Git & GitHub pt. 2

Autor: José Guillermo Sandoval Huerta

Fecha: 31 octubre 2023

Índice

Comandos avanzados

Los comandos avanzados de Git son herramientas que permiten a los usuarios de Git realizar tareas más complejas y personalizadas en su flujo de trabajo de control de versiones. Estos comandos son útiles para tener mayor flexibilidad y control sobre el repositorio. A continuación, se presentan algunos de estos comandos.

Pull Request

También conocido como PR es una característica fundamental en sistemas de control de versiones como Git, sirve para facilitar la colaboración y la revisión de código en proyectos de desarrollo de software [1].

Realizar un Pull Request es un proceso que va desde (1) hacer Fork a un repositorio, (2) clonar el repositorio, (3) crear una rama nueva, (4) realizar cambios, (5) confirmar y subir los cambios, (6) Crear el pull request, donde en el repositorio original se encontrara una opción para cera un nuevo Pull Request. Se tendrá que hacer clic en el botón "New Pull Request", se verificara los cambio y se da clic en el botón "Create Pull Request". Por ultimo (7) el colaborador inicial tendrá que aprobar el pull request y luego fusionará el pull request en el repositorio principal.

Entre sus principales funcionalidades están:

- **Solicitar una fusión de cambios**: Un Pull Request es una solicitud formal que un colaborador hace para fusionar sus cambios desde una rama (branch) de trabajo en su repositorio con otra rama, generalmente la rama principal del proyecto, como master o main. Esto permite que otros revisen y consideren los cambios antes de que se fusionen en la rama principal.

- **Facilita la revisión de código**: Los Pull Requests proporcionan un espacio para que los colaboradores revisen y comenten los cambios propuestos. Esto fomenta la revisión de código, lo que ayuda a mejorar la calidad del código y a detectar posibles errores antes de que se fusionen en la rama principal.

- **Discusión y colaboración**: Los comentarios en un Pull Request permiten a los colaboradores discutir los cambios propuestos. Esto fomenta la colaboración, la resolución de problemas y la comunicación efectiva entre los miembros del equipo de desarrollo.

- **Seguimiento de cambios**: Los Pull Requests mantienen un registro de los cambios propuestos y de la discusión que los rodea. Esto es útil para entender por qué se realizaron ciertos cambios y para mantener un historial de decisiones.

- **Integración continua**: Muchas plataformas de control de versiones y hospedaje de repositorios, como GitHub y GitLab, pueden configurarse para ejecutar automáticamente pruebas y verificaciones de calidad en los cambios propuestos antes de la fusión. Esto asegura que los cambios no rompan el código existente.

- **Autorización y control de acceso**: Los Pull Requests a menudo permiten que los mantenedores del proyecto tengan un control más granular sobre quién tiene permiso para fusionar cambios en la rama principal. Esto ayuda a garantizar la calidad del código y la seguridad del proyecto.

- **Documentación y contexto**: Los Pull Requests pueden incluir descripciones detalladas que explican el propósito de los cambios, proporcionando contexto adicional para los revisores.

Fork

Fork es una copia de un repositorio de Git existente al que se puede trabajar de manera independiente. Normalmente, se realiza un fork de un repositorio de un tercero o de un proyecto de código abierto para contribuir a él sin afectar directamente al repositorio original. Al hacer un fork, se crea una copia en la cuenta del usuario [2].

Para realizar un fork hay que realizar lo siguiente (1) ir al repositorio al que se quiere hacer fork, (2) en la parte superior derecha del repositorio, hacer click en el botón "Fork", (3) seleccionar la ubicación del fork, (4) una vez que se haya completado el fork, la página re direccionara a la nueva copia del repositorio.

Algunos aspectos importantes son los siguientes:

- **Propósito principal**: Los forks se utilizan comúnmente para colaborar en proyectos de código abierto. Cuando haces un fork de un proyecto de código abierto, puedes clonar la copia en tu máquina local, realizar cambios, y luego enviar un pull request al repositorio original para que los propietarios del proyecto consideren fusionar tus cambios.

