

3. Control de versiones

3.1 Definición

La gestión de varios cambios realizados en los elementos de un producto o en su configuración se denomina control de versiones. Una versión, revisión o versión de un producto es el estado estable en el que el producto se encuentra en un momento específico de desarrollo o modificación.

Los programas de control de versiones permiten a un grupo de personas trabajar juntos en el desarrollo de un proyecto, generalmente a través de Internet.

El control de versiones se realiza principalmente en la industria informática y se utiliza para controlar las diferentes versiones del código fuente, pero también en otras áreas como documentación, imágenes, sitios web y cualquier proyecto colaborativo que normalmente requiera trabajar en paralelo con personas en un equipo.

El control de versiones de código está integrado en el proceso de desarrollo de software de muchas empresas, especialmente si tienen más de un programador trabajando en el mismo proyecto.

3.2 Características

Los programas de control de versiones realizan funciones indispensables a lo largo del ciclo de vida de un proyecto, entre las cuales se encuentran:

- ‡ Permite controlar a los usuarios que trabajarán en paralelo en el proyecto:

- Establezca el usuario que tendrá acceso.

- O asignarles un tipo de acceso.

- ‡ Respecto a la documentación:

- Permite almacenar archivos.

- Permite realizar cambios en los archivos almacenados: modificar parcialmente los archivos, eliminar los archivos, cambiar el nombre de los archivos, mover los archivos,...

- Programar un historial detallado (cambio, fecha, razón, usuario,...) de la hora de la acción realizada

- .

- ‡ Respecto a la versión:

- Etiquetar el archivo en un momento determinado del desarrollo del proyecto para indicar una versión estable, permitiendo así su identificación posterior mediante esa etiqueta.

- Diseño detallado del historial de versiones (usuarios responsables, fechas,...). Permite recuperar todos o parte de los archivos de una versión.

- Compare las versiones mirando o reportando los cambios entre ellas.

- ‡ Con respecto al proyecto, se permite establecer ramas o ramas, es decir, bifurcar el proyecto en dos o más líneas que puedan desarrollarse en paralelo, respectivamente. Las ramas se pueden utilizar para ensayar nuevas características de forma independiente, sin interferir con la línea principal de desarrollo . Si la nueva funcionalidad es estable, la rama recién desarrollada puede fusionarse con la rama principal o el tronco.

Ejemplos:

Un proyecto en el que la versión 1.0 se considera estable y se cierra, y en él se crea una rama para la versión 1.1.

- ‡ En la rama de la versión 1.0 se puede seguir trabajando en la solución de nuevos errores que aparezcan y seguir creando nuevas versiones: 1.01 y posteriores.

- 

Una aplicación del manual de desarrollo de proyectos. Un departamento de una empresa solicitó que algunas partes del manual fueran adaptadas a su forma particular de funcionar.

- 

3.3 Operaciones básicas

El sistema de control de versiones tiene un repositorio local para cada proyecto, donde se almacenan todos los archivos del proyecto. El repositorio debe crearse y administrarse para contener proyectos (nuevos o importados de otro proyecto), tener la estructura de directorio necesaria y ser compartido por un grupo de usuarios autorizados. Adicionalmente podemos ter un repositorio remoto (GitHub, GitLab)

Crear un repositorio local (init)

Esta es la primera acción que permite crear un proyecto existente sin versión en la computadora del usuario por primera vez. Funciona a nivel de carpeta, así que lo que vamos a hacer es ponernos en una carpeta y crear el repositorio.

Añadir archivos al repositorio local (commit)

Esta acción permite actualizar el contenido del repositorio local con los cambios realizados en el código. También se conoce como publicación o check in. En Git veremos que esta operación se divide en dos partes: add y commit (o commit-a).

El usuario puede modificar cualquier archivo en su copia local o crear un nuevo archivo. Una vez que considere que los cambios están completos, debe confirmar los cambios realizados en el directorio de trabajo al repositorio local.

Los cambios deben ir acompañados de un dictamen justificativo para que todos los usuarios puedan conocerlos.

Este paso, en un entorno multiusuario, puede generar conflictos. Por ejemplo:

Los usuarios X e Y procesan el archivo A.

‡ El usuario X publica los cambios entre las líneas n1 y n2 en el archivo A.

‡ El usuario Y no descarga el archivo A del repositorio después de la publicación del usuario X.

El usuario Y realiza cambios entre las líneas n1 Y n2 Y luego intenta publicar esos cambios. En este punto, el sistema detecta que la copia local en la que el usuario Y está trabajando ha cambiado Y no puede realizar cambios automáticamente. El usuario Y debe resolver el conflicto fusionando los cambios o seleccionar uno Y descartar el otro.

Para favorecer el trabajo en equipo, se recomienda que las tareas asignadas al desarrollo se resuelvan a corto plazo.

Actualizar el directorio de trabajo desde el repositorio (check-out)

Esta acción permite actualizar el directorio de trabajo con los cambios realizados en el repositorio local (archivos modificados, directorios nuevos, archivos nuevos, directorios dejados en blanco en el directorio, etc.). También conocido como sincronización.

Cuando un usuario hace cambios en un archivo y lo publica en un repositorio, los directorios de trabajo de otros usuarios se vuelven obsoletos.

Crear ramas (branch)

Esta operación permite la bifurcación del proyecto en ramas, lo que dará lugar a una evolución paralela del código, de modo que se pueda fusionar entre ellos en cualquier momento.

Ejemplos:

- ‡ Cuando un software entra en producción, se cierra una de sus versiones y se puede crear una rama evolutiva en la que el proyecto continuará evolucionando, y otro fix en el que se arreglarán posibles errores.
- ‡ Creación de ramificaciones para realizar modificaciones en versiones que puedan causar inestabilidad en el tronco.

La gestión de las versiones en desarrollo se complica a medida que se establecen ramas.

Las ramas deben ser tratadas como una vida limitada, ya sea cerrando una versión, incorporando los cambios en la rama que creó la rama o usándola para crear un nuevo proyecto.

Descargar clone/pull repositorio remoto

Esta operación permite al usuario crear un directorio de trabajo en el disco duro local que contiene una copia de la versión del repositorio, generalmente la última versión. Haga esto la primera vez que descargue un proyecto desde un repositorio.

Fusión

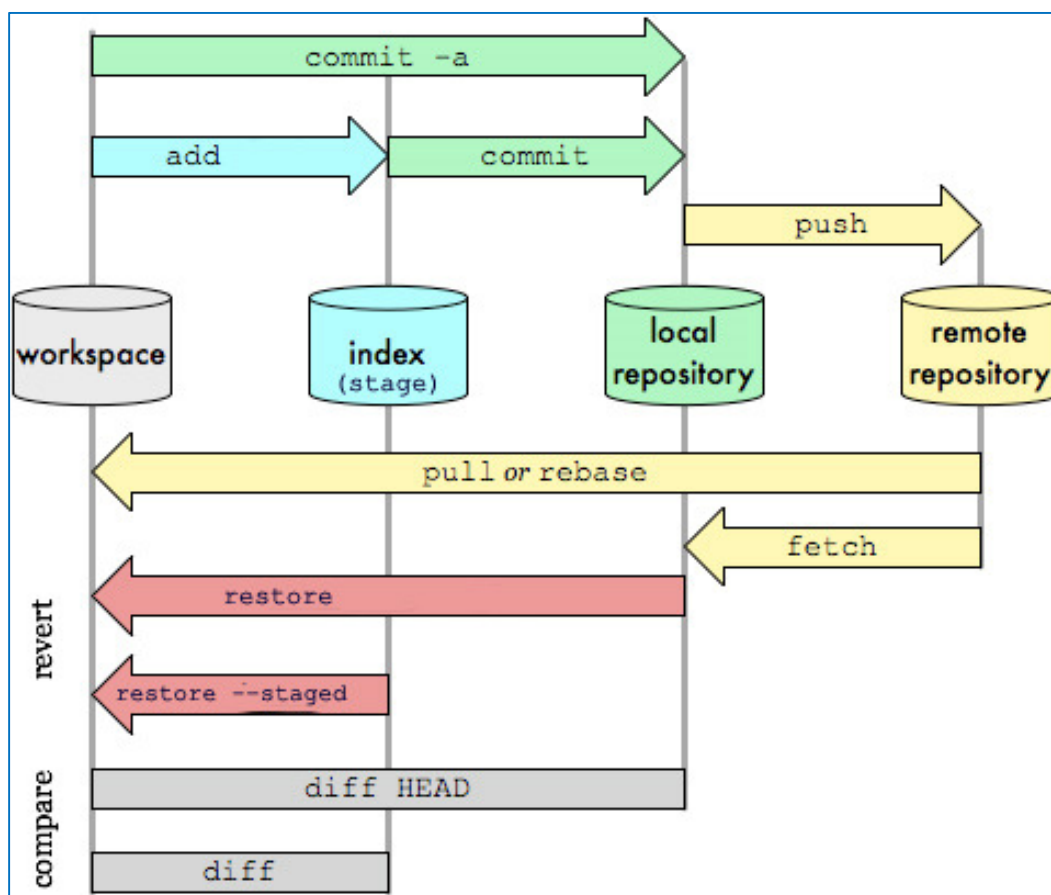
Esta acción permite que todos los cambios realizados entre dos versiones en una rama se apliquen a cualquier otra rama del repositorio.

Ejemplo: Supongamos que hay una rama de corrección y una rama de evolución, y se hacen cambios en la rama de corrección para corregir un error, entonces los cambios realizados en la rama de corrección se pueden fusionar con los cambios en la rama de evolución para que el error no esté incluido en la rama de evolución.

Funcionamiento diff

Puede ser interesante ver los cambios realizados en relación con el repositorio antes de confirmar los cambios en el repositorio. Esta operación se denomina diff.

La siguiente figura muestra un resumen del código que se mueve entre el directorio de trabajo, el repositorio local y el repositorio remoto. Comandos que veremos en la siguiente sección, en la herramienta git.



3.4 Clasificación

El programa de control de versiones se basa en:

Modelo cliente-servidor

Hay un repositorio centralizado de todo el código en el servidor, con un usuario responsable, y el usuario autorizado a acceder a través del cliente. Las tareas administrativas se hacen más fáciles a expensas de reducir la flexibilidad, ya que todas las decisiones fuertes, como la creación de nuevas sucursales, requieren la aprobación de la persona a cargo.

Modelo distribuido

Cada usuario trabaja directamente con su repositorio local, que actúa como cliente y servidor al mismo tiempo. No es necesario centralizar la toma de decisiones. Los diferentes repositorios pueden intercambiarse entre sí y mezclar revisiones.

Programas software libre

‡ Modelo cliente-servidor

o CVS (sistema de versiones simultáneas <http://www.cvshome.org/>): Desarrollado por GNU. Se distribuye bajo la licencia GPL.

SVN (subversion.apache.org/): el más popular hoy en día. Fue creado principalmente para mejorar CVS en el manejo de archivos binarios. Fue distribuido en el entorno Apache/BSD.

‡ Modelo distribuido

o Git (<http://git-scm.com/>): Diseñado por Linus Torvalds, está basado en BitKeeper e Monotone. Se distribuye bajo la licencia GNU GPL v2. Para proyectos de programación del kernel de Linux.

o Mercurial (<https://www.mercurial-scm.org/>): Creado por Matt Mackall. Se distribuye bajo la licencia GNU GPL.

Bazaar (<http://wiki.bazaar.canonical.com/>): Desarrollado por Canonical LTF y la comunidad. Facilita proyectos de software libre y de código abierto. Se distribuye bajo la licencia GNU GPL v2.

Programas software propietario

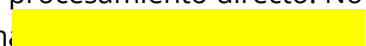
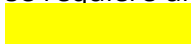
‡ Modelo cliente-servidor

Visual SourceSafe: Herramienta de control de versiones de Microsoft para equipos pequeños, integrada en el entorno de trabajo de Visual Studio y otras herramientas de desarrollo de Microsoft. Actualmente está siendo reemplazado por Visual Studio Team Foundation Server (<http://www.visualstudio.com/es-es>).

‡ Modelo distribuido

BitKeeper (<http://www.bitkeeper.com/>) es producido por Bitmover Inc.

Diferencias

	SVN	Ir
Modelo	Centralizado	Distribuido
Repositorio	Un repositorio central. Necesita tener una conexión con el repositorio.	Copia local de procesamiento directo. No se requiere un  
Visitas	Depende de la ruta de acceso	A todo o director.
Seguimiento	Basado en archivos	Contenido basado
Historial	Entonces no hay repositorio completo	Repositorios centrales y locales
Connectividad	En cada visita	Solo en la sincronización

4. Ir

4.1 Descripción

Es un software de control de versiones diseñado por Linus Torvalds para mantener la eficiencia y confiabilidad de una versión de una aplicación cuando tiene una gran cantidad de archivos de código fuente, como el kernel de Linux.

El diseño de Git se basa en BitKeeper y Monotone. Originalmente era un motor de sistema de control de versiones de bajo nivel en el que otros pueden realizar tareas, aunque ha evolucionado hasta convertirse en un sistema de control de versiones completo.

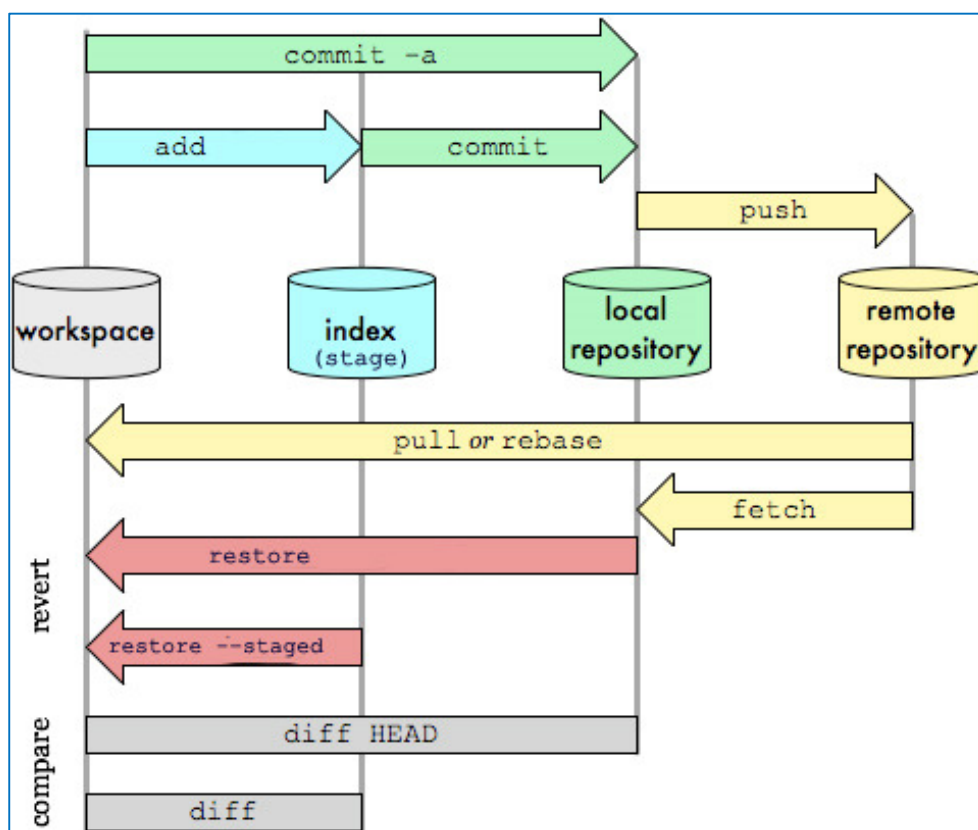
En Git, un archivo puede estar en tres estados:

untracked: Hay cambios en su copia local (directorio de trabajo), pero git no ha notificado estos cambios. Dado que estos archivos git no existen al mismo tiempo, no habrá posibilidad de recuperación si algo sale mal.

▪ **Preparación (staged):** En el punto intermedio anterior, el archivo se modifica y está listo para ser actualizado en el repositorio en el momento que deseamos, en este punto estamos listos para actualizar todos los archivos en el repositorio con la nueva versión. Esta zona se conoce como **índice**.

‡ **Committed:** Sus datos han sido actualizados de forma segura en el repositorio local. Más adelante, si les damos algunos cambios, es posible que lo recuperemos a este estado. Podríamos decir que una presentación tomará una foto del archivo y lo guardará permanentemente.

Estas tres situaciones determinarán el comando para mover un archivo de una situación o región a otra.



Boas prácticas: "Git Flow", "GitHub Flow", etc...

Los modelos de desarrollo de versiones varían. Un típico e "Git Flow", que se basa en las líneas que se muestran en la siguiente imagen:



Cada desarrollador o equipo de desarrollo puede usar Git de la manera que considere más adecuada. Sin embargo, una buena práctica es:

Débense utilizar 4 consejos de ramas: maestro, desarrollo, funciones, y corrección.

Maestro: Esta es la rama principal. Contiene repositorios que se lanzan en producción y, por lo tanto, debe mantenerse estable en todo momento.

Desarrollo: Es una rama extraída del "Maestro". Es la rama de integración, todas las nuevas funciones deben integrarse en esta rama. Una vez que se ha realizado la integración y se han corregido los errores (si los hubiera), es decir, se ha encontrado que la rama es estable, se puede realizar una fusión de "desarrollo" en la rama "maestra".

Características: Cada nueva función debe ejecutarse en una nueva rama específica para esa función. Estos deben ser eliminados del desarrollo. Una vez que se completa la función, la rama de desarrollo se fusiona, donde se integrará con otras funciones.

