



# CONTROL DE VERSIONES

Una introducción al uso de Git y Git Hub

## Resumen

Se presenta una breve descripción de los Sistemas de Control de Versiones (SCV) y de la estructura funcional de Git, con explicación de los comandos utilizados más comúnmente para el control de versiones local y remoto (GitHub)

**Ing. Marco Leon Mora Mendez**

SENA - Centro de Electricidad, Electrónica y Telecomunicaciones

## CONTENIDO

figuras .....	3
El Sistema de Control de Versiones - SCV .....	5
Clasificación de la arquitectura de un SCV .....	5
Local .....	5
Centralizado .....	6
Distribuido .....	6
Qué es Git .....	7
GitHub NO es Git .....	7
Instalación .....	7
Terminología de Git .....	8
Cómo funciona Git en repositorio local .....	10
Crear un repositorio local .....	10
Lo que no se incluirá .....	13
Deshacer cambios .....	14
Historial de cambios .....	15
Ramas alternas .....	16
Fusionar ramas .....	18
resolución de conflictos .....	19
Repositorio remoto .....	22
Crear un repositorio remoto .....	22
Subir cambios al remoto .....	23
Actualizar repositorio local desde remoto .....	24
Clonar un repositorio remoto .....	25
sugerencia del Flujo de trabajo y el uso de ramas .....	25
Sugerencia final .....	30
Anexo: Resumen de comandos .....	31
Instalar Git .....	31
GitHub para Windows .....	31
GitHub para Mac .....	31
Git para todas las plataformas .....	31
Configurar herramientas .....	31
Crear repositorios .....	31
Efectuar cambios .....	31

Cambios grupales .....	32
Refactorización de archivos.....	32
Suprimir el seguimiento de cambios .....	32
Guardar fragmentos .....	33
Repasar historial .....	33
Rehacer commits.....	33
Sincronizar cambios.....	34
Enlaces.....	35
CONTROL DE CAMBIOS .....	36

## FIGURAS

Figura 1. SCV en computador local.....	5
Figura 2. SCV centralizado .....	6
Figura 3. SCV distribuido .....	6
Figura 4. Instalando Git .....	8
Figura 5. Estructura de archivos de un proyecto.....	10
Figura 6. Resultado del comando 'git status'.....	11
Figura 7. Diagrama de estados de un archivo .....	12
Figura 8. Archivos en diferentes estados .....	13
Figura 9. Ejemplo del archivo '.gitignore' .....	14
Figura 10. Dos archivos en "staging area" .....	14
Figura 11. Deshacer cambios con 'git reset' .....	15
Figura 12. Deshacer un commit con 'git revert' .....	15
Figura 13. Mostrando el historial de cambios .....	16
Figura 14. Vista de varias ramas en Git. ....	16
Figura 15. Creando una nueva rama .....	17
Figura 16. Realizando cambios en la rama auxiliar.....	17
Figura 17. Sin cambios en "master".....	18
Figura 18. Mezclando sobre "master" .....	18
Figura 19. Resultado de la mezcla .....	19
Figura 20. Cambios para crear conflicto.....	19
Figura 21. Se presenta un conflicto .....	20
Figura 22. El editor muestra el conflicto .....	20
Figura 23. Resolviendo el conflicto.....	21
Figura 24. Creando un repositorio remoto.....	22
Figura 25. Subiendo Archivos al repositorio remoto.....	23
Figura 26. LA URL del proyecto.....	23
Figura 27. Verificando el repositorio remoto .....	23
Figura 28. Subiendo contenido al remoto .....	24
Figura 29. Repositorio remoto con copia del local.....	24
Figura 30. Bajando contenido del remoto.....	25
Figura 31. Arquitectura típica con Git Hub.....	25
Figura 32. Flujo de trabajo sugerido.....	26
Figura 33. Ramas principales.....	27

Figura 34. Rama de característica .....	28
Figura 35. Rama de corrección de errores .....	29

## EL SISTEMA DE CONTROL DE VERSIONES - SCV

“El control de versiones o "control de código fuente", es la práctica de gestionar los cambios en el código de software. Los sistemas de control de versiones (SCV) son herramientas de software que ayudan a los equipos de desarrollo de software a gestionar los cambios en el código fuente a lo largo del tiempo. A medida que los entornos de desarrollo se aceleran, los sistemas de control de versiones ayudan a los equipos de software a trabajar de forma más rápida e inteligente”. El SCV registra los cambios realizados en un archivo o conjunto de archivos a lo largo del tiempo, de modo que se pueda recuperar versiones específicas más adelante.

“El software de control de versiones realiza un seguimiento de todas las modificaciones en el código. Si se comete un error, los desarrolladores pueden ir hacia atrás en el tiempo y comparar las versiones anteriores del código para ayudar a resolver el error, al tiempo que se minimizan las interrupciones para todos los miembros del equipo”<sup>1</sup>.

Las principales ventajas de un SCV son:

1. Un completo historial de cambios a largo plazo de todos los archivos
2. Creación de ramas y fusiones
3. Trazabilidad

## CLASIFICACIÓN DE LA ARQUITECTURA DE UN SCV<sup>2</sup>

### LOCAL

Con el control de versiones local, todos los datos del proyecto se almacenan en una sola computadora y los cambios realizados en los archivos del proyecto se almacenan como versiones. Cada revisión contiene solamente las actualizaciones implementadas desde la versión anterior.

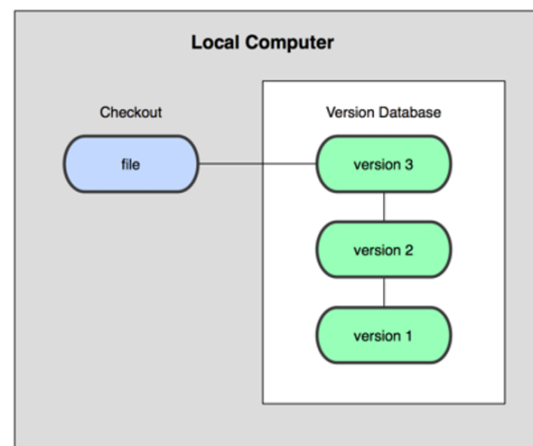


Figura 1. SCV en computador local

<sup>1</sup> <https://www.atlassian.com/es/git/tutorials/what-is-version-control>

<sup>2</sup> Bachiller, S.G. Introducción a Git y Git Hub, en

---

## CENTRALIZADO

En el CV centralizado existe un repositorio centralizado de todo el código, del cual es responsable un único usuario (o conjunto de ellos). Se facilitan las tareas administrativas a cambio de reducir flexibilidad, pues todas las decisiones fuertes (como crear una nueva rama) necesitan la aprobación del responsable. Algunos ejemplos son CVS (<https://cvs.nongnu.org/>) y Apache Subversion (<https://subversion.apache.org/>).

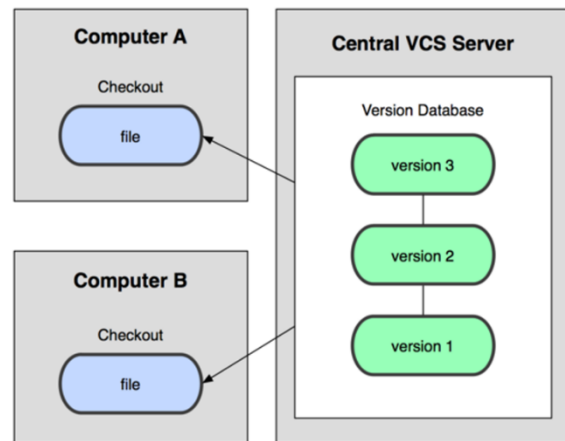


Figura 2. SCV centralizado

---

## DISTRIBUIDO

Cada usuario tiene su propio repositorio. Los distintos repositorios pueden intercambiar y mezclar revisiones entre ellos. Es frecuente el uso de un repositorio, que está normalmente disponible, que sirve de punto de sincronización de los distintos repositorios locales. Ejemplos: Git ([https:// git-scm.com/](https://git-scm.com/)) y Mercurial CSM (<https://www.mercurial-scm.org/>).

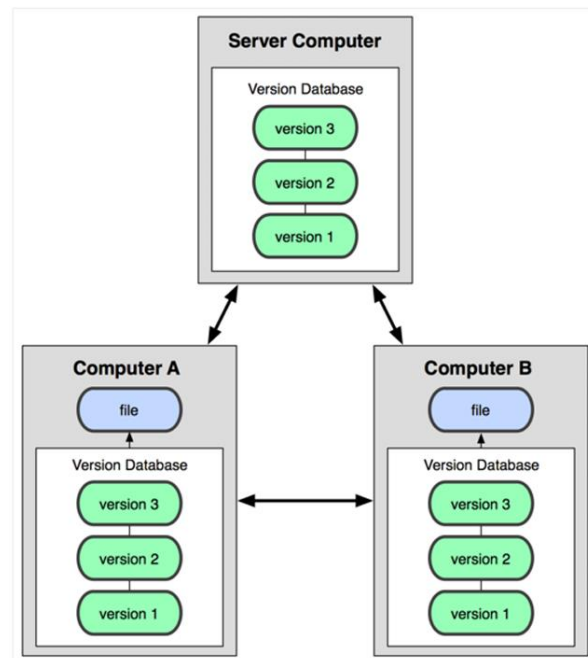


Figura 3. SCV distribuido

## QUÉ ES GIT

Git es el sistema de control de versiones moderno más utilizado del mundo. Git es un proyecto de código abierto que desarrolló originalmente Linus Torvalds, el famoso creador del *kernel* del sistema operativo Linux, en 2005. Este sistema funciona a la perfección en una amplia variedad de sistemas operativos e IDE (entornos de desarrollo integrados).

Git presenta una arquitectura distribuida, es un ejemplo de DVCS (sistema de control de versiones distribuido, por sus siglas en inglés). En lugar de tener un único espacio para todo el historial de versiones del software, como CVS o Subversion (también conocido como SVN), en Git, la copia de trabajo del código de cada desarrollador es también un repositorio que puede albergar el historial completo de todos los cambios.

