Contenidos a Trabajar

1. Fundamentos de GIT
2. Estados en GIT
3. Instalación de GIT - Copy
4. Comandos básicos de GIT
5. Flujos de trabajo

Fundamentos de Git

Entonces, ¿qué es Git en pocas palabras? Es muy importante entender bien esta sección, porque si entiendes lo que es Git y los fundamentos de cómo funciona, probablemente te será mucho más fácil usar Git efectivamente. A medida que aprendas Git, intenta olvidar todo lo que posiblemente conoces acerca de otros VCS como Subversion y Perforce. Hacer esto te ayudará a evitar confusiones sutiles a la hora de utilizar la herramienta. Git almacena y maneja la información de forma muy diferente a esos otros sistemas, a pesar de que su interfaz de usuario es bastante similar. Comprender esas diferencias evitará que te confundas a la hora de usarlo.

Copias instantáneas, no diferencias

La principal diferencia entre Git y cualquier otro VCS (incluyendo Subversion y sus amigos) es la forma en la que manejan sus datos. Conceptualmente, la mayoría de los otros sistemas almacenan la información como una lista de cambios en los archivos. Estos sistemas (CVS, Subversion, Perforce, Bazaar, etc.) manejan la información que almacenan como un conjunto de archivos y las modificaciones hechas a cada uno de ellos a través del tiempo.

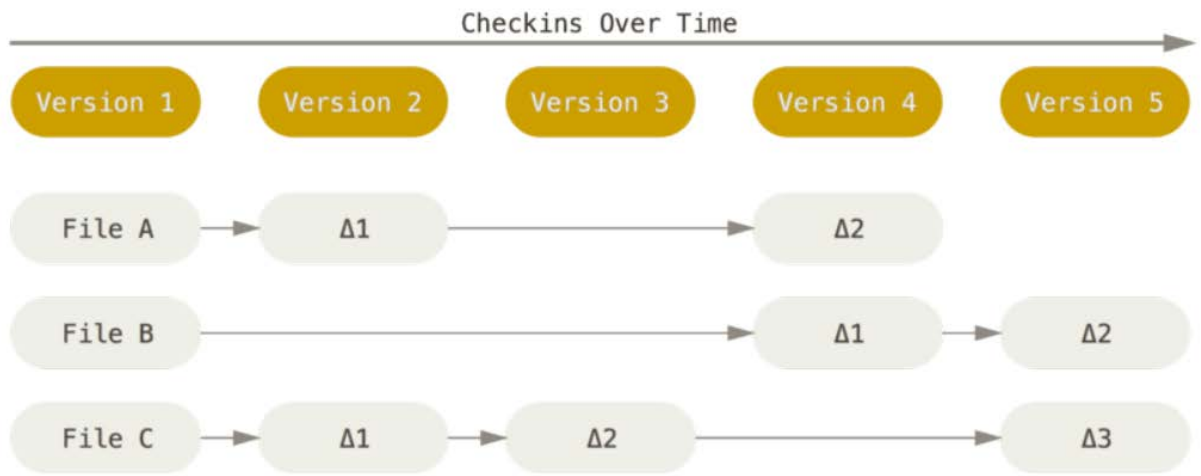


Figura 4. Almacenamiento de datos como cambios en una versión de la base de cada archivo.

Git no maneja ni almacena sus datos de esta forma. Git maneja sus datos como un conjunto de copias instantáneas de un sistema de archivos miniatura. Cada vez que confirmas un cambio, o guardas el estado de tu proyecto en Git, él básicamente toma una foto del aspecto de todos tus archivos en ese momento y guarda una referencia a esa copia instantánea. Para ser eficiente, si los archivos no se han modificado Git no almacena el archivo de nuevo, sino un enlace al archivo anterior idéntico que ya tiene almacenado. Git maneja sus datos como una secuencia de copias instantáneas.

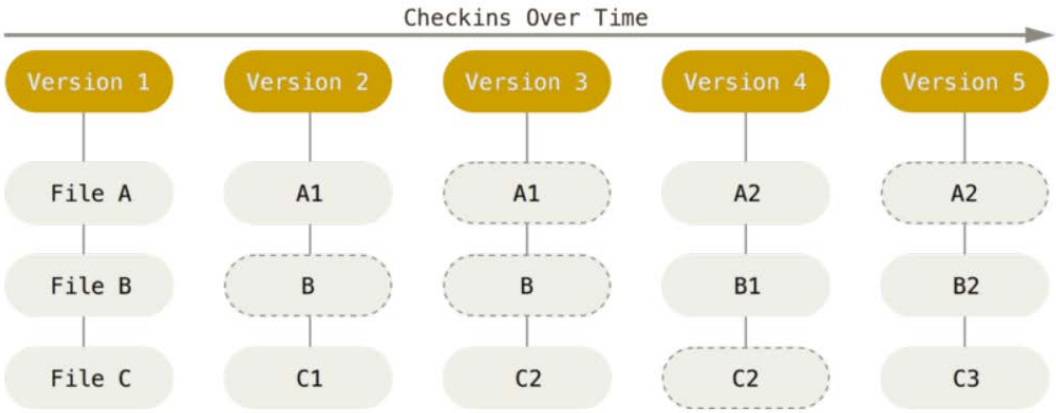


Figura 5. Almacenamiento de datos como instantáneas del proyecto a través del tiempo.

