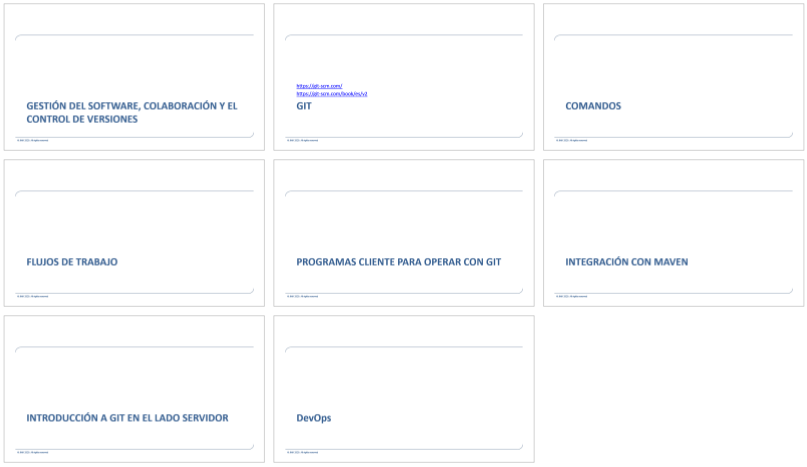
Control de Versiones

Contenidos



GESTIÓN DEL SOFTWARE, COLABORACIÓN Y EL CONTROL DE VERSIONES

Evolución del software

- Durante el desarrollo
  - El desarrollo del software siempre es progresivo, incluso en el ciclo de vida en cascada
  - El desarrollo evolutivo consiste, precisamente, en una evolución controlada (ciclo de vida espiral, prototipos evolutivos)
- Durante la explotación
  - Durante la fase de mantenimiento se realizan modificaciones sucesivas del producto
- Además de ello, el desarrollo de software se realiza normalmente en equipo, por lo que es necesario integrar varios desarrollos en un solo producto

Control de versiones

- Un control de versiones es un sistema que registra los cambios realizados en un archivo o conjunto de archivos a lo largo del tiempo, de modo que puedas recuperar versiones específicas más adelante.
- Los sistemas de control de versiones son herramientas de software que ayudan a los equipos de software a gestionar los cambios en el código fuente a lo largo del tiempo.
- A medida que los entornos de desarrollo se aceleran, los sistemas de control de versiones ayudan a los equipos de software a trabajar de forma más rápida e inteligente. Son especialmente útiles para los equipos de DevOps, ya que les ayudan a reducir el tiempo de desarrollo y a aumentar las implementaciones exitosas.

Características

- Mecanismo de almacenamiento de los elementos que deba gestionar (ej. archivos de texto, imágenes, documentación...).
- Posibilidad de realizar cambios sobre los elementos almacenados (ej. modificaciones parciales, añadir, borrar, renombrar o mover elementos).
- Registro histórico de las acciones realizadas con cada elemento o conjunto de elementos (normalmente pudiendo volver o extraer un estado anterior del producto).

Ventajas

- Un completo historial de cambios a largo plazo de todos los archivos.
  - Esto quiere decir todos los cambios realizados por muchas personas a lo largo de los años. Los cambios incluyen la creación y la eliminación de los archivos, así como los cambios de sus contenidos. Que, quien y cuando se realizaron los cambios. Facilita volver a una versión anterior.
- Creación de ramas y fusiones.
  - La creación de una "rama" permite mantener múltiples flujos de trabajo independientes los unos de los otros al tiempo que ofrece la facilidad de volver a fusionar ese trabajo, lo que permite que los desarrolladores verifiquen que los cambios de cada rama no entran en conflicto. Facilita el trabajo colaborativo.
- Trazabilidad.
  - Ser capaz de trazar cada cambio que se hace en el software y conectarlo con un software de gestión de proyectos y seguimiento de errores, además de ser capaz de anotar cada cambio con un mensaje que describa el propósito y el objetivo del cambio, no solo ayuda con el análisis de la causa raíz y la recopilación de información.

Conceptos

- Versión
  - “Versión” es la forma particular que adopta un objeto en un contexto dado.
- Revisión
  - Desde el punto de vista de evolución, es la forma particular de un objeto en un instante dado. Se suele denominar “revisión”.
- Configuración
  - Una “configuración” es una combinación de versiones particulares de los componentes (que pueden evolucionar individualmente) que forman un sistema consistente.
  - Desde el punto de vista de evolución, es el conjunto de las versiones de los objetos componentes en un instante dado

Conceptos

- Línea base
  - Llamaremos “línea base” a una configuración operativa del sistema software a partir del cual se pueden realizar cambios subsiguientes.
  - La evolución del sistema puede verse como evolución de la línea base
- Cambio
  - Un cambio representa una modificación específica a un archivo bajo control de versiones. La granularidad de la modificación considerada un cambio varía entre diferentes sistemas de control de versiones.
  - Un “cambio” es el paso de una versión de la línea base a la siguiente
  - Puede incluir modificaciones del contenido de algún componente, y/o modificaciones de la estructura del sistema, añadiendo o eliminando componentes

Conceptos

- Branch:
  - Una “ramificación”, bifurcación o variante es una bifurcación de la línea base u otra ramificación que a partir de ese momento evoluciona y se versiona por separado.
  - Las ramificaciones representan una variación espacial, mientras que las revisiones representan una variación temporal.
- Repositorio
  - El repositorio es el lugar en el que se almacenan los datos actualizados e históricos de cambios, a menudo en un servidor.
  - El repositorio permite ahorrar espacio de almacenamiento, evitando guardar por duplicado elementos comunes a varias versiones o configuraciones

Sistemas de Control de Versiones Manuales

- Un método de control de versiones, usado por muchas personas, es copiar los archivos a otro directorio (quizás indicando la fecha y hora en que lo hicieron, si son ingeniosos).
- Este método es muy común porque es muy sencillo, pero también es tremendamente propenso a errores.
- Es fácil olvidar en qué directorio te encuentras y guardar accidentalmente en el archivo equivocado o sobrescribir archivos que no querías.
- Consume mucho espacio al duplicar elementos que no se han modificado entre versiones.

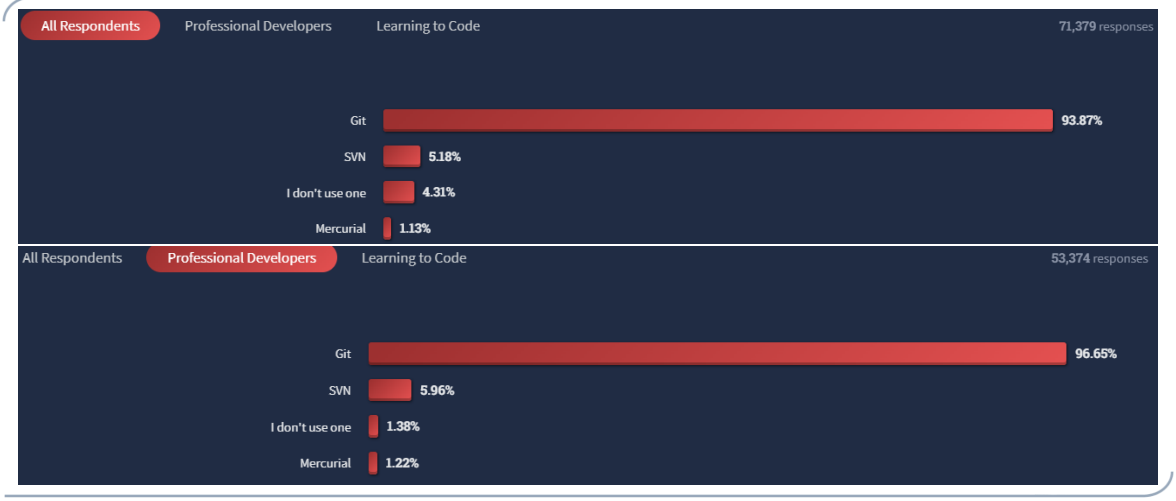
Sistemas de Control de Versiones Centralizados

- El siguiente gran problema con el que se encuentran las personas es que necesitan colaborar con desarrolladores en otros sistemas. Los sistemas de Control de Versiones Centralizados (CVCS por sus siglas en inglés) fueron desarrollados para solucionar varias personas que necesitan colaborar.
- Estos sistemas, como CVS, Subversion y Perforce, tienen un único servidor que contiene todos los archivos versionados y varios clientes que descargan los archivos desde ese lugar central. Este ha sido el estándar para el control de versiones por muchos años.
- Esta configuración ofrece muchas ventajas, especialmente frente a VCS locales, permite trabajar de forma exclusiva: para poder realizar un cambio es necesario comunicar al repositorio el elemento que se desea modificar y el sistema se encargará de impedir que otro usuario pueda modificar dicho elemento, hasta que sea liberado.
- Sin embargo, esta configuración también tiene serias desventajas:
  - Es el punto único de fallo que representa el servidor centralizado.
  - Sin conexión al servidor nadie podrá colaborar o guardar cambios en archivos en los que hayan estado trabajando.

Sistemas de Control de Versiones Distribuidos

- Los sistemas de Control de Versiones Distribuidos o DVCS (como Git, Mercurial, Bazaar o Darcs), los clientes no solo descargan la última copia instantánea de los archivos, sino que se replica completamente el repositorio, existe un repositorio local y otro central.
- De esta manera, si un servidor deja de funcionar y estos sistemas estaban colaborando a través de él, cualquiera de los repositorios disponibles en los clientes puede ser copiado al servidor con el fin de restaurarlo. Cada clon es realmente una copia completa de todos los datos.
- Se puede trabajar de forma local. Permite operaciones más rápidas.
- Cada usuario modifica la copia local y cuando el usuario decide compartir los cambios el sistema automáticamente intenta combinar las diversas modificaciones. El principal problema es la posible aparición de conflictos que deban ser solucionados manualmente o las posibles inconsistencias que surjan al modificar el mismo fichero por varias personas no coordinadas.