Además de contar con una arquitectura distribuida, Git se ha diseñado teniendo en cuenta el rendimiento, la seguridad y la flexibilidad.<sup>3</sup>

## GITHUB NO ES GIT

Git es un software de VCS local que permite a los desarrolladores guardar instantáneas de sus proyectos a lo largo del tiempo. Generalmente es mejor para uso individual.

GitHub es una plataforma basada en la web que incorpora las características de control de versiones de Git para que puedan ser utilizadas de forma colaborativa. También incluye características de gestión de proyectos y equipos, así como oportunidades para la creación de redes y la codificación social.

## INSTALACIÓN

En el sitio <https://www.atlassian.com/es/git/tutorials/install-git> se encuentra una buena guía de instalación de Git, para diversos sistemas operativos. Acá se expone el procedimiento para su instalación en el S.O. Windows.

Para descargar Git, puede ir a la página oficial en <https://git-scm.com/downloads>, o al sitio <https://gitforwindows.org/>. Descargar la última versión del instalador pulsando el botón “Download”.

Ejecute el instalador descargado, le pedirá autorización para instalar, debe aceptar.

---

<sup>3</sup> <https://www.atlassian.com/es/git/tutorials/what-is-git>

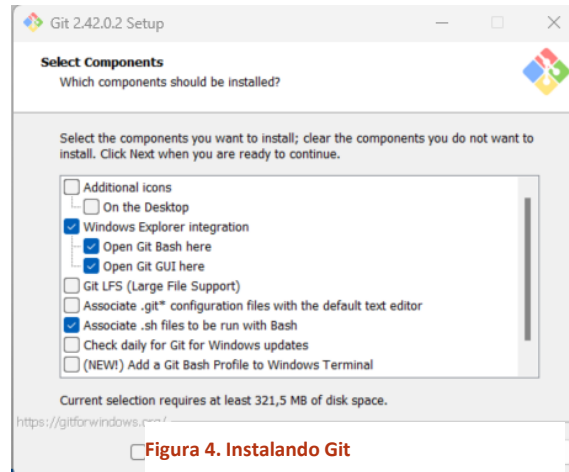


Clic en “Next”, verifique que estén activas las opciones “Open Git Bash here” y “Open Git GUI here”.

La primera es un emulador para Windows que despliega una consola de líneas de comando al estilo Linux.

La segunda es una Interfaz Gráfica de Usuario (GUI) que facilita el uso de Git.

En adelante haremos uso de la línea de comandos.



Continúe con la instalación, dando clic en el botón “Next”, dejando las demás opciones como están configuradas por defecto.

## TERMINOLOGÍA DE GIT

Se presenta a continuación los términos más utilizados, como referencia para las explicaciones posteriores.

### "BRANCH" (RAMA)

Un conjunto de archivos puede ser ramificado o bifurcado en un punto en el tiempo de manera que, a partir de ese momento, dos copias de esos archivos se pueden desarrollar a velocidades diferentes o en formas diferentes e independientes el uno del otro.

### "CHANGE" (CAMBIO)

Un cambio (o delta) representa una modificación específica de un documento bajo el control de versiones.

### "CHECKOUT" (DESPLEGAR)

Es crear una copia de trabajo local desde el repositorio. Un usuario puede especificar una revisión en concreto u obtener la última. El término '*checkout*' también se puede utilizar como un sustantivo para describir la copia de trabajo.

### "COMMIT" (CONFIRMAR)

Confirmar es escribir o mezclar los cambios realizados en la copia de trabajo del repositorio. Los términos 'commit' y 'checkin' también se pueden utilizar como sustantivos para describir la nueva revisión que se crea como resultado de confirmar.

un commit es un conjunto de cambios en los archivos que hemos dado por buenos y que queremos almacenar como una instantánea de cara al futuro. Los commits se relacionan unos con otros en una o varias secuencias para poder ir viendo la historia de un determinado archivo a lo largo del tiempo. Es el concepto central de todo sistema de control de código.

## "CONFLICT" (CONFLICTO)

Un conflicto se produce cuando diferentes partes realizan cambios en el mismo documento y el sistema es incapaz de conciliar los cambios. **Un usuario debe resolver el conflicto mediante la integración de los cambios, o mediante la selección de un cambio en favor del otro.**

## "HEAD" (CABEZA)

El concepto de HEAD se refiere al commit en el que está tu repositorio posicionado en cada momento. Por regla general HEAD suele coincidir con el último commit de la rama en la que estés, ya que habitualmente estás trabajando en lo último. HEAD es un nombre constante, refiriéndose siempre al *commit* en el que tenemos posicionado el repositorio en el momento actual. Independientemente de la rama en la que estemos, de cómo se llame la rama principal o el repositorio principal remoto.

## "MAIN" (PRINCIPAL)

Por regla general a *main* se la considera la rama principal y la raíz de la mayoría de las demás ramas. Lo más habitual es que en *main* se encuentre el "código definitivo", que luego va a producción, y es **la rama en la que se mezclan todas las demás tarde o temprano** para dar por finalizada una tarea e incorporarla al producto final (ver "MASTER").

## "MASTER" (MAESTRO)

No existe una diferencia práctica entre "MAIN" y "MASTER". Son solo nombres de ramas. Git siempre ha predeterminado que la rama principal se llame "MASTER". Pero algunos proveedores como Github han cambiado el nombre de su rama predeterminada a "MAIN".

## "MERGE" (FUSIONAR, INTEGRAR, MEZCLAR)

Una fusión o integración es una operación en la que se aplican dos tipos de cambios en un archivo o conjunto de archivos. Algunos escenarios de ejemplo son los siguientes:

- Un usuario, trabajando en un conjunto de archivos, actualiza o sincroniza su copia de trabajo con los cambios realizados y confirmados, por otros usuarios, en el repositorio.
- Un usuario intenta confirmar archivos que han sido actualizado por otros usuarios desde el último despliegue ('checkout'), y el software de control de versiones integra automáticamente los archivos (por lo general, después de preguntarle al usuario si se debe proceder con la integración automática, y en algunos casos sólo se hace si la fusión puede ser clara y razonablemente resuelta).
- Un conjunto de archivos se bifurca, un problema que existía antes de la ramificación se trabaja en una nueva rama, y la solución se combina luego en la otra rama.
- Se crea una rama, el código de los archivos es independiente editado, y la rama actualizada se incorpora más tarde en un único tronco unificado.

## "ORIGIN" (ORIGEN)

*origin* es simplemente el nombre predeterminado que recibe el repositorio remoto principal contra el que trabajamos. Cuando clonamos un repositorio por primera vez desde GitHub o cualquier otro sistema remoto, el nombre que se le da a ese repositorio "maestro" es precisamente *origin*.

## "REPOSITORY" (REPOSITORIO)

El repositorio es el lugar en el que se almacenan los datos actualizados e históricos de cambios. Puede ser un repositorio local o remoto.

### "REVISION" (REVISIÓN)

Una revisión es una versión determinada del código bajo control de versiones. Hay sistemas que identifican las revisiones con un contador (Ej. subversion). Hay otros sistemas que identifican las revisiones mediante un código de detección de modificaciones (Ej. Git usa SHA1).

### "TAG" (ETIQUETA)

Los tags permiten identificar de forma fácil revisiones importantes en el proyecto. Por ejemplo, se suelen usar tags para identificar el contenido de las versiones publicadas del proyecto.

### "TRUNK" (TRONCO)

La única línea de desarrollo que no es una rama (a veces también llamada línea base, línea principal o máster). Comúnmente a la rama principal, se le denomina "MAIN" o "MASTER".

## CÓMO FUNCIONA GIT EN REPOSITORIO LOCAL

En este apartado se describen las diferentes operaciones a realizar para el control de versiones en el repositorio local.

### CREAR UN REPOSITORIO LOCAL

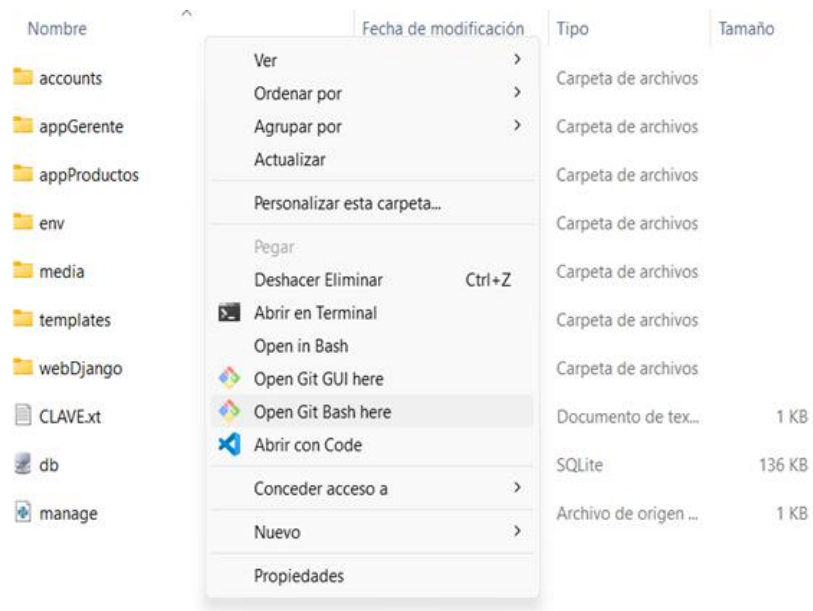


Figura 5. Estructura de archivos de un proyecto

Una vez instalado Git, el procedimiento normal para crear el repositorio es el siguiente:

En el directorio del proyecto a desarrollar se crean los archivos iniciales, por ejemplo, se podría tener el proyecto presentado en la figura 6, con algunas carpetas y archivos.

Desplegar el menú contextual y seleccionar "Open Git Bash here", se desplegará la consola de comandos.