Esta es una diferencia importante entre Git y prácticamente todos los demás VCS.

Hace que Git reconsidere casi todos los aspectos del control de versiones que muchos de los demás sistemas copiaron de la generación anterior. Esto hace que Git se parezca más a un sistema de archivos miniatura con algunas herramientas tremendamente poderosas desarrolladas sobre él, que a un VCS. Exploraremos algunos de los beneficios que obtienes al modelar tus datos de esta manera cuando veamos ramificación (branching) en Git.

Casi todas las operaciones son locales

La mayoría de las operaciones en Git sólo necesitan archivos y recursos locales para funcionar. Por lo general no se necesita información de ningún otro computador de tu red. Si estás acostumbrado a un CVCS donde la mayoría de las operaciones tienen el costo adicional del retardo de la red, este aspecto de Git te va a hacer pensar que los dioses de la velocidad han bendecido Git con poderes sobrenaturales. Debido a que tienes toda la historia del proyecto ahí mismo, en tu disco local, la mayoría de las operaciones parecen prácticamente inmediatas. Por ejemplo, para navegar por la historia del proyecto, Git no necesita conectarse al servidor para obtener la historia y mostrártela - simplemente la lee directamente de tu base de datos local. Esto significa que ves la historia del proyecto casi instantáneamente. Si quieres ver los cambios introducidos en un archivo entre la versión actual y la de hace un mes, Git puede buscar el archivo de hace un mes y hacer un cálculo de diferencias localmente, en lugar de tener que pedirle a un servidor remoto que lo haga, u obtener una versión antigua desde la red y hacerlo de manera local. Esto también significa que hay muy poco que no puedes hacer si estás desconectado o sin VPN.

Si te subes a un avión o a un tren y quieres trabajar un poco, puedes confirmar tus cambios felizmente hasta que consigas una conexión de red para subirlos. Si te vas a casa y no consigues que tu cliente VPN funcione correctamente, puedes seguir trabajando. En muchos otros sistemas, esto es imposible o muy engorroso.

En Perforce, por ejemplo, no puedes hacer mucho cuando no estás conectado al servidor. En Subversion y CVS, puedes editar archivos, pero no puedes confirmar los cambios a tu base de datos (porque tu base de datos no tiene conexión). Esto puede no parecer gran cosa, pero te sorprendería la diferencia que puede suponer.

Git tiene integridad

Todo en Git es verificado mediante una suma de comprobación (checksum en inglés) antes de ser almacenado, y es identificado a partir de ese momento mediante dicha suma. Esto significa que es imposible cambiar los contenidos de cualquier archivo o directorio sin que Git lo sepa. Esta funcionalidad está integrada en Git al más bajo nivel y es parte integral de su filosofía. No puedes perder información durante su transmisión o sufrir corrupción de archivos sin que Git sea capaz de detectarlo.

El mecanismo que usa Git para generar esta suma de comprobación se conoce como hash SHA-1. Se trata de una cadena de 40 caracteres hexadecimales (0-9 y a-f), y se calcula con base en los contenidos del archivo o estructura del directorio en Git. Un hash SHA-1se ve de la siguiente forma:

24b9da6552252987aa493b52f8696cd6d3b00373

Verás estos valores hash por todos lados en Git, porque son usados con mucha frecuencia. De hecho, Git guarda todo no por nombre de archivo, sino por el valor hash de sus contenidos.

Git generalmente solo añade información

Cuando realizas acciones en Git, casi todas ellas sólo añaden información a la base de datos de Git. Es muy difícil conseguir que el sistema haga algo que no se pueda enmendar, o que de algún modo borre información. Como en cualquier VCS, puedes perder o estropear cambios que no has confirmado todavía. Pero después de confirmar una copia instantánea en Git es muy difícil perderla, especialmente si envías tu base de datos a otro repositorio con regularidad.

Esto hace que usar Git sea un placer, porque sabemos que podemos experimentar sin peligro de estropear gravemente las cosas. Para un análisis más exhaustivo de cómo almacena Git su información y cómo puedes recuperar datos aparentemente perdidos.

Estados en GIT

Los Tres Estados

Ahora presta atención. Esto es lo más importante que debes recordar acerca de Git si quieres que el resto de tu proceso de aprendizaje prosiga sin problemas. Git tiene tres estados principales en los que se pueden encontrar tus archivos: confirmado (committed), modificado (modified), y preparado (staged). Confirmado: significa que los datos están almacenados de manera segura en tu base de datos local. Modificado: significa que has modificado el archivo pero todavía no lo has confirmado a tu base de datos. Preparado: significa que has marcado un archivo modificado en su versión actual para que vaya en tu próxima confirmación.

