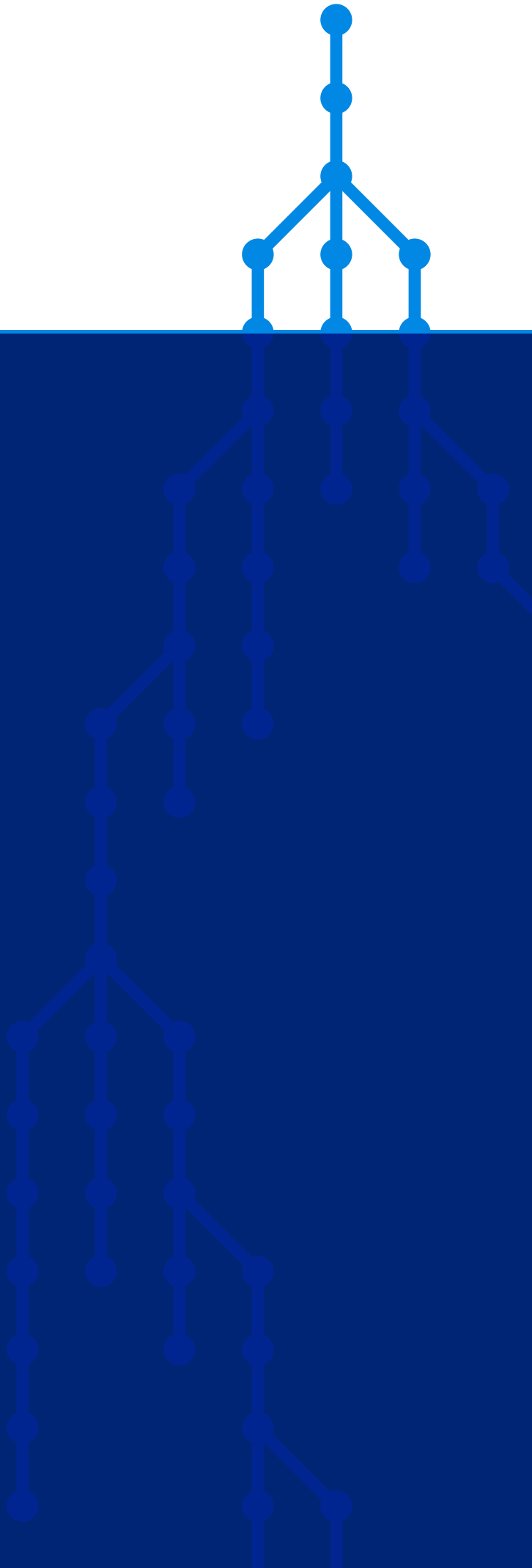
Formación DevOps

Git



GRAVITY

Contenidos

1. Introducción

¿Qué es Git?

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

¿Por qué usar un sistema de control de versiones como Git?

Un sistema de control de versiones como Git nos ayuda a guardar el historial de cambios y crecimiento de los archivos de nuestro proyecto.

En realidad, los cambios y diferencias entre las versiones de nuestros proyecto pueden tener similitudes, algunas veces los cambios pueden ser solo una palabra o una parte específica de un archivo específico. Git está optimizado para guardar todos estos cambios de forma atómica e incremental, o sea, aplicando cambios sobre los últimos cambios, estos sobre los cambios anteriores y así hasta el inicio de nuestro proyecto.

El comando para inicializar nuestro repositorio, o sea, indicarle a Git que queremos usar su sistema de control de versiones en nuestro proyecto, es `git init`.

El comando para que nuestro repositorio sepa de la existencia de un archivo o sus últimos cambios es `git add`. Este comando no almacena los cambios de forma definitiva, solo los guarda en algo que conocemos como “Staging Area”.

El comando para almacenar definitivamente todos los cambios, que se encuentren en el staging area es `git commit`, además añadiremos un mensaje para recordar muy bien qué cambios hicimos en este commit con el argumento `-m "Mensaje del commit"`.

