

2022 QA Success Blueprint

Guide made for QA managers to enable your contact center for success

Playvox

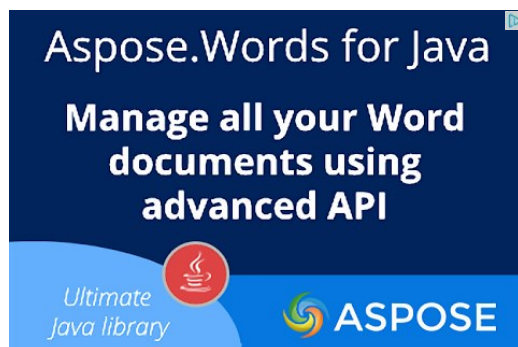
Download

Pro Git, el libro oficial de Git

2.2. Guardando cambios en el repositorio

Tienes un repositorio Git completo, y una copia de trabajo de los archivos de ese proyecto. Necesitas hacer algunos cambios, y confirmar instantáneas de esos cambios a tu repositorio cada vez que el proyecto alcance un estado que desees grabar.

Recuerda que cada archivo de tu directorio de trabajo puede estar en uno de estos dos estados: bajo seguimiento (*tracked*), o sin seguimiento (*untracked*). Los archivos bajo seguimiento son aquellos que existían en la última instantánea; pueden estar sin modificaciones, modificados, o preparados. Los archivos sin seguimiento son todos los demás — cualquier archivo de tu directorio que no estuviese en tu última instantánea ni está en tu área de preparación —. La primera vez que clonas un repositorio, todos tus archivos estarán bajo seguimiento y sin modificaciones, ya que los acabas de copiar y no has modificado nada.



A medida que editas archivos, Git los ve como modificados, porque los has cambiado desde tu última confirmación. Preparas estos archivos modificados y luego confirmas todos los cambios que hayas preparado, y el ciclo se repite. Este proceso queda ilustrado en la Figura 2-1.

File Status Lifecycle

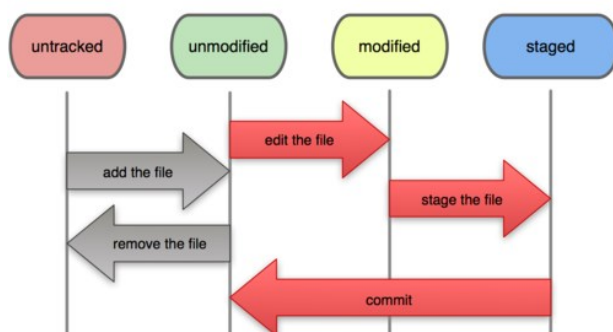


Figura 2.1 El ciclo de vida del estado de tus archivos

2.2.1. Comprobando el estado de tus archivos

Tu principal herramienta para determinar qué archivos están en qué estado es el comando `git status`. Si ejecutas este comando justo después de clonar un repositorio, deberías ver algo así:

```
$ git status
# On branch master
```

Indice de contenidos

1. Empezando

Capítulo 2. Fundamentos de Git

2.1. Obteniendo un repositorio Git

2.2. Guardando cambios en el repositorio

2.3. Viendo el histórico de confirmaciones

2.4. Deshaciendo cosas

2.5. Trabajando con repositorios remotos

2.6. Creando etiquetas

2.7. Consejos y trucos

2.8. Resumen

3. Trabajando con ramas en Git

4. Git en un servidor

5. Git en entornos distribuidos

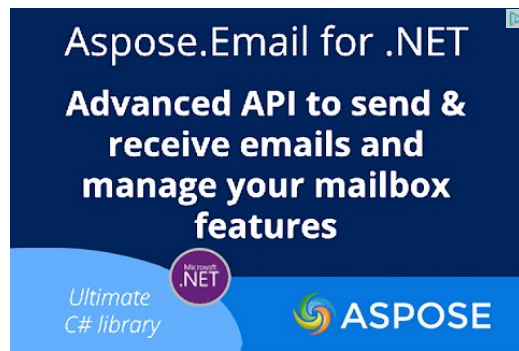
6. Las herramientas de Git

7. Personalizando Git

8. Git y otros sistemas

9. Funcionamiento interno de Git

```
nothing to commit (working directory clean)
```



Esto significa que tienes un directorio de trabajo limpio — en otras palabras, no tienes archivos bajo seguimiento y modificados —. Git tampoco ve ningún archivo que no esté bajo seguimiento, o estaría listado ahí. Por último, el comando te dice en qué rama estás. Por ahora, esa rama siempre es `master`, que es la predeterminada. No te preocupes de eso por ahora, el siguiente capítulo tratará los temas de las ramas y las referencias en detalle.

