

Git



Git es uno de los sistemas de control de versiones más utilizado en el mundo del desarrollo. Es un proyecto de código abierto que fue desarrollado por Linus Torvalds en 2005 y hoy día cuenta con un mantenimiento activo por toda la comunidad.

Git, al trabajar con versiones nos permitirá ver que cambios se han hecho en cada una de ellas y retroceder si fuera necesario revertir algún cambio. De esta manera, se nos generará un historial de todos los cambios que se han realizado en el proyecto y quien lo ha hecho.

¿Por qué utilizar GIT?

1. **Control de versiones**: GIT permite mantener un registro completo de todas las versiones de un proyecto, lo que permite a los desarrolladores trabajar de manera más eficiente, realizar cambios sin temor a perder el trabajo anterior y recuperar versiones anteriores del código en caso de ser necesario.
2. **Colaboración**: GIT permite que varios desarrolladores trabajen en un mismo proyecto al mismo tiempo, lo que permite a los equipos de trabajo coordinar mejor y trabajar de manera más eficiente en conjunto. Además, GIT permite la integración de los cambios realizados por diferentes desarrolladores, lo que reduce el riesgo de conflictos y errores.
3. **Ramificación**: GIT permite crear ramificaciones o "branches" de un proyecto, lo que permite a los desarrolladores experimentar con nuevas ideas sin afectar la versión principal del proyecto. Esto es especialmente útil en proyectos grandes y complejos.
4. **Seguridad**: GIT proporciona una serie de herramientas de seguridad, como la autenticación y el cifrado de datos, lo que garantiza la integridad del código y protege la propiedad intelectual del proyecto.

Cuando hablemos de GIT hay terminología fundamental que debemos de interpretar con facilidad.

- **REPOSITORIO**: es el espacio que contiene al proyecto y que almacena dentro todo el historial de cambios. Es el componente central de GIT y puede ser local o remoto. Un repositorio nuevo se inicia con el comando `git init`
- **STAGING AREA**: es un espacio intermedio entre el directorio de trabajo y el repositorio. Es el lugar donde se preparan los cambios antes de realizar un commit en el repositorio. Para añadir archivos a esta área lo realizamos con el siguiente comando `git add` . (para



- enviar todos los archivos) o `git add "nombrearchivo.extension"` (para enviar un archivo específico).
- **COMMIT:** son cada uno de los cambios registrados en el historial de git. Cada desarrollador "manda commits" de los cambios realizados. Cada commit está acompañado de una descripción. Para ejecutar un commit deben existir archivos en el staging área y se realiza a través de la siguiente línea `git commit -m "Mensaje descriptivo"`
 - **RAMAS o BRANCH:** son bifurcaciones que toma el proyecto. En git todo se trabaja con ramas, la principal se denomina "master/main/origin" donde está la versión que se encuentra en producción. La línea de comando `git branch "nueva-rama"` nos permitirá crear una nueva bifurcación del proyecto.
 - **MERGE:** se denomina así a la fusión de una rama (Branch) con otra rama. Para poder llevar adelante un merge, debemos situarnos en la rama a la que queremos fusionarle los cambios y luego ejecutar la siguiente línea de comando indicando la rama que vamos a fusionar: `git merge "nueva-rama"`
 - **CLON:** copia exacta del repositorio.
 - **FORK:** clon de un repositorio que toma su propio camino.

¿Por qué utilizar GIT para el desarrollo colaborativo?

En primer lugar, es un sistema **distribuido**. Cada desarrollador tiene una copia del proyecto en su sistema local. Lo que significa que no tienen que conectarse a un servidor ni conectarse a internet para continuar con el trabajo en cualquier momento.

Otro detalle importante, son las **ramas y las fusiones** que se generan para no comprometer a la rama principal, para tener integridad en la rama principal. De esta manera, si se necesitan realizar pruebas, agregar funcionalidades o simplemente arreglar un error se puede hacer en una rama secundaria y luego combinarla con la principal.

Git Bash

Si bien en nuestra cátedra nos enfocaremos al uso de GitHub Desktop para uso del trabajo colaborativo, es necesario interpretar los comandos y conocer el uso de Git Bash.

En primer lugar debemos descargar el software del siguiente [enlace](#). Una vez dentro del sitio web presionar el botón “Download for Windows”.



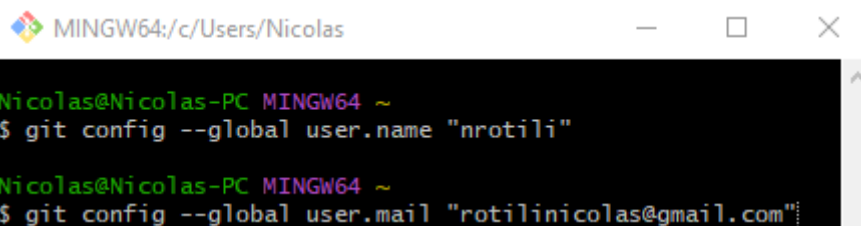
A continuación, procedemos con la instalación. El proceso es sumamente sencillo para el uso que nosotros le daremos por lo cual dejaremos la configuración que viene por defecto en todas las páginas y daremos “next” o “siguiente” hasta finalizar el proceso.

Culminada la instalación debemos configurar nuestra identidad de Git, se recomienda utilizar mismo usuario e email que poseen en GitHub (tema que trataremos más adelante) para evitar futuros conflictos, por lo cual si no tiene cuenta se recomienda generar una a través de [su sitio web](#).

Iniciamos la consola de Git (Git Bash) y ejecutamos las siguientes líneas de comando reemplazando los datos entre comillas por sus valores.

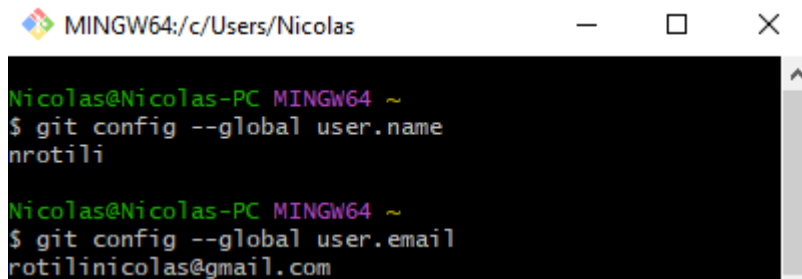
```
git config --global user.name "nrotili"
```

```
git config --global user.email "rotilinicol@gmail.com"
```



De esta manera, cada vez que se inicie un nuevo proyecto en git, tendrá sus datos como creador del mismo. Para validar si los datos fueron tomados, podemos escribir las siguiente líneas:

```
git config --global user.name  
git config --global user.email
```



```
MINGW64:/c/Users/Nicolas  
Nicolas@Nicolas-PC MINGW64 ~  
$ git config --global user.name  
nrotili  
Nicolas@Nicolas-PC MINGW64 ~  
$ git config --global user.email  
nrotilinicolas@gmail.com
```