- **Independencia**: El fork es completamente independiente del repositorio original. Esto significa que puedes hacer cambios en tu fork sin afectar al proyecto original y viceversa.

- **Clonación y sincronización**: Después de hacer un fork, puedes clonar tu copia en tu máquina local y trabajar en ella como en cualquier otro repositorio Git. Puedes sincronizar tu fork con el repositorio original de forma periódica para mantenerlo actualizado con los cambios realizados en el proyecto original.

- **Colaboración**: Cuando haces un fork de un proyecto y envías un pull request, estás proponiendo cambios al proyecto original. Los propietarios del proyecto original revisarán tus cambios y pueden decidir si los incorporan o no en su repositorio.

- **Diferencias con una rama**: A diferencia de crear una rama en un repositorio existente, un fork es una copia independiente del repositorio. Las ramas se utilizan para trabajar en cambios dentro del mismo repositorio, mientras que los forks se utilizan para crear una copia separada del repositorio original.

Rebase

El comando **git rebase** se utiliza para modificar el historial de confirmaciones. Permite reorganizar, combinar o eliminar confirmaciones, lo que puede ayudar a mantener un historial de cambios más limpio y coherente. Sin embargo, se debe usar con cuidado, ya que puede causar conflictos si se utiliza en colaboración con otros [3].
El uso de git rebase es común para evitar la creación de fusiones (merges) innecesarias y mantener un historial de confirmaciones más limpio.

Stash

Es una funcionalidad que permite guardar temporalmente cambios sin comprometerlos en una confirmación (commit) ni perderlos. El stash se puede entender como un área de almacenamiento temporal donde se puede guardar cambios en el directorio de trabajo que no estás listo para comprometer en un commit, pero que tampoco se desean perder. Sin embargo, es importante mencionar que el stash no es adecuado para almacenar cambios durante un largo período, ya que está diseñado para cambios temporales y transiciones rápidas entre tareas [4].

El stash es útil en situaciones en las que se está trabajando en una rama y es necesario cambiar de rama rápidamente sin confirmar los cambios en la rama actual. Para ello se ocupa el siguiene comando:

- **git stash save "<Mensaje descriptivo (opcional)>"**

Entre otros comandos de su funcionamiento tenemos los siguientes:

- **git stash pop** - recuperar cambios del stash.
- **git stash apply** - aplicar los cambios del stash sin eliminarlos del stash.
- **git stash list** - listar los cambios en el stash.
- **git stash drop stash@<n>** - eliminar cambios del stash. Donde <n> es el índice del stash que deseas eliminar.

Clean

El comando **git clean** en Git se utiliza para eliminar archivos no rastreados y directorios que no están bajo control de versiones [5]. Es útil cuando se desea eliminar archivos generados automáticamente, archivos temporales, o cualquier otro tipo de archivo no deseado que no debería formar parte de tu repositorio Git. El comando git clean es útil para mantener el directorio de trabajo limpio y eliminar elementos que no deben estar allí. Este comando emplea algunas opciones de filtrado como son:

- **git clean -n** - o **--dry-run**, muestra una lista de archivos que se eliminarán sin realmente eliminarlos. Es útil para ver qué archivos se eliminarán antes de ejecutar el comando.
- **git clean -f** - o **--force**, ejecuta el comando en modo forzado, lo que significa que los archivos se eliminarán sin necesidad de confirmación adicional.
- **git clean -i** - o **--interactive**, permite confirmar o rechazar cada archivo antes de eliminarlo.

Cabe mencionar que git clean es irreversible, por lo que es importante tener precaución al usarlo y asegurarse de que los archivos que se eliminarán no son necesarios en tu proyecto.