El comando para mandar nuestros commits a un servidor remoto, donde todos podamos conectar nuestros proyectos, es `git push`.

2. Comandos básicos en Git

Ciclo básico de trabajo en Git

Para iniciar un repositorio Git en tu proyecto, solo debes ejecutar el comando `git init` en las carpeta del proyecto.

Este comando se encargará de dos cosas: primero, crear una carpeta .git, donde se guardará toda la base de datos con cambios atómicos de nuestro proyecto; y segundo, crear un área que conocemos como Staging, que guardará temporalmente los cambios que hagamos a nuestros archivos y nos permitirá, más adelante, guardar estos cambios en el repositorio.

Ciclo de vida o estados de los archivos en Git.

Cuando trabajamos con Git, nuestros archivos pueden vivir y moverse entre 4 diferentes estados (cuando trabajamos con repositorios remotos pueden ser más estados):

- Archivos Tracked: son los archivos que están indexados en la base de datos de Git, no tienen cambios pendientes y sus últimas actualizaciones han sido guardadas en el repositorio gracias a los comandos `git add` y `git commit`.
- Archivos Staged: son archivos cuyos últimos cambios están en el area de Staging. Están indexados por Git y hay registro de ellos porque han sido afectados por el comando `git add`. Git está al tanto de la existencia de sus últimos cambios, pero todavía no han sido guardados definitivamente en el repositorio porque falta ejecutar el comando `git commit`.
- Archivos Unstaged: entiéndelos como archivos "Tracked pero no añadidos al area de Staging". Son archivos indexados por Git, pero con cambios que no han sido afectados por el comando git add, Git tiene registro de estos archivos, pero está desactualizado de sus últimos cambios, ya que solo existen en el working copy (carpeta de proyecto del disco duro).
- Archivos Untracked: son archivos que NO han sido indexados por Git, por lo que solo existen en el working copy. No han sido afectados por `git add`, así que Git no tiene registros de su existencia.

Recuerda: hay un caso especia, donde los archivos tienen dos estados al mismo tiempo: staged y untracked. Esto pasa cuando guardas los cambios de un archivo en el área de Staging (con el comando `git add`), pero antes de hacer commit para guardar los cambios en el repositorio, haces nuevos cambios que todavía no han sido guardados en el área de Staging (en realidad, todo sigue funcionando igual pero es un poco divertido).

Comandos para mover archivos entre los estados de Git

- `git status`: nos permite ver el estado de todos nuestros archivos y carpetas.
- `git add`: nos ayuda a mover archivos del estado Untracked o Unstaged al estado Staged. Podemos usar `git nombre-del-archivo-o-carpeta` para añadir archivos y carpetas individuales o `git add -A` para mover todos los archivos de nuestro proyecto (tanto Untrackeds como Unstageds).
- `git reset HEAD`: nos permite mover archivos del estado Staged a su estado anterior. Si los archivos venían de Unstaged, vuelven allí. Y lo mismo si venían de Untracked.
- `git commit`: nos ayuda a mover archivos de Unstaged a Staged de manera definitiva. Esta es una ocasión especial, los archivos quedarán guardados o actualizados en el repositorio. Git nos pedirá que indiquemos un mensaje para recordar los cambios que hicimos, usando el argumento `-m` para escribirlo `git commit -m "mensaje"`.
- `git rm`: este comando necesita alguno de los siguientes argumentos para poder ejecutarse correctamente:
  - `git rm --cached`: Mueve los archivos que le indiquemos al estado Untracked.
  - `git rm --force`: Elimina los archivos de Git y del disco duro. Git guarda el registro de la existencia de los archivos, por lo que podremos recuperarlos si fuera necesario usando comandos más avanzados.

¿Qué son las ramas en Git?

En esencia, Git es una base de datos muy precisa que almacena todos los cambios y crecimiento que se dan en nuestro proyecto. Los commits son la única forma de tener un registro de los cambios. Esta base de datos además añade un sistema de etiquetado en árbol, lo que permite recorrer los registros de un modo muchísimo más potente, usando ramas, las cuales permiten amplifican el potencial de Git al maximo.

