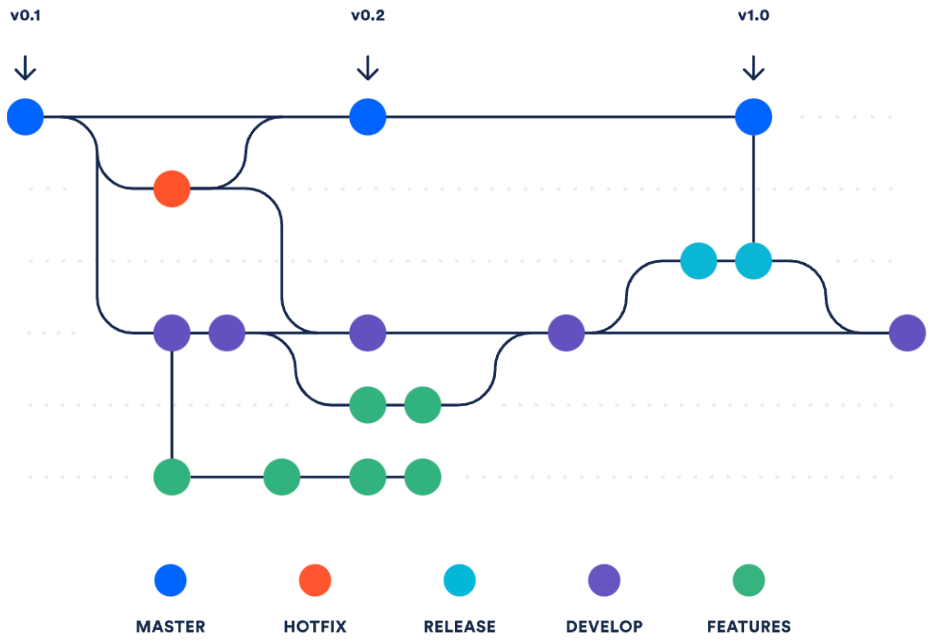
¿Qué es un sistema de control de versiones?

Un sistema de control de versiones (o VCS, por sus siglas en inglés), también conocido como sistema de control de revisiones o de fuentes, es una herramienta de software que monitoriza y gestiona cambios en un sistema de archivos. Asimismo, un VCS ofrece herramientas de colaboración para compartir e integrar dichos cambios en otros usuarios del VCS. Al operar al nivel del sistema de archivos, un VCS monitorizará las acciones de adición, eliminación y modificación aplicadas a archivos y directorios. Un repositorio es un término del VCS que describe cuando un VCS está monitorizando un sistema de archivos. En el alcance los archivos individuales de códigos fuente, un VCS monitorizará las adiciones, eliminaciones y modificaciones de las líneas de texto que contiene ese archivo. Entre las opciones populares de VCS del sector de software, se incluyen ***Git***, Mercurial, SVN y preforce.



¿Por qué necesito un software de control de versiones?

VCS es una valiosa herramienta con numerosos beneficios para un flujo de trabajo de equipos de software de colaboración. Cualquier proyecto de software que tiene más de un desarrollador manteniendo archivos de código fuente debe, sin duda, usar un VCS. Además, los proyectos mantenidos por una sola persona se beneficiarán enormemente del uso de un VCS. Se puede decir que no hay una razón válida para privarse del uso de un VCS en cualquier proyecto moderno de desarrollo de software.

Problemas que resuelve el uso de un VCS

Resolución de conflictos

Durante el ciclo de vida de un proyecto de software impulsado por equipos, es muy probable que los miembros del equipo tengan la necesidad de realizar cambios en el mismo archivo de código fuente al mismo tiempo. Un VCS monitoriza y ayuda en los conflictos entre varios desarrolladores. Estas operaciones de resolución de conflictos dejan un registro de auditoría que ofrece información sobre el historial de un proyecto.

Revertir y deshacer los cambios en el código fuente

Una vez que un VCS ha empezado a monitorizar un sistema de archivos de códigos fuente, conserva un historial de cambios y el estado del código fuente durante el historial de un proyecto. De esta forma, existe la posibilidad de "deshacer" o revertir un proyecto de

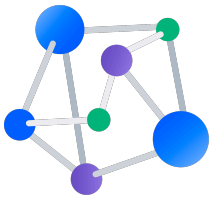
código fuente a un estado conocido reciente. Si se detecta un fallo en una aplicación en vivo, el código puede revertirse rápidamente a una versión estable conocida.

