Módulo 2. Git. Control de versiones con Git. Ramas, commits y buenas prácticas

CÓRDOBA CREA FUTURO

Introducción

Git es un sistema de control de versiones distribuido (DVCS) que permite rastrear los cambios en archivos de *software*, posibilita que múltiples desarrolladores colaboren en un proyecto, fusionen modificaciones de forma eficiente y regresen a versiones anteriores cuando sea necesario. Actúa como un historial detallado del proyecto y funciona de manera local, sin necesidad de conexión a Internet para la mayoría de las operaciones. Además, facilita la organización de proyectos, la gestión de versiones, el trabajo en equipo sin conflictos y la integridad del código mediante copias instantáneas y seguras.

Entre sus principales utilidades se encuentran las siguientes:

- Mantener un historial de cambios en los proyectos.
- Facilitar el trabajo en equipo sin sobrescribir código.
- Revertir cambios en caso de errores.
- Implementar metodologías como GitFlow.

En otras palabras, el control de versiones distribuido que ofrece Git implica que un clon local del proyecto constituye un repositorio completo. Estos repositorios locales plenamente funcionales permiten trabajar sin conexión o de forma remota con facilidad. Los desarrolladores confirman su trabajo de manera local y posteriormente sincronizan la copia del repositorio con la del servidor. Este modelo difiere del control de versiones centralizado, en el que es necesario sincronizar el código con un servidor antes de crear nuevas versiones.

¿Qué es Git?

Distribuido (DVCS). —

Cada desarrollador posee una copia completa del repositorio.

Basado en instantáneas (snapshots): —

en cada *commit* se guarda el estado completo del proyecto, no solo las diferencias.

Trabajo sin conexión: —

la mayoría de las operaciones se realizan de forma local.

Seguridad e integridad: —

utiliza un sistema de hash criptográfico (SHA1) para garantizar la integridad del código.

Además del control de versiones, Git facilita la colaboración entre desarrolladores. Permite trabajar en paralelo mediante el uso de ramas (*branches*), donde pueden desarrollarse nuevas funcionalidades sin afectar la versión principal. Posteriormente, estos cambios se integran mediante procesos de fusión (*merge*).

En la práctica, algunas acciones habituales son:

- crear una nueva rama con `git checkout -b nueva-funcionalidad`;
- preparar cambios con `git add archivo.txt`;
- confirmar cambios con `git commit -m "Descripción del cambio"`;

- enviar cambios a un repositorio remoto con `git push`.
- actualizar la copia local con `git pull`.

Por su flexibilidad y potencia, Git se ha consolidado como un estándar en el desarrollo de software y es compatible con la mayoría de las plataformas.

CONTINUAR

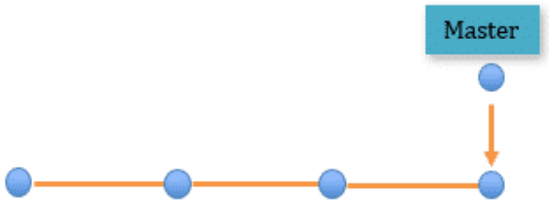
1. Aspectos básicos de Git

Cada vez que se guarda el trabajo, Git crea una confirmación (*commit*). Una confirmación es una instantánea (*snapshot*) de todos los archivos en un momento determinado. Si un archivo no ha cambiado entre una confirmación y la siguiente, Git reutiliza la versión previamente almacenada. Este diseño difiere de otros sistemas que almacenan una versión inicial del archivo y registran únicamente las diferencias a lo largo del tiempo.

Las confirmaciones se enlazan entre sí, formando un gráfico que representa el historial de desarrollo. Esto permite revertir el código a una confirmación anterior, examinar los cambios entre distintas confirmaciones y consultar información como el momento y el autor de cada modificación.

Cada confirmación se identifica mediante un hash criptográfico único generado a partir de su contenido. Gracias a este mecanismo, cualquier alteración en la información puede detectarse, lo que garantiza la integridad del repositorio.

Figura 1. Representación gráfica de confirmaciones en la rama principal (*master*)



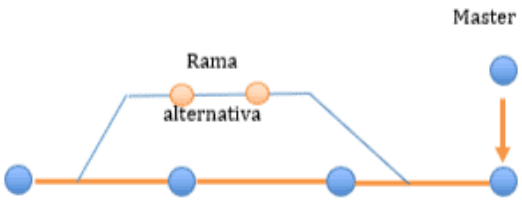
Fuente: elaboración propia

Ramas

Cada desarrollador guarda los cambios en su propio repositorio local. Como consecuencia, pueden generarse múltiples líneas de trabajo a partir de una misma confirmación. Git ofrece herramientas que permiten aislar esos cambios y, posteriormente, integrarlos nuevamente.

Las ramas son punteros ligeros que señalan una confirmación determinada y permiten desarrollar nuevas funcionalidades sin afectar la línea principal del proyecto. Una vez finalizado el trabajo en una rama, los cambios pueden fusionarse con la rama principal del equipo.

Figura 2. Representación gráfica de una rama alternativa y su posterior integración en la rama principal (*master*)



Fuente: elaboración propia

En Git, los archivos pueden encontrarse en tres estados: modificados (*modified*), almacenados en el área de preparación (*staged*) o confirmados (*committed*).

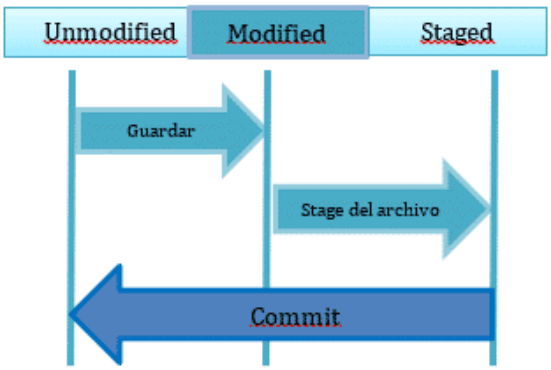
Cuando un archivo se modifica por primera vez, los cambios existen únicamente en el directorio de trabajo. Aún no forman parte de una confirmación ni del historial del proyecto. Para incluirlos en la siguiente confirmación, es necesario pasarlos al área de almacenamiento provisional (*staging area*). Esta área reúne todos los cambios que se incorporarán en el próximo *commit*.

Una vez revisados los archivos almacenados provisionalmente, se crea la confirmación, acompañada de un mensaje que describe las modificaciones

realizadas. A partir de ese momento, la confirmación pasa a integrar el historial del proyecto.

