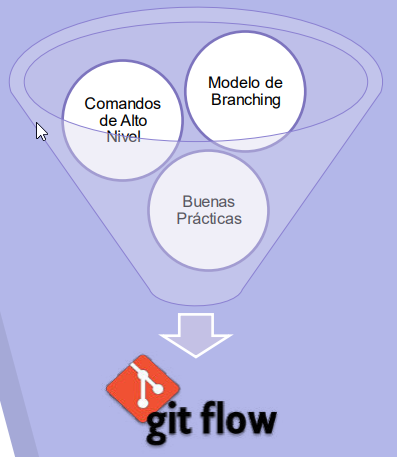
# Curso de Gitflow

¿QUÉ ES?

Git es un sistema de control de versiones, que nació para dar soporte al core Linux, cuya principal característica es que es capaz de mantener una gran cantidad de código fuente distribuido y gestionado por equipos de envergadura, de forma rápida y sencilla.

GitFlow

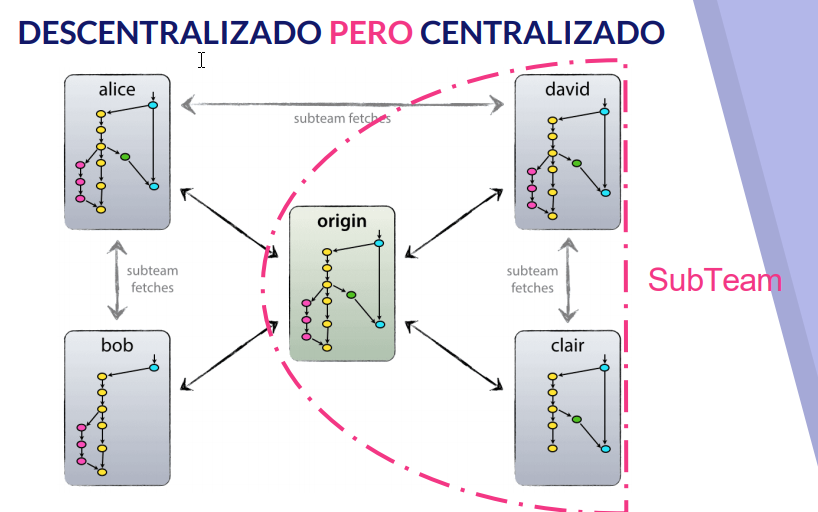


GitFlow es un modelo de branching que propone una estrategia de ramificación y de generación de versiones de productos usando un repositorio GIT. Ayuda a los equipos a asegurar la calidad del código y la eficiencia en la integración en equipos de más de un miembro.

Descentralizado pero Centralizado

Como dijo Vincent Driessen, GIT ha revolucionado la forma de hacer software. Antes con SVN o CVS los merges daban un poco de miedo, ya que los conflictos se generaban en el único repositorio que tienen, el remoto, y directamente sobre el trunk. Eso hace los conflictos extensibles a todos los miembros del equipo y además se generan directamente sobre la rama principal. Digamos que no favorece mucho el branching y en trabajo colaborativo, sin embargo, con git es extremadamente fácil, al contar con el repositorio local.

Por eso decimos que con git-flow partimos de un repositorio centralizado, pero que a la vez es descentralizado. Esto se debe a que todos tenemos nuestro repositorio local, como hemos comentado, pero a su vez apuntamos a un repositorio remoto llamado “origin”. Toda la comunicación entre los desarrolladores tiene que pasar por origin y no de un desarrollador a otro, lo que refleja todos los cambios que hace cada uno y les permite integrar código en sus repositorios locales y detectar conflictos de forma temprana.



