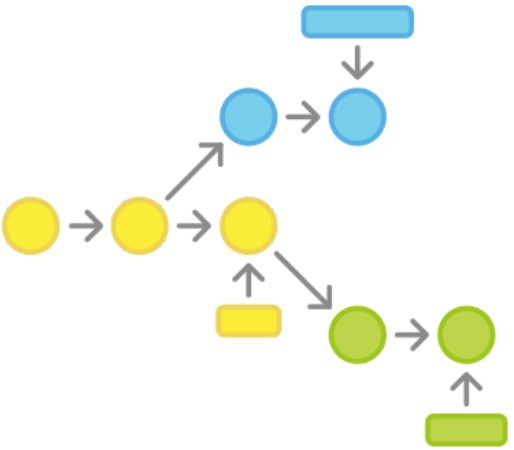
Git Tutorials

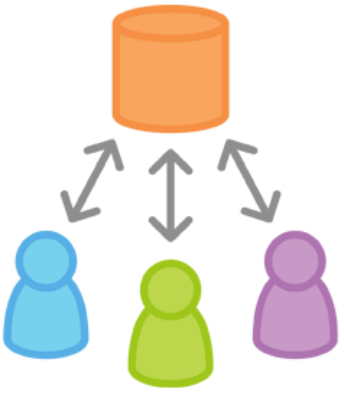
workflows Git.



La variedad de workflows posibles puede hacer que sea difícil saber por dónde empezar al usar Git en el espacio de trabajo. Esta página proporciona un punto de inicio con una visión global de los workflows más comunes para equipos de empresas.

Según vayas leyendo, recuerda que esos workflows están diseñados para ser directrices más que normas concretas. Queremos mostrarte lo que es posible, así que puedes mezclar y hacer coincidir aspectos de diferentes workflows para cumplir tus necesidades individuales.

workflow centralizado



Cambiar a un sistema de control de versiones distribuido puede parecer una tarea de gran envergadura, pero no tienes que cambiar tu workflow para aprovechar Git. Tu equipo puede desarrollar proyectos de la misma manera en que la que lo hace con Subversion.

Sin embargo, usar Git para mejorar el workflow del desarrollo presenta varias ventajas sobre SVN. Primero, da a cada desarrollador su propia copia local del proyecto entero. Este entorno aislado permite a cada desarrollador trabajar aislado de los cambios que haya en el proyecto—pueden añadir commits a su repositorio local y olvidarse completamente de los avances del proyectol hasta que sea conveniente para ellos.

Segundo, te da acceso al robusto modelo de crear ramas y fusiones de Git. Al contrario que en SVN, las ramas Git estás diseñadas para ser un mecanismo seguro para integrar código y compartir cambios entre repositorios.

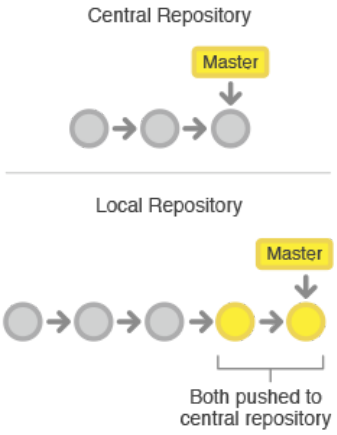
Cómo funciona

Igual que Subversión, el workflow de trabajo centralizado usa un repositorio central que sirve como punto de entrada único para todos lo cambios del proyecto. En vez de ser `trunk` , la rama de desarrollo predeterminado se llama `master` y todos los cambios se confirman en esta rama. Este workflow no necesita ninguna otra rama aparte de `master` .

Los desarrolladores empiezan clonando el repositorio central. En sus copias locales del proyecto, editan los archivos y confirman los cambios igual que harían con SVN; sin embargo, estos

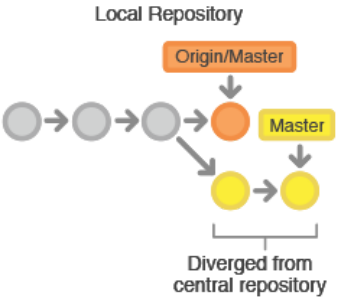
commits nuevos se almacenan localmente—están completamente aislados del repositorio central. Esto permite a los desarrolladores aplazar la sincronización con el repositorio central hasta el momento apropiado.