Digamos que añades un nuevo archivo a tu proyecto, un sencillo archivo `README`. Si el archivo no existía y ejecutas `git status`, verás tus archivos sin seguimiento así:

```
$ vim README
$ git status
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#   README
nothing added to commit but untracked files present (use "git add" to track)
```

Puedes ver que tu nuevo archivo `README` aparece bajo la cabecera “Archivos sin seguimiento” (“*Untracked files*”) de la salida del comando. Sin seguimiento significa básicamente que Git ve un archivo que no estaba en la instantánea anterior; Git no empezará a incluirlo en las confirmaciones de tus instantáneas hasta que se lo indiques explícitamente. Lo hace para que no incluyas accidentalmente archivos binarios generados u otros archivos que no tenías intención de incluir. Sí que quieres incluir el `README`, así que vamos a iniciar el seguimiento del archivo.

2.2.2. Seguimiento de nuevos archivos

Para empezar el seguimiento de un nuevo archivo se usa el comando `git add`. Iniciaremos el seguimiento del archivo `README` ejecutando esto:

```
$ git add README
```

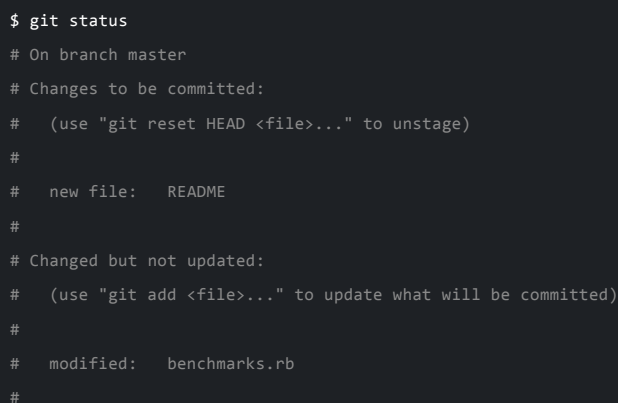
Si vuelves a ejecutar el comando `git status`, verás que tu `README` está ahora bajo seguimiento y preparado:

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   new file:   README
#
```

Puedes ver que está preparado porque aparece bajo la cabecera “Cambios a confirmar” (“Changes to be committed”). Si confirmas ahora, la versión del archivo en el momento de ejecutar `git add` será la que se incluya en la instantánea. Recordarás que cuando antes ejecutaste `git init`, seguidamente ejecutaste `git add` (archivos). Esto era para iniciar el seguimiento de los archivos de tu directorio. El comando `git add` recibe la ruta de un archivo o de un directorio; si es un directorio, añade todos los archivos que contenga de manera recursiva.

2.2.3. Preparando archivos modificados

Vamos a modificar un archivo que estuviese bajo seguimiento. Si modificas el archivo `benchmarks.rb` que estaba bajo seguimiento, y ejecutas el comando `status` de nuevo, verás algo así:

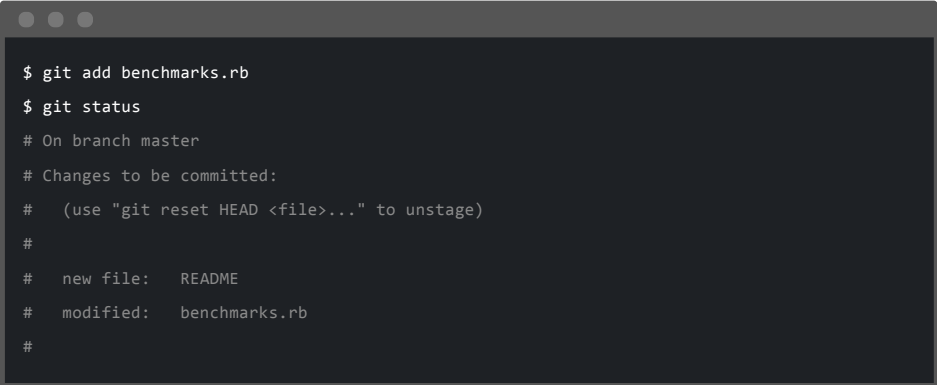
A terminal window with a dark background and light gray text. It shows the output of the 'git status' command. The output indicates that the repository is on the 'master' branch and lists changes to be committed. It shows a new file 'README' and a modified file 'benchmarks.rb'.