Sistemas de control de versiones



https://survey.stackoverflow.co/2022/#version-control-version-control-system

https://git-scm.com/

https://git-scm.com/book/es/v2

GIT

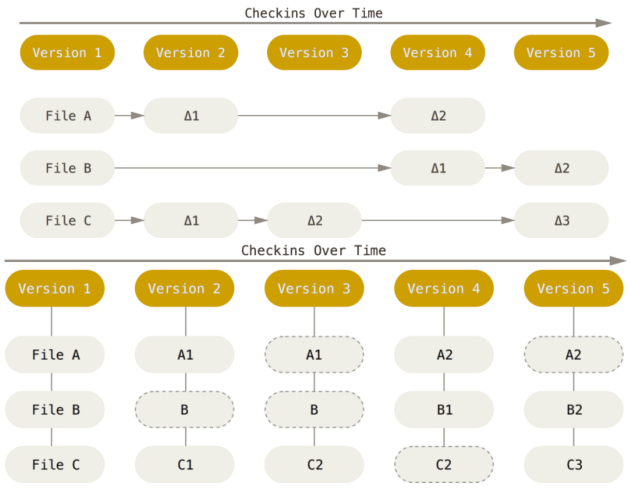
GIT

- Git (pronunciado "guit") es un sistema de control de versiones distribuido de código abierto y gratuito diseñado para manejar todo, desde proyectos pequeños a muy grandes, con velocidad y eficiencia.
- Git es, con diferencia, el sistema de control de versiones moderno más utilizado del mundo. Git es un proyecto de código abierto maduro y con un mantenimiento activo que desarrolló originalmente Linus Torvalds, el famoso creador del kernel del sistema operativo Linux, en 2005.
- Git es fácil de aprender y ocupa poco espacio con un rendimiento increíblemente rápido. Supera a las herramientas SCM como Subversion, CVS, Perforce y ClearCase con características como bifurcaciones locales económicas, áreas de preparación convenientes y múltiples flujos de trabajo. Funciona a la perfección en una amplia variedad de sistemas operativos e IDE (entornos de desarrollo integrados).
- No confundir con GitHub, que es una forja (plataforma de desarrollo colaborativo) para alojar proyectos utilizando el sistema de control de versiones Git, siendo la plataforma más importante de colaboración para proyectos Open Source.
- Instalación:
  - https://git-scm.com/

Fundamentos de Git

- La principal diferencia entre Git y cualquier otro VCS es la forma en la que manejan sus datos. Conceptualmente, la mayoría de los otros sistemas almacenan la información como una lista de cambios en los archivos, manejan la información que almacenan como un conjunto de archivos y las modificaciones hechas a cada uno de ellos a través del tiempo.
- Git maneja los datos como un conjunto de copias instantáneas de un sistema de archivos miniatura. Cada vez que se confirma un cambio o guarda el estado del 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 los datos como una secuencia de copias instantáneas.
- La mayoría de las operaciones en Git sólo necesitan archivos y recursos locales para funcionar. Por lo general no se necesita información de ningún otro equipo de tu red.
- Todo en Git es verificado mediante una suma de comprobación (checksum) antes de ser almacenado, y es identificado a partir de ese momento mediante dicha suma. Esto significa que es imposible cambiar los contenidos de cualquier archivo o directorio sin que Git lo detecte.
- Cuando realizas acciones en Git, casi todas ellas sólo añaden información a la base de datos de Git. Es muy difícil hacer algo que no se pueda enmendar, salvo que se le fuerce (--force).

Instantáneas, no diferencias



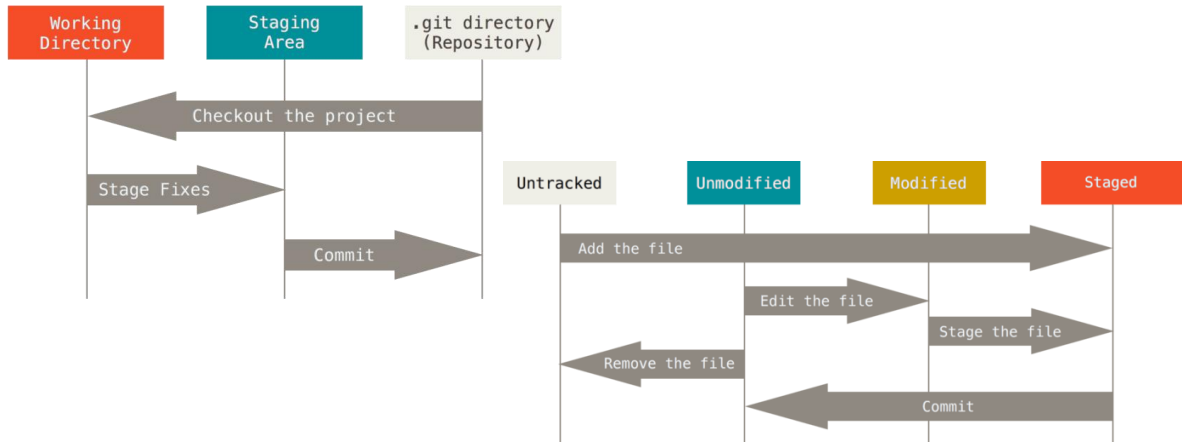
Estados

- Git tiene cuatro estados principales en los que se pueden encontrar tus archivos:
  - Modificado (modified): significa que has modificado el archivo, pero todavía no lo has confirmado a tu base de datos.
  - Preparado (staged): significa que has marcado un archivo modificado en su versión actual para que vaya en tu próxima confirmación.
  - Confirmado (committed): significa que los datos están almacenados de manera segura en tu base de datos local.
  - Sin seguimiento (Untracked): significa que Git no hace seguimiento del archivo.
- Esto nos lleva a las tres secciones principales de un proyecto de Git:
  - El directorio de Git (git directory) es donde se almacenan los metadatos y la base de datos de objetos para tu proyecto. Es la parte más importante de Git, y es lo que se copia cuando clonas un repositorio desde un repositorio central.
  - El área de preparación (staging area) es un archivo, generalmente contenido en tu directorio de Git, que almacena información acerca de lo que va a ir en tu próxima confirmación. A veces se le denomina índice ("index"), pero se está convirtiendo en estándar el referirse a ella como el área de preparación.
  - El directorio de trabajo (working directory) es una copia de una versión del proyecto (sandbox). Estos archivos se sacan de la base de datos comprimida en el directorio de Git, y se colocan en disco para que los puedas usar o modificar.

Flujo de trabajo

- El flujo de trabajo básico en Git es algo así:
  1. Modificas una serie de archivos en tu directorio de trabajo.
  2. Preparas los archivos, añadiéndolos a tu área de preparación.
  3. Confirmas los cambios, lo que toma los archivos tal y como están en el área de preparación y almacena esa copia instantánea de manera permanente en tu directorio de Git.
  4. Periódicamente, se sincronizan el repositorio local con el repositorio central para que los cambios estén disponibles para el resto de los colaboradores, actuando también como copia de seguridad del repositorio local.
- Si una versión concreta de un archivo está en el directorio de Git, se considera confirmada (committed). Si ha sufrido cambios desde que se obtuvo del repositorio, pero ha sido añadida al área de preparación, está preparada (staged). Y si ha sufrido cambios desde que se obtuvo del repositorio, pero no se ha preparado, está modificada (modified).

Flujo de trabajo



Terminología

- commit: un objeto Git, una instantánea de todo su repositorio comprimido en un SHA.
- branch (rama): un puntero móvil ligero a una confirmación.
- clone: una versión local de un repositorio, incluidas todas las confirmaciones y ramas.
- remote: un repositorio común en un servidor central (como GitHub) que todos los miembros del equipo usan para intercambiar sus cambios.
- fork: una copia de un repositorio del servidor central propiedad de un usuario diferente.
- pull request (solicitud de extracción): un lugar para comparar y discutir las diferencias introducidas en una rama con revisiones, comentarios, pruebas integradas y más
- HEAD: Última instantánea del commit, próximo padre, representa el punto de trabajo actual, el puntero HEAD se puede mover a diferentes ramas, etiquetas o confirmaciones cuando se usa git checkout.
- INDEX: Siguiente instantánea del commit propuesto, cache de los archivos ya revisados en el área de staging.
- Working Directory: es la copia local de los archivos de una versión del proyecto (sandbox).

Instalación y personalización de Git

- Instalación: https://git-scm.com/downloads
- La instalación suministra:
  - un CLI donde ejecutar los comandos de Git: git
  - una interfaz gráfica muy rudimentaria: git-gui
  - un visor gráfico del histórico: gitk
  - una Bash con las extensiones de Git (se configura en .bashrc): git-bash
- Git trae el comando config, que te permite obtener y establecer variables de configuración que controlan el aspecto y funcionamiento de Git.
- Lo primero que deberás hacer es establecer tu nombre de usuario (o alias) y dirección de correo electrónico. Los "commits" de Git usan esta información (quien), y es introducida de manera inmutable en los commits que envías (se debe anonimizar si se quiere mantener la privacidad en repositorios públicos):

```
$ git config --global user.name "usuario o nombre"
$ git config --global user.email mi.correo@example.com
$ git config --global user.email "ID+USERNAME@users.noreply.github.com"
```


Configuración de Git

- Git se configura a tres niveles, que sobrescriben los valores del nivel anterior:
  - Sistema: /etc/gitconfig — --system
  - Usuario: ~/.gitconfig o ~/.config/git/config (C:\Users\$USER) — --global
  - Repositorio: ./.git/config — *(por defecto)*
- Para conocer la configuración actual de Git, se puede utilizar el comando

```
$ git config --list
$ git config --list --show-origin
```

- Los valores que se pueden definir con:

```
$ git config --global core.editor "code --wait --new-window"
$ git config --global init.defaultBranch main
$ git config --global core.autocrlf false
```

- Por defecto, Git es sensible a mayúsculas y minúsculas en los nombres de archivos, esto se puede cambiar con:

```
$ git config --global core.ignorecase true
```

- Para editar la configuración si tener que modificar clave a clave:

```
$ git config --global --edit
```

VS Code como editor predeterminado

$ git config --global --edit

```
[core]
      editor = code --wait --new-window
[diff]
   tool = default-difftool
[difftool "default-difftool"]
   cmd = code --wait --diff $LOCAL $REMOTE
[merge]
   tool = code
[mergetool "code"]
   cmd = code --wait --merge $REMOTE $LOCAL $BASE $MERGED
```


Crear un repositorio Git

- Se puede obtener un proyecto Git de dos maneras: La primera es tomar un proyecto o directorio existente e importarlo en Git. La segunda es clonar un repositorio existente en Git desde otro servidor.
- Inicializando un repositorio en un directorio existente (esto crea un subdirectorio oculto llamado .git que contiene todos los archivos necesarios del repositorio local) [--initial-branch=main]:
  $ git init
- Para añadir los ficheros al área de preparación:
  $ git add .
- Para confirmar los cambios del área de preparación:
  $ git commit -m 'initial project version'
- Para vincular el repositorio local con el repositorio central y sincronizar los cambios:
  $ git branch -M master
  $ git remote add origin https://github.com/myuser/myrepository.git
  $ git push -u origin master
- Clonando un repositorio existente:
  $ git clone https://github.com/myuser/myrepository localdir

.git

- Cuando lanzas git init sobre una carpeta nueva o sobre una ya existente, Git crea la carpeta auxiliar .git; la carpeta donde se ubica prácticamente todo lo almacenado y manipulado por Git. Si deseas hacer una copia de seguridad de tu repositorio, con tan solo copiar esta carpeta a cualquier otro lugar ya tienes tu copia completa.
- En términos de sistema de ficheros, un repositorio Git es una carpeta que incluye un directorio oculto .git. Éste contiene toda la información necesaria para el control de versiones (algunos solo se crearán si son necesarios):
  - hooks: directorio con los scripts que se ejecutan en determinados eventos
  - info y logs: directorios con información extra sobre el repositorio y los commits
  - objects: directorio con los blobs, trees, commits, tags anotados
  - refs: directorio con referencias heads (branchs), original, remotes, tags
  - worktrees: directorio con referencias a los arboles de trabajo.
  - HEAD, FETCH_HEAD, ORIG_HEAD: ficheros con los diferentes punteros
  - index: fichero con el área de preparación
  - config: fichero de configuración local

.git

- Un repositorio Git es una base de datos de objetos. Los objetos se almacenan en ficheros bajo un "nombre de objeto" de 40 dígitos (CRC en SHA-1).
- Hay cuatro tipos diferentes de objetos:
  - Un objeto "blob" se utiliza para almacenar datos de archivos como un objeto binario largo. No hace referencia a nada más ni tiene atributos de ningún tipo, son las referencias finales.
  - Un objeto "árbol" vincula uno o más objetos "blob" a una estructura de directorios. Además, un objeto árbol puede hacer referencia a otros objetos árbol, creando así una jerarquía de directorios.
  - Un objeto de "confirmación" vincula dichas jerarquías de directorios en un grafo acíclico dirigido de revisiones: cada confirmación contiene el nombre de objeto de exactamente un árbol que designa la jerarquía de directorios en el momento de la confirmación. Además, una confirmación hace referencia a objetos de confirmación principales que describen el historial de cómo se llegó a esa jerarquía de directorios.
  - Un objeto "etiqueta" identifica simbólicamente otros objetos y puede utilizarse para firmarlos. Contiene el nombre y el tipo de otro objeto, un nombre simbólico y, opcionalmente, una firma.
- Git se gestiona mediante referencias a los nombres de objeto (SHA-1).