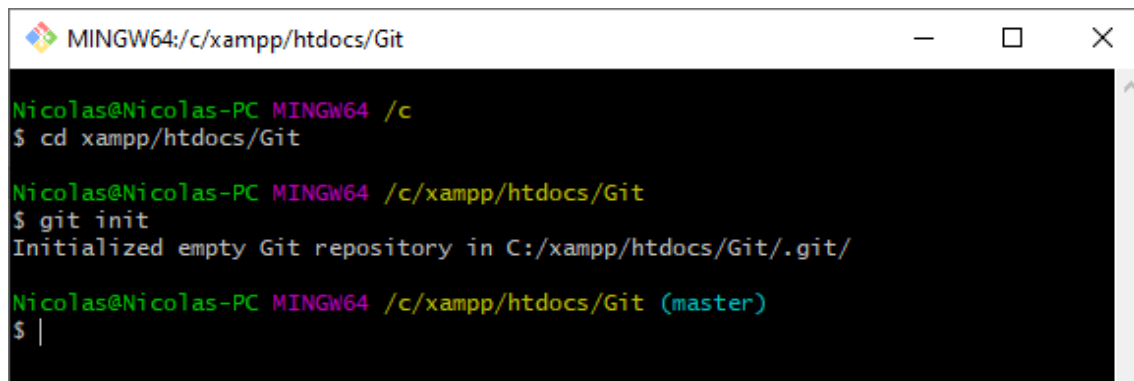
Git también nos da la oportunidad de crear configuraciones dentro de un repositorio específico diferente a las configuraciones globales. Para ello solo debemos eliminar la flag --global de la línea de comando.

Ahora ya estamos listos para poder llevar adelante nuestro primer proyecto con control de versiones.

Cuando un desarrollador arranca a trabajar en un proyecto, hay dos alternativas, o crea un proyecto nuevo o clona uno ya existente. Para ello existen dos líneas de comandos.

git init: se utiliza para crear un repositorio.

git clone: se utiliza para clonar un repositorio ya iniciado, comando que utilizaremos en próximas clases.



```
MINGW64:/c/xampp/htdocs/Git  
Nicolas@Nicolas-PC MINGW64 /c  
$ cd xampp/htdocs/Git  
Nicolas@Nicolas-PC MINGW64 /c/xampp/htdocs/Git  
$ git init  
Initialized empty Git repository in C:/xampp/htdocs/Git/.git/  
Nicolas@Nicolas-PC MINGW64 /c/xampp/htdocs/Git (master)  
$ |
```

En la figura anterior, estoy creando un repositorio nuevo. En primer lugar me posiciono en la carpeta que voy a llevar adelante el desarrollo (C:\xampp\htdocs\Git) y luego ejecuto el comando *git init*.

Si deseo clonar un proyecto ya existente me posiciono en la carpeta que contendrá el proyecto y ejecuto *git clone* seguido de la URL del repositorio Git que deseamos clonar.

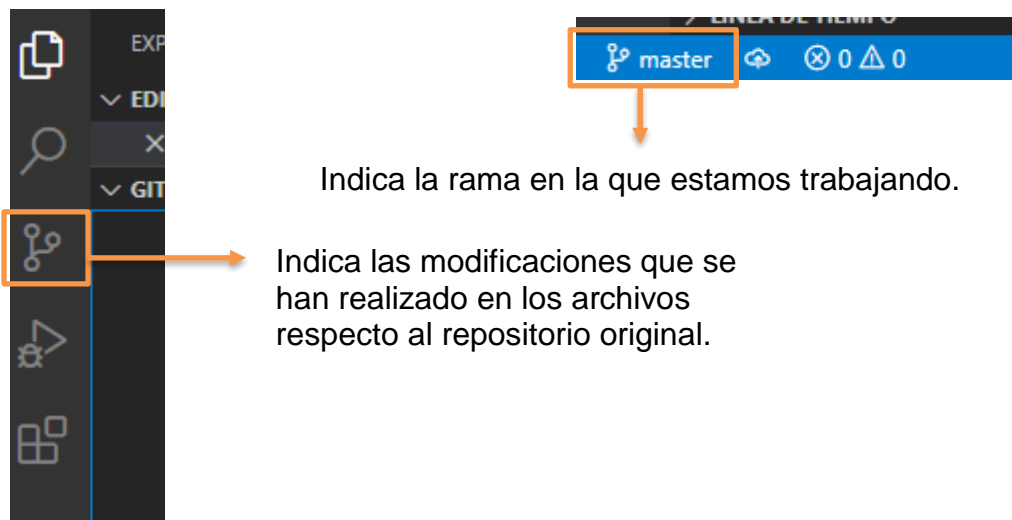
```
MINGW64:/c:/xampp/htdocs/bot-whatsapp
Nicolas@Nicolas-PC MINGW64 /c:/xampp/htdocs
$ git clone https://github.com/leifermendez/bot-whatsapp
Cloning into 'bot-whatsapp'...
remote: Enumerating objects: 480, done.
remote: Counting objects: 100% (214/214), done.
remote: Compressing objects: 100% (70/70), done.
remote: Total 480 (delta 184), reused 152 (delta 144), pack-reused 266
Receiving objects: 100% (480/480), 3.62 MiB | 317.00 KiB/s, done.
Resolving deltas: 100% (281/281), done.

Nicolas@Nicolas-PC MINGW64 /c:/xampp/htdocs
$ cd bot-whatsapp/

Nicolas@Nicolas-PC MINGW64 /c:/xampp/htdocs/bot-whatsapp (main)
$
```

Luego de cualquiera de estos dos pasos, ya tendríamos nuestro repositorio git inicializado.

Agregaremos un archivo index.html al proyecto para poder ver los cambios. VSCode, detectará que es un proyecto Git y nos añadirá tanto en la barra vertical izquierda como en la barra inferior, dos nuevas opciones.



Al agregar el nuevo archivo, el editor nos indicará que hay un archivo sin seguimiento y nos lo marcará con una "U".

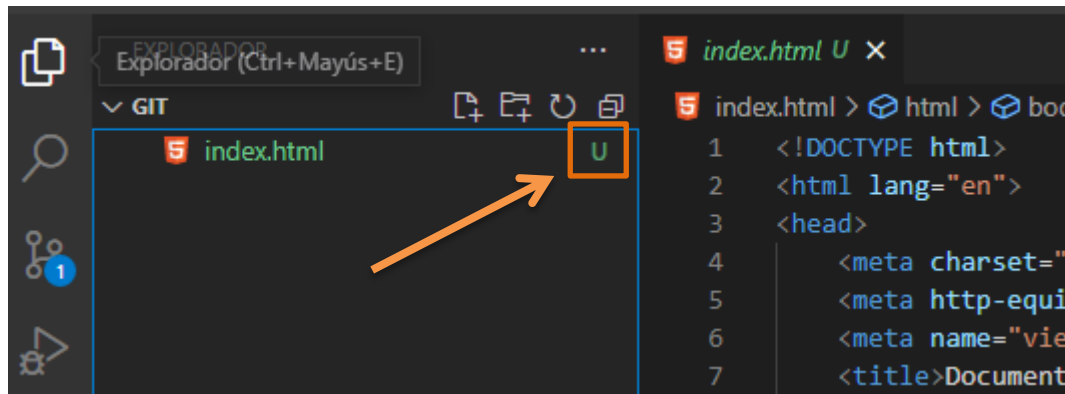


Fig. 1

Supongamos que el archivo ya está finalizado para subir por primera vez a producción. Ahora debemos enviar el archivo a la zona de preparación o “Staging Area”¹ y esto lo hacemos a través del comando `git add .` (se acompaña con el punto para enviar todos los archivos al staging, si quisiéramos enviar un solo archivo lo hacemos con la ruta en lugar del punto).