Para publicar cambios en el proyecto oficial, los desarrolladores "envían" su rama `master` local al repositorio central. Este es el equivalente de `svn commit`, excepto en que añade todos los commits locales que no están ya en la rama `master`.



Gestionar conflictos

El repositorio central representa el proyecto oficial, así que la historia de commit tiene que ser tratada como sagrada e inmutable. Si los commits locales de un desarrollador difieren de un repositorio central, Git denegará que se envíen los cambios porque se reescribirían los commits oficiales.



Antes de que el desarrollador pueda publicar esta funcionalidad, necesita recuperar (fetch) los commits de la versión central actualizados y reorganizar (rebase) sus cambios sobre ellos. Esto es como decir, "quiero añadir mis cambios a lo que los demás ya han hecho". El resultado es una historia perfectamente lineal, igual que un workflow SVN.

Si los cambios cambios locales entran directamente en conflicto con los commits del repositorio central, Git pausará la reorganización (rebase) y te dará la oportunidad de solucionar los conflictos de forma manual. Lo bueno de Git es que usa los mismos comandos `git status` y `git add` para generar commits y solucionar conflictos de fusión. Esto hace que sea fácil para los nuevos desarrolladores gestionar sus propias fusiones. Además, si tienen problemas, Git hace que sea muy fácil abortar la reorganización (rebase) entera e intentarlo de nuevo (o ir a buscar ayuda).

Ejemplo

Echemos un vistazo paso a paso sobre cómo colaboraría un equipo pequeño con este workflow. Veremos cómo dos desarroladores, John y Mary, pueden trabajar en funcionalidades diferentes y compartir sus aportaciones a través de un repositorio central.

Alguien inicializa el repositorio central

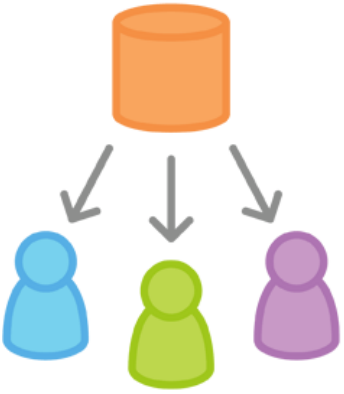
Primero alguien necesita crear el repositorio central en un servidor. Si es un proyecto nuevo, se puede inicializar en un repositorio vacío. Si no es así, será necesario importar un repositorio Git o SVN ya existente.

Los repositorios centrales siempre han de ser repositorios vacíos (no deben tener un directorio de trabajo), y se pueden crear de la siguiente forma:

```
ssh user@host
git init --bare /path/to/repo.git
```

Asegúrate de usar un nombre de usuario de SSH válido para usuer, el dominio o la dirección IP de tu servidor para el `host`, y la ubicación donde quieres almacenar tu reposición para `/path/to/repo.git`. Ten en cuenta que la extensión `.git` se añade normalmente al nombre del repositorio para indicar que es un repositorio básico.

Todo el mundo clona el repositorio central

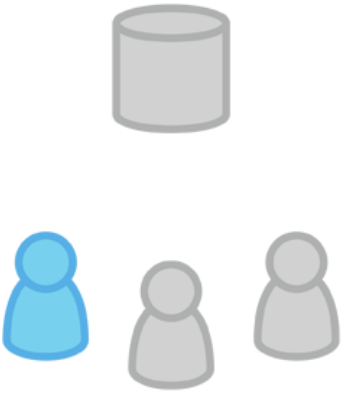


Luego, cada desarrollador crea una copia local del proyecto entero. Esto se consigue con el comando `git clone`:

```
git clone ssh://user@host/path/to/repo.git
```

Al clonar un repositorio, Git añade automáticamente un atajo con el nombre `origin` que apunta al repositorio “padre”, bajo el supuesto de que quieras interaccionar con él después.