Copia de seguridad externa del código fuente

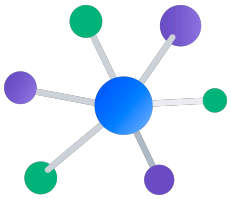
Al usar un VCS de forma colaborativa, se debe crear una instancia remota del VCS para compartir cambios entre desarrolladores. Esta instancia remota del VCS se puede alojar de forma externa con un tercero de confianza como GitHub, Bitbucket entre otros. A continuación, se convierte en una copia de seguridad externa. En un caso desafortunado (como el del robo de un portátil), la instancia remota del VCS conservará una copia del código fuente.

VCS de arquitectura centralizada y distribuida.

Cuando hablamos de los pros y los contras de cada arquitectura, la función de copia de seguridad externa es el principal punto de debate. Un VCS centralizado cuenta con un solo punto de error, que es la instancia remota del VCS central. Si se pierde dicha instancia, puede producir la pérdida de datos y productividad, y se deberá sustituir por otra copia del código fuente. Si se vuelve temporalmente no disponible, evitará que los desarrolladores envíen, fusionen o reviertan código. Un modelo distribuido de arquitectura evita estos obstáculos manteniendo una copia total del código fuente en cada instancia de VCS. Si se produce en el modelo distribuido cualquiera de los casos de error centralizados antes mencionados, se puede introducir una instancia de VCS al principal mitigando cualquier caída grave de productividad.



Distribuido



Centralizado

Las ventajas de las herramientas del control de versiones

Integrar un VCS en un proyecto de desarrollo de software favorece una variedad de ventajas ge gestión y organización. De forma predeterminada, solo un VCS ofrece las ventajas técnicas antes mencionadas sobre la resolución de conflictos de equipos y ayudas de colaboración. Un servicio alojado de VCS encapsula una VCS predeterminada y ofrece mejoras en las funciones. Este "VCS mejorado" es increíblemente potente y proporciona una visión transparente del proceso de desarrollo de software, que suele ser una tarea creativa opaca. Los siguientes puntos son algunas de las ventajas de alto nivel que ofrece un VCS alojado.

Git y control de versiones

Git es un software de control de versiones distribuido diseñado por Linus Torvalds, pensando en la eficiencia, la confiabilidad y compatibilidad del mantenimiento de versiones de aplicaciones cuando estas tienen un gran número de archivos de código fuente.

*Fuente: https://bitbucket.org/

Comenzar a utilizar Git

Lo primero es instalar Git en nuestro equipo de desarrollo (PC, Notebook, etc.). Para ello descargamos la ultima versión disponible compatible con el sistema operativo que estamos

utilizando, dado que es multi plataforma, se puede utilizar tanto en Windows, Linux y Mac en forma indistinta.

👉🏻 **Link de descarga:** https://git-scm.com/

La instalación es muy sencilla, y en general, solo se requiere ir aceptando cada paso de confirmación solicitado por el instalador.
De ser necesario, se puede acceder a una guía de instalación, en donde se detalla los pasos para cada sistema operativo.

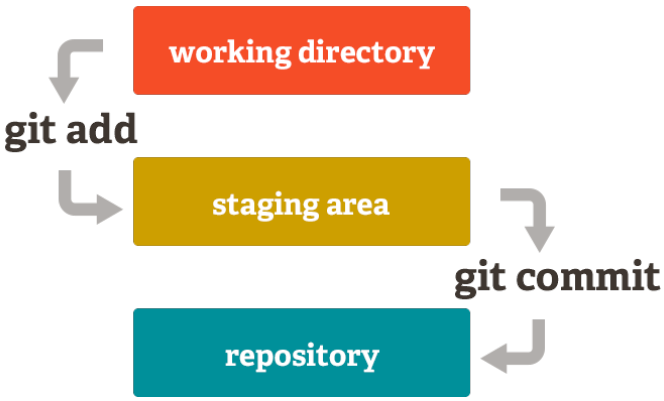
👉🏻 **Guía de instalación:** https://github.com/git-guides/install-git

Luego de la instalación, en el caso de Linux y Mac, Git se integrará de forma natural, accediendo mediante el terminal estándar del propio sistema operativo.
Git para Windows proporciona una emulación BASH utilizada para ejecutar Git desde la línea de comandos. La emulación BASH se comporta igual que el comando "git" en entornos LINUX y UNIX, por lo que para acceder a los comandos de Git, es necesario utilizar dicho emulador.

