Introducción a Git

Historia

En 2005, Linus Torvalds, el creador del kernel de Linux, sintió la necesidad de un nuevo sistema de control de versiones para el desarrollo del kernel después de perder el acceso gratuito a BitKeeper. Torvalds buscaba una herramienta que fuera rápida, eficiente, de diseño sencillo, con soporte para desarrollo no lineal, completamente distribuida y capaz de manejar proyectos de gran envergadura. Fue así como condujo a la creación de Git, un sistema de control de versiones distribuido gratuito y de código abierto que revolucionó la comunidad de desarrollo de software.

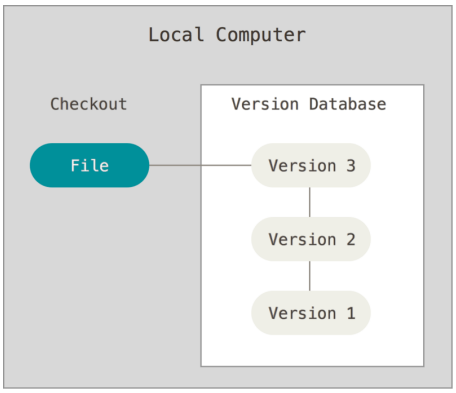
Sistema de Control de Versiones

Un sistema de control de versiones (*Version Control System* - VCS) desempeña un papel crucial en el seguimiento y gestión del historial de cambios en proyectos colaborativos. Ofrece a cada colaborador una visión unificada y consistente del proyecto, destacando un historial transparente de modificaciones, sus autores y cómo contribuyen al desarrollo global.

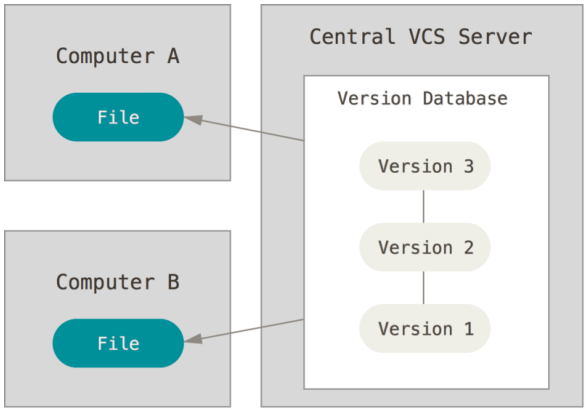
Un VCS registra los cambios de archivos a lo largo del tiempo, permitiendo revertir archivos seleccionados o incluso el proyecto completo a estados anteriores. También facilita la identificación de responsables de cambios que causaron algún problema, proporcionando una plataforma para la recuperación de archivos perdidos.

VCS: Local, Centralizado y Distribuido

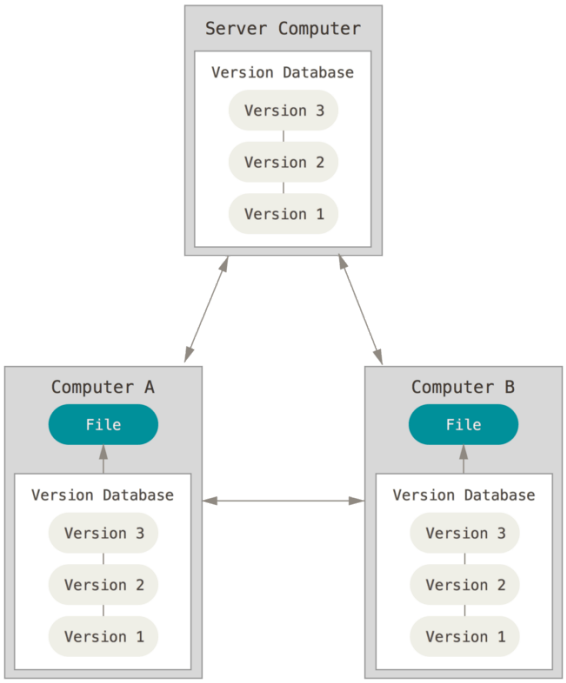
Local. Antes de los VCS, los desarrolladores solían tener una o más copias del proyecto dentro de diferentes directorios. Este era un método simple, pero, sin embargo, es propenso a errores, como confundirse en qué directorio deberías realizar los cambios o equivocarte al modificar o eliminar archivos que no deberían.