Si aún no lo ha hecho, es el momento de registrar dos variables globales en Git, con su nombre de usuario y correo. (El símbolo \$ no se digita, el aparece en la línea de comandos de la consola)

```
$ git config --global user.name "John Doe"
```

```
$ git config --global user.email johndoe@example.com
```

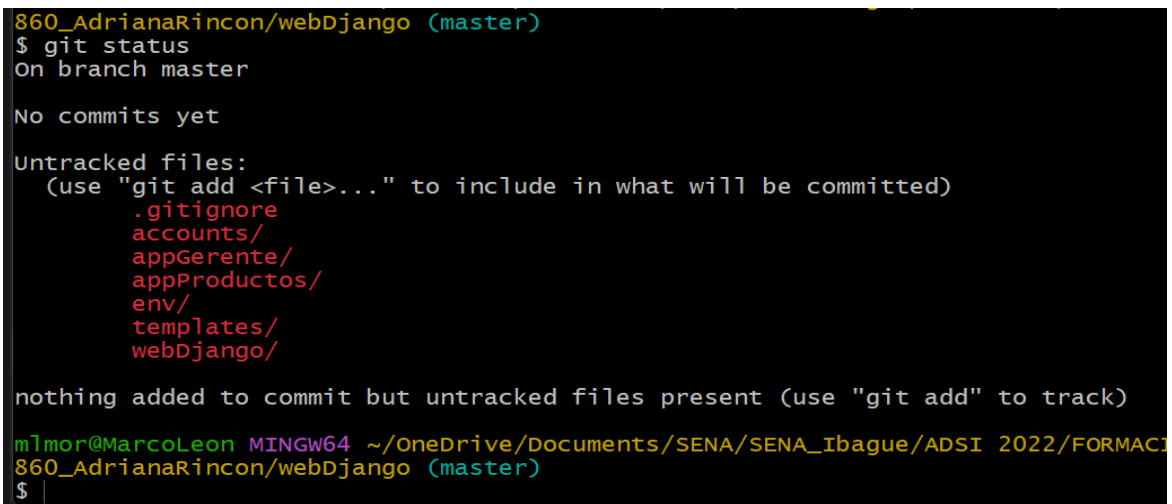
En la consola, escribir el comando:

```
$ git init
```

Con este comando se creará una carpeta ". git" con su estructura de carpetas y archivos internos.

Es una buena práctica en este punto crear un archivo ".gitignore" indicando los archivos y carpetas que no se incluirán en el repositorio (ver más adelante )

El comando `$ git status` mostrara el estado actual de los archivos en relación con el control de versiones. En la siguiente figura, observar el resultado de este comando:



```
860_AdrianaRincon/webDjango (master)
$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        .gitignore
        accounts/
        appGerente/
        appProductos/
        env/
        templates/
        webDjango/

nothing added to commit but untracked files present (use "git add" to track)
m1mor@MarcoLeon MINGW64 ~/OneDrive/Documents/SENA/SENA_Ibague/ADSI_2022/FORMACI
860_AdrianaRincon/webDjango (master)
$
```

Figura 6. Resultado del comando 'git status'

- Indica la rama activa: 'master'.
- La lista en rojo señala los archivos o carpetas a los que no se ha realizado versionado.
- Finalmente, sugiere el siguiente comando a ejecutar.
- Observar que, en la línea de la ruta (en amarillo), también se muestra el usuario y la rama activa (en la figura es master).

Para consultar un comando, use `$ git help <nombre-comando>` desplegará en un navegador la ayuda.

En la consola, ingresar los comandos:

```
$ git add -A
$ git commit -m "Commit inicial"
$ git status
```

Este último comando mostrará el mensaje “nothing to commit, working tree clean”. Señalando que, desde la anterior confirmación no se ha realizado ningún cambio. En este momento ya tenemos el repositorio con los archivos a los que se les hará seguimiento.

---

## EL ÁRBOL DE TRABAJO

El árbol de trabajo de Git hace referencia a los diferentes estados en que se pueden encontrar los archivos en el proceso de control de versiones.

La siguiente figura muestra que un archivo puede estar en el estado “*working area*” cuando, dentro de la rama de trabajo actual, ha sido modificado (o cuando es un nuevo archivo). El comando `git status` lo mostrara en color rojo.

En la figura no se muestra un estado adicional denominado “Stash”, para ocultar modificaciones en proceso, cuando se está trabajando en otra cosa, utilizado por usuarios más avanzados.

El repositorio remoto mostrado en la figura se tratará en la segunda parte de este documento.

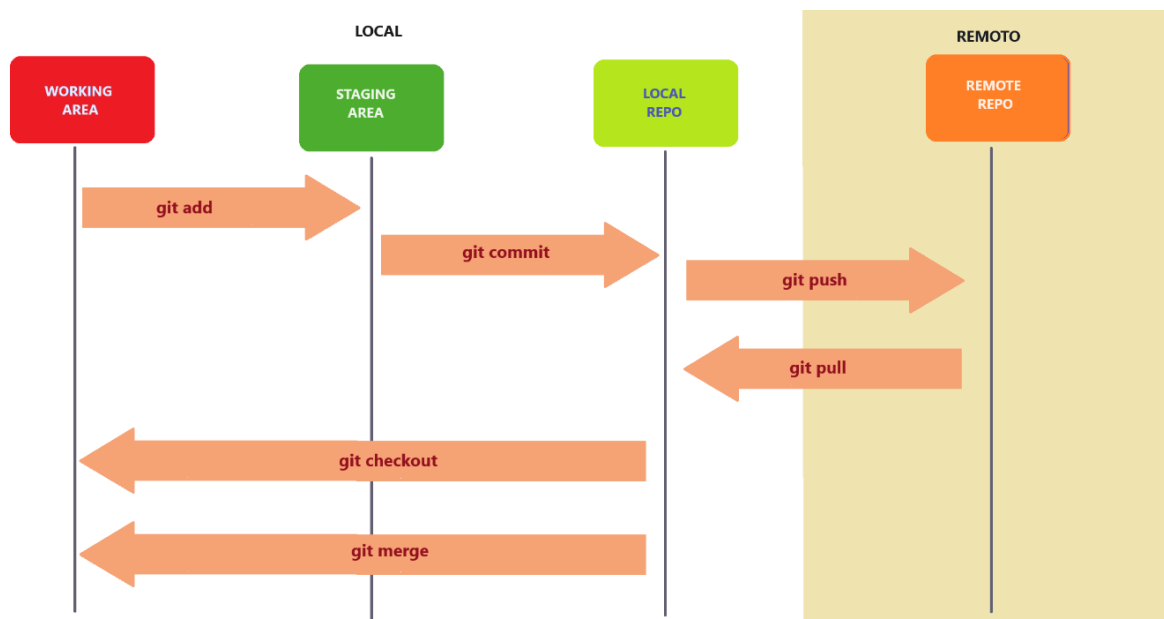
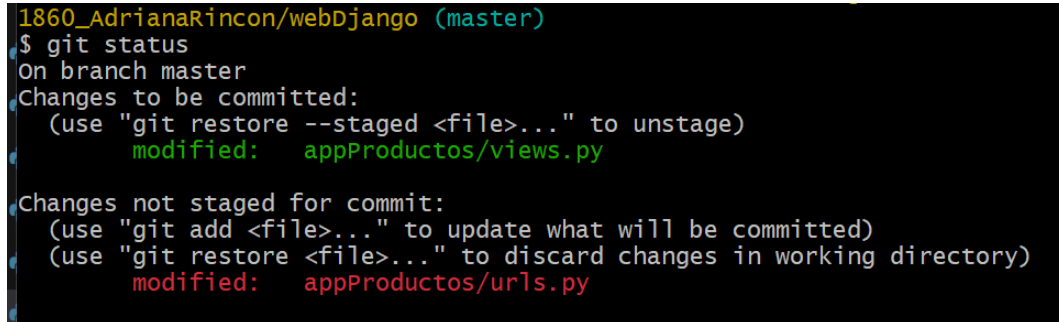


Figura 7. Diagrama de estados de un archivo

El comando `git add` le dice a Git que desea incluir las actualizaciones de un archivo en particular en la próxima confirmación. Sin embargo, `git add` realmente no afecta al repositorio de manera significativa; los cambios en realidad no se registran hasta que ejecuta `git commit`.

En nuestro repositorio de prueba, se verían así los archivos que han sido modificados:



```
1860_AdrianaRincon/webDjango (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   appProductos/views.py

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   appProductos/urls.py
```

Figura 8. Archivos en diferentes estados

El comando `git status` desplegará, en color verde, los archivos que ha sido modificados y que ya han pasado al estado intermedio con el comando `git add`.

El archivo “appProductos/urls.py” en rojo, indica que ha sido modificado, pero aún no ha pasado al estado intermedio.

El comando `git commit` pasará solo los archivos de “staging área” al repositorio local (los listados en verde).

Cuando se cambia de rama (`git checkout`) o se mezcla una rama en otra (`git merge`), los archivos de la nueva rama activa podrán ser editados, es decir estarán en “working area”.

---

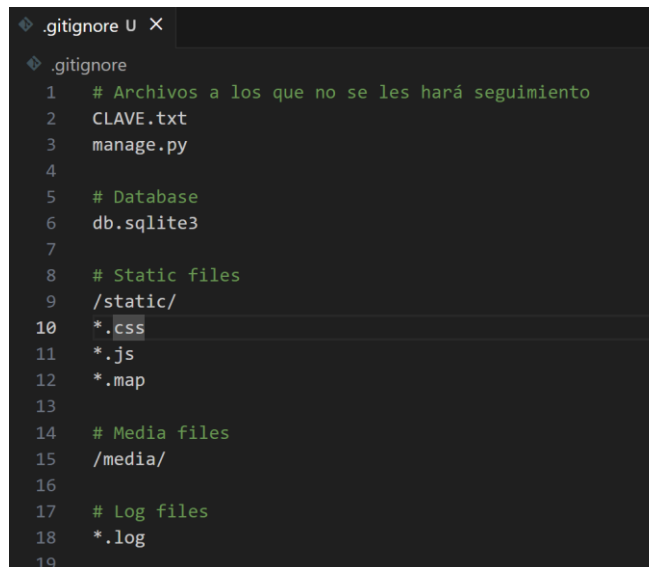
## LO QUE NO SE INCLUIRÁ

Cuando se deseen excluir archivos o carpetas del control de versiones, se deben registrar sus nombre en un archivo especial denominado “.gitignore”.

Se incluyen las carpetas y/o archivos que consideren no se les debe hacer versionado. En el ejemplo de la figura 5, que muestra un directorio de trabajo, no se incluirán las carpetas “env”, “media” ni los archivos “CLAVE.txt”, “db.sqlite3” ni “manage.py”, pues son objetos que no se modificaran en el proceso de desarrollo. Tampoco los archivos de tipo ‘css’, ‘js’, ‘log’.

