarpeta que cuelgue de ellas.

Excluir ficheros del repositorio. gitignore

Los patrones y reglas que git utiliza para interpretar los archivos .gitignore.

- Si el patrón no termina "/", git lo interpreta como un **shell glob pattern**.
  - En una expresión que contenga el símbolo "?", éste se corresponde con uno y sólo un carácter.
  - En una expresión que contenga el símbolo " * ", éste se corresponde con cualquier número de caracteres.
  - En una expresión que contenga corchetes [ ], éstos se corresponden con uno y sólo un carácter de los que están incluidos en los corchetes.
  - Si se necesita utilizar los caracteres ? * [ ] en un patrón, estos deberán escaparse con \
- Una "/" al principio del patrón indica el inicio de una ruta relativa a la carpeta en la que se encuentra el archivo .gitignore

Buenas Prácticas con git

1. Haz muchos commit

   Cada vez que hagas un cambio, por pequeño que sea, si altera la funcionalidad de tu código haz un *commit*. No esperes a tener varias modificaciones no relacionadas entre si para hacer un *commit*.

2. Comenta bien los commit

   Los *commit's* descriptivos hacen la revisión de tu código un trabajo mucho más fácil.

3. No hacer commit de desarrollos a medias.

   Haz un *commit* cuando el trabajo esté listo, esto no significa que solo puedas realizar un commit cuando la página, función, script, etc., esté lista, si no que es mejor dividir el trabajo en partes lógicas e ir completándolo y realizando *commit's* de cada parte cuando se haya terminado.

Buenas Prácticas con git

4. Hacer ramas en git es bueno.
   Si estás trabajando en equipo o desarrollas múltiples funcionalidades a la vez, cada una de estas debe tener una rama [*branch]* separada, de forma que los cambios creados por algunos no sean modificados por otro que está trabajando en algo distinto.

5. Estandarizar.
   Cuéntale a tu grupo de trabajo sobre como usas Git, lleguen a un acuerdo y estandaricen el trabajo, para que así sea realmente útil para todos trabajar con Git.

Buenas Prácticas. Mensajes del Commit

Los mensajes deben de tener una estructura.

El mensaje de un commit consiste en 3 diferentes partes separadas por una línea en blanco:

- el titulo.
- un cuerpo opcional.
- y un pie opcional.

```
type: subject

body

footer
```

Buenas Prácticas. Mensajes del Commit. Tipo/Asunto

Tipo	Descripción
feat	Una nueva característica.
fix	Se soluciono un bug.
docs	Se realizaron cambios en la documentación.
style	Se aplico formato, comas y puntos faltantes, etc.; Sin cambios en el código.
refactor	Refactorización del código en producción.
test	Se añadieron pruebas, refactorización de pruebas; Sin cambios en el código.
chore	Actualización de tareas de build, configuración del admin. de paquetes; Sin cambios en el código.

El asunto no debe contener mas de 50 caracteres, debe iniciar con una letra mayúscula y no terminar con un punto.
Debemos ser imperativos al redactar el commit, es decir hay que ser objetivos y muy importante tenemos que acostumbrarnos a escribirlos en **Inglés.**

Buenas Prácticas. Mensajes del Commit. Cuerpo

No todos los commit's son lo suficientemente complejos como para necesitar de un cuerpo, por lo que, **es opcional** y se usan en caso de que el commit requiera una explicación y contexto.

Utilizamos el cuerpo para explicar el **Qué y Porqué** de un commit y nunca el **Cómo**

Al escribir el cuerpo, requerimos de una línea en blanco entre el titulo y el cuerpo, además debemos limitar la longitud de cada línea a no mas de 72 caracteres.

Buenas Prácticas. Mensajes del Commit. Pie

El pie es **opcional** al igual que el cuerpo, pero este es usado para el seguimiento de los ID- commit's con incidencias.

Clientes Git

Clientes GUI de Git : https://git-scm.com/downloads/guis