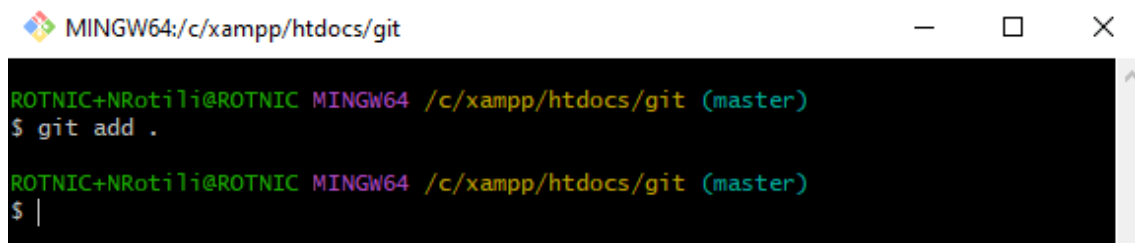


Fig. 2

Luego de ejecutar el comando podremos observar el cambio realizado en VS Code.

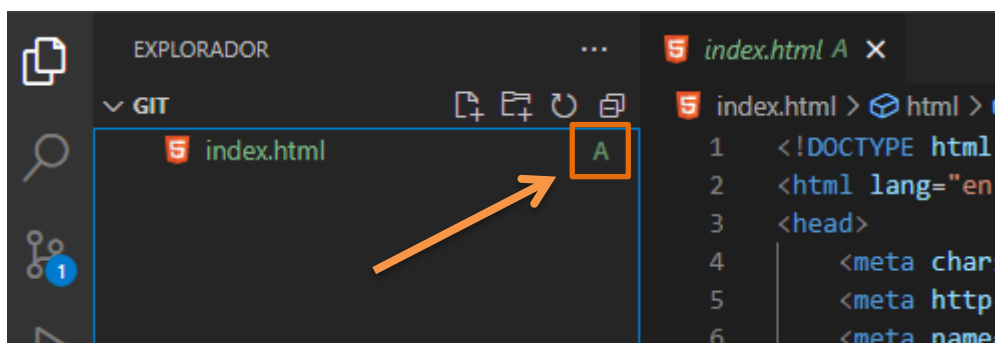
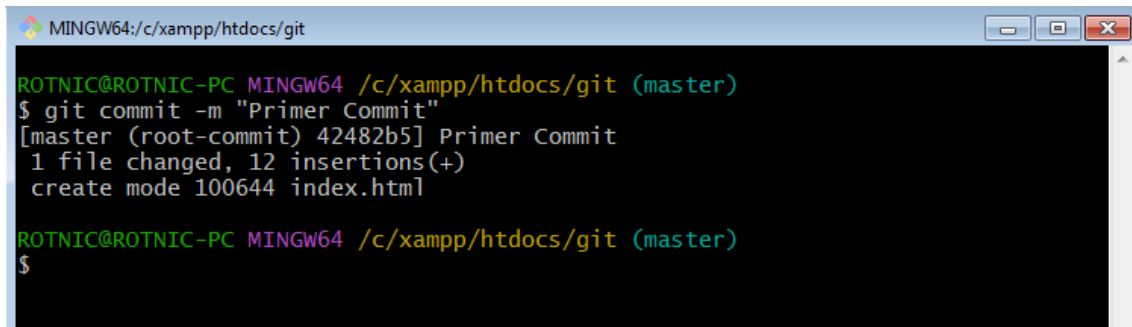


Fig. 3

Una vez que los archivos están en el staging área, se encuentran disponibles para enviar al repositorio a través de un commit.

¹ Lugar en el que se encuentran datos de un proyecto y sus cambios.



```
MINGW64:/c:/xampp/htdocs/git
ROTNIC@ROTNIC-PC MINGW64 /c:/xampp/htdocs/git (master)
$ git commit -m "Primer Commit"
[master (root-commit) 42482b5] Primer Commit
1 file changed, 12 insertions(+)
create mode 100644 index.html
ROTNIC@ROTNIC-PC MINGW64 /c:/xampp/htdocs/git (master)
$
```

Fig. 4

En esta oportunidad se utilizó el comando `git commit -m "Mensaje descriptivo"` pero se podría haber utilizado solo `git commit`, este abrirá una ventana de texto que nos pedirá el mensaje del commit y luego al guardarlo realizará la ejecución.

Si observamos Visual Studio, veremos que han desaparecido las letras que se encontraban al lado del nombre y en el apartado de Git no nos mostrará cambios pendientes.

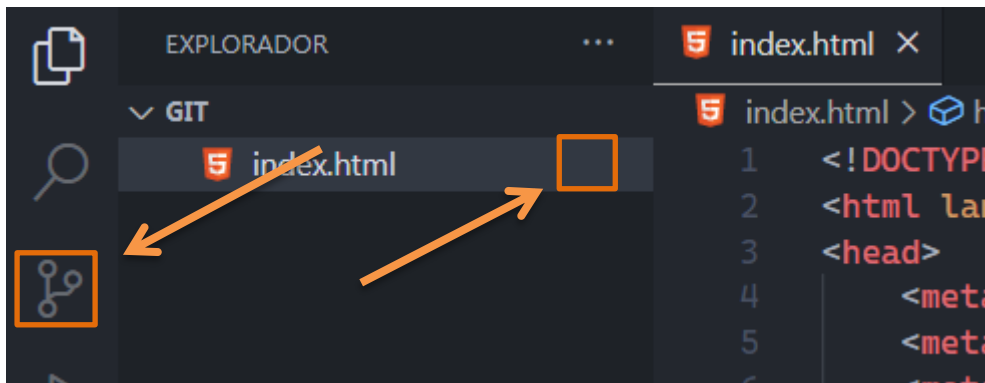
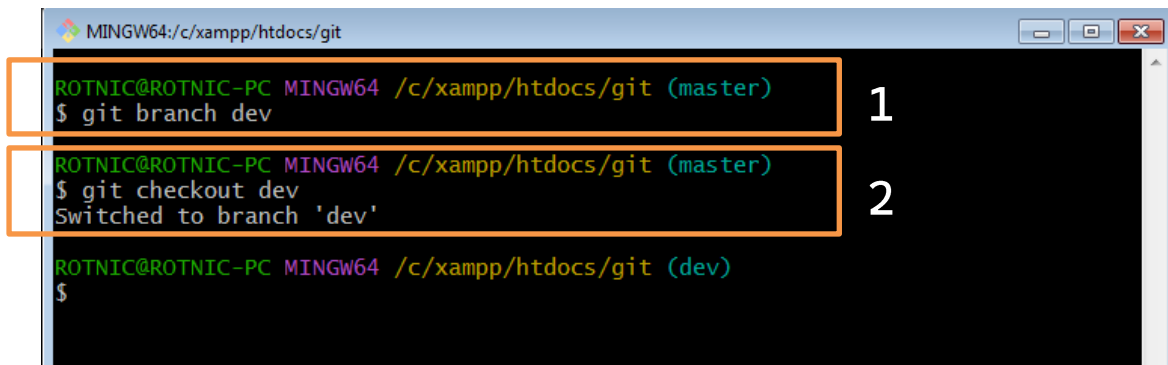


Fig. 5

De esta manera, se actualizan los cambios en el repositorio generado en un principio.

