



Git y GitHub

Carlos Vergara-Hernández

Viernes, 9 de junio de 2017

Área de Desigualdades en Salud, FISABIO

1. Antes de empezar...
2. Control de versiones
3. Primeros pasos con Git
4. Modificar el repositorio
5. Ramas de desarrollo
6. Git y GitHub
7. Opciones en GitHub

Antes de empezar...

1. Requisito: asegura que tienes instalado [Git](#).
 - Si necesitas ayuda para instalar Git, puedes consultar este [tutorial](#).

1. Requisito: asegura que tienes instalado [Git](#).
 - Si necesitas ayuda para instalar Git, puedes consultar este [tutorial](#).
2. Requisito: crea una cuenta en [GitHub](#), si es que no tenías.

1. Requisito: asegura que tienes instalado [Git](#).
 - Si necesitas ayuda para instalar Git, puedes consultar este [tutorial](#).
2. Requisito: crea una cuenta en [GitHub](#), si es que no tenías.
3. Opcional: instala una interfaz gráfica de usuario para [Git](#).
 - Si no te gusta trabajar en línea de comandos, las GUI's que recomiendo son [GitKraken](#) y [GitEye](#) (pero cualquiera valdría: p. ej., RStudio tiene una interfaz con Git bastante potable).
 - También puede resultar muy útil un buen procesador de textos compatible con Git. Os recomiendo probar [Atom](#) o [Sublime Text](#).

1. Requisito: asegura que tienes instalado [Git](#).
 - Si necesitas ayuda para instalar Git, puedes consultar este [tutorial](#).
2. Requisito: crea una cuenta en [GitHub](#), si es que no tenías.
3. Opcional: instala una interfaz gráfica de usuario para [Git](#).
 - Si no te gusta trabajar en línea de comandos, las GUI's que recomiendo son [GitKraken](#) y [GitEye](#) (pero cualquiera valdría: p. ej., RStudio tiene una interfaz con Git bastante potable).
 - También puede resultar muy útil un buen procesador de textos compatible con Git. Os recomiendo probar [Atom](#) o [Sublime Text](#).
4. Requisito: asegura que tienes instalado el paquete **devtools** de R.

Conexión Git-GitHub (1/2)

Es imprescindible que tengas bien introducidas en tu cuenta de GitHub las claves SSH.
Abre una terminal en cualquier directorio y sigue estos pasos:

Conexión Git-GitHub (1/2)

Es imprescindible que tengas bien introducidas en tu cuenta de GitHub las claves SSH. Abre una terminal en cualquier directorio y sigue estos pasos:

1. Ejecuta la instrucción `ssh-keygen -t rsa -b 4096 -C "tu_correoelectronico.com"`.

Conexión Git-GitHub (1/2)

Es imprescindible que tengas bien introducidas en tu cuenta de GitHub las claves SSH. Abre una terminal en cualquier directorio y sigue estos pasos:

1. Ejecuta la instrucción `ssh-keygen -t rsa -b 4096 -C "tu_correoelectronico.com"`.
 - Aprieta **Intro** las veces necesarias hasta que dejen de aparecer mensajes en la terminal. Sobre todo **NO INTRODUCAS UNA CONTRASEÑA**.

Conexión Git-GitHub (1/2)

Es imprescindible que tengas bien introducidas en tu cuenta de GitHub las claves SSH. Abre una terminal en cualquier directorio y sigue estos pasos:

1. Ejecuta la instrucción `ssh-keygen -t rsa -b 4096 -C "tu_correoelectronico.com"`.
 - Aprieta **Intro** las veces necesarias hasta que dejen de aparecer mensajes en la terminal. Sobre todo **NO INTRODUCAS UNA CONTRASEÑA**.
2. Ejecuta las siguientes instrucciones: primero `eval "$(ssh-agent -s)"`, seguido de `ssh-add ~/.ssh/id_rsa`.

Conexión Git-GitHub (1/2)

Es imprescindible que tengas bien introducidas en tu cuenta de GitHub las claves SSH. Abre una terminal en cualquier directorio y sigue estos pasos:

1. Ejecuta la instrucción `ssh-keygen -t rsa -b 4096 -C "tu_correoelectronico.com"`.
 - Aprieta **Intro** las veces necesarias hasta que dejen de aparecer mensajes en la terminal. Sobre todo **NO INTRODUCAS UNA CONTRASEÑA**.
2. Ejecuta las siguientes instrucciones: primero `eval "$(ssh-agent -s)"`, seguido de `ssh-add ~/.ssh/id_rsa`.
 - Si vieras que aparece un error en el último paso, deberás buscar el directorio `.ssh` en tu sistema y modificar la anterior ruta.

Conexión Git-GitHub (1/2)

Es imprescindible que tengas bien introducidas en tu cuenta de GitHub las claves SSH. Abre una terminal en cualquier directorio y sigue estos pasos:

1. Ejecuta la instrucción `ssh-keygen -t rsa -b 4096 -C "tu_correoelectronico.com"`.
 - Aprieta **Intro** las veces necesarias hasta que dejen de aparecer mensajes en la terminal. Sobre todo **NO INTRODUCAS UNA CONTRASEÑA**.
2. Ejecuta las siguientes instrucciones: primero `eval "$(ssh-agent -s)"`, seguido de `ssh-add ~/.ssh/id_rsa`.
 - Si vieras que aparece un error en el último paso, deberás buscar el directorio `.ssh` en tu sistema y modificar la anterior ruta.

Para finalizar esta configuración inicial, añade tu identidad a Git tecleando en una terminal, comprueba que se ha registrado satisfactoriamente y agrega la dirección de GitHub:

```
git config --global user.name "Carlos Vergara-Hernández"
git config --global user.email "vergara_car@gva.es"
git config user.name
git config user.email
ssh -T git@github.com
```

```
# Carlos Vergara-Hernández
# vergara_car@gva.es
```

Toca introducir la clave SSH que acabamos de generar en nuestra cuenta de GitHub:

Conexión Git-GitHub (2/2)

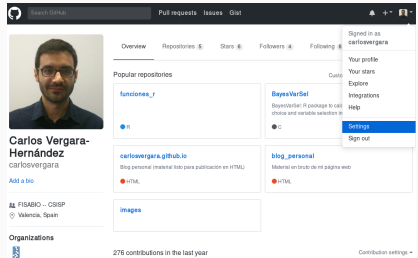
Toca introducir la clave SSH que acabamos de generar en nuestra cuenta de GitHub:

1. Ejecuta la instrucción `cat`
`~/ .ssh/id_rsa.pub` y copia el
contenido del archivo (o abre el archivo con
un editor de textos y cópialo a mano).

```
Enter file in which to save the key (/home/carlos/.ssh/id_rsa):
/home/carlos/.ssh/id_rsa already exists.
Overwrite (y/n)? y
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/carlos/.ssh/id_rsa.
Your public key has been saved in /home/carlos/.ssh/id_rsa.pub.
The key fingerprint is:
SHA256: IQQ24BZn/Tz8L4dSfL9JuMfcMsKvvgHMT+/Qr7gJZIMg vergara_car@gva.es
The key's randomart image is:
+----[RSA 4096]-----+
| .oo*.o. |
|.E=.oo. |
| o .o. =o |
| +.+.+. |
| oS+ + + |
| o .o= X + |
| ...o* = o |
| .o.oo o |
| =B+o. |
+----[SHA256]-----+
[carlos@csisp118 ~]$ cat ~/.ssh/id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAQCAIAFH50mAYY4htj64MvkYKHyme+yLa8GLs7yq05ze
gIowIC53fK4/VLWvEC/STyqyNmb8cEkZK6rNP0agu21v+HBSqznm31YK914rueoJwLd0bocQ3v00YS
UYQ3D0RQJbKrmDvqTFwzslrfZ1+h1KdSb5IEkw5yKwIvkmZ3ytPFE0XfR8S06F788oHuwwa7808TFqI
Navy8wm3jGJ7eP/suHkIng1KaZVNEb20FevvHoSusw41H0QpxK651N73xuN0KQ1bDA3H5r5eYmAWL
MqHqH660BLX1jlefyfj705oI75fnZu6lne8krFato30mZfrjgEv178j8cIZm3p+xx7B-d1Wm6TCq
Lavl6cV2tje2542PFGBE4jCM9xoc1r2r3SZfAolcy8BwtN7hdSCwLzhNaM5fC0f6yAjaq+H0m9qW3q
k2/QJYzzT1q5AyA2UHX0Ne7Da56bCK+DUvj5h8bxx/ENJoST0GPhNokd7L60ePT3bWeOfwBWBWJH1f
jRFZNG0W3JZd+ovb8UY4LD0VXXI8xTdnAtmKhYyigk5r123j5WHvms4YeiiT8J+qEjclmNveX/GbZ2h
LZA6rLPI3RZnu48g1YLZjdbZ9x4yvPdZLSSWmSBwg4s8gnNJ9yuqntIP4y+6vwSXLq8ySXXLmTTj1MC
b4Ds24jJ1uw== vergara_car@gva.es
[carlos@csisp118 ~]$
```

Toca introducir la clave SSH que acabamos de generar en nuestra cuenta de GitHub:

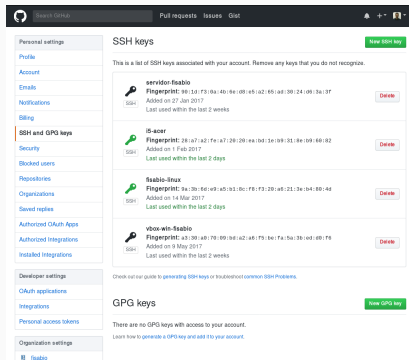
1. Ejecuta la instrucción `cat ~/.ssh/id_rsa.pub` y copia el contenido del archivo (o abre el archivo con un editor de textos y cópialo a mano).
2. Abre un navegador y ve a tu cuenta de GitHub. En la parte superior derecha, pincha en tu avatar y luego en *Settings*.



Conexión Git-GitHub (2/2)

Toca introducir la clave SSH que acabamos de generar en nuestra cuenta de GitHub:

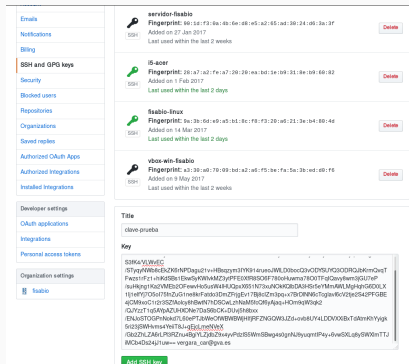
1. Ejecuta la instrucción `cat ~/.ssh/id_rsa.pub` y copia el contenido del archivo (o abre el archivo con un editor de textos y cópialo a mano).
2. Abre un navegador y ve a tu cuenta de GitHub. En la parte superior derecha, pincha en tu avatar y luego en *Settings*.
3. En el apartado *Personal settings*, pincha en *SSH and GPG keys*.



Conexión Git-GitHub (2/2)

Toca introducir la clave SSH que acabamos de generar en nuestra cuenta de GitHub:

1. Ejecuta la instrucción `cat ~/.ssh/id_rsa.pub` y copia el contenido del archivo (o abre el archivo con un editor de textos y cópialo a mano).
2. Abre un navegador y ve a tu cuenta de GitHub. En la parte superior derecha, pincha en tu avatar y luego en *Settings*.
3. En el apartado *Personal settings*, pincha en *SSH and GPG keys*.
4. En la sección *SSH keys*, haz clic en *New SSH key*, dale un título a la clave (por ejemplo: mi-ordenador-personal), pega la clave en el apartado *Key*, y haz clic en *Add SSH key* para terminar la operación.



Dado el carácter de estas sesiones, en la práctica procuraremos trabajar con RStudio o desde una GUI de Git, pero

Dado el carácter de estas sesiones, en la práctica procuraremos trabajar con RStudio o desde una GUI de Git, pero

- primero veremos el uso en línea de comandos, y
- en algunos casos habrá que usar la consola obligatoriamente.

Dado el carácter de estas sesiones, en la práctica procuraremos trabajar con RStudio o desde una GUI de Git, pero

- primero veremos el uso en línea de comandos, y
- en algunos casos habrá que usar la consola obligatoriamente.

Es aquí donde se presenta un problema, y es que Git usa comandos basados en Unix (Linux y Mac), con lo que los usuarios de Windows se verán algo perdidos al utilizar Git Bash. Esta transparencia va por vosotros. 😊

Dado el carácter de estas sesiones, en la práctica procuraremos trabajar con RStudio o desde una GUI de Git, pero

- primero veremos el uso en línea de comandos, y
- en algunos casos habrá que usar la consola obligatoriamente.

Es aquí donde se presenta un problema, y es que Git usa comandos basados en Unix (Linux y Mac), con lo que los usuarios de Windows se verán algo perdidos al utilizar Git Bash. Esta transparencia va por vosotros. 😊

Propósito	Comando	Ejemplo de uso	Descripción
Ver el directorio de trabajo	<code>pwd</code>	<code>pwd</code>	Imprime la ruta del directorio de trabajo.
Crear un directorio	<code>mkdir</code>	<code>mkdir curso</code>	Crea el directorio "curso" dentro del directorio de trabajo.
Cambiar de directorio	<code>cd</code>	<code>cd ~/Escritorio</code>	Cambia el directorio de trabajo al directorio "Escritorio".
Cambiar al directorio superior	<code>cd ..</code>	<code>cd ..</code>	Cambia el directorio de trabajo al directorio superior.
Mover un archivo o directorio	<code>mv</code>	<code>mv datos.csv curso/datos.csv</code>	Mueve "datos.csv" al directorio "curso/datos.csv".
Copiar un archivo o directorio	<code>cp</code>	<code>cp datos.csv curso/datos.csv</code>	Copia el archivo "datos.csv" en el directorio "curso/datos.csv".
Eliminar un archivo o directorio	<code>rm</code>	<code>rm curso/datos.csv</code>	Elimina el archivo "datos.csv" en el directorio "curso".
Imprimir texto en consola	<code>echo</code>	<code>echo "texto"</code>	Imprime "texto" en la consola.
Introducir texto en un archivo	<code>echo >></code>	<code>echo "texto" >> archivo</code>	Crear el documento "archivo" e introduce el texto especificado.
Obtener ayuda de un comando	<code>man</code>	<code>man git init</code>	Abre la ayuda para el comando <code>git init</code> .

Control de versiones

¿A quién no le ha pasado?

"FINAL".doc



FINAL.doc!



FINAL_rev.2.doc



FINAL_rev.6.COMMENTS.doc



FINAL_rev.8.comments5.
CORRECTIONS.doc



FINAL_rev.18.comments7.
corrections9.MORE.30.doc



FINAL_rev.22.comments49.
corrections.10. #@\$%WHYDID
ICOMETOGRADSCHOOL?????.doc

32966: Chalk © 2012

WWW.PHDCOMICS.COM

¿A quién no le ha pasado?

"FINAL".doc



FINAL.doc!



FINAL_rev.2.doc



FINAL_rev.6.COMMENTS.doc



FINAL_rev.8.comments5.
CORRECTIONS.doc



FINAL_rev.18.comments&7.
corrections9.MORE_30.doc

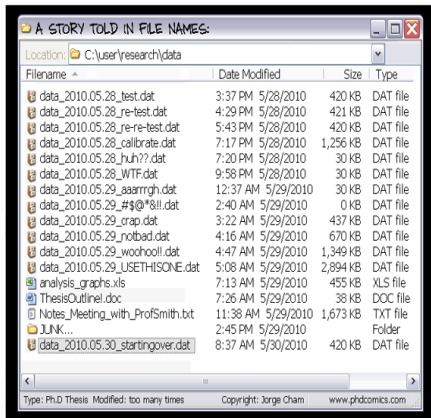


FINAL_rev.22.comments49.
corrections.10. #@\$%WHYDID
ICOMETOGRADSCHOOL?????.doc



32956 Chalk © 2012

WWW.PHDCOMICS.COM



El sistema de andar por casa:

El sistema de andar por casa:

- Copiar versiones de archivos y directorios dentro de un directorio cuyo nombre sea algo parecido a `./proyectoX_version7_06_09_2017`.

El sistema de andar por casa:

- Copiar versiones de archivos y directorios dentro de un directorio cuyo nombre sea algo parecido a `./proyectoX_version7_06_09_2017`.

El problema

Resulta fácil, sí. Pero... ¿y si olvidamos en qué directorio se está trabajando y se escriben contenidos en el lugar inapropiado?, ¿y si elegimos un nombre para el directorio poco afortunado y no localizamos la última versión?, ¿cómo se puede colaborar con otros desarrolladores?

El sistema de andar por casa:

- Copiar versiones de archivos y directorios dentro de un directorio cuyo nombre sea algo parecido a `./proyectoX_version7_06_09_2017`.

El problema

Resulta fácil, sí. Pero... ¿y si olvidamos en qué directorio se está trabajando y se escriben **contenidos en el lugar inapropiado**?, ¿y si elegimos un nombre para el directorio poco afortunado y **no localizamos la última versión**?, ¿cómo se puede **colaborar con otros desarrolladores**?

Definición de control de versiones

El sistema de andar por casa:

- Copiar versiones de archivos y directorios dentro de un directorio cuyo nombre sea algo parecido a `./proyectoX_version7_06_09_2017`.

El problema

Resulta fácil, sí. Pero... ¿y si olvidamos en qué directorio se está trabajando y se escriben **contenidos en el lugar inapropiado**?, ¿y si elegimos un nombre para el directorio poco afortunado y **no localizamos la última versión**?, ¿cómo se puede **colaborar con otros desarrolladores**?

En los inicios del desarrollo de software, esta dinámica de copiar y pegar versiones empezó a ser muy problemática y terminó desembocando de forma natural en el desarrollo de sistemas de control de versiones (VCS).

Definición de control de versiones

El sistema de andar por casa:

- Copiar versiones de archivos y directorios dentro de un directorio cuyo nombre sea algo parecido a `./proyectoX_version7_06_09_2017`.

El problema

Resulta fácil, sí. Pero... ¿y si olvidamos en qué directorio se está trabajando y se escriben **contenidos en el lugar inapropiado**?, ¿y si elegimos un nombre para el directorio poco afortunado y **no localizamos la última versión**?, ¿cómo se puede **colaborar con otros desarrolladores**?

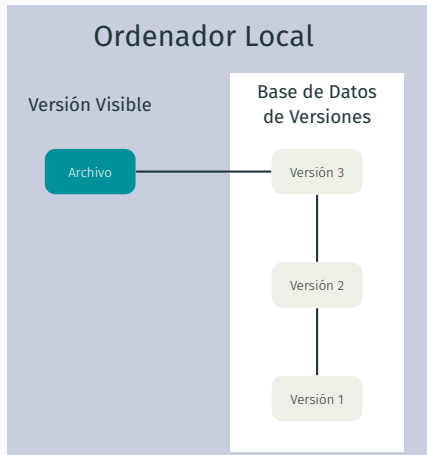
En los inicios del desarrollo de software, esta dinámica de copiar y pegar versiones empezó a ser muy problemática y terminó desembocando de forma natural en el desarrollo de sistemas de control de versiones (VCS).