Centralizado: Los VCS Centralizados (*Centralized Version Control System* - CVCS) emplean un único servidor que almacena todos los archivos versionados. Los usuarios obtienen copias de los archivos del repositorio central y trabajan en sus copias locales. Sin embargo, la vulnerabilidad radica en la dependencia de este servidor; si este falla, la colaboración se verá comprometida, ya que depende de la conexión constante con el servidor central.



Distribuido: Los VCS distribuidos (*Distributed Version Control System* - DVCS) permiten que cada usuario tenga una copia completa del repositorio, incluyendo el historial completo del proyecto en una máquina local. No hay una dependencia directa de un servidor central para trabajar, por lo que los usuarios pueden modificar de manera independiente en sus repositorios sin la necesidad de una conexión constante al repositorio central. Por otro lado, se pueden sincronizar sus cambios a través de un servidor central compartido o directamente entre los usuarios, permitiendo así una mayor flexibilidad y redundancia en el manejo de versiones, ya que cada copia del repositorio puede actuar como un respaldo completo y puede ser utilizado para respaldar o colaborar, incluso si otros servidores están temporalmente inaccesibles.



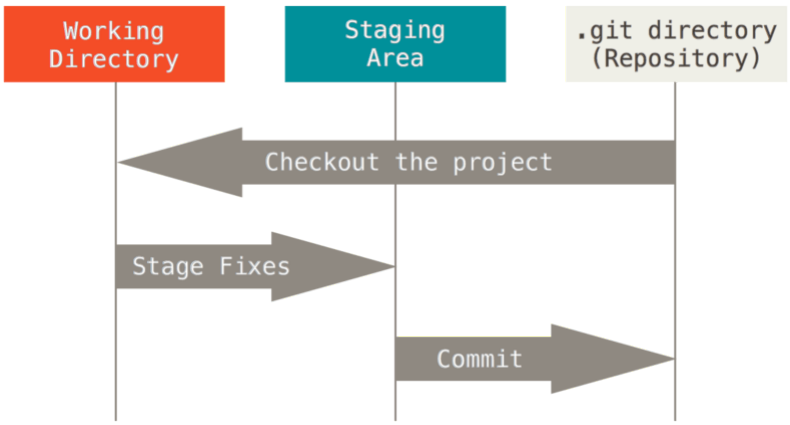
Git

Git se diferencia de otros VCS al pensar en sus datos como una serie de instantáneas de un sistema de archivos en miniatura. Es decir, cada vez que se realiza un *commit* en Git, se toma una *snapshot* de cómo se ven todos los archivos en ese momento y se almacena la referencia de esa instantánea. Esto es diferente al método de otros VCS que almacenan información como una lista de cambios basados en archivos.

Esta metodología hace que las operaciones sean mayoritariamente locales, sin la necesidad de información de otro equipo, mejorando la velocidad y la eficiencia. En lugar de modificar o eliminar datos, Git los agrega y evita el riesgo de pérdida significativa de información.

Git organiza los archivos en tres estados:

- ***Modified***. Significa que el archivo ha sido modificado, pero aún no se ha enviado al repositorio central. Esta etapa se encuentra en la sección de *Working Tree*.
- ***Staged***. Significa que el archivo modificado se ha marcado para ser incluido en el próximo *commit*. Esta etapa se encuentra en la sección *Staging Area*.
- ***Committed***. Significa que los archivos seleccionados han sido guardados en el repositorio central. Esta etapa se encuentra en la sección *Git Directory*



Ventajas y Desventajas

Al ser Git un DVCS, sus ventajas son evidentes: proporciona una visión unificada y consistente del proyecto, un historial transparente de cambios, la recuperación de archivos perdidos o dañados, información sobre versiones que introdujeron un error al proyecto, una gestión eficiente de ramificación para trabajar en nuevas características y correcciones de errores sin afectar al proyecto en producción, y un gestor de conflictos.

Sin embargo, el aprendizaje de Git presenta una curva de aprendizaje que puede ser difícil para algunas personas que son nuevas, ya que contiene conceptos que pueden llevar tiempo en comprender.