John trabaja en su funcionalidad

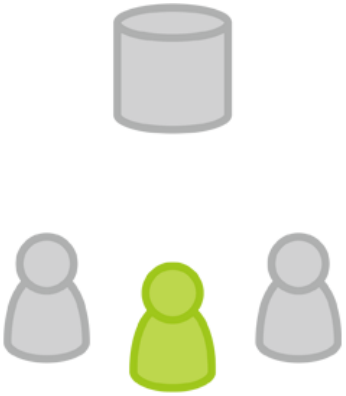


En su repositorio local, John puede desarrollar funcionalidades usando el proceso estándar Git commit: editar, preparar y confirmar. Si es estás familiarizado con el área de preparación, es una forma de preparar un commit sin tener que incluir cada cambio en el directorio de trabajo. Esto te permite crear commits altamente especializados, incluso si has hecho muchos cambios locales.

```
git status # Mira el estado de la reposición
git add # Prepara un archivo
git commit # Confirma un archivo
```

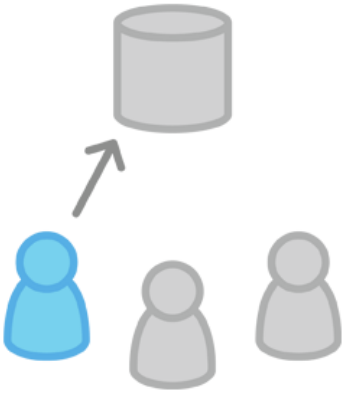
Recuerda que, ya que estos comandos crean commits locales, John puede repetir este proceso tantas veces como quiera sin preocuparse sobre qué pasa en el repositorio central. Esto puede ser muy útil para funcionalidades grandes que se tienen que dividir en partes más simples y atómicas.

Mary trabaja en su funcionalidad



Mientras tanto, Mary trabaja en su propia funcionalidad en su repositorio local usando el mismo proceso editar/preparar/confirmar. Igual que a John, a ella no le importa qué es lo que ocurre en el repositorio central, y a ella realmente no le preocupa qué es lo que está haciendo John en su repositorio local, ya que todos los repositorios son privados.

John publica su funcionalidad

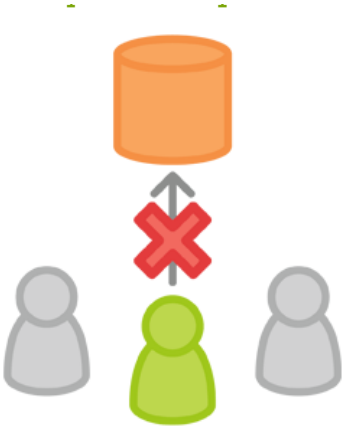


Cuando John su funcionalidad, tiene que publicar sus commits locales al repositorio central para que otros miembros puedan acceder. Puede hacerlo con el comando `git push` como:

```
git push origin master
```

Recuerda que `origin` es la conexión remota al repositorio central que Git ha creado cuando John lo clonó. El argumento `master` le dice a Git que haga la rama `master` de `origin` como su rama local `master`. Como el repositorio central no se ha actualizado desde que John lo clonó, esto no provocará ningún conflicto y el envío funcionará como debe.

Mary intenta publicar su funcionalidad



Veamos que pasa si Mary intenta envíar su funcionalidad después de que John haya publicado con éxito sus cambios al repositorio central. Ella puede usar el mismo comando para enviar (push):

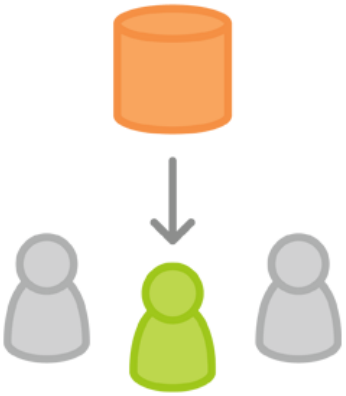
```
git push origin master
```

Pero ya que su historia local ha cambiado con respecto al repositorio central, Git rechazará la petición con un largo mensaje de error:

```
error: failed to push some refs to '/path/to/repo.git'
pista: Se han rechazado los cambios porque la punta de tu rama actual
está detrás
pista: su equivalente remoto. Fusiona los cambios remotos (p. ej. "git
pull")
pista: antes de enviar (push) de nuevo.
pista: ver la nota sobre avances rápidos en "git push --help" para más
detalles.
```

Esto evita que Mary sobrescriba commits oficiales. Ella ha de recibir las actualizaciones de John en su repositorio, integrarlas con sus cambios locales y entonces intentarlo de nuevo.