Cherry-pick

El comando **git cherry-pick** se utiliza en Git para aplicar commits específicos de una rama a otra. Esta operación permite seleccionar uno o varios commits individuales y copiarlos en una rama diferente. El cherry-pick toma los cambios introducidos por un commit y los aplica en la rama actual. Esto es útil cuando se desea traer cambios específicos de una rama a otra sin tener que fusionar la rama completa [6]. Para ejecutar este comando es por medio de la siguiente sentencia:

- **git cherry-pick <commit-hash>** - Siendo <commit-hash> el hash SHA-1 del commit que se desea aplicar.

El uso de este comando puede causar los siguientes procesos:

- **Conflictos**: Pueden ocurrir conflictos si los cambios del commit que estás aplicando entran en conflicto con los cambios existentes en la rama actual. Se deberán de resolver manualmente, similar a como se hace en una fusión (merge).

- **Orden de aplicación**: Los commits se aplican en el orden en que se especifiquen en el comando. Se puede aplicar varios commits en secuencia proporcionando múltiples hash SHA-1 en el comando.

- **Historia de commits**: Los commits copiados tendrán nuevos hash SHA-1 en la rama de destino, ya que se consideran commits nuevos con la misma funcionalidad.

Bisect

El comando **git bisect** se usa para encontrar un commit específico en la historia de un proyecto que introdujo un problema o un error. Se usa comúnmente cuando se está tratando de identificar exactamente cuál commit causó un fallo en el código. El comando git bisect realiza una búsqueda binaria (bisecting) para encontrar el commit problemático. Permitiendo ahorrar mucho tiempo y esfuerzo al no tener que revisar manualmente commits uno por uno en busca del culpable [3].

El proceso general de uso de git bisect implica los siguiente:

(1) Inicio del proceso: Inicialmente, se deberá decir a Git cuál fue el último commit en el que el proyecto funcionaba correctamente (un "commit bueno") y cuál fue el primer commit en el que se encontró el problema (un "commit malo"). Esto se hace usando los siguientes comandos:

- **git bisect start**

- **git bisect good <commit-bueno>**

- **git bisect bad <commit-malo>**

(2) Búsqueda binaria: Git comenzará a realizar una búsqueda binaria, seleccionando un commit intermedio entre el commit bueno y el commit malo. Luego, se deberá probar el proyecto para verificar si el error está presente en ese commit. Si el error está presente, se debe marcar el commit como "malo"; si no, como "bueno" usando:

- **git bisect good**

- **git bisect bad**

(3) Repetición del proceso: Git repetirá este proceso, seleccionando un nuevo commit intermedio, hasta que se encuentre el commit problemático. Una vez que Git haya encontrado el commit que causó el problema, proporcionará información sobre ese commit.

(4) Finalización del proceso: Para finalizar el proceso, se ejecuta **git bisect reset** para volver al estado normal de trabajo

Filter-Branch

El comando **git filter-branch** permite reescribir la historia de un repositorio, lo que implica aplicar filtros y transformaciones a los commits y su contenido [7]. Esta herramienta se utiliza en situaciones específicas en las que se necesita reestructurar o limpiar la historia de un repositorio. Es importante tener en cuenta que git filter-branch es una operación de reescritura de la historia y, por lo tanto, puede ser peligrosa si se utiliza incorrectamente. Además, una vez que se haya ejecutado git filter-branch, los commits reescritos tendrán nuevos hash SHA-1, lo que afectará a cualquier clon del repositorio existente y requerirá que todos los colaboradores actualicen sus copias locales del repositorio

Entre las principales funcionalidades están;

- **Eliminar archivos o carpetas específicos de todos los commits**: Permitiendo eliminar archivos o carpetas que se agregaron accidentalmente o que ya no deberían estar en el historial.

- **Cambiar la dirección de correo electrónico o nombre del autor**: Permitiendo cambiar la información del autor de los commits en caso de que se haya ingresado incorrectamente o se deba anonimizar.

- **Eliminar commits específicos**: Permitiendo eliminar commits enteros de la historia si, por ejemplo, contienen información sensible que no debe estar en el repositorio.

- **Fusionar múltiples commits en uno**: Permitiendo combinar varios commits en un solo commit para simplificar la historia.