Sistema de control de versiones

Sistema que registra cambios en un archivo o conjunto de archivos a lo largo del tiempo para poder recuperar versiones específicas en otro momento.

1. Local

Se almacenan versiones en una base de datos local y se tiene disponible la última por defecto.

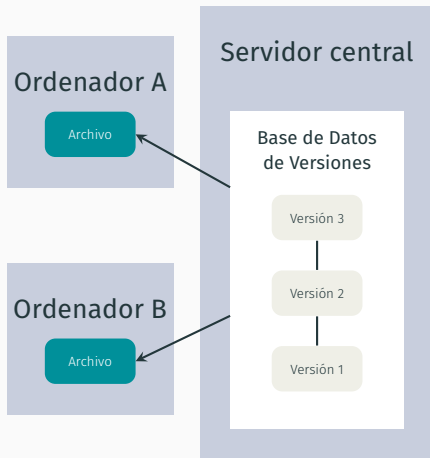


1. Local

Se almacenan versiones en una base de datos local y se tiene disponible la última por defecto.

2. Centralizado

Se traslada el almacenamiento a un servidor, de forma que se favorece la colaboración aunque solo el servidor tiene todas las versiones (los clientes solo tienen una).



1. Local

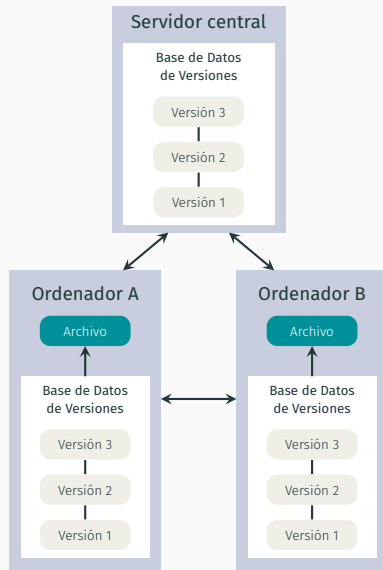
Se almacenan versiones en una base de datos local y se tiene disponible la última por defecto.

2. Centralizado

Se traslada el almacenamiento a un servidor, de forma que se favorece la colaboración aunque solo el servidor tiene todas las versiones (los clientes solo tienen una).

3. Distribuido

Los clientes almacenan todo el repositorio de versiones. Si el servidor se corrompe, cualquiera puede restablecerlo.



Ventajas de Git frente a otros VCS

Creado en 2005 como forma de colaborar en el desarrollo del kernel de Linux, Git presenta 4 diferencias fundamentales respecto a otros VCS:

Ventajas de Git frente a otros VCS

Creado en 2005 como forma de colaborar en el desarrollo del kernel de Linux, Git presenta 4 diferencias fundamentales respecto a otros VCS:

1. Mientras que otros VCS (o incluso Dropbox) guardan todos los cambios que afectan a los archivos controlados, Git solo almacena *imágenes* de momentos concretos, lo que permite ser mucho más selectivo.

Creado en 2005 como forma de colaborar en el desarrollo del kernel de Linux, Git presenta 4 diferencias fundamentales respecto a otros VCS:

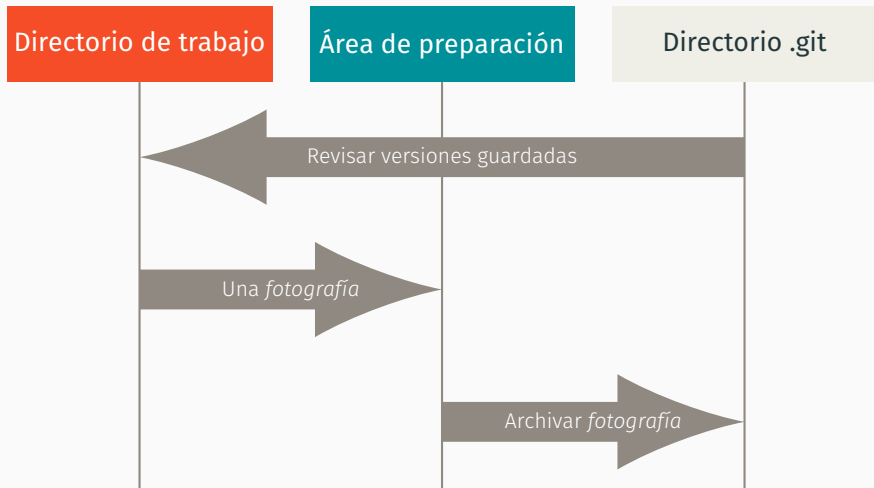
1. Mientras que otros VCS (o incluso Dropbox) guardan todos los cambios que afectan a los archivos controlados, Git solo almacena *imágenes* de momentos concretos, lo que permite ser mucho más selectivo.
2. La mayor parte de las operaciones que afectan al repositorio Git se realizan de manera local (p. ej., consultar historial, realizar un commit, agrupar commits o trabajo con ramas), aunque todas ellas pueden tener un soporte remoto en el servidor.

Creado en 2005 como forma de colaborar en el desarrollo del kernel de Linux, Git presenta 4 diferencias fundamentales respecto a otros VCS:

1. Mientras que otros VCS (o incluso Dropbox) guardan todos los cambios que afectan a los archivos controlados, Git solo almacena *imágenes* de momentos concretos, lo que permite ser mucho más selectivo.
2. La mayor parte de las operaciones que afectan al repositorio Git se realizan de manera local (p. ej., consultar historial, realizar un commit, agrupar commits o trabajo con ramas), aunque todas ellas pueden tener un soporte remoto en el servidor.
3. En Git todo funciona por suma de verificaciones y es refenciado por ese valor (SHA-1). Se trata de un valor de 40 caracteres hexadecimales (0-9, a-f) que identifican de manera unívoca cada operación de Git.

Creado en 2005 como forma de colaborar en el desarrollo del kernel de Linux, Git presenta 4 diferencias fundamentales respecto a otros VCS:

1. Mientras que otros VCS (o incluso Dropbox) guardan todos los cambios que afectan a los archivos controlados, Git solo almacena *imágenes* de momentos concretos, lo que permite ser mucho más selectivo.
2. La mayor parte de las operaciones que afectan al repositorio Git se realizan de manera local (p. ej., consultar historial, realizar un commit, agrupar commits o trabajo con ramas), aunque todas ellas pueden tener un soporte remoto en el servidor.
3. En Git todo funciona por suma de verificaciones y es refenciado por ese valor (SHA-1). Se trata de un valor de 40 caracteres hexadecimales (0-9, a-f) que identifican de manera unívoca cada operación de Git.
4. Generalmente el repositorio solo añade datos, de manera que resulta complicado perder información. P. ej., hasta una eliminación tiene una suma de verificación, y siempre se puede volver al commit previo a la eliminación y recuperar contenido. Esto es muy útil para establecer la trazabilidad de nuestro proyecto, de cara a aumentar su reproducibilidad.



Los archivos almacenados en un repositorio Git se organizan en tres estados distintos:

Los archivos almacenados en un repositorio Git se organizan en tres estados distintos:

1. Preparado (staged): un archivo ha sido marcado en la versión actual para ser almacenado en el próximo commit (se hace una *fotografía*).

Los archivos almacenados en un repositorio Git se organizan en tres estados distintos:

1. Preparado (staged): un archivo ha sido marcado en la versión actual para ser almacenado en el próximo commit (se hace una *fotografía*).
2. Modificado (modified): un archivo almacenado ha sufrido cambios pero todavía no ha sido preparado ni confirmado (hay cambios respecto a una versión previa).

Los archivos almacenados en un repositorio Git se organizan en tres estados distintos:

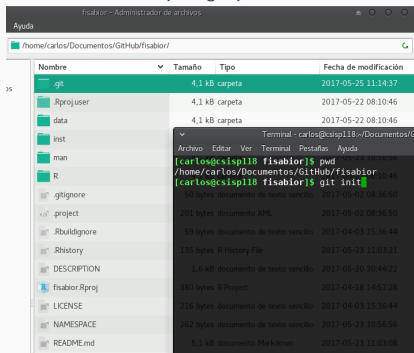
1. Preparado (staged): un archivo ha sido marcado en la versión actual para ser almacenado en el próximo commit (se hace una *fotografía*).
2. Modificado (modified): un archivo almacenado ha sufrido cambios pero todavía no ha sido preparado ni confirmado (hay cambios respecto a una versión previa).
3. Confirmado (committed): los datos están almacenados de forma segura en la base de datos local, el directorio `./ .git` (se archivan las *fotografía*).

Funcionamiento de Git

Los archivos almacenados en un repositorio Git se organizan en tres estados distintos:

1. Preparado (staged): un archivo ha sido marcado en la versión actual para ser almacenado en el próximo commit (se hace una *fotografía*).
2. Modificado (modified): un archivo almacenado ha sufrido cambios pero todavía no ha sido preparado ni confirmado (hay cambios respecto a una versión previa).
3. Confirmado (committed): los datos están almacenados de forma segura en la base de datos local, el directorio `./ .git` (se archivan las *fotografía*).

- Al iniciar un repositorio en el directorio raíz de nuestro proyecto, se crea el directorio `./ .git`, que es el lugar donde se almacenan metadatos y datos del proyecto.

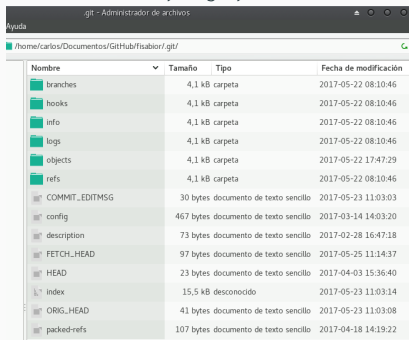


Funcionamiento de Git

Los archivos almacenados en un repositorio Git se organizan en tres estados distintos:

1. Preparado (staged): un archivo ha sido marcado en la versión actual para ser almacenado en el próximo commit (se hace una *fotografía*).
2. Modificado (modified): un archivo almacenado ha sufrido cambios pero todavía no ha sido preparado ni confirmado (hay cambios respecto a una versión previa).
3. Confirmado (committed): los datos están almacenados de forma segura en la base de datos local, el directorio `./ .git` (se archivan las *fotografía*).

- Al iniciar un repositorio en el directorio raíz de nuestro proyecto, se crea el directorio `./ .git`, que es el lugar donde se almacenan metadatos y datos del proyecto.
- Dentro del directorio `./ .git`, el área de preparación contiene información acerca de lo que se agregará en el próximo commit.



Nombre	Tamaño	Tipo	Fecha de modificación
branches	4,1 kB	carpeta	2017-05-22 08:10:46
hooks	4,1 kB	carpeta	2017-05-22 08:10:46
info	4,1 kB	carpeta	2017-05-22 08:10:46
logs	4,1 kB	carpeta	2017-05-22 08:10:46
objects	4,1 kB	carpeta	2017-05-22 17:47:29
refs	4,1 kB	carpeta	2017-05-22 08:10:46
COMMIT_EDITMSG	30 bytes	documento de texto sencillo	2017-05-23 11:03:03
config	467 bytes	documento de texto sencillo	2017-03-14 14:03:20
description	73 bytes	documento de texto sencillo	2017-02-28 16:47:18
FETCH_HEAD	97 bytes	documento de texto sencillo	2017-05-25 11:14:37
HEAD	23 bytes	documento de texto sencillo	2017-04-03 15:36:40
index	15,5 kB	desconocido	2017-05-23 11:03:14
ORIG_HEAD	41 bytes	documento de texto sencillo	2017-05-23 11:03:08
packed-refs	107 bytes	documento de texto sencillo	2017-04-18 14:19:22

Al iniciar un repositorio en un directorio, e ir añadiendo archivos a dicho directorio, Git detecta que no se está realizando un seguimiento de los mismos.

Al iniciar un repositorio en un directorio, e ir añadiendo archivos a dicho directorio, Git detecta que no se está realizando un seguimiento de los mismos.

1. Comenzar el seguimiento marcando archivos como staged.

Al iniciar un repositorio en un directorio, e ir añadiendo archivos a dicho directorio, Git detecta que no se está realizando un seguimiento de los mismos.

1. Comenzar el seguimiento marcando archivos como staged.
2. Realizar un commit de forma que se guarde una versión del estado de los archivos tal cual estaban al ser marcados como staged.

Al iniciar un repositorio en un directorio, e ir añadiendo archivos a dicho directorio, Git detecta que no se está realizando un seguimiento de los mismos.

1. Comenzar el seguimiento marcando archivos como staged.
2. Realizar un commit de forma que se guarde una versión del estado de los archivos tal cual estaban al ser marcados como staged.
3. Al modificar o borrar archivos, Git detecta los cambios respecto a la versión confirmada. Estos pueden ser marcados como staged cuando se desee y se puede realizar un nuevo commit.

Al iniciar un repositorio en un directorio, e ir añadiendo archivos a dicho directorio, Git detecta que no se está realizando un seguimiento de los mismos.

1. Comenzar el seguimiento marcando archivos como staged.
2. Realizar un commit de forma que se guarde una versión del estado de los archivos tal cual estaban al ser marcados como staged.
3. Al modificar o borrar archivos, Git detecta los cambios respecto a la versión confirmada. Estos pueden ser marcados como staged cuando se desee y se puede realizar un nuevo commit.
4. En cualquier momento se puede consultar una versión confirmada (committed) previa, o ver las diferencias entre un archivo y su versión actual editada.

Todos los comandos Git comienzan con (oh, sorpresa) **git**, seguido por la instrucción que corresponda según lo que se desee hacer y valores opcionales para la misma. P. ej.:

Todos los comandos Git comienzan con (oh, sorpresa) **git**, seguido por la instrucción que corresponda según lo que se desee hacer y valores opcionales para la misma. P. ej.:

1. Un comando sencillo.

Todos los comandos Git comienzan con (oh, sorpresa) **git**, seguido por la instrucción que corresponda según lo que se desee hacer y valores opcionales para la misma. P. ej.:

1. Un comando sencillo.

```
git init
```

Todos los comandos Git comienzan con (oh, sorpresa) **git**, seguido por la instrucción que corresponda según lo que se desee hacer y valores opcionales para la misma. P. ej.:

1. Un comando sencillo.

```
git init
```

2. Un comando dirigido a un archivo en concreto.

Todos los comandos Git comienzan con (oh, sorpresa) **git**, seguido por la instrucción que corresponda según lo que se desee hacer y valores opcionales para la misma. P. ej.:

1. Un comando sencillo.

```
git init
```

2. Un comando dirigido a un archivo en concreto.

```
git add archivo.txt
```


Todos los comandos Git comienzan con (oh, sorpresa) **git**, seguido por la instrucción que corresponda según lo que se desee hacer y valores opcionales para la misma. P. ej.:

1. Un comando sencillo.

```
git init
```

2. Un comando dirigido a un archivo en concreto.

```
git add archivo.txt
```

3. Un comando con una opción y un input.

Todos los comandos Git comienzan con (oh, sorpresa) **git**, seguido por la instrucción que corresponda según lo que se desee hacer y valores opcionales para la misma. P. ej.:

1. Un comando sencillo.

```
git init
```

2. Un comando dirigido a un archivo en concreto.

```
git add archivo.txt
```

3. Un comando con una opción y un input.

```
git commit -m "Un commit"
```

Todos los comandos Git comienzan con (oh, sorpresa) **git**, seguido por la instrucción que corresponda según lo que se desee hacer y valores opcionales para la misma. P. ej.:

1. Un comando sencillo.

```
git init
```

2. Un comando dirigido a un archivo en concreto.

```
git add archivo.txt
```

3. Un comando con una opción y un input.

```
git commit -m "Un commit"
```

4. Un comando con varias opciones.

Todos los comandos Git comienzan con (oh, sorpresa) **git**, seguido por la instrucción que corresponda según lo que se desee hacer y valores opcionales para la misma. P. ej.:

1. Un comando sencillo.

```
git init
```

2. Un comando dirigido a un archivo en concreto.

```
git add archivo.txt
```

3. Un comando con una opción y un input.

```
git commit -m "Un commit"
```

4. Un comando con varias opciones.

```
git log -n 4 --oneline --decorate --graph --all
```

Comandos Git de uso común

La mayor parte del tiempo solo utilizarás este conjunto de comandos. Por ello conviene tenerlos presentes.

Propósito	Comando y ejemplo de uso	Descripción
Iniciar un repositorio Git	<code>git init</code>	Crea un repositorio Git asociado al directorio de trabajo.
Comprobar el estado del repositorio	<code>git status</code>	Muestra el estado del repositorio.
Añadir un archivo al área de commit	<code>git add .</code>	Inicia el seguimiento de todos los archivos que no lo tuvieran.
Realizar un commit	<code>git commit -m "Un mensaje"</code>	Realiza un commit con el texto <i>"Un mensaje"</i> .
Ver el historial de commits	<code>git log --oneline --graph</code>	Muestra el historial de commits y un gráfico guía.
Ver cambios desde el último commit	<code>git diff</code>	—
Etiquetar o marcar una versión	<code>git tag "v1.0.0"</code>	Asocia la etiqueta "v1.0.0" con el último commit.
Mover o renombrar archivos	<code>git mv script.R r/script.R</code>	Mueve "script.R" al directorio "r/".
Crear o mostrar ramas	<code>git branch -a</code>	Muestra todas las ramas del repositorio (locales y remotas).
Navegar entre commits y ramas	<code>git checkout devel</code>	Cambia a la rama "devel".
Unir contenido entre ramas	<code>git merge devel</code>	Agrega el contenido de la rama "devel" a la rama actual.
Descargar contenido de repositorios remotos	<code>git pull origin</code>	Sincroniza con el remoto "origin".
Subir contenido a repositorios remotos	<code>git pull origin master</code>	Sincroniza con el remoto "origin" en la rama master.

Comandos Git de uso común

La mayor parte del tiempo solo utilizarás este conjunto de comandos. Por ello conviene tenerlos presentes.

Propósito	Comando y ejemplo de uso	Descripción
Iniciar un repositorio Git	<code>git init</code>	Crea un repositorio Git asociado al directorio de trabajo.
Comprobar el estado del repositorio	<code>git status</code>	Muestra el estado del repositorio.
Añadir un archivo al área de commit	<code>git add .</code>	Inicia el seguimiento de todos los archivos que no lo tuvieran.
Realizar un commit	<code>git commit -m "Un mensaje"</code>	Realiza un commit con el texto <i>"Un mensaje"</i> .
Ver el historial de commits	<code>git log --oneline --graph</code>	Muestra el historial de commits y un gráfico guía.
Ver cambios desde el último commit	<code>git diff</code>	—
Etiquetar o marcar una versión	<code>git tag "v1.0.0"</code>	Asocia la etiqueta "v1.0.0" con el último commit.
Mover o renombrar archivos	<code>git mv script.R r/script.R</code>	Mueve "script.R" al directorio "r/".
Crear o mostrar ramas	<code>git branch -a</code>	Muestra todas las ramas del repositorio (locales y remotas).
Navegar entre commits y ramas	<code>git checkout devel</code>	Cambia a la rama "devel".
Unir contenido entre rams	<code>git merge devel</code>	Agrega el contenido de la rama "devel" a la rama actual.
Descargar contenido de repositorios remotos	<code>git pull origin</code>	Sincroniza con el remoto "origin".
Subir contenido a repositorios remotos	<code>git pull origin master</code>	Sincroniza con el remoto "origin" en la rama master.