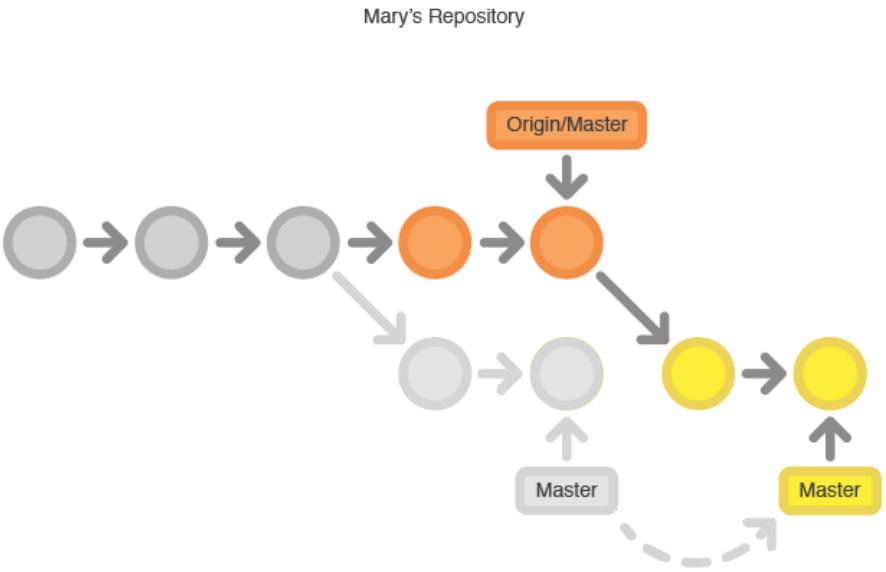
Mary reorganiza (rebase) sobre el commit o los commits de John



Puede usar `git pull` para incorporar los cambios en el repositorio central a su repositorio. Este comando es parecido a `svn update` —recibe la historia completa de commits del repositorio central al repositorio local de Mary e intenta integrarlo con sus commits locales:

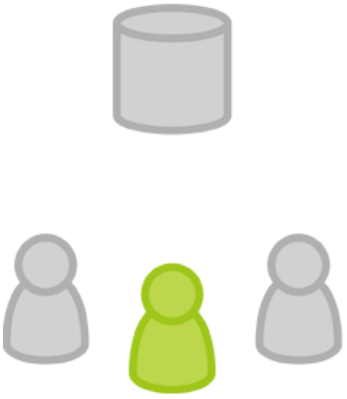
```
git pull --rebase origin master
```

La opción `--rebase` le dice a Git que mueva todos los commits de Mary a la punta de la rama `master` después de sincronizarla con los cambios del repositorio central, cómo se muestra a continuación:



Se podrá recibir (pull) incluso sin olvidas esta opción, pero podrías terminar con una "fusión confirmada" innecesaria cada vez que alguien necesite sincronizar con el repositorio central. Para este workflow, siempre es mejor reorganizar (rebase) en vez de generar una fusión confirmada.