Hotfix: Son errores que aparecen en la producción, por lo que deben ser parchados y liberados con urgencia. Es por eso que todas son ramas arrancadas del Maestro. Una vez que se corrigen los errores, se debe realizar una fusión de ramas en el maestro. Por último, para no dejar desactualizado, es necesario realizar la fusión de Master over Development.

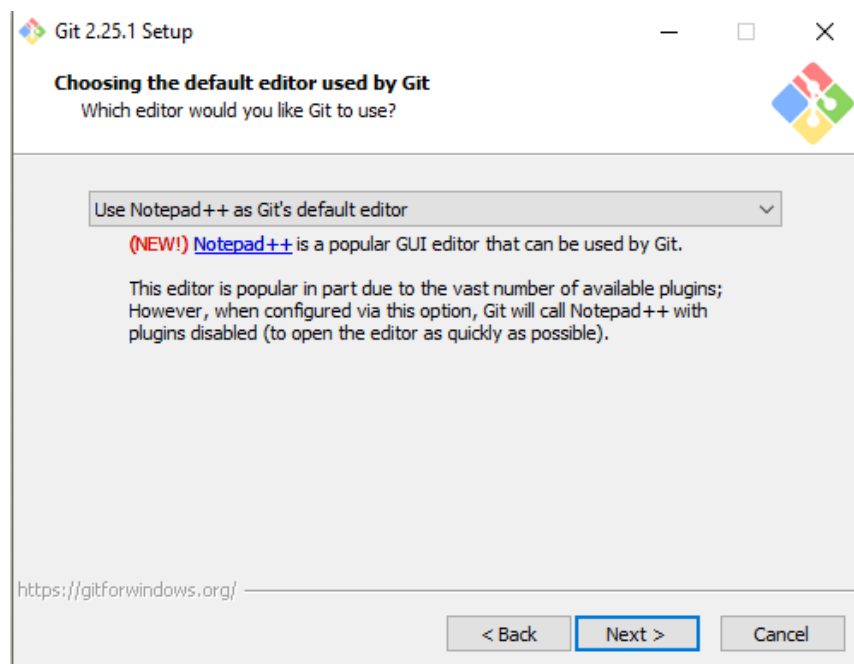
GitHub Flow ha cambiado ligeramente el modo de trabajo. No hay una rama de "desarrollo", pero las nuevas características comienzan directamente desde la rama principal. El despliegue también es diferente porque se realiza a nivel de característica y no a nivel maestro.

4.2 Instalación Git en Windows

Descargaremos la herramienta oficial de Git para Windows desde <https://git-scm.com/download/win>. Integración con el explorador de archivos.



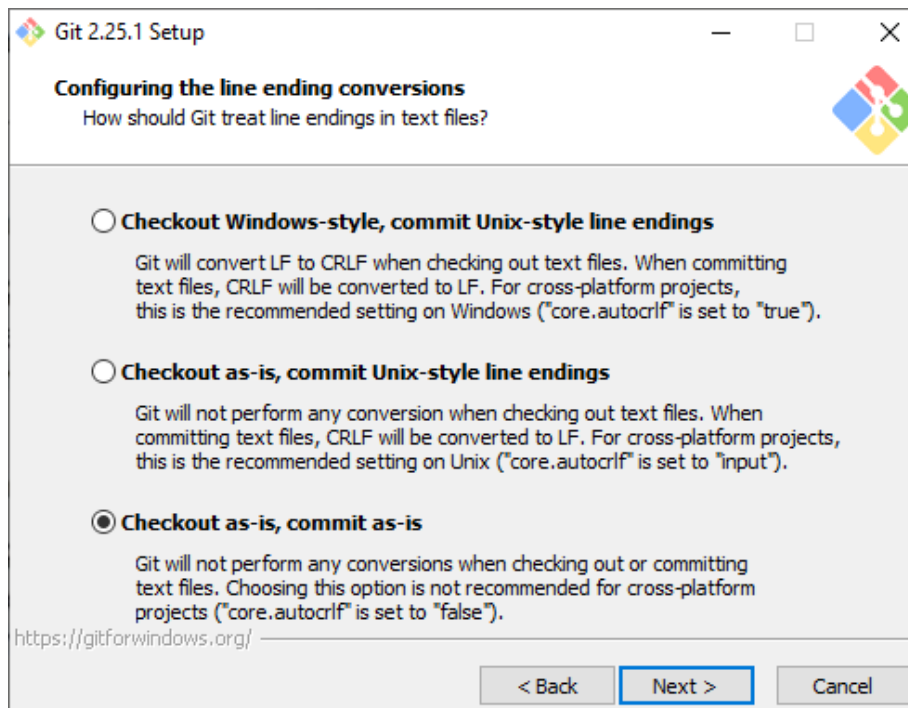
En la instalación hemos mantenido todas las ventanas, excepto seleccionando el editor predeterminado, marcando Notepad ++.



Si no marcamos esta opción, entonces al usar git podemos asociar Notepad++ con la aplicación con el siguiente comando:

```
Git config--global core.editor '"C:/Program Files
(x86) /notepad ++/notepad ++.exe '-multiinst-notabbar-nosession-noplugin "
```

Finalmente, en el lugar donde se indica el procesamiento de fin de línea (estilo Windows o Linux), marcamos la tercera opción: checkout como es, commit como es, ya que solo vamos a trabajar en un solo sistema operativo.



También utilizaremos una herramienta de visualización llamada Tkdifff para ver las diferencias entre los dos archivos de texto. Es un programa simple pero muy intuitivo que puede usar el color para ver las diferencias entre las líneas de código (la herramienta predeterminada que se ve en el gif es vimdiff, que es más complicada). Descargamos o .exe c <https://sourceforge.net/projects/tkdiff/> instalado, copiamos el archivo tkdiff.exe de c:\ProgramFiles (x86)\tkdiff a c:\ProgramFiles\git\usr\bin.

Hay otras opciones que no funcionan desde la consola, como Atlassian SourceTree o GitKraken.

4.3 Creación de un repositorio

El primer paso será crear una nueva carpeta llamada misproyectos y hacer clic derecho en: Git Bash Aquí

En la consola que aparece hemos creado una nueva carpeta para nuestro primer proyecto: mkdir proyecto1. En la consola tenemos comandos de estilo linux como pwd, cd, ls, rm-rf.

Para crear un repositorio git, ejecutamos creando una carpeta oculta. git: git init, que contiene la información necesaria para la administración de la herramienta. Podes comprobalo con ls -lart .git/.

```
Usuario@DESKTOP-VVH9EQM MINGW64 ~/Desktop/misproyectos
$ git init
Initialized empty Git repository in C:/Users/Usuario/Desktop/misproyectos/.git/

Usuario@DESKTOP-VVH9EQM MINGW64 ~/Desktop/misproyectos (master)
$ ls -lart .git/
total 11
drwxr-xr-x 1 Usuario 197121  0 feb. 27 16:27 ../
-rw-r--r-- 1 Usuario 197121 73 feb. 27 16:27 description
drwxr-xr-x 1 Usuario 197121  0 feb. 27 16:27 hooks/
drwxr-xr-x 1 Usuario 197121  0 feb. 27 16:27 info/
drwxr-xr-x 1 Usuario 197121  0 feb. 27 16:27 refs/
-rw-r--r-- 1 Usuario 197121 23 feb. 27 16:27 HEAD
-rw-r--r-- 1 Usuario 197121 130 feb. 27 16:27 config
drwxr-xr-x 1 Usuario 197121  0 feb. 27 16:27 ./
drwxr-xr-x 1 Usuario 197121  0 feb. 27 16:27 objects/
```

También debe proporcionar dos parámetros, el correo electrónico y nuestro nombre de usuario:

```
Git config--global user.name "Fernando RD"
```

```
Git config--global user.email "rdf@fernandowirtz.com"
```

Esta configuración almacenase en `./gitconfig`

4.4 Operaciones básicas

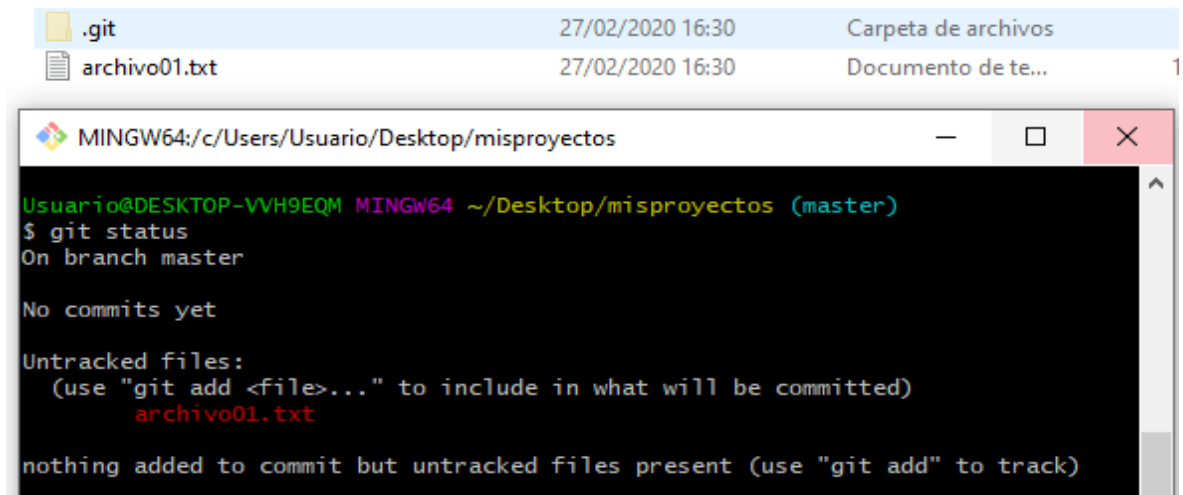
OLLE: Revise el gráfico al comienzo de este tema, que contiene el modo de operación git entre el directorio de trabajo/staged (o index)/Local Repository/Remote Repository.

Añadir archivos

Si ahora creamos un archivo en esa carpeta (por ejemplo nuestro código de aplicación), ahora Está fuera de git, no está grabado, no está grabado. Podemos usar

Estado git:

Estado git



Los archivos no registrados se muestran en rojo. El primer paso es ingresarlo en el sistema, es decir, dejar que git lo grabe como un archivo sobre la administración de versiones. Para iso ejecutamos el siguiente comando:

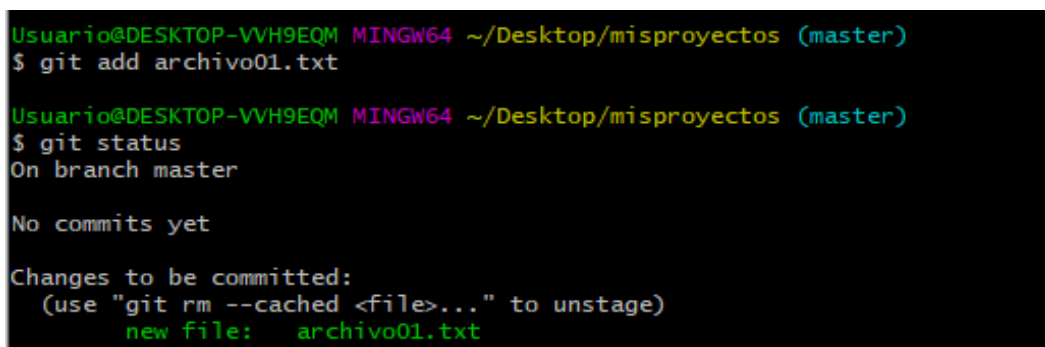
Git agregar

```
Git archivo add.txt → cargar archivo
```

```
git add carpeta < > engade toda a carpeta
```

Para añadir más archivos a la vez, podemos indicar todos los archivos en el directorio actual con puntos: `git add`. O con `-a` o `--all`. Utilice también la máscara: `git add*.txt`

Ahora repetimos `git status` para comprobar la situación. Comprobamos que se encuentra en el primer supuesto caso, es decir, `Estimado en` para ser enviado al repositorio.



Estos documentos aparecen en verde y están listos para su aprobación.

Ahora si hacemos **Commit git**, el documento se registrará en una "fotografía" o comentario, en el que con el tiempo podremos llevar a cabo nuestro trabajo. No es necesario especificar un nombre de archivo, simplemente envíe todos los archivos en el estado escalado (los archivos que se muestran en verde). Ejecutamos con el parámetro -m para indicar textualmente el objeto de esta revisión o versión

```
git commit -m "primer commit"
```

`git commit -a -m "second-commit"` Commit todos los archivos modificados o eliminados, pero no los nuevos (los archivos nuevos deben agregarse primero)

Si hacemos `git status` ahora veremos que nada está suspendido, mientras que a través de: `git log` podemos ver el historial de los cambios.

```
Usuario@DESKTOP-VVH9EQM MINGW64 ~/Desktop/misproyectos (master)
$ git commit -m "primer commit"
[master (root-commit) 88c6a88] primer commit
1 file changed, 4 insertions(+)
create mode 100644 archivo01.txt

Usuario@DESKTOP-VVH9EQM MINGW64 ~/Desktop/misproyectos (master)
$ git status
On branch master
nothing to commit, working tree clean

Usuario@DESKTOP-VVH9EQM MINGW64 ~/Desktop/misproyectos (master)
$ git log --oneline
88c6a88 (HEAD -> master) primer commit
```

Estas opciones de `git log` son interesantes: `git log --oneline --decorate --graph --all` diferencias entre archivos

Si modificamos un archivo que ya se ha comprometido, `git` detectará la diferencia entre los dos (la de la versión del directorio de trabajo y la de la zona de compromiso). Para ilustrar más, hemos modificado el archivo `archivo01.txt` insertando una línea.

```
Usuario@DESKTOP-VVH9EQM MINGW64 ~/Desktop/misproyectos (master)
$ git status
On branch master
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:   archivo01.txt

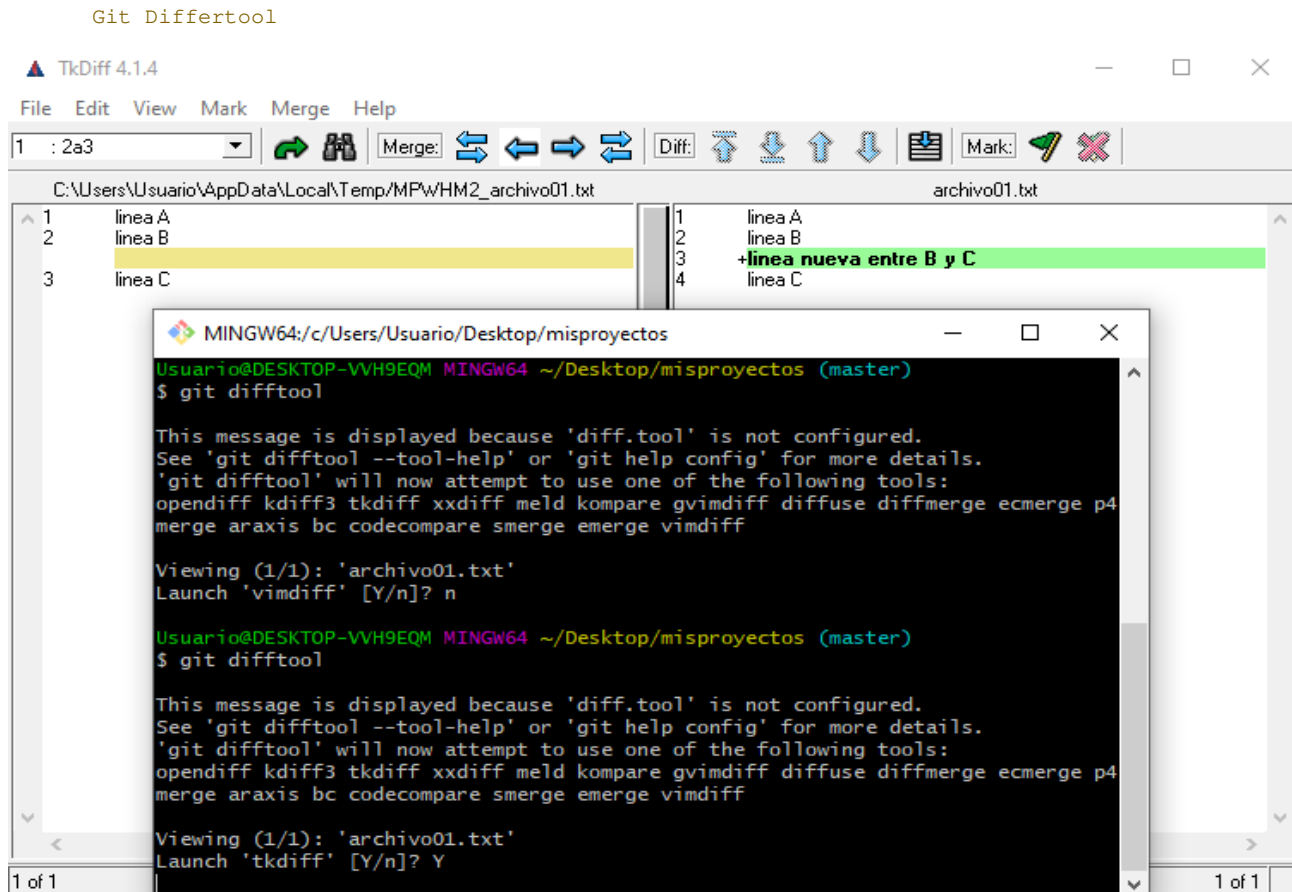
no changes added to commit (use "git add" and/or "git commit -a")
```

Depende de la diferencia **git** **Git Differential Tool**.