Keep Calm...

No te agobies: recuerda que hay interfaces de usuario (incluyendo RStudio) que permiten realizar la mayoría de estas tareas.

Primeros pasos con Git

Iniciar un repositorio (1/2)

Vamos a crear un repositorio Git en el directorio donde almacenamos el paquete que creamos ayer (**pintamapas**), para lo cual tenemos que abrir una terminal y cambiar el directorio de trabajo (o ir al directorio y abrir allí la terminal), que en mi caso es el directorio `~/pintamapas`.

Iniciar un repositorio (1/2)

Vamos a crear un repositorio Git en el directorio donde almacenamos el paquete que creamos ayer (**pintamapas**), para lo cual tenemos que abrir una terminal y cambiar el directorio de trabajo (o ir al directorio y abrir allí la terminal), que en mi caso es el directorio `~/pintamapas`.

1. El primer paso es cambiar al directorio de trabajo donde tenemos al mismo, y asegurarse de ello.

```
cd ~/pintamapas  
pwd
```

```
# /home/carlos/pintamapas
```

Iniciar un repositorio (1/2)

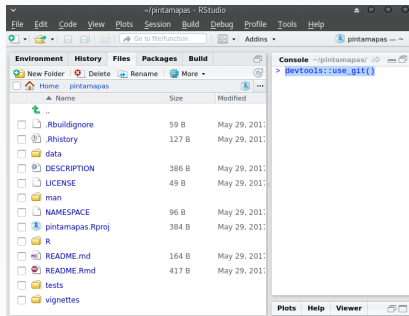
Vamos a crear un repositorio Git en el directorio donde almacenamos el paquete que creamos ayer (**pintamapas**), para lo cual tenemos que abrir una terminal y cambiar el directorio de trabajo (o ir al directorio y abrir allí la terminal), que en mi caso es el directorio `~/pintamapas`.

1. El primer paso es cambiar al directorio de trabajo donde tenemos al mismo, y asegurarse de ello.

```
cd ~/pintamapas  
pwd
```

```
# /home/carlos/pintamapas
```

- Una forma visual de comenzar es abriendo el proyecto de RStudio del paquete y ejecutando `devtools::use_git()`.



Iniciar un repositorio (1/2)

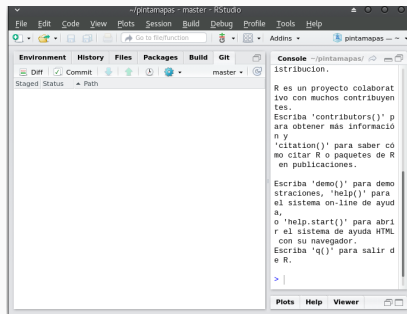
Vamos a crear un repositorio Git en el directorio donde almacenamos el paquete que creamos ayer (**pintamapas**), para lo cual tenemos que abrir una terminal y cambiar el directorio de trabajo (o ir al directorio y abrir allí la terminal), que en mi caso es el directorio `~/pintamapas`.

1. El primer paso es cambiar al directorio de trabajo donde tenemos al mismo, y asegurarse de ello.

```
cd ~/pintamapas  
pwd
```

```
# /home/carlos/pintamapas
```

- Una forma visual de comenzar es abriendo el proyecto de RStudio del paquete y ejecutando `devtools::use_git()`.
- Al reiniciar el proyecto aparece una pestaña *Git* en uno de los paneles (puede variar según configuración de RStudio).



2. Una forma alternativa de iniciar un repositorio es usando el comando `git init` en la terminal, lo que crea el repositorio en forma de un directorio oculto que almacenará toda la información relativa al mismo.

```
git init
```

```
# Initialized empty Git repository in /home/carlos/pintamapas/.git/
```

2. Una forma alternativa de iniciar un repositorio es usando el comando `git init` en la terminal, lo que crea el repositorio en forma de un directorio oculto que almacenará toda la información relativa al mismo.

```
git init
```

```
# Initialized empty Git repository in /home/carlos/pintamapas/.git/
```

3. Ahora utilizamos el comando `git status` para ver el estado del repositorio (debe estar vacío, pues acabamos de crearlo).

```
git status
```

```
# On branch master
```

```
#
```

```
# Initial commit
```

```
#
```

```
# nothing to commit (create/copy files and use "git add" to track)
```

Agregando contenido (1/3)

Ya estamos listos para que Git empiece a monitorizar nuestro trabajo. Vamos a crear dos archivos de texto, un documento Markdown y otro en texto plano. Esto podemos hacerlo desde un paradigma *apuntar-y-clic*, o desde la línea de comandos, que es lo que haremos.

Ya estamos listos para que Git empiece a monitorizar nuestro trabajo. Vamos a crear dos archivos de texto, un documento Markdown y otro en texto plano. Esto podemos hacerlo desde un paradigma *apuntar-y-clic*, o desde la línea de comandos, que es lo que haremos.

- La forma de hacerlo desde la terminal es imprimiendo un texto mediante **echo** (parecido a nuestro **print()** de R), y asignando el resultado (el texto) a un archivo:

Agregando contenido (1/3)

Ya estamos listos para que Git empiece a monitorizar nuestro trabajo. Vamos a crear dos archivos de texto, un documento Markdown y otro en texto plano. Esto podemos hacerlo desde un paradigma *apuntar-y-clic*, o desde la línea de comandos, que es lo que haremos.

- La forma de hacerlo desde la terminal es imprimiendo un texto mediante **echo** (parecido a nuestro **print()** de R), y asignando el resultado (el texto) a un archivo:

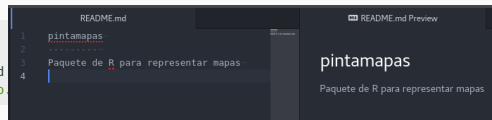
```
echo "pintamapas  
-----  
Paquete de R para representar mapas" >> README.md  
echo "Esto es un archivo de prueba en el proyecto." >> prueba.txt
```


Agregando contenido (1/3)

Ya estamos listos para que Git empiece a monitorizar nuestro trabajo. Vamos a crear dos archivos de texto, un documento Markdown y otro en texto plano. Esto podemos hacerlo desde un paradigma *apuntar-y-clic*, o desde la línea de comandos, que es lo que haremos.

- La forma de hacerlo desde la terminal es imprimiendo un texto mediante **echo** (parecido a nuestro **print()** de R), y asignando el resultado (el texto) a un archivo:

```
echo "pintamapas
-----
Paquete de R para representar mapas" >> README.md
echo "Esto es un archivo de prueba en el proyecto"
```



Agregando contenido (1/3)

Ya estamos listos para que Git empiece a monitorizar nuestro trabajo. Vamos a crear dos archivos de texto, un documento Markdown y otro en texto plano. Esto podemos hacerlo desde un paradigma *apuntar-y-clic*, o desde la línea de comandos, que es lo que haremos.

- La forma de hacerlo desde la terminal es imprimiendo un texto mediante **echo** (parecido a nuestro **print()** de R), y asignando el resultado (el texto) a un archivo:

```
echo "pintamapas  
-----  
Paquete de R para representar mapas" >> README.md  
echo "Esto es un archivo de prueba en el proyecto." >> prueba.txt
```

¿Sabrá Git lo que hemos hecho? Vamos a consultar el estado del repositorio.

Agregando contenido (1/3)

Ya estamos listos para que Git empiece a monitorizar nuestro trabajo. Vamos a crear dos archivos de texto, un documento Markdown y otro en texto plano. Esto podemos hacerlo desde un paradigma *apuntar-y-clic*, o desde la línea de comandos, que es lo que haremos.

- La forma de hacerlo desde la terminal es imprimiendo un texto mediante **echo** (parecido a nuestro **print()** de R), y asignando el resultado (el texto) a un archivo:

```
echo "pintamapas
-----
Paquete de R para representar mapas" >> README.md
echo "Esto es un archivo de prueba en el proyecto." >> prueba.txt
```

¿Sabrá Git lo que hemos hecho? Vamos a consultar el estado del repositorio.

```
git status
```

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

Agregando contenido (1/3)

Ya estamos listos para que Git empiece a monitorizar nuestro trabajo. Vamos a crear dos archivos de texto, un documento Markdown y otro en texto plano. Esto podemos hacerlo desde un paradigma *apuntar-y-clic*, o desde la línea de comandos, que es lo que haremos.

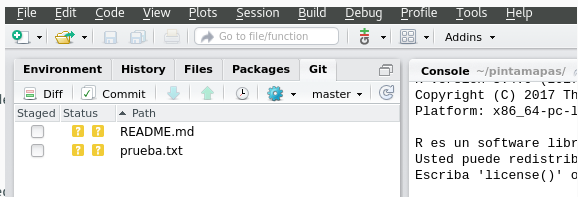
- La forma de hacerlo desde la terminal es imprimiendo un texto mediante **echo** (parecido a nuestro **print()** de R), y asignando el resultado (el texto) a un archivo:

```
echo "pintamapas
-----
Paquete de R para representar mapas" >> README.md
echo "Esto es un archivo de prueba en el proyecto." >> prueba.txt
```

¿Sabrá Git lo que hemos hecho? Vamos a consultar el estado del repositorio.

```
git status
```

```
# On branch master
#
# Initial commit
#
# Untracked files:
#   (use "git add <file>..." to include)
#
#   README.md
#   prueba.txt
#
# nothing added to commit but untracked
```



Agregando contenido (2/3)

Hacemos caso de Git y utilizamos `git add` para preparar el archivo `README.txt`.

Agregando contenido (2/3)

Hacemos caso de Git y utilizamos `git add` para preparar el archivo `README.txt`.

```
git add README.md
```

Agregando contenido (2/3)

Hacemos caso de Git y utilizamos `git add` para preparar el archivo `README.txt`.

```
git add README.md
```

De nuevo parece que no ha pasado nada. Veamos el estado del repositorio:

Agregando contenido (2/3)

Hacemos caso de Git y utilizamos `git add` para preparar el archivo `README.txt`.

```
git add README.md
```

De nuevo parece que no ha pasado nada. Veamos el estado del repositorio:

```
git status
```

```
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#   new file:   README.md
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#   prueba.txt
```


Agregando contenido (2/3)

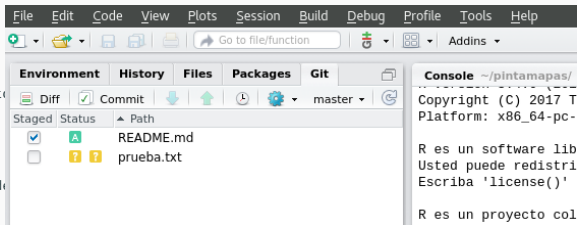
Hacemos caso de Git y utilizamos `git add` para preparar el archivo `README.txt`.

```
git add README.md
```

De nuevo parece que no ha pasado nada. Veamos el estado del repositorio:

```
git status
```

```
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to
#    remove staged file(s))
#
#   new file:   README.md
#
# Untracked files:
#   (use "git add <file>..." to include
#    in what will be committed)
#
#   prueba.txt
```



Agregando contenido (2/3)

Hacemos caso de Git y utilizamos `git add` para preparar el archivo `README.txt`.

```
git add README.md
```

De nuevo parece que no ha pasado nada. Veamos el estado del repositorio:

```
git status
```

```
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#   new file:   README.md
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#   prueba.txt
```

¡Perfecto! Ahora tenemos dos mensajes:

1. cambios preparados para ser confirmados (`README.md`),
2. archivos sin seguimiento (`prueba.txt`).

Agregando contenido (2/3)

Hacemos caso de Git y utilizamos `git add` para preparar el archivo `README.txt`.

```
git add README.md
```

De nuevo parece que no ha pasado nada. Veamos el estado del repositorio:

```
git status
```

```
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#   new file:   README.md
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#   prueba.txt
```

¡Perfecto! Ahora tenemos dos mensajes:

1. cambios preparados para ser confirmados (`README.md`),
2. archivos sin seguimiento (`prueba.txt`).

Las apariencias engañan

Puede parecer un coñazo que Git no inicie por su cuenta el seguimiento de los archivos, pero en realidad es una de sus grandes ventajas.

Agregando contenido (2/3)

Hacemos caso de Git y utilizamos `git add` para preparar el archivo `README.txt`.

```
git add README.md
```

De nuevo parece que no ha pasado nada. Veamos el estado del repositorio:

```
git status
```

```
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#   new file:   README.md
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#   prueba.txt
```

¡Perfecto! Ahora tenemos dos mensajes:

1. cambios preparados para ser confirmados (`README.md`),
2. archivos sin seguimiento (`prueba.txt`).

Las apariencias engañan

Puede parecer un coñazo que Git no inicie por su cuenta el seguimiento de los archivos, pero en realidad **es una de sus grandes ventajas**.

Agregando contenido (3/3)

Aún nos queda un archivo por preparar, y ahora vamos a emplear una posibilidad del comando `git add`: marcar como staged todo el contenido nuevo o modificado del directorio de trabajo.

Agregando contenido (3/3)

Aún nos queda un archivo por preparar, y ahora vamos a emplear una posibilidad del comando `git add`: marcar como staged todo el contenido nuevo o modificado del directorio de trabajo.

Cuidado

Muchas veces no interesa llevar un seguimiento de archivos basura (como los generados por \LaTeX , los que acompañan a algunas figuras o código compilado en C++), así que habitualmente bastará con marcar el archivo que contiene las instrucciones para crear los documentos finales, y no esos propios documentos.

Agregando contenido (3/3)

Aún nos queda un archivo por preparar, y ahora vamos a emplear una posibilidad del comando `git add`: marcar como staged todo el contenido nuevo o modificado del directorio de trabajo.

Cuidado

Muchas veces no interesa llevar un seguimiento de archivos basura (como los generados por \LaTeX , los que acompañan a algunas figuras o código compilado en C++), así que habitualmente bastará con marcar el archivo que contiene las instrucciones para crear los documentos finales, y no esos propios documentos.

```
git add .
```

Agregando contenido (3/3)

Aún nos queda un archivo por preparar, y ahora vamos a emplear una posibilidad del comando `git add`: marcar como staged todo el contenido nuevo o modificado del directorio de trabajo.

Cuidado

Muchas veces no interesa llevar un seguimiento de archivos basura (como los generados por \LaTeX , los que acompañan a algunas figuras o código compilado en C++), así que habitualmente bastará con marcar el archivo que contiene las instrucciones para crear los documentos finales, y no esos propios documentos.

```
git add .
```

Parece que no ha pasado nada (*one more time*), pero... . . .

```
git status
```

```
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#   new file:   README.md
#   new file:   prueba.txt
```


Agregando contenido (3/3)

Aún nos queda un archivo por preparar, y ahora vamos a emplear una posibilidad del comando `git add`: marcar como staged todo el contenido nuevo o modificado del directorio de trabajo.

Cuidado

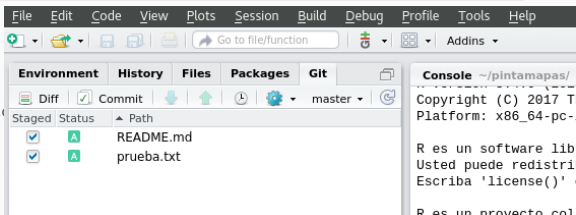
Muchas veces no interesa llevar un seguimiento de archivos basura (como los generados por \LaTeX , los que acompañan a algunas figuras o código compilado en C++), así que habitualmente bastará con marcar el archivo que contiene las instrucciones para crear los documentos finales, y no esos propios documentos.

```
git add .
```

Parece que no ha pasado nada (*one more time*), pero... .

```
git status
```

```
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to
#    #
#    new file:   README.md
#    new file:   prueba.txt
```



Agregando contenido (3/3)

Aún nos queda un archivo por preparar, y ahora vamos a emplear una posibilidad del comando `git add`: marcar como staged todo el contenido nuevo o modificado del directorio de trabajo.

Cuidado

Muchas veces no interesa llevar un seguimiento de archivos basura (como los generados por \LaTeX , los que acompañan a algunas figuras o código compilado en C++), así que habitualmente bastará con marcar el archivo que contiene las instrucciones para crear los documentos finales, y no esos propios documentos.

```
git add .
```

Parece que no ha pasado nada (*one more time*), pero... . . .

```
git status
```

```
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#   new file:   README.md
#   new file:   prueba.txt
```

¡Bien! Ahora en el repositorio solo tenemos cambios para ser confirmados, los que serán grabados en la próxima fotografía que Git haga del contenido.

Modificando contenido

Pero... ¿qué pasa si se modifica un archivo marcado como preparado pero todavía no ha sido confirmado?

Modificando contenido

Pero... ¿qué pasa si se modifica un archivo marcado como preparado pero todavía no ha sido confirmado?

Agregamos una nueva línea al documento `README.md` y vemos el estado del repositorio.

Modificando contenido

Pero... ¿qué pasa si se modifica un archivo marcado como preparado pero todavía no ha sido confirmado?

Agregamos una nueva línea al documento `README.md` y vemos el estado del repositorio.

```
echo "Este paquete funciona gracias a la funcion \`pintamapas()\`" >> README.md
git status
```

```
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#   new file:   README.md
#   new file:   prueba.txt
#
# 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:   README.md
```

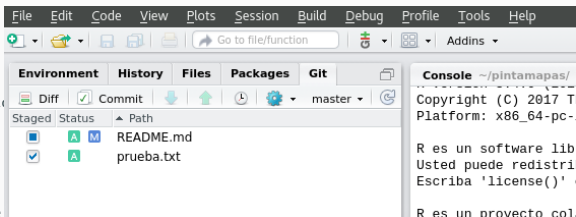
Modificando contenido

Pero... ¿qué pasa si se modifica un archivo marcado como preparado pero todavía no ha sido confirmado?

Agregamos una nueva línea al documento `README.md` y vemos el estado del repositorio.

```
echo "Este paquete funciona gracias a la funcion \`pintamapas()\`" >> README.md
git status
```

```
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to remove staged file(s))
#
#       new file:   README.md
#       new file:   prueba.txt
#
# 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:   README.md
```



Modificando contenido

Pero... ¿qué pasa si se modifica un archivo marcado como preparado pero todavía no ha sido confirmado?

Agregamos una nueva línea al documento `README.md` y vemos el estado del repositorio.

```
echo "Este paquete funciona gracias a la funcion \`pintamapas()\`" >> README.md
git status
```

```
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#   new file:   README.md
#   new file:   prueba.txt
#
# 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:   README.md
```

¿Cómo es posible que aparezca en los dos apartados?