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

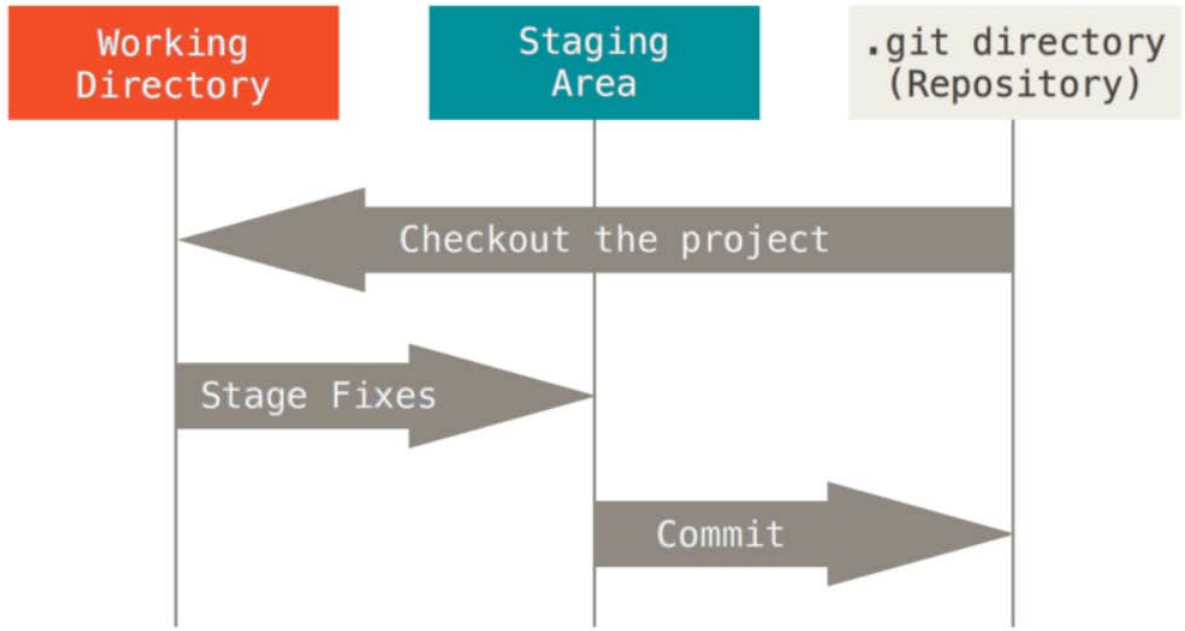


Figura 6. Directorio de trabajo, área de almacenamiento y el directorio Git.

El directorio de Git es donde se almacenan los metadatos y la base de datos de objetos para tu proyecto. Es la parte más importante de Git, y es lo que se copia cuando clonas un repositorio desde otra computadora.

El directorio de trabajo es una copia de una versión del proyecto. Estos archivos se sacan de la base de datos comprimida en el directorio de Git, y se colocan en disco para que los puedas usar o modificar.

El área de preparación es un archivo, generalmente contenido en tu directorio de Git, que almacena información acerca de lo que va a ir en tu próxima confirmación. A veces se le denomina índice ("index"), pero se está convirtiendo en estándar el referirse a ella como el área de preparación.

El flujo de trabajo básico en Git es algo así:

1. Modificas una serie de archivos en tu directorio de trabajo.
2. Preparas los archivos, añadiéndolos a tu área de preparación.
3. Confirmas los cambios, lo que toma los archivos tal y como están en el área de preparación y almacena esa copia instantánea de manera permanente en tu directorio de Git.

Si una versión concreta de un archivo está en el directorio de Git, se considera confirmada (committed). Si ha sufrido cambios desde que se obtuvo del repositorio, pero ha sido añadida al área de preparación, está preparada (staged). Y si ha sufrido cambios desde que se obtuvo del repositorio, pero no se ha preparado, está modificada (modified).

Instalación de GIT

Antes de empezar a utilizar Git, tienes que instalarlo en tu computadora. Incluso si ya está instalado, este es posiblemente un buen momento para actualizarlo a su última versión. Puedes instalarlo como un paquete, a partir de un archivo instalador o bajando el código fuente y compilándolo tú mismo.

NOTA
Este documento fue escrito utilizando la versión **2.0.0** de Git. Aun cuando la mayoría de los comandos que usaremos deben funcionar en versiones más antiguas de Git, es posible que algunos de ellos no funcionen o lo hagan ligeramente diferente si estás utilizando una versión anterior de Git.

Debido a que Git es particularmente bueno en preservar compatibilidad hacia atrás, cualquier versión posterior a 2.0 debe funcionar bien.

Instalación en Linux
Si quieres instalar Git en Linux a través de un instalador binario, en general puedes hacerlo mediante la herramienta básica de administración de paquetes que trae tu distribución. Si estás en Fedora por ejemplo, puedes usar yum:
$ yum install git

Si estás en una distribución basada en Debian como Ubuntu, puedes usar apt-get:
$ apt-get install git

Para opciones adicionales, la página web de Git tiene instrucciones de instalación en diferentes tipos de Unix. Puedes encontrar esta información en http://git-scm.com/download/linux.

Instalación en Mac
Hay varias maneras de instalar Git en un Mac. Probablemente la más sencilla es instalando las herramientas Xcode de Línea de Comandos. En Mavericks (10.9 o superior) puedes hacer esto desde el Terminal si intentas ejecutar *git* por primera vez. Si no lo tienes instalado, te preguntará si deseas instalarlo.

Si deseas una versión más actualizada, puedes hacerlo a partir de un instalador binario.

Un instalador de Git para OSX es mantenido en la página web de Git. Lo puedes descargar en http://git-scm.com/download/mac.