Si quisiéramos subirlo a la nube, como por ejemplo GitHub, debemos seguir los pasos que nos da la plataforma al generar un repositorio nuevo.

Siempre es recomendable pero no obligatorio trabajar en ramas que no sean la principal. Cada empresa tiene su forma de escribir la nomenclatura para sus ramas, aquí crearemos una a modo de ejemplo y le pondremos dev (en inglés, abreviatura de developer) que será nuestra branch de desarrollo. Generalmente, si se trabaja en equipo, cada desarrollador tiene su propia rama de desarrollo que depende de la rama dev.



```

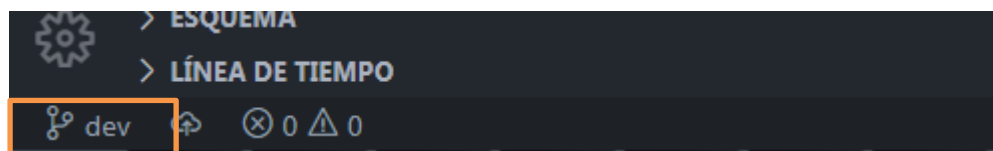
MINGW64:/c:/xampp/htdocs/git
ROTNIC@ROTNIC-PC MINGW64 /c:/xampp/htdocs/git (master)
$ git branch dev
ROTNIC@ROTNIC-PC MINGW64 /c:/xampp/htdocs/git (master)
$ git checkout dev
Switched to branch 'dev'
ROTNIC@ROTNIC-PC MINGW64 /c:/xampp/htdocs/git (dev)
$
  
```

Fig. 6

Para poder utilizar ramas, hay algunas líneas de comandos que debemos de recordar:

- *git branch nombre*: permite crear una nueva rama con el nombre que le indiquemos (recuadro 1)
- *git checkout nombre*: cambio a la rama indicada (recuadro 2)

Así como nosotros indicamos en Git que la rama en la que vamos a trabajar es dev, visual studio también realiza el cambio automáticamente y lo podremos ver en la barra inferior.

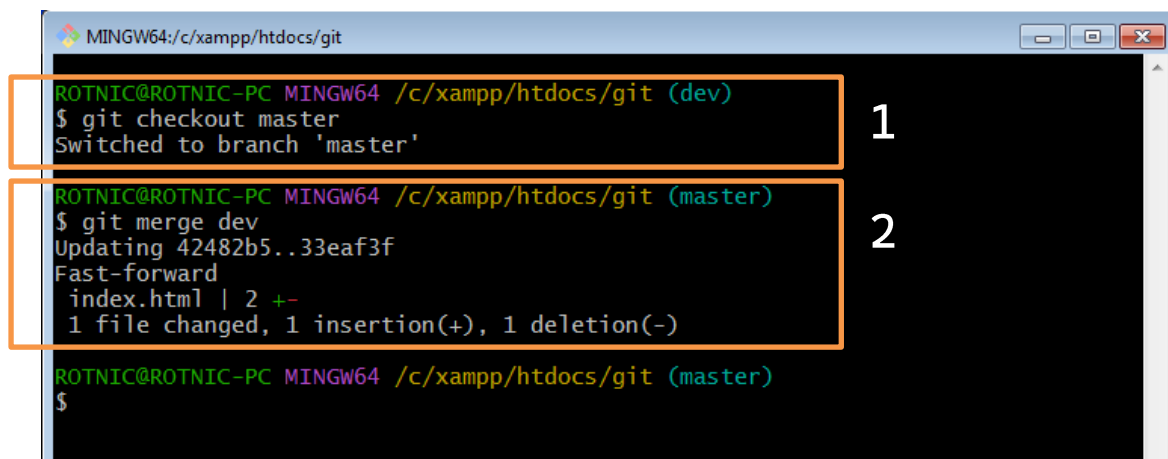


Mientras nosotros trabajemos en esta nueva rama, todos los cambios se mantendrán allí hasta que nosotros le indiquemos que se combine con la rama master.

Una vez finalizados los cambios, volvemos a repetir los pasos anteriores.

- *git add .*
- *git commit -m "descripción"*

Mientras nosotros trabajemos en esta nueva rama, todos los cambios se mantendrán allí hasta que nosotros le indiquemos que se combine con la rama master.

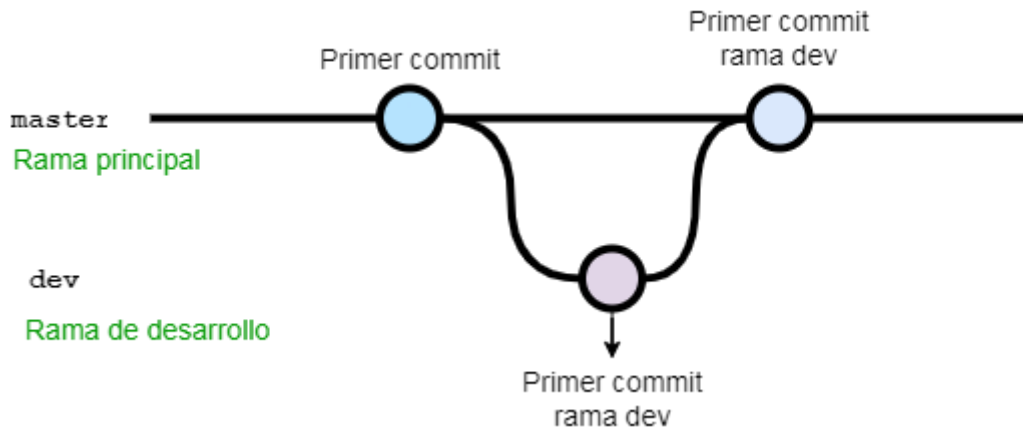


```

MINGW64:/c:/xampp/htdocs/git
ROTNIC@ROTNIC-PC MINGW64 /c:/xampp/htdocs/git (dev)
$ git checkout master
Switched to branch 'master'
ROTNIC@ROTNIC-PC MINGW64 /c:/xampp/htdocs/git (master)
$ git merge dev
Updating 42482b5..33eaf3f
Fast-forward
 index.html | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
ROTNIC@ROTNIC-PC MINGW64 /c:/xampp/htdocs/git (master)
$
  
```


Para poder llevar adelante la combinación de las ramas, debemos situarnos en la rama en la que se va a combinar la de “dev”, en este caso es “master” y eso lo hacemos como se indica en el recuadro 1. Luego usamos el comando `git merge dev` que combinará la rama dev en la branch master (recuadro 2).

Nuestro flujo de trabajo en forma gráfica quedó determinado de la siguiente manera.



Nuestro flujo de trabajo en forma gráfica quedó determinado de la siguiente manera.

Cuando trabajen con GitHub en forma colaborativa, hay que tener cuidado antes de solicitar un pull request o realizar el merge de una de nuestras ramas.