.gitignore .gitattribute .gitkeep

- A veces, tendrás algún tipo de archivo que no quieres que Git añada automáticamente o más aun, que ni siquiera quieras que aparezca como no rastreado. Este suele ser el caso de archivos generados automáticamente como trazas o archivos creados por el sistema de compilación. En estos casos, se puede crear un archivo de texto llamado .gitignore que liste patrones de los archivos a excluir.
- Las reglas usan patrones globales y puedes incluir en el archivo .gitignore las siguientes:
  - Ignorar las líneas en blanco y aquellas que comiencen con #.
  - Emplear patrones glob estándar que se aplicarán recursivamente a todo el directorio del repositorio local.
  - Los patrones pueden comenzar en barra (/) para evitar recursividad.
  - Los patrones pueden terminar en barra (/) para especificar un directorio.
  - Los patrones pueden negarse, ignorar el ignorar, al añadir al principio el signo de exclamación (!).
- Algunos de los ajustes, denominados atributos, pueden ser especificados para un camino (path) concreto, de tal forma que Git los aplicará únicamente para una carpeta o para un grupo de archivos determinado. Y se pueden fijar mediante un archivo .gitattribute, con cosas tales como indicar diferentes estrategias de fusión, comparar archivos no textuales, conversiones de fin de línea …
- Dado que Git trabaja a nivel de ficheros, no es capaz de gestionar carpetas vacías. Para poder solucionar el problema de compartir estructuras de carpetas iniciales, se suele utilizar la convención de crear un archivo vacío llamado .gitkeep que se elimina al aparecer el primer archivo de la carpeta.

Alias de comandos

- Git permite crear alias de comandos que puede hacer que tu experiencia con Git sea más simple, sencilla y familiar. Un alias es un nombre o abreviatura asociado a comandos de uso común o un atajo a comandos que requieran múltiples parámetros.

```
$ git config --global alias.st status
$ git config --global alias.unstage 'reset HEAD --'
$ git config --global alias.last 'log -1 HEAD'
$ git config --global alias.flog 'log --pretty=format:"%Cgreen%h%Creset %an, %ar (%ad) : %s%Cred%d"'
$ git config --global alias.glog 'log --pretty=format:"%h %s" --graph'
```

- Los alias se invocan usando su nombre, Git simplemente sustituye el nuevo comando por lo que sea que hayas puesto en el alias, por lo que puedes añadir parámetros adicionales de comando original:

```
$ git flog  -10
```

Trabajar con un repositorio Git

- Descarga el historial de cambios del repositorio sin incorporar cambios
  $ git fetch
- Descarga los cambios del repositorio central y los mezcla en la rama local
  $ git pull
- Enumera todos los archivos nuevos o modificados de los cuales se van a guardar cambios
  $ git status
- Registra los cambios del archivo permanentemente en el historial de versiones
  $ git commit --m"[descriptive message]"
- Sube todos los commits de la rama local al repositorio central
  $ git push

Commit

- Un commit es una instantánea (snapshot) de los cambios en el repositorio. En el se guardan los ficheros nuevos o modificados que se han añadido al área de preparación. Cada commit tiene una serie de elementos
  - un SHA (Secure Hash Algorithm) que es un identificador único
  - un autor, una fecha y hora de creación
  - un confirmador, una fecha y hora de confirmación
  - un mensaje que describe los cambios realizados
  - una referencia al commit padre (excepto el primer commit)
  - un árbol cambios en los archivos y directorios
- Los mensajes de commit son una parte muy importante de Git. Un buen mensaje de commit debe
  - comienza con un verbo en imperativo con la inicial en mayúsculas
  - no se termina el mensaje con un punto
  - no ser más largo de 50 (70) caracteres
  - ser conciso y descriptivo, no debe ser un resumen de los cambios
  - contar su qué y sobre todo por qué
  - reflejar la granularidad del commit

Efectuar cambios

$ git add [paths]
- Guarda el estado del archivo en el área de preparación para realizar un commit

$ git status
- Enumera todos los archivos nuevos o modificados de los cuales se van a guardar cambios

$ git diff
- Muestra las diferencias entre archivos que no se han enviado aún al área de espera

$ git diff --staged
- Muestra las diferencias del archivo entre el área de espera y la última versión del archivo

$ git mv [file-original] [file-renamed]
- Cambia el nombre del archivo y lo prepara para ser guardado

$ git rm [paths]
- Borra el archivo del directorio activo y lo pone en el área de espera en un estado de eliminación

$ git rm --cached [paths]
- Retira el archivo del historial de control de versiones, pero preserva el archivo a nivel local

$ git commit -m "[descriptive message]"
- Registra los cambios del archivo permanentemente en el historial de versiones

$ git commit -a -m "[descriptive message]"
- Si quieres saltarte el área de preparación, preparando automáticamente todos los archivos rastreados antes de confirmarlos

Paths

- Los paths son los nombres de ficheros y directorios que se quieren incluir en una operación de Git.
- Su estructura sigue los criterios del sistema operativo y utiliza los siguientes caracteres especiales:
  - . : Directorio actual y subdirectorios
  - .. : Directorio padre
  - / : Separador de directorios
  - ~ : Directorio home del usuario
  - * : Comodín para cualquier cadena de caracteres
  - ? : Comodín para un solo carácter
  - [] : Comodín para un rango de caracteres

Etiquetado

- Git tiene la posibilidad de etiquetar puntos específicos del historial como importantes. Esta funcionalidad se usa típicamente para marcar versiones de lanzamiento (v1.0, por ejemplo).
- Listar las etiquetas disponibles en Git es sencillo. Simplemente escribe git tag:
  $ git tag
- Una etiqueta ligera es muy parecida a una rama que no cambia - simplemente es un puntero a un commit específico. Para crear una etiqueta ligera:
  $ git tag v1.0.0
- Sin embargo, las etiquetas anotadas se guardan en la base de datos de Git como objetos enteros:
  $ git tag -a v1.2.0 -m 'version 1 release 2'
- También puedes etiquetar commits mucho tiempo después de haberlos hecho.
  $ git tag v1.1.0 af2fda8
- Por defecto, el comando git push no transfiere las etiquetas a los servidores remotos. Si quieres enviar varias etiquetas a la vez, puedes usar la opción --tags del comando git push.
  $ git push origin --tags
- Puedes ver la información de la etiqueta junto con el commit que está etiquetado con:
  $ git show v1.4

SEMVER

- El sistema SEMVER (Semantic Versioning) es un conjunto de reglas para proporcionar un significado claro y definido a las versiones de los proyectos.
- El sistema SEMVER se compone de 3 números, siguiendo la estructura X.Y.Z:
  - X se denomina Major: indica cambios rupturistas
  - Y se denomina Minor: indica cambios compatibles con la versión anterior
  - Z se denomina Patch: indica resoluciones de bugs (compatibles)
- Básicamente, cuando se arregla un bug se incrementa el patch, cuando se introduce una mejora se incrementa el minor y cuando se introducen cambios que no son compatibles con la versión anterior, se incrementa el major.
- De este modo cualquier desarrollador sabe qué esperar ante una actualización de su librería favorita. Si sale una actualización donde el major se ha incrementado, sabe que tendrá que ensuciarse las manos con el código para pasar su aplicación existente a la nueva versión.

Repasar historial

$ git log
    Enumera el historial de versiones para la rama actual
$ git shortlog
    resume todos los commits
$ git show [commit]
    Produce metadatos y cambios de contenido del commit especificado
$ git diff [first-branch]...[second-branch]
    Muestra las diferencias de contenido entre dos ramas
$ git blame [file]
    Permite conocer el autor de la última modificación de cada línea de un fichero y en que commit se incluyó el cambio.
$ git checkout [commit]
    Regresa al pasado, sustituye el directorio de trabajo con una versión anterior
$ gitk es un visor gráfico del histórico.

Git muestra la información usando el comando less, que permite desplazarse por la información mostrada. Los comandos básicos son :h help, :f -> next page, :b -> previous page, :q -> quit

Repasar historial

$ git log: Enumera el historial de versiones para la rama actual
- -2 : Muestra solamente las últimas n confirmaciones.
- -p : Muestra las diferencias introducidas en cada confirmación
- --follow [file] : Enumera el historial de versiones para el archivo, incluidos los cambios de nombre
- --pretty=format:"%h - %an, %ar : %s" : Muestra las confirmaciones usando un formato alternativo. Las posibles opciones son oneline, short, full, fuller y format (mediante el cual puedes especificar tu propio formato).
- --graph: Muestra el historial de commits en forma de grafo
- --decorate: Muestra las referencias de los commits (HEAD, master, ...)
- --oneline: Muestra los commits en una sola línea
- --all: Muestra todos los commits, no solo los de la rama actual
- --author: Filtra los commits por autor
- --since --until: Filtran los commits por fecha
- --grep: Filtra los commits por mensaje
- --no-merges: Muestra solo los commits que no son merges
- --stat: Muestra estadísticas de los cambios en los commits
- --patch: Muestra los cambios en los commits

Referenciar confirmaciones

- Cada commit tiene asociado un código hash SHA-1 de 40 caracteres hexadecimales que lo identifica de manera única. Las referencias pueden ser absolutas:
  - SHA-1 (git log): Es el identificador del commit, pero es largo y difícil de recordar.
  - SHA-1 corto: Git es lo suficientemente inteligente como para descifrar el "commit" al que te refieres si le entregas los primeros caracteres, siempre y cuando la parte de SHA-1 sea de al menos 4 caracteres y no sea ambigua.
  - Branch: se puede usar el nombre de la rama (por defecto, hace referencia al último commit).
  - Tag: se puede usar el nombre de etiqueta
  - Message: se puede usar el mensaje asociado al commit (:/cadena)
  - Simbólica: el HEAD, puntero al commit actual
- Las referencias relativas permiten acceder a commits anteriores o posteriores a una referencia absoluta y utilizan los siguientes operadores como sufijos:
  - HEAD@{n} referencias obtenidas por git reflog
  - @{yesterday} referencias temporales
  - ~n n-ésimo ancestro del selector
  - ^ el padre del selector o ^^ el abuelo
- Se pueden establecer rangos entre dos selectores:
  - .. (dos puntos) Pide a Git que resuelva un rango de commits que es alcanzable desde un "commit" pero que no es alcanzable desde el otro. git log master..hotfix
  - ... (tres puntos) Especifica todos los commits que son alcanzables por alguna de las dos referencias, pero no por las dos al mismo tiempo. git log master...hotfix

Deshacer y rehacer

- Uno de las acciones más comunes a deshacer es cuando confirmas un cambio antes de tiempo y olvidas agregar algún archivo, o te equivocas en el mensaje de confirmación.