También puedes instalarlo como parte del instalador de Github para Mac. Su interfaz gráfica de usuario tiene la opción de instalar las herramientas de línea de comandos.

Puedes descargar esa herramienta desde el sitio web de Github para Mac en http://mac.github.com

Instalación en Windows

También hay varias maneras de instalar Git en Windows. La forma más oficial está disponible para ser descargada en el sitio web de Git. Solo tienes que visitar http://gitscm.com/download/win y la descarga empezará automáticamente. Fíjate que éste es un proyecto conocido como Git para Windows (también llamado msysGit), el cual es diferente de Git. Para más información acerca de este proyecto visita http://msysgit.github.io/.

Otra forma de obtener Git fácilmente es mediante la instalación de GitHub para Windows. El instalador incluye la versión de línea de comandos y la interfaz de usuario de Git. Además funciona bien con Powershell y establece correctamente "caching" de credenciales y configuración CRLF adecuada. Aprenderemos acerca de todas estas cosas un poco más adelante, pero por ahora es suficiente mencionar que éstas son cosas que deseas. Puedes descargar este instalador del sitio web de GitHub para Windows en http://windows.github.com.

Comandos básicos de GIT

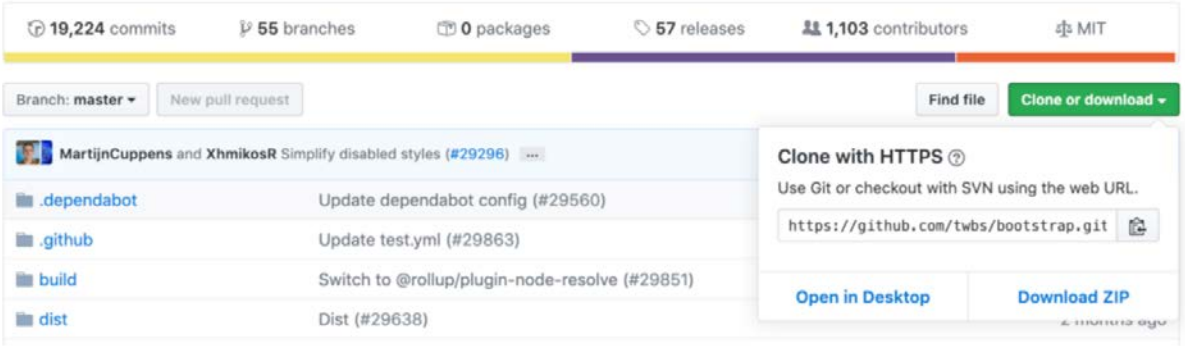
1. Git clone

Git clone es un comando para descargarte el código fuente existente desde un repositorio remoto (como Github, por ejemplo). En otras palabras, Git clone básicamente realiza una copia idéntica de la última versión de un proyecto en un repositorio y la guarda en tu ordenador.

Hay un par de formas de descargar el código fuente, pero una de ellas es la de **clonar de la forma con https**:

git clone <https://link-con-nombre-del-repositorio>

Por ejemplo, si queremos descargar un proyecto desde Github, todo lo que necesitamos es hacer clic sobre el botón verde (clonar o descargar), copiar la URL de la caja y pegarla después del comando git clone que he mostrado más arriba.



Código fuente de Bootstrap en Github

Esto hará una copia del proyecto en tu espacio de trabajo local y así podrás empezar a trabajar con él.

2. Git branch

Las ramas (branch) son altamente importantes en el mundo de Git. Usando ramas, varios desarrolladores pueden trabajar en paralelo en el mismo proyecto simultáneamente. Podemos usar el comando git branch para crearlas, listarlas y eliminarlas.

Creando una nueva rama:

git branch <nombre-de-la-rama>

Este comando creará una rama en local. Para enviar (push) la nueva rama al repositorio remoto, necesitarás usar el siguiente comando:

git push <nombre-remoto> <nombre-rama>

Visualización de ramas:

git branch
git branch --list

Borrar una rama:

git branch -d <nombre-de-la-rama>

3. Git checkout

Este es también uno de los comandos más utilizados en Git. Para trabajar en una rama, primero tienes que cambiarte a ella. Usaremos **git checkout** principalmente para cambiarte de una rama a otra. También lo podemos usar para chequear archivos y commits.

git checkout <nombre-de-la-rama>

Hay algunos pasos que debes seguir para cambiarte exitosamente entre ramas:

- Los cambios en tu rama actual tienen que ser confirmados o almacenados en el guardado rápido (stash) antes de que cambies de rama.
- La rama a la que te quieras cambiar debe existir en local.

Hay también un comando de acceso directo que te permite crear y cambiarte a esa rama al mismo tiempo:

git checkout -b <nombre-de-tu-rama>

Este comando crea una nueva rama en local (-b viene de rama (branch)) y te cambia a la rama que acabas de crear.