El almacenamiento provisional permite seleccionar de manera precisa qué cambios se incluirán en cada confirmación. Esto facilita dividir modificaciones extensas en confirmaciones más pequeñas y organizadas, lo que mejora la revisión del historial y la localización de cambios específicos.

Figura 3. Estados de los archivos en Git y flujo hacia la confirmación



Fuente: elaboración propia

Git ofrece múltiples ventajas para el desarrollo de software, especialmente cuando se trabaja en equipo y se gestionan distintas versiones de un mismo proyecto.

- **Desarrollo simultáneo**. Todos los usuarios disponen de su propia copia local del código y pueden trabajar simultáneamente en sus propias ramas. Git funciona sin conexión para la mayoría de las operaciones, ya que estas se realizan de forma local.
- **Versiones de lanzamiento más rápidas**. Las ramas permiten un desarrollo flexible y paralelo. La rama principal contiene el código estable y de calidad desde el cual se publica. Las ramas de características contienen trabajo en

curso y se combinan con la rama principal una vez finalizadas. Al separar la versión estable del desarrollo activo, se facilita la administración del código y la publicación de actualizaciones con mayor rapidez

- **Integración incorporada**. Debido a su amplia adopción, Git se integra con la mayoría de las herramientas y productos. Los principales entornos de desarrollo ofrecen compatibilidad integrada, y muchas herramientas permiten la integración e implementación continuas, pruebas automatizadas y seguimiento de tareas. Esta integración simplifica el flujo de trabajo diario.
- **Sólido soporte de la comunidad**. Git es de código abierto y se ha consolidado como estándar en el control de versiones, contando con una comunidad activa que aporta documentación y soporte.
- **Git funciona con cualquier equipo**. El uso de Git junto con una herramienta de administración de código fuente puede aumentar la productividad del equipo al fomentar la colaboración, aplicar directrices, automatizar procesos y mejorar la visibilidad y la trazabilidad del trabajo. Puede utilizarse con herramientas independientes o con plataformas que centralizan estas funciones en un mismo entorno.
- **Directivas de rama**. Las plataformas como GitHub permiten configurar reglas para proteger ramas importantes. Estas directivas pueden impedir inserciones directas, exigir revisiones previas o verificar que las compilaciones se completen correctamente antes de aprobar la integración.

Figura 4. Infografía explicativa sobre el funcionamiento básico de Git y el trabajo con ramas

Fuente: ED Team, 2019, https://goo.su/USPpCJ

Las solicitudes de incorporación de cambios permiten analizar el código con el equipo antes de integrarlo en la rama principal. Las discusiones que se generan en este proceso resultan valiosas para garantizar la calidad del código y ampliar los conocimientos dentro del equipo. Plataformas como GitHub ofrecen un entorno que facilita este procedimiento, ya que permiten revisar modificaciones en los archivos, dejar comentarios, inspeccionar confirmaciones, visualizar compilaciones y aprobar los cambios antes de su integración. Las solicitudes de incorporación de cambios integran la revisión y la combinación de código en un único proceso colaborativo. Cuando un desarrollador añade una nueva funcionalidad o corrige un error, crea una solicitud para iniciar el proceso de integración en la rama principal. Posteriormente, otros miembros del equipo tienen la oportunidad de revisar y aprobar el código antes de que se complete la combinación. Asimismo, estas solicitudes pueden utilizarse para revisar trabajos

en curso y obtener comentarios sobre los cambios, sin que exista la obligación de integrarlos. En cualquier momento, el responsable puede cancelar la solicitud si lo considera necesario.