  $ git add ejemplo1
  $ git commit -m 'ejemplo'
  $ git add ejemplo2
  $ git commit **--amend** *-m 'ejemplos'*
- Se puede sacar el archivo del área de preparación, preservando su contenido

  $ git reset [file]
- Se pueden deshacer los cambios en un archivo sobre escribiéndolo con la versión anterior. Es un comando peligroso, el archivo con los cambios no se podrá recuperar. El comando retira el archivo del área de espera, si está en ella.

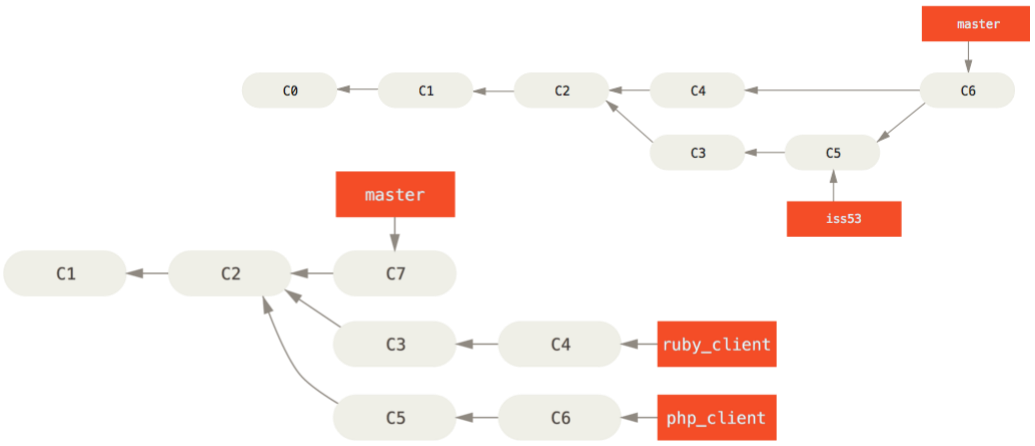
  $ git checkout -- [file]

- Advertencia: La regla de oro de Git establece que no se deben modificar commits que ya han sido compartidos.

Ramificaciones

- Git permite dividir la rama principal de desarrollo (master) y a partir de ahí continuar trabajando de forma independiente: solución de problemas, nueva versión, experimentación … Git promueve un ciclo de desarrollo donde las ramas se crean y se unen entre sí, incluso varias veces en el mismo día.
- Una rama Git es simplemente un apuntador móvil apuntando a una confirmación. La rama por defecto de Git es la rama master (main en GitHub). Con la primera confirmación de cambios que realicemos, se creará esta rama principal master apuntando a dicha confirmación. En cada confirmación de cambios que realicemos, la rama irá avanzando automáticamente.
- Se pueden crear nuevas ramificaciones (bifurcaciones) y saltar a ellas o a otras existentes en cualquier momento. Los cambios en una rama no afectan a las otras ramas. Mediante un apuntador especial, denominado HEAD, Git sabe en qué rama estás en cada momento y todas las operaciones por defecto se realizan en dicha rama.
- Una rama puede fusionarse (merge) con otra rama, momento en que podrán aparecer discrepancias si se han modificado archivos en paralelo, las que no pueda gestionar Git automáticamente se deberán resolver manualmente.

Ramificaciones



Ramificaciones

- Para crear una nueva ramificación:
  $ git branch hotfix
- Para saltar de una rama a otra se utiliza los comandos checkout y switch.
  $ git checkout hotfix
  $ git switch hotfix *(v2.23, este comando se centra exclusivamente en operaciones de ramas)*
- Para crear y saltar a la nueva ramificación:
  $ git checkout -b hotfix *(git switch -c hotfix)*
- Para crear la rama remota en el repositorio central y subir la primera vez:
  $ git push --set-upstream origin hotfix *(git push -u origin hotfix)*
- Los cambios, confirmaciones y otras se realizan sobre la rama actual. Una vez terminados los trabajos en la rama se pueden fusionar con la rama principal (si hay cambios en la principal se pueden producir conflictos que habrá que solucionar):
  $ git checkout master
  $ git merge hotfix -m "Merge branch 'hotfix'"
- Es importante borrar la rama fusionada que ya vamos a necesitar
  $ git branch -d hotfix

Guardado rápido

- No se puede cambiar de rama si hay cambios pendientes en el directorio de trabajo sin pasarlos al área de preparación, pero no se deben pasar al área de preparación cambios a medias. El guardado rápido toma tu directorio de trabajo actual, los archivos modificados y cambios almacenados, y lo guarda en un saco de cambios sin terminar que puedes volver a usar en cualquier momento.
- Para almacenar temporalmente todos los archivos modificados de los cuales se tiene al menos una versión guardada
  $ git stash
- Para enumerar todos los grupos de cambios que están guardados temporalmente
  $ git stash list
- Para restaurar los archivos guardados más recientemente
  $ git stash pop
- Para elimina el grupo de cambios más reciente que se encuentra guardado temporalmente
  $ git stash drop
  $ git clean -d -n

Trabajar con Remotos

- Los repositorios remotos son versiones de tu proyecto que están hospedadas en Internet o en cualquier otra red. Puedes tener varios de ellos, y en cada uno tendrás generalmente permisos de solo lectura o de lectura y escritura. Colaborar con otras personas implica gestionar estos repositorios remotos enviando y trayendo datos de ellos cada vez que necesites compartir tu trabajo.

  $ git clone https://github.com/myuser/myrepository.git

  $ git remote -v
- Para añadir repositorios remotos:

  $ git remote add other https://github.com/otheruser/myrepository.git

  $ git fetch other *($ git pull <remote> <branch>)*

  $ git remote show other
- Si quieres cambiar el nombre de la referencia de un remoto:

  $ git remote rename other colabora
- Si por alguna razón quieres eliminar un remoto:

  $ git remote remove colabora

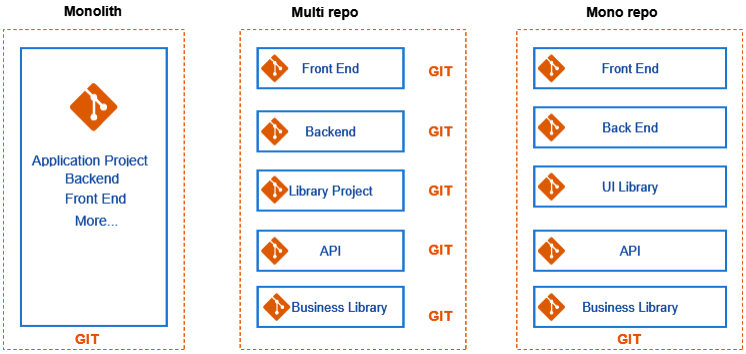
Entornos distribuidos

- La capacidad de Git de trabajar con remotos permite la implementación de flujos de trabajo distribuidos. Git permite contribuir adecuadamente a un proyecto, de manera fácil tanto para ti como para el responsable del proyecto, y mantener adecuadamente un proyecto con múltiples desarrolladores. Git permite trabajar en un entorno distribuido como colaborador o como integrador.
- En los entornos distribuidos se establece una diferencia entre autor (author) y confirmador (committer). El autor es la persona que escribió originalmente el trabajo, mientras que el confirmador es quien lo aplicó. Por tanto, si mandas un parche a un proyecto, y uno de sus miembros lo aplica, ambos recibiréis reconocimiento, tú como autor y el miembro del proyecto como confirmador.

  $ git log --pretty=fuller

Estilos de desarrollo

- Monolito: un proyecto en un repositorio.
- Múltirepo: múltiples proyectos, un repositorio por proyecto.
- Monorepo: múltiples proyectos en un único repositorio.



Submódulos

- Los submódulos permiten mantener un repositorio de Git como un subdirectorio de otro repositorio de Git. Esto le permite clonar otros repositorios en su proyecto y mantener sus commits separados. Los submódulos agregarán el subproyecto en un directorio llamado igual que el repositorio.
  $ git submodule add https://github.com/myuser/subproject.git
- El archivo .gitmodules, en la raíz del repositorio principal, es un archivo de configuración que almacena la asignación entre la URL del proyecto y el subdirectorio local en el que lo ha insertado. Si tienes múltiples submódulos, tendrá múltiples entradas en este archivo. Otras personas, cuando clonan este proyecto, saben de dónde obtener los proyectos de submódulos.
- Al cambiarte al directorio del submódulo, pasas al repositorio del submódulo y se maneja con los comandos habituales. El comando foreach de submódulo permite ejecutar algún comando arbitrario en cada submódulo:
  $ git submodule foreach 'git status'
- Hay una forma más sencilla buscar y fusionar manualmente los subdirectorios:
  $ git submodule update --remote
- Se puede pedir a Git que verifique que todos sus submódulos se hayan elevado correctamente antes de empujar el proyecto principal:
  $ git push --recurse-submodules=check

Arboles de trabajo

- Uno de los problemas más habituales en Git es tener que modificar una rama distinta a la que tenemos actualmente. Eso implica que, si estamos en medio de un trabajo, tendríamos que hacer un commit o un stash, lo cual a veces es bastante molesto.
- Con los arboles de trabajo (worktree) podemos crear un subdirectorio de trabajo asociado a una rama que contenga su propio repositorio local privado (.git). Las operaciones realizadas en el subdirectorio se realizan sobre la rama asociada en el repositorio privado local, aunque use el mismo repositorio remoto. Se puede crear la rama a la vez que el subdirectorio.

  ```
  $ git worktree add -b emergency-fix ./emergency-fix master
  $ git worktree add ./emergency-fix emergency-fix
  $ cd emergency-fix
  …
  $ git commit -am "modificado en emergency-fix"
  $ git push --set-upstream origin emergency-fix        solo la primera vez
  ```
- Para mostrar el listado de directorios y espacios de trabajo.

  ```
  $ git worktree list
  ```
- Para borrarar un espacio de trabajo y, posteriormente, eliminar de subdirectorio de trabajo.

  ```
  $ git worktree remove ./emergency-fix
  $ git worktree prune
  ```
- Los arboles de trabajo (git worktree) cuentan con comandos adicionales como move, repair, lock o unlock.

Integrar cambios

- La integración de cambios se produce cuando se combinan los cambios de diferentes orígenes (ramas, remotos, …) y hay dos formas de hacerlo: la fusión (merge) y la reorganización (rebase).
- La fusión mezcla los cambios de la rama indicada con la confirmación actual o rama actual creando una nueva versión (git diff permite consultas los cambios a integrar).

  ```
  $ git switch master
  $ git merge hotfix -m "Merge branch 'hotfix'"
  ```
- Cuando intentas fusionar una rama con otra rama accesible siguiendo directamente el historial de la primera; Git simplifica las cosas avanzando el puntero, ya que no hay ningún otro trabajo divergente a fusionar, aplicando los cambios a la rama actual dejándolos pendientes de confirmación. Esto es lo que se denomina “avance rápido” (“fast forward”).
- Cuando la rama actual a evolucionado (commits) desde que se hizo la bifurcación de la rama a fusionar, Git realizará una fusión a tres bandas, utilizando las dos instantáneas apuntadas por el extremo de cada una de las ramas y por el ancestro común a ambas. Git identifica automáticamente el mejor ancestro común para realizar la fusión de las ramas. En lugar de simplemente avanzar el apuntador de la rama, Git crea una nueva instantánea (snapshot) resultante de la fusión a tres bandas; y crea automáticamente una nueva confirmación (commit) que apunta a ella. Este proceso se conoce como "fusión confirmada" y su particularidad es que tiene más de un padre.