4. Git status

El comando de git status nos da toda la información necesaria sobre la rama actual.

git status

Podemos encontrar información como:

- Si la rama actual está actualizada
- Si hay algo para confirmar, enviar o recibir (pull).
- Si hay archivos en preparación (staged), sin preparación(unstaged) o que no están recibiendo seguimiento (untracked)
- Si hay archivos creados, modificados o eliminados

```
Cem-MacBook-Pro:my-new-app cem$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory
)

        modified:   src/App.js

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

        src/components/
```

git status nos da información acerca del archivo y las ramas

5. Git add

Cuando creamos, modificamos o eliminamos un archivo, estos cambios suceden en local y no se incluirán en el siguiente commit (a menos que cambiemos la configuración).

Necesitamos usar el comando git add para incluir los cambios del o de los archivos en tu siguiente commit.

Añadir un único archivo:

git add <archivo>

Añadir todo de una vez:

git add -A

Si revisas la captura de pantalla que he dejado en la sección 4, verás que hay nombres de archivos en rojo - esto significa que los archivos sin preparación. Estos archivos no serán incluidos en tus commits hasta que no los añadas.

Para añadirlos, necesitas usar el git add:

```
Cem-MacBook-Pro:my-new-app cem$ git add -A
Cem-MacBook-Pro:my-new-app cem$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   src/App.js
        new file:   src/components/myFirstComponent.js
```

Los archivos en verde han sido añadidos a la preparación gracias al git add

Importante: El comando git add no cambia el repositorio y los cambios que no han sido guardados hasta que no utilicemos el comando de confirmación git commit.

6. Git commit

Este sea quizás el comando más utilizado de Git. Una vez que se llega a cierto punto en el desarrollo, queremos guardar nuestros cambios (quizás después de una tarea o asunto específico).

Git commit es como establecer un punto de control en el proceso de desarrollo al cual puedes volver más tarde si es necesario.

También necesitamos escribir un mensaje corto para explicar qué hemos desarrollado o modificado en el código fuente.

```
git commit -m "mensaje de confirmación"
```

Importante: Git commit guarda tus cambios únicamente en local.

7. Git push

Después de haber confirmado tus cambios, el siguiente paso que quieres dar es enviar tus cambios al servidor remoto. Git push envía tus commits al repositorio remoto.

```
git push <nombre-remoto> <nombre-de-tu-rama>
```

De todas formas, si tu rama ha sido creada recientemente, puede que tengas que cargar y subir tu rama con el siguiente comando:

```
git push --set-upstream <nombre-remoto> <nombre-de-tu-rama>
```

or

```
git push -u origin <nombre-de-tu-rama>
```

Importante: Git push solamente carga los cambios que han sido confirmados.

8. Git pull

El comando **git pull** se utiliza para recibir actualizaciones del repositorio remoto. Este comando es una combinación del **git fetch** y del **git merge** lo cual significa que cundo usemos el git pull recogeremos actualizaciones del repositorio remoto (git fetch) e inmediatamente aplicamos estos últimos cambios en local (git merge).

```
git pull <nombre-remoto>
```

Esta operación puede generar conflictos que tengamos que resolver manualmente.

9. Git revert

A veces, necesitaremos deshacer los cambios que hemos hecho. Hay varias maneras para deshacer nuestros cambios en local y/o en remoto (dependiendo de lo que necesitemos), pero necesitaremos utilizar cuidadosamente estos comandos para evitar borrados no deseados.

Una manera segura para deshacer nuestras commits es utilizar **git revert**. Para ver nuestro historial de commits, primero necesitamos utilizar el **git log --oneline:**

```
Cem-MacBook-Pro:my-new-app cem$ git log --oneline
3321844 (HEAD -> master) test
e64e7bb Initial commit from Create React App
```

histórico de git en mi rama master

Entonces, solo necesitamos especificar el código de comprobación que encontrarás junto al commit que queremos deshacer:

git revert 3321844

Después de esto, verás una pantalla como la de abajo -tan solo presiona **shift + q** para salir:

```
Revert "test"

This reverts commit 332184490ef2b5db289d85ed3f1a13ff2d5f94b9.

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# Committer: Cem <cem@Cem-MacBook-Pro.local>
#
# On branch master
# Changes to be committed:
#       modified:   src/App.js
#       deleted:    src/components/myFirstComponent.js
#
~
~
~
~
~
~
~
~
~
"~/Desktop/my-new-app/.git/COMMIT_EDITMSG" 14L, 384C
```

El comando git revert deshará el commit que le hemos indicado, pero creará un nuevo commit deshaciendo la anterior:

```
Cem-MacBook-Pro:my-new-app cem$ git log --oneline
cd7fe6f (HEAD -> master) Revert "test"
3321844 test
e64e7bb Initial commit from Create React App
```

commit generado con el git revert

La ventaja de utilizar **git revert** es que no afecta al commit histórico. Esto significa que puedes seguir viendo todos los commits en tu histórico, incluso los revertidos.

Otra medida de seguridad es que todo sucede en local a no ser que los enviemos al repositorio remoto. Por esto es que git revert es más seguro de usar y es la manera preferida para deshacer los commits.

10. Git merge

Cuando ya hayas completado el desarrollo de tu proyecto en tu rama y todo funcione correctamente, el último paso es fusionar la rama con su rama padre (dev o master). Esto se hace con el comando git merge.

Git merge básicamente integra las características de tu rama con todos los commits realizados a las ramas dev (o master). Es importante que recuerdes que tienes que estar en esa rama específica que quieres fusionar con tu rama de características.