Modificando contenido

Pero... ¿qué pasa si se modifica un archivo marcado como preparado pero todavía no ha sido confirmado?

Agregamos una nueva línea al documento `README.md` y vemos el estado del repositorio.

```
echo "Este paquete funciona gracias a la funcion \`pintamapas()\`" >> README.md
git status
```

```
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#   new file:   README.md
#   new file:   prueba.txt
#
# 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:   README.md
```

¿Cómo es posible que aparezca en los dos apartados?

Las tres áreas

Git solo confirma los archivos tal y como están en el área de preparación, no en nuestro directorio de trabajo.

Nuestro primer commit

Vamos a añadir la última modificación al área de preparación, de forma que será confirmada en nuestro primer commit y este sea un fiel reflejo de nuestro directorio de trabajo.

Nuestro primer commit

Vamos a añadir la última modificación al área de preparación, de forma que será confirmada en nuestro primer commit y este sea un fiel reflejo de nuestro directorio de trabajo.

```
git add .
```

Nuestro primer commit

Vamos a añadir la última modificación al área de preparación, de forma que será confirmada en nuestro primer commit y este sea un fiel reflejo de nuestro directorio de trabajo.

```
git add .
```

Ya estamos listos para la confirmación. Hay dos formas de hacerla:

1. tecleando `git commit`, lo que abrirá un editor de textos para que introduzcamos un mensaje de confirmación.
2. introduciendo el mensaje de confirmación junto con la propia instrucción (mensajes cortos), gracias a la opción `-m`.

Nuestro primer commit

Vamos a añadir la última modificación al área de preparación, de forma que será confirmada en nuestro primer commit y este sea un fiel reflejo de nuestro directorio de trabajo.

```
git add .
```

Ya estamos listos para la confirmación. Hay dos formas de hacerla:

1. tecleando `git commit`, lo que abrirá un editor de textos para que introduzcamos un mensaje de confirmación.
2. introduciendo el mensaje de confirmación junto con la propia instrucción (mensajes cortos), gracias a la opción `-m`.

```
git commit -m "Primer commit: se crean archivos README.md y prueba.txt"
```

```
# [master (root-commit) a0a792d] Primer commit: se crean archivos README.md y prueba.txt
# 2 files changed, 5 insertions(+)
# create mode 100644 README.md
# create mode 100644 prueba.txt
```

Nuestro primer commit

Vamos a añadir la última modificación al área de preparación, de forma que será confirmada en nuestro primer commit y este sea un fiel reflejo de nuestro directorio de trabajo.

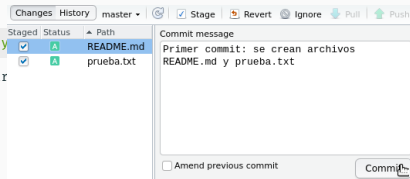
```
git add .
```

Ya estamos listos para la confirmación. Hay dos formas de hacerla:

1. tecleando `git commit`, lo que abrirá un editor de textos para que introduzcamos un mensaje de confirmación.
2. introduciendo el mensaje de confirmación junto con la propia instrucción (mensajes cortos), gracias a la opción `-m`.

```
git commit -m "Primer commit: se crean archivos README.md y prueba.txt"
```

```
# [master (root-commit) a0a792d] Primer commit: se crean ar
# 2 files changed, 5 insertions(+)
# create mode 100644 README.md
# create mode 100644 prueba.txt
```



Nuestro primer commit

Vamos a añadir la última modificación al área de preparación, de forma que será confirmada en nuestro primer commit y este sea un fiel reflejo de nuestro directorio de trabajo.

```
git add .
```

Ya estamos listos para la confirmación. Hay dos formas de hacerla:

1. tecleando `git commit`, lo que abrirá un editor de textos para que introduzcamos un mensaje de confirmación.
2. introduciendo el mensaje de confirmación junto con la propia instrucción (mensajes cortos), gracias a la opción `-m`.

```
git commit -m "Primer commit: se crean archivos README.md y prueba.txt"
```

```
# [master (root-commit) a0a792d] Primer commit: se crean archivos README.md y prueba.txt
# 2 files changed, 5 insertions(+)
# create mode 100644 README.md
# create mode 100644 prueba.txt
```

El mensaje nos devuelve información útil, como los archivos afectados y el número de líneas de código que incorpora o elimina (en este caso ninguna es eliminada).

Nuestro primer commit

Vamos a añadir la última modificación al área de preparación, de forma que será confirmada en nuestro primer commit y este sea un fiel reflejo de nuestro directorio de trabajo.

```
git add .
```

Ya estamos listos para la confirmación. Hay dos formas de hacerla:

1. tecleando `git commit`, lo que abrirá un editor de textos para que introduzcamos un mensaje de confirmación.
2. introduciendo el mensaje de confirmación junto con la propia instrucción (mensajes cortos), gracias a la opción `-m`.

```
git commit -m "Primer commit: se crean archivos README.md y prueba.txt"
```

```
# [master (root-commit) a0a792d] Primer commit: se crean archivos README.md y prueba.txt
# 2 files changed, 5 insertions(+)
# create mode 100644 README.md
# create mode 100644 prueba.txt
```

El mensaje nos devuelve información útil, como los archivos afectados y el número de líneas de código que incorpora o elimina (en este caso ninguna es eliminada).

Este comando (como muchos otros) tiene varias opciones, aunque nosotros solo utilizaremos unas pocas. Si quieres consultarlas prueba a acceder a la ayuda (`man git commit`).

Seguimiento de commits (1/2)

Una forma fácil de consultar el historial de commits es mediante el registro, accesible con `git log`.

Seguimiento de commits (1/2)

Una forma fácil de consultar el historial de commits es mediante el registro, accesible con `git log`.

```
git log
```

```
# commit a0a792d1dcf1c51bde948d3eb3a81112dae882fd
# Author: Carlos Vergara-Hernández <carlos.vergara@uv.es>
# Date:   Fri Jun 2 19:15:16 2017 +0200
#
#     Primer commit: se crean archivos README.md y prueba.txt
```

Seguimiento de commits (1/2)

Una forma fácil de consultar el historial de commits es mediante el registro, accesible con `git log`.

```
git log
```

```
# commit a0a792d1dcf1c51bde948d3eb3a81112dae882fd
```

```
# Auth Changes History master (all commits) Search Pull
```

# Date	Subject	Author	Date	SHA
#	a0a792d1dcf1c51bde948d3eb3a81112dae882fd Primer commit: se crean archivos README.md y prueba.txt	Carlos Vergara-Hernández <carlos.vergara@	2017-06-02	a8e7b291
#	Primer commit: se crean archivos README.md y prueba.txt			

Seguimiento de commits (1/2)

Una forma fácil de consultar el historial de commits es mediante el registro, accesible con `git log`.

```
git log
```

```
# commit a0a792d1dcf1c51bde948d3eb3a81112dae882fd
# Author: Carlos Vergara-Hernández <carlos.vergara@uv.es>
# Date:   Fri Jun 2 19:15:16 2017 +0200
#
#     Primer commit: se crean archivos README.md y prueba.txt
```

Ahora sabemos quién realizó el último commit y cuándo lo hizo... ¡aunque en realidad solo tenemos un commit! Vamos a añadir algo más de contenido (copiar un archivo y añadir contenido a otro, con un commit por medio):

Seguimiento de commits (1/2)

Una forma fácil de consultar el historial de commits es mediante el registro, accesible con `git log`.

```
git log
```

```
# commit a0a792d1dcf1c51bde948d3eb3a81112dae882fd
# Author: Carlos Vergara-Hernández <carlos.vergara@uv.es>
# Date:   Fri Jun 2 19:15:16 2017 +0200
#
#    Primer commit: se crean archivos README.md y prueba.txt
```

Ahora sabemos quién realizó el último commit y cuándo lo hizo... ¡aunque en realidad solo tenemos un commit! Vamos a añadir algo más de contenido (copiar un archivo y añadir contenido a otro, con un commit por medio):

```
mkdir -p inst/docs && cp README.md inst/docs/README.md
git add "*.md" # git add también acepta patrones.
git commit -m "Segundo commit: se añade copia de README.md en inst/docs/"
echo "Las dependencias del paquete son \`sp\` y \`RColorBrewer\`" >> README.md
git status
```

```
# [master 5906fbf] Segundo commit: se añade copia de README.md en inst/docs/
# 1 file changed, 4 insertions(+)
# create mode 100644 inst/docs/README.md
# 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:   README.md
#
# no changes added to commit (use "git add" and/or "git commit -a")
```

Tenemos un archivo en seguimiento que ha sufrido modificaciones desde el último commit. ¿Hay alguna forma de saber cuáles han sido?

Tenemos un archivo en seguimiento que ha sufrido modificaciones desde el último commit. ¿Hay alguna forma de saber cuáles han sido?

```
git diff
```

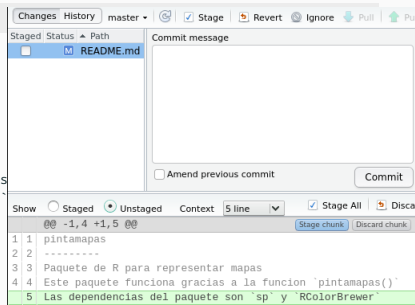
```
# diff --git a/README.md b/README.md
# index 69f2535..218ee92 100644
# --- a/README.md
# +++ b/README.md
# @@ -2,3 +2,4 @@ pintamapas
# -----
# Paquete de R para representar mapas
# Este paquete funciona gracias a la funcion `pintamapas()`
# +Las dependencias del paquete son `sp` y `RColorBrewer`
```

Seguimiento de commits (2/2)

Tenemos un archivo en seguimiento que ha sufrido modificaciones desde el último commit. ¿Hay alguna forma de saber cuáles han sido?

git diff

```
# diff --git a/README.md b/README.md
# index 69f2535..218ee92 100644
# --- a/README.md
# +++ b/README.md
# @@ -2,3 +2,4 @@ pintamapas
# -----
#  Paquete de R para representar mapas
#  Este paquete funciona gracias a la funcion `pintamapas`
# +Las dependencias del paquete son `sp` y `RColorBrewer`
```



Tenemos un archivo en seguimiento que ha sufrido modificaciones desde el último commit. ¿Hay alguna forma de saber cuáles han sido?

```
git diff
```

```
# diff --git a/README.md b/README.md
# index 69f2535..218ee92 100644
# --- a/README.md
# +++ b/README.md
# @@ -2,3 +2,4 @@ pintamapas
# -----
# Paquete de R para representar mapas
# Este paquete funciona gracias a la funcion `pintamapas()`
# +Las dependencias del paquete son `sp` y `RColorBrewer`
```

De forma resumida, **git diff** nos dice los cambios que ha sufrido un archivo. Este comando también tiene varias opciones (como comparar la versión de nuestro directorio de trabajo con la almacenada en un commit concreto, y no el último).

Seguimiento de commits (2/2)

Tenemos un archivo en seguimiento que ha sufrido modificaciones desde el último commit. ¿Hay alguna forma de saber cuáles han sido?

```
git diff
```

```
# diff --git a/README.md b/README.md
# index 69f2535..218ee92 100644
# --- a/README.md
# +++ b/README.md
# @@ -2,3 +2,4 @@ pintamapas
# -----
# Paquete de R para representar mapas
# Este paquete funciona gracias a la funcion `pintamapas()`
# +Las dependencias del paquete son `sp` y `RColorBrewer`
```

De forma resumida, **git diff** nos dice los cambios que ha sufrido un archivo. Este comando también tiene varias opciones (como comparar la versión de nuestro directorio de trabajo con la almacenada en un commit concreto, y no el último).

Tenemos un par de commits, así que vamos a consultar el registro a ver cómo queda, empleando algunas opciones chulas (un poco más adelante nos serán de gran ayuda):

Seguimiento de commits (2/2)

Tenemos un archivo en seguimiento que ha sufrido modificaciones desde el último commit. ¿Hay alguna forma de saber cuáles han sido?

```
git diff
```

```
# diff --git a/README.md b/README.md
# index 69f2535..218ee92 100644
# --- a/README.md
# +++ b/README.md
# @@ -2,3 +2,4 @@ pintamapas
# -----
# Paquete de R para representar mapas
# Este paquete funciona gracias a la funcion `pintamapas()`
# +Las dependencias del paquete son `sp` y `RColorBrewer`
```

De forma resumida, **git diff** nos dice los cambios que ha sufrido un archivo. Este comando también tiene varias opciones (como comparar la versión de nuestro directorio de trabajo con la almacenada en un commit concreto, y no el último).

Tenemos un par de commits, así que vamos a consultar el registro a ver cómo queda, empleando algunas opciones chulas (un poco más adelante nos serán de gran ayuda):

```
git log --oneline --decorate --graph
```

```
# * 5906fbf (HEAD -> master) Segundo commit: se añade copia de README.md en inst/docs/
# * a0a792d Primer commit: se crean archivos README.md y prueba.txt
```

Ya hemos visto que con la opción **-m** podemos añadir directamente un mensaje en el commit. Adicionalmente, con la opción **-a** podemos saltarnos el paso previo (**git add .**) y marcar como staged aquellos archivos modificados y hacer un commit en una única línea.

Ya hemos visto que con la opción `-m` podemos añadir directamente un mensaje en el commit. Adicionalmente, con la opción `-a` podemos saltarnos el paso previo (`git add .`) y marcar como staged aquellos archivos modificados y hacer un commit en una única línea.

```
git commit -a -m "Tercer commit: especificar dependencias en README.md"
```

```
# [master 617fcfa] Tercer commit: especificar dependencias en README.md  
# 1 file changed, 1 insertion(+)
```

Ya hemos visto que con la opción `-m` podemos añadir directamente un mensaje en el commit. Adicionalmente, con la opción `-a` podemos saltarnos el paso previo (`git add .`) y marcar como staged aquellos archivos modificados y hacer un commit en una única línea.

```
git commit -a -m "Tercer commit: especificar dependencias en README.md"
```

```
# [master 617fcfa] Tercer commit: especificar dependencias en README.md  
# 1 file changed, 1 insertion(+)
```

Otra forma de trabajar es lanzando las dos instrucciones en la misma línea con el operador `&&`:

Ya hemos visto que con la opción `-m` podemos añadir directamente un mensaje en el commit. Adicionalmente, con la opción `-a` podemos saltarnos el paso previo (`git add .`) y marcar como staged aquellos archivos modificados y hacer un commit en una única línea.

```
git commit -a -m "Tercer commit: especificar dependencias en README.md"
```

```
# [master 617fcfa] Tercer commit: especificar dependencias en README.md
# 1 file changed, 1 insertion(+)
```

Otra forma de trabajar es lanzando las dos instrucciones en la misma línea con el operador `&&`:

```
echo "Este archivo es nuevo" >> inst/docs/nuevo_doc.txt
git add . && git commit -m "Cuarto commit: se incorpora inst/docs/nuevo_doc.txt"
```

```
# [master 6b1bce2] Cuarto commit: se incorpora inst/docs/nuevo_doc.txt
# 1 file changed, 1 insertion(+)
# create mode 100644 inst/docs/nuevo_doc.txt
```

Ya hemos visto que con la opción `-m` podemos añadir directamente un mensaje en el commit. Adicionalmente, con la opción `-a` podemos saltarnos el paso previo (`git add .`) y marcar como staged aquellos archivos modificados y hacer un commit en una única línea.

```
git commit -a -m "Tercer commit: especificar dependencias en README.md"
```

```
# [master 617fcfa] Tercer commit: especificar dependencias en README.md
# 1 file changed, 1 insertion(+)
```

Otra forma de trabajar es lanzando las dos instrucciones en la misma línea con el operador `&&`:

```
echo "Este archivo es nuevo" >> inst/docs/nuevo_doc.txt
git add . && git commit -m "Cuarto commit: se incorpora inst/docs/nuevo_doc.txt"
```

```
# [master 6b1bce2] Cuarto commit: se incorpora inst/docs/nuevo_doc.txt
# 1 file changed, 1 insertion(+)
# create mode 100644 inst/docs/nuevo_doc.txt
```

Por último, podemos etiquetar commits con `git tag`, de forma que resulte sencilla su localización (muy útil a la hora de identificar versiones de desarrollo):

Ya hemos visto que con la opción `-m` podemos añadir directamente un mensaje en el commit. Adicionalmente, con la opción `-a` podemos saltarnos el paso previo (`git add .`) y marcar como staged aquellos archivos modificados y hacer un commit en una única línea.

```
git commit -a -m "Tercer commit: especificar dependencias en README.md"
```

```
# [master 617fcfa] Tercer commit: especificar dependencias en README.md
# 1 file changed, 1 insertion(+)
```

Otra forma de trabajar es lanzando las dos instrucciones en la misma línea con el operador `&&`:

```
echo "Este archivo es nuevo" >> inst/docs/nuevo_doc.txt
git add . && git commit -m "Cuarto commit: se incorpora inst/docs/nuevo_doc.txt"
```

```
# [master 6b1bce2] Cuarto commit: se incorpora inst/docs/nuevo_doc.txt
# 1 file changed, 1 insertion(+)
# create mode 100644 inst/docs/nuevo_doc.txt
```

Por último, podemos etiquetar commits con `git tag`, de forma que resulte sencilla su localización (muy útil a la hora de identificar versiones de desarrollo):

```
git tag "v0.0.1"
git tag
```

```
# v0.0.1
```


Estamos tomando un...



Vuelta al trabajo



Modificar el repositorio

Deshaciendo commits (1/4)

En Git hay infinidad de opciones, y a la hora de deshacer cambios (p. ej., volver a una versión previa, desmarcar un archivo como staged o limpiar el directorio de trabajo) hay varios abordajes posibles.

Deshaciendo commits (1/4)

En Git hay infinidad de opciones, y a la hora de deshacer cambios (p. ej., volver a una versión previa, desmarcar un archivo como staged o limpiar el directorio de trabajo) hay varios abordajes posibles.

¡Solo para usuarios avanzados!

Salvo que sepas a la perfección lo que haces, NUNCA BORRES CONTENIDO DE GIT.

Los commits ocupan muy poco espacio y no conviene alterar el historial de Git ()

Deshaciendo commits (1/4)

En Git hay infinidad de opciones, y a la hora de deshacer cambios (p. ej., volver a una versión previa, desmarcar un archivo como staged o limpiar el directorio de trabajo) hay varios abordajes posibles.

¡Solo para usuarios avanzados!

Salvo que sepas a la perfección lo que haces, NUNCA BORRES CONTENIDO DE GIT.

Los commits ocupan muy poco espacio y no conviene alterar el historial de Git ()

Empecemos por lo más sencillo: hemos añadido al área de preparación un documento y queremos retirarlo para que solo figure en el directorio de trabajo:

Deshaciendo commits (1/4)

En Git hay infinidad de opciones, y a la hora de deshacer cambios (p. ej., volver a una versión previa, desmarcar un archivo como staged o limpiar el directorio de trabajo) hay varios abordajes posibles.

¡Solo para usuarios avanzados!