Resolución de conflictos

- Un merge conflict se produce cuando hay discrepancias al fusionar versiones que entran en conflicto y Git no puede determinar automáticamente qué es correcto:
  - La versión original modificada es distinta de la versión actual.
  - Difieren las versiones en diferentes ramas.
  - Se va a sobre escribir un cambio local pendiente.
- Podemos sacar más información ejecutando el comando git status.
- Git agrega algunas líneas adicionales en el archivo para marcar el conflicto:

```
<<<<<<< HEAD
=======
>>>>>>> commit_name
```

- La línea ======= es el "centro" del conflicto. Todo el contenido entre el centro y la línea <<<<<<< HEAD es contenido que existe en la rama principal actual a la que apunta la referencia HEAD. Por el contrario, todo el contenido entre el centro y >>>>>>> commit_name es contenido que está presente en la nueva fusión.
- La forma más directa de resolver un conflicto de fusión manualmente es editar el archivo conflictivo, eliminar las marcas, dejando una de las dos secciones, las dos o ninguna. Las utilidades de los entornos pueden ayudar a resolver los conflictos.

```
$ git mergetool --tool-help
```

- Una vez resuelto el conflicto o conflictos, se debe preparar el nuevo contenido fusionado con git add y confirmar la solución con git commit.

Rerere

- Git dispone de una utilidad llamada “rerere” que puede resultar útil si estás haciendo muchas integraciones y reorganizaciones, o si mantienes una rama puntual de largo recorrido.
- Rerere significa “reuse recorded resolution” (reutilizar resolución grabada), es una forma de simplificar la resolución de conflictos. Cuando “rerere” está activo, Git mantendrá un conjunto de imágenes anteriores y posteriores a las integraciones correctas, de forma que, si detecta que hay un conflicto que parece exactamente igual a otro ya corregido previamente, usará esa misma corrección sin causarte molestias. Esta funcionalidad consta de dos partes: un parámetro de configuración y un comando.
- El parámetro de configuración es rerere.enabled y es bastante útil ponerlo en tu configuración global (la resolución se grabará en la caché por si la necesitas en un futuro):

```
$ git config --global rerere.enabled true
```

- Si fuera necesario, puedes interactuar con la caché de “rerere” usando:

```
$ git rerere.
```

- Sin ningún parámetro adicional, Git comprueba su base de datos de resoluciones en busca de coincidencias con cualquier conflicto durante la integración actual e intenta resolverlo (lo hace automáticamente si rerere.enabled sea true). También existen subcomandos para ver qué se grabará, para eliminar de la caché una resolución específica y para limpiar completamente la caché.

Solicitud de incorporación de cambios

- Las pull requests, solicitud de incorporación de cambios, son una funcionalidad disponible en muchos servidores de GIT (GitHub, …) que facilita la colaboración entre desarrolladores.
- En su forma más sencilla, las solicitudes de incorporación de cambios son un mecanismo para que los desarrolladores notifiquen a los miembros de su equipo que han terminado una función. Una vez la rama de función está lista, el desarrollador realiza la solicitud de incorporación de cambios mediante el repositorio central. Así, todas las personas involucradas saben que deben revisar el código y fusionarlo con la rama principal.
- Pero la solicitud de incorporación de cambios es mucho más que una notificación: es un foro especializado para debatir sobre una función propuesta. Si hay algún problema con los cambios, los miembros del equipo pueden publicar feedback en las solicitudes de incorporación de cambios e incluso modificar la función al enviar confirmaciones de seguimiento. El seguimiento de toda esta actividad se realiza directamente desde la solicitud de incorporación de cambios.
- Cuando realizas una pull request, lo que haces es solicitar que otro desarrollador (el mantenedor del proyecto) incorpore (o haga un pull) una rama de tu repositorio (fork) al suyo.

Reorganizar

- La manera más sencilla de integrar ramas es la fusión. Sin embargo, también hay otra forma de hacerlo: puedes capturar todos los cambios confirmados en una rama y re aplicarlos sobre otra. Esto es lo que en Git se denomina reorganizar (rebasing):

  $ git switch hotfix

  $ git rebase master
- Haciendo que Git vaya al ancestro común de ambas ramas (donde estás actualmente y de donde quieres reorganizar), saque las diferencias introducidas por cada confirmación en la rama donde estás, guarde esas diferencias en archivos temporales, reinicie (reset) la rama actual hasta llevarla a la misma confirmación que la rama de donde quieres reorganizar, y finalmente, vuelva a aplicar ordenadamente los cambios.

  $ git switch master

  $ git merge hotfix
- El reorganizar nos deja un historial más claro (lineal), para cuando quieras estar seguro de que tus confirmaciones (commits) se pueden aplicar limpiamente sobre una rama remota, pero deja de ser un registro de todo lo que ha pasado dado que reescribimos la historia.

Reorganizar

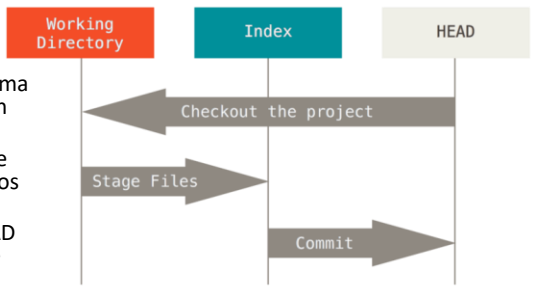
- El rebase tiene un modo interactivo que permite reescribir la historia de un repositorio, cambiando el orden de los commits, modificando los mensajes de los commits, eliminando commits, fusionando commits, …

  $ git rebase -i
- Los comandos más comunes son
  - pick: utiliza el commit
  - reword: utiliza el commit, pero permite cambiar el mensaje
  - edit: utiliza el commit, pero para en él para hacer cambios
  - squash: fusiona el commit con el anterior
  - fixup: fusiona el commit con el anterior, pero mantiene el mensaje del anterior
  - drop: elimina el commit
- Otra forma de mover trabajo de una rama a otra es entresacarlo (cherry-pick), como hacer un rebase para un único commit. Toma el parche introducido en un commit e intenta reaplicarlo en la rama en la que estás actualmente. Esto es útil si hay varios commits en una rama y sólo quieres integrar uno de ellos, o si sólo tienes un commit en una rama puntual y prefieres entresacarlo en lugar de hacer una reorganización (rebase). Si en la rama actual se quiere añadir un commit de otra rama:

  $ git cherry-pick <commit>

Reset y Checkout

- El HEAD es el puntero a la referencia de bifurcación actual, que es, a su vez, un puntero al último commit realizado en esa rama, eso significa que HEAD será el padre del próximo commit que se cree.
- El Index es el siguiente commit propuesto, el Área de Preparación, que Git rellena con una lista de todos los contenidos del archivo que fueron revisados (git add) por última vez en tu directorio de trabajo y cómo se veían cuando fueron revisados originalmente.
- El directorio de trabajo es una caja de arena, un file system de conveniencia, donde aparecen, se sustituyen y desaparecen los ficheros respaldados en la base de datos Git.
- El cambio de rama o la confirmación clonación cambia el HEAD para que apunte a la nueva “ref” de la rama, rellena su Índice con la instantánea de esa confirmación y luego copia los contenidos del Índice en tu Directorio de Trabajo.
- Los comandos reset y checkout cambian directamente HEAD, Index y WorkingDir, por lo que pueden ser peligrosos y requieren conocer bien estos conceptos.



RefLog

- Una de las cosas que Git hace en segundo plano, mientras tu estás trabajando a distancia, es mantener un “reflog”, un log de a dónde se apuntan las referencias de tu HEAD y tu rama en los últimos meses.

  $ git reflog
- El comando git reflog examina un registro de donde han estado todas las cabezas de tus ramas mientras trabajas para encontrar commits que puedes haber perdido a través de la reescritura de historias.

  $ git log -g

Checkout

- El comando checkout mueve el puntero de referencia HEAD a un commit específico.
  - El último de otra rama (lo habitual), con el nombre de la rama
  - En la misma rama.
- El movimiento puede ser de commit completo o restringirse a un fichero concreto del commit.

  $ git checkout HEAD~3
- El checkout a nivel de archivo, en lugar de mover el HEAD, lleva al directorio de trabajo el fichero del checkout con el contenido que tenía en el commit especificado (recupera una versión anterior) y lo deja como modificado o staged (los cambios actuales se pierden):

  $ git checkout HEAD~3 README.md
- Para recuperar el valor original del último commit:

  $ git checkout HEAD README.md
- El comando retira el archivo del área de espera, si está en ella.

  $ git checkout -- [file]

Reset

- El comando reset mueve el puntero de referencia de una rama (acompañado por el HEAD), a un commit específico, normalmente un commit anterior de la misma rama. Estaremos 'deshaciendo' los commits posteriores que quedarán huérfanos y se eliminarán la próxima vez que Git haga limpieza.
- Sus efectos sobre las working y staging areas dependen de la opción seleccionada:
  - soft: no se modifican las working y staging areas. Cambia todos los archivos a "cambios pendientes de confirmar".
  - mixed: (valor por defecto) el contenido del commit apuntado por la rama se refleja en la working area
  - hard: el contenido del commit apuntado por la rama se refleja en las working y staging areas
- Deshace los últimos cambios dejando en el staging area los cambios ya confirmados:
  $ git reset --soft [commit]
- Deshace todos los commits después de [commit], preservando los cambios localmente
  $ git reset [commit]
- Desecha todo el historial y regresa al commit especificado
  $ git reset --hard [commit]

PELIGRO: Es un comando difícilmente reversible (salvo que se preserve un repositorio central) o, directamente, irreversible. Una vez movido el HEAD no aparecen los commit posteriores en el log.

Restore (v2.23)

- Git permite restaurar las rutas especificadas en el árbol de trabajo con contenido de una fuente de restauración. Si se rastrea una ruta, pero no existe en la fuente de restauración, se eliminará para que coincida con la fuente.
- El comando también se puede utilizar para restaurar el contenido del índice con --staged, o restaurar tanto el árbol de trabajo como el índice con --staged --worktree.
- Para restaurar el fichero desde el índice:
  $ git restore README.md
- Para revertir el fichero dos revisiones anteriores:
  $ git restore --source master~2 README.md
- Para restaurar todos los archivos en el directorio actual
  $ git restore .
- Para restaurar un archivo en el índice para que coincida con la versión en HEAD (lo mismo que usar git reset)
  $ git restore --staged README.md

Administración

- Para limpiar el árbol de trabajo eliminando recursivamente los archivos que no están bajo control de versiones, comenzando desde el directorio actual.

  $ git clean
- Los ficheros grandes borrados que penalizan las clonaciones, ficheros con secretos, … para borrar realmente un archivo de tu historial Git, has de reescribir todas las confirmaciones desde su incorporación:

  $ git filter-branch --tree-filter 'rm -f passwords.txt' HEAD

  $ git filter-branch --force --index-filter "git rm --cached --ignore-unmatch *.tgz" --prune-empty --tag-name-filter cat -- --all

  $ git push origin --force --all

  Estas técnicas son destructivas y alteran el historial de confirmaciones de cambio.