`git diferencia`

```
Usuario@DESKTOP-VVH9EQM MINGW64 ~/Desktop/misproyectos (master)
$ git diff
diff --git a/archivo01.txt b/archivo01.txt
index 9359161..7f12cdb 100644
--- a/archivo01.txt
+++ b/archivo01.txt
@@ -1,3 +1,4 @@
 línea A
 línea B
+línea nueva entre B y C
 línea C
```

Y a través de la herramienta tkdiff que hemos instalado:



git diff y git diff --staged comparan las diferencias de archivos entre aquellos que están en el directorio de trabajo y el repositorio (commit), no aquellos que ya están en staging, es decir, no tienen en cuenta los archivos que ya hemos añadido. Para comparar staged con lo que ya existe en el área de compromiso, agregamos el parámetro --staged --cached

Desenfocar y renombrar archivos

Siempre lo haremos y `git rm` `git mv` git registre los cambios por separado. Después de cualquiera de estas dos operaciones, debemos hacer un commit, es decir, confirmar los cambios en el HEAD. Si eliminamos los renombradores fuera de git, se destracked, es decir, no se registran en git.

Deshacer cambios en un archivo

Ahora vamos a tratar el caso de la manera inversa, si con add y commit pasamos los archivos desde el directorio de trabajo a stage y commit respectivamente, ahora veremos lo contrario, como pasar los cambios ya aprobados (ya sea en stage o en commit) a nuestro directorio de trabajo, sobrescribiendo la versión que tenemos en el directorio de trabajo.

Ordenes **Git check out** Tiene varias funciones que veremos a lo largo del tema, a través de Ejemplo para cambiar nuestras ramas, y también primero votar para ver los commits anteriores, y luego votar hasta la situación actual, que es el último commit, pero en este caso lo usaremos para pas **Archivo de la sección de presentación** Al directorio de trabajo para que descartemos los cambios que hemos hecho en el directorio de trabajo.

`git check-out` archivos (o archivos con máscaras)
`git checkout.` (Recuperar todos los documentos con puntos adjuntos)

```

Usuario@DESKTOP-VVH9EQM MINGW64 ~/Desktop/proy (master)
$ git status
On branch master
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:   file.txt

no changes added to commit (use "git add" and/or "git commit -a")

Usuario@DESKTOP-VVH9EQM MINGW64 ~/Desktop/proy (master)
$ git checkout file.txt
Updated 1 path from the index

```

En el ejemplo, hemos realizado cambios en el directorio de trabajo. Al ejecutar checkout, perderemos estos cambios devolviendo la situación del último commit. Esta última versión se llama HEAD.

También podemos recuperar archivos de commits anteriores, para lo cual añadimos el ID COMMIT al comando.

`git checkout idRevision archivo`

```

Usuario@DESKTOP-VVH9EQM MINGW64 ~/Desktop/proy (master)
$ git log --oneline
6f220a7 (HEAD -> master) commit 3
240c547 commit 2
7ca7425 commit 1

Usuario@DESKTOP-VVH9EQM MINGW64 ~/Desktop/proy (master)
$ git checkout 7ca7425 file.txt
Updated 1 path from b54fb3a

```

Reanudará el commit, pero desde el primer commit, no el último.

El último commit se llama HEAD, como puedes ver en el ejemplo, por lo que a veces verás suspendido ese parámetro, aunque no es exacto porque toma la última versión por defecto.

`git checkout HEAD archivo`

Restablecimiento de

Para saltar desde la sección de escenario al directorio de trabajo **Restablecimiento de** Esto es útil cuando pensamos que estamos listos para pasar los cambios al repositorio, pero queremos modificar algo nuevo.

```

Usuario@DESKTOP-VVH9EQM MINGW64 ~/Desktop/proy (master)
$ git add file.txt

Usuario@DESKTOP-VVH9EQM MINGW64 ~/Desktop/proy (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   file.txt

Usuario@DESKTOP-VVH9EQM MINGW64 ~/Desktop/proy (master)
$ git reset file.txt
Unstaged changes after reset:
M       file.txt

Usuario@DESKTOP-VVH9EQM MINGW64 ~/Desktop/proy (master)
$ git status
On branch master
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:   file.txt

no changes added to commit (use "git add" and/or "git commit -a")

```

Otra herramienta para deshacer cambios es **git restore** que puede deshacer los cambios que aún no han sido confirmados de la copia en el área de confirmación. Es similar a checkout y reset, pero lo usaremos en el caso de un archivo eliminado en el directorio de trabajo/escala que queremos recuperar.

- Si el archivo se elimina fuera de git y no se añaden cambios, el archivo eliminado se encuentra en el directorio de trabajo (en rojo en el estado de git) y el comando a usar será:

Git restaurar fich

```

Usuario@DESKTOP-VVH9EQM MINGW64 ~/Desktop/proy (master)
$ git status
On branch master
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        deleted:    file.txt

no changes added to commit (use "git add" and/or "git commit -a")

Usuario@DESKTOP-VVH9EQM MINGW64 ~/Desktop/proy (master)
$ git restore file.txt

Usuario@DESKTOP-VVH9EQM MINGW64 ~/Desktop/proy (master)
$ git status
On branch master
nothing to commit, working tree clean

```

- Si el archivo es borrado por git rm, o fuera de git, pero se ha añadido la eliminación, estará en el área de etapa, por lo que primero haremos lo siguiente:

Git restore--staged fich

Para llevarlo al caso del ejemplo anterior, es decir, al área del directorio de trabajo, y luego como el ejemplo anterior:

Git restaurar fich

```

Usuario@DESKTOP-VVH9EQM MINGW64 ~/Desktop/proy (master)
$ git rm file.txt
rm 'file.txt'

Usuario@DESKTOP-VVH9EQM MINGW64 ~/Desktop/proy (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        deleted:    file.txt

Usuario@DESKTOP-VVH9EQM MINGW64 ~/Desktop/proy (master)
$ git restore --staged file.txt

Usuario@DESKTOP-VVH9EQM MINGW64 ~/Desktop/proy (master)
$ git status
On branch master
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        deleted:    file.txt

no changes added to commit (use "git add" and/or "git commit -a")

Usuario@DESKTOP-VVH9EQM MINGW64 ~/Desktop/proy (master)
$ git restore file.txt

```

Diferentes situaciones de los archivos a la hora de borrar /restaurar

Borrados por fuera de git (p.ej. borrado desde el explorador de archivos)

* Si creo un archivo y lo borro (por fuera de git)

Con un git status no vemos nada. Ahí git no sabe nada de él y por lo tanto no es capaz de recuperarlo con restore. Suponemos que nunca se hizo commit sobre ese archivo.

* Si creo un archivo, le hago add y lo borro (por fuera de git)

Con un git status vemos el delete en rojo (un borrado no registrado para hacer commit de ese borrado) Si hacemos un restore, git lo conoce (lo tuvo en área de stage un tiempo) así que sí es capaz de recuperarlo.

* Si creo un archivo, le hago add+commit y lo borro (por fuera de git)

Igual que el caso anterior. Con un git status vemos el delete en rojo (un borrado no registrado para hacer commit de ese borrado) Si hacemos un restore, git lo conoce (lo tiene en el HEAD) así que sí es capaz de recuperarlo.

Borrados con "git rm"

La diferencia fundamental con el borrado por fuera, es que pasa ese borrado al área de stage. Es decir, es un borrado registrado para hacerle un commit a ese borrado y por tanto borrar el archivo que estuviese en HEAD (porque le hicimos un commit anteriormente).

* Si creo un archivo y lo borro con git rm

ahí git no sabe nada de él, no nos deja hacer git rm (y por tanto no lo borra)

* Si creo un archivo, le hago add y lo borro con git rm

No deja hacer git rm directamente, porque git rm borra del working directory y este ya está en stage/index. Tiene miedo a que borremos algo que tenemos listo para hacer commit.

Habría que hacer git rm -f (es f de force) y en ese caso lo elimina totalmente o bien git rm --cached, esto elimina el paso add. En cualquier caso, volveríamos al caso inicial, no es capaz de recuperarlo, porque el fichero no está registrado.

* Si creo un archivo, le hago add+commit y lo borro con git rm

Este es el caso más habitual. Es un archivo que ya está en el HEAD, ya hemos hecho commit previamente.

Al hacer git rm borra el archivo del working directory, y deja el borrado en staged (delete en verde)

Lo normal ahora sería confirmar dicho borrado en el HEAD haciendo un git commit.

Pero también podríamos arrepentirnos y haríamos primero git restore --staged para pasar ese cambio de staged/index a working directory (veríamos el delete en rojo) y a continuación git restore para recuperarlo.

Deshacer cambios en el historial de compromisos

En esta subsección, no discutiremos los cambios realizados en un archivo específico, sino el historial de los cambios realizados, es decir, modificar los commites del repositorio.

Restablecer-duro

En la sección anterior vimos el comando de restablecimiento que pasa archivos desde la sección de etapas al directorio de trabajo. La **--duro** de restablecimiento elimina completamente los compromisos anteriores, sin dejar rastro. Esto es peligroso porque perderemos el historial desde el historial de confirmaciones (por ejemplo, si anteriormente hemos empujado un servidor remoto, podría haber alguna confusión de confirmaciones remotas que no existen localmente). Por ejemplo:

```
Git reset--hard idrevision
```

```
Git reset--cabeza dura ~ 2
```

La coma después de HEAD indica el número de revisión, HEAD es el último commit, HEAD ~1 es el penúltimo, y así sucesivamente. (La coma está escrita en **Alt + 1** **2** **3** en el teclado numérico)


```

Usuario@DESKTOP-VVH9EQM MINGW64 ~/Desktop/proy (master)
$ git log --oneline
b0c59f2 (HEAD -> master) commit 5
a60ffa9 commit 4
6f220a7 commit 3
240c547 commit 2
7ca7425 commit 1

Usuario@DESKTOP-VVH9EQM MINGW64 ~/Desktop/proy (master)
$ git reset --hard HEAD~2
HEAD is now at 6f220a7 commit 3

Usuario@DESKTOP-VVH9EQM MINGW64 ~/Desktop/proy (master)
$ git log --oneline
6f220a7 (HEAD -> master) commit 3
240c547 commit 2
7ca7425 commit 1

```

En el ejemplo: `git reset --hard head~2` será lo mismo que: `git reset 6f220a7`

Como se muestra en la imagen, necesitamos especificar en commit lo que queremos mover, esto será Situación definitiva, no es lo que queremos y Prohibición.

En principio: `git reset --hard head` no tendría ningún sentido, ya que llevaría el repositorio local a su estado actual, ¡pero ya está allí! Este comando se utiliza para deshacerse rápidamente de todos los cambios que realizamos sin confirmación, ya sea en el directorio de trabajo o en la etapa.

Volver a la vuelta

Volver a la vuelta

Tiene una función similar a `reset --hard`, ya que permite volver a situaciones anteriores

En el área de presentación. La mayor diferencia entre ellos es que `revert` no los elimina sin dejar rastro

Commit, de lo contrario se creará un nuevo commit, descartando los commits anteriores uno por uno.

Podemos especificar un solo compromiso para revertir, o un rango de ellos (usando entre el más reciente y el más antiguo..., pero el más antiguo no está incluido).

Ejemplos:

```

git revert HEAD//retrocede el último commit
Git revert 899001f--no-edit//reverter o commit 899001f git revert head...
head ~ 2--no-edit//reverter os dous ult.commits

```

Al igual que en `reset-hard`, podemos usar `hard` y `hard` más comas o commits ID. – La opción `no-edit` no abre el editor por defecto para añadir nombres a los cambios.

```

Usuario@DESKTOP-VVH9EQM MINGW64 ~/Desktop/proy (master)
$ git log --oneline
b938ef3 (HEAD -> master) commit 4
6f220a7 commit 3
240c547 commit 2
7ca7425 commit 1

Usuario@DESKTOP-VVH9EQM MINGW64 ~/Desktop/proy (master)
$ git revert b938ef3...240c547
[master 99e2589] Revert "commit 4"
1 file changed, 1 insertion(+), 2 deletions(-)
[master 693e640] Revert "commit 3"
1 file changed, 1 insertion(+), 2 deletions(-)

Usuario@DESKTOP-VVH9EQM MINGW64 ~/Desktop/proy (master)
$ git log --oneline
693e640 (HEAD -> master) Revert "commit 3"
99e2589 Revert "commit 4"
b938ef3 commit 4
6f220a7 commit 3
240c547 commit 2
7ca7425 commit 1

```


Euben:

```
Usuario@DESKTOP-VVH9EQM MINGW64 ~/Desktop/proy (master)
$ git log --oneline
b938ef3 (HEAD -> master) commit 4
6f220a7 commit 3
240c547 commit 2
7ca7425 commit 1

Usuario@DESKTOP-VVH9EQM MINGW64 ~/Desktop/proy (master)
$ git revert HEAD...HEAD~2
[master d5cc34d] Revert "commit 4"
1 file changed, 1 insertion(+), 2 deletions(-)
[master 0680ac5] Revert "commit 3"
1 file changed, 1 insertion(+), 2 deletions(-)

Usuario@DESKTOP-VVH9EQM MINGW64 ~/Desktop/proy (master)
$ git log --oneline
0680ac5 (HEAD -> master) Revert "commit 3"
d5cc34d Revert "commit 4"
b938ef3 commit 4
6f220a7 commit 3
240c547 commit 2
7ca7425 commit 1
```

Si queremos revertir los compromisos intermedios, debemos tener cuidado de no entrar en conflicto.

IMPORTANTE: Con `reset --hard` indicamos el compromiso a alcanzar y con `revert` indicamos el compromiso (o múltiples compromisos) a eliminar.

Pago de pago

Para terminar este párrafo, echemos un vistazo a otra característica del comando `checkout`. Además de devolver un archivo desde la zona de commit a la zona del directorio de trabajo, y cambiar de una rama (o incluso crear una rama) como veremos más adelante, con `checkout` también podemos movernos a un "vistazo" antes de commit, ver el estado del repositorio en ese momento y luego volver a su estado normal.

Para "acceder" a un commit anterior:

```
git checkout 899001f
```

Nese momento, dirá que a cabeza está "detached", desacoplada. Y volviendo a la situación actual:

```
git checkout master

Usuario@DESKTOP-VVH9EQM MINGW64 ~/Desktop/proy (master)
$ git log --oneline
b938ef3 (HEAD -> master) commit 4
6f220a7 commit 3
240c547 commit 2
7ca7425 commit 1

Usuario@DESKTOP-VVH9EQM MINGW64 ~/Desktop/proy (master)
$ git checkout 240c547
Note: switching to '240c547'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by switching back to a branch.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -c with the switch command. Example:

    git switch -c <new-branch-name>

Or undo this operation with:

    git switch -

Turn off this advice by setting config variable advice.detachedHead to false

HEAD is now at 240c547 commit 2

Usuario@DESKTOP-VVH9EQM MINGW64 ~/Desktop/proy ((240c547...))
$ git checkout master
Previous HEAD position was 240c547 commit 2
Switched to branch 'master'
```

Misión 4.5. Operaciones básicas en Git

Instale git en Windows y tldiff y luego haga lo siguiente con los archivos html y css verificando el resultado de los cambios realizados en su navegador.

A) Crear una carpeta llamada myweb (en el repositorio git)

b) Incorporar el archivo index.html en esta carpeta:

```
< html >
  < cabeza >
    < link rel="stylesheet " type="text/css " href="style.css " >
  < /head >
  < cuerpo >
    < h1 > Hola mundo</h1 >
  < /body >
< /html >
```

Y un archivo style.css:

```
Cuerpo {color de fondo: rosa;}
h1 {color: azul;}
```

Compruebe el estado de git.

- c) Pasar el archivo a staged, comprobar el estado, pasarlo al repositorio local (puedes llamar a commit: "v1") y luego comprobar de nuevo el estado y los registros . Supongamos que ha terminado su web aquí y ya ha sido enviado al cliente.
- d) Ahora se le pide que realice algunos cambios: modifique un poco el archivo index.html y luego ejecute tldiff para ver la diferencia entre los archivos de su directorio de trabajo y los de su repositorio local (en comparación con los archivos que usted commit). Utilice estos nuevos cambios para crear un nuevo commit (puede llamarlo commit: "v2").
- e) Modificar el archivo style.css y el guardado. Ahora te arrepientes de los cambios y aún no has hecho ninguna adición o commit a esos cambios. ¿Cómo puedo restaurar style.css a la versión original en el repositorio local (en el área de confirmación)?
- f) Modificar el archivo style.css y guardado. Ahora usted se arrepiente de los cambios, pero usted ha hecho una adición pero no ha comprometido los cambios. ¿Cómo puedo restaurar style.css a la versión original en el repositorio local (en el área de confirmación)?
- g) Volver a hacer algunos cambios en index.html y style.css, esta vez si lleva esos cambios al repositorio y mira los registros. Al igual que los dos archivos añadidos anteriormente, solo realice una acción, un commit, y elija agregar juntos (puedes llamarlo commit: "v3")
- h) A tus clientes no les gusta el cambio, por lo que te piden que devuelvas la web a la situación anterior (toda la web, no lo hagas mirando qué archivos has modificado). Es decir, queremos devolver la misma situación que en commit "v2". Hay dos maneras de hacerlo: mantener el historial de commits, crear un nuevo commit para deshacer los cambios realizados desde el último commit, o eliminar completamente el commit "v3" y mantener el commit "v2". Queremos hacerlo de la primera manera, preservando la historia.
- i) Mostrar el registro de cuatro comisiones.
- j) Eliminar el archivo style.css (borrandoo out git, not:git rm) ¿Qué estado mantiene el repositorio ? Devuelve la situación anterior mediante la recuperación de archivos utilizando restaurar.
- k) Borra o archivo style.css (borrandoo con: git rm) En que estado queda o repositorio? Volver a la situación anterior. Pista: primero necesitas hacer una instrucción para que alcance el estado del punto anterior y luego recuperarlo.
- l) Ahora queremos volver al estado inicial, que es el estado del primer commit "v1 ", y el resto de commits que queremos hacer en el ejercicio desaparezcan como si hubiéramos empezado de nuevo. Mostra o log.