Por primera vez, es necesaria una mínima configuración que nos permita identificarnos como dueños de los cambios. Al usar el sufijo —global garantizamos que esta configuración servirá para todos los proyectos manejados por git. Para ello utilizaremos los siguientes comandos:

git config --global user.name “Nombre Apellido”
git config --global user.email “nombre@ejemplo.com”

Estructura de Git



- **Working directory:** (Directorio de trabajo) Es el directorio o carpeta donde se aloja el proyecto.
- **Staging area:** Área de estadificación o índice. Área intermedia donde se le da seguimiento a los archivos con los que estamos trabajando, para luego poder ser enviados al repositorio. Todos los archivos que hayan sido creado o modificados y que no estén en el Staging area no serán enviados al repositorio por lo que no habrá un control o seguimiento de los cambios.
- **Repository:** (Repositorio). Este es el área de almacenamiento virtual del proyecto, donde se guardan todas las versiones del código que fueron confirmadas desde el Staging área.

Principales comandos en orden de uso (local)

git init

Inicializar un nuevo proyecto. Para poder utilizar Git en un proyecto, nuevo o ya existente es necesario por única vez inicializarlo, para ello d*ebemos posicionarnos en la carpeta donde se encuentra el proyecto* y desde la consola ejecutar git init

Inmediatamente ejecutado el comando, dentro de la raid de la carpeta del proyecto se creará una carpeta oculta .git la cual contiene todos los archivos necesarios para que git gestione el proyecto. Se recomienda no tocar nada del contenido de la misma para garantizar el correcto funcionamiento.

git add

Agrega el o los archivos del working directory al staging area para el seguimiento de los cambios.

Agregar un archivo específico: *git add nombre_del_archivo*

Agregar todos los archivos que aun están sin seguimiento en el proyecto: git add .

(git add seguido de un punto)

git status

Muestra el estado en el que están los archivos del proyecto, los que están siendo controlados (dentro del staging area) y los que aun están fuera del staging area. Estos últimos no se tomarán en cuenta al momento de darle la orden de pasar al repositorio.

git commint -m “comentario de referencia de los cambios”

pasa todos los archivos que están en el staging area al repositorio (es como crear nuestra foto del código). El comentario o etiqueta del commit debe estar entre comillas ya que lo que contenga dentro es texto libre. Debe ser lo mas descriptivo posible para poder darle un seguimiento a los cambios al revisar el histórico, sobre todo cuando trabajamos con un equipo y cada integrante sepa de manera rápida de que se trata el conjunto de cambios.

git branch

Este comando tiene dos funciones dependiendo como se lo utilice:

- **Sin sufijo:** Muestra la o las ramas (branch) definidas en el proyecto. Una rama es una bifurcación del código tomando como base el estado actual del código donde declaramos el nuevo branch. Cada acción que se tome en una rama o branch es independiente a las demás. Un caso práctico de uso sería tener una rama master o de producción, y en paralelo un branch de desarrollo, de manera que cuando se aprueben los cambios generados en esta rama, se puedan llevar a la rama master sin romper la solución productiva. Se pueden declarar cuantas ramas sean necesarias en cualquier momento y reintegrarlas a otras en el momento que sea necesario.
- **Con sufijo:** por ejemplo *git brunch* ***development*** Aquí se creará una nueva rama a partir del branch donde estamos situados tomando como base una foto actual del código.

git merge nombre_del_branch_a_fusionar

Este comando permite traer a la rama donde estamos posicionados los cambios aceptados (todos los que pasaron por commit) de otra rama.