GitFlow define dos tipos de ramas, las principales y las auxiliares. Las principales diferencias entre estos dos grupos de ellas son:

Ramas Principales (master y develop):

No se pueden instanciar

Su nombre no tiene complementos, se llaman master y develop “a secas”

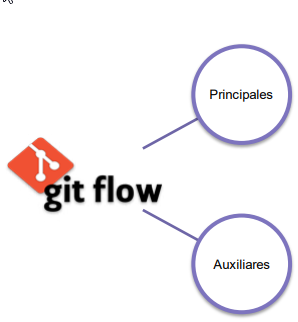
No se puede commitear código directamente sobre ella, para no perder el control de los cambios, solo recibe merges de otras ramas

Ramas Auxiliares (feature, release y hotfix):

Se pueden instanciar tantas veces como características, versiones y defectos haya. No hay límite.

Su nombre consta de una raíz, que corresponde al tipo de rama, más la característica implementada o versión. Este complemento la hace inconfundible

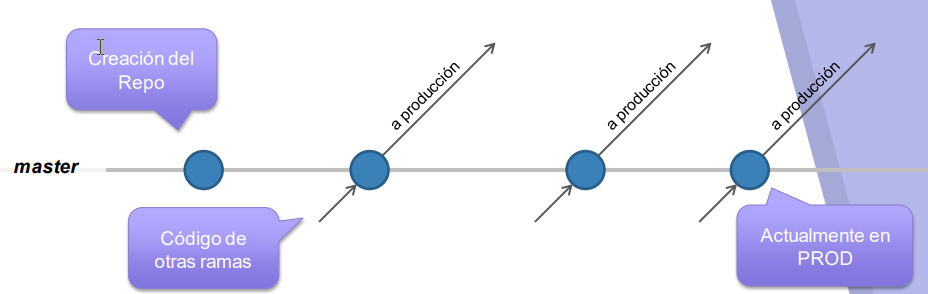
El objetivo es trabajar directamente sobre ella, subiendo código código a la rama. Luego se mergea con las ramas principales.



La rama master es la rama principal del repositorio y contiene el código que está actualmente en producción o próximo a subir en una situación estable.

No se hace commit de código de ella, salvo circunstancias muy especiales y por alguien con altos conocimientos en el producto.

Por regla general, recibe merges de ramas release o hotfix. En esta rama están etiquetadas todas las versiones del producto.

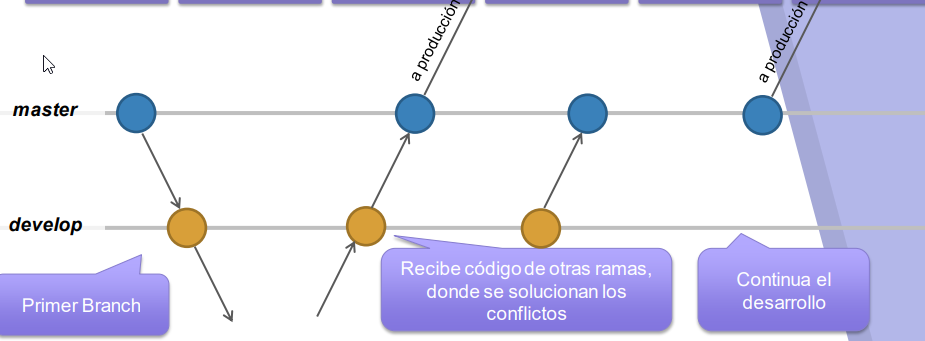


La rama develop es la rama principal del repositorio que contiene, además del contenido de master, características desarrolladas que aún no están en producción.

En esta rama se está desarrollando código de forma continua, eso sí, todo el contenido que recibe a través de merges de ramas features principalmente, está testeado y verificado, por lo que es código estable.

No se hace commit de código de ella, salvo circunstancias muy especiales, al igual que en la rama master.

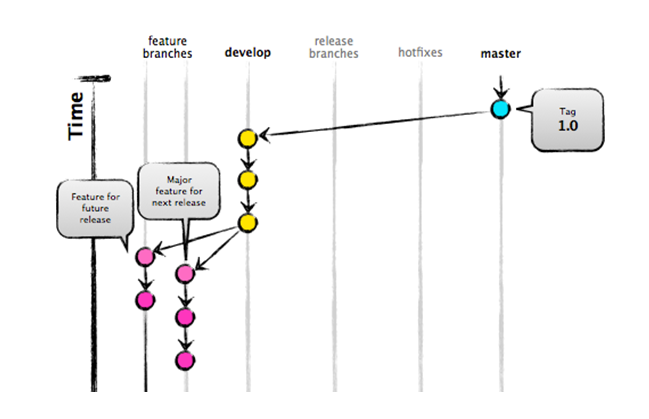
El principal objetivo de esta rama es evitar conflictos es master, por lo que el código se depura en esta rama primero.