4.5 Trabajar con las sucursales

Una sucursal le permite trabajar con diferentes versiones del mismo conjunto de archivos en el mismo repositorio. Normalmente una rama emerge de una situación (por ejemplo, la rama maestra) y desde entonces tiene "vida propia" y evoluciona de forma independiente.

Los ejemplos típicos de ramificaciones son los que corresponden a las fases de desarrollo de nuevas funcionalidades, o parches que debemos desarrollar rápidamente en el código, separados de otras líneas de trabajo, como hemos visto en la sección "Buenas prácticas": "Git Flow", "GitHub Flow", etc.

Podemos "movernos" entre diferentes ramas sin necesidad de mover directorios. Al cambiar de ramas, git cambia el contenido del directorio de trabajo, pero mantiene todas las ramas actualizadas. Veremos cómo se pueden fusionar diferentes "ramas" mediante la resolución de conflictos.

El comando para crear una nueva rama es:

Rama

```
git branch novarama (copia da rama real)
git branch novarama MASTER (maestro o copia de la rama maestra)
git branch novarama branch (copia de la rama maestra)
```

Para cambiar de una rama a otra hacemos: git checkout **rama**

Ahora podemos trabajar en la nueva rama, todos los cambios solo afectarán a esa rama, mientras que las otras ramificaciones permanecen igual.

Si vamos a crear una rama y movernos a ella, en lugar de ejecutar los dos comandos anteriores

Podemos usar **Check-b** Y creado con el contenido de la rama en la que estábamos en ese momento.

```
git checkout -b novarama
```

Depende del conjunto de ramas que tenemos en el proyecto: **Rama**ámetros.

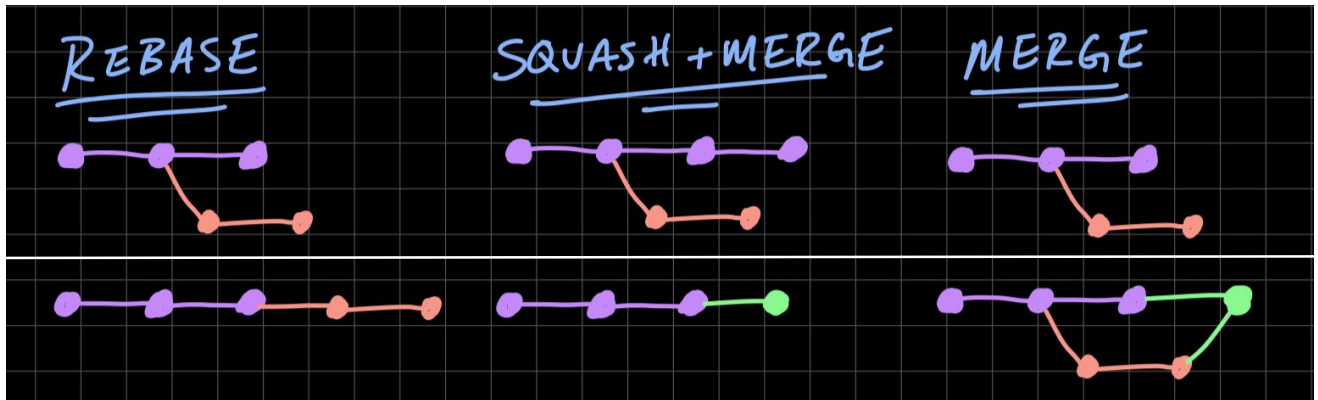
```
git branch (con la opción -a para mostrar los ocultos, con -v para informar el último commit)
```

Para hacer una fusión de ramas, nos ponemos en la rama de destino **Fusión** git merge branch Este es uno de los pasos más importantes y críticos, ya que los cambios realizados en la rama especificada se fusionarán en la rama actual, dejando solo el código único. La fusión se realiza línea por línea y puede producirse un conflicto si dos ramas diferentes hacen cambios en la misma línea, como veremos en la siguiente sección.

fast forward es el término que se aplica a la fusión cuando la sucursal original está totalmente actualizada en todos los aspectos. Es decir, no necesitamos ninguno de los ramales ahora, captando el cambio en el origen de los ramales tenemos la situación final que estamos buscando. En este caso, git es inteligente y, en lugar de hacer una fusión "normal" fusionando las líneas de una rama y otra, hace que HEAD apunte al estado más reciente de la rama fuente, en lugar de hacer cambios y commit en la rama actual. Dada esta situación, desde que creamos la sucursal original hasta la fusión, no debería haber cambios en la sucursal actual.

Hay otras dos formas de realizar la fusión de ramificaciones, use el comando para **Rebase** o con **Mezcla-calabaza**. El resultado final es el mismo, una fusión cambiada, pero la diferencia es que en rebase, los commits de la rama fuente se aplican a la rama de destino (como si fueran originalmente cometidos en la rama de destino), en el caso de squash, los commits de esa rama se pierden, por lo que el último commit se fusiona.

Podemos decir que merge conserva todo el historial, rebase "aplana" el gráfico de cambios, mientras que squash deja un gráfico más limpio. En el siguiente ejemplo, podemos tener la rama "main" en púrpura y la rama "branch" en naranja. Hemos visto resultados en tres casos de fusión.



<https://matt-rickard.com/squash-merge-or-rebase/>

Para cerrar el tema de las ramas, veremos cómo eliminaremos las ramas (por ejemplo Una vez fusionado con otros contenidos, ya no lo necesitamos). O hacemos co comando:

Branch-d

Git Branch-d Rama

Misión 4.6. Gestión de sucursales

La tarea es hacer lo siguiente.

- A) Crear un repositorio similar a la tarea 4.5 y commit ambos archivos (el primero se puede usar eliminando la carpeta. git)
- b) Crear una rama llamada "novaWeb" y cambiar a ella.

Ha tenido que realizar algunos commits antes desde que se creó la rama desde el área de commits.

- c) Eliminar esta nueva rama style.css, crear style2.css y modificar index.html con el nuevo archivo css y mostrar el estado

```
< html >
  < cabeza >
    < link rel="stylesheet " type="text/css " href="style2.css"></head>

    < cuerpo >
      < h1>Nova Web</h1></

< /html >
```

Y un archivo style2.css:

```
Cuerpo {color de fondo: negro;} h1 {color:
```

- d) Asegúrese de que estás en la sucursal novaWeb y commit los cambios.
- e) Mostrar el contenido de la página web de esta nueva rama en el navegador. Cambiar el maestro y mostrar el contenido web en el navegador.

- f) Mostrar el registro con las opciones: oneline, all y graph (verá un asterisco que muestra el último commit de cada rama).
- g) Fusionar la rama creada con la rama maestra y mostrar la rama maestra en el navegador. gi
- h) Mostrar el estado y los registros.
- i) Eliminar la rama creada.

4.6 Resolución de errores y conflictos

Ver las diferencias

Hemos visto que el comando diff es una buena herramienta para comprobar las diferencias entre los archivos ubicados en diferentes áreas del repositorio (directorio de trabajo, staged, commit), pero también se puede utilizar para comprobar las diferencias entre los archivos en diferentes momentos (en diferentes commits). Por ejemplo, compare la situación actual con los dos commits de fai:

```
Git diff cabeza ~ 2
```

Conflictos en los cambios de sucursal

Como vimos en la sección anterior, la "fusión" fusiona ramas juntas (representadas en la rama en la que estamos), pero podemos encontrar conflictos, es decir, hacer cambios al mismo código en diferentes ramas al mismo tiempo.

Situación 1:

1. Creamos una rama de MASTER en la que modificamos un archivo en una línea
2. Creamos otra rama desde MASTER, en la que modificamos el mismo archivo, pero en otra línea
3. Hacemos la fusión de ramas del punto 1 en MASTER.
4. Fusionamos las ramas del punto 2 en MASTER

¿Qué sucederá?

En este caso, no habrá problema. Lo que hace Git es fusionar ramas en línea y fuera de línea.

Situación 2:

1. Creamos una rama de MASTER en la que modificamos un archivo en una línea
2. Creamos otra rama desde MASTER, en la que modificamos el mismo archivo en la misma línea.
3. Hacemos la fusión de ramas del punto 1 en MASTER.
4. Fusionamos las ramas del punto 2 en MASTER

¿Qué sucederá?

- No hay problema con el paso 3 en este caso, ya que todavía no hay conflictos.
- En el paso 4 se encuentra que el MASTER actual (después del paso 3) es diferente al que dejaste, y este es un conflicto que git no sabe cómo resolver.

Soluciones:

- La solución es modificar la relación de líneas en el archivo con lo que consideramos correcto (la primera Rama, la segunda o lo que queramos) y envíe ese documento. Podemos ver la diferencia con git diff rama1 rama2 (o git diff tool)
- Si hay demasiados conflictos y no se pueden resolver fácilmente modificando los archivos involucrados, podemos deshacer la última fusión: git merge --abort

Misión 4.7. Conflictos entre ramas

- A) Crear el mismo repositorio que en la tarea 4.5 con los archivos index.html y style.css. (Puedes usar el método anterior eliminando la carpeta. git)
- b) Crear una rama llamada cambios01 desde la rama maestra. En esta nueva rama, cree un archivo style2.css similar a la tarea 4.6, elimine style.css y deje que index.html use la nueva hoja de estilos.
- c) Crear una rama llamada cambios02 desde la rama principal (¡Nota! No comienza desde cambios01) y cambiar el contenido de la etiqueta < h1 >.
- d) Mostrar los registros con las opciones all y oneline
- e) Mostrar el contenido de la página web en el navegador en tres casos, es decir, cambiar de MASTER a Cambio01E a Cambio02
- f) Fusión de MASTER y cambios01.
- g) Fusión de MASTER y cambios02. ¿Hay conflictos? Explique por qué. En caso afirmativo, compruebe las líneas en conflicto y ajuste la situación.
- h) Eliminar las ramas que ya no son necesarias.

Misión 4.8. Muchos conflictos entre sucursales

- A) Crear el mismo repositorio que en la tarea 4.5 (con index.html y style.css)
- b) Crear una rama llamada cambios01 desde la rama maestra. En esta nueva rama, cree un archivo style2.css similar a la tarea 4.6, elimine style.css y deje que index.html use la nueva hoja de estilos. Modificar el contenido de la etiqueta < h1 >. Añadir/enviar cambios.
- c) Crear una rama llamada cambios02 desde la rama maestra (¡nota! No comienza con user01) y cambiar el contenido de la etiqueta < h1 >.
- d) Mostrar los registros con las opciones all y oneline
- e) Visualizar en el navegador el contenido de la página web en tres situaciones, es decir, cambiar de MASTER a Cambios01 y Cambios02.
- f) Fusión de MASTER y cambios01.
- g) Fusión de MASTER y cambios02. ¿Hay conflictos? Explique por qué. En caso afirmativo, compruebe las líneas en conflicto y ajuste la situación.
- h) Eliminar las ramas que ya no son necesarias.

4.7 Repositorios remotos

Los repositorios remotos son versiones de su proyecto que están alojadas en internet o en la intranet de la empresa. Usted puede tener varios, algunos de los cuales usted será el propietario, otros usted será el colaborador, muchos de los cuales tendrá acceso a la lectura.

La forma de trabajar con ellos, a nivel general, es llevar sus datos localmente (pull), modificarlos y cargar (push) los cambios.

Tenemos dos herramientas bien conocidas para gestionar nuestras versiones de código, en Remota y compartida con otros usuarios o con el público en general: GitHub y GitLab. Ambos tienen algunas características en común.

- Tienen una interfaz web y actúan como una plataforma social para compartir conocimientos y trabajo.
- Podemos tener un número ilimitado de repositorios públicos (vista, No Commit), pero también pueden ser privados.
- Tener cuenta corporativa (pagada).
- Permite ramas e comunicarnos cos usuarios (pull request).

La mayor diferencia es que podemos instalar GitLab en nuestro servidor, mientras que el primero siempre debe ser utilizado a través del sitio web github.com. Por otro lado, en la versión web gratuita, GitLab ofrece mejores condiciones de espacio y cantidad de contribuyentes por proyecto.


Conexión remota

Para conectarnos a un repositorio, primero crearemos un repositorio local usando git init y luego crearemos un repositorio vacío en GitHub/GitLab:


Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)

Owner

 fernando

Repository name *


/ myweb 

Great repository names are short and memorable. Need inspiration? How about **probable-barnacle**?

Description (optional)

Probas

☐  **Public**
Anyone can see this repository. You choose who can commit.

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

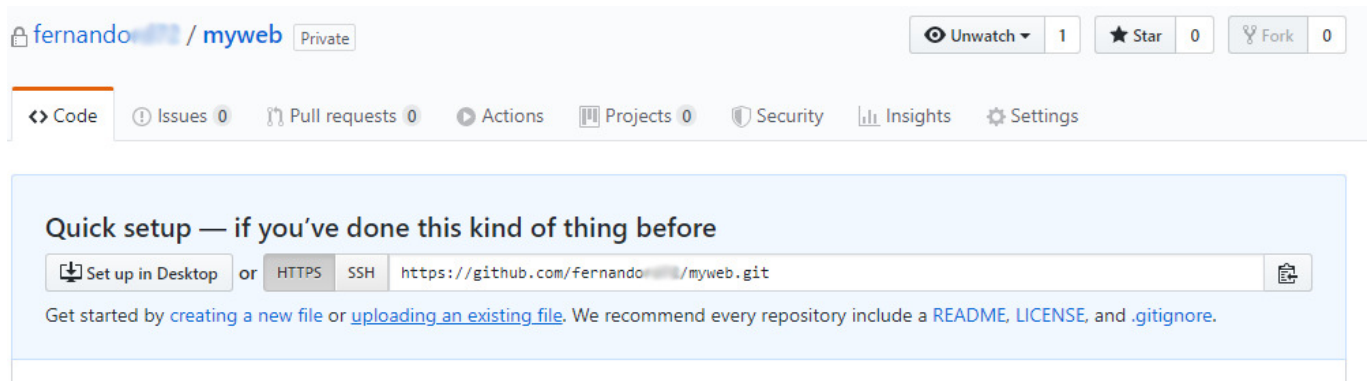
Skip this step if you're importing an existing repository.

☐ **Initialize this repository with a README**
This will let you immediately clone the repository to your computer.

Add .gitignore: **None** | Add a license: **None** 

Create repository

Obtendo a URL de conexión:



Luego lo conectamos remotamente a **Add remote origen url** :

```
Git remote agrega origen https://github.com/fernandoxxx/myweb.git
```

"Origen" no es una palabra reservada, puedes poner cualquier nombre, pero es un estándar llamarlo así en el repositorio principal. De hecho, si tuviéramos varios repositorios propios, tendrían nombres diferentes. Por supuesto, los repositorios con los que trabajamos también le asignarán otros nombres.

Para ver el control remoto que hemos configurado, simplemente lo usamos, a través de **Remoto** opciones **-v** para obtener más información.

```
git remote -v
```

A partir de entonces, en los comandos usaremos este nombre en lugar de la url real.

Para profundizar en la información: o mostrar nombre de origen remoto **Mostrar origen remoto**

Contraseñas en Github: Github ha cambiado su política de seguridad para el acceso remoto desde la consola o aplicaciones como Netbeans y ya no puede utilizar contraseñas como lo hacemos cuando accedemos a un repositorio desde una página web de Github.

Para la consola (y otras aplicaciones), debemos crear un token de acceso personal en la página de Github y luego usar ese token en lugar de una contraseña. También necesitamos comprobar nuestra dirección de correo electrónico en la página de Github antes de crear el token. En los siguientes enlaces obtendrás información detallada sobre cómo hacer ambos pasos.

<https://docs.github.com/en/get-started/signing-up-for-github/verifying-your-email-address>

<https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/creation-a-personal-access-token>

Es importante destacar que el token tiene un amplio tiempo de vencimiento a lo largo del proceso, lo que incluye la operación de "recompra" y guardarlo en un archivo, ya que luego no podremos acceder a él en Github.