Por ejemplo, cuando quieres fusionar tu rama de características en la rama dev:

Primero, debes cambiarte a la rama dev:

git checkout dev

Antes de fusionar, debes actualizar tu rama dev local:

git fetch

Por último, puedes fusionar tu rama de características en la rama dev:

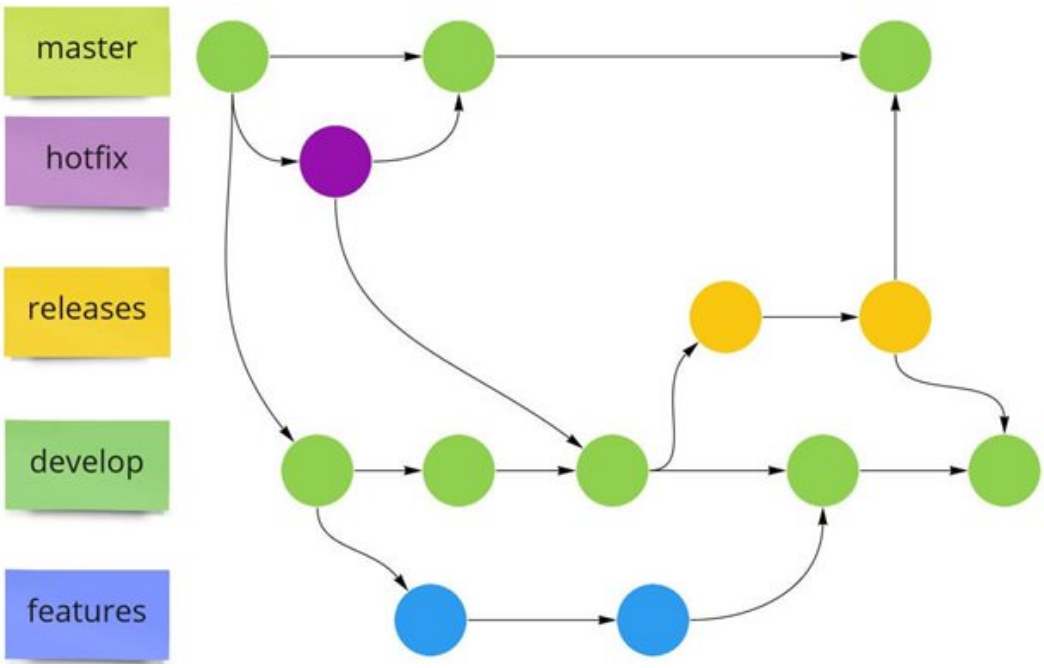
git merge <nombre-de-la-rama>

Pista: Asegúrate de que tu rama dev tiene la última versión antes de fusionar otras ramas, si no, te enfrentarás a conflictos u otros problemas no deseados.

Flujos de Trabajo

Estos son los cinco tipos de flujo de trabajo Git que podemos utilizar en un proyecto.

Git Flow



Es el flujo de trabajo más popular y extendido. Se basa en dos ramas principales con una vida infinita. Para cada tarea que se le asigna a un desarrollador se crea una rama feature en la cual se llevará a cabo la tarea. Una vez que ha finalizado, realizará un *pull request* (validación) contra develop para que validen el código.

Pasamos a detallar las dos ramas principales que se utilizan:

- **Master**: Contiene el **código de producción**. Todo el código de desarrollo, a través del uso de releases, se mergea (fusiona) en esta rama en algún momento.
- **Develop**: Contiene código de pre-producción. Cuando un desarrollador finaliza su feature, lo mergea contra esta rama.

Durante el ciclo de desarrollo, se usan varios tipos de ramas para dar soporte:

- **Feature**: Por cada tarea que se realiza, se crea una nueva rama para trabajar en ella. Esta rama parte de develop.
- **Hotfix**: Parte de master. Rama encargada de corregir una incidencia crítica en producción.

- **Releases**: Parte de develop. Rama encargada de generar valor al producto o proyecto. Contiene el código que se desplegará, y una vez que se han probado las features integradas en la release, se "mergeará" a la rama master.

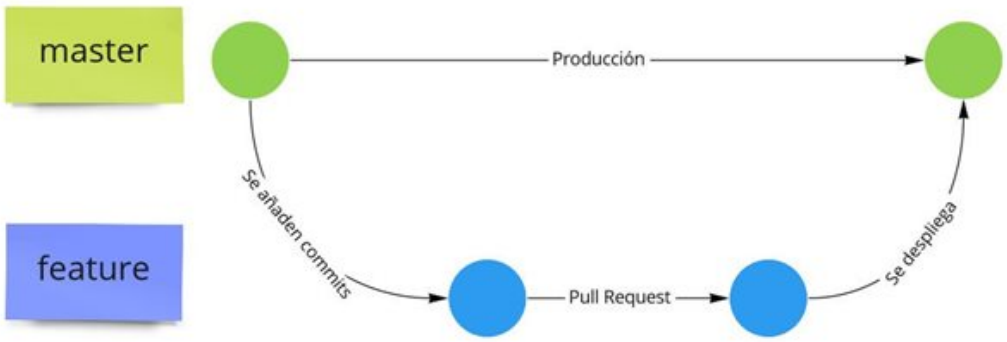
Pros

- Fácil de comprender el flujo de ramas.
- Ideal para **productos estables** que no requieren de desplegar cambios inmediatamente.
- Muy recomendable cuando el equipo tiene todo tipo de desarrolladores. El control de las features más la release hace que el código no se deteriore.
- Perfecto para productos **open-source** en los que pueden colaborar todo tipo de desarrolladores.