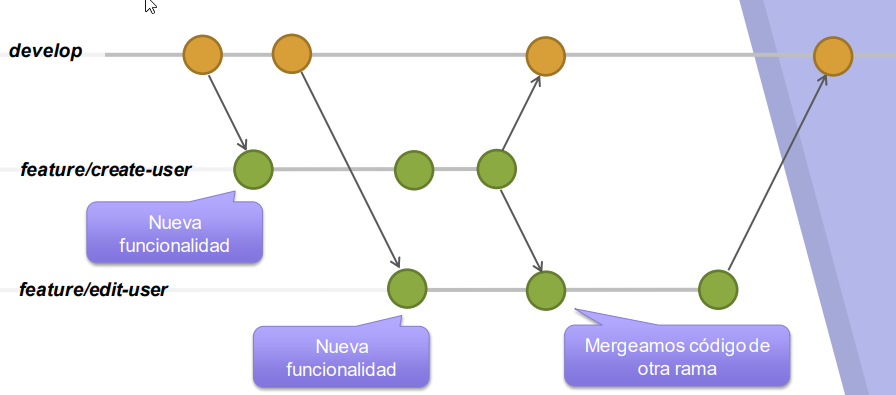


Cada vez que se va a desarrollar una nueva funcionalidad o modificar una existente, se genera una rama de tipo feature, a partir de develop, cuyo nombre está formado por esta palabra y por la característica a implementar, por ejemplo: “feature/nueva-interfaz”.

Sobre ella se sube el código a través de commits del equipo que debe ser probado antes de mergear con develop, por lo que durante este proceso hay una funcionalidad en curso codificándose, pudiendo trabajar varios miembros a la vez en ella.

Una buena práctica es hacerlas lo más atómicas posibles, pudiendo generar más de una para cubrir una característica y mergeando código de unas a otras.

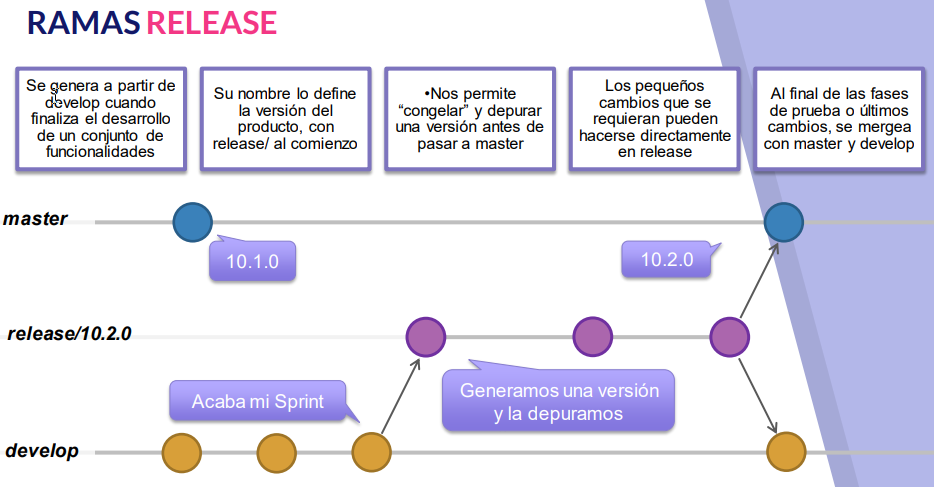




Estas ramas se generan cuando quiere realizarse la tentativa de entrega de una versión, habitualmente cuando se aproxima la finalización de un sprint.

