codenotch

Fundamentos de Programación

Tema 6 | Git

codenotch

Índice

codenotch

INTRODUCCIÓN

¿Qué es GIT?

Posiblemente alguno de vosotros haya oído hablar de **GIT** pero, ¿qué es GIT realmente?

GIT es un **sistema de control de versiones** de archivos, documentos, proyecto etc. Es decir, gestiona todos los cambios que se realizan sobre los elementos de algún producto o configuración.

Existen otros software de control de versiones pero GIT es el más extendido y el que mejor resultados ha dado.

codenotch

DEFINICIÓN

Uso de GIT

GIT está pensado para la eficiencia, la confiabilidad del control de versiones de cualquier proyecto.

Lo normal es que cuando se trabaja en un proyecto, lo hagan varias personas o incluso varios equipos de una misma empresa, usando los mismos archivos y las mismas fuentes para trabajar todos de una manera más eficiente.

Por lo tanto se hace necesario algún tipo de software que controle todos los cambios que se hacen sobre los archivos, se puedan aceptar para adaptarlos al proyecto final o descartarlos en caso de no ser correctos.

La manera más común de trabajar es mediante el uso de ramas. Esto no es más que hacer una copia del proyecto sobre la que cada desarrollador puede trabajar de manera independiente en su equipo.

codenotch

DEFINICIÓN

Uso de GIT

Cuando se pretenden unir los cambios al proyecto, el desarrollador pedirá unir los cambios de su rama para que estos se incorporen y ya estén disponibles para todos.

A continuación vamos a ver y comentar algunos de los conceptos básicos de lo que GIT aporta a cualquier proyecto.

codenotch

DEFINICIÓN

Conceptos básicos de Git

Los conceptos y funciones de GIT son muchos y muy potentes.

Aquí vamos a ver y comentar algunos de los principales:

- Fuerte apoyo al desarrollo no lineal.
- Gestión distribuida.
- Eficiencia en proyectos grandes.
- Acceso a todos los cambios realizados a lo largo de la historia.
- Velocidad de acceso.

codenotch

INTRODUCCIÓN

Instalación de GIT (“Git Bash”)

Para instalar GIT debemos seguir los siguientes pasos en nuestro equipo:

1. Ir a la página de descargas de GITHUB: https://gitforwindows.org/

1. Este archivo que se descarga es un .exe que tenéis que ejecutar.

1. Una vez instalado tendréis la versión de línea de comandos y la interfaz gráfica de usuario estándar.

1. Abrid “Git Bash” para empezar a trabajar con GIT.

codenotch

INTRODUCCIÓN

Entorno GIT BASH

GIT BASH es el entorno de consola que vamos a utilizar para trabajar con GIT.

Es mucho más rápido que cualquier gestor gráfico y, además, es bueno empezar aprendiendo GIT mediante sus comandos de utilización en lugar de utilizar un componente gráfico para usuarios, ya que, aunque la curva de aprendizaje sea mayor, el control y el aprendizaje será total.

Tened en cuenta que GIT es una herramienta muy potente pero, a su vez, es muy peligrosa.