Por ejemplo, tenemos una rama donde estamos desarrollando código para el login. Una vez que terminamos, necesitamos integrar esos cambios a la rama master o principal, entonces nos posicionamos en master (git checkout master) y ejecutamos git merge login donde login es el nombre de la rama donde se encuentra el desarrollo a fusionar.

git checkout

Permite pasar de la rama actual a otra, por ejemplo, si estamos en la rama master, *git checkout development* nos posicionará ahora en la rama development.

git stash

Este comando es de mucha ayuda si necesitamos guardar los cambios de forma temporal para retomar el desarrollo en otro momento. Por ejemplo, estamos trabajando en un bugfix o corrección, pero se nos ha encargado una tarea de urgencia que requiere dejar de lado por un rato lo que estamos haciendo. Para ello, usamos git stash, lo que permitirá guardar y ocultar los cambios, volver al estado original del código, tomar el desarrollo de urgencia y luego retomar la tarea del bugfix.

Procedimiento:

Utilizamos ***git status*** para ver exactamente que es lo que vamos a guardar, es decir, cuales son los archivos que contienen cambios hasta el momento.

Luego con ***git stash*** guardará los cambios. A continuación se mostrará un mensaje que indica que los cambios se han guardado y ocultado en <nombre-del-branch>. El branch ahora se verá como antes hacer los cambios.

git stash list muestra una lista con todos los cambios ocultos guardados en el branch en el que estamos situado.

git stash show muestra el detalle de los cambios ocultos guardados.

git stash apply recupera un conjunto de cambios ocultos guardado manteniendo una copia oculta. Es necesario luego ejecutar git add si queremos pasar el conjunto de cambios al staging area.

git stash pop similar al comando anterior, pero en este caso se eliminará la copia oculta luego de haber recuperado los cambios.

Trabajando con repositorios remotos

git clone url_del_repo_remoto

Se utiliza para traer (crea una copia local) todo el código subido en una rama del repositorio remoto al local. De esta manera podemos trabajar localmente offline y cuando sea necesario subir esos cambios al repo remoto.

git remote

Se utiliza para crear, ver y eliminar conexiones remotas con otros repositorios.

- **git remote** Muestra las conexiones remotas con otros repositorios .
- **git remote -v** Similar al anterior pero incluye la url de cada conexión remota.
- **git remote add <nombre> <url_repo_remoto>** Crea una nueva conexión remota.

Mas info en: https://docs.github.com/es/get-started/getting-started-with-git/managing-remote-repositories

git push

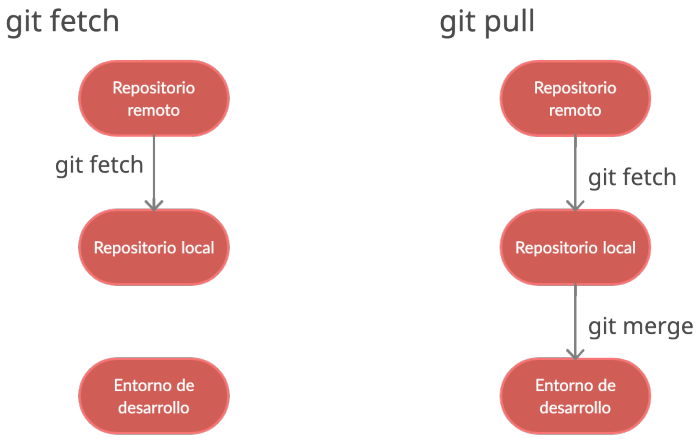
Sube los cambios aceptados (ya pasados con commit al repo local) y así podrá ser accedido por los demás integrantes del equipo que tengas acceso al repo remoto.

git fetch

Hace que el repositorio Git local se actualice con la última información que hay en el repositorio remoto, pero no hace ninguna transferencia de archivos al espacio de trabajo local o working directory (el código que se ve en editor por ejemplo). Podría decirse que sirve para comprobar si hay algún cambio y traerlo a tu repositorio local.

git pull

Comprueba si hay cambios en el repositorio remoto y, en caso de que los haya, se trae esos archivos al repositorio local y actualiza el espacio de trabajo o working directory. Simplificando mucho la explicación, git pull hace un git fetch seguido de un git merge.



📝 Es aconsejable mantener el repositorio local actualizado para evitar posibles conflictos o peor aún, trabajar con código desactualizados/obsoleto.

👉🏻 Git Ebook: https://github.com/progit/progit2/releases/download/2.1.351/progit.pdf

👉🏻 Lista completa de todos los comandos: https://git-scm.com/docs/git#_git_commands

👉🏻 Para más información sobre Git: https://git-scm.com/docs