Esto congela los desarrollos, que prosiguen en develop, y agrupa una serie de nuevas características en la nueva rama release, que puede llamarse, por ejemplo: “release/v2.2.0”.

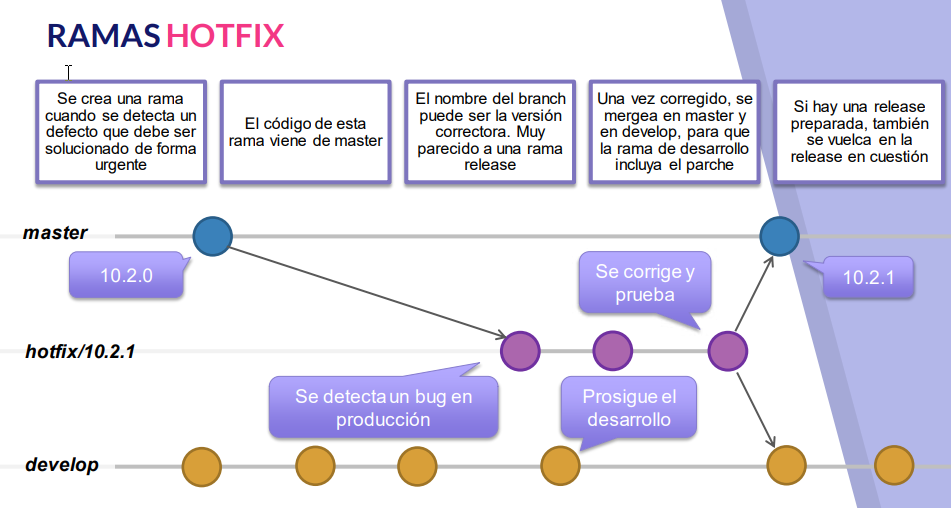
Sobre ella pueden realizarse pequeños cambios, corrección de defectos, establecer versiones de producto… pero no desarrollar nuevas características. Una vez finalizado y testeado, se mergea con master, como nueva versión que es candidata a subir a producción; y con develop, ya que hay defectos corregidos que deben volcarse a la rama de desarrollo.



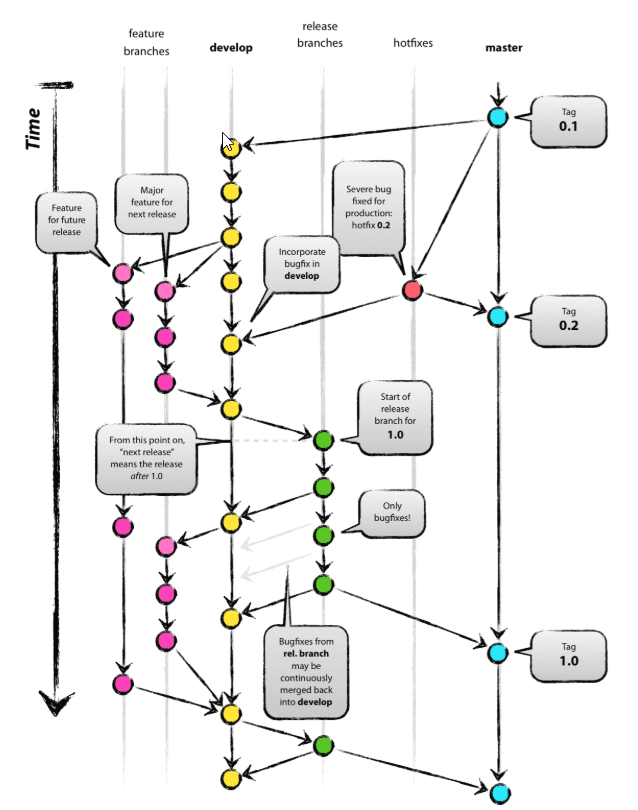
Estas ramas se generan habitualmente cuando se detecta un defecto en producción que debe ser corregido con agilidad. Para ello se genera una rama de tipo hotfix, que parte del código existente en master.