- **Realizar conversiones de formato o renombrar archivos masivamente**: Permitiendo realizar tareas de transformación en los commits, como cambiar el formato de los archivos o cambiar nombres en toda la historia del proyecto.

En la mayoría de los casos, es preferible utilizar otras técnicas de Git, como git reset, git rebase o git commit --amend, para realizar cambios más simples en la historia del repositorio. git filter-branch se reserva para situaciones en las que se necesita realizar cambios más complejos y profundos en la historia del proyecto.

Reflog

Reflog es un registro que almacena un historial detallado de las referencias (ramas y etiquetas) en el repositorio. Contiene información sobre los cambios en las referencias, como los commits a los que apuntan, y permite recuperar información que de otro modo podría perderse. La palabra "reflog" es una abreviatura de "reference log" o "registro de referencias". El reflog es especialmente útil en situaciones en las que se ha perdido accidentalmente una referencia, como una rama o una etiqueta, o cuando se han realizado cambios que deseas deshacer [8]. Para ejecutar este proceso es por medio del siguiente comando:

- **git reflog**

Este comando mostrará una lista de eventos que involucran referencias en el repositorio, incluyendo información sobre los commits a los que apuntaban en cada evento. Es importante tener en cuenta que el reflog solo almacena un historial limitado de eventos y referencias. Con el tiempo, los eventos antiguos pueden ser eliminados del reflog para evitar que crezca demasiado. Por lo tanto, es útil utilizar el reflog con prontitud cuando se necesita recuperar información perdida o deshacer cambios.

Referencias

[1] J. Domingo Muñoz, «¿Cómo colaborar en un proyecto de software libre? ¿Qué es un Pull Request?,» josedomingo.org, 2023. [En línea]. Available: https://www.josedomingo.org/pledin/2022/09/que-es-pull-requests/#:~:text=Podemos%20definir%20un%20Pull%20Request,acci%C3%B3n%20no%20tiene%20mucho%20sentido.. [Último acceso: 31 Octubre 2023].

[2] K. T. «¿Qué es fork en Git?,» keepcoding, 20 Julio 2022. [En línea]. Available: https://keepcoding.io/blog/que-es-fork-en-git/. [Último acceso: 31 Octubre 2023].

[3] J. Holcombe, «Git Avanzado: Comandos Avanzados Además de los Básicos,» Kinsta, 21 Agosto 2023. [En línea]. Available: https://kinsta.com/es/blog/git-avanzado/. [Último acceso: 31 Octubre 2023].

[4] J. Carrillo, «Git Stash Explicado: Cómo Almacenar Temporalmente los Cambios Locales en Git,» freecodecamp, 6 Febrero 2021. [En línea]. Available: https://www.freecodecamp.org/espanol/news/git-stash-explicado/. [Último acceso: 31 Octubre 2023].

[5] «Git Clean: limpiar tu proyecto de archivos no deseados,» Platzi, [En línea]. Available: https://platzi.com/clases/1557-git-github/19983-git-clean-limpiar-tu-proyecto-de-archivos-no-desea/. [Último acceso: 31 Octubre 2023].

[6] L. Burgos, «¿Cómo usar git cherry-pick sin morir en el intento?,» Medium, 13 Octubre 2020. [En línea]. Available: https://luisburgosv.medium.com/c%C3%B3mo-usar-git-cherry-pick-sin-morir-en-el-intento-fab92ba1ee7b. [Último acceso: 31 Octubre 2023].

[7] Sebastian, «Rewriting history with git filter-branch,» sebastian-feldmann, 21 Noviembre 2019. [En línea]. Available: https://sebastian-feldmann.info/rewriting-history-with-git-filter-branch/. [Último acceso: 31 Octubre 2023].

[8] «Git Reset y Reflog: úsese en caso de emergencia,» Platzi, [En línea]. Available: https://platzi.com/clases/1557-git-github/19988-git-reset-y-reflog-usese-en-caso-de-emergencia/. [Último acceso: 31 Octubre 2023].