Como concepto, todo commit se aplica sobre una rama. Por defecto, siempre empezamos en la rama master y a partir de ella podremos crear nuevas ramas, para crear flujos de trabajo independientes.
En esencia, crear nueva rama en Git, se trata de apuntar un commit (de cualquier rama), desde la nueva rama y continuar el trabajo de una parte específica de nuestro proyecto sin afectar el flujo de trabajo principal (el cual continúa en la rama principal). Para situarnos en la nueva nueva rama debemos hacer `git checkout`.

Podemos crear todas las ramas y commits que queramos. De hecho, podemos y debemos aprovechar el registro de cambios de Git para crear ramas, traer versiones viejas del código, arreglarlas y combinarlas de nuevo para mejorar el proyecto.

Para combinar el contenido de estas ramas se usa el comando `git merge`, el cual mezcla el contenido de una rama en otra, ten en cuenta que este proceso puede generar conflictos ya que algunos archivos pueden ser diferentes en ambas ramas. Pero no te preocupes, Git es muy inteligente e intentará unir estos cambios automáticamente, pero no siempre lo consigue, y en esos casos, debemos ser nosotros los que resolvamos estos conflictos “a mano”.

Crea un repositorio de Git y haz tu primer commit

Para indicar a Git que queremos crear un nuevo repositorio para utilizar su sistema de control de versiones. Solo debemos posicionarnos en la carpeta raíz de nuestro proyecto y ejecutar el comando `git init`.

> **Recuerda:** Al ejecutar este comando (y de aquí en adelante) vamos a tener una nueva carpeta oculta llamada .git con toda la base de datos con cambios atómicos de nuestro proyecto.

Git está optimizado para trabajar en equipo, por lo tanto, debemos darle un poco de información sobre nosotros. No debemos hacerlo todas las veces que ejecutamos un comando, basta con ejecutar solo una sola vez los siguientes comandos con tu información:

```
git config --global user.email "tu@email.com"
git config --global user.name "Tu Nombre"
```

Existen muchas otras configuraciones de Git que puedes encontrar ejecutando el comando `git config --list` (o solo `git config` para ver una explicación más detallada).

Analizar cambios en los archivos de tu proyecto con Git

El comando `git show` nos muestra los cambios que han existido sobre un archivo y es muy útil para detectar cuándo se produjeron ciertos cambios, qué se rompió y cómo lo podemos solucionar.

Si queremos ver la diferencia entre una versión y otra, no necesariamente todos los cambios desde la creación del archivo, podemos usar el comando `git diff commitA commitB`.
Puedes obtener el ID de tus commits con el comando `git log`.

Volver en el tiempo en nuestras ramas

El comando `git checkout <commit-id>` nos permite viajar en el tiempo. Podemos volver a cualquier versión anterior de un archivo específico o incluso del proyecto entero. Esta también es la forma de crear ramas y movernos entre ellas utilizando el modificador `-b`, por lo tanto si estamos en una rama, pero queremos crear una nueva y en el mismo acto movernos a ella usaremos el comando `git checkout -b <new-branch-name>`.

También hay una forma de movernos en el tiempo un poco más tajante: usando el comando `git reset`. En este caso, no solo "volvemos en el tiempo", sino que borramos todos los cambios que existieran después de este commit para esa rama, restaurando completamente la rama al punto que hayamos indicado en el comando.

Hay dos formas de usar `git reset`: con el argumento `--hard`, borrando toda la información que tengamos en el área de staging (y perdiendo todo lo correspondiente a esa rama). O bien usando el argumento `--soft`, que mantiene todos los cambios aplicados en los archivos del área de staging, para que podamos aplicar continuar nuestro trabajo desde ese punto en adelante.

Git reset vs. Git rm

`git reset` y `git rm` son comandos con utilidades muy diferentes, pero aún así se confunden muy fácilmente.