Sobre ella pueden realizarse las correcciones que sean necesarias para arreglar el bug, para luego volver a mergear con master, añadiéndole un nuevo tag de versión.

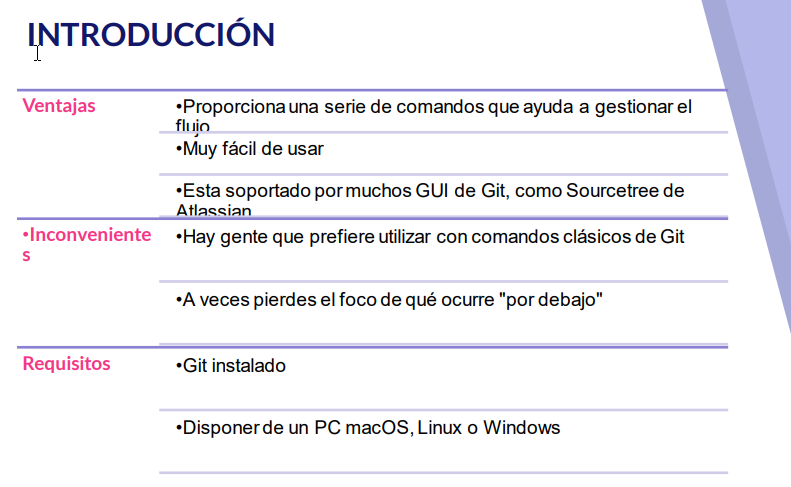
También es necesario mergear con develop para que el defecto también quede corregido en desarrollo.



Uniéndolo todo, el proceso cobra sentido, y todos los tipos de ramas se complementan los unos a los otros para alcanzar los objetivos que propone GitFlow.