Salvo que sepas a la perfección lo que haces, NUNCA BORRES CONTENIDO DE GIT.

Los commits ocupan muy poco espacio y no conviene alterar el historial de Git ()

Empecemos por lo más sencillo: hemos añadido al área de preparación un documento y queremos retirarlo para que solo figure en el directorio de trabajo:

```
echo "Contenido nuevo" >> README.md
git add .
git rm --cached README.md
git status

# rm 'README.md'
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   deleted:    README.md
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#   README.md
```

Deshaciendo commits (1/4)

En Git hay infinidad de opciones, y a la hora de deshacer cambios (p. ej., volver a una versión previa, desmarcar un archivo como staged o limpiar el directorio de trabajo) hay varios abordajes posibles.

¡Solo para usuarios avanzados!

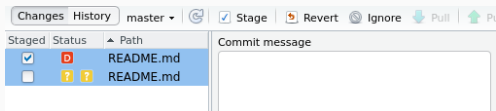
Salvo que sepas a la perfección lo que haces, NUNCA BORRES CONTENIDO DE GIT.

Los commits ocupan muy poco espacio y no conviene alterar el historial de Git ()

Empecemos por lo más sencillo: hemos añadido al área de preparación un documento y queremos retirarlo para que solo figure en el directorio de trabajo:

```
echo "Contenido nuevo" >> README.md
git add .
git rm --cached README.md
git status
```

```
# rm 'README.md'
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       deleted:    README.md
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       README.md
```



También podemos revertir un archivo modificado en el directorio de trabajo a la versión almacenada en un commit (por defecto se vuelve a la versión del último commit):

Deshaciendo commits (2/4)

También podemos revertir un archivo modificado en el directorio de trabajo a la versión almacenada en un commit (por defecto se vuelve a la versión del último commit):

```
git status
git add .
git reset HEAD README.md
git checkout -- README.md
```

```
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   deleted:    README.md
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#   README.md
#
# Unstaged changes after reset:
# M README.md
```

Deshaciendo commits (2/4)

También podemos revertir un archivo modificado en el directorio de trabajo a la versión almacenada en un commit (por defecto se vuelve a la versión del último commit):

```
git status
git add .
git reset HEAD README.md
git checkout -- README.md
```

```
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   deleted:    README.md
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#   README.md
#
# Unstaged changes after reset:
# M README.md
```

Tras esto, el directorio de trabajo queda limpio (se descartó cualquier cambio del archivo `README.md`).

```
git status
```

```
# On branch master
# nothing to commit, working tree clean
```

Deshaciendo commits (3/4)

Cómo no, es posible deshacer commits o volver a un commit concreto. Veamos por dónde nos quedamos:

Deshaciendo commits (3/4)

Cómo no, es posible deshacer commits o volver a un commit concreto. Veamos por dónde nos quedamos:

```
git log --oneline -n 3
```

```
# 6b1bce2 Cuarto commit: se incorpora inst/docs/nuevo_doc.txt  
# 617fcfa Tercer commit: especificar dependencias en README.md  
# 5906fbf Segundo commit: se añade copia de README.md en inst/docs/
```

Deshaciendo commits (3/4)

Cómo no, es posible deshacer commits o volver a un commit concreto. Veamos por dónde nos quedamos:

```
git log --oneline -n 3
```

```
# 6b1bce2 Cuarto commit: se incorpora inst/docs/nuevo_doc.txt  
# 617fcfa Tercer commit: especificar dependencias en README.md  
# 5906fbf Segundo commit: se añade copia de README.md en inst/docs/
```

Existen cuatro commits. Gracias al comando **git reset** se puede retroceder a tantos commits como se desee. P. ej., podemos retroceder un commit, descartando el contenido del cuarto:

Deshaciendo commits (3/4)

Cómo no, es posible deshacer commits o volver a un commit concreto. Veamos por dónde nos quedamos:

```
git log --oneline -n 3
```

```
# 6b1bce2 Cuarto commit: se incorpora inst/docs/nuevo_doc.txt  
# 617fcfa Tercer commit: especificar dependencias en README.md  
# 5906fbf Segundo commit: se añade copia de README.md en inst/docs/
```

Existen cuatro commits. Gracias al comando **git reset** se puede retroceder a tantos commits como se desee. P. ej., podemos retroceder un commit, descartando el contenido del cuarto:

```
git reset HEAD~1  
git log --oneline -n 3
```

```
# 617fcfa Tercer commit: especificar dependencias en README.md  
# 5906fbf Segundo commit: se añade copia de README.md en inst/docs/  
# a0a792d Primer commit: se crean archivos README.md y prueba.txt
```

Usar con precaución

Aconsejo no utilizar esta instrucción (o ser muy precavido al hacerlo), pues según las opciones que se escojan (o dejen de escogerse) existe riesgo de pérdida de datos.

Eliminar un commit altera el historial del repositorio. Los commits no ocupan casi espacio: no hay ningún problema en guardarlos todos y recuperar versiones previas de un documento.

Deshaciendo commits (3/4)

Cómo no, es posible deshacer commits o volver a un commit concreto. Veamos por dónde nos quedamos:

```
git log --oneline -n 3
```

```
# 6b1bce2 Cuarto commit: se incorpora inst/docs/nuevo_doc.txt
# 617fcfa Tercer commit: especificar dependencias en README.md
# 5906fbf Segundo commit: se añade copia de README.md en inst/docs/
```

Existen cuatro commits. Gracias al comando **git reset** se puede retroceder a tantos commits como se desee. P. ej., podemos retroceder un commit, descartando el contenido del cuarto:

```
git reset HEAD~1
git log --oneline -n 3
```

```
# 617fcfa Tercer commit: especificar dependencias en README.md
# 5906fbf Segundo commit: se añade copia de README.md en inst/docs/
# a0a792d Primer commit: se crean archivos README.md y prueba.txt
```

Usar con precaución

Aconsejo no utilizar esta instrucción (o ser muy precavido al hacerlo), pues según las opciones que se escojan (o dejen de escogerse) **existe riesgo de pérdida de datos**.

Eliminar un commit altera el historial del repositorio. Los commits **no ocupan casi espacio**: no hay ningún problema en guardarlos todos y recuperar versiones previas de un documento.

La práctica más habitual es tratar de evitar la edición de commits y trabajar en base a archivos concretos con `git checkout -- archivo` (como ya vimos hace un momento). Vamos a ver otro ejemplo.

La práctica más habitual es tratar de evitar la edición de commits y trabajar en base a archivos concretos con `git checkout -- archivo` (como ya vimos hace un momento). Vamos a ver otro ejemplo.

- P. ej., imaginemos que hemos realizado un commit, editamos un archivo, y tras ello queremos deshacer los cambios volviendo a la versión almacenada en el commit.

La práctica más habitual es tratar de evitar la edición de commits y trabajar en base a archivos concretos con `git checkout -- archivo` (como ya vimos hace un momento). Vamos a ver otro ejemplo.

- P. ej., imaginemos que hemos realizado un commit, editamos un archivo, y tras ello queremos deshacer los cambios volviendo a la versión almacenada en el commit.

```
echo "Modifo este archivo añadiendo una nueva línea..." >> inst/docs/nuevo_doc.txt  
git status -sb
```

```
# ## master  
# ?? inst/docs/nuevo_doc.txt
```

La práctica más habitual es tratar de evitar la edición de commits y trabajar en base a archivos concretos con `git checkout -- archivo` (como ya vimos hace un momento). Vamos a ver otro ejemplo.

- P. ej., imaginemos que hemos realizado un commit, editamos un archivo, y tras ello queremos deshacer los cambios volviendo a la versión almacenada en el commit.

```
echo "Modifo este archivo añadiendo una nueva línea..." >> inst/docs/nuevo_doc.txt
git status -sb
```

```
# ## master
# ?? inst/docs/nuevo_doc.txt
```

- Con la instrucción `git checkout -- documento` se puede volver a la versión almacenada en el último commit.

Deshaciendo commits (4/4)

La práctica más habitual es tratar de evitar la edición de commits y trabajar en base a archivos concretos con `git checkout -- archivo` (como ya vimos hace un momento). Vamos a ver otro ejemplo.

- P. ej., imaginemos que hemos realizado un commit, editamos un archivo, y tras ello queremos deshacer los cambios volviendo a la versión almacenada en el commit.

```
echo "Modifo este archivo añadiendo una nueva línea..." >> inst/docs/nuevo_doc.txt
git status -sb
```

```
# ## master
# ?? inst/docs/nuevo_doc.txt
```

- Con la instrucción `git checkout -- documento` se puede volver a la versión almacenada en el último commit.

```
git checkout -- inst/docs/nuevo_doc.txt
git status -sb
```

```
# error: pathspec 'inst/docs/nuevo_doc.txt' did not match any file(s) known to git.
# ## master
# ?? inst/docs/nuevo_doc.txt
```

Mover o renombrar archivos

A la hora de mover o renombrar objetos, hacemos uso del comando `git mv` `archivo_entrada` `archivo_salida`, el cual nos pide la ruta del archivo de entrada y la del archivo de salida.

Mover o renombrar archivos

A la hora de mover o renombrar objetos, hacemos uso del comando `git mv` `archivo_entrada` `archivo_salida`, el cual nos pide la ruta del archivo de entrada y la del archivo de salida.

- Vamos a renombrar el archivo `nuevo_doc.txt` como `doc_renombrado.txt`, y a mover el archivo `prueba.txt` a `inst/docs/`:

```
git mv inst/docs/nuevo_doc.txt inst/docs/doc_renombrado.txt
git status -sb
```

```
# fatal: not under version control, source=inst/docs/nuevo_doc.txt, destination=inst/docs/doc_renombrado.txt
# ## master
# ?? inst/docs/nuevo_doc.txt
```

- También podemos mover un documento a un directorio diferente:

Mover o renombrar archivos

A la hora de mover o renombrar objetos, hacemos uso del comando `git mv` `archivo_entrada` `archivo_salida`, el cual nos pide la ruta del archivo de entrada y la del archivo de salida.

- Vamos a renombrar el archivo `nuevo_doc.txt` como `doc_renombrado.txt`, y a mover el archivo `prueba.txt` a `inst/docs/`:

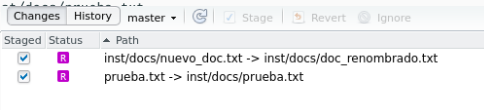
```
git mv inst/docs/nuevo_doc.txt inst/docs/doc_renombrado.txt
git status -sb
```

```
# fatal: not under version control, source=inst/docs/nuevo_doc.txt, destination=inst/docs/doc_renombrado.txt
# ## master
# ?? inst/docs/nuevo_doc.txt
```

- También podemos mover un documento a un directorio diferente:

```
git mv prueba.txt inst/docs/prueba.txt
git status -sb
```

```
# ## master
# R prueba.txt -> inst/docs/prueba.txt
# ?? inst/docs/nuevo_
```



Staged	Status	Path
<input checked="" type="checkbox"/>	R	inst/docs/nuevo_doc.txt -> inst/docs/doc_renombrado.txt
<input checked="" type="checkbox"/>	R	prueba.txt -> inst/docs/prueba.txt

Mover o renombrar archivos

A la hora de mover o renombrar objetos, hacemos uso del comando `git mv` `archivo_entrada` `archivo_salida`, el cual nos pide la ruta del archivo de entrada y la del archivo de salida.

- Vamos a renombrar el archivo `nuevo_doc.txt` como `doc_renombrado.txt`, y a mover el archivo `prueba.txt` a `inst/docs/`:

```
git mv inst/docs/nuevo_doc.txt inst/docs/doc_renombrado.txt
git status -sb
```

```
# fatal: not under version control, source=inst/docs/nuevo_doc.txt, destination=inst/docs/doc_renombrado.txt
# ## master
# ?? inst/docs/nuevo_doc.txt
```

- También podemos mover un documento a un directorio diferente:

```
git mv prueba.txt inst/docs/prueba.txt
git status -sb
```

```
# ## master
# R prueba.txt -> inst/docs/prueba.txt
# ?? inst/docs/nuevo_doc.txt
```

- El estado nos indica lo sucedido con estos archivos. Solo queda confirmar los cambios (no es necesario pasar por la etapa de staged).

Mover o renombrar archivos

A la hora de mover o renombrar objetos, hacemos uso del comando `git mv` `archivo_entrada` `archivo_salida`, el cual nos pide la ruta del archivo de entrada y la del archivo de salida.

- Vamos a renombrar el archivo `nuevo_doc.txt` como `doc_renombrado.txt`, y a mover el archivo `prueba.txt` a `inst/docs/`:

```
git mv inst/docs/nuevo_doc.txt inst/docs/doc_renombrado.txt
git status -sb
```

```
# fatal: not under version control, source=inst/docs/nuevo_doc.txt, destination=inst/docs/doc_renombrado.txt
# ## master
# ?? inst/docs/nuevo_doc.txt
```

- También podemos mover un documento a un directorio diferente:

```
git mv prueba.txt inst/docs/prueba.txt
git status -sb
```

```
# ## master
# R prueba.txt -> inst/docs/prueba.txt
# ?? inst/docs/nuevo_doc.txt
```

- El estado nos indica lo sucedido con estos archivos. Solo queda confirmar los cambios (no es necesario pasar por la etapa de staged).

```
git commit -m "Quinto commit: se cambia de nombre de archivo"
```

```
# [master 75355b2] Quinto commit: se cambia de nombre de archivo
# 1 file changed, 0 insertions(+), 0 deletions(-)
# rename prueba.txt => inst/docs/prueba.txt (100%)
```

Ya comentamos que en ocasiones no deseamos realizar un control de versiones de algunos archivos (como toda la cascada de archivos secundarios a \LaTeX o archivos de caché), pero solo hablamos acerca de no añadirlos a la zona de preparación.

Ya comentamos que en ocasiones no deseamos realizar un control de versiones de algunos archivos (como toda la cascada de archivos secundarios a \LaTeX o archivos de caché), pero solo hablamos acerca de no añadirlos a la zona de preparación.

No obstante, estos archivos continuarán apareciendo como «no seguidos» cada vez que miremos el estado del repositorio, y esto es bastante molesto. Existe una forma de ocultar estos archivos a ojos de Git: el archivo **.gitignore**:

1. Contiene los nombres de los archivos o directorios que no deseamos que sean seguidos por Git.
2. Los nombres pueden ser literales o formar una expresión regular que represente un patrón.
3. En un mismo repositorio pueden coexistir varios archivos **.gitignore**.

Ya comentamos que en ocasiones no deseamos realizar un control de versiones de algunos archivos (como toda la cascada de archivos secundarios a \LaTeX o archivos de caché), pero solo hablamos acerca de no añadirlos a la zona de preparación.

No obstante, estos archivos continuarán apareciendo como «no seguidos» cada vez que miremos el estado del repositorio, y esto es bastante molesto. Existe una forma de ocultar estos archivos a ojos de Git: el archivo **.gitignore**:

1. Contiene los nombres de los archivos o directorios que no deseamos que sean seguidos por Git.
2. Los nombres pueden ser literales o formar una expresión regular que represente un patrón.
3. En un mismo repositorio pueden coexistir varios archivos **.gitignore**.

Veamos un ejemplo útil. Cuando trabajamos con R y RStudio, normalmente no queremos que se siga el historial de sesiones, los datos guardados de forma automática, los archivos de RStudio, o datos de caché.

Ya comentamos que en ocasiones no deseamos realizar un control de versiones de algunos archivos (como toda la cascada de archivos secundarios a \LaTeX o archivos de caché), pero solo hablamos acerca de no añadirlos a la zona de preparación.

No obstante, estos archivos continuarán apareciendo como «no seguidos» cada vez que miremos el estado del repositorio, y esto es bastante molesto. Existe una forma de ocultar estos archivos a ojos de Git: el archivo **.gitignore**:

1. Contiene los nombres de los archivos o directorios que no deseamos que sean seguidos por Git.
2. Los nombres pueden ser literales o formar una expresión regular que represente un patrón.
3. En un mismo repositorio pueden coexistir varios archivos **.gitignore**.

Veamos un ejemplo útil. Cuando trabajamos con R y RStudio, normalmente no queremos que se siga el historial de sesiones, los datos guardados de forma automática, los archivos de RStudio, o datos de caché.

El archivo **.gitignore** que utilizaríamos para ignorarlos podría ser parecido al siguiente:

```
.Rhistory  
.RData  
.Rproj.user/  
/cache/
```

```
# [master cc8c246] Se añade el archivo .gitignore y archivos renombrados  
# 2 files changed, 6 insertions(+)
```

Ramas de desarrollo

Uno de los elementos más exitosos de Git es la posibilidad de utilizar ramas de desarrollo, que son vías alternativas de código que no ponen el peligro al código que sabemos que funciona. Los comandos fundamentales a la hora de trabajar con ramas son `git branch`, `git checkout rama` y `git merge rama`. Vamos a ver su funcionamiento básico.

Uno de los elementos más exitosos de Git es la posibilidad de utilizar ramas de desarrollo, que son vías alternativas de código que no ponen el peligro al código que sabemos que funciona. Los comandos fundamentales a la hora de trabajar con ramas son `git branch`, `git checkout rama` y `git merge rama`. Vamos a ver su funcionamiento básico.

1. En nuestro proyecto no sabemos en qué rama estamos trabajando, pero podemos comprobarlo fácilmente:

Uno de los elementos más exitosos de Git es la posibilidad de utilizar ramas de desarrollo, que son vías alternativas de código que no ponen el peligro al código que sabemos que funciona. Los comandos fundamentales a la hora de trabajar con ramas son `git branch`, `git checkout rama` y `git merge rama`. Vamos a ver su funcionamiento básico.

1. En nuestro proyecto no sabemos en qué rama estamos trabajando, pero podemos comprobarlo fácilmente:

```
git branch
```

```
# * master
```

Uno de los elementos más exitosos de Git es la posibilidad de utilizar ramas de desarrollo, que son vías alternativas de código que no ponen el peligro al código que sabemos que funciona. Los comandos fundamentales a la hora de trabajar con ramas son `git branch`, `git checkout rama` y `git merge rama`. Vamos a ver su funcionamiento básico.

1. En nuestro proyecto no sabemos en qué rama estamos trabajando, pero podemos comprobarlo fácilmente:

```
git branch
```

```
# * master
```

2. Queremos realizar algunas pruebas sin poner en peligro el contenido principal, así que creamos una nueva rama llamada **desarrollo** y nos cambiamos a ella.

Uno de los elementos más exitosos de Git es la posibilidad de utilizar ramas de desarrollo, que son vías alternativas de código que no ponen el peligro al código que sabemos que funciona. Los comandos fundamentales a la hora de trabajar con ramas son `git branch`, `git checkout rama` y `git merge rama`. Vamos a ver su funcionamiento básico.