Para una explicación del archivo “.gitignore” consulte, por ejemplo:

<https://www.atlassian.com/git/tutorials/saving-changes/gitignore#:~:text=gitignore%20files%20contain%20patterns%20that,gitignore%20files%20in%20your%20repository>



```

.gitignore
1  # Archivos a los que no se les hará seguimiento
2  CLAVE.txt
3  manage.py
4
5  # Database
6  db.sqlite3
7
8  # Static files
9  /static/
10 *.css
11 *.js
12 *.map
13
14 # Media files
15 /media/
16
17 # Log files
18 *.log
19

```

Figura 9. Ejemplo del archivo '.gitignore'

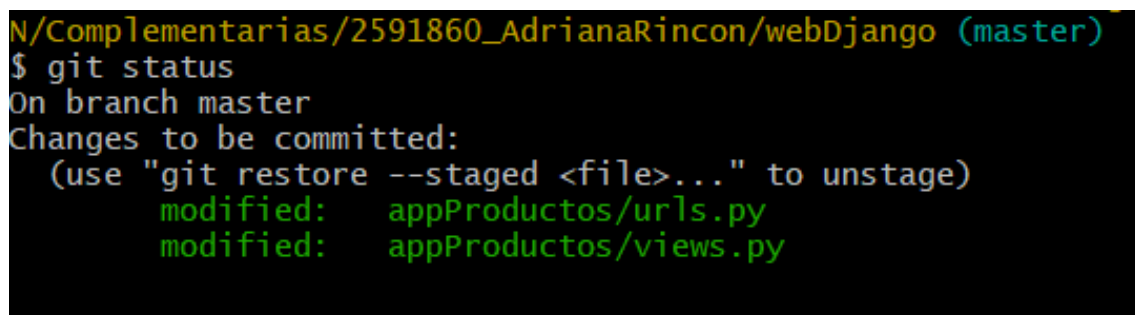
---

## DESHACER CAMBIOS

Hay dos posibilidades, archivos modificados que ya estén en el estado intermedio - “staging área” o archivos a los que ya se les haya hecho el “commit” y sus cambios ya estén registrados en el área de repositorio local.

---

## DESHACER CAMBIOS ANTES DEL “COMMIT”



```

N/Complementarias/2591860_AdrianaRincon/webDjango (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   appProductos/urls.py
        modified:   appProductos/views.py

```

Figura 10. Dos archivos en "staging area"

Como ejemplo, en la figura anterior se muestran dos archivos pendientes de “commit”, vamos a deshacer los cambios realizados en el archivo “urls.py”.

```
$ git reset appProductos/urls.py
```

Regresa el archivo a working area.

Con un nuevo comando `$ git status` podemos observar que el archivo ahora lo muestra en rojo:

```

ntarias/2591860_AdrianaRincon/webDjango (master)
$ git reset appProductos/urls.py
Unstaged changes after reset:
M      appProductos/urls.py

mlmor@MarcoLeon MINGW64 ~/OneDrive/Documents/SENA/SENA_Ibague/ADSI_2022/FO
ntarias/2591860_AdrianaRincon/webDjango (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   appProductos/views.py

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   appProductos/urls.py

```

Figura 11. Deshacer cambios con 'git reset'

---

## DESHACER COMMITS

El siguiente comando:

```
$ git revert HEAD --no-edit
```

Revierte el ultimo commit.

```

ntarias/2591860_AdrianaRincon/webDjango (master)
$ git revert HEAD --no-edit
[master c0ae445] Revert "Segundos cambios de prueba"
Date: Sat Nov 11 11:37:42 2023 -0500
2 files changed, 3 deletions(-)

```

Figura 12. Deshacer un commit con 'git revert'

---

## HISTORIAL DE CAMBIOS

Para ver el historial use el comando:

```
$ git log
```

Despliega los commits realizados.



```

ntarias/2591860_AdrianaRincon/webDjango (master)
$ git log
commit c0ae4457769a4664551d533a24a729428c608d55 (HEAD -> master)
Author: marcoleonmora <mlmoram@misena.edu.co>
Date: Sat Nov 11 11:37:42 2023 -0500

    Revert "Segundos cambios de prueba"

    This reverts commit 3eb77c5234e49bdc7c99922a408ee1182e9b9569.

commit 3eb77c5234e49bdc7c99922a408ee1182e9b9569
Author: marcoleonmora <mlmoram@misena.edu.co>
Date: Sat Nov 11 11:32:17 2023 -0500

    Segundos cambios de prueba

commit 2fa08870e828c75fda174ee76f9957080eb185b0
Author: marcoleonmora <mlmoram@misena.edu.co>
Date: Fri Nov 10 18:55:39 2023 -0500

    Commit inicial

```

Figura 13. Mostrando el historial de cambios

En la figura anterior se observan los dos ‘commits’ y el ‘revert’ realizados.

## RAMAS ALTERNAS

los sistemas de control de código fuente como Git trabajan con el concepto de **ramas** (o *branches* en inglés). Una rama no es más que un nombre que se da a un *commit*, a partir del cual se empieza a trabajar de manera independiente y con el que se van a enlazar nuevos *commits* de esa misma rama. Las ramas pueden mezclarse de modo que todo el trabajo hecho en una de ellas pase a formar parte de otra rama<sup>4</sup>

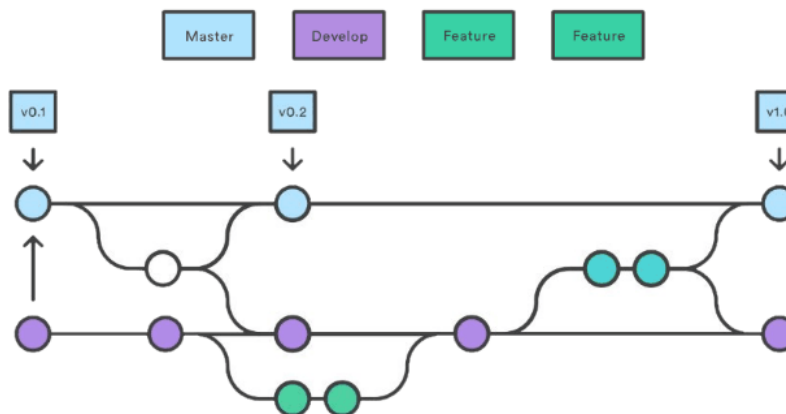


Figura 14. Vista de varias ramas en Git.

En la figura anterior se observa la evolución, en el tiempo, de izquierda a derecha, de varias ramas en un repositorio.

<sup>4</sup> <https://www.campusmvp.es/recursos/post/git-los-conceptos-de-master-origin-y-head.aspx>

1. En el ejemplo, desde la rama "Develop" se hace "commit" y se crea una rama "Master" a la que se le agrega una etiqueta "v0.1",
2. De la rama "Master" se ha creado una rama (circulo blanco) para corregir algún error, mientras tanto, en "develop" se ha realizado un "commit" (segundo circulo morado).
3. De la rama "Develop" se ha creado una nueva rama "Feature" para agregar una nueva funcionalidad el proyecto. La rama auxiliar (blanco) se ha mezclado ("merge") tanto con "Master", como con "Develop" para asegurar la corrección del error en las dos ramas. Se elimina la rama auxiliar. Se ha agregado una etiqueta "v0.2" a "Master".
4. Con el desarrollo de la nueva funcionalidad, se ha realizado un "commit" (segundo circulo verde)
5. Esta rama se ha mezclado con "Develop", la rama verde se elimina.
6. Se crea una nueva rama (azul) desde "Develop".
7. Se realiza un "Commit" en la rama azul y posteriormente se fusiona tanto con "Master", como con "Develop". En "Master" se coloca una etiqueta "v1.0".

En nuestro código de ejemplo, vamos a crear una rama denominada "Prueba-1":

```
$ git checkout Prueba-1
```

Se crea la rama "Prueba-1" y se cambia a la nueva rama ("Working area").

Nota: Con "git Branch Prueba-1" se crea la rama, pero no se cambia a ella.

```
FORMACION/Complementarias/2591860_AdrianaRincon/webDjango (master)
$ git checkout Prueba-1
Switched to branch 'Prueba-1'

mImor@MarcoLeon MINGW64 ~/OneDrive/Documents/SENA/SENA_Ibague/ADSI_2022/
FORMACION/Complementarias/2591860_AdrianaRincon/webDjango (Prueba-1)
$ |
```

Figura 15. Creando una nueva rama

En la nueva rama se realizan algunos cambios en el archivo "urls.py" y se guardan los cambios:

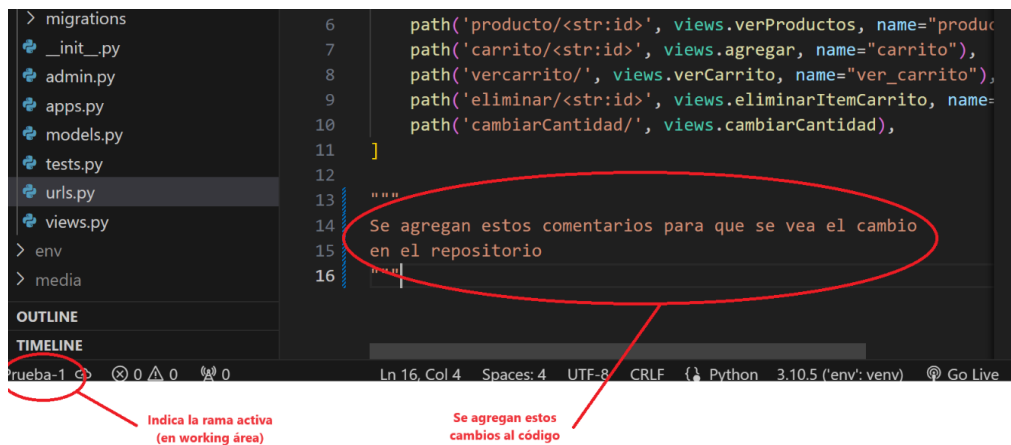


Figura 16. Realizando cambios en la rama auxiliar

```
$ git status
```

Visualiza en rojo el archivo modificado en la nueva rama.

```
$ git add -A
```

```
$ git commit -m "Cambios realizados en rama auxiliar"
```

```
$ git checkout master
```

Si se cambia a la rama "Master".