`git rm`

Este comando nos ayuda a eliminar archivos de Git sin eliminar su historial del sistema de versiones. Esto quiere decir que si necesitamos recuperar el archivo solo debemos "viajar en el tiempo" y recuperar el último commit antes de borrar el archivo en cuestión.

`git rm --cached`: Elimina los archivos del área de Staging y del próximo commit pero los mantiene en nuestro disco duro.

`git rm --force`: Elimina los archivos de Git y del disco duro. Git siempre guarda todo, por lo que podemos acceder al registro de la existencia de los archivos, de modo que podremos recuperarlos si es necesario (pero debemos usar comandos más avanzados).

`git reset`

Este comando nos ayuda a volver en el tiempo. Pero no como `git checkout` que nos deja ir, mirar, pasear y volver. Con `git reset` volvemos al pasado sin la posibilidad de volver al futuro. Borramos la historia y la debemos volver a escribir. No hay vuelta atrás. (Por supuesto, como en todo Git, existen modos de trabajo avanzado que aprovechan este comportamiento, sin que esto suponga una perdida de historia definitiva)

> Recuerda: `git reset` es un comando es muy delicado y debemos usarlo con conocimiento de qué estamos haciendo.

Recuerda que `git reset` se debe usar indicando siempre una de estas dos opciones:

- git reset --soft: Borramos todo el historial y los registros de Git pero guardamos los cambios que tengamos en Staging, así podemos aplicar las últimas actualizaciones a un nuevo commit.
- git reset --hard: Borra todo. Absolutamente todo. Toda la información de los commits y del área de staging se borra del historial.

¡Pero nos falta algo!

`git reset HEAD`: Este es el comando para sacar archivos del área de Staging. No para borrarlos ni nada de eso, solo para que los últimos cambios de estos archivos no se envíen al último commit, a menos que cambiemos de opinión y los incluyamos de nuevo en staging con `git add`, por supuesto.

¿Para qué nos vale esto?

Imagina el siguiente caso

Hacemos cambios en los archivos de un proyecto para una nueva actualización. Todos los archivos con cambios se mueven al área de staging con el comando `git add`. Pero te das cuenta de que uno de esos archivos no está listo todavía. Actualizaste el archivo pero ese cambio no debe ir en el próximo commit.

¿Qué podemos hacer?

Bueno, todos los cambios están en el área de Staging, incluido el archivo con los cambios que no están listos. Esto significa que debemos sacar ese archivo de Staging para poder hacer commit de todos los demás.

Si usamos `git rm`, lo que haremos será ¡eliminar este archivo completamente del indice de Git! Aunque todavía, nos quedará en el historial de cambios de este archivo, la eliminación del archivo como su última actualización. Pero recuerda, en este caso no buscábamos eliminar el archivo, solo dejarlo como estaba y actualizarlo después, pero no en este commit.

En cambio, si usamos `git reset HEAD`, lo único que haremos será mover estos cambios del area de Staging al Working Copy y al estado Unstaged. Seguiremos teniendo los últimos cambios del archivo, el repositorio mantendrá el archivo (no con sus últimos cambios pero sí con los últimos en los que hicimos commit) y no habremos perdido nada.

Finalmente lo mas común para deshacer cambios del area de Staging es usar `git restore --stage <file>`.

3. Flujo de trabajo en Git

Introducción a las ramas de Git

Las ramas son la forma de hacer cambios en nuestro proyecto sin afectar el flujo de trabajo de la rama principal. Esto porque queremos trabajar una parte muy específica de la aplicación o simplemente experimentar.

> Recuerda: La cabecera o HEAD representa la rama y el commit de esa rama donde estamos trabajando.

Por defecto, esta cabecera aparecerá en el último commit de nuestra rama principal. Pero podemos cambiarlo al crear una rama (`git branch <rama>`, `git checkout -b <rama>`) o movernos en el tiempo a cualquier otro commit de cualquier otra rama con los comandos (`git reset <commit-id>`, `git checkout <rama-o-commit-id>`).