Hooks

- Git cuenta con mecanismos para lanzar scripts de usuario cuando suceden ciertas acciones importantes, llamados puntos de enganche (hooks). Hay dos grupos de esos puntos de lanzamiento: los del lado cliente y los del lado servidor. Los puntos de enganche del lado cliente están relacionados con operaciones tales como la confirmación de cambios (commit) o la fusión (merge). Los del lado servidor están relacionados con operaciones tales como la recepción de contenidos enviados (push) a un servidor. Estos puntos de enganche pueden utilizarse para multitud de aplicaciones.
- Los puntos de enganche se guardan en la subcarpeta hooks de la carpeta Git. En la mayoría de proyectos, estará en .git/hooks. Por defecto, esta carpeta contiene unos cuantos scripts de ejemplo. Algunos de ellos son útiles por sí mismos; pero su misión principal es la de documentar las variables de entrada para cada script. Todos los ejemplos se han escrito como scripts de shell, con algo de código Perl embebido en ellos. Pero cualquier tipo de script ejecutable que tenga el nombre adecuado puede servir igual de bien. Los puedes escribir en Ruby o en Python o en cualquier lenguaje de scripting con el que trabajes. Si quieres usar los ejemplos que trae Git, tendrás que renombrarlos, ya que los ejemplos acaban su nombre en .sample.
- Para activar un punto de enganche para un script, pon el archivo correspondiente en la carpeta hooks; con el nombre adecuado y con la marca de ejecutable. A partir de ese momento, será automáticamente lanzado cuando se dé la acción correspondiente. Advertencia: los puntos de enganche del lado del cliente no se copian cuando clonas el repositorio.

Tareas de bajo nivel

- Existen unos cuantos comandos para realizar tareas de bajo nivel (comandos de fontanería) y que se diseñaron para poder ser utilizados de forma encadenada al estilo UNIX, para ser utilizados en scripts, para conocer/manipular la estructura interna de Git o la resolución de problemas. Estos comandos son los que utiliza el propio Git internamente para construir los comandos de más alto nivel conocidos como los "comandos de porcelana".
  - cat-file
  - check-ignore
  - checkout-index
  - commit-tree
  - count-objects
  - diff-index
  - for-each-ref
  - hash-object
  - ls-files
  - ls-tree
  - merge-base
  - read-tree
  - rev-list
  - rev-parse
  - show-ref
  - symbolic-ref
  - update-index
  - update-ref
  - verify-pack
  - write-tree

COMANDOS

Comandos

- Configurar: Configura la información del usuario para todos los repositorios locales
  - $ git config --global user.name "[name]"
    - Establece el nombre que estará asociado a tus commits
  - $ git config --global user.email "[email address]"
    - Establece el e-mail que estará asociado a sus commits
- Crear repositorios: Inicializa un nuevo repositorio u obtiene uno de una URL existente
  - $ git init [project-name]
    - Crea un nuevo repositorio local con el nombre especificado
  - $ git remote add origin [url]
    - Especifica el repositorio remoto para su repositorio local
  - $ git clone [url]
    - Descarga un proyecto y todo su historial de versiones
- Suprimir el seguimiento de cambios: Un archivo de texto llamado .gitignore suprime la creación accidental de versiones para archivos y rutas que concuerdan con los patrones especificados
  - $ git ls-files --others --ignored --exclude-standard
    - Enumera todos los archivos ignorados en este proyecto

Comandos

- Efectuar cambios
  - $ git status
    - Enumera todos los archivos nuevos o modificados de los cuales se van a guardar cambios
  - $ git diff
    - Muestra las diferencias entre archivos que no se han enviado aún al área de espera
  - $ git diff --staged
    - Muestra las diferencias del archivo entre el área de espera y la última versión del archivo
  - $ git add [file]
    - Guarda el estado del archivo en preparación para realizar un commit
  - $ git reset [file]
    - Mueve el archivo del área de espera, pero preserva su contenido
  - $ git mv [file-original] [file-renamed]
    - Cambia el nombre del archivo y lo prepara para ser guardado
  - $ git rm [file]
    - Borra el archivo del directorio activo y lo pone en el área de espera en un estado de eliminación
  - $ git rm --cached [file]
    - Retira el archivo del historial de control de versiones, pero preserva el archivo a nivel local
  - $ git commit -m"[descriptive message]"
    - Registra los cambios del archivo permanentemente en el historial de versiones
  - $ git commit -a -m"[descriptive message]"
    - Si quieres saltarte el área de preparación, preparando automáticamente todos los archivos rastreados antes de confirmarlos

Comandos

- Sincronizar cambios: Registrar un marcador para un repositorio e intercambiar historial de versiones
  - $ git fetch [bookmark]
    - Descarga todo el historial del marcador del repositorio sin incorporar cambios
  - $ git merge [bookmark]/[branch]
    - Combina la rama del marcador con la rama local actual
  - $ git push [alias] [branch]
    - Sube todos los commits de la rama local al repositorio central
  - $ git pull
    - Descarga el historial del marcador e incorpora cambios
- Branches (cambios grupales): Nombra una serie de commits y combina esfuerzos ya completados
  - $ git branch
    - Enumera todas las ramas en el repositorio actual
  - $ git branch [branch-name]
    - Crea una nueva rama
  - $ git checkout [branch-name]
    - Cambia a la rama especificada y actualiza el directorio activo
  - $ git merge [branch-name]
    - Combina el historial de la rama especificada con la rama actual
  - $ git branch -d [branch-name]
    - Borra la rama especificada

Comandos

- Repasar historial: Navega e inspecciona la evolución de los archivos de proyecto
  - $ git log
    - Enumera el historial de versiones para la rama actual
  - $ git log --follow [file]
    - Enumera el historial de versiones para el archivo, incluidos los cambios de nombre
  - $ git diff [first-branch]...[second-branch]
    - Muestra las diferencias de contenido entre dos ramas
  - $ git show [commit]
    - Produce metadatos y cambios de contenido del commit especificado
- Rehacer commits: Borra errores y elabora un historial de reemplazo
  - $ git reset [commit]
    - Deshace todos los commits después de [commit], preservando los cambios localmente
  - $ git reset --hard [commit]
    - Desecha todo el historial y regresa al commit especificado

Comandos

- Guardar fragmentos: Almacena y restaura cambios incompletos

  $ git stash

  Almacena temporalmente todos los archivos modificados de los cuales se tiene al menos una versión guardada

  $ git stash pop

  Restaura los archivos guardados más recientemente

  $ git stash list

  Enumera todos los grupos de cambios que están guardados temporalmente

  $ git stash drop

  Elimina el grupo de cambios más reciente que se encuentra guardado temporalmente

GitHub

- echo "# Mi repositorio" >> README.md
- git init
- git add .
- git commit -m "initial commit"
- git remote add origin https://github.com/myuser/myrepository.git
- git branch -M main
- git push -u origin main

- https://github.com/github/gitignore

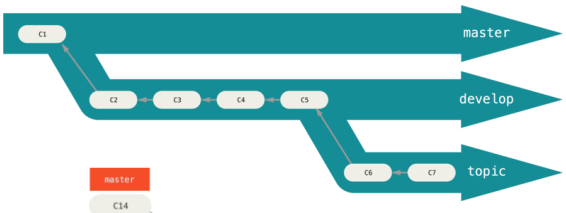
FLUJOS DE TRABAJO

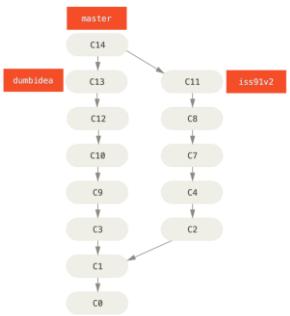
Flujo de trabajo

- Git es un sistema de control de versiones que suministra las herramientas para guardar y recuperar versiones, en local y remoto, realizar bifurcaciones y fusiones, … (lo que podemos hacer).
- Git plantea una gran libertad en la forma de trabajar en torno a un proyecto: pequeño o grande, individual o colaborativos, trivial o critico, …
- Sin embargo, para coordinar el trabajo de un grupo de personas en torno a un proyecto es necesario acordar como se va a trabajar con Git. A estos acuerdos se les llama flujo de trabajo (como lo vamos a hacer).
- Un flujo de trabajo de Git es una pauta o una recomendación acerca del uso de Git para realizar trabajo de forma uniforme y productiva. Los flujos de trabajo más populares son git-flow, GitHub-flow y One Flow.

Flujos de trabajo

- Ramas de Largo Recorrido:
  - El fusionar una rama a otra varias veces a lo largo del tiempo es fácil de hacer en Git. Esto te posibilita tener varias ramas siempre abiertas, e irlas usando en diferentes etapas del ciclo de desarrollo; realizando fusiones entre ellas.
  - Es habitual mantener ramas únicamente con el código totalmente estable (el código que ha sido o que va a ser liberado) y otras ramas paralelas en las que trabajan y realizan pruebas. Estas ramas paralelas no suelen estar siempre en un estado estable; pero cada vez que sí lo están, pueden ser fusionadas con una rama estable.
- Ramas Puntuales
  - Una rama puntual es aquella rama de corta duración que abres para un tema o para una funcionalidad determinada y eliminas una vez resuelto.
  - En Git es muy habitual el crear, trabajar con, fusionar y eliminar ramas varias veces al día.





Flujos de trabajo centralizado

- En sistemas centralizados, habitualmente solo hay un modelo de colaboración, el flujo de trabajo centralizado. Un repositorio o punto central que acepta código y todos sincronizan su trabajo con él. Unos cuantos desarrolladores son nodos de trabajo, consumidores de dicho repositorio, y sincronizan con ese punto.
- Esto significa que, si dos desarrolladores clonan desde el punto central, y ambos hacen cambios, solo el primer desarrollador en subir sus cambios lo podrá hacer sin problemas. El segundo desarrollador debe fusionar el trabajo del primero antes de subir sus cambios, para no sobrescribir los cambios del primero.
- Este flujo de trabajo es atractivo para mucha gente porque es un paradigma con el que muchos están familiarizados y cómodos, pero requiere organización y disciplina para evitar desperdiciar un tiempo excesivo realizando las fusiones de los cambios cuando entran en conflictos.
- Para evitar estos conflictos es conveniente definir unas pautas que establezcan quien, cuando y como puede modificar los ficheros y realizar las confirmaciones.

Flujo de Trabajo Administrador-Integración

- Debido a que Git permite tener múltiples repositorios remotos, es posible tener un flujo de trabajo donde cada desarrollador tenga acceso de escritura a su propio repositorio público y acceso de lectura a todos los demás. Este escenario a menudo incluye un repositorio canónico que representa el proyecto "oficial". Para contribuir a ese proyecto, creas tu propio clon público del proyecto y haces pull con tus cambios. Luego, puede enviar una solicitud al administrador del proyecto principal para que agregue los cambios. Entonces, el administrador agrega el repositorio como remoto, prueba los cambios localmente, los combina en su rama y los envía al repositorio. El proceso funciona de la siguiente manera:
  1. El administrador del proyecto hace un push al repositorio público.
  2. El contribuidor clona ese repositorio y realiza los cambios.
  3. El contribuidor realiza un push con su copia pública del proyecto.
  4. El contribuidor envía un correo electrónico al administrador pidiendo que haga pull de los cambios.
  5. El administrador agrega el repositorio del contribuidor como remoto y fusiona ambos localmente.
  6. El administrador realiza un push con la fusión del código al repositorio principal.