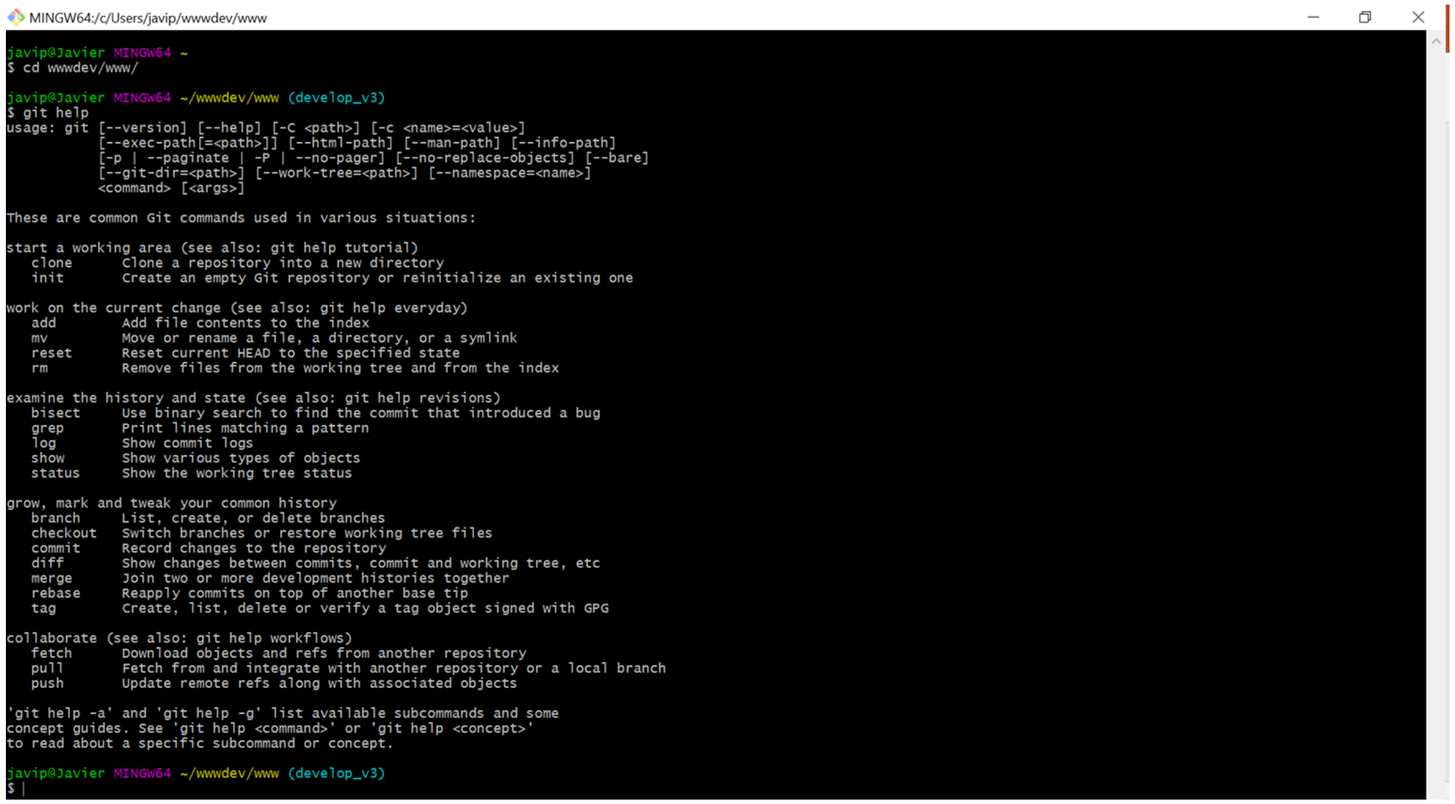
Se podría arruinar un proyecto entero si no sabemos bien qué estamos incorporando a él como cambios.

codenotch

INTRODUCCIÓN

Entorno GIT BASH

Este es el aspecto que tiene nuestra consola de GIT BASH:



codenotch

CONFIGURACIÓN

Configuración de GIT

GIT por defecto trae una herramienta llamada "git config" que te permite obtener y reestablecer variables de configuración importantes y necesarias para cada usuario que use la herramienta de control de versiones.

Lo primero que haremos es configurar nuestra identidad. Esto es totalmente indispensable para cualquier proyecto GIT en el que trabajemos, pues así, en el conjunto global del proyecto y de los desarrolladores, el sistema será capaz de identificarnos y saber quién hizo qué cambio. Esta es la manera de hacerlo:

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

codenotch

CONFIGURACIÓN

Configuración de GIT

Ahora bien, siempre que queramos podemos cambiar nuestra configuración de GIT que acabamos de inicializar. De la misma manera, seremos capaces de consultarla. Para ello debemos hacer lo siguiente:

Con este comando, como veis, sacamos toda la configuración de nuestro GIT.

```
$ git config --list
user.name=Scott Chacon
user.email=schacon@gmail.com
color.status=auto
color.branch=auto
color.interactive=auto
color.diff=auto
...
```

Para cualquier tipo de ayuda o sugerencia sobre el uso de la configuración de GIT, podéis ejecutar el siguiente comando, y os mostrará todas las posibilidades disponibles:

```
$ git help config
```

codenotch

Comandos Básicos

codenotch

DEFINICIÓN

GIT INIT

Para empezar a usar GIT en un proyecto, incluyendo el seguimiento de los archivos existentes, debemos ir al directorio del proyecto y escribir este comando (git init)