Fusión de ramas con Git merge

El comando `git merge` nos permite crear un nuevo commit con la combinación de dos ramas (la rama donde nos encontramos cuando ejecutamos el comando y la rama que indiquemos después del comando).

```
# Crear un nuevo commit en la rama master combinando
# los cambios de la rama cabecera:
git checkout master
git merge --no-ff cabecera
# Crear un nuevo commit en la rama cabecera combinando
# los cambios de cualquier otra rama:
git checkout cabecera
git merge --no-ff cualquier-otra-rama
```

Asombroso ¿verdad?, es como si Git tuviera super poderes para saber qué cambios queremos conservar de una rama y cuales de la otra. El problema es que no siempre puede adivinar, sobretodo en algunos casos donde dos ramas tienen actualizaciones diferentes en ciertas líneas que colisionan en los archivos. Esto lo conocemos como un conflicto .

> Recuerda: Al ejecutar el comando `git checkout` para cambiar de rama o commit puedes perder el trabajo que no hayas guardado. Guarda tus cambios antes de hacer `git checkout`.

Solución de conflictos al hacer merge

Git nunca borra nada a menos que nosotros se lo indiquemos.

Cuando usamos los comandos `git merge` o `git checkout` estamos cambiando de rama o creando un nuevo commit, no borrando ramas ni commits.

> Recuerda: Puedes borrar commits con `git reset` y ramas con `git branch -d`.

Git es muy inteligente y puede resolver algunos conflictos automáticamente: cambios, nuevas líneas, entre otros. Pero algunas veces no sabe cómo resolver estas diferencias, por ejemplo, cuando dos ramas diferentes hacen cambios distintos a una misma línea.

Esto lo conocemos como conflicto y lo podemos resolver manualmente, solo debemos hacer el merge, ir a nuestro editor de código y elegir si queremos quedarnos con alguna de estas dos versiones o algo diferente. Algunos editores de código como VSCode, nos ayudan a resolver estos conflictos sin necesidad de borrar o escribir líneas de texto.

> Recuerda: Tras un conflicto, siempre debemos crear un nuevo commit para aplicar los cambios del merge.

Si Git puede resolver el conflicto por si solo, hará commit automáticamente. Pero, en caso de no pueda resolverlo, debemos solucionarlo y hacer el commit nosotros.

Los archivos con conflictos por el comando `git merge` entran en un nuevo estado que conocemos como **Unmerged**. Funcionan muy parecido a los archivos en estado Unstaged, algo así como un estado intermedio entre Untracked y Unstaged, solo debemos ejecutar `git add` para pasarlos al área de staging y `git commit` para aplicar los cambios en el repositorio.

Flujo de trabajo en repositorios remotos

Por ahora, nuestro proyecto vive únicamente en nuestro ordenador. Esto significa que no hay forma de que otros miembros del equipo colaboren en él. Para solucionar esto están los servidores remotos.

Estos servidores remotos lo que van a hacer es guardar el mismo repositorio que tienes en tu ordenador y darnos una URL con la que todos podremos acceder a los archivos del proyecto para descargarlos, hacer cambios y volverlos a enviar al servidor remoto para que otras personas vean los cambios, comparen sus versiones y creen nuevas propuestas para el proyecto.

Ahora debemos aprender algunos nuevos comandos:
`git clone <remote-server-url>`: Nos permite descargar el repositorio completo a nuestra maquina, y por tanto toda las ramas, e historia de cambios de los archivos nuestro proyecto, incluido por supuesto la última versión de la rama principal.

`git push`: Luego de hacer `git add` y `git commit` tendremos que ejecutar este comando para mandar los cambios de la rama al servidor remoto.

`git fetch --prune`: Lo usamos para sincronizar nuestro repositorio local, contra el repositorio remoto, y por tanto traer todas las actualizaciones y cambios almacenadas en el servidor remoto y guardarlas en nuestra copia local (en el caso de que hayan, por supuesto).