Asimismo, la interfaz de línea de comandos es menos amigable para usuarios que están acostumbrados a interfaces gráficas, aunque actualmente existen interfaces disponibles para Git. Además, Git no ofrece un control de acceso en términos de archivos, sino más bien en términos de ramas, es decir, no se pueden controlar los permisos para archivos individuales.

Comandos Básicos de Git

Los comandos son utilizados para el manejo de Git y desempeñan un papel fundamental al realizar diversas acciones, como copiar, crear, modificar y combinar código. Es decir, estos comandos son herramientas que posibilitan la administración de los repositorios en Git.

Estos comandos pueden ejecutarse directamente desde la línea de comando o utilizando interfaces gráficas como GitHub Desktop, Sourcetree, GitKraken, entre otros. A continuación, se muestran los comandos básicos para utilizar Git:

Git init

Este comando se utiliza para crear un nuevo repositorio Git, ya sea para convertir un proyecto existente sin versionar a un repositorio Git o para inicializar un repositorio nuevo y vacío en un directorio específico.

```
git init [nombre-del-directorio-de-destino]
```

Este comando crea un subdirectorio oculto llamado `.git` en el directorio actual, que contiene toda la información necesaria de Git.

- **hooks:** esta carpeta almacena *scripts* personalizables que se pueden ejecutar en respuesta a eventos específicos en Git, como pre-commit, post-commit, pre-push, entre otros.
- **info:** esta carpeta almacena archivos globales de configuración y exclusiones específicas del repositorio.
- **logs:** esta carpeta almacena archivos de historial que registran información sobre las acciones realizadas en el repositorio.
- **objects:** esta carpeta almacena todos los objetos del repositorio, como blobs (*Binary Large Object*).
- **refs:** esta carpeta almacena las referencias a ramas y las etiquetas.
- **config:** este archivo almacena la configuración específica del repositorio, como el nombre de usuario y el correo electrónico.
- **HEAD:** este archivo apunta al *commit* actualmente seleccionado en la rama actual.
- **index:** este archivo contiene el área de preparación (*Staging Area*) que almacena los cambios que se añadirán en el siguiente *commit*.

Git clone

Este comando se utiliza para clonar un repositorio Git existente en un nuevo directorio. Este repositorio puede ser tanto un repositorio remoto como local. Decir **clonar** hace referencia a obtener una copia completa de un repositorio remoto, incluyendo todo el historial de cambios y las ramas disponibles. Después de clonar un repositorio, este quedará vinculado al repositorio remoto o local y se podrán realizar otras operaciones.

```
git clone <path-del-repositorio> [nombre-del-directorio-de-destino]
```

Git add

Este comando se utiliza para almacenar provisionalmente los cambios seleccionados en el área de preparación. Estos cambios incluyen la adición, eliminación o modificación de archivos.

En otras palabras, permite controlar qué cambios se incluirán en el próximo `commit`, permitiendo seleccionar desde archivos específicos hasta todos los cambios.

```
git add <nombre-del-archivo>
```

- Para añadir todos los cambios en el directorio actual, excluyendo las eliminaciones de archivos:

```
git add .
```

- Para añadir cambios que reflejan solo modificación o eliminación pero no adición de archivos nuevos.

```
git add -u o git add --update
```

- Para añadir todos cambios en el directorial actual:

```
git add -A o git add --all
```

Git commit

Este comando se utiliza para confirmar los cambios preparados en el área de preparación. Se ejecuta después de utilizar `git add` para registrar los cambios en el historial del repositorio y agregarlos a la rama actual.

El comando `git commit` va acompañado de la opción `-m` para incluir un mensaje descriptivo. Este mensaje debe ser informativo y explicar el motivo de los cambios realizados.

```
git commit -m "Mensaje del commit"
```

Git status

Este comando se utiliza para obtener el estado actual del repositorio Git. Proporciona información sobre los cambios pendientes, archivos sin seguimiento, archivos modificados y aquellos en el área de preparación. Básicamente, ofrece una visión completa del estado de la rama actual. Este comando es útil para determinar si se requiere ejecutar `git add` antes de realizar un `git commit`.

```
git status
```

Git branch

Este comando se utiliza para listar, crear o eliminar ramas en un repositorio.

