¿Qué es Git?

Git es una herramienta que ayuda en el control de las diferentes versiones que se generan durante las diferentes etapas que existen en el proceso de desarrollo de software, fue creado en 2005 por Linus Torvalds y permite a los desarrolladores a controlar y administrar los cambios en el código fuente a lo largo del tiempo.

Unos de los principales objetivos del uso de herramientas de control de versiones (como Git) es rastrear los cambios que surgen en el código.

¿Cómo funciona Git?

Git maneja sus datos como un conjunto de copias instantáneas de un sistema de archivos miniatura. Cada vez que se confirma un cambio, o se guarda el estado de un proyecto en Git, él básicamente toma una foto del aspecto de todos los archivos en ese momento y guarda una referencia a esa copia instantánea. Para ser eficiente, si los archivos no se han modificado, Git no almacena el archivo de nuevo, sino un enlace al archivo anterior idéntico que ya tiene almacenado. Git maneja sus datos como una secuencia de copias instantáneas.

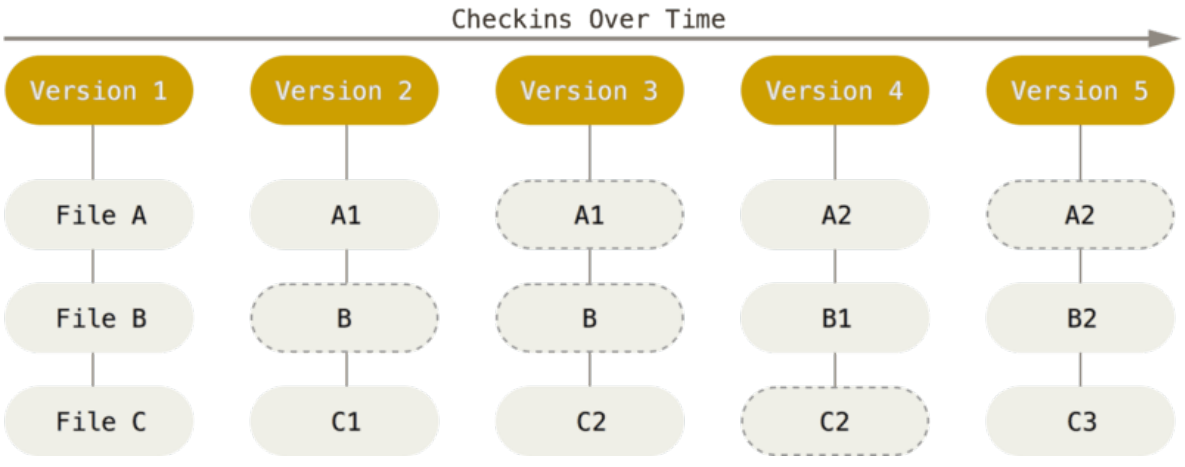


Figura 1. Almacenamiento de datos como instantáneas del proyecto a través del tiempo.

Una de las muchas ventajas de Git es que cuenta con una arquitectura distribuída, lo que significa que cada desarrollador tiene una copia local del repositorio, esto permite trabajar sin conexión y tener todos los cambios en el equipo local, para posteriormente fusionar los cambios con el repositorio principal.

Comandos básicos en Git

El uso de Git es muy fácil, para poder manejar Git existen comandos para la manipulación de las acciones, algunos de los comandos esenciales son:

- Iniciamos GIT en la carpeta donde esta el proyecto

  ```
  git init
  ```

- Clonamos el repositorio de github o bitbucket

  ```
  git clone <url>
  ```

- Añadimos todos los archivos para el commit

  ```
  git add .
  ```

- Hacemos el primer commit

  ```
  git commit –m "Texto que identifique por que se hizo el commit"
  ```

- subimos al repositorio

  ```
  git push origin master
  ```

- Añadimos todos los archivos para el commit

  ```
  git add .
  ```

- Cargar en el HEAD los cambios realizados

  ```
  git commit –m "Texto que identifique por que se hizo el commit"
  ```

- Agregar y Cargar en el HEAD los cambios realizados

  ```
  git commit –a –m "Texto que identifique por que se hizo el commit"
  ```

- Subimos al repositorio

  ```
  git push <origien> <branch>
  ```

- Muestra los logs de los commits

  ```
  git log
  ```

- Saca un archivo del commit

  ```
  git reset HEAD <archivo>
  ```

- Devuelve el ultimo commit que se hizo y pone los cambios en staging

  ```
  git reset --soft HEAD^
  ```

- Devuelve el ultimo commit y todos los cambios

  ```
  git reset --hard HEAD^
  ```

- Devuelve los 2 ultimo commit y todos los cambios

  ```
  git reset --hard HEAD^^
  ```