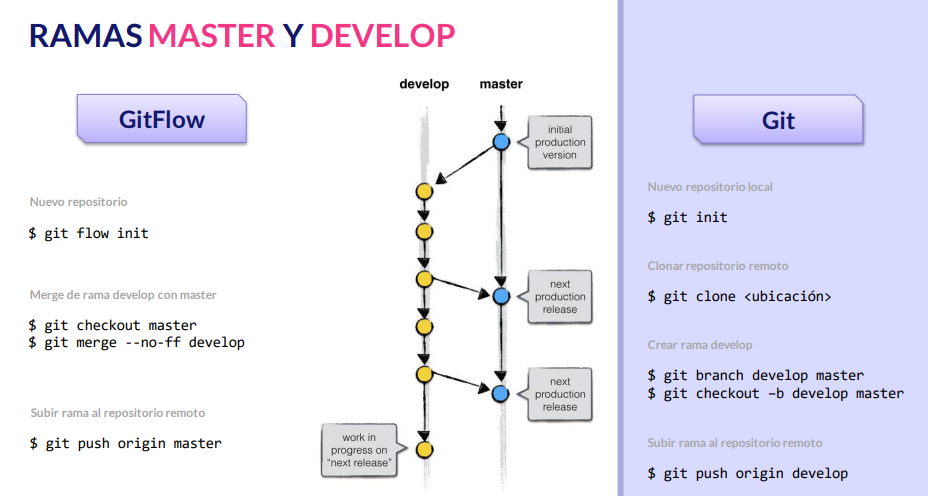


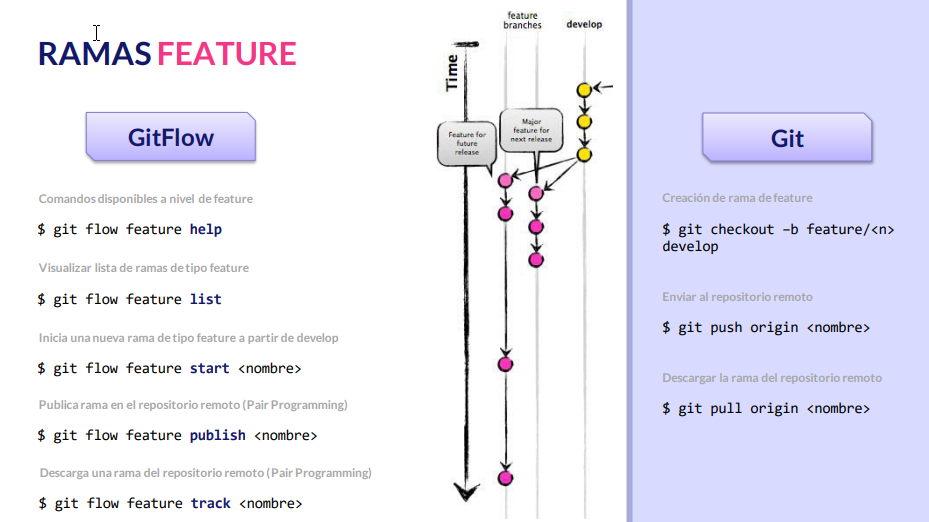


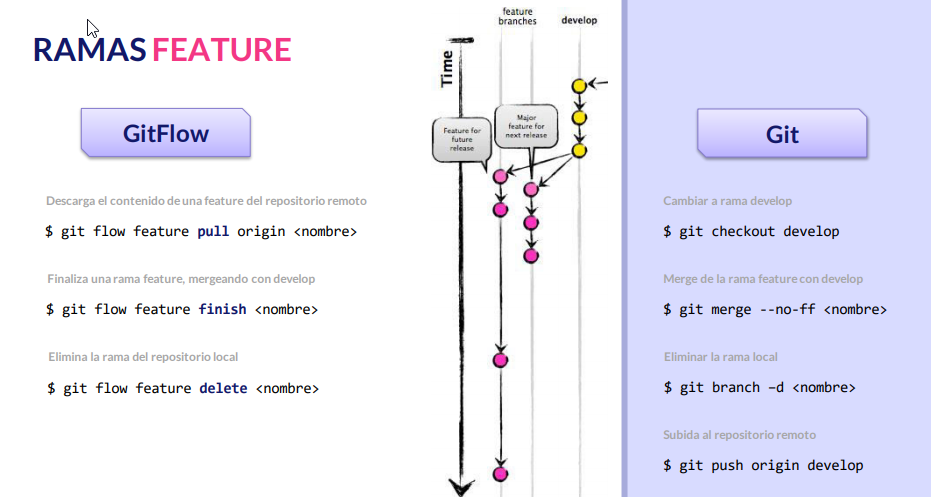


Comandos Git:

|  |
| --- |
|  |







Comandos de Git:

Iniciar el servicio en una carpeta.

|  |
| --- |
| Git init |

Añadir archivos a git.

|  |
| --- |
| Git add |

Ver el estado de la carpeta y sus archivos a cambiar.

|  |
| --- |
| Git status |

Confirmar cambios en git.

|  |
| --- |
| Git commit |

Añadir un archivo de configuración de donde se marcará los archivos a ignorar.

|  |
| --- |
| Echo “\*.tmp” > .gitignore  #aceptamos los cambios con los comandos.  Git add .gitignore  Git commit -m “añadiendo el .gitignore” |

Para mirar los logs:

|  |
| --- |
| Git log --oneline |

Añadir todos los ficheros para confirmarlos.