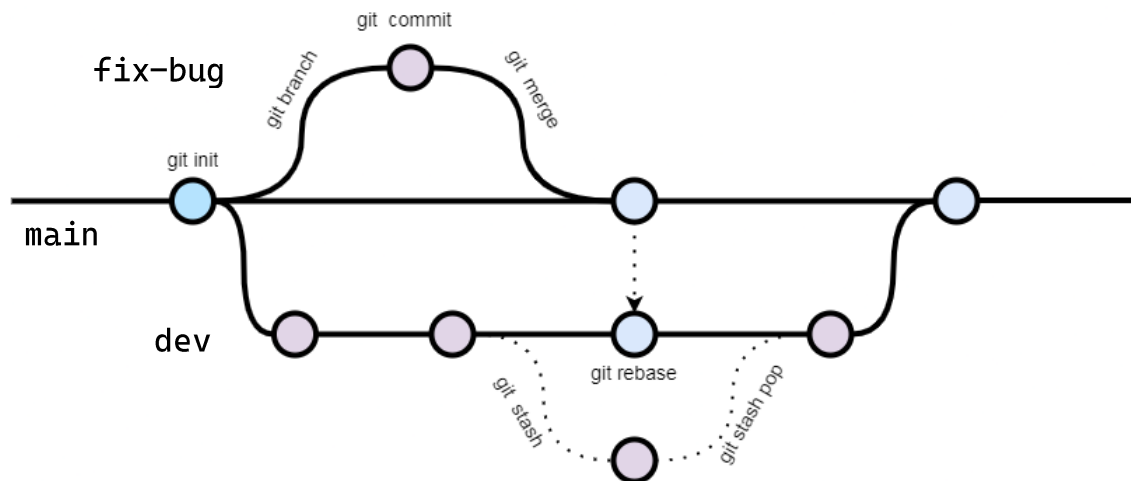


Fig. 7

Supongamos que desde la rama main creamos una nueva rama llamada “dev” en la que vamos a realizar nuestro trabajo. Sin embargo, mientras nos encontramos trabajando en esta branch, otro desarrollador se encuentra arreglando un error que se encontró en producción en la rama “fix-bug” al que luego de solucionarlo, realiza un pull request que luego es aprobado por el líder del proyecto para realizar el merge a producción.

Desde nuestra rama “dev” realizamos los commits que consideramos necesarios hasta terminar el trabajo. Antes de solicitar nuestro pull request, debemos consultar a la rama main si hubo cambios en el proyecto que no



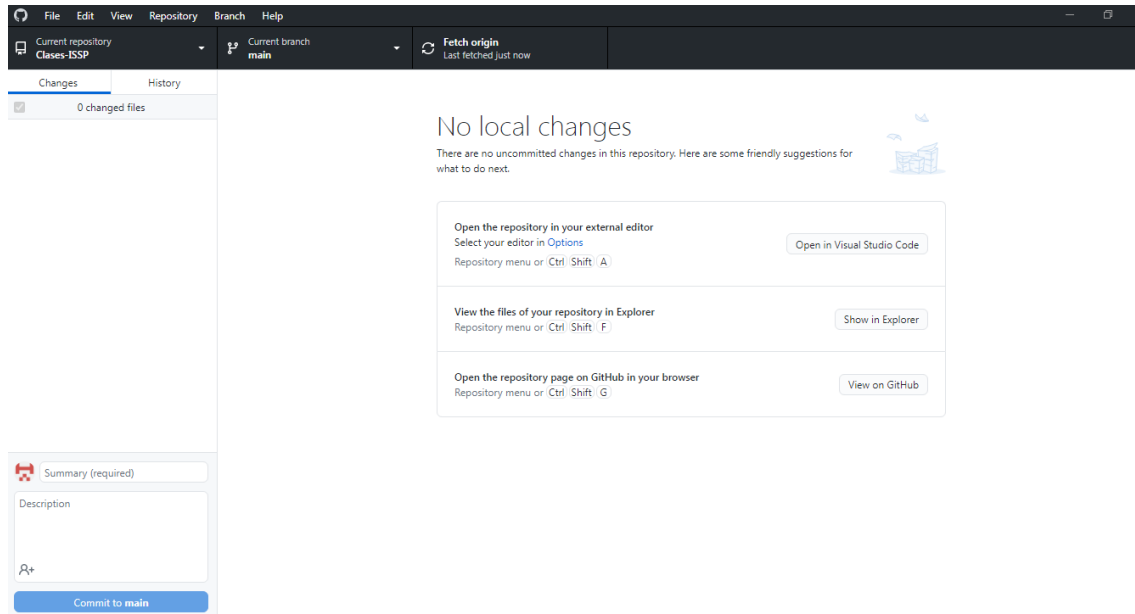
están contenidos en nuestra rama. En el caso ilustrativo de la Fig. 9, debemos integrar a “dev” los cambios que se agregaron desde “fix-bug” a “main”.

Para ello, debemos en primer lugar ejecutar el comando `git stash` (almacena temporalmente el código para trabajar en otra cosa y luego poder volver a trabajar con ellos). Luego debemos ejecutar `git rebase main` (main debe ser reemplazado por el nombre de la rama) para aplicar en nuestra rama, los nuevos cambios aplicados a la branch de producción. De esta manera, tendríamos en nuestra branch, los cambios aplicados por el desarrollador que arreglo los errores que se encontraban.

Posterior a ello, debemos integrar nuestros cambios que se encontraban en el stash y lo hacemos con la línea `git stash pop`. Finalizados estos pasos, realizamos nuestro nuevo commit y solicitamos el pull request de nuestra rama para que sea aprobado y combinado con la rama de producción.

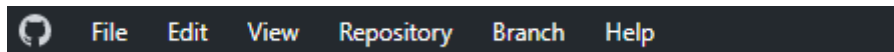
GitHub Desktop

GitHub Desktop a diferencia de Git Bash, contiene GUI (graphical user interface) lo que hace que su uso sea más amigable para los usuarios.

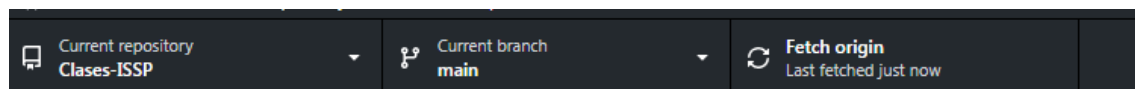


Esta interfaz podemos separarla en 4 partes.

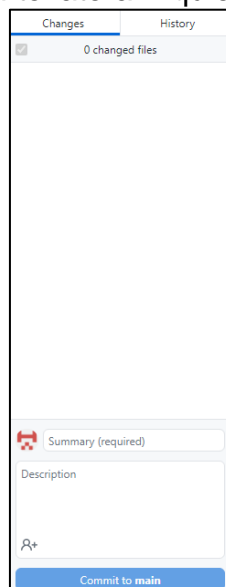
1. Menú de herramientas (ubicado en la parte superior).