`git pull`: Básicamente, `git fetch` y `git merge` al mismo tiempo, o en caso de usar el modificador `–rebase` sería equivalente a `git fetch + git rebase`.

Recuerda:

- `git merge`: Lo necesitamos para combinar los últimos cambios del servidor remoto y nuestro directorio de trabajo.
- `git rebase`: Reubica el trabajo de la rama actual, a partir de un nuevo punto de historia (nueva base), lo cual nos permite resolver los posibles conflictos de un modo mucho más limpio y simple para finalmente combinar los cambios usando `git merge --no-ff`, lo cual fuerza la generación de un commit de mezcla en la rama donde se fusiona la nueva rama generada tras el rebase.

4. Trabajando con repositorios remotos en GitLab

Uso de GitLab

GitLab es una plataforma que nos permite guardar y gestionar repositorios de Git, la cual podemos usar como servidor remoto y en la que podemos ejecutar algunos comandos de forma visual e interactiva (sin necesidad de la consola de comandos).

Podemos crear o importar repositorios, crear grupos, subgrupos y proyectos de trabajo, descubrir repositorios de otras personas, contribuir a proyectos de terceros, etc.

El archivo README.md lo veremos por defecto al entrar a un repositorio. Es una muy buena práctica configurarlo para describir el proyecto, los requerimientos y las instrucciones que debemos seguir para contribuir correctamente.

Para clonar un repositorio desde GitLab (o cualquier otro servidor remoto) debemos copiar la URL (por ahora, usando HTTPS) y ejecutar el comando `git clone <url>`. Esto descargara una copia completa del repositorio del proyecto que se encuentra en GitLab.

Sin embargo, esto solo funciona para las personas que quieren empezar a contribuir en el proyecto. Si queremos conectar el repositorio de GitLab con nuestro repositorio local, el que creamos con `git init`, debemos ejecutar las siguientes instrucciones:

```
# Primero: Guardar la URL del remoto con el nombre “origin”
git remote add origin <url>
# Segundo: Verificar que se haya asignado correctamente
git remote -v
# Tercero: Usaremos git fetch y git merge para actualizar,
# o solo git pull con el flag --allow-unrelated-histories
git pull origin master --allow-unrelated-histories
# Por último, haremos git push para guardar en GitLab:
git push origin master
```

Tags y versiones en Git

Los tags o etiquetas nos permiten asignar versiones a los commits con cambios más importantes o significativos de nuestro proyecto.

Comandos para trabajar con etiquetas:

- Crear un nuevo tag y asignarlo a un commit:
  `git tag -a <tag-name> <commit-id>`
- Borrar un tag en el repositorio local:
  `git tag -d <tag-name>`
- Listar los tags de nuestro repositorio local:
  `git tag` o `git show-ref --tags`
- Publicar un tag en el repositorio remoto:
  `git push origin --tags.`
- Borrar un tag del repositorio remoto:
  `git tag -d <tag-name>` o `git push origin :refs/tags/<tag-name>`

Manejo de ramas en GitLab

Puedes trabajar con ramas que nunca envías a GitLab, así como pueden haber ramas importantes en GitLab que nunca usas en el repositorio local. Lo importantes es que aprendas a manejarlas para trabajar correctamente.

- Crear una rama en el repositorio local:
  `git branch <branch-name>` o `git checkout -b <branch-name>`
- Publicar una rama local al repositorio remoto:
  `git push origin <branch-name>`

Recuerda que podemos ver gráficamente nuestro entorno y flujo de trabajo local con Git usando Sourcetree / GitKraken o cualquier otra GUI.

5. Flujos de trabajo con remotos

Trabajando con merge requests

En un entorno profesional, normalmente se bloquea las ramas master y develop, y para enviar código a dichas rama, se pasa por un code review y luego de su aprobación se combina el código de las mismas a través de los llamados merge request, incorporándose a la rama develop que desemboca en el entorno de desarrollo.