Veamos un ejemplo:

```
javip@Javier MINGW64 ~
$ cd OneDrive/Codenotch/fundamentos/

javip@Javier MINGW64 ~/OneDrive/Codenotch/fundamentos
$ git init
Initialized empty Git repository in C:/Users/javip/OneDrive/Codenotch/fundamentos/.git/

javip@Javier MINGW64 ~/OneDrive/Codenotch/fundamentos (master)
$
```

codenotch

DEFINICIÓN

GIT INIT

Si nos fijamos en la captura de pantalla podemos observar los siguiente:

1. Nos hemos posicionado sobre el directorio donde queremos empezar a usar GIT, en este caso el directorio “fundamentos”.

1. Dentro de ese directorio ejecutamos el comando **“git init”**. Esto lo que hace es crear un repositorio de GIT, para ello inicializa el nuevo directorio “.git”, donde se guardaran todos los archivos sobre los que hagamos cambios y los propios cambios que hemos realizado.

1. Si os fijáis, ahora además, en el “path”(ruta) del directorio, en color azul, nos pone “(master)”, que es la rama principal del proyecto GIT que acabamos de empezar. Todos los cambios que suban aquí y serán los que sean válidos para el proyecto, ya que como hemos dicho, esta rama “master” será nuestra rama principal del proyecto.

codenotch

DEFINICIÓN

GIT CLONE

Si, de otro modo, lo que tenemos que hacer es unirnos a un proyecto ya existente con un repositorio GIT creado y funcionando, el comando que debemos usar es **“git clone”**.

Este comando hace una copia de todo el proyecto existente en el equipo que ejecuta el comando. Es decir, descarga desde la dirección indicada un proyecto completo y funcional para que un usuario pueda empezar a trabajar con él de manera local y, más tarde, poder hacer una petición de cambios a este.

Este es un ejemplo de como usar el comando:

```
$ git clone git://github.com/schacon/grit.git
```

codenotch

DEFINICIÓN

GIT CLONE

Vamos a ver qué pasa si lo ejecutamos en nuestro entorno de GIT BASH.

```
javip@Javier MINGW64 ~/OneDrive/Codenotch/fundamentos (master)
$ git clone git://github.com/schacon/grit.git mygrit
Cloning into 'mygrit'...
remote: Enumerating objects: 4051, done.
remote: Total 4051 (delta 0), reused 0 (delta 0), pack-reused 4051
Receiving objects: 100% (4051/4051), 2.04 MiB | 1.85 MiB/s, done.
Resolving deltas: 100% (1465/1465), done.
Checking out files: 100% (1384/1384), done.
```

Si analizamos las partes, podemos ver lo siguiente:

- Se utiliza el comando **“git clone”** como primer comando.
- A este le añadimos una URL como siguiente argumento de la llamada. Esta URL será la del proyecto que vamos a clonar en nuestro equipo y la encontraremos en el proyecto remoto de git.
- De manera opcional, podemos indicar el nombre de una carpeta, donde se guardaría.

codenotch

DEFINICIÓN

GIT STATUS

Para saber en todo momento el estado de los archivos que tenemos en nuestro local y sobre los que estamos trabajando en el proyecto, podemos usar **“git status”**.

Este comando es uno de los más usados mientras se está desarrollando, ya nos dice exactamente qué archivos se han modificado en el proyecto.

Veamos un ejemplo.

```
javip@Javier MINGW64 ~/OneDrive/Codenotch/fundamentos/mygrit (master)
$ git status
On branch master
Your branch is up to date with 'origin/master'.

nothing to commit, working tree clean
```