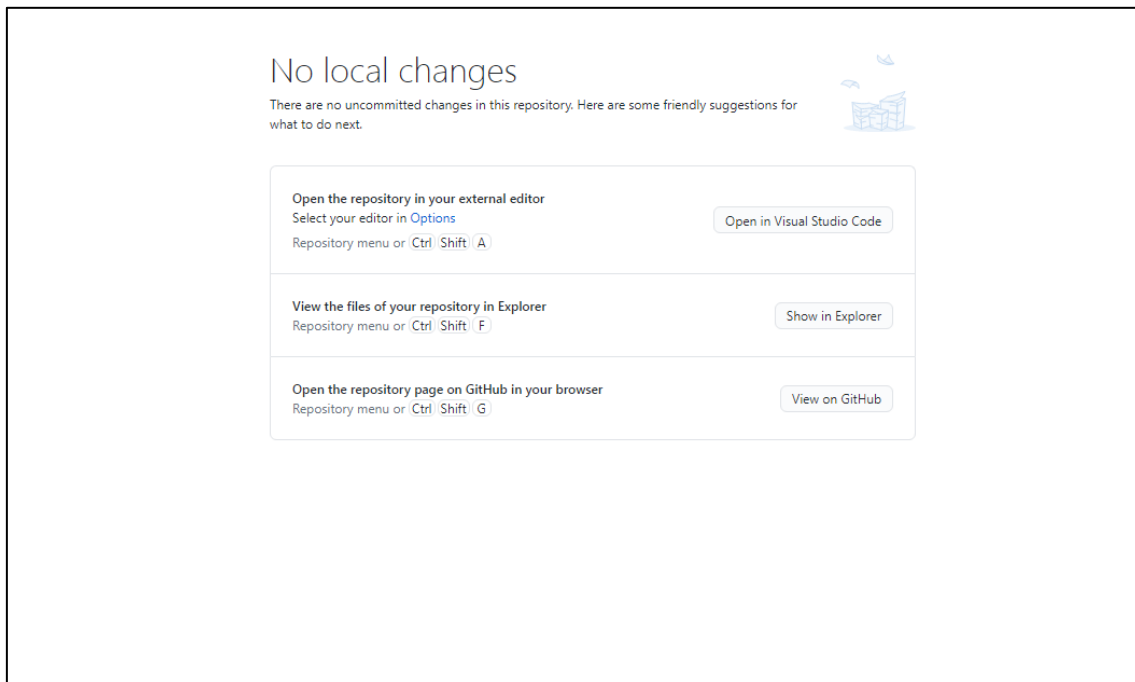
2. Menú de acceso rápido a herramientas del repositorio. (ubicado debajo del menú de herramientas).



3. Menú de cambios (parte lateral izquierda)

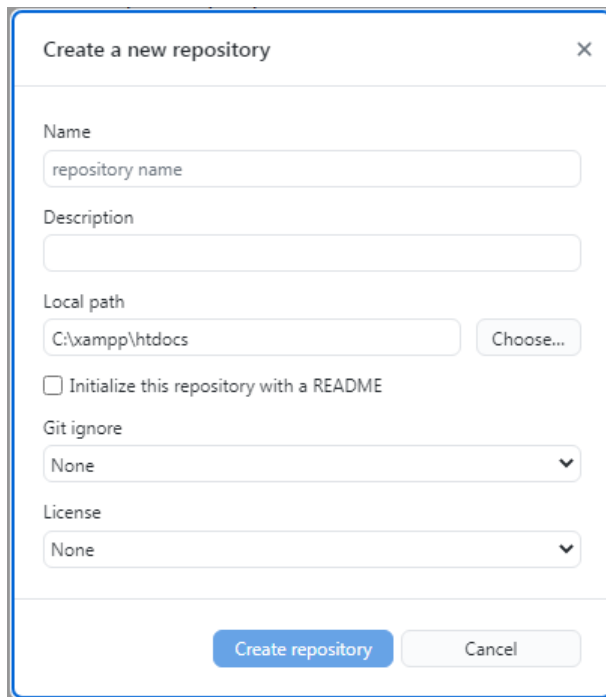


4. Menú de cambios (centro de GUI).



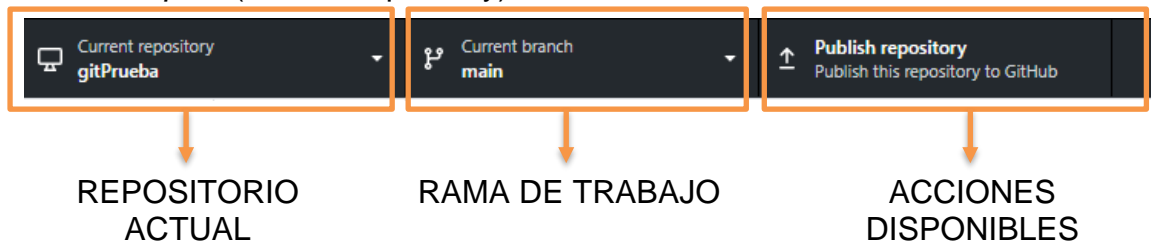
Las acciones que podemos realizar desde la interfaz gráfica son exactamente las mismas que desde la terminal (git bash) solo que aquí las hacemos con unos simples click.

Para poder crear nuestro primer proyecto nos ubicamos en el menú de herramientas *File > New Repository*. Si quisiéramos, desde *File* también tenemos la opción para añadir a la nube un repositorio ya iniciado (*Add local repository*) o clonar uno ya existente (*Clone repository*) desde la nube propia o una URL.

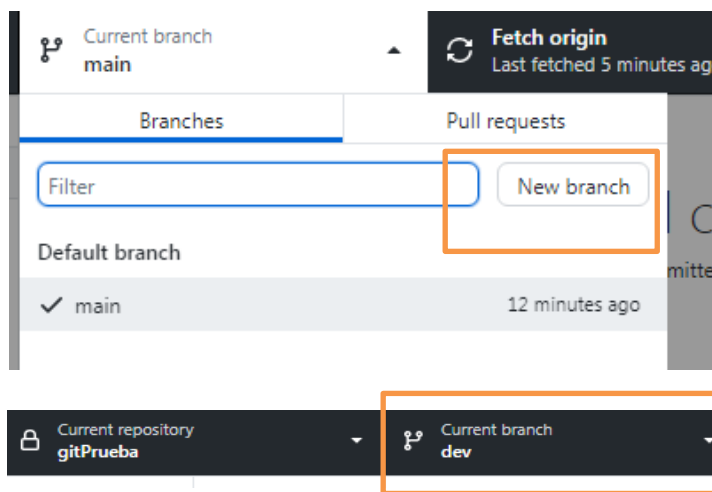


Cuando creamos un nuevo repositorio nos pedirá un nombre (con el que reconoceremos el proyecto), una descripción (no obligatoria) y la ruta donde se alojará el proyecto. Además, nos da la posibilidad de iniciar un README (archivo markdown para primera vista en repositorios de la nube), contener un archivo gitignore (para ignorar archivos o carpetas) y una licencia en caso que existiera.

Una vez creado el repositorio en local, nos permitirá publicarlo en la nube. Podremos realizar esta acción desde el *menú de cambios* o *menú de acceso rápido* (Publish repository).