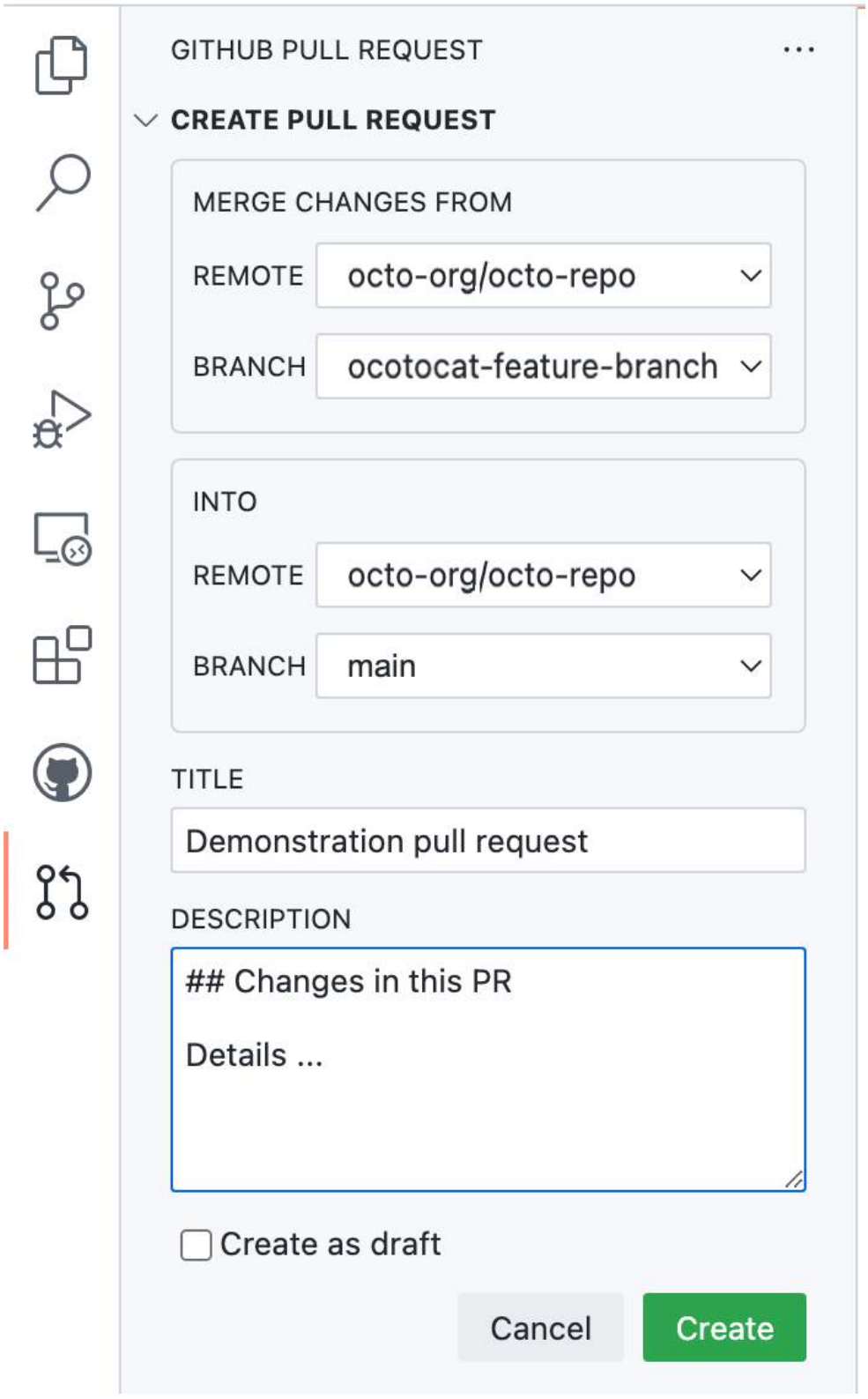
Obtención de revisiones de código

La revisión de código realizada como parte de una solicitud de incorporación de cambios no se limita a detectar errores evidentes, ya que esa función corresponde principalmente a las pruebas. Una revisión adecuada permite identificar problemas menos visibles que podrían generar dificultades mayores en etapas posteriores.

Las revisiones contribuyen a evitar combinaciones incorrectas y compilaciones fallidas que afectan la productividad del equipo. Al detectar inconvenientes antes de la integración, se protegen las ramas importantes frente a cambios no deseados.

Asimismo, fomentan la colaboración y la comunicación entre los desarrolladores, y permiten contar con un registro claro de las modificaciones realizadas entre la rama principal y las ramas de trabajo.

Figura 5. Creación de una solicitud de incorporación de cambios en GitHub



Fuente: GitHub, s.f., https://goo.su/ncACIzn

Las revisiones de calidad comienzan con comentarios bien fundamentados. Para obtener retroalimentación valiosa en una solicitud de incorporación de cambios, conviene tener en cuenta los siguientes aspectos:

- Solicitar la revisión a las personas adecuadas.
- Asegurarse de que los revisores comprendan qué hace el código.
- Proporcionar comentarios útiles y constructivos.
- Responder a las observaciones de manera oportuna.

Al asignar revisores, es importante seleccionar a quienes conozcan el funcionamiento del código, pero también puede resultar enriquecedor incluir desarrolladores de otras áreas, ya que pueden aportar perspectivas diferentes.

Asimismo, debe proporcionarse una descripción clara de los cambios y una compilación en la que la corrección o la nueva funcionalidad pueda observarse en funcionamiento. Los revisores, por su parte, deberían formular comentarios concretos cuando no estén de acuerdo con alguna decisión, identificar el problema y proponer alternativas específicas. Este tipo de observaciones facilita la comprensión y mejora el proceso de colaboración.

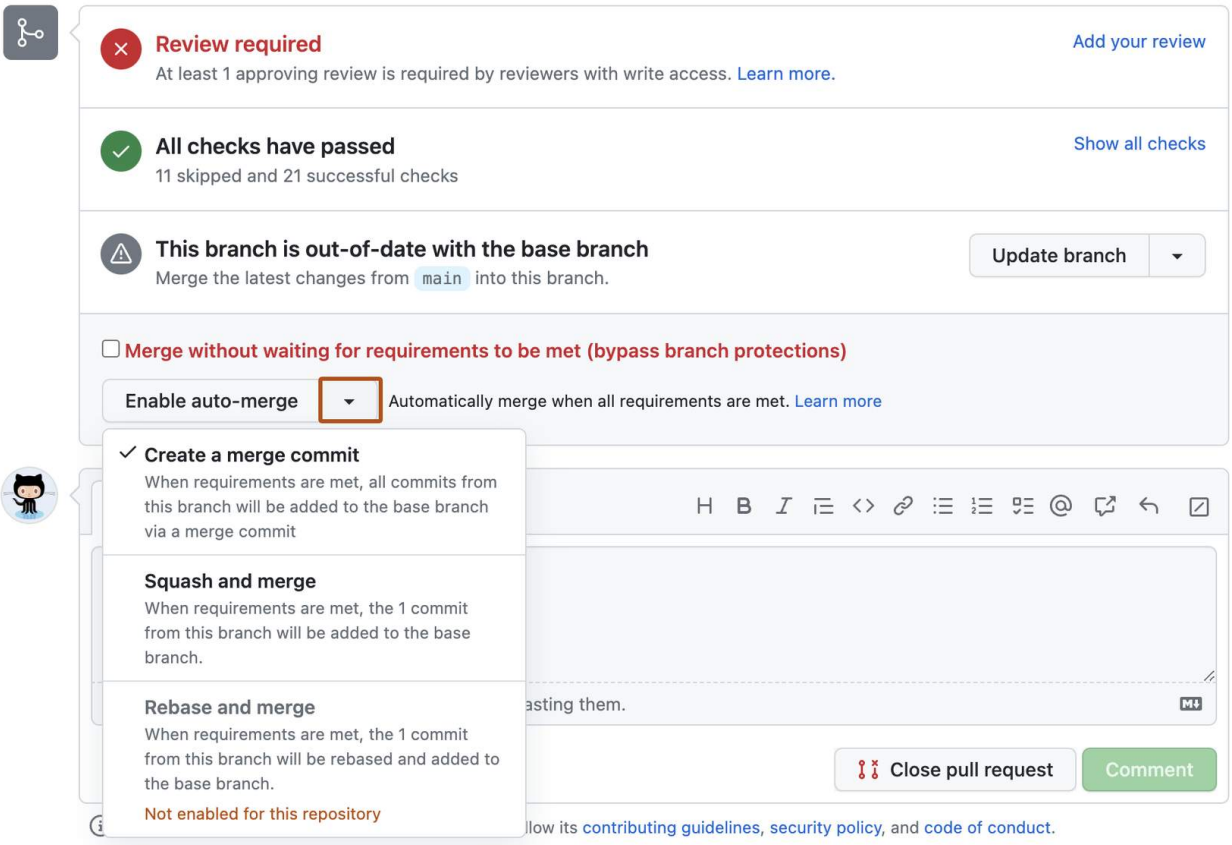
El responsable de la solicitud debe responder a los comentarios, aceptar sugerencias o explicar las razones por las que decide no aplicarlas. En algunos casos, ciertas propuestas pueden ser pertinentes, pero exceder el alcance de la solicitud. En tales situaciones, es recomendable convertirlas en nuevos elementos de trabajo y desarrollarlas en ramas independientes.

Existen ramas críticas en un repositorio, como la rama main, que los equipos procuran mantener siempre en buen estado. Por ello, en plataformas como GitHub, suele requerirse una solicitud de incorporación de cambios para realizar modificaciones en estas ramas.