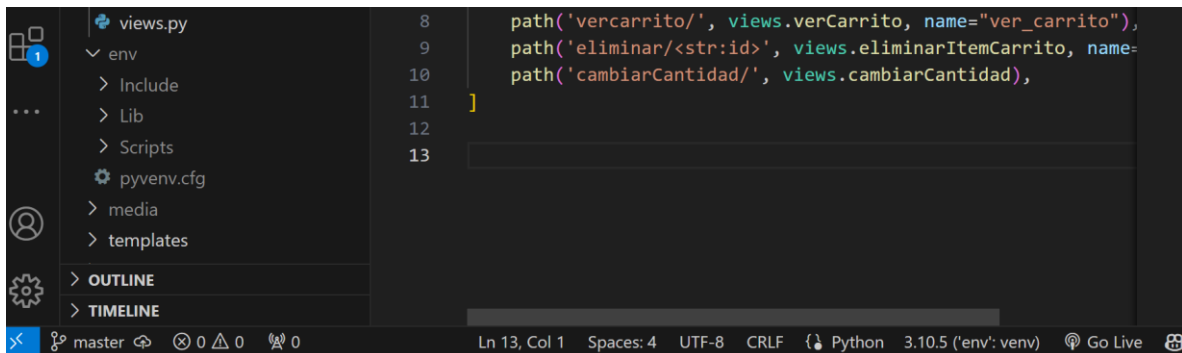


Figura 17. Sin cambios en "master"

En el archivo "urls.py" de la rama "master" no hay cambios.

---

## FUSIONAR RAMAS

Ahora, se fusionarán los cambios realizados en la rama auxiliar, sobre la rama "master". Debe estar en "master":

```
$ git merge Prueba-1
```

```
FORMACION/Complementarias/2591860_AdrianaRincon/webDjango (master)
$ git merge Prueba-1
Updating c0ae445..ddb14
Fast-forward
 appProductos/urls.py | 5 ++++-
 1 file changed, 4 insertions(+), 1 deletion(-)

mlmor@MarcoLeon MINGW64 ~/OneDrive/Documents/SENA/SENA_Ibague/ADSI 2022/
FORMACION/Complementarias/2591860_AdrianaRincon/webDjango (master)
$
```

Figura 18. Mezclando sobre "master"

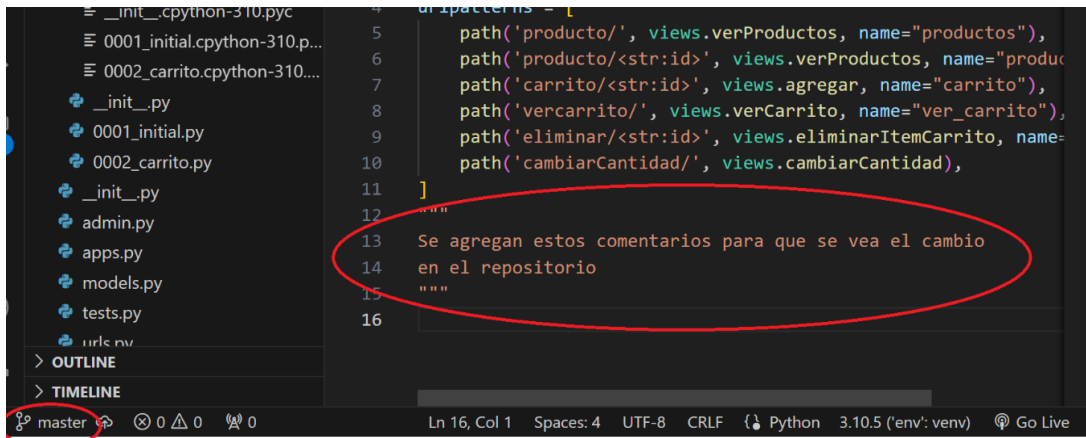


Figura 19. Resultado de la mezcla

Los cambios se reflejan en la rama “master”.

## RESOLUCIÓN DE CONFLICTOS

En algunas ocasiones, los procesos de fusión no suelen ser fluidos. Si hay modificaciones dispares en una misma porción de un mismo archivo en las dos ramas distintas que se pretende fusionar, Git no será capaz de fusionarlas directamente.

Esta situación de conflicto es una de las razones por lo que se recomienda que no se trabaje el desarrollo de las mismas características o la solución de errores en dos ramas diferentes al mismo tiempo.

A modo de ejemplo, para forzar un conflicto realizaremos, en las dos ramas, diferentes cambios en las mismas líneas de código del archivo, haciendo “commit” en las dos ramas y posteriormente se hará el procedimiento de fusión.

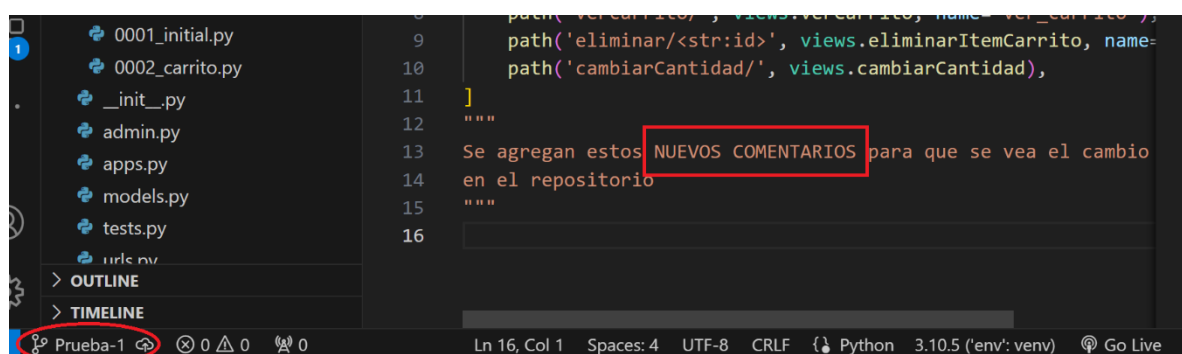


Figura 20. Cambios para crear conflicto

Después de realizar el “add” y el “commit”, se pasa a la rama “master” y se intenta el “merge”.

```
$ git checkout Prueba-1
```

```
$ git add -A
```

```
$ git commit -m "Cambios realizados en Prueba-1 para crear un conflicto"
```

```
$ git checkout master
$ git add -A
$ git commit -m "Cambios realizados en master para crear un conflicto"

$ git merge Prueba-1
```

Da el siguiente resultado:

```
mImor@MarcoLeon MINGW64 ~/OneDrive/Documents/SENA/SENA_Ibague/ADSI_2022/
FORMACION/Complementarias/2591860_AdrianaRincon/webDjango (master)
$ git merge Prueba-1
Auto-merging appProductos/urls.py
CONFLICT (content): Merge conflict in appProductos/urls.py
Automatic merge failed; fix conflicts and then commit the result.

mImor@MarcoLeon MINGW64 ~/OneDrive/Documents/SENA/SENA_Ibague/ADSI_2022/
FORMACION/Complementarias/2591860_AdrianaRincon/webDjango (master|MERGIN
G)
```

Figura 21. Se presenta un conflicto

En el editor (Visual Studio Code) se muestra en detalle el conflicto:

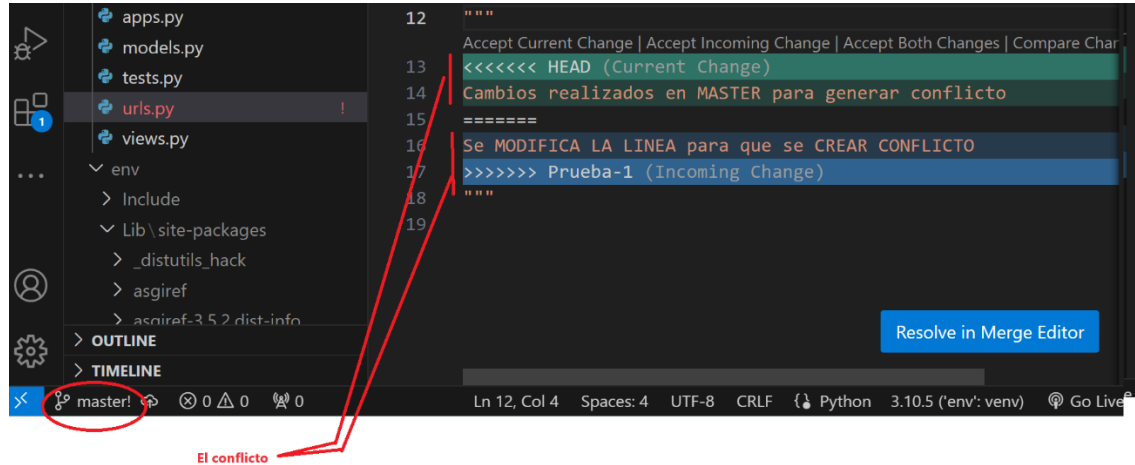


Figura 22. El editor muestra el conflicto

Al dar clic en el botón “Resolve in Merge Editor” se muestran los dos archivos con las diferencias:

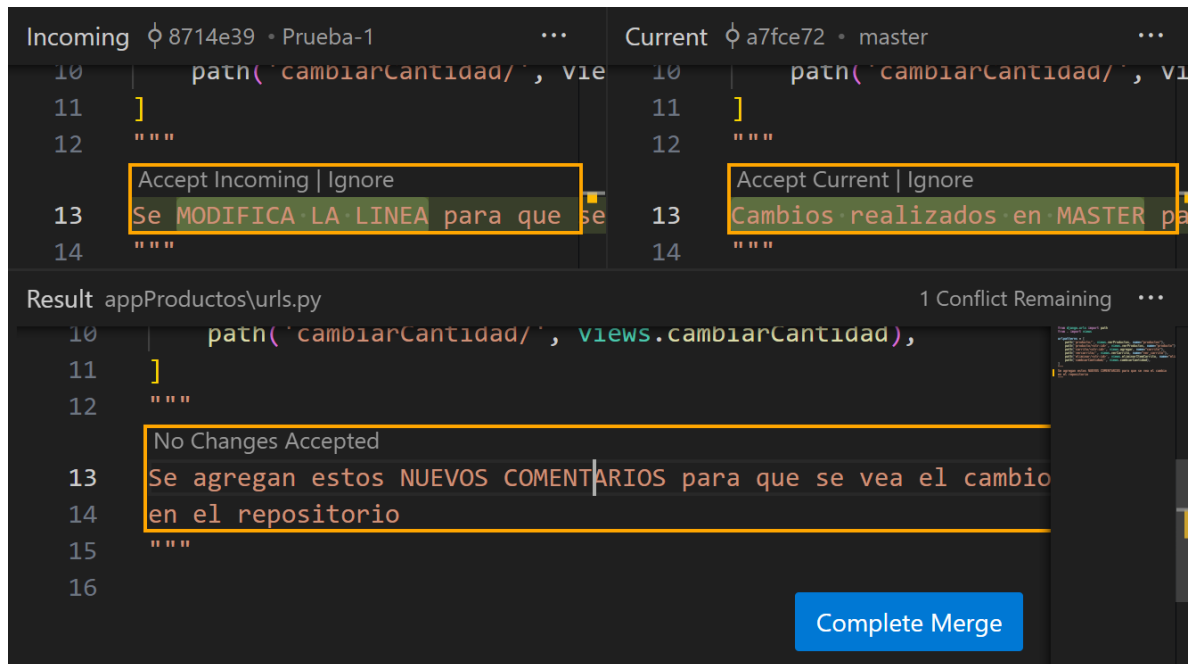


Figura 23. Resolviendo el conflicto

Se corrige manualmente y se completa la mezcla. Resta realizar un nuevo “commit” para asegurar los cambios.

## REPOSITORIO REMOTO


Git es un **sistema de control de código distribuido**. Esto quiere decir que, aunque todos los desarrolladores tienen una copia exacta del mismo repositorio en su disco duro, existen uno o más repositorios remotos contra los que trabajamos, y que son los que almacenan el estado final del producto. Estos repositorios remotos se suelen llamar simplemente "remotos", y no todo el mundo tiene permiso para enviar *commits* hacia ellos (lo que se llama hacer un *push*).

En esta sección se tratará el uso del repositorio remoto y el flujo de trabajo sugerido para proyectos en desarrollo, mejoramiento y producción.


### CREAR UN REPOSITORIO REMOTO

En primer lugar, debe tener una cuenta en <https://github.com/>. Realice el proceso de registro, seleccione la versión según sus necesidades, confirme el correo electrónico, la autenticación y el perfil.

Ingresa a GitHub con su cuenta y realice uno de los siguientes procedimientos: crear un repositorio nuevo, clonar un repositorio remoto en su repositorio local, Importar el código, etc. Debe consultar la documentación oficial para realizar este proceso<sup>5</sup>.

Una vez ha ingresado con su usuario, clic en el botón  y diligencie los datos solicitados, como en la figura.


Required fields are marked with an asterisk (\*).


Owner *	Repository name *
 marcoleonmora	/ pruebaGit
	✓ pruebaGit is available.

Great repository names are short and memorable. Need inspiration? How about [potential-adventure](#) ?

Description (optional)

Repositorio para demostrar las funcionalidades de GITHUB, como apoyar del documento "Gui Git Git Hub"

☒  **Public**  
Anyone on the internet can see this repository. You choose who can commit.

☐  **Private**  
You choose who can see and commit to this repository.

Initialize this repository with:

☒ **Add a README file**  
This is where you can write a long description for your project. [Learn more about READMEs.](#)

Add .gitignore


.gitignore template: **None**


Choose which files not to track from a list of templates. [Learn more about ignoring files.](#)

Choose a license

License: **None**

A license tells others what they can and can't do with your code. [Learn more about licenses.](#)

This will set  **main** as the default branch. Change the default name in your [settings](#).

 You are creating a public repository in your personal account.

[Create repository](#)

Figura 24. Creando un repositorio remoto

En el caso de ejemplo que hemos venido utilizando, se pueden subir los archivos seleccionando la opción "Add file -> Upload files".

<sup>5</sup> <https://docs.github.com/es/get-started>

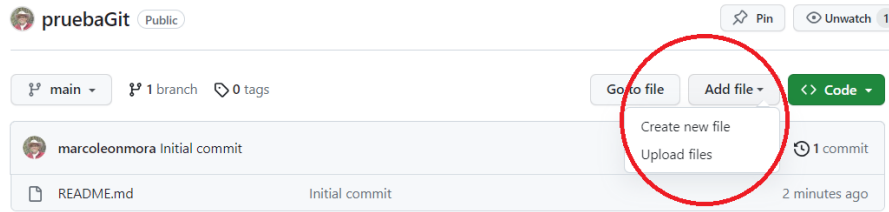


Figura 25. Subiendo Archivos al repositorio remoto

Otra opción es pulsar el botón “Code” y copiar la URL.

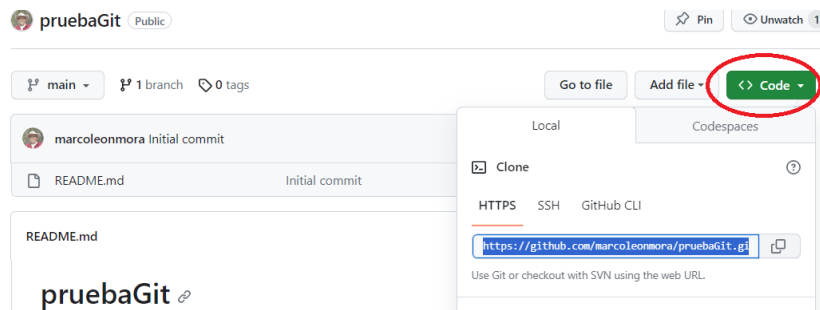


Figura 26. LA URL del proyecto

Desde la consola, usar el siguiente comando, agregando al URL copiada:

```
$ git remote add origin https://github.com/marcoleonmora/pruebaGit.git
```

```
$ git remote -v
```

Para verificar el repositorio remoto:

```
mlmor@MarcoLeon MINGW64 ~/OneDrive/Documents/SENA/SENA_Ibague/ADSI_2022/
FORMACION/Complementarias/2591860_AdrianaRincon/webDjango (master)
$ git remote -v
origin https://github.com/marcoleonmora/pruebaGit.git (fetch)
origin https://github.com/marcoleonmora/pruebaGit.git (push)
```

Figura 27. Verificando el repositorio remoto

## SUBIR CAMBIOS AL REMOTO

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

Para subir el contenido del repositorio local al remoto.



```

mImor@MarcoLeon MINGW64 ~/OneDrive/Documents/SENA/SENA_Ibague/ADSI 2022/
FORMACION/Complementarias/2591860_AdrianaRincon/webDjango (master)
$ git push origin master
Enumerating objects: 121, done.
Counting objects: 100% (121/121), done.
Delta compression using up to 8 threads
Compressing objects: 100% (119/119), done.
Writing objects: 100% (121/121), 435.63 KiB | 11.77 MiB/s, done.
Total 121 (delta 35), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (35/35), done.
remote:
remote: Create a pull request for 'master' on GitHub by visiting:
remote:   https://github.com/marcoleonmora/pruebaGit/pull/new/master
remote:
To https://github.com/marcoleonmora/pruebaGit.git
 * [new branch]      master -> master

```

Figura 28. Subiendo contenido al remoto

De nuevo en el sitio web de GitHub, por defecto el repositorio estará ubicado en la rama “main”, si cambia a la rama “master” se verán las carpetas y archivos del proyecto.

The screenshot shows the GitHub interface for the repository 'pruebaGit'. At the top, it indicates the repository is public. A yellow banner states 'master had recent pushes 2 minutes ago' with a 'Compare & pull request' button. Below this, it shows '2 branches' and '0 tags'. A status bar indicates 'This branch is 8 commits ahead, 1 commit behind main.' with a 'Contribute' button. The commit history table is as follows:

Commit	Author	Message	Time
c3d5dac	marcoleonmora	Conflicto resuelto	13 minutes ago
		accounts	Commit inicial
		appGerente	Commit inicial
		appProductos	Conflicto resuelto
		templates	Commit inicial
		webDjango	Commit inicial
		.gitignore	Commit inicial

Figura 29. Repositorio remoto con copia del local

## ACTUALIZAR REPOSITORIO LOCAL DESDE REMOTO

Para realizar el proceso inverso y actualizar el repositorio local desde el remoto, necesidad que se presenta en el caso de trabajar con otros desarrolladores, se utiliza el comando “git pull”, que extrae y descarga contenido desde un repositorio remoto y actualizar al instante el repositorio local para reflejar ese contenido.

`$ git pull`

Para bajar el contenido del repositorio remoto al local.

```

m1mor@MarcoLeon MINGW64 ~/OneDrive/Documents/SENA/SENA_Ibague/ADSI 2022
FORMACION/Complementarias/2591860_AdrianaRincon/webDjango (master)
$ git pull
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 699 bytes | 63.00 KiB/s, done.
From https://github.com/marcoleonmora/pruebaGit
 * [new branch]      main      -> origin/main
Already up to date.

```

Figura 30. Bajando contenido del remoto

## CLONAR UN REPOSITORIO REMOTO

Cuando se tiene un repositorio remoto, se puede clonar en su repositorio local con el siguiente comando:

```
$ git clone https://github.com/marcoleonmora/pruebaGit.git
```

## SUGERENCIA DEL FLUJO DE TRABAJO Y EL USO DE RAMAS<sup>6</sup>

Para aumentar la experiencia positiva al utilizar Git, se recomienda una serie de procedimientos que debe seguir cada miembro del equipo, de manera coordinada, para lograr un proceso de desarrollo de software bien administrado.

Se utiliza una arquitectura centralizada, con un repositorio remoto (origin) y una cantidad de repositorios locales (según los desarrolladores del equipo). Cada desarrollador realizara un “push” o un “pop” para subir sus cambios o actualizar su repositorio local con los cambios realizados por otros miembros del equipo.