Flujo de Trabajo Dictador-Tenientes

- Esta es una variante de un flujo de trabajo de múltiples repositorios. Generalmente es utilizado por grandes proyectos con cientos de colaboradores.
- Varios administradores de integración están a cargo de ciertas partes del repositorio. Se les llaman “tenientes”. Todos los tenientes tienen un gerente de integración conocido como el “dictador benevolente”. El repositorio del dictador benevolente sirve como el repositorio de referencia del cual todos los colaboradores necesitan realizar pull.
- El proceso funciona así:
  1. Los desarrolladores trabajan en su propia rama especifica y fusionan su código en la rama master, la cual, es una copia de la rama del dictador.
  2. Los tenientes fusionan el código de las ramas master de los desarrolladores en sus ramas master de tenientes.
  3. El dictador fusiona la rama master de los tenientes a su rama master de dictador.
  4. El dictador hace push del contenido de su rama master al repositorio para que otros fusionen los cambios a sus ramas.

Flujos de trabajo comunes

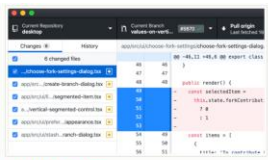
- Git-Flow (Creado en 2010 por Vincent Driessen)
  - Es el flujo de trabajo más conocido. Está pensado para aquellos proyectos que tienen entregables y ciclos de desarrollo bien definidos. Está basado en dos grandes ramas con infinito tiempo de vida (ramas master y develop) y varias ramas de apoyo, unas orientadas al desarrollo de nuevas funcionalidades (feature-*), otras al arreglo de errores (hotfix-*) y otras a la preparación de nuevas versiones de producción (release-*).
- GitHub-Flow (Creado en 2011 por el equipo de GitHub)
  - Es la forma de trabajo sugerida por las funcionalidades propias de GitHub e intenta simplificar la gestión de ramas, trabajando directamente sobre la rama master y generando e integrando las distintas features directamente a esta rama. Está centrado en un modelo de desarrollo iterativo y de despliegue constante. Está basado en cuatro principios:
    - Todo lo que está en la rama master está listo para ser puesto en producción
    - Para trabajar en algo nuevo, debes crear una nueva rama a partir de la rama master con un nombre descriptivo. El trabajo se irá integrando sobre esa rama en local y regularmente también a esa rama en el servidor
    - Cuando se necesite ayuda o información o cuando creemos que la rama está lista para integrarla en la rama master, se debe abrir una pull request (solicitud de integración de cambios).
    - Alguien debe revisar y visar los cambios para fusionarlos con la rama master. que integrados se pueden poner en producción.
- One Flow (Creado en 2015 por Adam Ruka)
  - En él cada nueva versión de producción está basada en la versión previa de producción. La mayor diferencia con Git Flow es que no tiene rama de desarrollo.

Pautas y buenas prácticas

- Desarrollar mediante ramas, habiendo establecido una estrategia de ramificación, fusión y des ramificación
- Acordar una convención de nomenclatura coherente
- Acordar un formato de código fuente común
- Escribir mensajes de confirmación descriptivos
- Realizar cambios incrementales y pequeños
- Mantener confirmaciones atómicas y sin errores
- No subir ficheros innecesarios al repositorio
- Evitar las confirmaciones indiscriminadas que cubran múltiples aspectos
- Fusionar con frecuencia y con cuidado
- Obtener antes de empezar, obtener antes de enviar
- Aprender y mejorar continuamente

PROGRAMAS CLIENTE PARA OPERAR CON GIT

GUI Clients

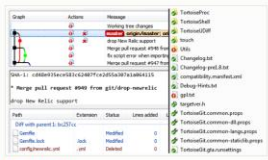


GitHub Desktop
Platforms: Mac, Windows
Price: Free
License: MIT

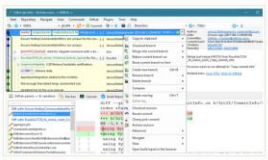
SourceTree
Platforms: Mac, Windows
Price: Free
License: Proprietary

GitKraken Desktop
Platforms: Linux, Mac, Windows
Price: Free / $48+/user annually
License: Proprietary

Magit
Platforms: Linux, Mac, Windows
Price: Free
License: GNU GPL

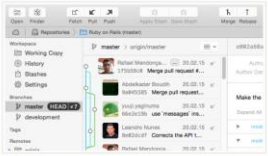


TortoiseGit
Platforms: Windows
Price: Free
License: GNU GPL



Git Extensions
Platforms: Windows
Price: Free
License: GNU GPL

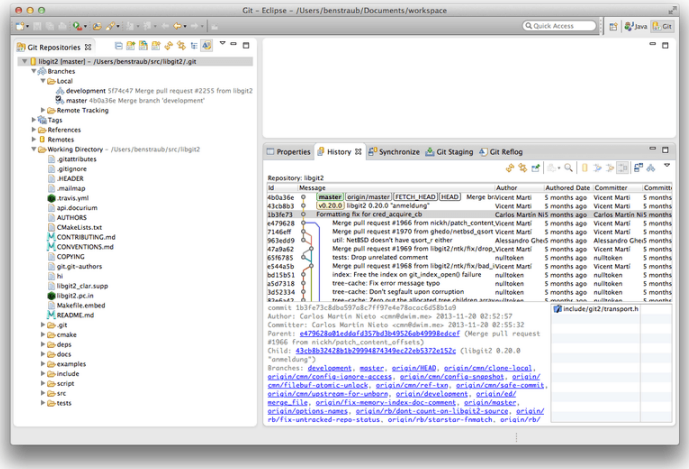
SmartGit
Platforms: Linux, Mac, Windows
Price: Free for non-commercial use / $59/user annually
License: Proprietary



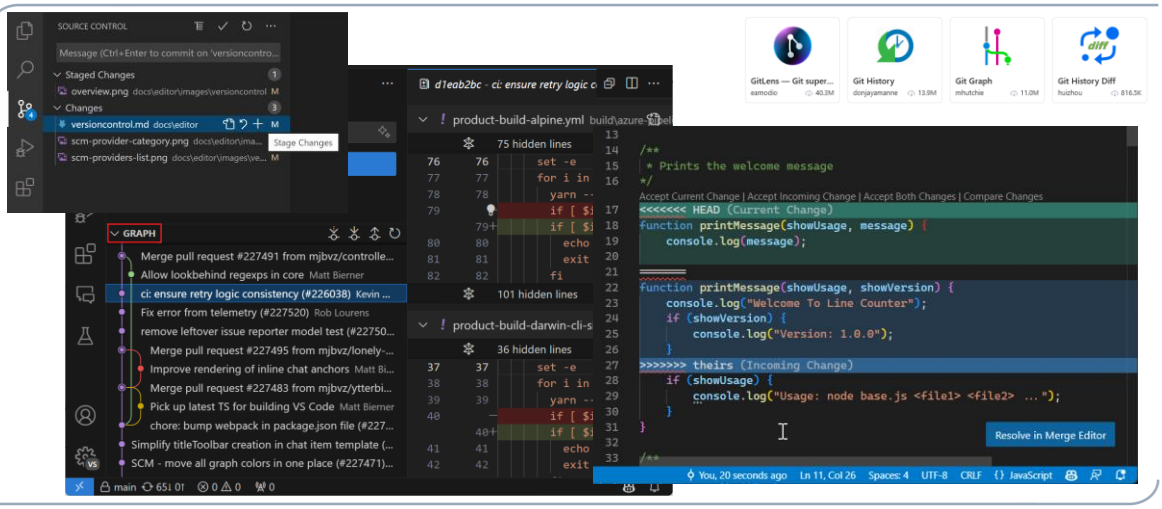
Tower
Platforms: Mac, Windows
Price: $69+/user annually (Free 30-day trial)
License: Proprietary

https://git-scm.com/downloads/guis

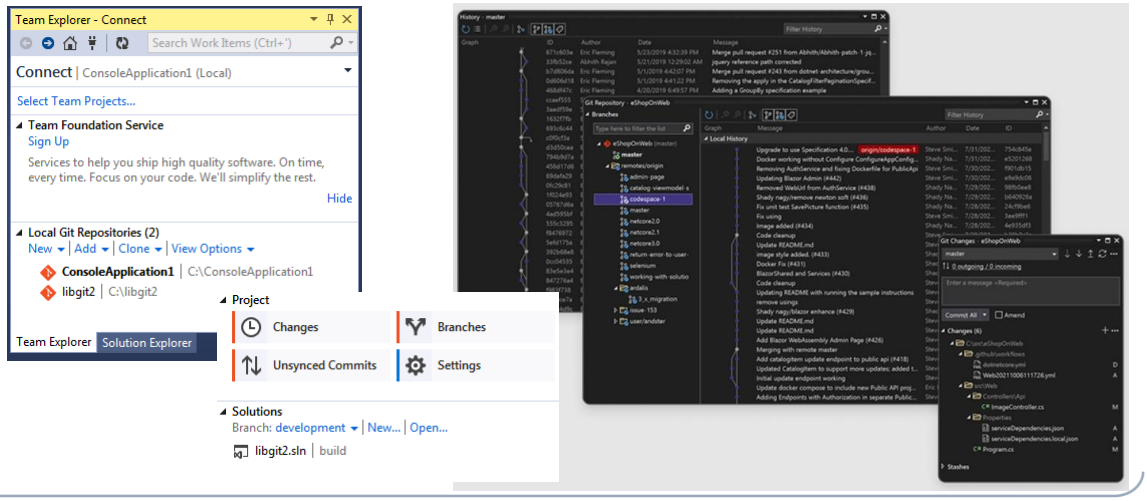
Git en Eclipse



Git en Visual Studio Code



Git en Visual Studio



INTEGRACIÓN CON MAVEN

Integración con Maven

- Apache Maven SCM permite gestionar directamente acciones sobre el SCM en el que se encuentra el proyecto, para ello es necesario configurar el elemento <scm> del pom.xml. Es posible conectarse a varios tipos de repositorios como pueden ser Git o Subversion.

```
<scm>
  <connection>scm:git:https://github.com/myuser/myrepository.git</connection>
  <developerConnection> scm:git:https://github.com/myuser/myrepository.git </developerConnection>
  <tag>HEAD</tag> <!-- especifica la etiqueta bajo la que se encuentra este proyecto, HEAD (raíz) -->
  <url>https://github.com/myuser/myrepository</url> <!-- repositorio accesible públicamente -->
</scm>
```

- Donde connection requiere acceso de lectura para que Maven pueda encontrar el código fuente y developerConnection requiere una conexión que otorgue acceso de escritura.
- La configuración incluida en cada proyecto se puede utilizar para gestionar mediante Maven las acciones propias de un sistema gestor de la configuración, como pueden ser checkout, update o commit, además se podrá realizar a través de plugins la generación de versiones release creando un tag automáticamente. Es importante configurar la conexión de forma completa, aunque no se realice la gestión de la configuración mediante Apache Maven, ya que cierta información será explotada desde los sistemas externos corporativos como Jenkins y SonarQube.