Los desarrolladores que intenten insertar cambios directamente en las ramas protegidas verán rechazados sus envíos.

Asimismo, pueden añadirse condiciones adicionales a las solicitudes de incorporación de cambios para aplicar un mayor nivel de control y calidad en las ramas clave.

Figura 6. Configuración de requisitos y opciones de fusión en una solicitud de incorporación de cambios en GitHub



Fuente: GitHub, s.f.a., https://goo.su/UEIpdqc

CONTINUAR

2. Instalación de Git

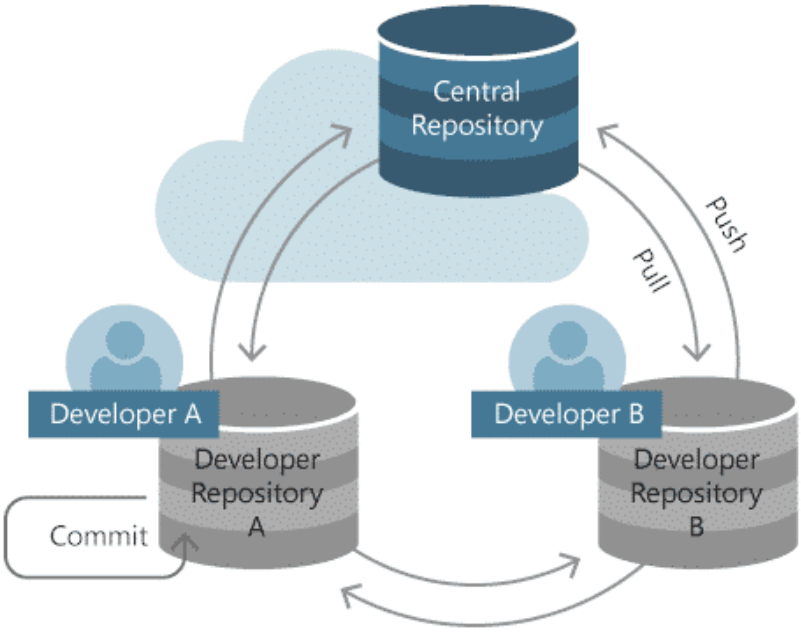
Un repositorio de Git es una carpeta en la que se realiza el seguimiento de los cambios de un proyecto. En un mismo equipo pueden existir varios repositorios, cada uno almacenado en su propia carpeta y funcionando de manera independiente. Por ello, los cambios realizados en un repositorio no afectan a los demás.

Cada repositorio contiene todas las versiones de los archivos que forman parte del proyecto. A diferencia de otros sistemas de control de versiones que almacenan únicamente las diferencias entre archivos, Git guarda instantáneas completas. Esta información se almacena en una carpeta oculta denominada .git, junto con otros datos necesarios para la gestión del código. El almacenamiento se realiza de manera eficiente, por lo que conservar múltiples versiones no implica un uso excesivo de espacio en disco.

El trabajo con Git se realiza mediante comandos ejecutados en el repositorio local del equipo. Incluso cuando se comparten cambios o se reciben actualizaciones, las acciones se aplican primero al repositorio local. Este enfoque centrado en el entorno local es lo que define a Git como un sistema de control de versiones distribuido: cada repositorio es autónomo y su responsable debe mantenerlo sincronizado con los cambios realizados por otros integrantes del equipo.

En la práctica, la mayoría de los equipos utiliza un repositorio central alojado en un servidor al que todos pueden acceder para coordinar su trabajo. Este repositorio suele hospedarse en plataformas de administración de código fuente, como GitHub o Azure DevOps, que incorporan funcionalidades adicionales para facilitar la colaboración.

Figura 7. Modelo de repositorio central con repositorios locales de desarrolladores



Fuente: Microsoft, s.f., https://goo.su/Gw3xyfc

Para descargar Git, se debe acceder al sitio oficial correspondiente al sistema operativo:

- Windows: https://git-scm.com/downloads/win

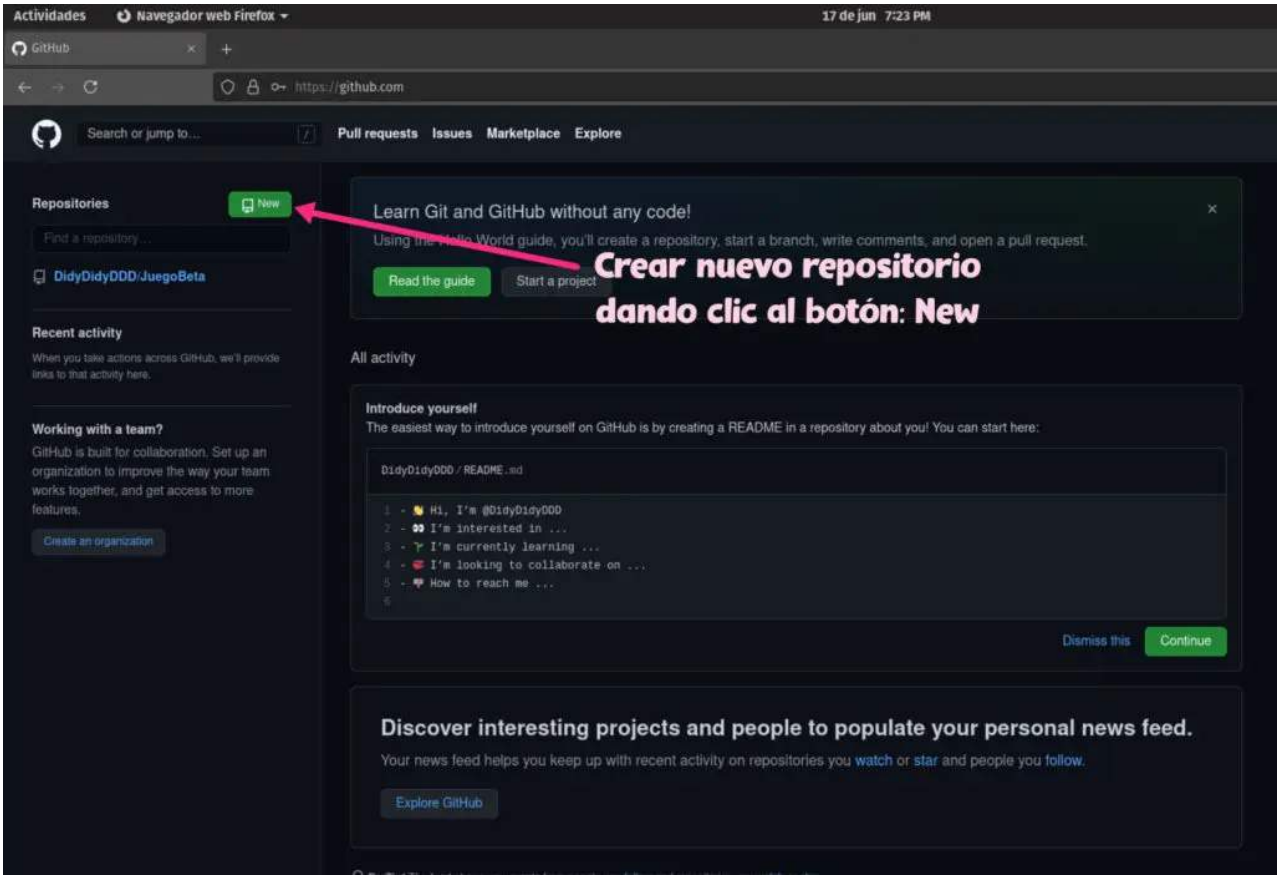
- MacOS o Linux: https://learn.microsoft.com/es-es/devops/develop/git/install-and-set-up-git