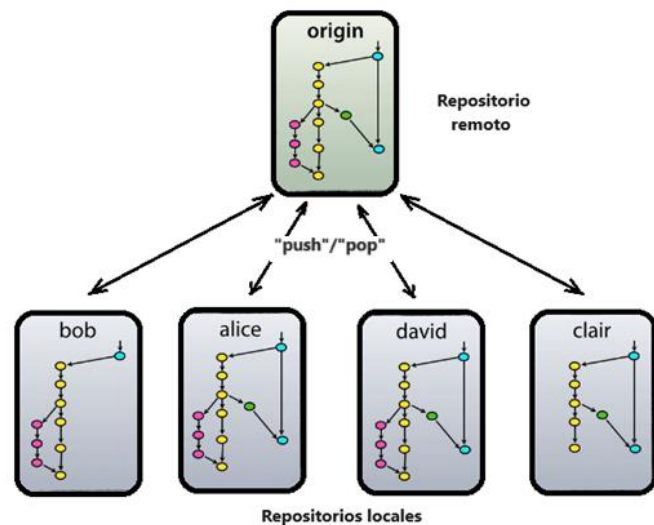


Figura 31. Arquitectura típica con Git Hub

<sup>6</sup> Adaptado de <https://nvie.com/posts/a-successful-git-branching-model/>

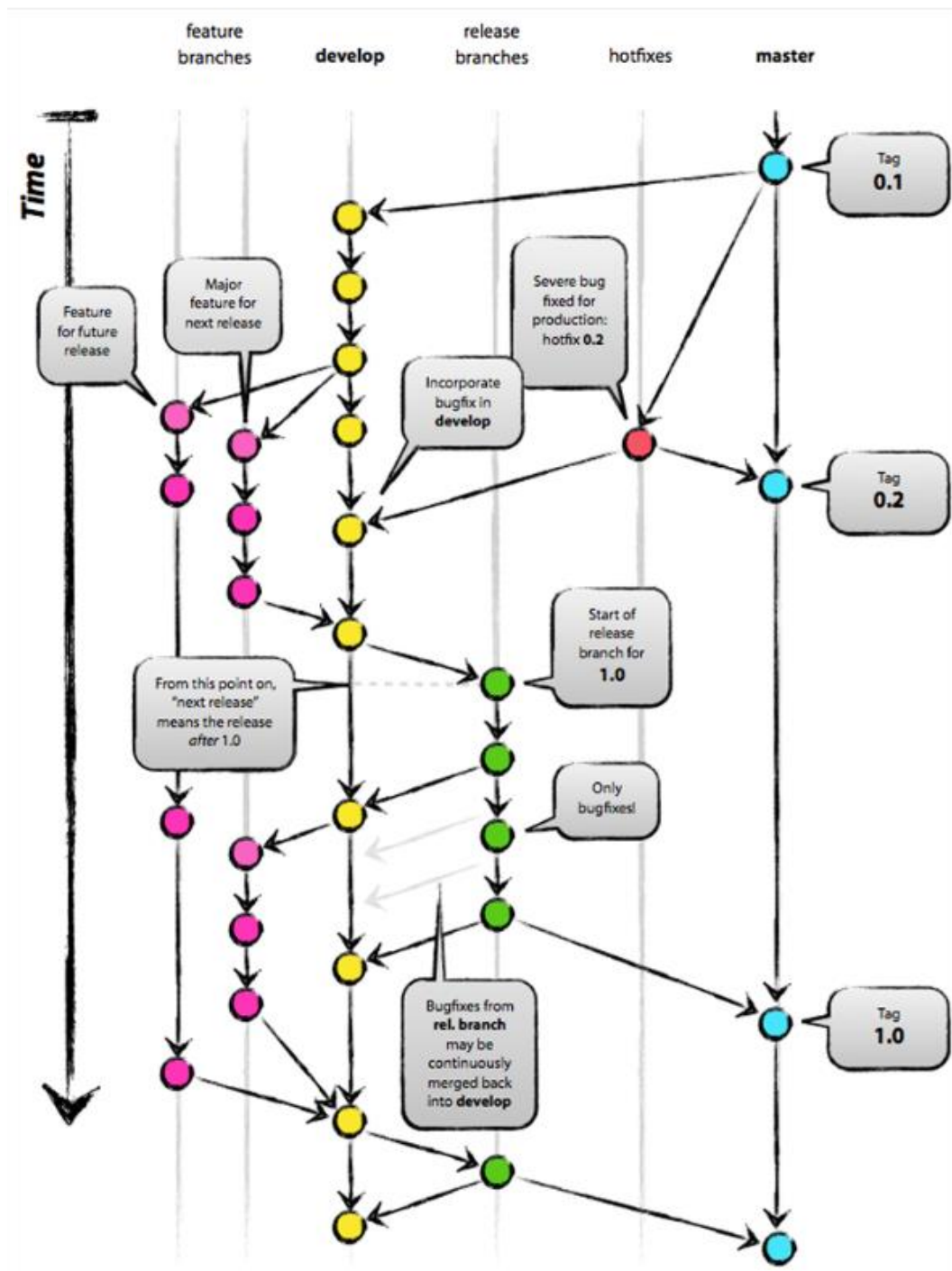


Figura 32. Flujo de trabajo sugerido

La figura anterior muestra la estructura típica, según el flujo de trabajo sugerido y que se explicara con mayor detalle en los siguientes apartados.

---

## LAS RAMAS PRINCIPALES

El repositorio central tendrá dos ramas principales que perduran en el tiempo: **Master** y **develop**.

La rama **origin/master** (La rama 'master' en el repositorio 'origin') que es la rama principal donde el HEAD del código fuente siempre refleja un estado *listo para producción*.

**origin/develop** es la rama principal donde el HEAD del código fuente siempre refleja un estado con los últimos cambios de desarrollo listos para la próxima versión.

Cuando el código fuente en la rama **develop** alcanza un punto estable y está listo para ser publicado, todos los cambios deben fusionarse nuevamente en **master** y luego etiquetarse con un número de versión. Por lo tanto, cada vez que los cambios se vuelven a fusionar en **master**, se trata de una nueva versión de producción.

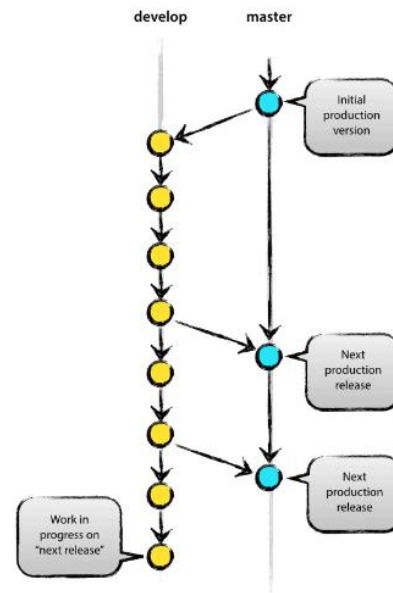


Figura 33. Ramas principales

---

## LAS RAMAS DE APOYO

Además de las ramas principales **master** y **develop**, el modelo de desarrollo utiliza una variedad de ramas de soporte para ayudar al desarrollo paralelo entre los miembros del equipo, facilitar el seguimiento de las funciones, prepararse para los lanzamientos de producción y ayudar a solucionar rápidamente los problemas de producción. A diferencia de las ramas principales, estas ramas siempre tienen un tiempo de vida limitado, ya que eventualmente serán eliminadas.

Los diferentes tipos de ramas auxiliares que podremos utilizar son:

- Ramas de características o funcionalidades
- Ramas de lanzamiento
- Ramas de revisión

Desde el punto de vista técnico, estas ramas no son en absoluto "especiales". Los tipos de ramas de apoyo se clasifican según cómo las *usamos*.

---

## RAMAS DE CARACTERÍSTICAS

Se utilizan para desarrollar una nueva característica del software. Puede ramificarse desde la rama "develop" y deben fusionarse nuevamente con "develop".

Se sugiere nombrar estas ramas con algún nombre que refleje la característica, exceptuando "master", "develop", "release-\*" o "hotfix-\*".

Las ramas de características (o a veces llamadas ramas temáticas) se utilizan para desarrollar nuevas características para el próximo lanzamiento u otro en el futuro. Al comenzar el desarrollo de una característica, la versión de destino en la que será incorporada esta característica bien puede ser desconocido en ese punto. La esencia de una rama característica es que existe mientras la característica esté en desarrollo, pero eventualmente lo hará fusionarse de nuevo en `develop` (para agregar definitivamente la nueva característica al próximo lanzamiento) o descartado (en caso de un experimento decepcionante).

Las ramas de características generalmente existen solo en los repositorios de los desarrolladores, no en `origin`.

1. Crear la rama desde la rama “`develop`”:

```
$ git checkout -b nombre-nueva-rama
```

2. Incorporar la característica ya terminada a la rama de desarrollo:

```
$ git checkout develop
```

cambia a la rama de desarrollo.

```
$ git merge --no-ff nombre-nueva-rama
```

actualiza la rama de desarrollo desde la rama de característica. La bandera “`--no-ff`” hace que la fusión siempre cree un nuevo commit.

```
$ git branch -d nombre-nueva-rama
```

elimina la rama de característica

```
$ git push origin develop
```

Actualiza el repositorio remoto

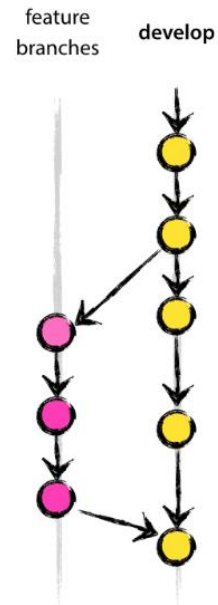


Figura 34. Rama de característica

## RAMA DE LANZAMIENTO

Las ramas de lanzamiento se crean a partir de la rama `develop` y deben fusionarse nuevamente con `develop` y `master`.