- Para listar todas las ramas presentes en el repositorio local, ejecuta el comando sin argumentos.

```
git branch
```

- Para crear una nueva rama, especifica el nombre deseado. Este comando no cambia automáticamente a la nueva rama.

```
git branch <nombre-de-la-nueva-rama>
```

- Para eliminar una rama que ya ha sido fusionada con la rama actual.

```
git branch -d <nombre-de-la-rama>
```

- Para forzar la eliminación de una rama.

```
git branch -D <nombre-de-la-rama>
```

Git checkout

Este comando se utiliza para cambiar entre ramas o versiones específicas de archivos. También es posible utilizarlo para crear nuevas ramas.

- Para cambiar de una rama a otra en el repositorio se debe especificar la nueva rama actual y el directorio de trabajo se actualizará con los archivos de esa rama.

```
git checkout <nombre-de-la-rama>
```

- Para crear una nueva rama, se debe especificar el nombre y añadir la opción `-b`. Este comando si cambia automáticamente a la nueva rama.

```
git checkout -b <nombre-de-la-nueva-rama>
```

- Para regresar un archivo específico a su estado en el último commit.

```
git checkout -- <nombre-del-archivo>
```

- Para regresar todos los archivos a su estado en el último commit.

```
git checkout -f
```

Git merge

Este comando se utiliza para combinar cambios de una rama a otra. Permite fusionar los cambios de una rama secundaria en una rama superior, integrando así las actualizaciones realizadas en ambas ramas.

Este comando debe ser ejecutado desde la rama a la cual se desea fusionar los cambios y la fusión se realizará automáticamente si no hay conflictos entre las versiones de las ramas. Si hay conflictos, Git señalará los archivos conflictivos y requerirá que se resuelvan manualmente antes de completar la fusión.

```
git merge <nombre-de-la-rama-a-incorporar>
```

Git pull

Este comando se utiliza para recuperar y fusionar los cambios desde un repositorio remoto en el repositorio local. Es decir, actualiza el repositorio local con los cambios más recientes del repositorio remoto y realiza automáticamente la fusión con la rama actual.

Si solo se ejecuta `git pull` sin especificar el repositorio remoto y la rama remota, Git utilizará los valores predeterminados configurados, generalmente están asociados con el repositorio y la rama desde el cual se clonó el repositorio local.

```
git merge <repositorio-remoto> <rama-remota>
```

Git push

Este comando se utiliza para enviar los cambios locales confirmados a un repositorio remoto. Ayuda a mantener sincronizados el repositorio local y el remoto, permitiendo así que otros colaboradores puedan acceder a los cambios que se han realizado.

Ejecutar `git push` hará que Git intente subir los cambios de la rama local especificada al repositorio remoto. Es importante tener permisos y acceso al repositorio remoto para realizar esta operación.

```
git merge <repositorio-remoto> <rama-local>
```

Git fetch

Este comando se utiliza para recuperar cambios desde un repositorio remoto, pero no fusiona esos cambios en la rama local actual. Sirve para actualizar la información sobre las ramas remotas y obtener una vista previa de los cambios antes de decidir fusionarlos.

```
git fetch
```

Git log

Este comando se utiliza para mostrar el historial de commits en un repositorio. Proporciona información detallada sobre cada commit, incluyendo el identificador del commit, el autor, la fecha y la descripción del mensaje del commit.

```
git log
```

Git help

Este comando se utiliza para obtener información detallada sobre otros comandos de Git. Proporciona documentación en línea para cualquier comando de Git específico.

```
git help <comando>
```

Trabajando con repositorios en GitHub

GitHub es una plataforma para el desarrollo colaborativo de software basado en el sistema de control de versiones Git. Facilita la contribución entre desarrolladores al alojar repositorios de código en la nube, lo que permite a los equipos trabajar de forma conjunta y administrar el flujo de trabajo.

Algunos términos relacionados con el flujo de trabajo con repositorios que facilitan la colaboración en cambios son: *branches* (ramas), *merge* (fusión) y *conflicts* (conflictos).

Branches

Una rama representa una línea independiente de desarrollo que permite trabajar en funcionalidades específicas de manera aislada. Esto incluye la corrección de errores, el desarrollo de nuevas características o la experimentación con el código sin comprometer otras ramas en el repositorio. Cada repositorio cuenta con una rama por defecto conocida como `main`.