Instalación de Git

Existen dos formas de crear un repositorio en Git: puede generarse a partir del código almacenado en una carpeta del equipo local o puede clonarse desde un repositorio ya existente.

Si el código se encuentra únicamente en el equipo, se debe crear un repositorio local dentro de esa carpeta para que Git comience a realizar el seguimiento de los cambios. Sin embargo, en la mayoría de los casos el proyecto ya está alojado en un repositorio remoto, por lo que resulta recomendable clonarlo en el equipo local para comenzar a trabajar.

Figura 8. Creación de un nuevo repositorio en GitHub mediante el botón «New»



Fuente: Ney, s.f., https://goo.su/hz94N

Crear un nuevo repositorio a partir de código existente

Para crear un nuevo repositorio desde una carpeta existente en el equipo, se utiliza el siguiente comando:

```
git init
```

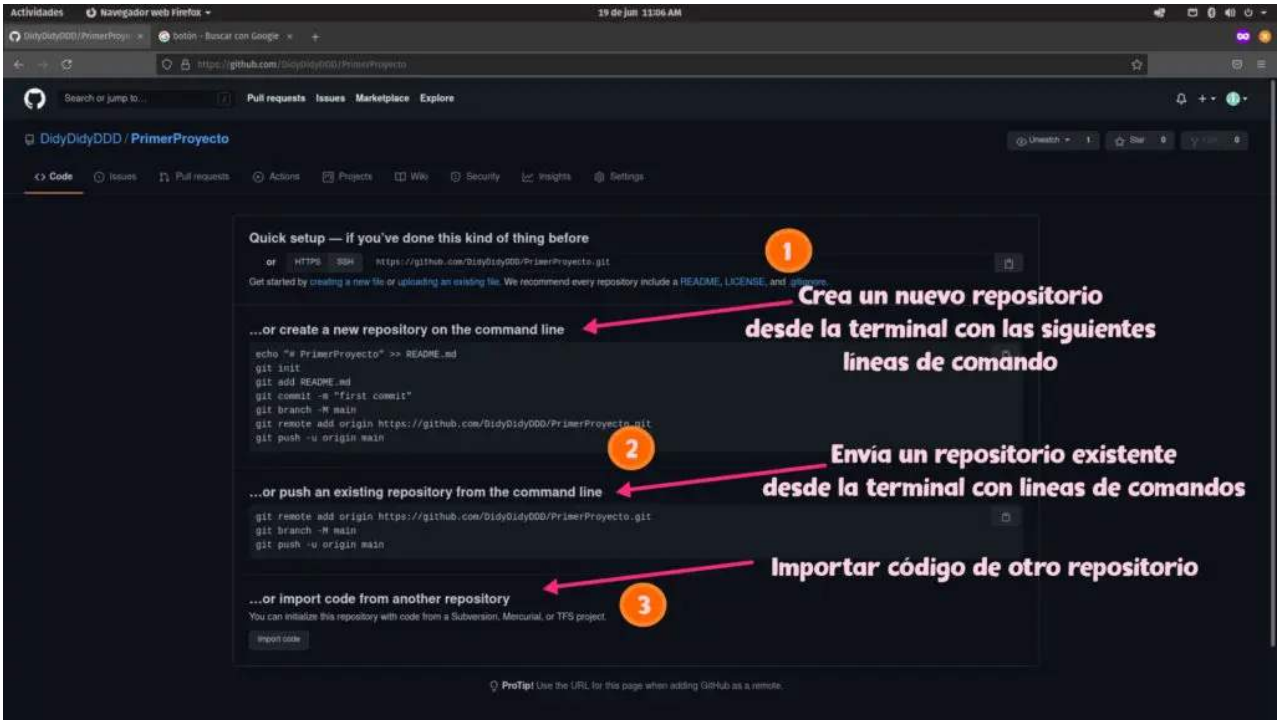
Este comando debe ejecutarse en la carpeta raíz que contiene el código. Con ello, se crea el repositorio local. A continuación, se agregan los archivos a la primera confirmación mediante:

`git add --all`

`git commit -m "Initial commit"`

En la siguiente imagen se muestran las opciones disponibles desde la interfaz de GitHub para crear o importar un repositorio.

Figura 9. Opciones para crear o importar un repositorio desde la interfaz de GitHub



Fuente: Ney, s.f., https://goo.su/hz94N

Crear un nuevo repositorio a partir de un repositorio remoto

Para copiar el contenido de un repositorio existente en el equipo local, se utiliza el comando:

```
git clone https://xxxxxxxx
```

Cuando se clona un repositorio, se copia todo su contenido junto con el historial completo desde el repositorio original.