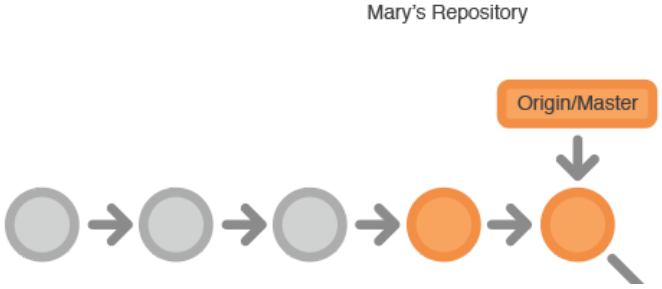
Mary soluciona el conflicto de fusión

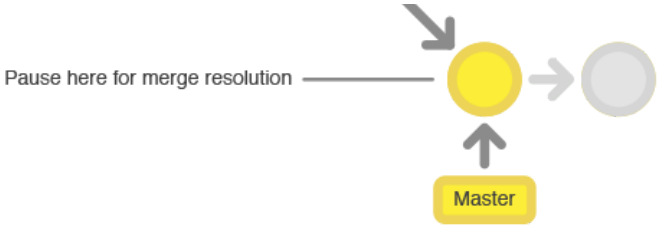


Reorganizar (rebase) trabajos transfiriendo cada commit local a la rama `master` que se ha subido de una vez. Esto significa que puedes atrapar los conflictos de fusión commit por commit, en vez de solucionarlos todos en una enorme fusión confirmada. Esto mantiene tus commits todo los especializados que es posible y hace que la historia del proyecto quede limpia. A su vez hace que sea mucho más fácil saber dónde están los bugs y, si fuera necesario, volver atrás en los cambios con un impacto mínimo en el proyecto.

Si Mary y John están trabajado en funcionalidades que no tienen relación, no es probable que el proceso de reorganización (rebase) genere conflictos. Pero si lo hiciera, Git pausaría la reorganización en el commit actual y mostraría el siguiente mensaje, junto con algunas instrucciones pertinentes:

```
CONFLICT (content): Merge conflict en "some-file"
```





Lo bueno de Git es que cualquiera puede solucionar sus propios conflictos de fusión. En nuestro ejemplo, Mary podría simplemente `git status` para ver dónde está el problema. Los archivos en conflicto aparecerán en la sección de rutas sin fusionar

```
# Rutas sin fusionar:
# (usa "git reset HEAD <some-file>..." para sacar del área de
preparación)
# (usa "git add/rm <some-file>..." de forma apropiada para marcar la
solución)
#
# ambos modifican: <some-file>>
```

Entonces ella puede editar el archivo como quiera. Una vez que esté contenta con el resultado, puede preparar los archivos de la forma habitual y dejar que `git rebase` haga el resto:

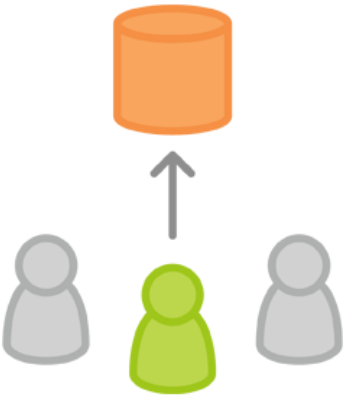
```
git add <some-file>
git rebase --continue "/some-file"
```

Y ese es todo. Git irá al siguiente commit y repetirá el proceso con cualquier otro commit que genere conflictos.

Si llegas a este punto y te das cuenta de que no tienes ni idea de lo que está pasando, no entres en pánico. Simplemente ejecuta el siguiente comando y volverás a a donde empezaste antes de habar ejecutado `git pull --rebase`:

```
git rebase --abort
```

Mary publica su funcionalidad con éxito



Después de haber acabado la sincronización con el repositorio central, podrá publicar los cambios con éxito:

```
git push origin master
```

A dónde ir desde aquí

Como ves, es posible copiar el entorno de desarrollo tradicional de Subversion usando solo unos cuantos comandos Git. Esto es genial para equipos que vienen de usar SVN, pero no aprovecha la naturaleza distribuida de Git.

Si tu equipo está contento con el workflow centralizado pero quiere racionalizar sus esfuerzos colaborativos, merece la pena explicar los beneficios del Feature Branch Workflow. Al dedicar una rama aislada a cada funcionalidad, es posible iniciar discusiones a fondo sobre nuevas sobre nuevos añadidos antes de integrarlos en el proyecto oficial.

PREVIOUS
Overview

NEXT
Feature Branch Workflow