1. En nuestro proyecto no sabemos en qué rama estamos trabajando, pero podemos comprobarlo fácilmente:

```
git branch
```

```
# * master
```

2. Queremos realizar algunas pruebas sin poner en peligro el contenido principal, así que creamos una nueva rama llamada **desarrollo** y nos cambiamos a ella.

```
git branch desarrollo  
git checkout desarrollo
```

```
# Switched to branch 'desarrollo'
```

Uno de los elementos más exitosos de Git es la posibilidad de utilizar ramas de desarrollo, que son vías alternativas de código que no ponen en peligro al código que sabemos que funciona. Los comandos fundamentales a la hora de trabajar con ramas son `git branch`, `git checkout rama` y `git merge rama`. Vamos a ver su funcionamiento básico.

1. En nuestro proyecto no sabemos en qué rama estamos trabajando, pero podemos comprobarlo fácilmente:

```
git branch
```

```
# * master
```

2. Queremos realizar algunas pruebas sin poner en peligro el contenido principal, así que creamos una nueva rama llamada **desarrollo** y nos cambiamos a ella.

```
git branch desarrollo  
git checkout desarrollo
```

```
# Switched to branch 'desarrollo'
```

Explicación

- Con `git branch desarrollo` se crea una rama con ese nombre.
- Con `git checkout desarrollo` se cambia de rama.

Uno de los puntos fuerte de las ramas es que al cambiar de una a otra también se cambia el contenido del directorio de trabajo, es decir, los archivos vuelven a la versión contenida en el último commit de la rama a la que se cambie. Esta utilidad es ideal a la hora de colaborar entre personas o en el desarrollo de ideas locas. P. ej.:

Uno de los puntos fuerte de las ramas es que al cambiar de una a otra también se cambia el contenido del directorio de trabajo, es decir, los archivos vuelven a la versión contenida en el último commit de la rama a la que se cambie. Esta utilidad es ideal a la hora de colaborar entre personas o en el desarrollo de ideas locas. P. ej.:

- Estamos dentro de la rama desarrollo y creamos un archivo que creemos podría mejorar el proyecto:

```
echo 'Creamos un archivo en la nueva rama' >> desarrollo.txt
git status
```

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

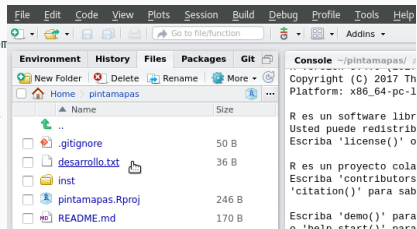
Edición en paralelo (1/2)

Uno de los puntos fuerte de las ramas es que al cambiar de una a otra también se cambia el contenido del directorio de trabajo, es decir, los archivos vuelven a la versión contenida en el último commit de la rama a la que se cambie. Esta utilidad es ideal a la hora de colaborar entre personas o en el desarrollo de ideas locas. P. ej.:

- Estamos dentro de la rama desarrollo y creamos un archivo que creemos podría mejorar el proyecto:

```
echo 'Creamos un archivo en la nueva rama' >> desarrollo.txt
git status
```

```
# On branch desarrollo
# Untracked files:
#   (use "git add <file>..." to include in what will be com
#
#   desarrollo.txt
#
# nothing added to commit but untracked files present (use
```



Uno de los puntos fuerte de las ramas es que al cambiar de una a otra también se cambia el contenido del directorio de trabajo, es decir, los archivos vuelven a la versión contenida en el último commit de la rama a la que se cambie. Esta utilidad es ideal a la hora de colaborar entre personas o en el desarrollo de ideas locas. P. ej.:

- Estamos dentro de la rama desarrollo y creamos un archivo que creemos podría mejorar el proyecto:

```
echo 'Creamos un archivo en la nueva rama' >> desarrollo.txt
git status
```

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

- Antes de que finalicemos con su edición, nos damos cuenta de que tenemos un *bug* en la rama principal, así que guardamos los cambios realizados con la intención de volver a la rama principal a solucionarlo:

```
git add . && git commit -m "Primer commit en desarrollo: se incorpora un archivo"
```

```
# [desarrollo 8e8e766] Primer commit en desarrollo: se incorpora un archivo
# 1 file changed, 1 insertion(+)
# create mode 100644 desarrollo.txt
```

- Cambiamos de rama, volviendo a la principal:

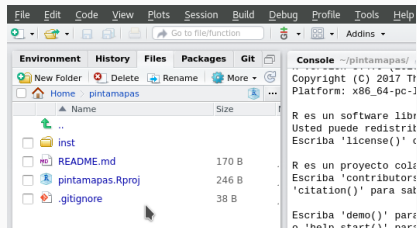
```
git checkout master
```

```
# Switched to branch 'master'
```

- Cambiamos de rama, volviendo a la principal:

```
git checkout master
```

```
# Switched to branch 'master'
```



- Cambiamos de rama, volviendo a la principal:

```
git checkout master
```

```
# Switched to branch 'master'
```

- Solucionamos el problema en la rama principal, y realizamos un commit:

```
echo "Solucionamos el problema en la rama master" >> solucion_bug.txt  
git add . && git commit -m "Sexto commit: se soluciona un bug"
```

```
# [master c75ce22] Sexto commit: se soluciona un bug  
# 1 file changed, 1 insertion(+)  
# create mode 100644 solucion_bug.txt
```

- Cambiamos de rama, volviendo a la principal:

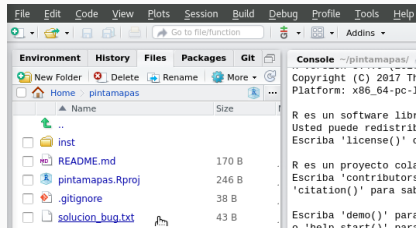
```
git checkout master
```

```
# Switched to branch 'master'
```

- Solucionamos el problema en la rama principal, y realizamos un commit:

```
echo "Solucionamos el problema en la rama master" >> solucion_bug.txt  
git add . 66 git commit -m "Sexto commit: se soluciona un bug"
```

```
# [master c75ce22] Sexto commit: se soluciona un bug  
# 1 file changed, 1 insertion(+)  
# create mode 100644 solucion_bug.txt
```



- Cambiamos de rama, volviendo a la principal:

```
git checkout master
```

```
# Switched to branch 'master'
```

- Solucionamos el problema en la rama principal, y realizamos un commit:

```
echo "Solucionamos el problema en la rama master" >> solucion_bug.txt  
git add . && git commit -m "Sexto commit: se soluciona un bug"
```

```
# [master c75ce22] Sexto commit: se soluciona un bug  
# 1 file changed, 1 insertion(+)  
# create mode 100644 solucion_bug.txt
```

- Para seguir experimentando con las mejoras que dejamos a medias volvemos a la rama **desarrollo** y continuamos con el trabajo que iniciamos:

```
git checkout desarrollo  
echo 'Continuamos con nuestro trabajo' >> desarrollo.txt
```

```
# Switched to branch 'desarrollo'
```

- Cambiamos de rama, volviendo a la principal:

```
git checkout master
```

```
# Switched to branch 'master'
```

- Solucionamos el problema en la rama principal, y realizamos un commit:

```
echo "Solucionamos el problema en la rama master" >> solucion_bug.txt  
git add . && git commit -m "Sexto commit: se soluciona un bug"
```

```
# [master c75ce22] Sexto commit: se soluciona un bug  
# 1 file changed, 1 insertion(+)  
# create mode 100644 solucion_bug.txt
```

- Para seguir experimentando con las mejoras que dejamos a medias volvemos a la rama **desarrollo** y continuamos con el trabajo que iniciamos:

```
git checkout desarrollo  
echo 'Continuamos con nuestro trabajo' >> desarrollo.txt
```

```
# Switched to branch 'desarrollo'
```

- Cambiamos de rama, volviendo a la principal:

```
git checkout master
```

```
# Switched to branch 'master'
```

- Solucionamos el problema en la rama principal, y realizamos un commit:

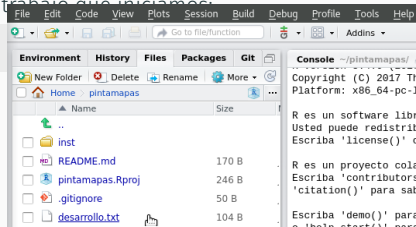
```
echo "Solucionamos el problema en la rama master" >> solucion_bug.txt  
git add . && git commit -m "Sexto commit: se soluciona un bug"
```

```
# [master c75ce22] Sexto commit: se soluciona un bug  
# 1 file changed, 1 insertion(+)  
# create mode 100644 solucion_bug.txt
```

- Para seguir experimentando con las mejoras que dejamos a medias volvemos a la rama **desarrollo** y continuamos con el trabajo que iniciamos:

```
git checkout desarrollo  
echo 'Continuamos con nuestro trabajo' >> desarrollo.txt
```

```
# Switched to branch 'desarrollo'
```



Unir dos ramas

Una vez haya finalizado la experimentación, podemos decidir si ha merecido la pena (incorporamos en la rama principal el archivo de **desarrollo**) o si ha sido una pequeña pérdida de tiempo (borramos la rama **desarrollo**).

Imaginemos que estamos en el primer caso, lo que deberíamos hacer es lo siguiente:

Una vez haya finalizado la experimentación, podemos decidir si ha merecido la pena (incorporamos en la rama principal el archivo de **desarrollo**) o si ha sido una pequeña pérdida de tiempo (borramos la rama **desarrollo**).

Imaginemos que estamos en el primer caso, lo que deberíamos hacer es lo siguiente:

1. Guardar los cambios en la rama de desarrollo.

```
git add . && git commit -m "Segundo commit en desarrollo: se finaliza la mejora"
```

```
# [desarrollo cf91af8] Segundo commit en desarrollo: se finaliza la mejora  
# 1 file changed, 1 insertion(+)
```

Una vez haya finalizado la experimentación, podemos decidir si ha merecido la pena (incorporamos en la rama principal el archivo de **desarrollo**) o si ha sido una pequeña pérdida de tiempo (borramos la rama **desarrollo**).

Imaginemos que estamos en el primer caso, lo que deberíamos hacer es lo siguiente:

1. Guardar los cambios en la rama de desarrollo.

```
git add . && git commit -m "Segundo commit en desarrollo: se finaliza la mejora"
```

```
# [desarrollo cf91af8] Segundo commit en desarrollo: se finaliza la mejora  
# 1 file changed, 1 insertion(+)
```

2. Cambiar a la rama principal.

```
git checkout master
```

```
# Switched to branch 'master'
```

Unir dos ramas

Una vez haya finalizado la experimentación, podemos decidir si ha merecido la pena (incorporamos en la rama principal el archivo de **desarrollo**) o si ha sido una pequeña pérdida de tiempo (borramos la rama **desarrollo**).

Imaginemos que estamos en el primer caso, lo que deberíamos hacer es lo siguiente:

1. Guardar los cambios en la rama de desarrollo.

```
git add . && git commit -m "Segundo commit en desarrollo: se finaliza la mejora"
```

```
# [desarrollo cf91af8] Segundo commit en desarrollo: se finaliza la mejora  
# 1 file changed, 1 insertion(+)
```

2. Cambiar a la rama principal.

```
git checkout master
```

```
# Switched to branch 'master'
```

3. Agregar el contenido de la rama de desarrollo a la rama principal y se borra la rama de desarrollo.

```
git merge desarrollo -m "Séptimo commit: Unión de dos ramas"  
git branch -d desarrollo
```

```
# Merge made by the 'recursive' strategy.  
# desarrollo.txt | 2 ++  
# 1 file changed, 2 insertions(+)  
# create mode 100644 desarrollo.txt  
# Deleted branch desarrollo (was cf91af8).
```

Unir dos ramas

Una vez haya finalizado la experimentación, podemos decidir si ha merecido la pena (incorporamos en la rama principal el archivo de **desarrollo**) o si ha sido una pequeña pérdida de tiempo (borramos la rama **desarrollo**).

Imaginemos que estamos en el primer caso, lo que deberíamos hacer es lo siguiente:

1. Guardar los cambios en la rama de desarrollo.

```
git add . && git commit -m "Segundo commit en desarrollo: se finaliza la mejora"
```

```
# [desarrollo cf91af8] Segundo commit en desarrollo: se finaliza la mejora  
# 1 file changed, 1 insertion(+)
```

2. Cambiar a la rama principal.

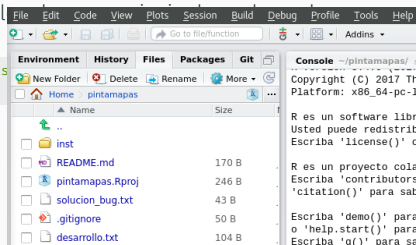
```
git checkout master
```

```
# Switched to branch 'master'
```

3. Agregar el contenido de la rama de desarrollo a la rama de desarrollo.

```
git merge desarrollo -m "Séptimo commit: Unión de dos ramas"  
git branch -d desarrollo
```

```
# Merge made by the 'recursive' strategy.  
# desarrollo.txt | 2 ++  
# 1 file changed, 2 insertions(+)  
# create mode 100644 desarrollo.txt  
# Deleted branch desarrollo (was cf91af8).
```



Git y GitHub

En la introducción, al hablar sobre Git, vimos que casi todo el trabajo se realiza de modo local.

En la introducción, al hablar sobre Git, vimos que casi todo el trabajo se realiza de modo local.

No obstante, es posible almacenar un repositorio de forma remota:

1. dentro de un servidor con Git instalado,
2. servicios de almacenamiento Web específicos.

En la introducción, al hablar sobre Git, vimos que casi todo el trabajo se realiza de modo local.

No obstante, es posible almacenar un repositorio de forma remota:

1. dentro de un servidor con Git instalado,
2. servicios de almacenamiento Web específicos.

Al depositar el repositorio en un lugar público, este es accesible a todo el mundo, lo que facilita tanto el uso como la colaboración en el desarrollo de software.

En la introducción, al hablar sobre Git, vimos que casi todo el trabajo se realiza de modo local.

No obstante, es posible almacenar un repositorio de forma remota:

1. dentro de un servidor con Git instalado,
2. servicios de almacenamiento Web específicos.

Al depositar el repositorio en un lugar público, este es accesible a todo el mundo, lo que facilita tanto el uso como la colaboración en el desarrollo de software.

De forma paralela, también se puede trabajar con un repositorio remoto privado (cualquiera de las dos opciones previas lo permiten), con lo que se dispone de un mecanismo para compartir y trabajar material dentro de un grupo.

No todo es GitHub...

GitHub es la plataforma de referencia a la hora de establecer repositorios remotos y ofrece dos modalidades de repositorios:

No todo es GitHub...

GitHub es la plataforma de referencia a la hora de establecer repositorios remotos y ofrece dos modalidades de repositorios:

1. Públicos: cualquier persona puede acceder al contenido del mismo (descarga del material, modificación de contenidos y realización de solicitudes de unión de los mismos).
2. Privados: el contenido no es de acceso público. Solo puede visualizarlo la persona que lo crea y quienes sean autorizados por ella.

No todo es GitHub...

GitHub es la plataforma de referencia a la hora de establecer repositorios remotos y ofrece dos modalidades de repositorios:

1. Públicos: cualquier persona puede acceder al contenido del mismo (descarga del material, modificación de contenidos y realización de solicitudes de unión de los mismos).
2. Privados: el contenido no es de acceso público. Solo puede visualizarlo la persona que lo crea y quienes sean autorizados por ella.

Inconveniente

La cuenta gratuita de GitHub no permite crear repositorios privados.

No todo es GitHub...

GitHub es la plataforma de referencia a la hora de establecer repositorios remotos y ofrece dos modalidades de repositorios:

1. Públicos: cualquier persona puede acceder al contenido del mismo (descarga del material, modificación de contenidos y realización de solicitudes de unión de los mismos).
2. Privados: el contenido no es de acceso público. Solo puede visualizarlo la persona que lo crea y quienes sean autorizados por ella.

Inconveniente

La cuenta gratuita de GitHub no permite crear repositorios privados.

Fundamentalmente hay dos alternativas a GitHub:

No todo es GitHub...

GitHub es la plataforma de referencia a la hora de establecer repositorios remotos y ofrece dos modalidades de repositorios:

1. Públicos: cualquier persona puede acceder al contenido del mismo (descarga del material, modificación de contenidos y realización de solicitudes de unión de los mismos).
2. Privados: el contenido no es de acceso público. Solo puede visualizarlo la persona que lo crea y quienes sean autorizados por ella.

Inconveniente

La cuenta gratuita de GitHub no permite crear repositorios privados.

Fundamentalmente hay dos alternativas a GitHub:

- [Bitbucket](#). Pro: proyectos privados ilimitados. Con: los equipos de trabajo en el plan gratuito solo pueden alcanzar 5 miembros.
- [GitLab](#). Pro: número de proyectos privados y de colaboradores ilimitado.

No todo es GitHub...

GitHub es la plataforma de referencia a la hora de establecer repositorios remotos y ofrece dos modalidades de repositorios:

1. Públicos: cualquier persona puede acceder al contenido del mismo (descarga del material, modificación de contenidos y realización de solicitudes de unión de los mismos).
2. Privados: el contenido no es de acceso público. Solo puede visualizarlo la persona que lo crea y quienes sean autorizados por ella.

Inconveniente

La cuenta gratuita de GitHub no permite crear repositorios privados.

Fundamentalmente hay dos alternativas a GitHub:

- [Bitbucket](#). Pro: proyectos privados ilimitados. Con: los equipos de trabajo en el plan gratuito solo pueden alcanzar 5 miembros.
- [GitLab](#). Pro: número de proyectos privados y de colaboradores ilimitado.

Recomendación

Comienza con GitHub y una vez hayas logrado un buen manejo de la plataforma, piensa en probar GitLab.

Valora la posibilidad de solicitar una cuenta para tu organización en GitHub (hay planes gratuitos).

Existen cuatro posibles vías de trabajo en GitHub:

Existen cuatro posibles vías de trabajo en GitHub:

1. Crear un repositorio remoto vacío asociado a un proyecto desde cero.

Existen cuatro posibles vías de trabajo en GitHub:

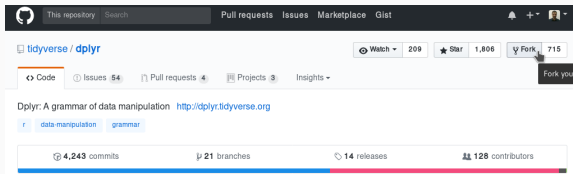
1. Crear un repositorio remoto vacío asociado a un proyecto desde cero.
2. Crear un repositorio remoto y asociarlo a un repositorio local iniciado previamente.

Existen cuatro posibles vías de trabajo en GitHub:

1. Crear un repositorio remoto vacío asociado a un proyecto desde cero.
2. Crear un repositorio remoto y asociarlo a un repositorio local iniciado previamente.
3. Colaborar en un proyecto clonando el repositorio de forma local (`git clone ruta_al_repositorio_remoto`).

Existen cuatro posibles vías de trabajo en GitHub:

1. Crear un repositorio remoto vacío asociado a un proyecto desde cero.
2. Crear un repositorio remoto y asociarlo a un repositorio local iniciado previamente.
3. Colaborar en un proyecto clonando el repositorio de forma local (`git clone ruta_al_repositorio_remoto`).
4. Colaborar en un proyecto clonando el repositorio de forma remota (*fork*).