En la línea de comandos, debe ubicarse en la carpeta donde se desea descargar el repositorio y ejecutar:

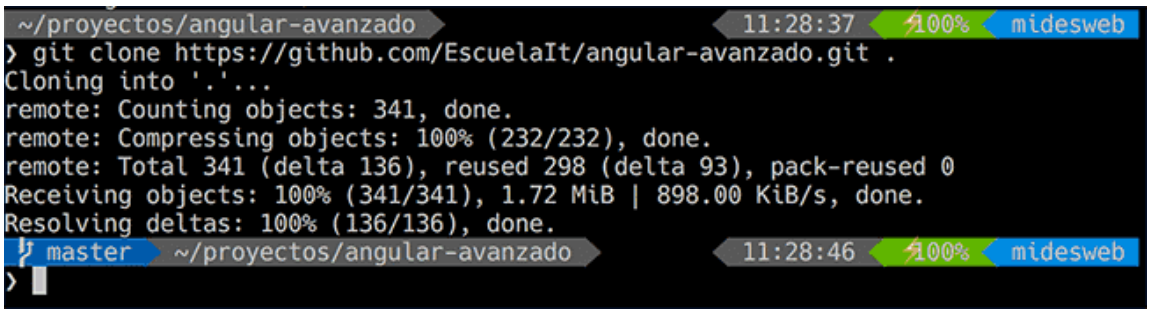
```
git clone https://<fabrikam.visualstudio.com/DefaultCollection/Fabrikam/_git/FabrikamProject>
```

Es importante utilizar la dirección URL real del repositorio existente. Esta dirección, denominada URL de clonación, apunta al servidor donde el equipo coordina los cambios. Puede obtenerse desde el equipo de trabajo o mediante el botón correspondiente en el sitio donde se aloja el repositorio.

No es necesario agregar archivos ni crear una confirmación inicial al clonar el repositorio, ya que durante la operación de clonación se copia

todo el contenido junto con su historial.

Figura 10. Estado del proceso de clonación de un repositorio en la línea de comandos



Fuente: Desarrollo Web, 2018, https://goo.su/LVeSMZ

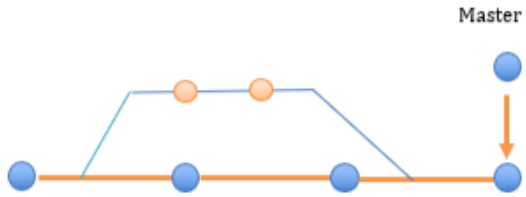
Guardar y compartir código: Git

Guardar y compartir versiones de código con un equipo son acciones habituales al utilizar un sistema de control de versiones. Git propone un flujo de trabajo sencillo para estas tareas, que se compone de tres pasos:

- Crear una nueva rama para el trabajo.
- Realizar la confirmación (*commit*) de los cambios.
- Insertar la rama en el repositorio remoto para compartirla con el equipo.

Git facilita la organización del trabajo mediante el uso de ramas. Cada corrección de errores, nueva funcionalidad, prueba incorporada o ajuste en la configuración puede desarrollarse en una rama independiente. Estas ramas son ligeras y locales al equipo de desarrollo, por lo que no requieren coordinación inmediata con otros usuarios hasta el momento en que se decide insertarlas en el repositorio compartido.

Figura 11. Flujo de trabajo con ramas y confirmaciones en la rama principal (*master*)



Fuente: elaboración propia

Working, Staging, Commit

En Git, los archivos atraviesan tres estados principales: modificado, preparado y confirmado.

Modificado (*working directory*)

Es el estado en el que se encuentra un archivo después de realizar cambios. Git detecta que el archivo ha sido modificado, pero aún no ha sido agregado al área de preparación. En esta etapa, los cambios solo existen en el directorio de trabajo.

Preparado (*staging area*)

El área de preparación es una zona intermedia donde se almacenan temporalmente los cambios que se incluirán en la próxima confirmación. Cuando un archivo modificado se agrega mediante el comando `git add nombre_archivo` pasa del directorio de trabajo al área de preparación y queda listo para formar parte del siguiente *commit*.

Confirmado (*repository*)

Una vez que los archivos están en el área de preparación, se utiliza el comando `git commit -m "Mensaje descriptivo"` para crear una confirmación. En ese

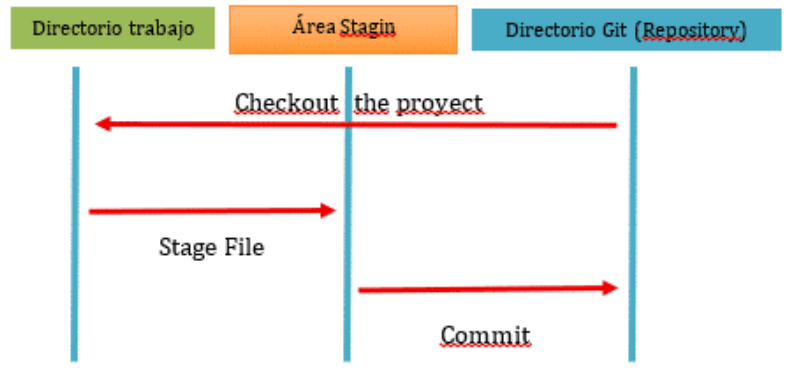
momento, Git genera una instantánea de los cambios preparados y la guarda en el repositorio local, construyendo así el historial de versiones del proyecto.

El flujo básico de trabajo en Git puede resumirse de la siguiente manera:

- Se modifica un archivo en el directorio de trabajo.
- Se utiliza el comando `git add` para mover el archivo modificado al área de preparación (*staging area*).
- Se emplea el comando `git commit` para crear una confirmación y guardar los cambios preparados en el repositorio local.

Este proceso permite registrar los cambios de forma organizada y construir el historial del proyecto.

Figura 12. Flujo entre directorio de trabajo, área de preparación y repositorio en Git



Fuente: elaboración propia

Comandos Git —

Git dispone de una amplia variedad de comandos que permiten gestionar repositorios, ramas y confirmaciones. A continuación, se presenta un listado con algunos de los comandos más utilizados y su propósito principal.

Tabla 1. Comandos básicos de Git

Comando	Propósito	Uso básico
`git init`	Inicializa un nuevo repositorio en el directorio actual.	`git init`
`git clone`	Crea una copia local de un repositorio existente.	`git clone <repository_url>`
`git add`	Prepara los cambios para la próxima confirmación.	`git add <file>` o `git add .`
`git commit`	Guarda los cambios preparados con un mensaje descriptivo.	`git commit -m "Mensaje"`
`git status`	Muestra el estado de los archivos del directorio de trabajo.	`git status`
`git push`	Envía confirmaciones al repositorio remoto.	`git push origin <branch_name>`
`git pull`	Obtiene y fusiona cambios desde un repositorio remoto.	`git pull`
`git branch`	Lista, crea o elimina ramas.	`git branch` o `git branch <branch_name>`
`git checkout`	Cambia de rama o restaura archivos.	`git checkout <branch_name>`
`git merge`	Fusiona una rama con la rama actual.	`git merge <branch_name>`

`git log`	Muestra el historial de confirmaciones.	`git log`
`git diff`	Compara diferencias entre archivos, confirmaciones o ramas.	`git diff`

Fuente: elaboración propia

En la siguiente tabla se resumen otros comandos de uso frecuente que amplían las funcionalidades de Git.