Para realizar pruebas de usuario, enviamos el contenido de la rama release a servidores que normalmente se les llama staging develop (QA), luego de que se realizan las pruebas pertinentes tanto de código como de la aplicación, aplica un tag y se cierra la rama de release contra master y develop, entregándose el resultado binario y probado en el servidor de producción, y dejando las ramas develop y master actualizadas a la última entrega en productivo realizada.

> Recuerda: Mientras se está probando una rama release no se detiene el avance de funcionalidades contra develop

Ignorando archivos con .gitignore

No todos los archivos que agregas a un proyecto deben ir al repositorio remoto, pej.: un archivo con passwords; son archivos que nadie debe ver, ni deben pertenecer a la base de código del repositorio. Para estos casos, indicaremos los ficheros, carpetas, o extensiones que queremos que Git ignore, y de este modo no nos aparezcan ni siquiera como Untracked

Readme.md, una excelente práctica

Incluir un archivo README.md en la raiz de tu repositorio, es una excelente práctica ya que nos permite informar y presentar nuestro proyecto a los colaboradores, con un aspecto formateado de una manera facil, gracias al uso de Markdown.

6. Multiples entornos de trabajo

Git Rebase: Reorganizando el trabajo

Con rebase puedes recoger todos los cambios confirmados en una rama y ponerlos sobre otra.

```
# Cambiamos a la rama que queremos traer los cambios
git checkout experiment
# Aplicamos rebase para traer los cambios de la rama que queremos
git rebase develop
```

Cuando queremos mezclar nuestro trabajo en develop, debemos asegurarnos que nuestra rama, parte del ultimo estado de la rama develop para ello, haremos rebase de nuestra rama sobre develop y de este modo, solo cuando se cumpla esta condición, podremos realizar la mezcla de nuestra rama con develop.

Git Stash: Guardar cambios en memoria y recuperarlos después

Cuando necesitamos regresar en el tiempo porque borramos alguna línea de código pero no queremos pasarnos a otra rama porque nos daría un error ya que debemos pasar ese "mal cambio" que hicimos al Stage, podemos usar `git stash` para regresar el cambio anterior que hicimos.

`git stash` es típico cuando estamos haciendo cambios que no merecen una rama o no merecen un rebase si no simplemente estamos probando algo y luego quieres volver rápidamente a tu versión anterior la cual es la correcta.

Git Clean: Limpiar tu proyecto de archivos no deseados

A veces creamos archivos cuando estamos realizando nuestro proyecto que realmente no forman parte de nuestro directorio de trabajo, que no se debería agregar y lo sabemos.

- Para saber qué archivos vamos a borrar:
  `git clean --dry-run`
- Para borrar todos los archivos listados (que no son carpetas):
  `git clean -f`

Git Cherry-Pick: Traer commits al head de un branch

Existe un mundo alternativo en el cual vamos avanzando en una rama pero necesitamos en nuestra rama de trabajo, uno de esos avances de la rama de otro compañero, para eso utilizamos el comando `git cherry-pick <commit-id>`.

Cherry Pick, permite disponer cambios ajenos en nuestra rama, reconstruyendo la historia.

> Recuerda: Debes usar `cherry-pick` con cuidado y sabiendo que es lo que se hace en todo momento.

7. Comandos en Git para casos de emergencia

Reconstruir commits en Git con amend

A veces hacemos un commit, pero resulta que no queríamos mandarlo porque faltaba algo más. Utilizamos `git commit --amend`, amend en inglés es enmendar y lo que hará es que los cambios que hicimos nos lo agregará al commit anterior.

Git Reset: Usar en caso de emergencia

¿Qué pasa cuando todo se rompe y no sabemos qué está pasando? Pues con `git reset <HEAD-hash>` volveremos al anterior estado en que el proyecto funcionaba.

Recuerda:

- `git reset --soft <HEAD-hash>`: Nos mantiene lo que tengamos en staging disponible.
- `git reset --hard <HEAD-hash>`: Nos restaura absolutamente todo el estado, incluyendo lo que tuvieramos en staging.