Contras

- No es el más indicado si tu proyecto necesita iterar muy rápido y subir a producción varias veces al día o semana.
- En caso de que el proyecto utilice varias herramientas de integración continua, la rama **develop** puede convertirse en una **rama redundante de master**.
- El uso de la rama master como rama protegida. Muchas herramientas de automatización usan la rama master por defecto.
- Gran complejidad en las ramas creadas de hotfix y releases. La integración continua elimina la necesidad de la creación de estas ramas, facilitando el despliegue.

GitHub Flow



La diferencia principal con GitFlow es **la desaparición de la rama develop**. Se basa en los siguientes principios:

- Todo lo que haya en la **rama master debe ser desplegado.**
- Para cualquier característica nueva, crearemos una rama de master, usando un nombre descriptivo.
- Debemos hacer commit en esta rama en local y hacer push con el mismo nombre en el server.
- Si necesitamos feedback, utilizaremos las herramientas de mergeo como **pull request.**
- Una vez revisado el código, podemos mergear contra master.
- Una vez mergeado contra master, debemos desplegar los cambios.

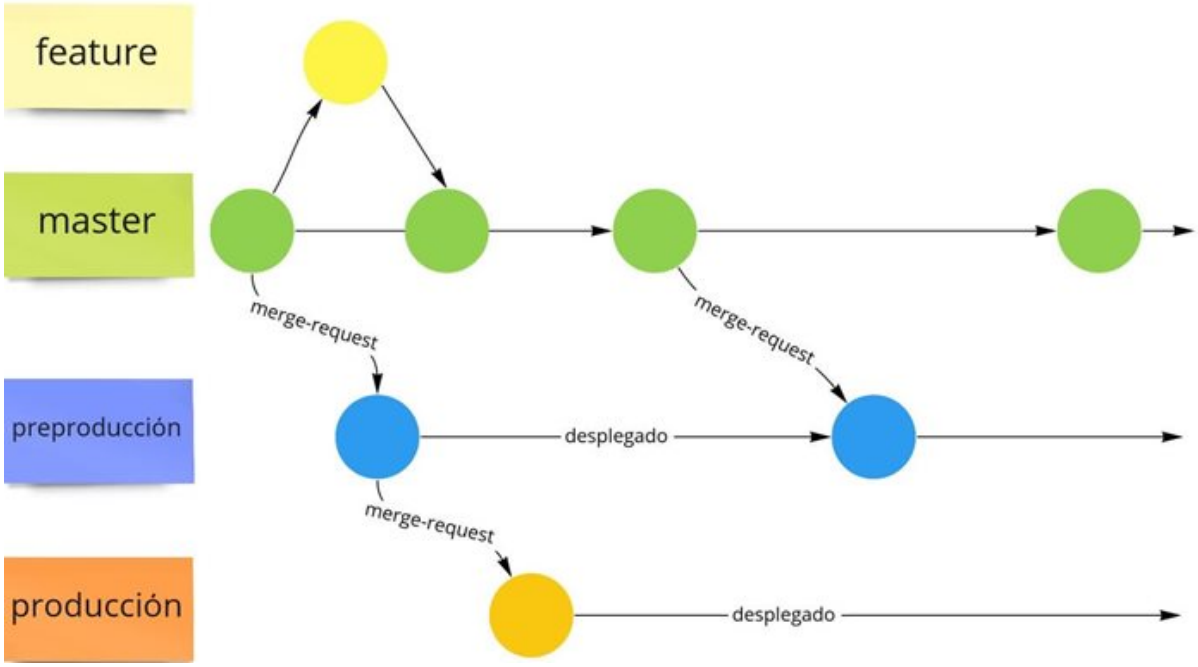
Pros

- Útil y amigable con las actuales herramientas de CI/CC.
- Recomendado para **features de duración corta** (diarias o incluso de horas).
- Flujo ligero y recomendado si el proyecto requiere de una **entrega de valor constante.**

Contras

- **Inestabilidad de master** si no se utilizan las herramientas de testing/PR correctamente.
- No recomendado para múltiples entornos productivos.
- Dependiendo el producto, podemos tener **restricciones de despliegues,** sobre todo en **aplicaciones SaaS**.

GitLab Flow



GitLab Flow es una alternativa/extensión de **GitHub Flow y Git Flow,** que nace debido a las carencias que adoptan estos dos flujos. Mientras que una de las consignas de GitHub es que todo lo que haya en master es desplegado, hay ciertos casos en los que no es posible cumplirlo o no se necesita. Por ejemplo: aplicaciones iOS cuando pasan a la App Store Validation o incluso tener ventanas de despliegue por la naturaleza del cliente.

El flujo propone utilizar master, ramas features y ramas de entorno. Una vez que una feature está finalizada hacemos una **merge request contra maste**r. Una vez que master tiene varias features, llevamos a cabo una **merge request a preproducción** con el conjunto de features anteriores, que a su vez, son candidatas de **pasar a producción haciendo otro merge request**. De esta manera conseguimos que el código subido a producción sea muy estable, ya que validamos featues tanto a nivel individual como en lote.