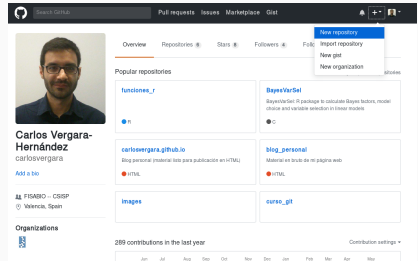


Los pasos a seguir para crear un repositorio en GitHub son muy sencillos, aunque varían ligeramente en función de si estamos creando un proyecto desde cero o queremos dar soporte remoto a un proyecto preexistente.

Crear un repositorio en GitHub

Los pasos a seguir para crear un repositorio en GitHub son muy sencillos, aunque varían ligeramente en función de si estamos creando un proyecto desde cero o queremos dar soporte remoto a un proyecto preexistente.

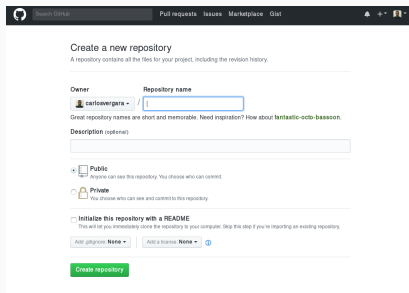
1. En la esquina superior derecha de la página de GitHub, veréis un icono con el símbolo +, haced clic en él y seleccionad *New repository*.



Crear un repositorio en GitHub

Los pasos a seguir para crear un repositorio en GitHub son muy sencillos, aunque varían ligeramente en función de si estamos creando un proyecto desde cero o queremos dar soporte remoto a un proyecto preexistente.

1. En la esquina superior derecha de la página de GitHub, veréis un icono con el símbolo +, haced clic en él y seleccionad *New repository*.
2. Aquí podemos seleccionar varias opciones: para nuestro propósito (dar cobertura remota a un proyecto preexistente), sencillamente introducimos un nombre para el repositorio y lo creamos.

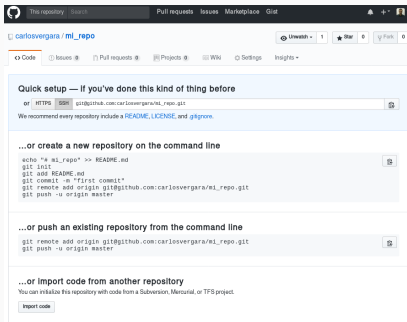


The screenshot shows the GitHub 'Create a new repository' interface. At the top, there's a navigation bar with links for 'Pull requests', 'Issues', 'Marketplace', and 'Git'. Below this, the main heading is 'Create a new repository' with a subtitle 'A repository contains all the files for your project, including the revision history.' The form includes an 'Owner' dropdown set to 'carlovergara', a 'Repository name' text input field, and a 'Description (optional)' text area. There are two radio button options for visibility: 'Public' (selected) and 'Private'. At the bottom, there's a checkbox for 'Initialize this repository with a README' and two dropdown menus for 'Add a license' and 'Add a README'.

Crear un repositorio en GitHub

Los pasos a seguir para crear un repositorio en GitHub son muy sencillos, aunque varían ligeramente en función de si estamos creando un proyecto desde cero o queremos dar soporte remoto a un proyecto preexistente.

1. En la esquina superior derecha de la página de GitHub, veréis un icono con el símbolo +, haced clic en él y seleccionad *New repository*.
2. Aquí podemos seleccionar varias opciones: para nuestro propósito (dar cobertura remota a un proyecto preexistente), sencillamente introducimos un nombre para el repositorio y lo creamos.
3. Se crea un repositorio vacío en vuestra cuenta, y GitHub nos indica opciones muy interesantes.



Tenemos un repositorio local con algo de contenido y uno remoto que está vacío: vamos a igualar las cosas.

Tenemos un repositorio local con algo de contenido y uno remoto que está vacío: vamos a igualar las cosas.

1. En primer lugar agregamos la dirección SSH del repositorio remoto a nuestro repositorio local, y la denominamos **origin**:

```
git remote add origin git@github.com:carlosvergara/pintamapas.git
git remote -v
```

```
# origin    git@github.com:carlosvergara/pintamapas.git (fetch)
# origin    git@github.com:carlosvergara/pintamapas.git (push)
```

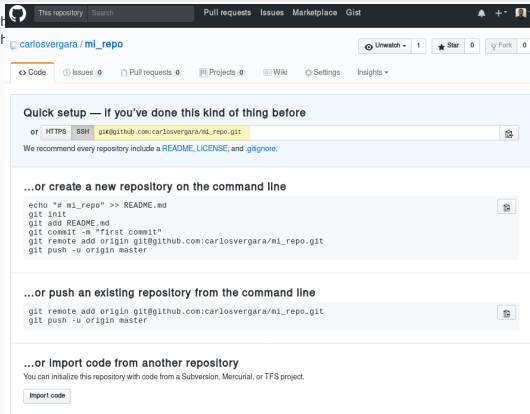
Tenemos un repositorio local con algo de contenido y uno remoto que está vacío: vamos a igualar las cosas.

1. En primer lugar agregamos la dirección SSH del repositorio remoto a nuestro repositorio local, y la denominamos **origin**:

```
git remote add origin git@github.com:carlosvergara/pintamapas.git
git remote -v
```

```
# origin
# origin
```

```
git@github:
git@github:
```



Tenemos un repositorio local con algo de contenido y uno remoto que está vacío: vamos a igualar las cosas.

1. En primer lugar agregamos la dirección SSH del repositorio remoto a nuestro repositorio local, y la denominamos **origin**:

```
git remote add origin git@github.com:carlosvergara/pintamapas.git
git remote -v
```

```
# origin    git@github.com:carlosvergara/pintamapas.git (fetch)
# origin    git@github.com:carlosvergara/pintamapas.git (push)
```

2. Ahora subimos todo el contenido local a GitHub.

Tenemos un repositorio local con algo de contenido y uno remoto que está vacío: vamos a igualar las cosas.

1. En primer lugar agregamos la dirección SSH del repositorio remoto a nuestro repositorio local, y la denominamos **origin**:

```
git remote add origin git@github.com:carlosvergara/pintamapas.git
git remote -v
```

```
# origin    git@github.com:carlosvergara/pintamapas.git (fetch)
# origin    git@github.com:carlosvergara/pintamapas.git (push)
```

2. Ahora subimos todo el contenido local a GitHub.

```
git push -u origin master
```

```
# To github.com:carlosvergara/pintamapas.git
# * [new branch]      master -> master
# Branch master set up to track remote branch master from origin.
```

Tenemos un repositorio local con algo de contenido y uno remoto que está vacío: vamos a igualar las cosas.

1. En primer lugar agregamos la dirección SSH del repositorio remoto a nuestro repositorio local, y la denominamos **origin**:

```
git remote add origin git@github.com:carlosvergara/pintamapas.git
git remote -v
```

```
# origin    git@github.com:carlosvergara/pintamapas.git (fetch)
# origin    git@github.com:carlosvergara/pintamapas.git (push)
```

2. Ahora subimos todo el contenido local a GitHub.

```
git push -u origin master
```

```
# To github.com:carlosvergara/pintamapas.git
# * [new branch]      master -> master
# Branch master set up to track remote branch master from origin.
```

¿Se ha creado una nueva rama?

Sí, pero se ha creado una nueva rama llamada **master** en el repositorio remoto. ¡Recordad que acabamos de crearlo y **estaba completamente vacío**!

Sincronización: locales a remoto

A partir de ahora, cada vez que hagamos cambios en el repositorio a nivel local podemos sincronizar su contenido con el remoto y viceversa.

- Empecemos por crear un nuevo contenido

```
echo "Este paquete ha sido desarrollado en el curso de investigación reproducible" >> README.md
git add . 66 git commit -m "Octavo commit: se hacen modificaciones para probar el trabajo con GitHub"
```

```
# [master c738c72] Octavo commit: se hacen modificaciones para probar el trabajo con GitHub
# 1 file changed, 1 insertion(+)
```


Sincronización: locales a remoto

A partir de ahora, cada vez que hagamos cambios en el repositorio a nivel local podemos sincronizar su contenido con el remoto y viceversa.

- Empecemos por crear un nuevo contenido

```
echo "Este paquete ha sido desarrollado en el curso de investigación reproducible" >> README.md
git add . 66 git commit -m "Octavo commit: se hacen modificaciones para probar el trabajo con GitHub"
```

```
# [master c738c72] Octavo commit: se hacen modificaciones para probar el trabajo con GitHub
# 1 file changed, 1 insertion(+)
```

- Ahora toca sincronizar el contenido que tenemos en el ordenador con el repositorio de GitHub.

```
git push -u origin master
```

```
# To github.com:carlosvergara/pintamapas.git
# 6f90079..c738c72 master -> master
# Branch master set up to track remote branch master from origin.
```

Sincronización: locales a remoto

A partir de ahora, cada vez que hagamos cambios en el repositorio a nivel local podemos sincronizar su contenido con el remoto y viceversa.

- Empecemos por crear un nuevo contenido

```
echo "Este paquete ha sido desarrollado en el curso de investigación reproducible" >> README.md
git add . 66 git commit -m "Octavo commit: se hacen modificaciones para probar el trabajo con GitHub"
```

```
# [master c738c72] Octavo commit: se hacen modificaciones para probar el trabajo con GitHub
# 1 file changed, 1 insertion(+)
```

- Ahora toca sincronizar el contenido que tenemos en el ordenador con el repositorio de GitHub.

```
git push -u origin master
```

```
# To github.com:carlosvergara/pintamapas.git
# 6f90079..c738c72 master -> master
# Branch master set up to track remote branch master from origin.
```

- Aunque el mensaje anterior ya nos informaba al respecto, nos aseguramos de que los repositorios están sincronizados...

```
git status
```

```
# On branch master
# Your branch is up-to-date with 'origin/master'.
# nothing to commit, working tree clean
```

y vemos que todo está en orden.

Sincronización: remoto a local (1/2)

Ahora imaginemos que trabajamos en varios ordenadores (portátil propio y PC en la oficina) y nos vemos en una situación parecida a esta:

Sincronización: remoto a local (1/2)

Ahora imaginemos que trabajamos en varios ordenadores (portátil propio y PC en la oficina) y nos vemos en una situación parecida a esta:

1. Llega el viernes y desde la oficina creamos nuevo contenido, lo guardamos en nuestro repositorio local y lo subimos a GitHub para tenerlo a buen recaudo.

```
echo "Modificación del viernes" >> README.md
git add . && git commit -m "Noveno commit: queda trabajo pendiente para el fin de semana"
git push -u origin master
```

```
# [master 4102eb9] Noveno commit: queda trabajo pendiente para el fin de semana
# 1 file changed, 1 insertion(+)
# To github.com:carlosvergara/pintamapas.git
#   c738c72..4102eb9  master -> master
# Branch master set up to track remote branch master from origin.
```

Sincronización: remoto a local (1/2)

Ahora imaginemos que trabajamos en varios ordenadores (portátil propio y PC en la oficina) y nos vemos en una situación parecida a esta:

1. Llega el viernes y desde la oficina creamos nuevo contenido, lo guardamos en nuestro repositorio local y lo subimos a GitHub para tenerlo a buen recaudo.

```
echo "Modificación del viernes" >> README.md
git add . && git commit -m "Noveno commit: queda trabajo pendiente para el fin de semana"
git push -u origin master
```

```
# [master 4102eb9] Noveno commit: queda trabajo pendiente para el fin de semana
# 1 file changed, 1 insertion(+)
# To github.com:carlosvergara/pintamapas.git
#   c738c72..4102eb9  master -> master
# Branch master set up to track remote branch master from origin.
```

2. El sábado nos apetece darle un pequeño adelanto al proyecto con nuestro portátil, pero no disponemos de la última versión que se trabajó el viernes, así que tendremos que descargarla desde el remoto y unirla a nuestra rama local. Esto se hace con una sola instrucción:

```
git pull
```

```
# From github.com:carlosvergara/pintamapas
# * branch          master      -> FETCH_HEAD
#   c738c72..4102eb9  master      -> origin/master
# Updating c738c72..4102eb9
# Fast-forward
#  README.md | 1 +
#  1 file changed, 1 insertion(+)
```

3. Trabajamos y al rato nos preguntamos qué hacemos con historias de la oficina un sábado por la tarde... decidimos guardar el contenido, subir los cambios al remoto y salir de juerga.

```
echo "Modificación del sábado" >> README.md
git add . 66 git commit -m "Décimo commit: se adelanta trabajo el sábado"
git push -u origin master
```

```
# [master acfa787] Décimo commit: se adelanta trabajo el sábado
# 1 file changed, 1 insertion(+)
# To github.com:carlosvergara/pintamapas.git
# 4102eb9..acfa787 master -> master
# Branch master set up to track remote branch master from origin.
```

- Trabajamos y al rato nos preguntamos qué hacemos con historias de la oficina un sábado por la tarde... decidimos guardar el contenido, subir los cambios al remoto y salir de juerga.

```
echo "Modificación del sábado" >> README.md
git add . 66 git commit -m "Décimo commit: se adelanta trabajo el sábado"
git push -u origin master
```

```
# [master acfa787] Décimo commit: se adelanta trabajo el sábado
# 1 file changed, 1 insertion(+)
# To github.com:carlosvergara/pintamapas.git
# 4102eb9..acfa787 master -> master
# Branch master set up to track remote branch master from origin.
```

- Tras un fin de semana de juerga, el lunes incorporamos las modificaciones del sábado con la misma instrucción que ya empleamos en su día.

```
git pull

# From github.com:carlosvergara/pintamapas
# * branch          master      -> FETCH_HEAD
# 4102eb9..acfa787  master      -> origin/master
# Updating 4102eb9..acfa787
# Fast-forward
#  README.md | 1 +
#  1 file changed, 1 insertion(+)
```

Sincronización: remoto a local (2/2)

- Trabajamos y al rato nos preguntamos qué hacemos con historias de la oficina un sábado por la tarde... decidimos guardar el contenido, subir los cambios al remoto y salir de juerga.

```
echo "Modificación del sábado" >> README.md
git add . && git commit -m "Décimo commit: se adelanta trabajo el sábado"
git push -u origin master
```

```
# [master acfa787] Décimo commit: se adelanta trabajo el sábado
# 1 file changed, 1 insertion(+)
# To github.com:carlosvergara/pintamapas.git
# 4102eb9..acfa787 master -> master
# Branch master set up to track remote branch master from origin.
```

- Tras un fin de semana de juerga, el lunes incorporamos las modificaciones del sábado con la misma instrucción que ya empleamos en su día.

```
git pull
```

```
# From github.com:carlosvergara/pintamapas
# * branch      master      -> FETCH_HEAD
# 4102eb9..acfa787 master    -> origin/master
# Updating 4102eb9..acfa787
# Fast-forward
#  README.md | 1 +
#  1 file changed, 1 insertion(+)
```

Del mismo modo que hemos trabajado por nuestra cuenta, y en caso de tratarse de un proyecto grupal con varios colaboradores, podríamos seguir exactamente los mismos pasos: subir nuestro contenido y descargar lo que otros colaboradores subieron previamente.

Clonando un repositorio remoto generamos una copia local de su contenido, lista para ser desarrollada. El comando básico es `git clone "opts"`.

Clonando un repositorio remoto generamos una copia local de su contenido, lista para ser desarrollada. El comando básico es `git clone "opts"`.

Por ejemplo, nuestro grupo ha desarrollado [un paquete de R](#) para establecer un esquema de directorios y diversas plantillas de informes estadísticos en proyectos de análisis. Imaginemos que deseamos colaborar en él añadiendo o modificando alguna función: nuestro primer paso será clonar el repositorio.

Clonando un repositorio remoto generamos una copia local de su contenido, lista para ser desarrollada. El comando básico es `git clone "opts"`.

Por ejemplo, nuestro grupo ha desarrollado [un paquete de R](#) para establecer un esquema de directorios y diversas plantillas de informes estadísticos en proyectos de análisis. Imaginemos que deseamos colaborar en él añadiendo o modificando alguna función: nuestro primer paso será clonar el repositorio.

```
git clone https://github.com/fisabio/fisabior ~/fisabior  
# Cloning into '/home/carlos/fisabior'...
```

Clonando un repositorio remoto generamos una copia local de su contenido, lista para ser desarrollada. El comando básico es `git clone "opts"`.

Por ejemplo, nuestro grupo ha desarrollado [un paquete de R](#) para establecer un esquema de directorios y diversas plantillas de informes estadísticos en proyectos de análisis. Imaginemos que deseamos colaborar en él añadiendo o modificando alguna función: nuestro primer paso será clonar el repositorio.

```
git clone https://github.com/fisabio/fisabior ~/fisabior
```

```
# Cloning into '/home/carlos/fisabior'...
```

Esta instrucción descarga todo el contenido del [remoto](#) en el directorio `~/docs/fisabior/`.

Clonando un repositorio remoto generamos una copia local de su contenido, lista para ser desarrollada. El comando básico es `git clone "opts"`.

Por ejemplo, nuestro grupo ha desarrollado [un paquete de R](#) para establecer un esquema de directorios y diversas plantillas de informes estadísticos en proyectos de análisis. Imaginemos que deseamos colaborar en él añadiendo o modificando alguna función: nuestro primer paso será clonar el repositorio.

```
git clone https://github.com/fisabio/fisabior ~/fisabior
```

```
# Cloning into '/home/carlos/fisabior'...
```

Esta instrucción descarga todo el contenido del [remoto](#) en el directorio `~/docs/fisabior/`.

Ahora ya podemos trabajar en el paquete en una copia local y, llegado el momento, realizar una solicitud de unión de nuestro trabajo con el repositorio remoto.

Opciones en GitHub

Pull Request (1/2)

Hasta ahora el trabajo se ha centrado en un repositorio propio, en el cual solo una persona hace y deshace a su voluntad. No obstante, la principal ventaja de Git es en la colaboración.

Hasta ahora el trabajo se ha centrado en un repositorio propio, en el cual solo una persona hace y deshace a su voluntad. No obstante, la principal ventaja de Git es en la colaboración.

Pongámonos en situación:

1. Hemos clonado un repositorio de GitHub en el que queremos colaborar.
2. Creamos una rama de desarrollo adaptada al tema que nos interesa.
3. Realizamos cambios en el código de una o varias funciones.
4. Probamos que los cambios introducidos son válidos.
5. Confirmamos los cambios mediante un commit en nuestra rama.
6. Subimos nuestros cambios a nuestro remoto en GitHub

Pull Request (1/2)

Hasta ahora el trabajo se ha centrado en un repositorio propio, en el cual solo una persona hace y deshace a su voluntad. No obstante, la principal ventaja de Git es en la colaboración.