Esta rama principal se asigna automáticamente al crear la primera rama del repositorio y generalmente es la que se muestra cuando alguien visita o descarga el repositorio en GitHub sin especificar una rama diferente. Se pueden crear nuevas ramas a partir de ramas existentes, y el primer nivel de sub-ramas se derivará de `main`. Se requiere acceso de escritura al repositorio para realizar estas acciones.

Una vez que se completa el desarrollo de un objetivo en una rama, se puede abrir una "incorporación de cambios" para combinar los cambios de esa rama con otra. Después de la fusión, la rama ya no será necesariamente requerida y se puede eliminar. Sin embargo, es importante destacar que no se puede eliminar una rama que esté vinculada a un *merge*.

Las ramas también pueden tener protecciones mediante permisos otorgados por administradores del repositorio o roles personalizados con permisos específicos. Estas protecciones pueden incluir restricciones para eliminar ramas, realizar fusiones, requerir revisiones y verificar estados, entre otras medidas.

Algunos comandos relacionados con ramas son: `git branch` y `git checkout`.

Merge

La fusión, o *merge*, es una operación que permite combinar cambios realizados en diferentes ramas de un repositorio. Esta función permite mantener la coordinación y garantizar que las modificaciones se integren de manera efectiva entre ramas.

El proceso de *merge* sigue la siguiente secuencia de acciones:

- **Creación de ramas**. Se crean ramas para trabajar en diferentes líneas de desarrollo, cada una dedicada a una funcionalidad específica.
- **Cambios**. Cada rama experimenta cambios que incluyen modificaciones en el proyecto de acuerdo con la funcionalidad asignada.
- **Pull Request**. La solicitud de fusión se genera cuando se desean fusionar los cambios de una rama en la rama principal o en otra rama. Este *Pull Request** (PR) constituye una solicitud formal para integrar los cambios de una rama a otra.
- **Revisión**. Previo a la fusión, puede ser necesario que otros miembros del equipo, posiblemente con roles más elevados, revisen el código propuesto en el PR. Esto puede involucrar la realización de comentarios, sugerencias de cambios, aceptación o rechazo de los mismos.
- **Automerge** o **Resolución de Conflictos**. Si no existen conflictos entre los cambios propuestos y la rama de destino, GitHub realiza la fusión automáticamente. No obstante, en caso de conflictos, estos deben ser resueltos manualmente para proceder con la fusión.
- **Fusión completa**. Posterior a la fusión, los cambios se incorporan en la rama de destino. Es relevante destacar que la rama de origen no se elimina automáticamente, para hacerlo se debe realizarse manualmente a través del comando `git branch -d`.

Conflicts

Los conflictos hacen referencia a situaciones en las que la fusión automática de cambios entre ramas no es posible debido a alteraciones diferentes en el mismo fragmento de código o archivo. Es decir, existen

modificaciones conflictivas en ambas ramas que están siendo fusionadas, generando divergencias en el historial de cambios, lo que resulta en un conflicto.

Los conflictos se identifican cuando se intenta realizar un PR para fusionar cambios entre ramas. Si hay conflictos, se marcarán las líneas específicas en las que existen. Para solucionar estos conflictos, generalmente se siguen los siguientes pasos:

- **Identificación de conflictos**. Al crear un PR, se notifica la existencia de conflictos, y los desarrolladores a cargo revisarán las líneas marcadas como conflictivas para visualizar las diferencias de código.
- **Solución manual**. Los desarrolladores deben ajustar manualmente el código para resolver las diferencias y eliminar los conflictos. Esta solución puede realizarse mediante la combinación de ambas modificaciones o seleccionando una sola versión.
- **Commit de resolución**. Se debe realizar un `commit` para registrar la resolución de conflictos y continuar con el proceso de fusión.
- **Actualización del PR**. Una vez completados los pasos anteriores, el PR debe actualizarse para que GitHub detecte el nuevo *commit* y continúe con el proceso de fusión, el cual deberá completarse manualmente.

Cabe destacar que actualmente existen herramientas que proporcionan interfaces visuales para facilitar el proceso de resolución de conflictos, ya que permiten visualizar las diferencias entre las ramas y elegir qué versiones se mantendrán para solucionar el conflicto de cambios.