La naturaleza de este flujo no requiere generar ramas de releases, ya que cada entorno será desplegado con cada merge request aceptada.

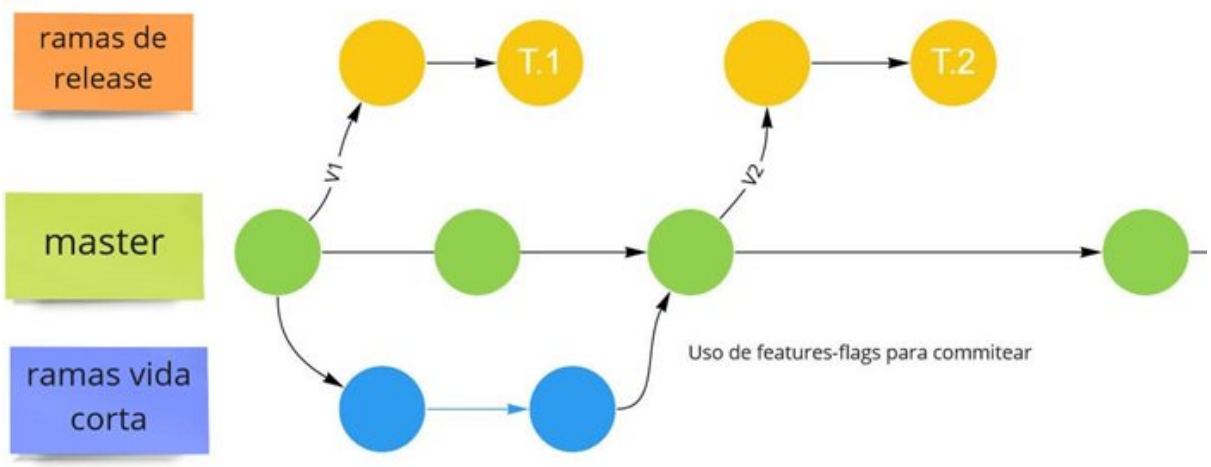
Pros

- Mucha confianza en la **versión de producción.**
- Ciclo de desarrollo muy seguro. Se revisa el código tanto a nivel individual en la feature, como a nivel global al pasar a preproducción o producción, mitigando el impacto.
- Previene el “overheap” de crear releases, *taggings*, merge a develop.

Contras

- Requiere de un equipo que valide las MR tanto de las features, como de los diferentes entornos.
- **Tiempo demasiado alto** para la entrega de valor. Desde que se crea y se valida la feature, hasta que llega a producción, tiene que pasar por muchas validaciones.
- **Más complejo** que GitHub Flow.

Trunk-based Flow



Este flujo es muy similar a GitHub Flow, con la característica nueva de las releases branch y el **cambio de filosofía** que presenta. Los principios que rigen este flujo son los siguientes:

- Los desarrolladores debemos colaborar siempre sobre el trunk (o master).
- Bajo ningún concepto debemos crear features branch utilizando documentación técnica. En caso de que la evolución sea compleja, haremos uso de **features flags** (condicionales en el código) para activar o desactivar la nueva carcterística.
- Preferiblemente usaremos la metodología **Pair-Programming** en vez de hacer uso de PR.

Se sacan ramas de release para poder desplegar el código en diferentes entornos, ya sea mobile, web, etc.

Pros

- Muy útil si nuestro proyecto necesita iterar rápido y entregar valor lo antes posible.
- Responde muy bien a proyectos agile pequeños.
- Preparado para equipos que utilizan **pair-programming**.
- Funciona muy bien con un **equipo experimentado y cerrado**.

Contras

- Debemos ser responsables de nuestro código y hacer el esfuerzo de subir **código de calidad**.
- El proyecto debe tener QA y CD muy maduro, de lo contrario se introducirán muchos **bugs** en la rama trunk.
- No es recomendable utilizarlo en proyectos **open-source**, ya que requieren de una verificación extra de código.

Master-only Flow

Uso de features-flags para commitear

El flujo de trabajo usa solo una rama infinita. Usaremos master en esta descripción, ya que probablemente sea el nombre más común y ya es una convención de Git, pero también puedes usar, por ejemplo, current, default, mainline.

Realizaremos cada **feature o hotfix** sobre la **misma rama,** testearemos y haremos commit **en local**. Una vez que este cambio se ha aprobado, haremos push contra master en origin, desplegándose en producción inmediatamente.

Este flujo está orientado a **proyectos con desarrolladores muy experimentados** y en el cual se fomente el pair-programming.

Debemos usar **features-flags** para poder **integrar el código en la rama master** y evitar conflictos a lo largo del tiempo.

Pros

- Solo mantenemos una rama.
- Tendremos el histórico del proyecto limpio.
- Con el uso de pair-programming y desarrolladores experimentados, **la entrega de valor en producción es inmediata**.

Contras

- No soporta múltiples entornos productivos.
- Requiere de **desarrolladores experimentados** para no dejar inestable la rama principal.
- El proyecto requiere de **guidelines de código muy estrictas.**