Tabla 2. Comandos adicionales de Git

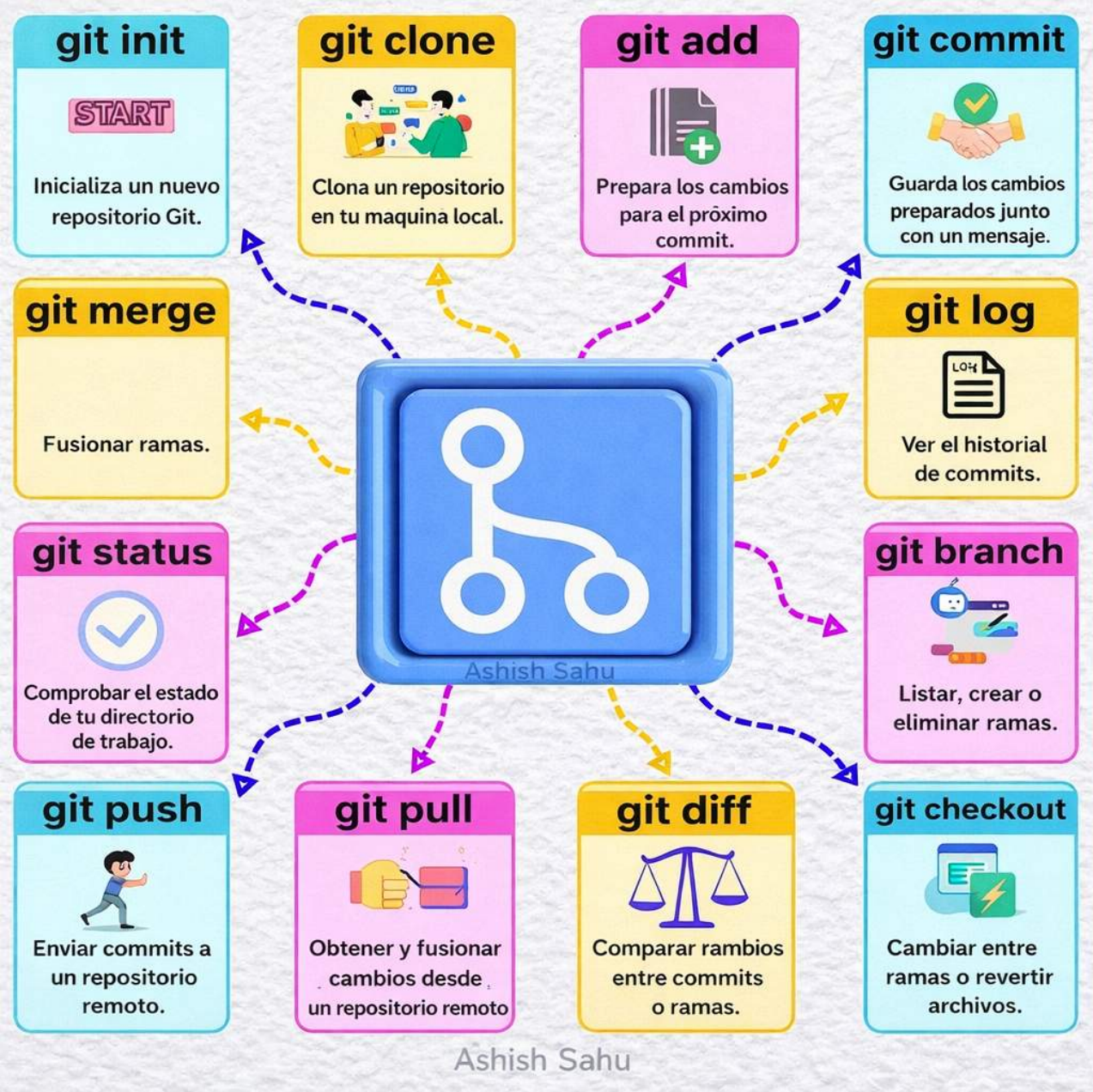
Comando	Descripción
`git fetch`	Recupera actualizaciones de un repositorio remoto sin fusionarlas.
`git rebase`	Reaplica confirmaciones sobre otra confirmación base.
`git reset`	Deshace cambios en el directorio de trabajo o en el historial.
`git stash`	Guarda cambios temporalmente para utilizarlos más adelante.

`git tag`	Marca puntos específicos en el historial con una etiqueta.
`git remote`	Administra conexiones a repositorios remotos.
`git show`	Muestra información detallada sobre una confirmación.
`git revert`	Crea una nueva confirmación que deshace una anterior.
`git blame`	Indica quién modificó cada línea de un archivo.
`git cherry-pick`	Aplica confirmaciones específicas de otra rama.
`git rm`	Elimina archivos del directorio de trabajo y del área de preparación.
`git archive`	Genera un archivo comprimido del proyecto.

Fuente: elaboración propia

El siguiente esquema presenta de manera visual algunos de los comandos más utilizados en Git y su función dentro del trabajo diario con repositorios.

Figura 13. Comandos básicos de Git y su propósito



Fuente: elaboración propia

¿git merge o git rebase?

Aunque la diferencia puede parecer técnica, ambos comandos permiten combinar ramas, pero lo hacen de formas distintas. Veamos esta diferencia con un ejemplo.

Escenario inicial

Se tienen dos ramas:

`main: A → B → C → D`

`feature: E → F → G (basados en B)`

Se desea integrar los cambios de la rama `feature` en *main*. Existen dos formas principales de hacerlo.

Opción 1: `git merge`

`git checkout main`

`git merge feature`

Al utilizar `git merge`, se crea un nuevo *commit* de fusión que une ambas ramas. El historial conserva las dos líneas de trabajo tal como se desarrollaron y permite identificar con claridad el punto exacto en el que se produjo la integración. Este enfoque resulta especialmente adecuado cuando se trabaja en equipo y se desea mantener el contexto real del desarrollo, ya que ofrece trazabilidad sobre el momento en que una rama fue incorporada

a otra. Como desventaja, el historial puede volverse más ramificado y, si existen muchas fusiones paralelas, su lectura puede resultar más compleja.