Por ejemplo, si la versión 1.1.5 es la versión de producción actual y se tiene una nueva versión ya casi lista. El estado de `develop` está listo para el “next release” y se decidió que se convertiría en la versión 1.2 (en lugar de 1.1.6 o 2.0). Entonces se crea una rama de lanzamiento con un nombre que refleje el nuevo número de versión:

```
$ git checkout -b release-1.2 develop
```

Se crea la nueva rama desde `develop` y el flujo de trabajo cambia a ella.

```
$ git commit -a -m "Número de versión cambia a 1.2"
```

Se realiza el commit para aceptar los cambios realizados

Esta nueva rama existirá hasta cuando se decida realizar el despliegue de la nueva versión. Primero se debe fusionar con `master` y después se realiza la fusión con `develop`, para que los futuros lanzamientos contengan también

```
$ git checkout master
```

Se cambia a la rama master

```
$ git merge --no-ff release-1.2
```

Se fusiona con master

```
$ git tag -a 1.2
```

Se inserta una etiqueta en la rama master

```
$ git checkout develop
```

Se cambia a la rama develop

```
$ git merge --no-ff release-1.2
```

Se fusiona con develop

```
$ git branch -d release-1.2
```

Se elimina la rama release-1.2

## RAMAS DE CORRECCIÓN DE ERRORES (HOTFIX)

Las ramas de corrección de errores se crean a partir de la rama master y deben fusionarse nuevamente con develop y master. La convención es nombrarlas "hotfix-\*".

Las ramas Hotfix son muy parecidas a las ramas de lanzamiento, ya que también se prepara para un nuevo lanzamiento de producción, aunque no planificado. Surgen de la necesidad de actuar inmediatamente sobre un estado no deseado de la versión actual de producción. Cuando se debe resolver un error crítico en una versión de producción, de manera inmediata, se puede ramificar una rama de revisión y etiquetar la rama maestra para marcar la nueva versión de producción. También hay que actualizar la rama de desarrollo para corregir el error en los nuevos lanzamientos.

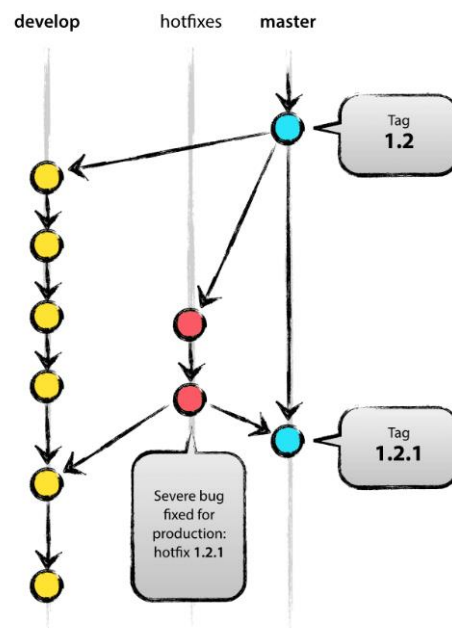


Figura 35. Rama de corrección de errores

```
$ git checkout -b hotfix-1.2.1 master
```

Se crea la rama hotfix-1.2.1 desde la rama master.

```
$ git commit -a -m "Número de versión cambia a 1.2.1 por corrección de error"
```

Se realiza el commit para aceptar los cambios realizados. Se podrían realizar varios commit durante el proceso de corrección del error.

```
$ git checkout master
```

Se cambia a la rama master.

```
$ git merge --no-ff hotfix-1.2.1
```

Se fusiona con master

```
$ git tag -a 1.2.1
```

Se inserta una etiqueta de versión en la rama master

```
$ git checkout develop
```

Se cambia a la rama develop

```
$ git merge --no-ff hotfix-1.2.1
```

Se fusiona con develop

**Atención:** Si al momento de corregir un error, se tiene activa una rama de lanzamiento, la rama hotfix-\* se debe fusionar con esta y no con develop.

```
$ git branch -d hotfix-1.2.1
```

Se elimina la rama hotfix-1.2.1

## SUGERENCIA FINAL

El presente documento ha presentado unas indicaciones muy resumidas de los comandos a emplear en Git y GitHub. Quedan temas muy importantes, pero más avanzados, sobre todo en lo que atañe a equipos de desarrollo.

Se recomienda consultar en la documentación oficial, los diversos argumentos que acompañan a los comandos a utilizar, que pueden hacer variar significativamente el funcionamiento de los mismos.

## ANEXO: RESUMEN DE COMANDOS<sup>7</sup>

### INSTALAR GIT

GitHub proporciona clientes de escritorio que incluyen una interfaz gráfica de usuario para las acciones más comunes que se pueden realizar en un repositorio y una edición de Git en la línea de comandos actualizada automáticamente para escenarios más avanzados.

---

#### GITHUB PARA WINDOWS

[windows.github.com](https://windows.github.com)

---

#### GITHUB PARA MAC

[mac.github.com](https://mac.github.com)

Distribuciones de Git para sistemas Linux y POSIX se encuentran disponibles en el sitio web oficial Git SCM.

---

#### GIT PARA TODAS LAS PLATAFORMAS

[git-scm.com](https://git-scm.com)

### CONFIGURAR HERRAMIENTAS

Configura la información del usuario para todos los repositorios locales

\$ git config --global user.name "[name]"	Establece el nombre que estará asociado a tus commits
\$ git config --global user.email "[email address]"	Establece el e-mail que estará asociado a sus commits

### CREAR REPOSITORIOS

Inicializa un nuevo repositorio u obtiene uno de una URL existente

\$ git init [project-name]	Crea un nuevo repositorio local con el nombre especificado
\$ git clone [url]	Descarga un proyecto y todo su historial de versiones

### EFFECTUAR CAMBIOS

Revisa cambios y crea un commit

---

<sup>7</sup> Tomado de [https://training.github.com/downloads/es\\_ES/github-git-cheat-sheet/](https://training.github.com/downloads/es_ES/github-git-cheat-sheet/)



\$ git status	Enumera todos los archivos nuevos o modificados de los cuales se van a guardar cambios
\$ git diff	Muestra las diferencias entre archivos que no se han enviado aún al área de espera
\$ git add [file]	Guarda el estado del archivo en preparación para realizar un commit
\$ git diff --staged	Muestra las diferencias del archivo entre el área de espera y la última versión del archivo
\$ git reset [file]	Mueve el archivo del área de espera, pero preserva su contenido
\$ git commit -m "[descriptive message]"	Registra los cambios del archivo permanentemente en el historial de versiones

## CAMBIOS GRUPALES

Nombra una serie de commits y combina esfuerzos ya completados

\$ git Branch	Enumera todas las ramas en el repositorio actual
\$ git branch [branch-name]	Crea una nueva rama
\$ git switch -c [branch-name]	Cambia a la rama especificada y actualiza el directorio activo
\$ git merge [branch-name]	Combina el historial de la rama especificada con la rama actual
\$ git branch -d [branch-name]	Borra la rama especificada

## REFACTORIZACIÓN DE ARCHIVOS

Reubica y retira los archivos de los cuales se tiene una versión

\$ git rm [file]	Borra el archivo del directorio activo y lo pone en el área de espera en un estado de eliminación
\$ git rm --cached [file]	Retira el archivo del historial de control de versiones, pero preserva el archivo a nivel local
\$ git mv [file-original] [file-renamed]	Cambia el nombre del archivo y lo prepara para ser guardado

## SUPRIMIR EL SEGUIMIENTO DE CAMBIOS

Excluye los archivos temporales y las rutas

\*.log

build/  
temp-\*

Un archivo de texto llamado '.gitignore' suprime la creación accidental de versiones para archivos y rutas que concuerdan con los patrones especificados

\$ git ls-files --others --ignored --exclude-standard

Enumera todos los archivos ignorados en este proyecto

## GUARDAR FRAGMENTOS

Almacena y restaura cambios incompletos

\$ git stash

Almacena temporalmente todos los archivos modificados de los cuales se tiene al menos una versión guardada

\$ git stash pop

Restaura los archivos guardados más recientemente

\$ git stash list

Enumera todos los grupos de cambios que estan guardados temporalmente

\$ git stash drop

Elimina el grupo de cambios más reciente que se encuentra guardado temporalmente

## REPASAR HISTORIAL

Navega e inspecciona la evolución de los archivos de proyecto

\$ git log

Enumera el historial de versiones para la rama actual

\$ git log --follow [file]

Enumera el historial de versiones para el archivo, incluidos los cambios de nombre

\$ git diff [first-branch]...[second-branch]

Muestra las diferencias de contenido entre dos ramas

\$ git show [commit]

Produce metadatos y cambios de contenido del commit especificado

## REHACER COMMITS

Borra errores y elabora un historial de reemplazo

\$ git reset [commit]

Deshace todos los commits después de [commit], preservando los cambios localmente

\$ git reset --hard [commit]

Desecha todo el historial y regresa al commit especificado

## SINCRONIZAR CAMBIOS

Registrar un marcador para un repositorio e intercambiar historial de versiones

\$ git fetch [bookmark]	Descarga todo el historial del marcador del repositorio
\$ git merge [bookmark]/[branch]	Combina la rama del marcador con la rama local actual
\$ git push [alias] [branch]	Sube todos los commits de la rama local a GitHub
\$ git pull	Descarga el historial del marcador e incorpora cambios

## ENLACES

<https://www.opentix.es/sistema-de-control-de-versiones/>

<https://www.atlassian.com/es/git/tutorials/what-is-version-control>

<https://www.atlassian.com/es/git/tutorials/what-is-git>

<https://www.atlassian.com/es/git/tutorials/install-git>

<https://nvie.com/posts/a-successful-git-branching-model/>

<https://cvs.nongnu.org/>

<https://subversion.apache.org/>

<https://www.mercurial-scm.org/>

<https://git-scm.com/downloads>

<https://gitforwindows.org/>

<https://nvie.com/posts/a-successful-git-branching-model/>

<https://ndpsoftware.com/git-cheatsheet.html#loc=index;>

<https://www.atlassian.com/git/tutorials/saving-changes/gitignore#:~:text=gitignore%20files%20contain%20patterns%20that,gitignore%20files%20in%20your%20repository>

[https://training.github.com/downloads/es\\_ES/github-git-cheat-sheet/](https://training.github.com/downloads/es_ES/github-git-cheat-sheet/)

CONTROL DE CAMBIOS

Nombre	Centro	Fecha	Razón del Cambio