```
$ git status

# On branch master

# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   new file:   README
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
#   modified:   benchmarks.rb
#
```

El archivo `benchmarks.rb` aparece bajo la cabecera “Modificados pero no actualizados” (“Changed but not updated”) — esto significa que un archivo bajo seguimiento ha sido modificado en el directorio de trabajo, pero no ha sido preparado todavía —. Para prepararlo, ejecuta el comando `git add` (es un comando multiuso — puedes utilizarlo para empezar el seguimiento de archivos nuevos, para preparar archivos, y para otras cosas como marcar como resueltos archivos con conflictos de unión —). Ejecutamos `git add` para preparar el archivo `benchmarks.rb`, y volvemos a ejecutar `git status`:

A terminal window showing the execution of 'git add benchmarks.rb' followed by 'git status'. The status output now shows 'benchmarks.rb' as a modified file, alongside the 'README' file.

```
$ git add benchmarks.rb
$ git status

# On branch master

# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   new file:   README
#   modified:   benchmarks.rb
#
```

Ambos archivos están ahora preparados y se incluirán en tu próxima confirmación. Supón que en este momento recuerdas que tenías que hacer una pequeña modificación en `benchmarks.rb` antes de confirmarlo. Lo vuelves abrir, haces ese pequeño cambio, y ya estás listo para confirmar. Sin embargo, si vuelves a ejecutar

git status verás lo siguiente:

```
$ vim benchmarks.rb
$ git status
# On branch master
#
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   new file:   README
#   modified:   benchmarks.rb
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
#   modified:   benchmarks.rb
#
```

¿Pero qué...? Ahora `benchmarks.rb` aparece listado como preparado y como no preparado. ¿Cómo es posible? Resulta que Git prepara un archivo tal y como era en el momento de ejecutar el comando `git add`. Si haces `git commit` ahora, la versión de `benchmarks.rb` que se incluirá en la confirmación será la que fuese cuando ejecutaste el comando `git add`, no la versión que estás viendo ahora en tu directorio de trabajo. Si modificas un archivo después de haber ejecutado `git add`, tendrás que volver a ejecutar `git add` para preparar la última versión del archivo:

```
$ git add benchmarks.rb
$ git status
# On branch master
#
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   new file:   README
#   modified:   benchmarks.rb
#
```

2.2.4. Ignorando archivos

A menudo tendrás un tipo de archivos que no quieras que Git añada automáticamente o te muestre como no versionado. Suelen ser archivos generados automáticamente, como archivos de log, o archivos generados por tu compilador. Para estos casos puedes crear un archivo llamado `.gitignore`, en el que listas los patrones de nombres que deseas que sean ignorados. He aquí un archivo `.gitignore` de ejemplo:

```
$ cat .gitignore
*.o
*.a
*~
```

La primera línea le dice a Git que ignore cualquier archivo cuyo nombre termine en `.o` o `.a` — archivos objeto que suelen ser producto de la compilación de código —. La segunda línea le dice a Git que ignore todos los archivos que terminan en tilde (`~`), usada por muchos editores de texto, como Emacs, para marcar archivos temporales. También puedes incluir directorios de log, temporales, documentación generada automáticamente, etc. Configurar un archivo `.gitignore` antes de empezar a trabajar suele ser una buena idea, para así no confirmar archivos que no quieres en tu repositorio Git.

Las reglas para los patrones que pueden ser incluidos en el archivo `.gitignore` son:

- Las líneas en blanco, o que comienzan por #, son ignoradas.
- Puedes usar patrones glob estándar.
- Puedes indicar un directorio añadiendo una barra hacia delante (/) al final.
- Puedes negar un patrón añadiendo una exclamación (!) al principio.

Los patrones glob son expresiones regulares simplificadas que pueden ser usadas por las shells. Un asterisco (*) reconoce cero o más caracteres; [abc] reconoce cualquier carácter de los especificados entre corchetes (en este caso, a, b, o c); una interrogación (?) reconoce un único carácter; y caracteres entre corchetes separados por un guión ([0-9]) reconoce cualquier carácter entre ellos (en este caso, de 0 a 9).

He aquí otro ejemplo de archivo .gitignore:

```
# a comment - this is ignored
*.a      # no .a files

!lib.a   # but do track lib.a, even though you're ignoring .a files above
/TODOD   # only ignore the root TODOD file, not subdir/TODOD
build/   # ignore all files in the build/ directory
doc/*.txt # ignore doc/notes.txt, but not doc/server/arch.txt
```

2.2.5. Viendo tus cambios preparados y no preparados

Si el comando `git status` es demasiado impreciso para ti — quieres saber exactamente lo que ha cambiado, no sólo qué archivos fueron modificados — puedes usar el comando `git diff`. Veremos `git diff` en más detalle después; pero probablemente lo usarás para responder estas dos preguntas: ¿qué has cambiado pero aún no has preparado?, y ¿qué has preparado y estás a punto de confirmar? Aunque `git status` responde esas preguntas de manera general, `git diff` te muestra exactamente las líneas añadidas y eliminadas — el parche, como si dijésemos —.

Supongamos que quieres editar y preparar el archivo `README` otra vez, y luego editar el archivo `benchmarks.rb` sin prepararlo. Si ejecutas el comando `status`, de nuevo verás algo así:

```
$ git status
# On branch master
#
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   new file:   README
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
#   modified:   benchmarks.rb
#
```

Para ver lo que has modificado pero aún no has preparado, escribe `git diff`:

```
$ git diff
diff --git a/benchmarks.rb b/benchmarks.rb
index 3cb747f..da65585 100644
--- a/benchmarks.rb
+++ b/benchmarks.rb
@@ -36,6 +36,10 @@ def main
    @commit.parents[0].parents[0].parents[0]
  end
```

```

+   run_code(x, 'commits 1') do
+       git.commits.size
+   end
+
+   run_code(x, 'commits 2') do
+       log = git.commits('master', 15)
+       log.size
+   end
end

```

Ese comando compara lo que hay en tu directorio de trabajo con lo que hay en tu área de preparación. El resultado te indica los cambios que has hecho y que todavía no has preparado.

Si quieres ver los cambios que has preparado y que irán en tu próxima confirmación, puedes usar `git diff --cached`. (A partir de la versión 1.6.1 de Git, también puedes usar `git diff --staged`, que puede resultar más fácil de recordar). Este comando compara tus cambios preparados con tu última confirmación:

```

$ git diff --cached
diff --git a/README b/README
new file mode 100644
index 0000000..03902a1
--- /dev/null
+++ b/README2
@@ -0,0 +1,5 @@
+grit
+by Tom Preston-Werner, Chris Wanstrath
+http://github.com/mojombo/grit
+
+Grit is a Ruby library for extracting information from a Git repository

```

Es importante indicar que `git diff` por sí solo no muestra todos los cambios hechos desde tu última confirmación — sólo los cambios que todavía no están preparados —. Esto puede resultar desconcertante, porque si has preparado todos tus cambios, `git diff` no mostrará nada.

Por poner otro ejemplo, si preparas el archivo `benchmarks.rb` y después lo editas, puedes usar `git diff` para ver las modificaciones del archivo que están preparadas, y las que no lo están:

```

$ git add benchmarks.rb
$ echo '# test line' >> benchmarks.rb
$ git status
# On branch master
#
# Changes to be committed:
#
#   modified:   benchmarks.rb
#
# Changed but not updated:
#
#   modified:   benchmarks.rb
#

```

Ahora puedes usar `git diff` para ver qué es lo que aún no está preparado:

```

$ git diff
diff --git a/benchmarks.rb b/benchmarks.rb
index e445e28..86b2f7c 100644
--- a/benchmarks.rb
+++ b/benchmarks.rb
@@ -127,3 +127,4 @@ end
main()

##pp Grit::GitRuby.cache_client.stats
+# test line

```

Y `git diff --cached` para ver los cambios que llevas preparados hasta ahora:

```
$ git diff --cached
diff --git a/benchmarks.rb b/benchmarks.rb
index 3cb747f..e445e28 100644
--- a/benchmarks.rb
+++ b/benchmarks.rb
@@ -36,6 +36,10 @@ def main
    @commit.parents[0].parents[0].parents[0]
  end

+  run_code(x, 'commits 1') do
+    git.commits.size
+  end
+
  run_code(x, 'commits 2') do
    log = git.commits(`master`, 15)
    log.size
```

2.2.6. Confirmando tus cambios

Ahora que el área de preparación está como tú quieres, puedes confirmar los cambios. Recuerda que cualquier cosa que todavía esté sin preparar — cualquier archivo que hayas creado o modificado, y sobre el que no hayas ejecutado `git add` desde su última edición — no se incluirá en esta confirmación. Se mantendrán como modificados en tu disco.

En este caso, la última vez que ejecutaste `git status` viste que estaba todo preparado, por lo que estás listo para confirmar tus cambios. La forma más fácil de confirmar es escribiendo `git commit`:

```
$ git commit
```

Al hacerlo, se ejecutará tu editor de texto. (Esto se configura a través de la variable de entorno `$EDITOR` de tu shell — normalmente vim o emacs, aunque puedes configurarlo usando el comando `git config --global core.editor` como vimos en el [Capítulo 1](#))

El editor mostrará el siguiente texto (este ejemplo usa Vim):

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   README
#       modified:  benchmarks.rb
~
~
~
".git/COMMIT_EDITMSG" 10L, 283C
```

Puedes ver que el mensaje de confirmación predeterminado contiene la salida del comando `git status` comentada, y una línea vacía arriba del todo. Puedes eliminar estos comentarios y escribir tu mensaje de confirmación, o puedes dejarlos para ayudarte a recordar las modificaciones que estás confirmando. (Para un recordatorio todavía más explícito de lo que has modificado, puedes pasar la opción `-v` a `git commit`. Esto provoca que se añadan también las diferencias de tus cambios, para que veas exactamente lo que hiciste.) Cuando sales del editor, Git crea tu confirmación con el mensaje que hayas especificado (omitiendo los comentarios y las diferencias).

Como alternativa, puedes escribir tu mensaje de confirmación desde la propia línea de comandos mediante la opción `-m`:

```
$ git commit -m "Story 182: Fix benchmarks for speed"
[master]: created 463dc4f: "Fix benchmarks for speed"

2 files changed, 3 insertions(+), 0 deletions(-)
create mode 100644 README
```

¡Acabas de crear tu primera confirmación! Puedes ver que el comando `commit` ha dado cierta información sobre la confirmación: a qué rama has confirmado (master), cuál es su suma de comprobación SHA-1 de la confirmación (463dc4f), cuántos archivos se modificaron, y estadísticas acerca de cuántas líneas se han añadido y cuántas se han eliminado.

Recuerda que la confirmación registra la instantánea de tu área de preparación. Cualquier cosa que no preparases sigue estando modificada; puedes hacer otra confirmación para añadirla a la historia del proyecto. Cada vez que confirmas, estás registrando una instantánea de tu proyecto, a la que puedes volver o con la que puedes comparar más adelante.

2.2.7. Saltándote el área de preparación

Aunque puede ser extremadamente útil para elaborar confirmaciones exactamente a tu gusto, el área de preparación es en ocasiones demasiado compleja para las necesidades de tu flujo de trabajo. Si quieres saltarte el área de preparación, Git proporciona un atajo. Pasar la opción `-a` al comando `git commit` hace que Git prepare todo archivo que estuviese en seguimiento antes de la confirmación, permitiéndote obviar toda la parte de `git add`:

```
$ git status

# On branch master

#

# Changed but not updated:

#

#   modified:   benchmarks.rb

#

$ git commit -a -m 'added new benchmarks'

[master 83e38c7] added new benchmarks

1 files changed, 5 insertions(+), 0 deletions(-)
```

Fíjate que no has tenido que ejecutar `git add` sobre el archivo `benchmarks.rb` antes de hacer la confirmación.