"Cargar" información a un repositorio remoto

Lo contrario es "cargar" los cambios realizados localmente a un repositorio remoto (vamos a hacer los cambios utilizando el proceso que ya vimos: modificar archivo + agregar + commit). El comando para cargar al repositorio remoto es **git push** :

```
git push origin nomeRama
```

origin es el nombre de la conexión remota a la que nos referimos, y en el nombre de la rama pondremos la rama a la que queremos conectar, muchas veces será master: git push origin master

Push tiene la opción **(-u)** (significa upstream) que configura estas ramas locales y remotas como las ramas por defecto, así las veces siguientes, se podría hacer solo git push sin ningún parámetro más:

```
git commit -m version1
Git remoto agrega origen https://github.com/user1/repo1.git
Git push origin maestro
git commit-am versión 2
Git empuje
```

"Descargar" información de un repositorio remoto

El siguiente paso es "ver" la estructura del repositorio remoto, es decir, las diferentes "ramas". Veremos más adelante que las ramas son diferentes versiones del programa (una en producción, una en el desarrollo de nuevas características, la otra puede estar parchando algunos errores, etc.). Siempre habrá una rama principal, la que está siendo producida, llamada maestra. Vamos a descargar con **Obtener origen** ¿Y qué?

```
git fetch origin
```

Una vez hecho esto, hemos descargado el archivo, pero en una rama local oculta (llamada origin/master) Ahora necesitamos pasar el contenido descargado a la rama master en nuestro repositorio local. Iso Vamos a utilizar **Fusión de origen/maestro:**

```
Git merge origine/maestro
```

Debido a que las dos operaciones generalmente se hacen juntas, tenemos un comando para hacer ambas operaciones sucesivamente. Él es el director y es más útil que los dos primeros.

```
Git pull original maestro
```

Para traernos el contenido de una rama remota: git pull origin rama1 pero debemos estar situados en local en la rama con el mismo nombre. Si en local estuviésemos en otra rama, como por ejemplo master, al hacer el pull haría un merge entre la rama1 remota y mi rama local.

También tenemos un comando git que ejecuta git init y pull al mismo tiempo, y se llama clone.

```
git clone https://github.com/nomerepositorio.
```

Punto indica copiar a la carpeta actual

También tenemos una opción en checkout para trabajar con sucursales remotas:

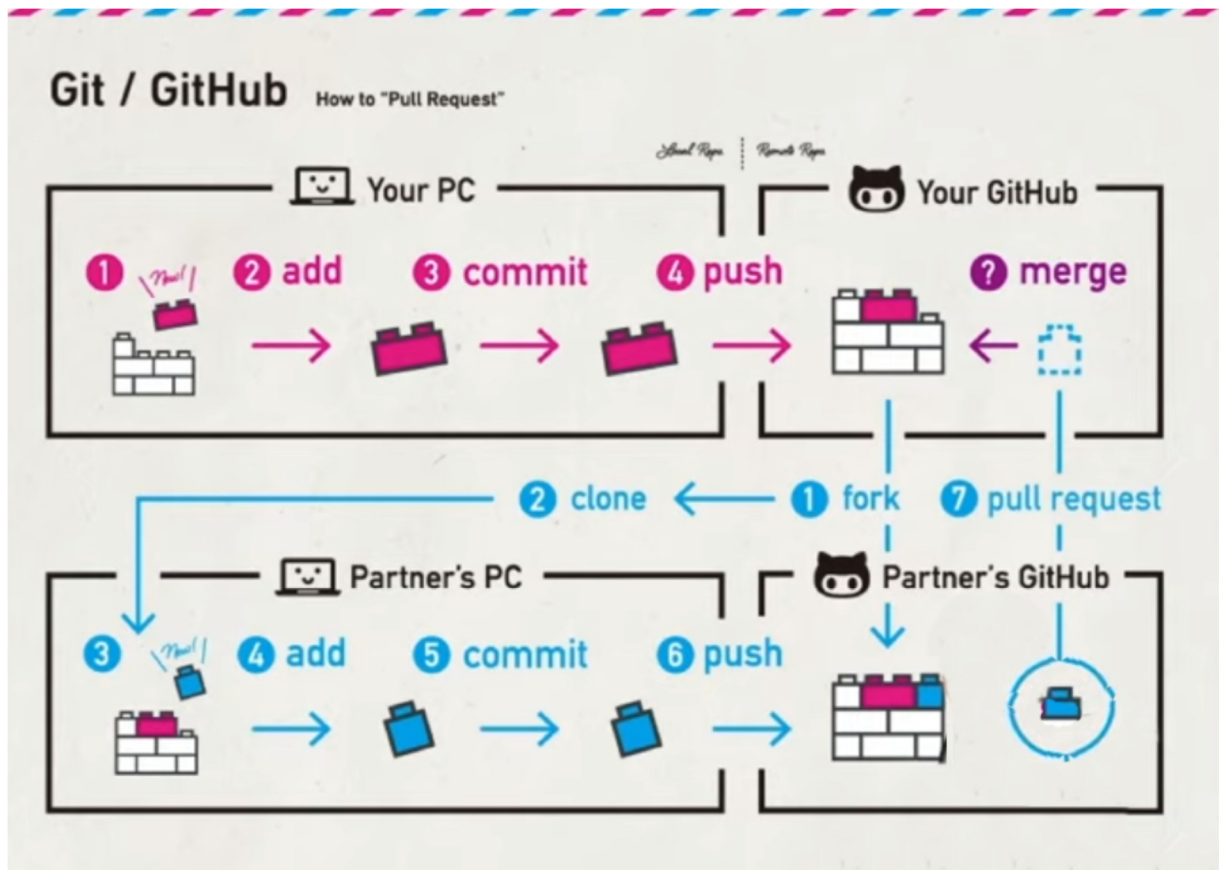
Check-t origen/

Numeración: Si existe una rama remota llamada "nomeRama", al ejecutar este comando

Cree una sucursal local con el mismo nombre para rastrear la sucursal remota con el mismo nombre.

Petición de tirar

Es una solicitud que hacemos para modificar un proyecto en GitHub que no es nuestro, por ejemplo para contribuir a un proyecto de software libre. El propietario es quien hará los cambios, por lo que nuestro trabajo será conseguir una copia del proyecto (fork), llevarlo a local (clone), modificarlo (commit), subirlo a nuestra cuenta (push) y hacer una solicitud de integración (pull request), y el propietario se encargará de realizar la fusión (merge).



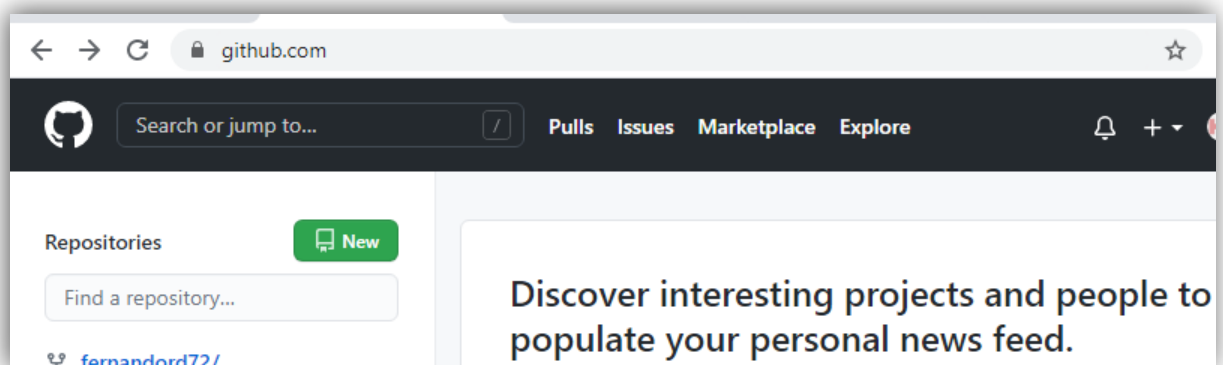
Este proceso será realizado entonces por dos usuarios diferentes, uno es el propietario del repositorio original (en este caso usaremos fernandord 72) y el otro usuario (en este caso wirtzPruebas), que hará una copia del repositorio original, lo modificará y le pedirá al usuario inicial que integre los cambios propuestos.

A continuación dividimos este proceso en tres etapas:

1. O usuario fernandord72 crear un repositorio en Github.
2. wirtzpruebas El usuario crea una copia (fork) del repositorio anterior, modifica la copia y hace una solicitud de pull al usuario inicial.
3. Fernandord72 vuelve a aceptar los cambios (fusiones) propuestos por wirtzPruebas.

Pleg request: Crear o repositorio en GitHub

Comenzamos con un repositorio público en GitHub con la cuenta fernandord72. Podemos crearlo desde la propia herramienta haciendo clic en el botón [nuevo]



Adquirir:

Muéstranos la URL de la conexión:

Desde nuestro ordenador podemos subir archivos a este repositorio mediante los procedimientos habituales: add, commit, push....

```
usuario-1@DESKTOP-CF3G654 MINGW64 ~/Desktop/repo1 (master)
$ git status
On branch master

No commits yet

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

nothing added to commit but untracked files present (use "git add" to track)

usuario-1@DESKTOP-CF3G654 MINGW64 ~/Desktop/repo1 (master)
$ git add .

usuario-1@DESKTOP-CF3G654 MINGW64 ~/Desktop/repo1 (master)
$ git commit -m "primeira versão"
[master (root-commit) 049d2de] primeira versão
1 file changed, 1 insertion(+)
create mode 100644 archivo.txt

usuario-1@DESKTOP-CF3G654 MINGW64 ~/Desktop/repo1 (master)
$ git remote add origin https://github.com/fernandord72/repo1.git

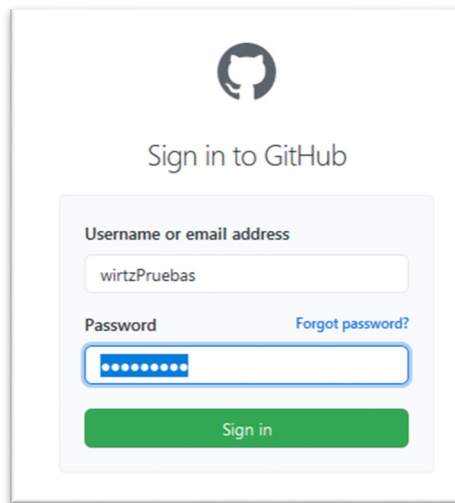
usuario-1@DESKTOP-CF3G654 MINGW64 ~/Desktop/repo1 (master)
$ git push origin master
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Writing objects: 100% (3/3), 232 bytes | 77.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/fernandord72/repo1.git
 * [new branch]      master -> master

usuario-1@DESKTOP-CF3G654 MINGW64 ~/Desktop/repo1 (master)
$ |
```

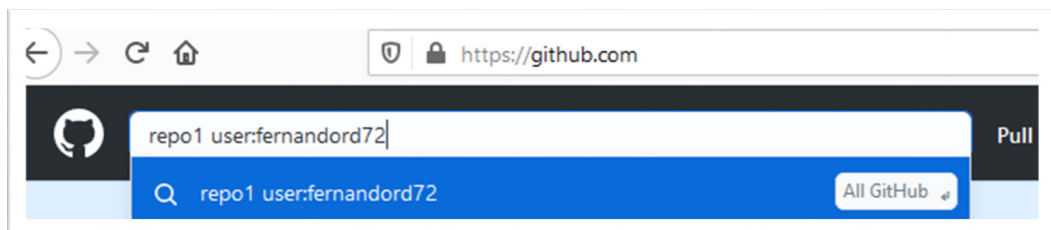
Ahora otro usuario puede hacer una "solicitud de pull" o una solicitud de modificación desde el repositorio, todo desde el propio sitio web de GitHub.

Facendo una solicitud de tiro

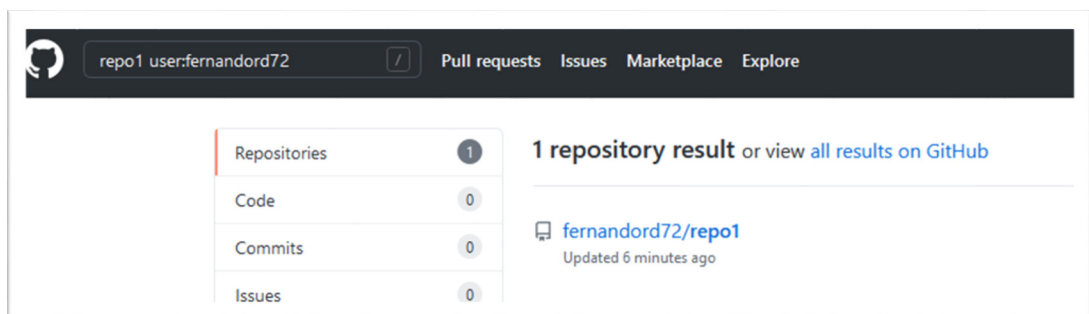
Primero otro usuario (wirtzPruebas) inicia sesión:



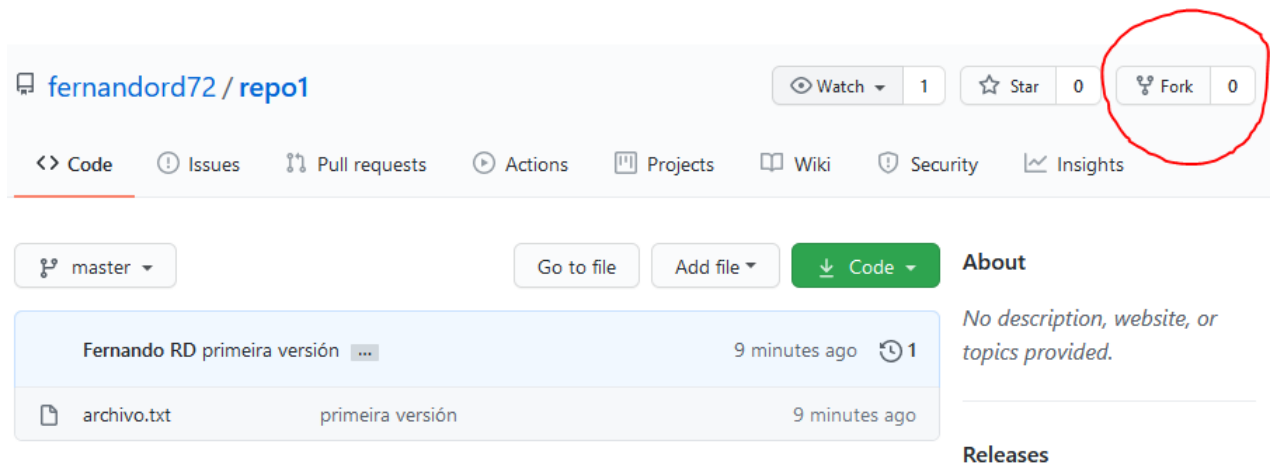
Busque el repositorio (repo1 de fernandord72) en el que desea realizar una "petición de pull":



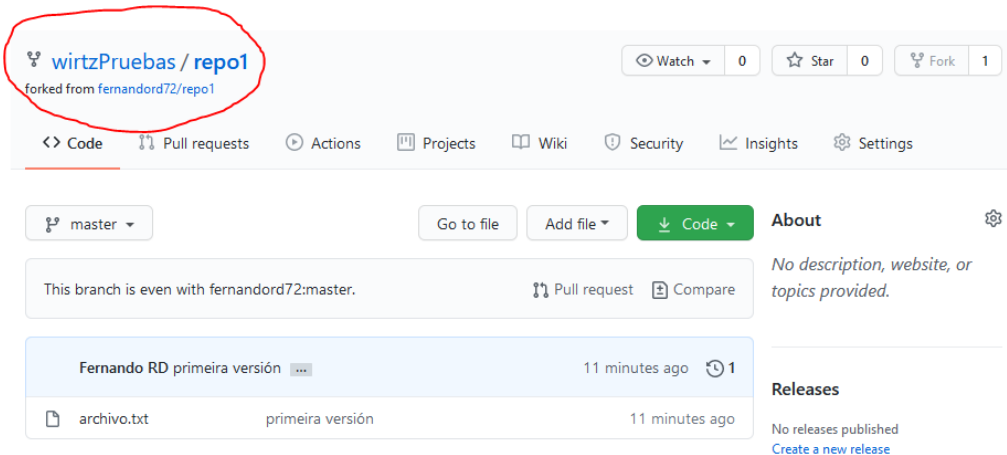
No importa cuán público sea, no hay problema en buscarlo.



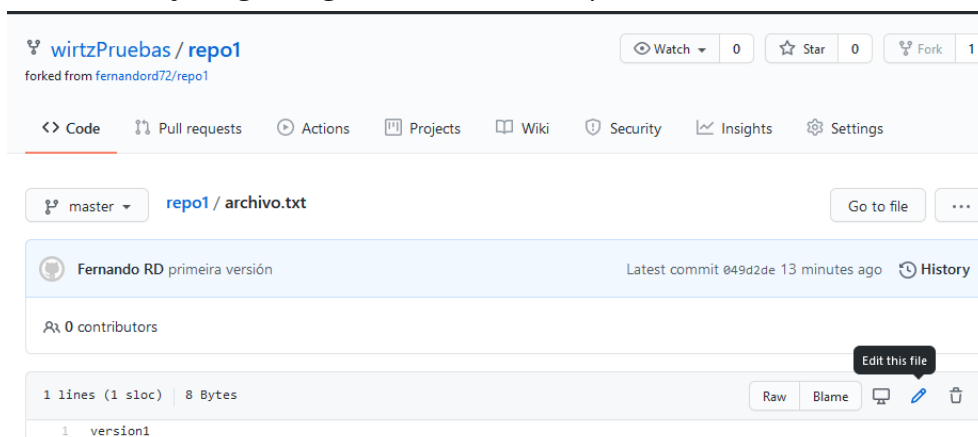
Seleccionarlo y haga clic en el botón [Fork] para crear una copia del repositorio original en su cuenta y modificarlo a voluntad sin afectar a otros repositorios.