- Crea un branch

  ```
  git branch <nameBranch>
  ```

- Lista los branches

  ```
  git branch
  ```

- Comando -d elimina el branch y lo une al master

  ```
  git branch -d <nameBranch>
  ```

- Lista un estado actual del repositorio con lista de archivos modificados o agregados

  ```
  git status
  ```

- Busca los cambios nuevos y actualiza el repositorio

  ```
  git pull origin <nameBranch>
  ```

- Une el branch actual con el especificado

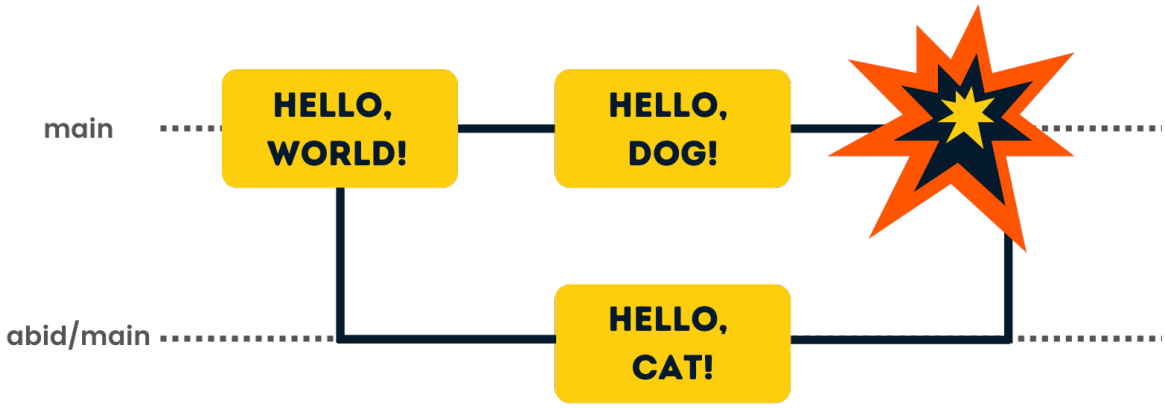
  ```
  git merge <nameBranch>
  ```

- Borrar un archivo del repositorio

  ```
  git rm <archivo>
  ```

Trabajando con repositorios en GitHub (Ramas, Merge, Conflictos) – Generado por medio de ChatGPT

Imaginemos un escenario en el que dos programadores, Ana y Juan, están trabajando en un proyecto hospedado en GitHub. Ambos tienen sus propios repositorios remotos y están trabajando en ramas separadas.



¿Cómo se genera un conflicto?

Ana y Juan trabajan en sus ramas respectivas, por ejemplo, ‘ana-feature’ y ‘juan-feature’.
Ambos hacen cambios en el mismo archivo, digamos ‘app.java’, en líneas cercanas pero no iguales.
Cuando Ana intenta hacer un merge de su rama con la rama principal (‘main’), se produce un conflicto debido a las diferencias en ‘app.java’.

¿Cómo solucionar el conflicto?

Ana descarga los cambios más recientes de la rama principal (‘main’) a su repositorio local.
Ana abre ‘app.java’ y busca las secciones marcadas por Git como conflictivas. Estas secciones mostrarán las diferencias entre su versión y la versión de Juan.
Ana edita manualmente las secciones conflictivas, eligiendo qué cambios mantener y cómo combinarlos correctamente.
Después de resolver los conflictos, Ana agrega los archivos modificados (‘app.java’) al área de preparación (‘git add’) y realiza el commit para completar el merge (‘git commit’).

¿Cómo evitar conflictos a futuro?

Comunicación: Ana y Juan deben comunicarse regularmente sobre las áreas en las que están trabajando para evitar cambios simultáneos en las mismas partes del código.
Ramas pequeñas y frecuentes: Es mejor dividir el trabajo en tareas más pequeñas y crear ramas específicas para cada tarea. Esto reduce la posibilidad de conflictos al fusionar.
Revisión de cambios: Antes de hacer un merge, es útil revisar los cambios realizados por otros colaboradores para anticipar posibles conflictos y resolverlos antes de fusionar las ramas.

Siguiendo estos pasos y prácticas, Ana y Juan pueden trabajar de manera colaborativa en el proyecto, resolver conflictos de manera efectiva y evitar problemas similares en el futuro.

Fuentes de consulta

- **https://www.hostgator.mx/blog/que-es-git/**
- **https://git-scm.com/book/es/v2/Inicio---Sobre-el-Control-de-Versiones-Fundamentos-de-Git**
- **https://www.w3schools.com/git/default.asp**
- **https://www.datacamp.com/tutorial/how-to-resolve-merge-conflicts-in-git-tutorial**
- **https://chat.openai.com**