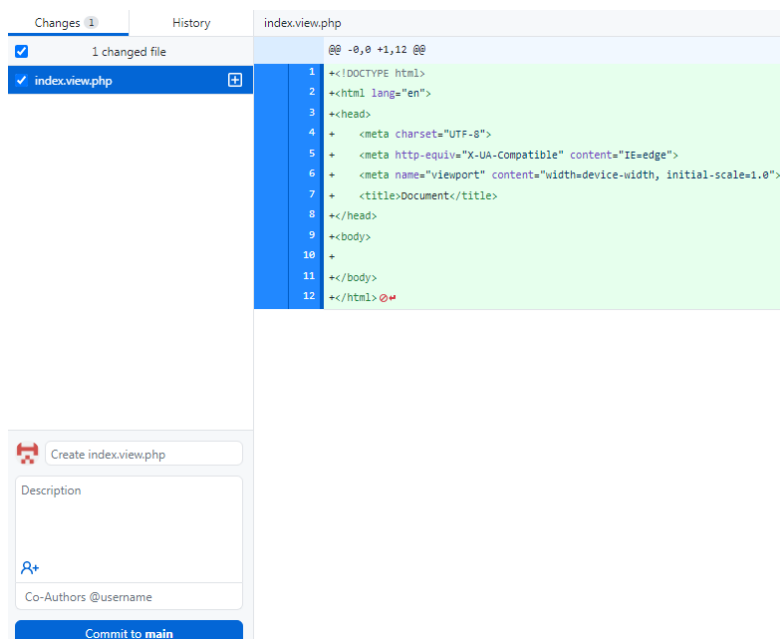
Recordamos que cuando trabajamos con git, es recomendable utilizar ramas distintas a la “master” o “main”. Para crear branch nuevas, podemos hacerlo desde el *menú de herramientas* (Branch > New branch) o desde el *menú de acceso rápido*.



Pulsando sobre el botón “New Branch”, nos abrirá un cuadro de diálogo en el que deberemos colocar el nombre de la rama.

Cuando la creamos, nos cambiará la rama de trabajo y nos permitirá publicar la rama en la nube.

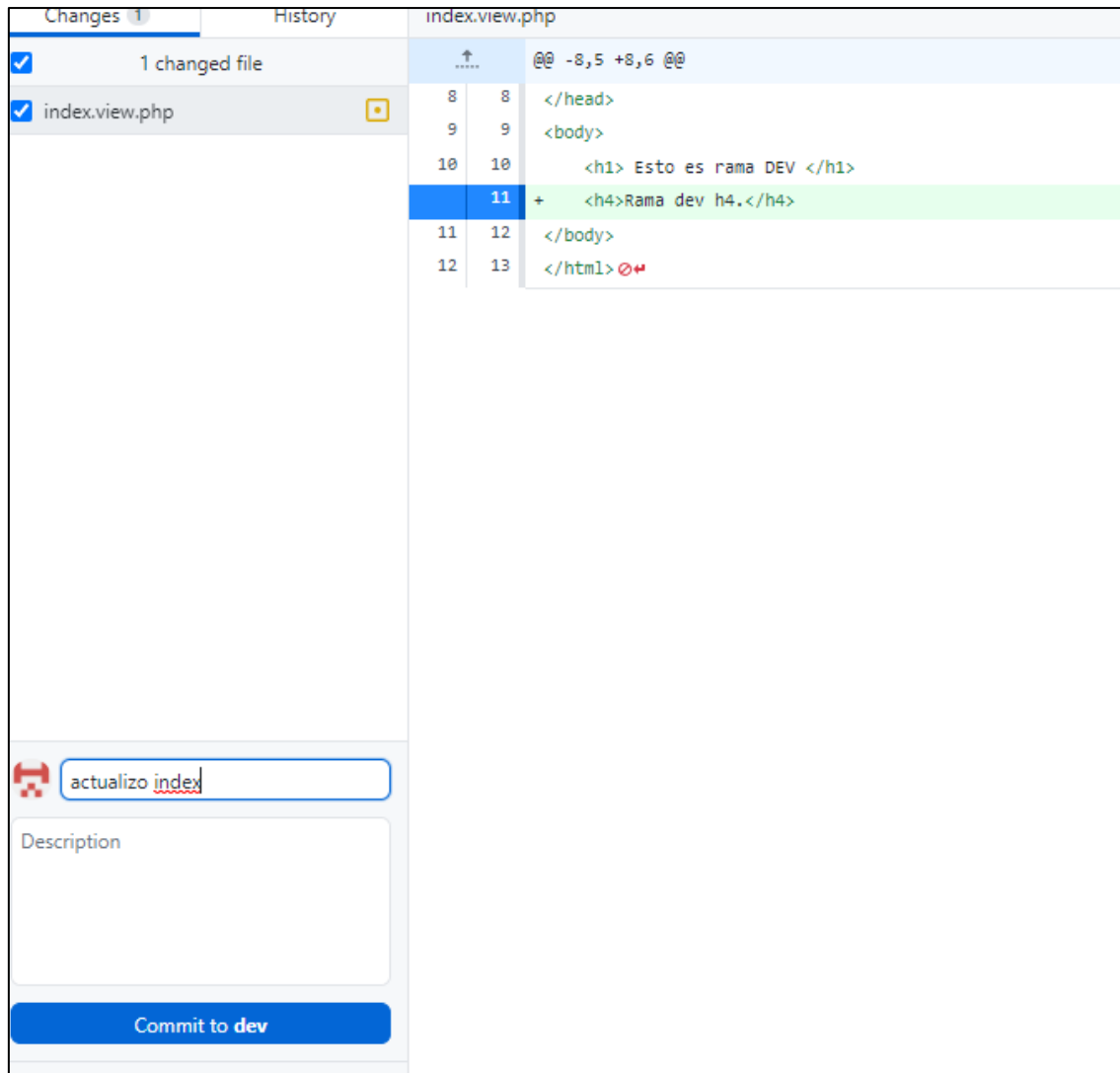
Si quisiéramos volver a la rama main, solo basta con hacer click sobre “Current branch” y seleccionar la rama correspondiente.



A medida que vayamos realizando modificaciones en el proyecto, el *menú de cambios* irá reflejando las actualizaciones realizadas. Una vez terminadas, cuando estamos listos para hacer el commit, solo bastará con completar los campos de la parte inferior del menú de cambios y presionar el botón “Commit to ...”. Luego desde el menú de acceso rápido, tendremos disponible

la acción para realizar el push a la nube.

Trabajaremos ahora sobre una rama *dev*, en la que realizaremos cambios para enviar a producción. En primer lugar debemos asegurarnos que en “*current branch*” tengamos seleccionada la nueva rama. Luego, en el archivo creado anteriormente (*index.view.php*) realizaremos una simple modificación y finalmente, realizar el commit dentro de la rama *dev*.



Al igual que cuando utilizamos bash, debemos realizar *stash* y *rebase* por si algún otro desarrollador realizó modificaciones en la rama principal y evitar futuros conflictos en el código. Para ello, debemos ir *Branch > Stash all changes* y luego a *Branch > Rebase current Branch* (fig. 10). Luego de esta segunda opción, se nos abrirá una nueva ventana en la que seleccionaremos nuestra rama *master* o *main* y daremos click en el botón “Rebase” (fig. 11).