2.2.8. Eliminando archivos

Para eliminar un archivo de Git, debes eliminarlo de tus archivos bajo seguimiento (más concretamente, debes eliminarlo de tu área de preparación), y después confirmar. El comando `git rm` se encarga de eso, y también elimina el archivo de tu directorio de trabajo, para que no lo veas entre los archivos sin seguimiento.

Si simplemente eliminas el archivo de tu directorio de trabajo, aparecerá bajo la cabecera *“Modificados pero no actualizados”* (*“Changed but not updated”*) (es decir, *sin preparar*) de la salida del comando `git status`:

```
$ rm grit.gemspec

$ git status

# On branch master

#

# Changed but not updated:

#   (use "git add/rm <file>..." to update what will be committed)

#

#       deleted:    grit.gemspec

#
```

Si entonces ejecutas el comando `git rm`, preparas la eliminación del archivo en cuestión:

```
$ git rm grit.gemspec

rm 'grit.gemspec'

$ git status

# On branch master

#

# Changes to be committed:

#   (use "git reset HEAD <file>..." to unstage)

#

#       deleted:    grit.gemspec

#
```

La próxima vez que confirmes, el archivo desaparecerá y dejará de estar bajo seguimiento. Si ya habías modificado el archivo y lo tenías en el área de preparación, deberás forzar su eliminación con la opción `-f`. Ésta es una medida de seguridad para evitar la eliminación accidental de información que no ha sido

registrada en una instantánea, y que por tanto no podría ser recuperada.

Otra cosa que puede que quieras hacer es mantener el archivo en tu directorio de trabajo, pero eliminarlo de tu área de preparación. Dicho de otro modo, puede que quieras mantener el archivo en tu disco duro, pero interrumpir su seguimiento por parte de Git. Esto resulta particularmente útil cuando olvidaste añadir algo a tu archivo `.gitignore` y lo añadiste accidentalmente, como un archivo de log enorme, o un montón de archivos `.a`. Para hacer esto, usa la opción `--cached`:

```
$ git rm --cached readme.txt
```

El comando `git rm` acepta archivos, directorios, y patrones glob. Es decir, que podrías hacer algo así:

```
$ git rm log/*.log
```

Fijate en la barra hacia atrás (`\`) antes del `*`. Es necesaria debido a que Git hace su propia expansión de rutas, además de la expansión que hace tu shell. Este comando elimina todos los archivos con la extensión `.log` en el directorio `log/`. También puedes hacer algo así:

```
$ git rm \*~
```

Este comando elimina todos los archivos que terminan en `~`.

2.2.9. Moviendo archivos

A diferencia de muchos otros VCSs, Git no hace un seguimiento explícito del movimiento de archivos. Si renombas un archivo, en Git no se almacena ningún metadato que indique que lo has renombrado. Sin embargo, Git es suficientemente inteligente como para darse cuenta — trataremos el tema de la detección de movimiento de archivos un poco más adelante —.

Por tanto, es un poco desconcertante que Git tenga un comando `mv`. Si quieres renombrar un archivo en Git, puedes ejecutar algo así:

```
$ git mv file_from file_to
```

Y funciona perfectamente. De hecho, cuando ejecutas algo así y miras la salida del comando `status`, verás que Git lo considera un archivo renombrado:

```
$ git mv README.txt README
$ git status
# On branch master
# Your branch is ahead of 'origin/master' by 1 commit.
#
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       renamed:    README.txt -> README
#
```

Sin embargo, esto es equivalente a ejecutar algo así:

```
$ mv README.txt README
$ git rm README.txt
$ git add README
```

Git se da cuenta de que es un renombrado de manera implícita, así que no importa si renombas un archivo de este modo, o usando el comando `mv`. La única diferencia real es que `mv` es un comando en vez de tres — es más cómodo —. Y lo que es más importante, puedes usar cualquier herramienta para renombrar un archivo, y preocuparte de los `add` y `rm` más tarde, antes de confirmar.

© 2006-2022 uniwebsidad

[Contacto](#) [Aviso legal](#)

Recursos sobre:

[css](#)

[diseño](#)

[drupal](#)

[JavaScript](#)

[PHP](#)

[programación](#)

[Python](#)

[ruby](#)

[Symfony](#)

5.516 días online

Este sitio utiliza cookies propias y de terceros. Sigue navegando para aceptar nuestra [Política de Cookies](#) o [ajusta tu configuración](#).

ACEPTAR