Pongámonos en situación:

1. Hemos clonado un repositorio de GitHub en el que queremos colaborar.
2. Creamos una rama de desarrollo adaptada al tema que nos interesa.
3. Realizamos cambios en el código de una o varias funciones.
4. Probamos que los cambios introducidos son válidos.
5. Confirmamos los cambios mediante un commit en nuestra rama.
6. Subimos nuestros cambios a nuestro remoto en GitHub

Una vez hecho esto, podemos iniciar un proceso para que los cambios que hemos realizado sean aceptados en el repositorio original. Esto es conocido como Pull Request, y en GitHub todo el flujo de trabajo de equipo gira en torno a él.

Definición

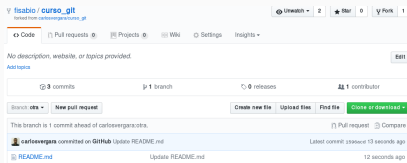
Un Pull Request es una solicitud al gestor de un repositorio para que, una vez que hemos editado su contenido de alguna forma (introduciendo una mejora o solucionando un *bug*), dichos cambios sean recogidos dentro de la rama de trabajo principal y formen parte activa del proyecto.

Para proyectos públicos, la forma más sencilla de acceso a un repositorio es mediante un fork, que es una copia del repositorio en tu cuenta de GitHub pero teniendo en cuenta el origen del mismo.

Pull Request (2/2)

Para proyectos públicos, la forma más sencilla de acceso a un repositorio es mediante un fork, que es una copia del repositorio en tu cuenta de GitHub pero teniendo en cuenta el origen del mismo.

1. En el momento en que se detecta una divergencia con el mismo se hace visible un botón que sugiere iniciar un Pull Request.



Pull Request (2/2)

Para proyectos públicos, la forma más sencilla de acceso a un repositorio es mediante un fork, que es una copia del repositorio en tu cuenta de GitHub pero teniendo en cuenta el origen del mismo.

1. En el momento en que se detecta una divergencia con el mismo se hace visible un botón que sugiere iniciar un Pull Request.
2. Al iniciar el proceso, se nos informa si las ramas pueden ser unidas de forma automática (algo deseable).

Comparing changes

Choose two branches to see what's changed or to start a new pull request. If you need to, you can also [compare across forks](#).

base fork: carloswegara/tutoro_git • base: eta ... head fork: finabla/tutoro_git • compare: eta

✓ Able to merge. These branches can be automatically merged.

Create pull request Discuss and review the changes in this comparison with others.

1 commit 1 file changed 0 commit comments 1 contributor

Comments on May 29, 2017

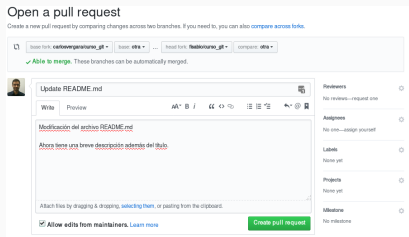
carloswegara Update README.md 1398ac

Showing 1 changed file with 1 addition and 1 deletion. United Split

Pull Request (2/2)

Para proyectos públicos, la forma más sencilla de acceso a un repositorio es mediante un fork, que es una copia del repositorio en tu cuenta de GitHub pero teniendo en cuenta el origen del mismo.

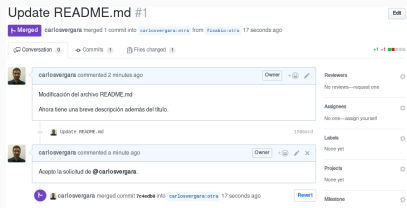
1. En el momento en que se detecta una divergencia con el mismo se hace visible un botón que sugiere iniciar un Pull Request.
2. Al iniciar el proceso, se nos informa si las ramas pueden ser unidas de forma automática (algo deseable).
3. Cuando se crea el Pull Request, introducimos un mensaje que sea útil y permita identificar fácilmente los cambios y su motivación.



Pull Request (2/2)

Para proyectos públicos, la forma más sencilla de acceso a un repositorio es mediante un fork, que es una copia del repositorio en tu cuenta de GitHub pero teniendo en cuenta el origen del mismo.

1. En el momento en que se detecta una divergencia con el mismo se hace visible un botón que sugiere iniciar un Pull Request.
2. Al iniciar el proceso, se nos informa si las ramas pueden ser unidas de forma automática (algo deseable).
3. Cuando se crea el Pull Request, introducimos un mensaje que sea útil y permita identificar fácilmente los cambios y su motivación.
4. Ahora le corresponde al propietario aceptar la solicitud y unir las ramas, solicitar más información al respecto o rechazarla.





El desarrollo de software no finaliza con el lanzamiento del producto: es raro que no aparezcan pequeños fallos (bugs) o que el programa no responda bien a un determinado problema (mejoras potenciales)

El desarrollo de software no finaliza con el lanzamiento del producto: es raro que no aparezcan pequeños fallos (bugs) o que el programa no responda bien a un determinado problema (mejoras potenciales)

1. Además, conforme un proyecto va creciendo y nuevos desarrolladores se involucran, es conveniente la gestión acerca de quién hace qué, cuándo y de qué manera.

Problems with parallel #2


 Closed gongardo opened this issue on 6 Apr · 1 comment


 gongardo commented on 6 Apr • edited by carlosvergara Collaborator +10

Anabel reportó que el paralelo le cuesta mucho tiempo. Aquí está el código que le daba problemas (corrido en una mac de 64bits)

```
p27 <- read.table("../RetumsSchooling.txt", header=FALSE, skip = 30)
p27 <- p27[,4:]
names(p27) <- c("y","educ","urban","capi2","mc","mc","b","all11y","mother_1","mother_2","%
object <- Priors(formula="y~.", data=p27, prior.betas="p27$11m", n.keep=10, L1se.test=0, prior.mean13
```

[RetumsSchooling.txt](#)

 gongardo assigned carlosvergara and gongardo on 6 Apr


 carlosvergara commented 13 days ago • edited Owner +10

Lo he probado en el servidor, tanto en la versión secuencial (`feynmanse1::feyn()`) como en paralelo con 40 núcleos (`feynmanse1::feyn(40)`), y todo parece correcto: la reducción de tiempo con la ejecución en paralelo entra dentro de lo razonable dada la cantidad de variables.



Concretamente, la versión secuencial (un núcleo) costó 10.81 horas, mientras que la versión en paralelo (cuarenta núcleos) se resolvió en 43 minutos. Las características del servidor son:

- Linux x86_64 (kernel 4.9.0-1, Debian 9.8),
- Intel Xeon ES-2683 2.10GHz (32 núcleos con 64 hilos),
- 250 GB de RAM,
- R 3.3.3.

Aunque la mejora pueda parecer pequeña (unas 15 veces más rápido y no 40), creo que es lo que cabría esperar como consecuencia de los overheads de comunicación entre nodos.

 carlosvergara closed this 13 days ago

Assignees

 carlosvergara
 gongardo

Labels

None yet


Projects

None yet

Milestones



No milestones


Notifications

 Unsubscribe

You're receiving notifications because you modified the open/close state.

2 participants

 Lock conversation

El desarrollo de software no finaliza con el lanzamiento del producto: es raro que no aparezcan pequeños fallos (bugs) o que el programa no responda bien a un determinado problema (mejoras potenciales)

1. Además, conforme un proyecto va creciendo y nuevos desarrolladores se involucran, es conveniente la gestión acerca de quién hace qué, cuándo y de qué manera.
2. Los issues en GitHub permiten mantener una comunicación sobre estos temas de forma centralizada, sin necesidad de intercambiar correos con varias personas, y pudiendo referenciar directamente tanto a colaboradores como commits del repositorio.

Problems with parallel #2

gongardo opened this issue on 6 Apr · 1 comment

gongardo commented on 6 Apr • edited by carlosvergara Collaborator +1

Anabel reportó que el paralelo le cuesta mucho tiempo. Aquí está el código que le daba problemas (corrido en una mac de 64bits)

```
p27 <- read.table("../RetornosSchooling.txt", header=FALSE, skip = 30)
p27 <- p27[,4:]
naes(p27) <- c("y","educ","urban","capi2","mc","tc","b","allily","mother_1","mother_2","n
object<- PValueFormula("y~.", data=p27, prior.beta="p27.11m", n.keep=10, L1se.test=0, prior.mean=0)
```

[RetornosSchooling.txt](#)

gongardo assigned carlosvergara and gongardo on 6 Apr

carlosvergara commented 15 days ago • edited Owner +1

Lo he probado en el servidor, tanto en la versión secuencial (`!systemctl stop p27`) como en paralelo con 40 núcleos (`!systemctl start p27`), y todo parece correcto: la reducción de tiempo con la ejecución en paralelo entra dentro de lo razonable dada la cantidad de variables.

Concretamente, la versión secuencial (un núcleo) costó 10.81 horas, mientras que la versión en paralelo (cuarenta núcleos) se resolvió en 43 minutos. Las características del servidor son:

- Linux x86_64 (kernel 4.9.0-1, Debian 9.8),
- Intel Xeon ES-2683 2.10GHz (32 núcleos con 64 hilos),
- 250 GB de RAM,
- R 3.3.3.

Aunque la mejora pueda parecer pequeña (unas 15 veces más rápido y no 40), creo que es lo que cabría esperar como consecuencia de los overheads de comunicación entre nodos.

carlosvergara closed this 13 days ago

Assignees

- carlosvergara
- gongardo

Labels

- None yet

Projects

- None yet

Milestones

- No milestones

Notifications

Unsubscribe

You're receiving notifications because you modified the open/close state.

2 participants

- gongardo
- carlosvergara

Lock conversation

El desarrollo de software no finaliza con el lanzamiento del producto: es raro que no aparezcan pequeños fallos (bugs) o que el programa no responda bien a un determinado problema (mejoras potenciales)

1. Además, conforme un proyecto va creciendo y nuevos desarrolladores se involucran, es conveniente la gestión acerca de quién hace qué, cuándo y de qué manera.
2. Los issues en GitHub permiten mantener una comunicación sobre estos temas de forma centralizada, sin necesidad de intercambiar correos con varias personas, y pudiendo referenciar directamente tanto a colaboradores como commits del repositorio.
3. El ejemplo de la derecha pertenece al un paquete de R que ha sido recientemente adaptado a GitHub: [BayesVarSel](#).

Problems with parallel #2

gongardo opened this issue on 6 Apr · 1 comment

gongardo commented on 6 Apr • edited by carlosvergara Collaborator

Anabel reportó que el paralelo le cuesta mucho tiempo. Aquí está el código que le daba problemas (corrido en una mac de 64bits)

```
p27 <- read.table("../RetornosSchooling.txt", header=FALSE, skip = 30)
p27 <- p27[,4:]
naes(p27) <- c("y","educ","urban","c4pct","mc","tc","b","allcity","mother_1","mother_2","%
object<- Priors(formula="y~", data=p27, prior.betas="gbl1lmer", n.keep=10, L1se.test=T, prior.mean1s
```

[RetornosSchooling.txt](#)

gongardo assigned carlosvergara and gongardo on 6 Apr

carlosvergara commented 15 days ago • edited Owner

Lo he probado en el servidor, tanto en la versión secuencial (`bayesvarsel::bvs()`) como en paralelo con 40 núcleos (`bayesvarsel::bvsP()`), y todo parece correcto: la reducción de tiempo con la ejecución en paralelo entra dentro de lo razonable dada la cantidad de variables.

Concretamente, la versión secuencial (un núcleo) costó 10.81 horas, mientras que la versión en paralelo (cuarenta núcleos) se resolvió en 43 minutos. Las características del servidor son:

- Linux x86_64 (kernel 4.9.0-1, Debian 9.8),
- Intel Xeon ES-2683 2.10GHz (32 núcleos con 64 hilos),
- 250 GB de RAM,
- R 3.3.3.

Aunque la mejora pueda parecer pequeña (unas 15 veces más rápida y no 40), creo que es lo que cabría esperar como consecuencia de los overheads de comunicación entre nodos.

carlosvergara closed this 13 days ago

Assignees

- carlosvergara
- gongardo

Labels

- None yet

Projects

- None yet

Milestones

- No milestones

Notifications

Unsubscribe

You're receiving notifications because you modified the open/close state.

2 participants

-

Lock conversation

En muchas ocasiones un documento `README.md` no es suficiente para mostrar todo lo que abarca un proyecto, y es ahí donde aparecen las wikis asociadas a cada repositorio.

En muchas ocasiones un documento **README.md** no es suficiente para mostrar todo lo que abarca un proyecto, y es ahí donde aparecen las wikis asociadas a cada repositorio.

Su finalidad es albergar información que no vaya a sufrir grandes modificaciones a medio- largo plazo. Tienen utilidades muy útiles, como:

En muchas ocasiones un documento **README.md** no es suficiente para mostrar todo lo que abarca un proyecto, y es ahí donde aparecen las wikis asociadas a cada repositorio.

Su finalidad es albergar información que no vaya a sufrir grandes modificaciones a medio- largo plazo. Tienen utilidades muy útiles, como:

- Crear un menú lateral de navegación.
- Agregar un pie de página personalizado.
- Edición directa desde GitHub (mostrando la previsualización del contenido formateado a HTML).

En muchas ocasiones un documento `README.md` no es suficiente para mostrar todo lo que abarca un proyecto, y es ahí donde aparecen las wikis asociadas a cada repositorio.

Su finalidad es albergar información que no vaya a sufrir grandes modificaciones a medio- largo plazo. Tienen utilidades muy útiles, como:

- Crear un menú lateral de navegación.
- Agregar un pie de página personalizado.
- Edición directa desde GitHub (mostrando la previsualización del contenido formateado a HTML).

En este [URL](#) puedes consultar un ejemplo de wiki muy bien terminada sobre el lenguaje de modelado bayesiano [Stan](#).

Pero esto va aún más allá: es posible generar gratuitamente una web estática de manera sencilla gracias a [GitHub Pages](#).

Pero esto va aún más allá: es posible generar gratuitamente una web estática de manera sencilla gracias a [GitHub Pages](#).

- Una forma perfecta de publicitar un currículum individual, grupal o la creación de un proyecto consiste en proporcionarle una ventana al mundo gracias a una página web.
 - No obstante, generar y mantener una web suele ser bastante costoso (HTML, CSS, JavaScript y PHP).

Pero esto va aún más allá: es posible generar gratuitamente una web estática de manera sencilla gracias a [GitHub Pages](#).

- Una forma perfecta de publicitar un currículum individual, grupal o la creación de un proyecto consiste en proporcionarle una ventana al mundo gracias a una página web.
 - No obstante, generar y mantener una web suele ser bastante costoso (HTML, CSS, JavaScript y PHP).
- Hay dos motores de páginas web estáticas que trabajan con texto en Markdown y ambos combinan perfectamente con GitHub Pages:
 1. [Jekyll](#).
 - Creado por el equipo desarrollador de GitHub. Solo tienes que escoger un [tema](#), crear el repositorio que albergará la web, y agregar contenido: [GitHub Pages](#) se encarga de transformar el texto Markdown en HTML, en base al tema que hayas escogido.
 2. [Hugo](#).
 - Desarrollado en Go, es más rápido y menos lioso que Jekyll, aunque su uso en GitHub Pages [requiere de dos pasos adicionales](#) además de la elección de un [tema](#).

Pero esto va aún más allá: es posible generar gratuitamente una web estática de manera sencilla gracias a [GitHub Pages](#).

- Una forma perfecta de publicitar un currículum individual, grupal o la creación de un proyecto consiste en proporcionarle una ventana al mundo gracias a una página web.
 - No obstante, generar y mantener una web suele ser bastante costoso (HTML, CSS, JavaScript y PHP).
- Hay dos motores de páginas web estáticas que trabajan con texto en Markdown y ambos combinan perfectamente con GitHub Pages:
 1. [Jekyll](#).
 - Creado por el equipo desarrollador de GitHub. Solo tienes que escoger un [tema](#), crear el repositorio que albergará la web, y agregar contenido: [GitHub Pages](#) se encarga de transformar el texto Markdown en HTML, en base al tema que hayas escogido.
 2. [Hugo](#).
 - Desarrollado en Go, es más rápido y menos lioso que Jekyll, aunque su uso en GitHub Pages [requiere de dos pasos adicionales](#) además de la elección de un [tema](#).

Recomendación

Aunque requiere de un mínimo trabajo extra, mi recomendación es que uséis Hugo, principalmente porque hay un paquete de R llamado [blogdown](#) que, pese a estar en fase de desarrollo, permite crear y mantener una web desde la consola de R, con una gran complicidad con RStudio.

1. Información acerca del repositorio.

- Crea siempre un documento **README.md** en el directorio raíz: GitHub lo transforma directamente en HTML en la página inicial del repositorio, y resulta muy útil antes de bucear en su contenido.

1. Información acerca del repositorio.

- Crea siempre un documento **README.md** en el directorio raíz: GitHub lo transforma directamente en HTML en la página inicial del repositorio, y resulta muy útil antes de bucear en su contenido.

2. Información acerca de actualizaciones.

- A medida que un repositorio crece es importante añadir un documento que recopile los grandes hitos del mismo, y ese es el papel del documento **NEWS.md**.

1. Información acerca del repositorio.

- Crea siempre un documento **README.md** en el directorio raíz: GitHub lo transforma directamente en HTML en la página inicial del repositorio, y resulta muy útil antes de bucear en su contenido.

2. Información acerca de actualizaciones.

- A medida que un repositorio crece es importante añadir un documento que recopile los grandes hitos del mismo, y ese es el papel del documento **NEWS.md**.

3. Información acerca de cómo contribuir al repositorio.

- Todos tenemos nuestras manías a la hora de escribir código y es conveniente informar de las mismas antes de iniciar una discusión. En el archivo **CONTRIBUTING.md** podemos vomitar todas nuestras neuras-nerds, como pudiera ser el uso de [espacios en lugar de tabulaciones](#) al [sangrar líneas](#) ([aviso](#): esto es café para *muyyyy* cafeteros).

1. Información acerca del repositorio.

- Crea siempre un documento **README.md** en el directorio raíz: GitHub lo transforma directamente en HTML en la página inicial del repositorio, y resulta muy útil antes de bucear en su contenido.

2. Información acerca de actualizaciones.

- A medida que un repositorio crece es importante añadir un documento que recopile los grandes hitos del mismo, y ese es el papel del documento **NEWS.md**.

3. Información acerca de cómo contribuir al repositorio.

- Todos tenemos nuestras manías a la hora de escribir código y es conveniente informar de las mismas antes de iniciar una discusión. En el archivo **CONTRIBUTING.md** podemos vomitar todas nuestras neuras-nerds, como pudiera ser el uso de [espacios en lugar de tabulaciones al sangrar líneas](#) (aviso: esto es café para *muyyyy* cafeteros).

4. Algo de empatía.

- En la medida de lo posible, facilita el *Pull Request* de manera que no haya conflictos: eso hará muy feliz al propietario del repositorio.

Muchas gracias por la atención

Referencias bibliográficas

Chacon, S. & Straub, B. (2014). *Pro Git* (2ª ed.). Apress.

Wickham, H. (2015). *Advanced R*. Boca Raton, FL: CRC.