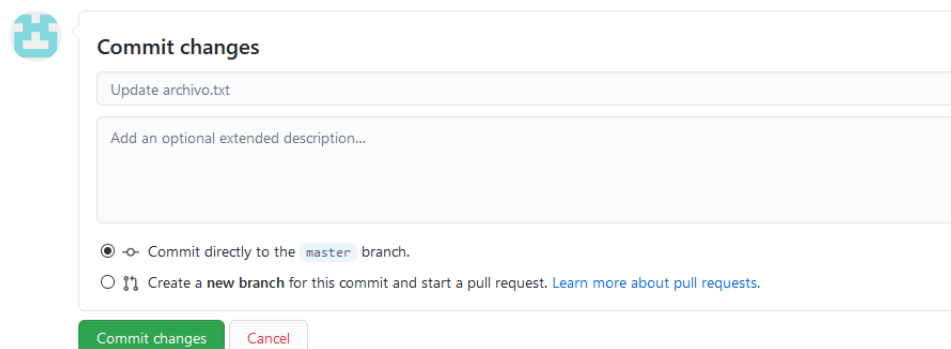
Obtener un nuevo repositorio:



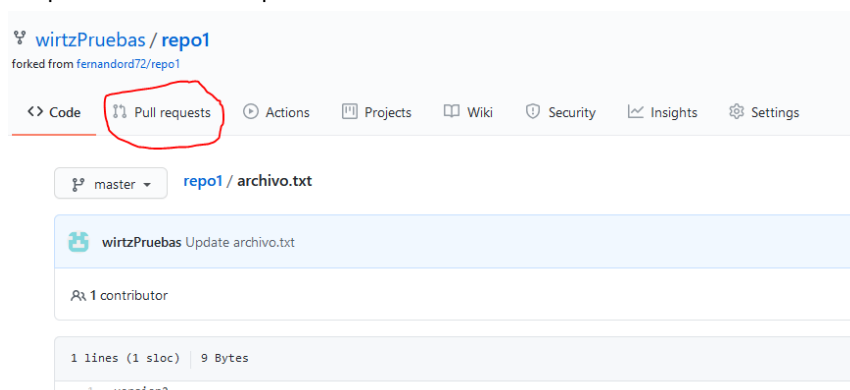
Al hacer clic en el archivo y luego en el botón de editar, podemos modificarlo inmediatamente y para hacerlo rápidamente, podemos descargarlo localmente, modificarlo y luego cargarlo como en el repositorio inicial.



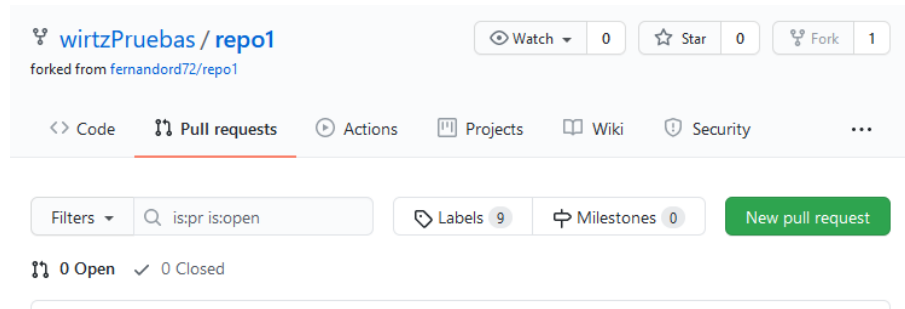
Hacemos las modificaciones y guardamos los cambios haciendo clic en [Enviar cambios] en la parte inferior de la pantalla.



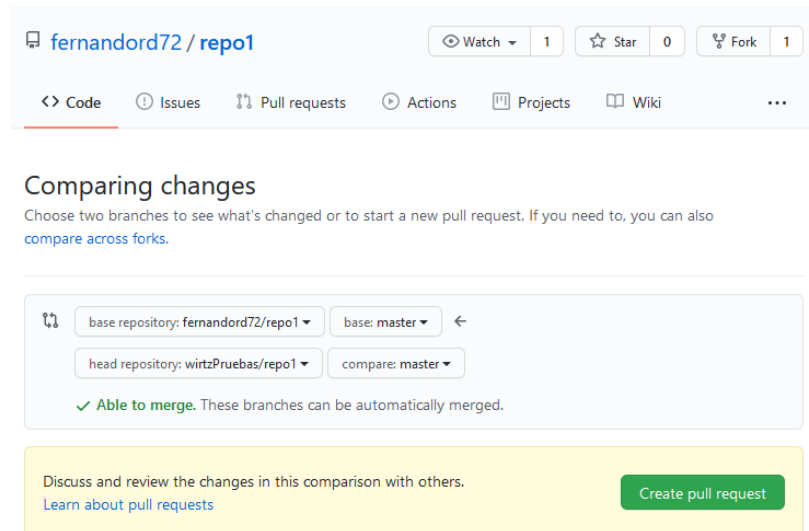
Ahora puedes completar la solicitud pull haciendo clic en el botón con ese texto:



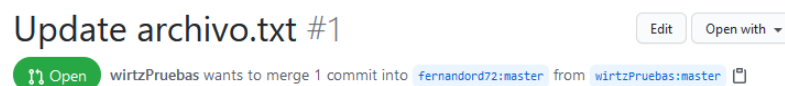
Ventana de llegada:



Hagamos clic en el botón verde [Nueva solicitud pull] para obtener un resumen de cómo se desarrollará la integración:



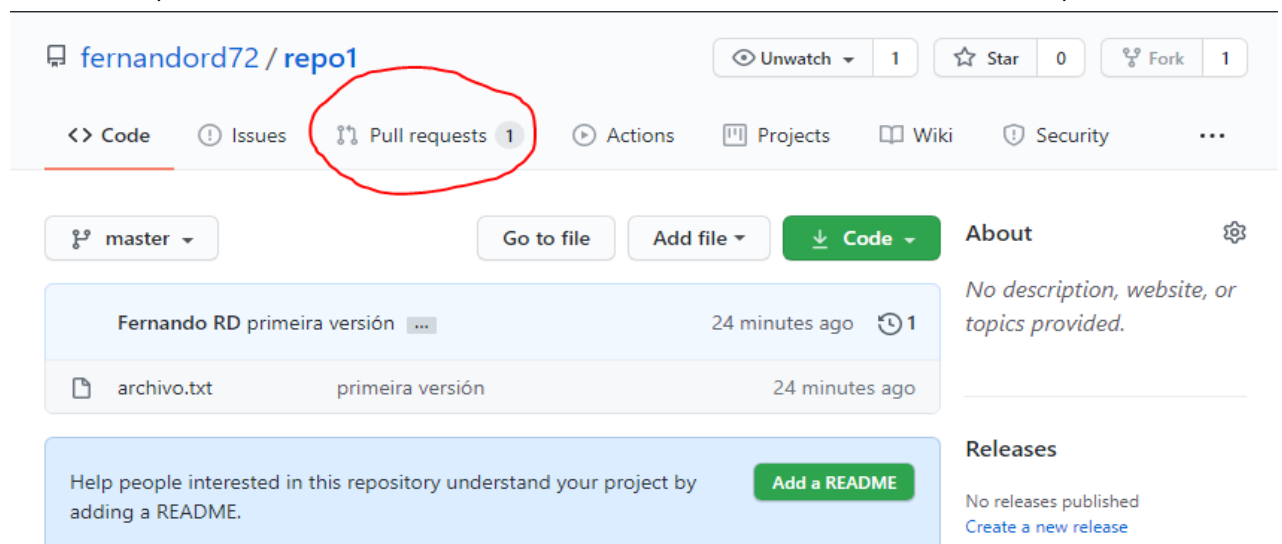
Confirmamos pulsando [Crear Pull Request]



Aquí termina el trabajo de que el usuario emite una solicitud de pull (wirtzPruebas).

Aceptar una solicitud de tiro (fusión)

Cuando el usuario que es propietario del repositorio (fernandord72) inicie sesión, verá que tiene una solicitud de pull (también recibirá un correo electrónico notificándola de la solicitud de pull):



Podemos hacer clic en el archivo para ver los cambios antes de aceptarlos:

Update archivo.txt #1

Open wirtzPruebas wants to merge 1 commit into `fernandord72:master` from `wirtzPruebas:master`

Conversation **0** Commits **1** Checks **0** Files changed **1**

Changes from all commits File filter... Jump to...

Update archivo.txt

This commit does not belong to any branch on this repository.

wirtzPruebas committed 9 minutes ago **Verified**

▼ 2 archivo.txt

...	...	@@ -1 +1 @@
1	-	version1
1	+	version2

Finalmente, regresa a la página anterior y haga clic en [Combine Pull Request] para integrar los cambios de otro usuario en este repositorio:

Update archivo.txt #1

Open wirtzPruebas wants to merge 1 commit into `fernandord72:master` from `wirtzPruebas:master`

Conversation **0** Commits **1** Checks **0** Files changed **1**

wirtzPruebas commented 4 minutes ago **First-time contributor**

No description provided.

Update archivo.txt **Verified** ee9dce8

Add more commits by pushing to the `master` branch on `wirtzPruebas/repo1`.

Continuous integration has not been set up
GitHub Actions and several other apps can be used to automatically catch bugs and enforce style.

This branch has no conflicts with the base branch
Merging can be performed automatically.

Merge pull request ▼ You can also [open this in GitHub Desktop](#) or view [command line instructions](#).

Conserva los cambios incorporados como nuevos commits:

fernandord72 / repo1 Unwatch ▼ **1** Star **0** Fork **1**

Code Issues Pull requests Actions Projects Wiki Security

History for `repo1 / archivo.txt`

Commits on Mar 4, 2021

Update archivo.txt **Verified** ee9dce8

wirtzPruebas committed 23 minutes ago

primeira versão 049d2de

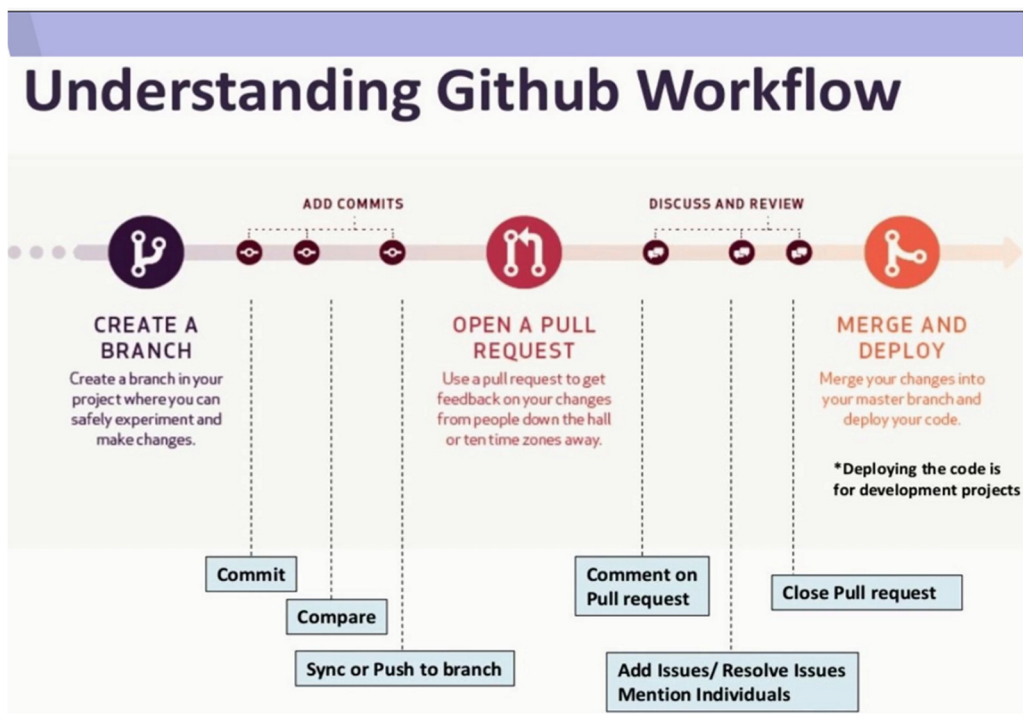
Fernando RD committed 40 minutes ago

El usuario que haga una solicitud de pull también recibirá un correo electrónico notificándolo de la fusión.

Misión 4.9. Operaciones de GitHub

Crea una cuenta en GitHub y:

- A) Crear un repositorio localmente, como en la tarea 4.5
- b) Crear un repositorio vacío en GitHub.
- c) "Sube" o repositorio a GitHub
- d) Eliminar el repositorio local y descargarlo con un clon.
- e) Crear una rama localmente con cambios similares a la tarea 4.6
- f) Nueva rama "ascendente"
- g) En Github, no local: comparar la nueva rama con MASTER y luego fusionar. Esto se hace en el botón "Comparar y Pull Solicitud" en la nueva rama.
- h) Desde la consola git local, descargue la rama Master (pull) al local
- i) Haga una solicitud de pull en el repositorio de GitHub de algún socio" (agregue algunos < p > párrafos al archivo index.html.
- j) Compruebe y acepte las "solicitudes de pull" que le hagan sus compañeros de trabajo.



4.8 Otros comandos

Culpa

Cuando trabajamos en equipo, puede ser interesante saber que has hecho algunos cambios específicos, como acordar coordinar nuevos cambios (o poner la responsabilidad en ti si estás equivocado...). Para ello, tenemos el mando. **Culpa**

```
Git culpa nomearquivo
```

Podemos indicar quién modificó un archivo, pero en líneas específicas.

```
Git black -L 12,14 src/Main.java
```


Renombrar un repositorio remoto

Para cambiar el nombre que le damos al repositorio cuando lo asignamos a "Agregar fuente", usaremos **Renombramiento remoto** `nomeRepo` y borrarlo (por ejemplo, si ya no participa en el proyecto) `nomeRepo RM remoto`

Etiquetas

Una etiqueta o un **Etiquetas** nombre que podemos asignar a una rama específica (es decir, un commit específico) en un momento específico. Podemos usarlo para nombrar versiones de software, o marcar "logros".

```
Git tag-a v1-m'Release v0.1'
```

Luego es fácil volver a esas versiones (muy útil en proyectos ágiles, como marcar el final de cada sprint semanal). El comando `checkout` se utiliza cuando cambiamos de una rama a otra.

```
git checkout v1
```

Escondido

Si estamos trabajando en el código y tenemos que cambiar otra rama por alguna razón, pero no queremos commit nuestro trabajo porque está en estado parcial, podemos realizar `stashing` como el comando `stash`, es decir, "congelar" los **Escondido** cambios que hemos hecho desde el último commit y almacenarlos temporalmente, dejando el directorio de trabajo limpio y sin ningún commit pendiente.

```
Git escondido
```

Ahora podríamos pasar a la nueva tarea, y cuando podríamos reanudar el trabajo anterior:

```
Lista de guardar git
```

Se muestra una lista de "stashes" y luego selecciona la que queremos:

```
git stash aplica stash@{X}, X es
```

el número que se muestra en la lista.

Rebase

Ya hemos visto este comando para la fusión de ramas, que también permite "fusionar" múltiples compromisos de historial en un solo SON. Imagínense que en un gran proyecto hicimos 100 compromisos el año pasado y 100 más este año. Podemos estar interesados, limpiando, poniendo juntos las promesas del año pasado, por qué ya no estamos interesados, separándolas, una sola voz.

```
Git rebase--interactive--root
```

La opción interactiva nos permite seleccionar el rango de commit "manualmente" a través de una pantalla, mientras que la opción `root` significa que se muestra desde el principio.

Esta pérdida de historial (pérdida de compromisos) puede ser peligrosa porque nunca volveremos a una situación en la que los compromisos fueron eliminados.

Ejemplo ejercicio de examen Git (con solución)

Supón que estás en una carpeta con un solo archivo llamado datos.txt e inicias una consola de comandos git. Se pide lo siguiente:

- Inicia un repositorio git.
- Pasa el archivo a la zona staged
- Haz un commit con el nombre v1
- Supón que has modificado el archivo, haz un nuevo commit con nombre v2 ¿puedes pasarlo a staged y hacer commit con un solo comando?
- Supón que has modificado de nuevo el archivo, pero te arrepientes de los cambios (no has ejecutado ningún comando git sobre él) ¿Cómo recuperas la versión del archivo que está en tu repositorio local?
- Supón que has modificado de nuevo el archivo, pero te arrepientes de los cambios (ya lo has pasado a la zona staged) ¿Cómo recuperas la versión del archivo del repositorio local?
- Por ahora tienes dos commits, el de nombre v1 y el de nombre v2. ¿Como volverías todo el repositorio a la situación v1?
Nota: en este caso es un archivo por simplificar, pero podrían ser muchos. Hay 2 formas de hacerlo, conservando el histórico de commits, creando un nuevo commit que deshace los cambios del último, o bien eliminado totalmente el commit "v2" e quedando solo el commit "v1". Queremos hacerlo de la primera forma, conservando el histórico.
- Supón que borramos el archivo datos.txt por fuera de git. Vuelve a la situación anterior recuperando el archivo.
- Supón que borramos el archivo datos.txt con git rm. Vuelve a la situación anterior recuperando el archivo.
- Ahora queremos volver a la situación del primer commit "v1", queremos que desaparezcan el resto commits hechos previamente, como si volviésemos al final del punto c)
- Crea una rama llamada 'r1' a partir de la situación actual.
- Moverse a "r1"
- Supón que modificas el archivo datos.txt. Realiza los pasos necesarios para fusionarla en la rama principal ¿En qué casos habría conflictos?
- Borra la rama 'r1'
- ¿Cómo configuras un repositorio remoto por ejemplo en Github?
- ¿Qué comando git emplearías para subir el repositorio local al remoto?