Buscar en archivos y commits de Git con Grep y log

A medida que nuestro proyecto se hace grande vamos a querer buscar ciertas cosas. Por ejemplo: ¿cuántas veces en nuestro proyecto utilizamos la palabra 'color'?

Para buscar utilizaremos el comando `git grep color` y nos buscará en todo el proyecto los archivos donde aparezca la palabra 'color'.

- Con `git grep -n color` nos saldrá un output el cual nos dirá en qué línea está lo que estamos buscando.
- Con `git grep -c color` nos saldrá un output el cual nos dirá cuántas veces se repite esa palabra y en qué archivo.
- Si queremos buscar cuántas veces utilizamos un atributo de HTML lo hacemos con `git grep -c "<p>"`.

8. Bonus

Comandos y recursos en Git

Algunos comandos interesantes

- `git shortlog`: cuántos commits ha hecho cada miembro del equipo.
- `git shortlog -sn`: muestra las personas que han hecho ciertos commits, junto con la cantidad.
- `git shortlog -sn --all`: incluye todos los commits, incluso los borrados.
- `git shortlog -sn --all --no-merges`: todos los commits sin incluir los merges.
- `git config --global alias.nombre “sentencia”`: añade un nuevo alias global.
  - `git config --global alias.stats “shortlog -sn --all --no-merges”`.
- `git <comando> --help`: para conocer cómo funciona un comando.
- `git branch -r`: muestra las ramas del repositorio remoto.
- `git branch -a`: muestra todas las ramas locales y remotas.

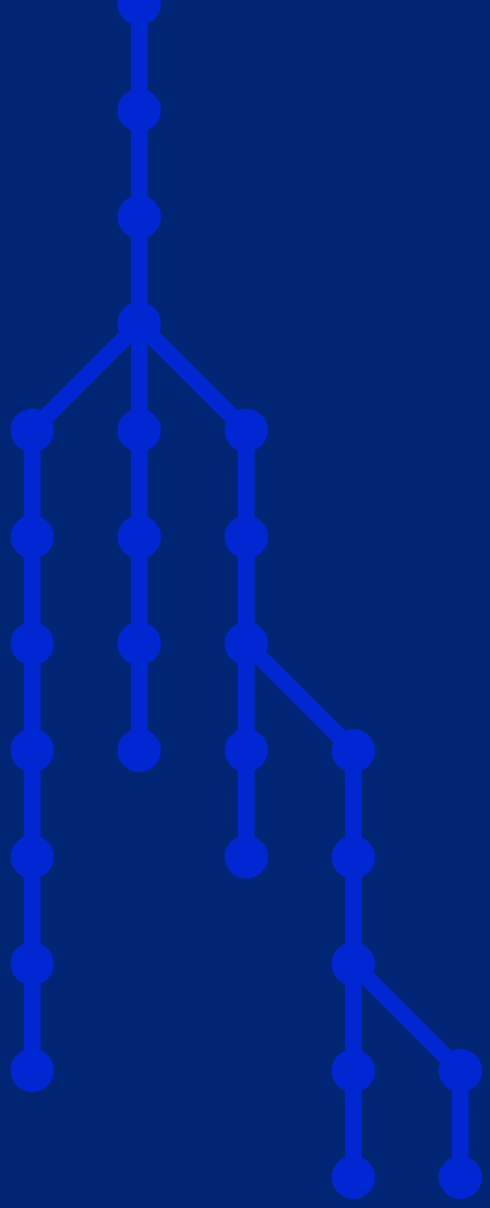
9. Referencias

• Git - Book
https://git-scm.com/book/es/v2

• Learning Git Branching
https://learngitbranching.js.org/

• Conventional Commits
https://www.conventionalcommits.org/en/v1.0.0-beta.2/

Elaborado por José Barragán
jose.barragan@gravity.es



Git

Formación DevOps

GRAVITY

www.gravity.es