Maven Goals

- El Maven SCM Plugin dispone de los siguientes objetivos:
  - scm:add - comando para agregar archivo
  - scm:bootstrap - comando para verificar y construir un proyecto
  - scm:branch - ramifica el proyecto
  - scm:changelog - comando para mostrar las revisiones del código fuente
  - scm:check-local-modification - falla la compilación si hay alguna modificación local
  - scm:checkin - comando para confirmar cambios
  - scm:checkout - comando para obtener el código fuente
  - scm:diff - comando para mostrar la diferencia de la copia de trabajo con la remota
  - scm:edit - comando para iniciar la edición en la copia de trabajo
  - scm:export - comando para obtener una nueva copia exportada
  - scm:list - comando para obtener la lista de archivos del proyecto
  - scm:remove - comando para marcar un conjunto de archivos para su eliminación
  - scm:status - comando para mostrar el estado de scm de la copia de trabajo
  - scm:tag - comando para etiquetar una determinada revisión
  - scm:unedit - comando para detener la edición de la copia de trabajo
  - scm:update - comando para actualizar la copia de trabajo con los últimos cambios
  - scm:update-subprojects : comando para actualizar todos los proyectos en una compilación de varios proyectos
  - scm:validate - valida la información de scm en el pom

Maven SCM Plugin

- Opcionalmente, el Maven SCM Plugin permite configurar los diferentes goals:

```
<build>
  ...
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-scm-plugin</artifactId>
      <version>2.1.0</version>
      <configuration>
        <goals>install</goals>
      </configuration>
    </plugin>
  </plugins>
  ...
</build>
```


INTRODUCCIÓN A GIT EN EL LADO SERVIDOR

Protocolos

- Git puede usar cuatro protocolos principales para transferir datos: Local, HTTP, Secure Shell (SSH) y Git.
- El más básico es el Protocolo Local, donde el repositorio remoto es simplemente otra carpeta en el disco. Se utiliza habitualmente cuando todos los miembros del equipo tienen acceso a un mismo sistema de archivos, como por ejemplo un punto de montaje NFS, o en el caso menos frecuente de que todos se conectan al mismo computador.
- El protocolo HTTP "Inteligente" funciona de forma muy similar a los protocolos SSH y Git, pero se ejecuta sobre puertos estándar HTTP/S y puede utilizar los diferentes mecanismos de autenticación HTTP. Esto significa que puede resultar más fácil para los usuarios, puesto que se pueden identificar mediante usuario y contraseña (usando la autenticación básica de HTTP) en lugar de usar claves SSH.
- SSH es un protocolo muy habitual para alojar repositorios Git en hostings privados. Esto es así porque el acceso SSH viene habilitado de forma predeterminada en la mayoría de los servidores, y si no es así, es fácil habilitarlo. Además, SSH es un protocolo de red autenticado sencillo de utilizar.
- El protocolo Git es un "demonio" (daemon) especial, que viene incorporado con Git. Escucha por un puerto dedicado (9418) y nos da un servicio similar al del protocolo SSH; pero sin ningún tipo de autentificación. El protocolo Git es el más rápido de todos los disponibles, pero por su falta de autentificación esta destinado a crear servidores privados.

Servidor GIT

- Crear un nuevo repositorio vacío:
  $ git clone --bare my_project my_project.git
- Colocar el repositorio vacío en un servidor
  $ scp -r my_project.git user@git.example.com:/opt/git
- Configurar un "demonio" sirviendo repositorios mediante el protocolo "Git".
  $ git daemon --reuseaddr --base-path=/opt/git/ /opt/git/
- Para arrancar el demonio sin necesidad de reiniciar la máquina:
  $ initctl start local-git-daemon
- Configurar los protocolos SSH y HTTP Inteligente.
- Git trae un script CGI, denominado GitWeb, que es el que usaremos como visualizador web. Esto arranca un servidor HTTPD (servidor de red "heavy duty") en el puerto 1234, y luego arranca un navegador que abre esa página.
  $ git instaweb --httpd=webrick

GitLab

- El demonio Git con el GUI GitWeb es demasiado simple. Si buscas un servidor Git más moderno, con todas las funciones, tienes algunas soluciones de código abierto que puedes utilizar en su lugar. GitLab es una de las más populares y, aunque es algo más complejo que GitWeb y requiere algo más de mantenimiento, es una opción con muchas más funciones.
- GitLab es una aplicación web con base de datos, por lo que su instalación es algo más complicada. Por suerte, es un proceso muy bien documentado y soportado.

```
docker run --detach --name gitlab --hostname gitlab.example.com \
  --env GITLAB_OMNIBUS_CONFIG="external_url 'http://gitlab.example.com'" \
  --publish 443:443 --publish 8080:80 --publish 22:22 \
  --volume ./config:/etc/gitlab --volume ./logs:/var/log/gitlab \
  --volume ./data:/var/opt/gitlab --shm-size 256m gitlab/gitlab-ce
```

- Visita la URL de GitLab (http://localhost:8080) e inicia sesión con el nombre de usuario root y la contraseña obtenida con el siguiente comando (que se elimina automáticamente en el primer reinicio del contenedor después de 24 horas.):

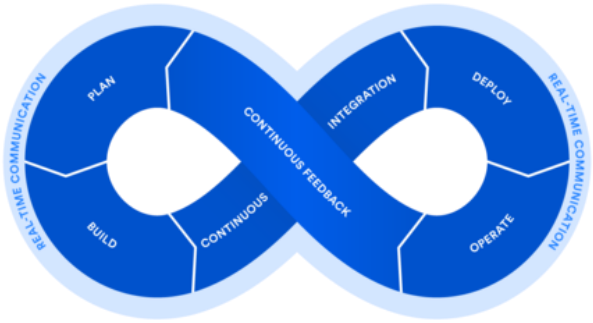
```
docker exec -it gitlab sh -c "grep 'Password:' /etc/gitlab/initial_root_password"
```


DevOps

Introducción

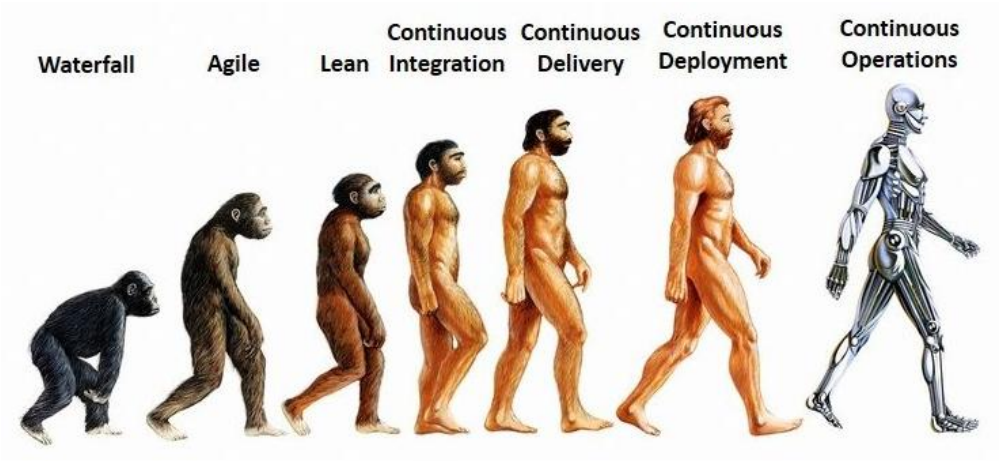
- El término DevOps como tal se popularizó en 2009, a partir de los “DevOps Days” celebrados primero en Gante (Bélgica) y está fuertemente ligado desde su origen a las metodologías ágiles de desarrollo software.
- DevOps está destinado a denotar una estrecha colaboración entre lo que antes eran funciones puramente de desarrollo, funciones puramente operativas y funciones puramente de control de calidad.
- Debido a que el software debe lanzarse a un ritmo cada vez mayor, el antiguo ciclo de desarrollo-prueba-lanzamiento de "cascada" se considera roto. Los desarrolladores también deben asumir la responsabilidad de la calidad de los entornos de prueba y lanzamiento.

Ciclo de vida



- Planificar: identificar qué funcionalidad se quiere resolver
- Construir: el desarrollo puro, escribir código, pruebas unitarias y documentar lo que se vea necesario
- Integración Continua: automatizar desde el código hasta el entorno de producción
- Desplegar: automatizar el paso a producción
- Operar: vigilar el correcto funcionamiento del entorno, monitorizar
- Feed back: verificar que la funcionalidad tiene valor para el usuario o el retorno esperado

Movimiento DevOps

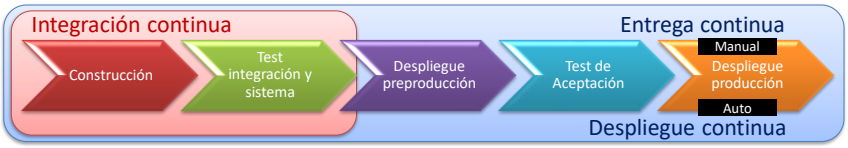


Ciclo “continuo”

- Muchas empresas se encuentran en algún lugar entre la cascada y las metodologías ágiles. DevOps realmente comienza donde está el “Lean” en la imagen. A medida que se eliminan los cuellos de botella y se comienza a brindar coherencia, podemos comenzar a avanzar hacia la integración continua, la entrega continua y, tal vez, hasta la implementación y las operaciones continuas.
- Con la integración continua, los desarrolladores necesitan escribir pruebas unitarias y se crea un proceso de construcción automatizado. Cada vez que un desarrollador verifica el código, se ejecutan automáticamente las pruebas unitarias y, si alguna de ellas falla, la construcción completa falla. Esta es una mejora con respecto al modelo anterior porque se introducen menos errores en el control de calidad y la compilación solo contiene código de trabajo que reduce la acumulación de defectos.
- Al automatizar las pruebas unitarias en los procesos de construcción, eliminamos muchos errores humanos, mejorando así la velocidad y confiabilidad.

Ciclo “continuo”

- La entrega continua CD (continuous delivery) hace referencia a entregar las actualizaciones a los usuarios según estén disponibles sobre una base sólida y constante.
- El despliegue continuo CD (continuous deployment) es la automatización del despliegue de la entrega continua, que no exista intervención humana a la hora de realizar el despliegue en producción.
- Llegar a CI y CD son el objetivo que muchas empresas se esfuerzan por cumplir y están aprovechando el DevOps para ayudarles a lograrlo. Sin embargo, algunas empresas deben ir más allá porque pueden realizar entregas muchas veces al día o tener una presencia web muy popular.
- En la implementación continua, con solo presionar un botón pasa por la construcción, las pruebas automatizadas, la automatización de los entornos y la producción. Esto se vuelve importante con respecto a las operaciones continuas.



Prácticas de DevOps

- Más allá del establecimiento de una cultura de DevOps, los equipos ponen en práctica el método DevOps implementando determinadas prácticas a lo largo del ciclo de vida de las aplicaciones.
- Algunas de estas prácticas ayudan a agilizar, automatizar y mejorar una fase específica.
- Otras abarcan varias fases y ayudan a los equipos a crear procesos homogéneos que favorezcan la productividad.
- Estas practicas incluyen:
  - Control de versiones
  - Desarrollo ágil de software
  - Automatización de pruebas
  - Integración y entrega continuas (CI/CD)
  - Infraestructura como código
  - Administración de configuración
  - Supervisión continua

Control de versiones

- Control de versiones es la práctica de administrar el código y la documentación por versiones, haciendo un seguimiento de las revisiones y del historial de cambios para facilitar la revisión y la recuperación del código.
- Esta práctica suele implementarse con sistemas de control de versiones, como Git, que permite que varios desarrolladores colaboren para crear código. Estos sistemas proporcionan un proceso claro para fusionar mediante combinación los cambios en el código que tienen lugar en los mismos archivos, controlar los conflictos y revertir los cambios a estados anteriores.
- El uso del control de versiones es una práctica de DevOps fundamental que ayuda a los equipos de desarrollo a trabajar juntos, dividir las tareas de programación entre los miembros del equipo y almacenar todo el código para poder recuperarlo fácilmente si fuese necesario.
- El control de versiones es también un elemento necesario en otras prácticas, como la integración continua y la infraestructura como código.

Herramientas de Automatización