Solución

- git init
- git add datos.txt, también git add .
- git commit -m v1
- git commit -a -m v2 (puedo hacerlo en un solo comando porque ya hice un add previamente de ese archivo)
- git checkout datos.txt
- Git Restablecer Datos.txt y luego: git checkout datos.txt
- git revert HEAD o también git revert idUltimoCommit
- git restore datos.txt
- Git restore--staged datos.txt y luego: git restore datos.txt
- git reset --hard HEAD ~ 1 o también: git reset --hard idPrimerCommit
- git branch r1
- git checkout r1
- Git agregar datos.txt
git commit -m cambiosrama
git checkout master
Git fusión r1
habría conflictos si se hubiese modificado en master y en esta rama las mismas líneas del mismo archivo
- git branch -d r1
 - git remote agrega origen https://github.com/usuario/repositorio.git
- git push origin maestro

4.9 Control de versiones en NetBeans con Git

NetBeans tiene un plugin que se puede utilizar como cliente de Git. Os pasos a seguir para que NetBeans funcione como cliente de Git son:

Inicialización del repositorio

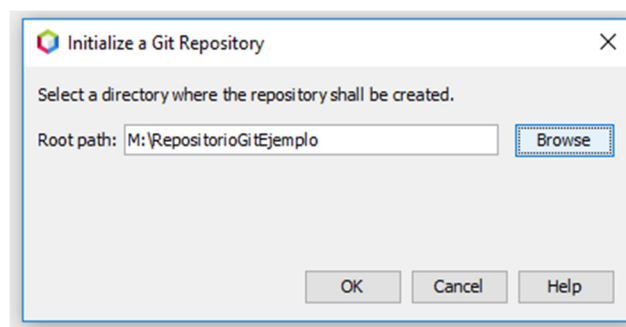
-Posibilidad de crear repositorios a partir de proyectos existentes. Haga clic derecho en el proyecto > Control de versiones > Inicializar el repositorio Git...

-Creación de repositorios sin proyectos existentes, implementación de importaciones. Menú superior Team > Git > Inicialice Repository...

A partir de ahora, las carpetas de proyectos podrán mostrar insignias de colores y los archivos también serán nombrados con diferentes colores. Estos colores cambian según el estado del archivo a medida que realizamos modificaciones.

Ubicación del repositorio.

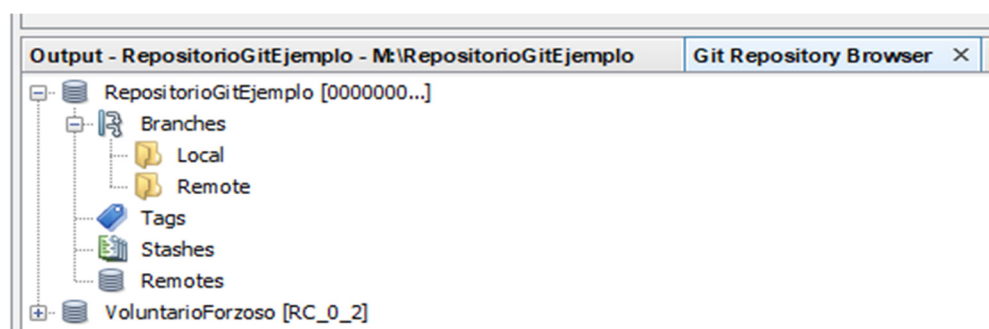
Solo puede haber un repositorio dentro de una carpeta.



En la salida de NetBeans podemos ver los comandos utilizados para crear un repositorio local, así como la ubicación.

```
Output - RepositorioGitEjemplo - M:\RepositorioGitEjemplo  X Git Repository Browser
==[IDE]== 3 abr. 2019 8:06:03 Initializing ...
Initializing repository
Creating git M:\RepositorioGitEjemplo/.git directory
git init M:\RepositorioGitEjemplo
==[IDE]== 3 abr. 2019 8:06:03 Initializing ... finished.
|
```

En la salida del repositorio podemos ver la estructura básica para crear el proyecto.

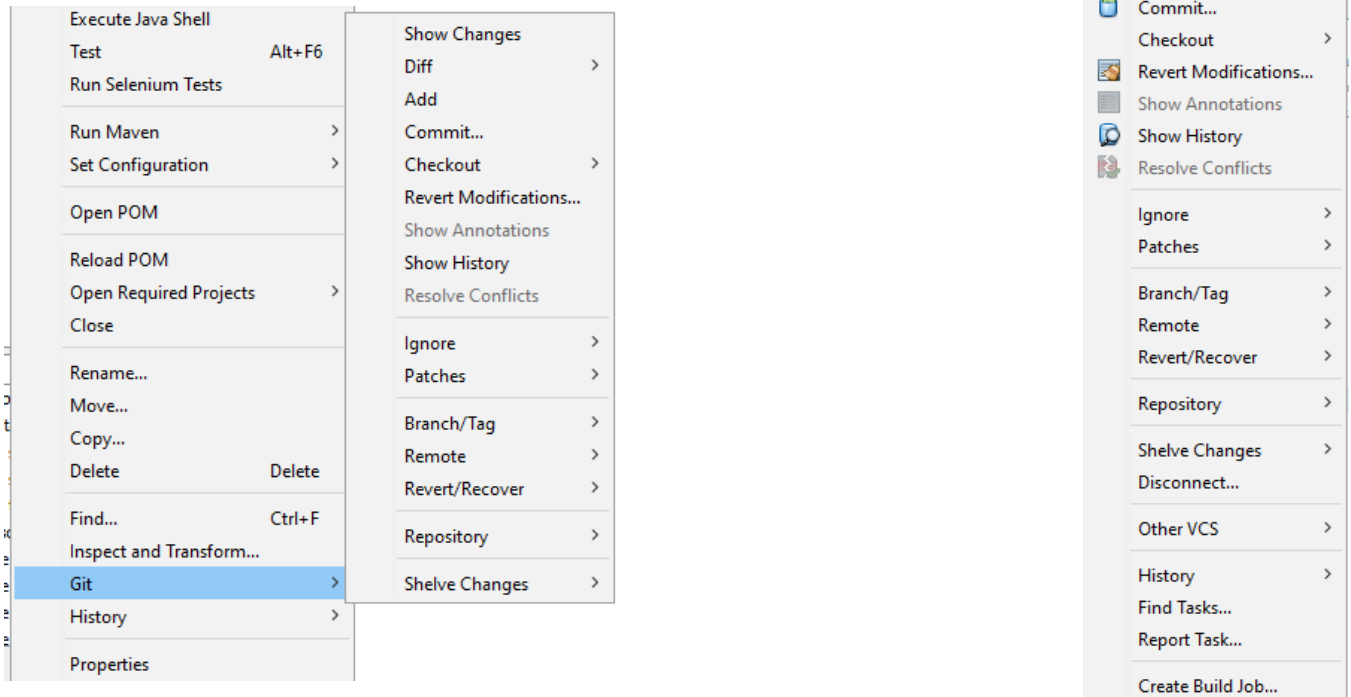


Configuración inicial

Configuración Podemos navegar y configurar en el menú: Equipo > Repositorio > Configuración global abierta. Podemos ver los usuarios y correos electrónicos para los que se identificarán los cambios realizados en todos los repositorios.

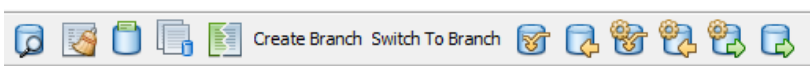
Acciones disponibles

Netbeans funciona de manera similar a la vista anterior a través de la interfaz de comandos, pero accede a estas acciones a través de un menú. Podemos acceder de dos maneras: desde el menú superior Equipo o haciendo clic derecho en Proyecto > Git



Podemos ver las opciones que se manejan con el comando git, tales como: add, commit, checkout, branch, etc. En la opción Remote tenemos comandos que utilizan repositorios remotos, tales como: push, fetch, pull, etc.

En el menú Ver podemos activar la barra de herramientas de Git para acceder rápidamente a los comandos de uso común.



Traballando en local

Cuando trabajamos localmente, los archivos nuevos aparecerán en verde y los archivos modificados aparecerán en azul. Los que no han sido modificados, es decir, los mismos desde el último commit, se mostrarán en negro.

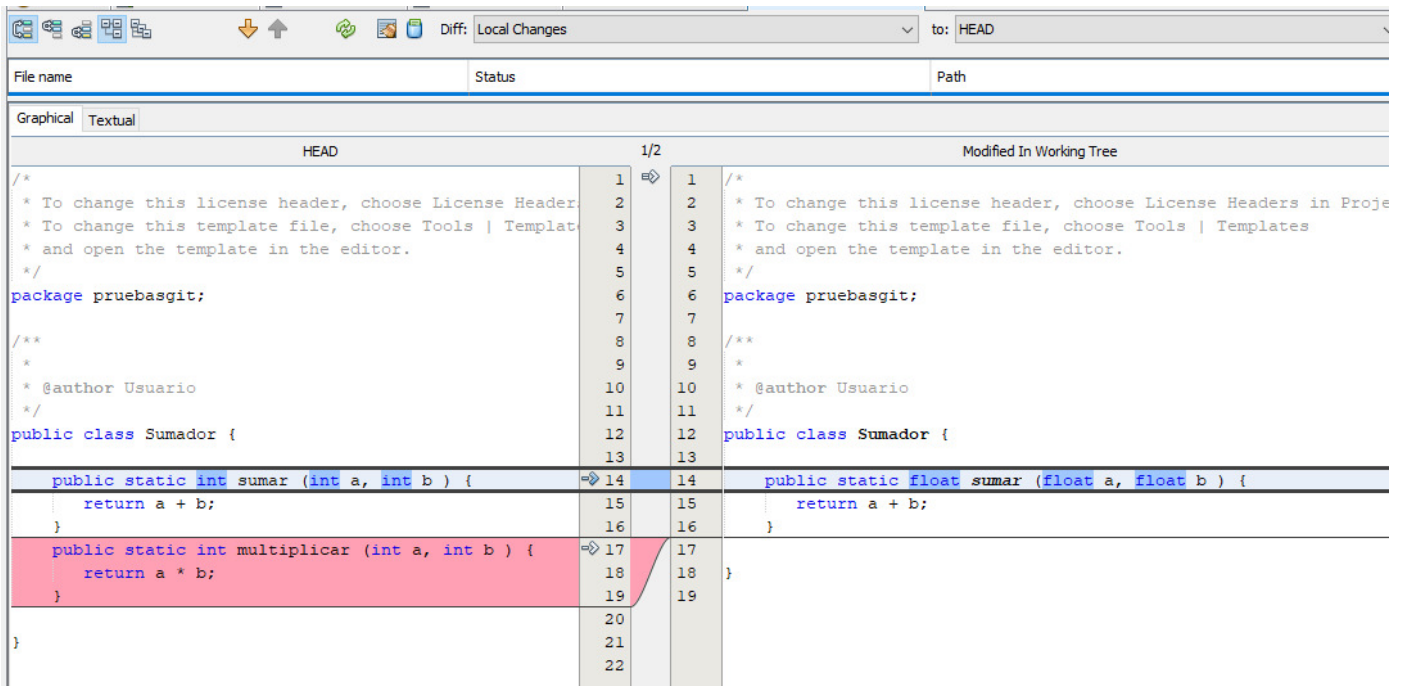
IMPORTANTE: Vamos a trabajar como si hubiéramos empezado desde la línea de comando, podremos realizar operaciones como commit, diff, checkout, etc., por lo que aquí no vamos a repetir la funcionalidad de cada comando, solo algunas particularidades.

En el momento del commit, podemos hacerlo desde el área de etapa (es decir, los archivos que hemos añadido anteriormente) o directamente desde todos los archivos modificados, independientemente de si los agregamos o no.

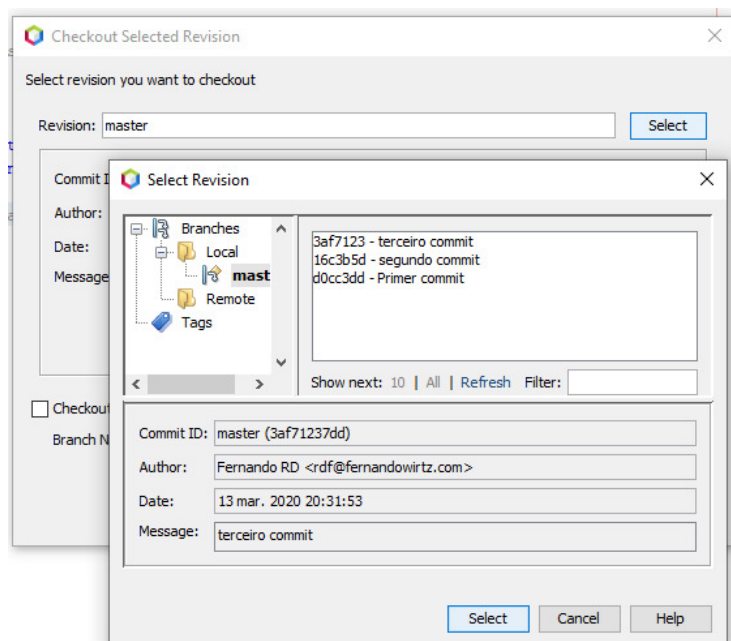


A través de la opción del menú Diff podemos comprobar las diferencias entre las diferentes versiones de un archivo. Es importante recordar que HEAD representa la última versión que hemos enviado, la versión "en producción", y el árbol de trabajo representa la versión en la que estamos trabajando.

El rojo indica el texto eliminado y el azul indica el texto modificado o agregado.



Para deshacer los cambios desarrollados en el directorio de trabajo, utilizaremos la opción del menú Checkout que, como hemos visto desde la línea de comandos, permite restaurar la versión del área de commit (HEAD) con la opción del menú Checkout archivo, así como restaurar la revisión de algunas versiones de commit intermedias con la opción del menú Checkout.



A través de la opción del menú Mostrar historia podemos ver la información completa de todas las situaciones por las que ha pasado cada archivo.

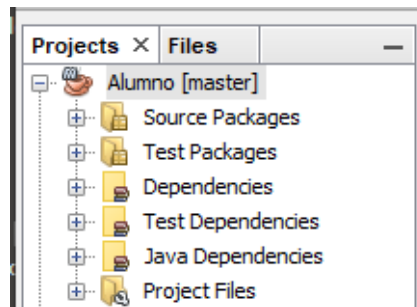
Gestión de sucursales

El concepto de ramificación es el mismo que ya hemos visto en la consola de Git. Desde el menú git > branch , podemos elegir entre crear una rama (crear una rama) y cambiar una rama (cambiar una rama).

Para saber en qué rama estamos trabajando en cada momento, podemos marcar la opción:

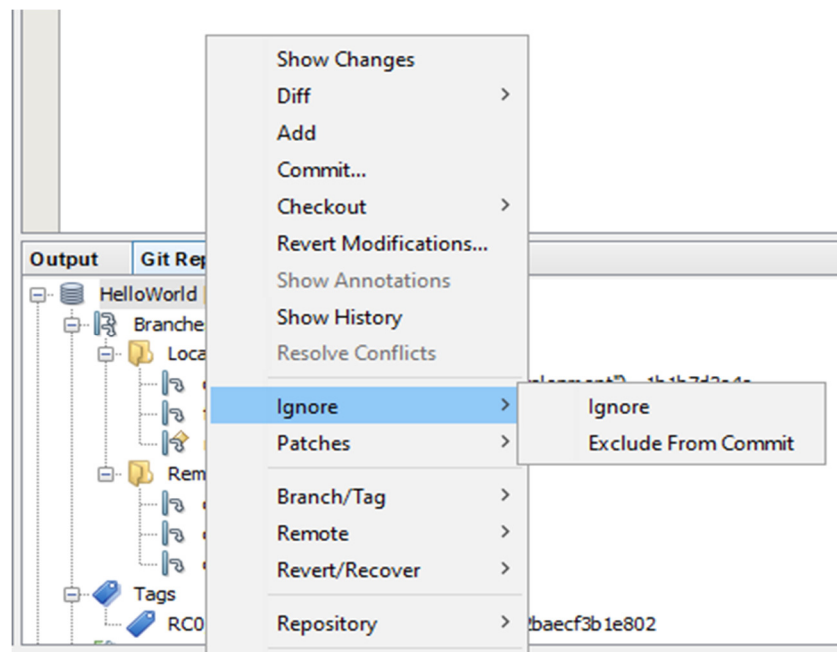
Muestra

Etiqueta de control de versiones Desde el menú Ver, el nombre de la sucursal aparecerá en la ventana Proyecto, junto al nombre del proyecto.



Ignorar archivos en el repositorio

Los archivos inherentes al entorno de desarrollo, o los archivos que contienen datos de configuración, no deben sincronizarse (al igual que las plantillas que realizan posibles cambios con valores de prueba).



Trabaja con servidores remotos.