|  |
| --- |
| Git add .  Git commit -m “tercer cambio” |

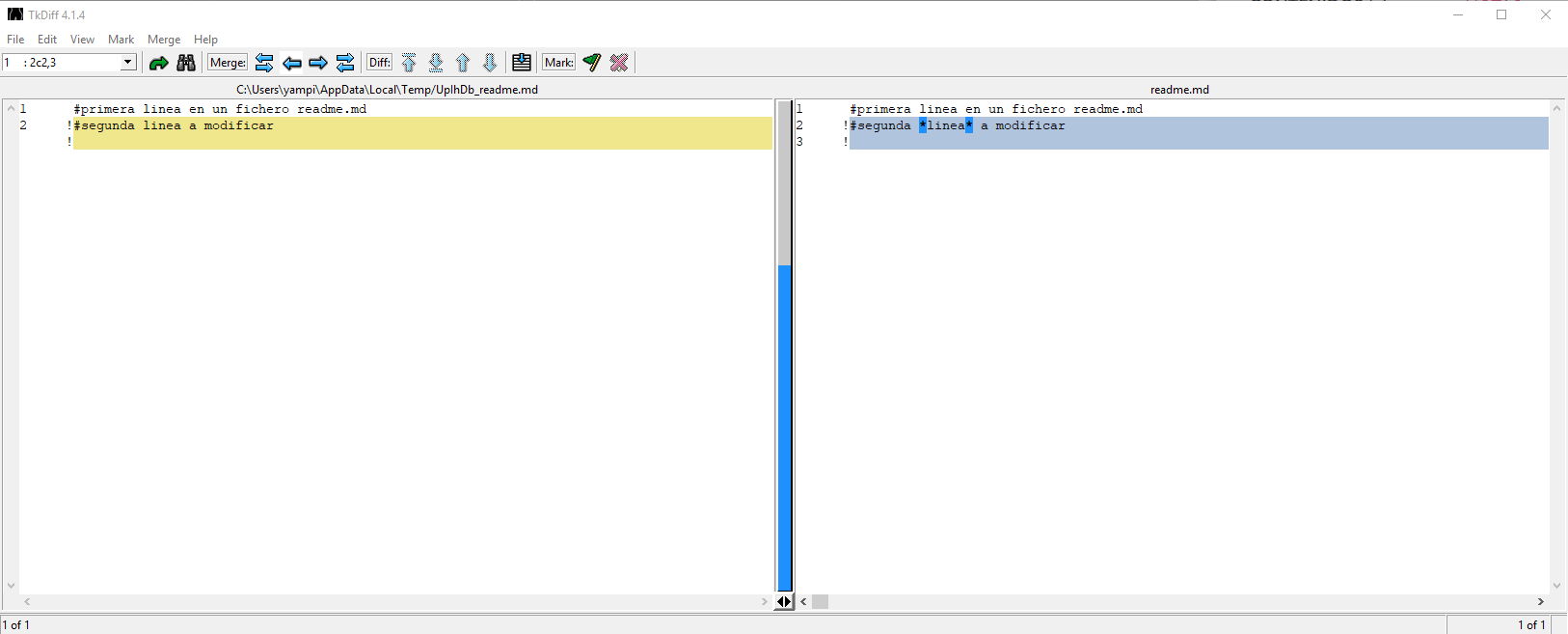
Una vez añadido unos cambios en un archivo que se agregó a git este puede realizar el comando

|  |
| --- |
| Git diff |

Para ver los cambios.

|  |
| --- |
| Git difftool |

Al copiarlo a los archivos binarios del usuario en la carpeta de GIT podemos abrirlo de modo gráfico.



Borrar ficheros

|  |
| --- |
| Git rm fichero |

Mover ficheros

|  |
| --- |
| git mv readme2.md readme.md |

Ver los log completo

|  |
| --- |
| git log --pretty=format:"%h %an %ar - %s" |

Ver al detalle los cambios realizado en los archivos de log.

|  |
| --- |
| Git show ID |

Ver los cambios añadimos de la carpeta a pendiente de “commit”.

|  |
| --- |
| git diff –staged |

## Remoto:

Inicializar el git y su infraestructura.

|  |
| --- |
| Git init |

Clonar

|  |
| --- |
| git remote add origin https://github.com/cdryampi/referendum2030.git |

Ver la información del repositorio y sus versiones(ramas-branches).

|  |
| --- |
| git fetch origin |

Listar las versiones disponibles del repositorio a vincular.

|  |
| --- |
| Git Branch -v -a |

Descargarse una rama o el master del repositorio

|  |
| --- |
| git pull origin master |

Modo sueve:

|  |
| --- |
| Git clone https://github.com/cdryampi/referendum2030.git |

Ejemplo de forzar la copia a la carpeta actual.

|  |
| --- |
| git clone https://gitlab.com/juancarlos.rubio/nuevo . |