Opción 2: `git rebase`

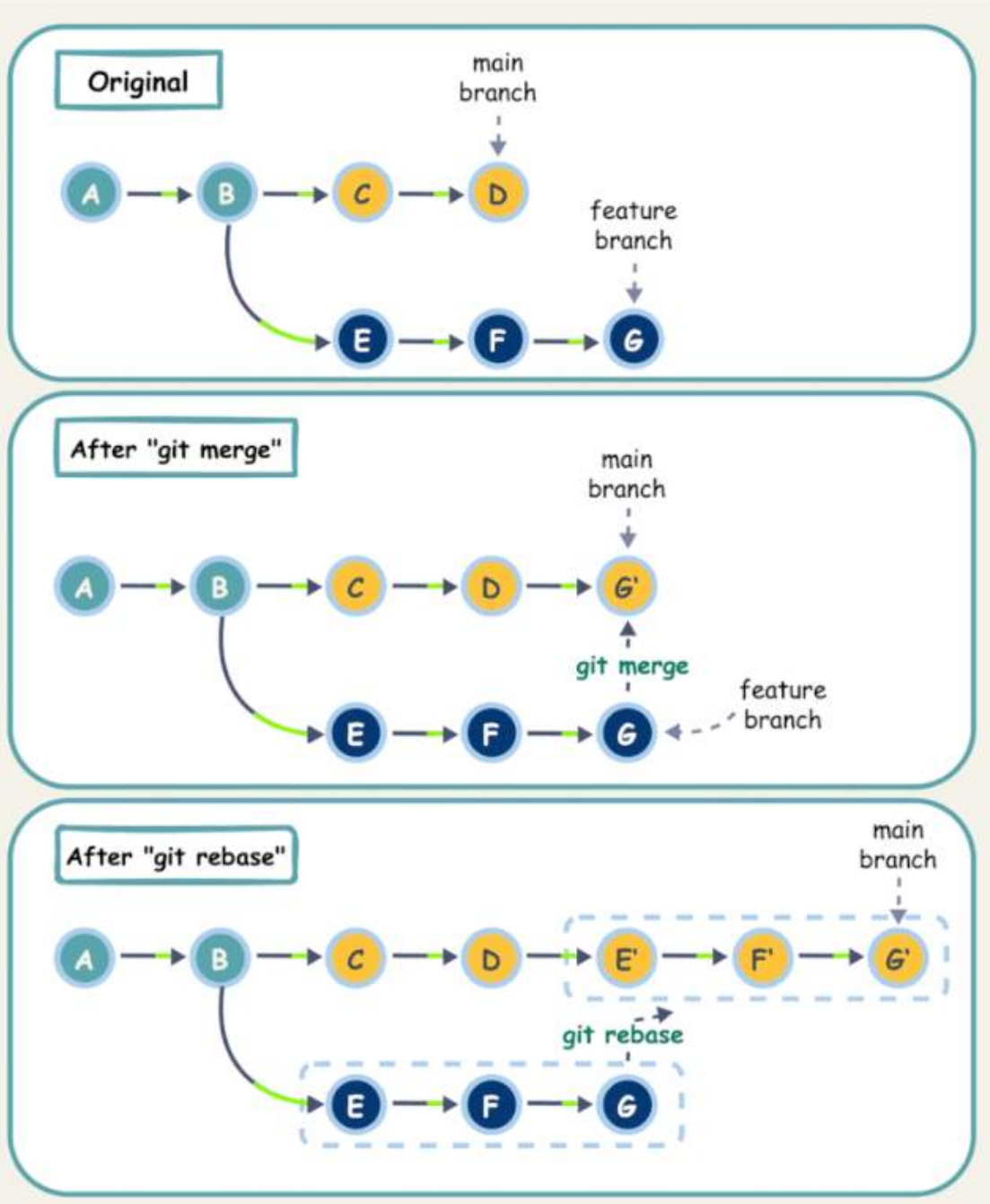
```
git checkout feature
```

```
git rebase main
```

Al utilizar `git rebase`, los *commits* `E → F → G` se reubican y se colocan después de D. El resultado es un historial lineal y más limpio, como si los cambios de la rama *feature* se hubieran creado posteriormente a los de *main*. Este enfoque resulta conveniente cuando se trabaja de manera individual en una rama, antes de abrir una solicitud de incorporación de cambios o cuando se prefiere una secuencia clara y ordenada del desarrollo. Sin embargo, como desventaja, debe tenerse en cuenta que `git rebase` reescribe los *commits*, por lo que no se recomienda utilizarlo si otras personas están trabajando sobre la misma rama, ya que podría generar conflictos o alterar el contexto original del trabajo.

En síntesis, `git merge` conserva el historial tal como ocurrió, mientras que `git rebase` reorganiza la historia para que resulte más lineal. Ambos enfoques son válidos; la elección depende del contexto en el que se utilicen.

Figura 14. Git merge vs. git rebase



Fuente: Durán García, 2024, https://goo.su/jzlJ7

CONTINUAR

Referencias

Desarrollo Web, (2018). *Clonar un repositorio: Git clone*. https://desarrolloweb.com/articulos/git-clone-clonar-repositorio.html

ED Team, (2019). *¿Sabes qué es Git?* https://ed.team/comunidad/sabes-que-es-git

GitHub, (s.f.). *Crear una solicitud de incorporación de cambios*. https://docs.github.com/es/enterprise-cloud@latest/pull-requests/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/creating-a-pull-request

GitHub, (s.f.a.). *Fusionar una solicitud de cambios automáticamente*. https://docs.github.com/es/enterprise-cloud@latest/pull-requests/collaborating-with-pull-requests/incorporating-changes-from-a-pull-request/automatically-merging-a-pull-request

Microsoft, (s.f.). *Configuración de un repositorio de Git*. https://learn.microsoft.com/es-es/devops/develop/git/set-up-a-git-repository

Ney, (s.f.). *¿Cómo crear un repositorio en GitHub desde Linux?* https://ney.one/como-crear-un-repositorio-en-github-desde-linux/

Durán García, M. (2024). La diferencia entre git merge y git rebase. *LinkedIn*. https://www.linkedin.com/posts/midudev_la-diferencia-entre-git-merge-y-git-rebase-activity-7163890483402940416-Zb8Y/?originalSubdomain=es

Referencias bibliográficas de consulta

Atlassian, (s.f.). *Usa Git correctamente. Aprende a usar Git con tutoriales, noticias y consejos.* https://www.atlassian.com/es/git

Chacon, S., & Straub, B. (2014). *Pro Git* (2nd ed.). Apress. https://git-scm.com/book/es/v2

Chacon, S., & Straub, B. (2014). *Pro Git* (edición en español). https://archive.org/details/2014ProGitEs

Rodríguez, D. (2024). *Mejores prácticas en Git. Analytics Lane.* https://www.analyticslane.com/2024/11/20/mejores-practicas-en-git/

CONTINUAR