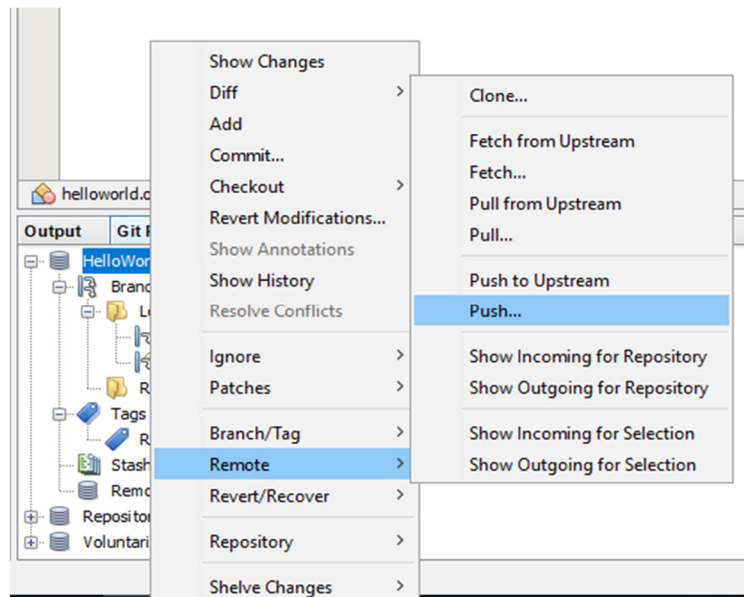
Hasta ahora, todos los cambios se han realizado localmente, por lo que ahora es el momento de aplicarlos a un servidor centralizado para que el resto del equipo pueda verlos.

Si ya tenemos un repositorio en GitHub:

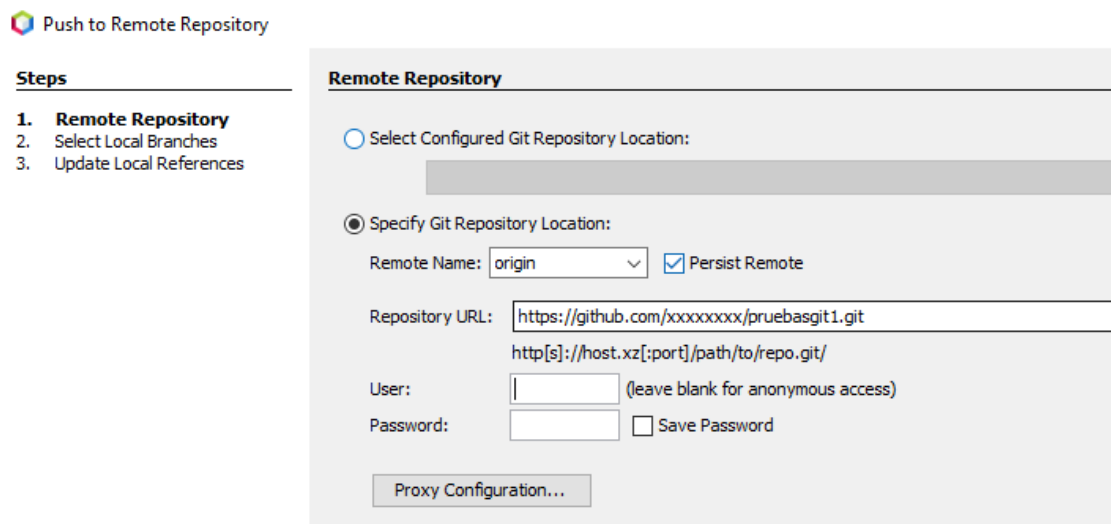
- En su cuenta de GitHub, vaya a su repositorio y haga clic en [Clonar] para obtener la URL del repositorio.
- e logotipo, ya en NetBeans: Team > Git > Clone. Allí, rellenos la URL de nuestro repositorio remoto, el usuario, el token de acceso personal (no su contraseña) y la carpeta de destino (si es necesario).
- En el último paso del proceso ofreses crear un proyecto desde un repositorio remoto y nosotros respondemos que sí.
- Ahora vamos a trabajar con el proyecto local de la manera mencionada en las secciones anteriores, cuando queramos subir el commit a GitHub haremos clic derecho en el proyecto > Git > Remote > Push. Cabe señalar que los compromisos realizados localmente no se reflejarán en servidores externos siempre que no se envíen.

Si queremos copiar un proyecto local en un repositorio vacío de GitHub:

- Creamos un repositorio sin contenido en GitHub. Al crearlo, obtendremos una URL.
- En nuestro proyecto Netbeans: Git > Remote > Push:



- Ahora tenemos que indicar la URL de nuestro repositorio en GitHub y nuestras credenciales, recordando que tenemos que ingresar nuestro token personal, no nuestra contraseña de GitHub:



Para obtener un token personal, ve a tu perfil en Github: Configuración > Configuración del desarrollador. Mira este video para obtener más información: <https://www.youtube.com/watch?v=wnMu5yJ5iL8>

—Finalmente instruimos a las ramas para sincronizarlas, generalmente son o master.

-Además, si el usuario ha realizado cambios en el repositorio remoto, podemos "descargar" estos cambios al repositorio local haciendo clic derecho en Proyecto > Git > Remote > Pull.

Misión 4.10. Git con Netbeans

A) Crear un proyecto con dos archivos: una clase de la siguiente manera:

```
Clase pública sumador {  
    Flotador estático público sumar (flotador a, flotador b) {  
        Devuelve a + b;  
    }  
}
```

Y un programa que llama a algunos de sus métodos

```
Clase pública newmain {  
    public static void main (String [] args) {System.out.println(  
  
    }  
}
```

Hace que el proyecto tenga el control de versiones de git. ¿De qué color son los nombres de los archivos?

li) Presentación de proyectos. ¿De qué color son los nombres de archivos ahora?

c) Empuje el proyecto a un nuevo repositorio en su cuenta de GitHub.

d) Crear localmente una rama llamada "dev" en la que se borra la clase de verano del punto anterior y crear una nueva rama llamada Restador:

```
Restador de clase pública {  
    Restar de flotador estático público (flotador a, flotador b) {  
        Devolver a-b;  
    }  
}
```

Llamar al nuevo método en lugar del método anterior en la clase NewMain que contiene main ():

Commit y push desde una nueva rama.

e) Proba en Netbeans a hacer git > Branch > Swith to... Para cambiar de una rama a otra, verá cómo ver el archivo Sumador.java o Restador.java en la ventana del proyecto, dependiendo de la rama en la que se encuentre.

f) Ve a tu cuenta de GitHub y toma dos capturas de pantalla, una muestra la rama master y la otra muestra la rama dev.

g) En Netbeans, compruebe las diferencias de la clase NewMain de la versión master con respecto a la versión dev.

h) Crear una rama llamada "hotfix" en la que cambias las líneas: System.out.println(

```
System.out.println (Sumador.sumar(16f, 4f));
```

i) Confirmar la rama de parches y aplicar los cambios a la rama maestra. ¿Es posible avanzar rápidamente?

j) Aplicar los cambios de la rama dev a master. Si hay contradicciones, resuélvelas.

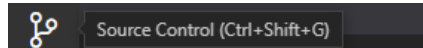
k) Realizar un push desde todas las ramas al repositorio remoto.

l) Mostrar el historial de los cambios realizados.

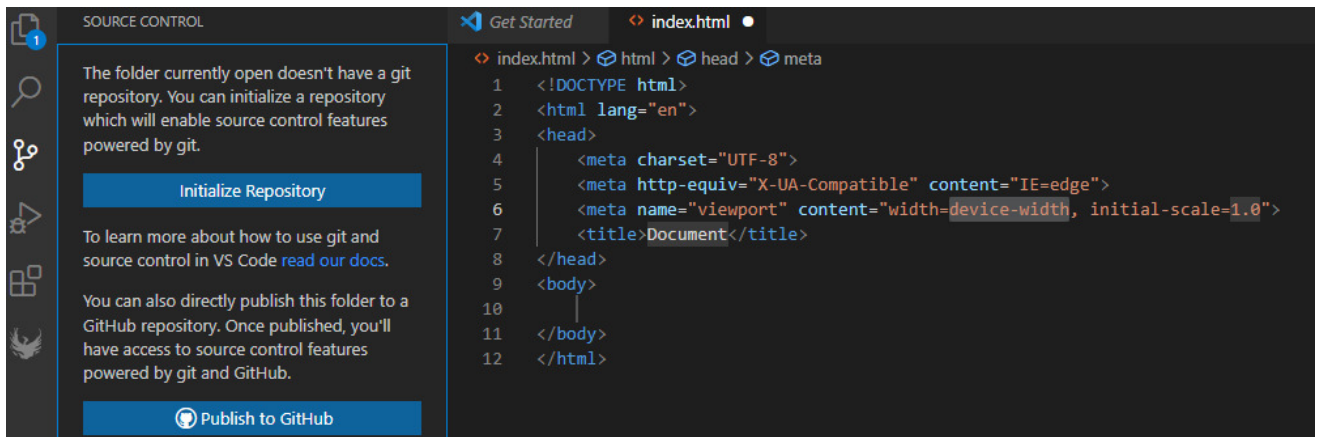
4.10 Control de versiones del código de Visual Studio con Git

El código de Visual Studio ofrece control de versiones de forma nativa, sin instalar extensiones

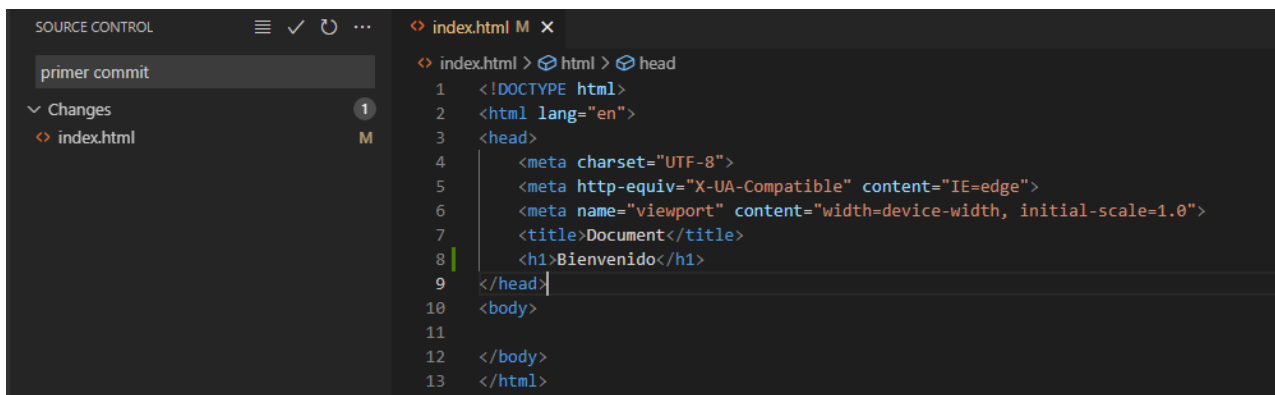
A través de los botones de la barra lateral.



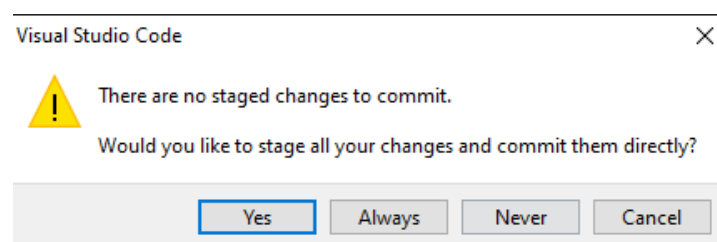
Primero abriremos nuestra carpeta de proyectos y luego podemos presionar el botón anterior. Haga clic para abrir el panel "Control de código fuente", desde el cual podemos realizar todas las acciones que podemos realizar a través del terminal de comandos. La primera acción será inicializar el repositorio [Iniciar el repositorio].



Entonces funcionaremos normalmente, editando nuestros archivos. Para commit tenemos el icono en el título del panel de control de código fuente y el resto de los comandos de Git en el icono. El IDE marcará con diferentes colores la ubicación de las líneas de código y la ubicación de los archivos de letras de color en el editor.

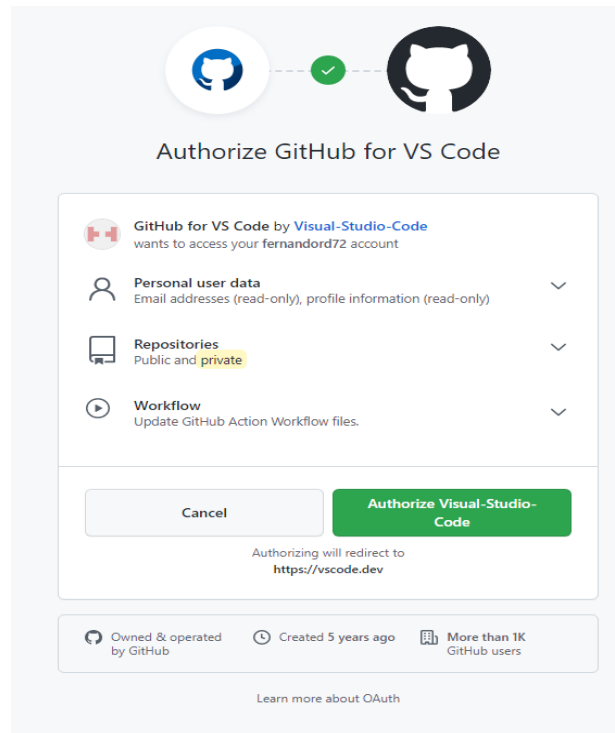


Al realizar el commit, si no pasamos el archivo a la zona escalada (lo que hace el comando add), haga esto automáticamente marcando la opción Always para que siempre se haga.

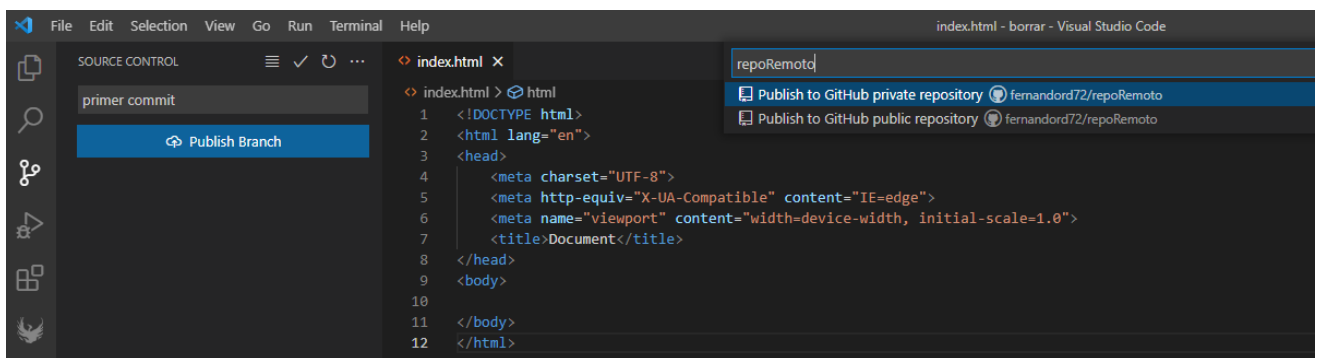


Nos solicita un mensaje de commit al commit y nos pregunta si queremos cargar la rama a GitHub a través del botón [Publicar rama].

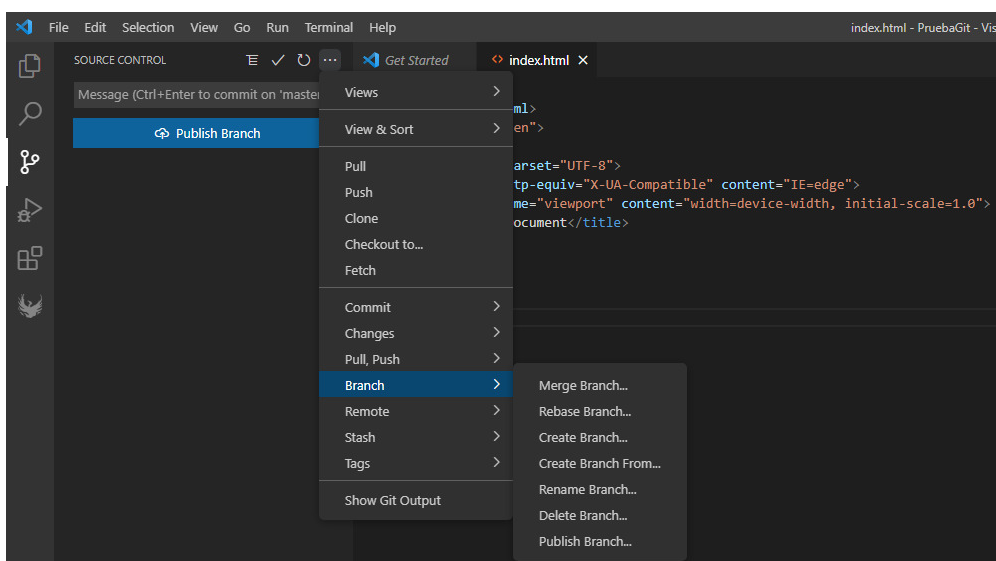
Primero nos pedirá nuestras credenciales en Github, y esta plataforma le pedirá confirmación de permisos.



Al ejecutar una "rama de liberación" por primera vez en cada proyecto, nos permite crear repositorios remotos desde el propio IDE, ofreciendo la opción de crear privado o público.



Para el resto de las tareas utilizaremos los iconos que hemos anotado en el panel de control de código fuente:



Por último, recuerde que hay una gran cantidad de extensiones en Visual Studio Code para facilitar el uso de Git y GitHub. Ejemplo: "Pull Request and Issues" proporciona la funcionalidad de trabajar con repositorios remotos, como el icono [Pull Request] en el encabezado del panel de control de fuente.