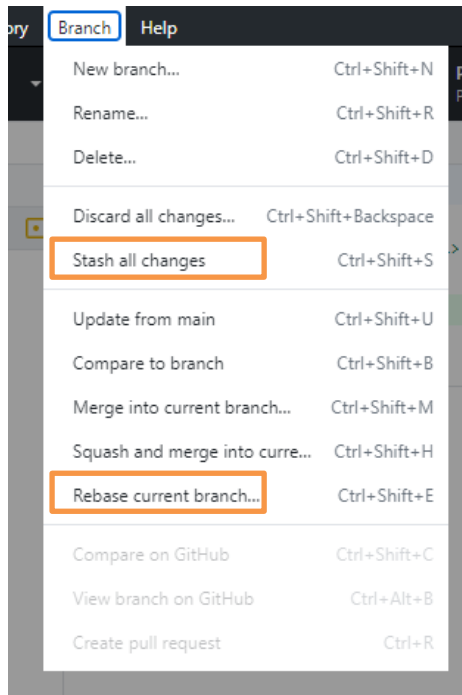


Fig. 9

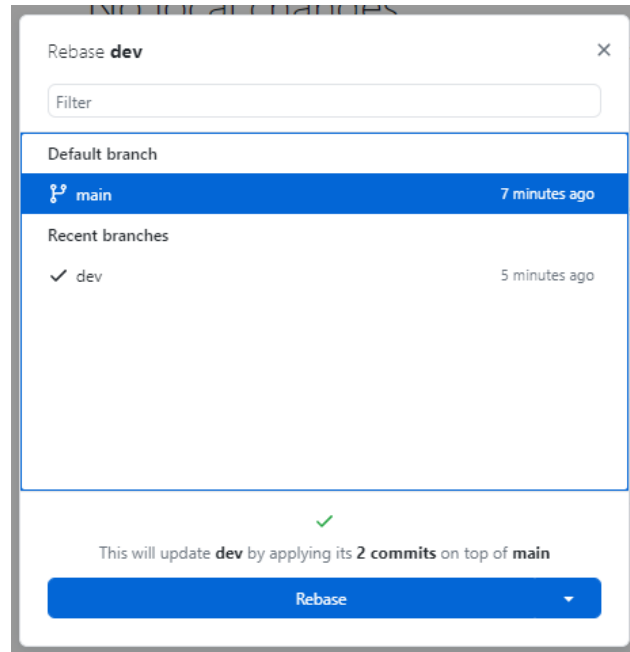


Fig. 8

Si existieran cambios en la rama principal, nos diría que hay conflictos que resolver, esto lo haremos desde vscode.

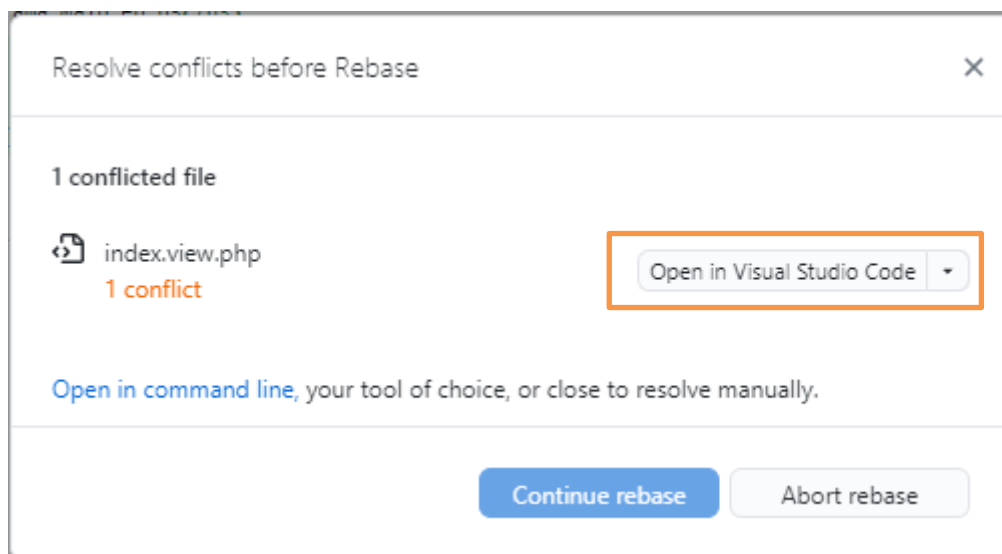


Fig. 10

Cuando se nos abra el editor, veremos que nos marcará en dos colores distintos los cambios actuales, y los cambios que estaban en nuestra rama. Desde ahí mismo, podremos decidir, si aceptar los que vienen de rama main (Accept current change), dejar solo los de la rama descartando los de main (Accept incoming change) o aceptar ambos cambios (Accept both changes), en mi caso aceptaré ambos.


```

Accept Current Change | Accept Incoming Change | Accept Both Changes | Compare Changes
<----- HEAD (Current Change)
  <h2>Esto es rama MAIN</h2>
  <h3>Esto es rama Main en h3</h3>
=====
  <h1> Esto es rama DEV </h1>
>>>>>> 0cd4510 (Update index.view.php) (Incoming Change)
</body>

```

Luego de resolver “los conflictos” GitHub Desktop nos permitirá continuar con el rebase habilitándonos el botón que antes se encontraba deshabilitado (Fig.12). No debemos olvidarnos de recuperar el stash desde el botón “View Stash y luego RESTORE”.

No local changes

There are no uncommitted changes in this repository. Here are some friendly suggestions for what to do next.



View your stashed changes

You have 1 change in progress that you have not yet committed.

When a stash exists, access it at the bottom of the Changes tab to the left.

[View stash](#)

Stashed changes

[Restore](#)
[Discard](#)

Restore will move your stashed files to the Changes list.

index.view.php

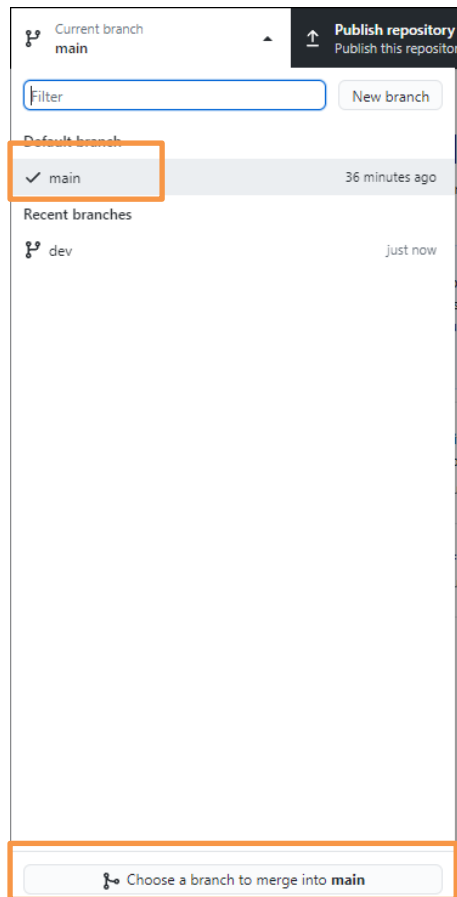


↑

@@ -10,6 +10,6 @@

10	10	<h2>Esto es rama MAIN</h2>
11	11	<h3>Esto es rama Main en h3</h3>
12	12	<h1> Esto es rama DEV </h1>
13	-	<h4>Rama dev h4.</h4>
	+	<h4>DEV Stash</h4>
14	14	</body>
15	15	</html> @+*

Terminados estos pasos, estamos en condiciones de solicitar el pull request para que nuestro devops apruebe la fusión de ramas, o en caso de que tengamos que hacerlo nosotros, procedemos de la siguiente manera.



Debemos asegurarnos de que no tenemos commits pendientes, luego cambiamos a la rama master/main y presionamos el botón que se encuentra en la parte inferior del menú de ramas.

En la nueva ventana, seleccionaremos la rama que deseamos combinar con main y pulsamos el botón “create merge commit”.