codenotch

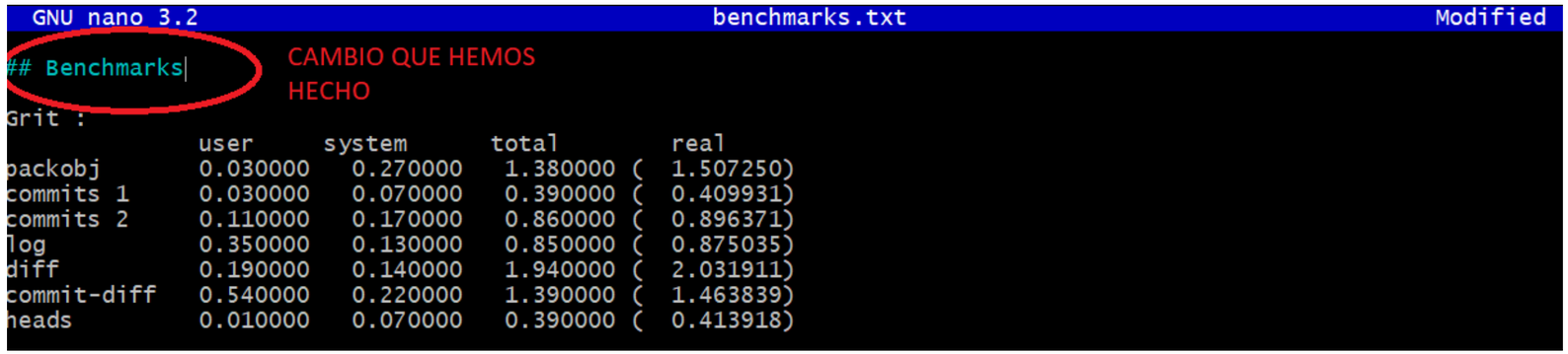
DEFINICIÓN

GIT STATUS

Vemos que, como en el proyecto que nos hemos clonado a nuestro equipo no hemos hecho ningún cambio aún, “git status” nos indica varias cosas:

- Estamos en la rama principal o “master”.
- La rama está actualizada con la última versión del proyecto.
- No hay ningún cambio en nuestro equipo para incorporar al proyecto.

Vamos a hacer ahora un pequeño cambio, como es un comentario sin influencia, en uno de los archivos del proyecto:



codenotch

DEFINICIÓN

GIT STATUS

Observamos el comentario que hemos hecho en el archivo. A continuación lo guardamos con esa modificación y, vamos a ver que nos dice GIT cuando consultamos el estado del proyecto.

```
javip@Javier MINGW64 ~/OneDrive/Codenotch/fundamentos/mygrit (master)
$ git status
On branch master
Your branch is up to date with 'origin/master'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   benchmarks.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

codenotch

DEFINICIÓN

GIT ADD

Como vemos, nos detecta cambios en el proyecto, concretamente en el archivo que hemos modificado.

Nos dice que estos cambios están hechos en local, pero no están en el **“stage”**.

Esto significa, que no los hemos añadido o, mejor dicho, “no están marcados para incorporarse al proyecto”.

Para hacer esto que nos indica GIT debemos utilizar el comando **GIT ADD**.

codenotch

DEFINICIÓN

GIT ADD

Git add nos permite añadir los cambios de nuestros archivos al **"stage"** de GIT.

El **"stage"** es un estado que indica que esos cambios están preparados para ser incorporados al proyecto.

Vamos a ver un ejemplo

```
javip@Javier MINGW64 ~/OneDrive/Codenotch/fundamentos/mygrit (master)
$ git add benchmarks.txt

javip@Javier MINGW64 ~/OneDrive/Codenotch/fundamentos/mygrit (master)
$ git status
On branch master
Your branch is up to date with 'origin/master'.

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   benchmarks.txt
```

codenotch

DEFINICIÓN

GIT ADD

Vemos que simplemente hemos llamado al comando **“git add”** seguido del archivo que queremos añadir al **“stage”**.

Si quisiéramos añadir más archivos, solo tendríamos que poner sus nombres uno tras otro separados por espacios o, si queremos añadirlos todos de golpe, usar el siguiente comando:

```
javip@Javier MINGW64 ~/OneDrive/Codenotch/fundamentos/mygrit (master)
$ git add .
```

Ahora, al consultar el estado del proyecto, vemos como el color del archivo ha cambiado de rojo a verde y, además, GIT nos dice que los cambios están preparados para ser **“committed”**.

Pero, ¿qué es esto de ser **“committed”**? A continuación vamos a verlo en el siguiente punto.

codenotch

DEFINICIÓN

GIT COMMIT

Git commit es la confirmación de tus cambios. Todos los cambios que se hayan dejado preparados en el “stage” con el comando “git add” serán incluidos en esta confirmación.

En cambio, si algún archivo está modificado pero no se ha confirmado en el “stage” no serán incluidos en la confirmación.

Vamos a ver como se haría esto:

```
javip@Javier MINGW64 ~/OneDrive/Codenotch/fundamentos/mygrit (master)
$ git commit -m "My first commit"
[master b1eba7a] My first commit
 1 file changed, 2 insertions(+)
```

Vemos las partes del comando ejecutado:

- El comando principal **“git commit”**.
- La opción –m para incluir un mensaje descriptivo de los cambios.
- El mensaje descriptivo entre comillas.

codenotch

DEFINICIÓN

GIT COMMIT

Es realmente importante poner un comentario con sentido y más o menos preciso por cada “commit” que se haga, ya que después, cualquier persona que revise estos cambios, se puede hacer una idea global de lo que hay en el “commit”.

Lo podemos ver como el titular de nuestros cambios. Así, seremos capaces de crear un log de cambios más completo y preciso.

Además, vemos que GIT nos ha indicado que en este “commit” se ha cambiado un archivo y ha habido 2 inserciones de líneas (una línea en blanco y la línea del comentario).

codenotch

DEFINICIÓN

GIT PUSH

Ahora nuestros cambios están confirmados y “committed”, pero no se han incorporado al proyecto aún. ¿Cómo podemos hacer esto? En este caso, lo que tenemos que ejecutar es el comando “git push”.

GIT PUSH incorpora los cambios confirmados al proyecto global, de este modo, todos los colaboradores del proyecto ya los tienen disponible y pueden descargarlos para incorporarlos a sus clones del proyecto en cada uno de sus equipos.

De hecho, si hacemos “git status”, vemos que se nos dice que nuestro local está por delante del proyecto final en 1 commit.

```
javip@Javier MINGW64 ~/OneDrive/Codenotch/fundamentos/mygrit (master)
$ git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
```

codenotch

DEFINICIÓN

GIT PUSH

Para incorporar estos cambios al proyecto principal, solo hay que ejecutar el comando siguiente:

```
javip@Javier MINGW64 ~/OneDrive/Codenotch/fundamentos/mygrit (master)
$ git push
```

Ahora, nuestro comentario estará en el proyecto global, y todos los colaboradores lo tendrán disponible.

De esta afirmación surge la siguiente pregunta, ¿cómo podemos descargar los cambios de otro colaborador en nuestro proyecto para tenerlo actualizado?

codenotch

DEFINICIÓN

GIT PULL

GIT PULL descarga todas las actualizaciones y cambios del proyecto en nuestro equipo de manera local. Así, somos capaces de incorporar lo que el resto de colaboradores ha hecho.

Simplemente se ejecuta el comando:

```
javip@Javier MINGW64 ~/OneDrive/Codenotch/fundamentos/mygrit (master)
$ git pull
Already up to date.
```

Como vemos, en este caso, nadie ha hecho cambios, y GIT nos indica que estamos al día.
Si los hubiera, se nos listarían todos los archivos modificados, creados o borrados; y nuestro proyecto quedaría en el mismo estado que el proyecto global.

codenotch

DEFINICIÓN

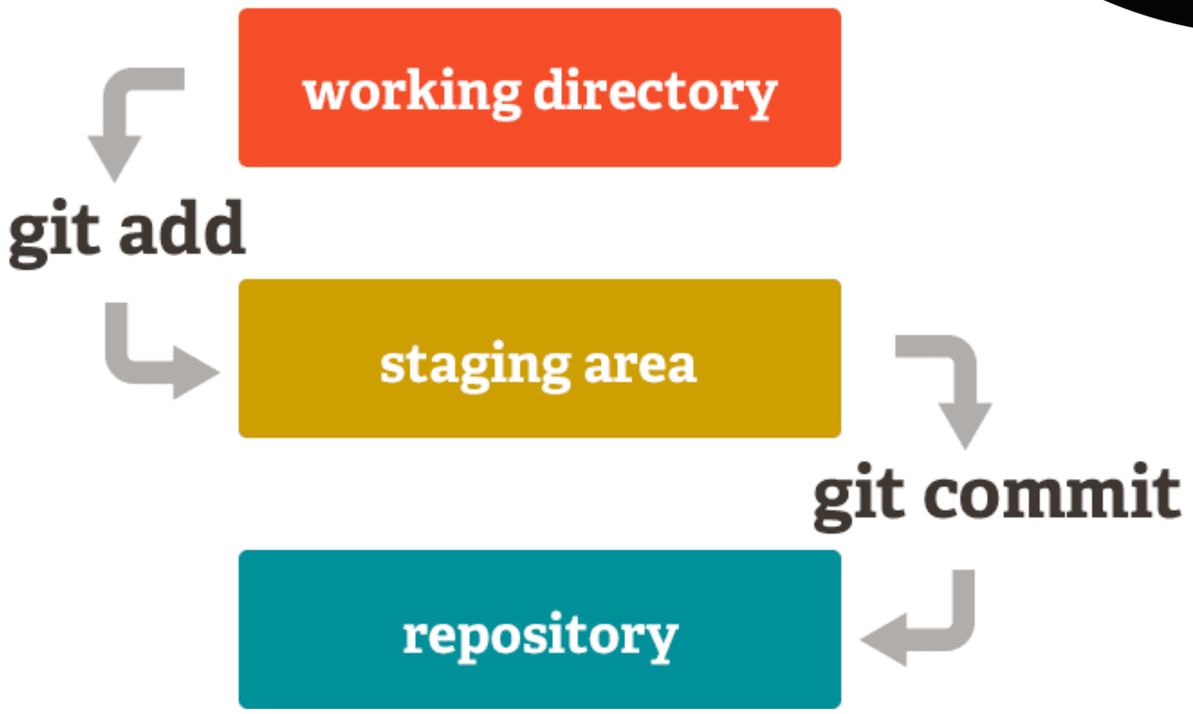
GIT RESET

El comando "git reset" es una herramienta compleja y versátil para deshacer cambios.

Se invoca principalmente de tres formas distintas que se corresponden con los argumentos de líneas de comandos:

- ***--soft***
- ***--mixed***
- ***--hard***

Cada uno de los tres argumentos se corresponde con los tres mecanismos de gestión de estados de git: ***repositorio, staged, working area.***



codenotch

DEFINICIÓN

GIT RESET

Por lo tanto el comando reset nos permite sacar archivos que estén en el estado de **commit**.

--soft: todos los ficheros de los commits que se eliminan pasan al área de staging.
--mixed: todos los ficheros de los commits que se eliminan se quedan en el working area.
--hard: todos los ficheros de los commits que se eliminan, se eliminan definitivamente.

```
git reset --soft HEAD~ || git reset --mixed HEAD~  ||  git reset --hard HEAD~
```

Además lo podemos usar si por error se ha subido un fichero al área del staged y se le quiere volver al estado working área a través de este comando:

```
git reset HEAD /path/to/file
```

codenotch

Ramas

codenotch

DEFINICIÓN

¿Qué son las ramas en GIT?

Una rama de GIT es una “instantánea” o copia del proyecto en un momento concreto y para un fin determinado.

El uso de ramas es fundamental cuando se trabaja en equipo ya que, entre otras cosas, evita conflictos en el proyecto principal. ¿Y qué es esto de los conflictos?

Imaginemos que un desarrollador tiene una tarea “Añadir la fecha de cumpleaños al formulario de registro” y otro tiene otra tarea “Cambiar los colores del formulario de registro”.

Muy seguramente, ambos van a tocar el mismo archivo que contiene el código del formulario de registro. Y esto podría derivar en un problema a la hora de incorporar ambos cambios, pues en las mismas líneas existirían cambios distintos

codenotch

DEFINICIÓN

¿Qué son las ramas en GIT?

Usando ramas evitamos esto. Se crea una rama, y cuando se hace “git push”, se hace sobre esa rama, es decir, sobre esa copia concreta del proyecto.

Luego, cuando se revisen los cambios y se decidan incorporar, lo que se hace es incorporar esa rama a la rama principal o “master”, que es el proyecto global, y así estará disponible para todos los desarrolladores.

Vamos a ver como crear una rama, consultar la rama en la que estamos, las ramas disponibles y cómo cambiar de una rama a otra; entre otras cosas.

codenotch

DEFINICIÓN

GIT BRANCH

GIT BRANCH, junto con todas sus opciones, va a ser nuestro gran aliado para el trabajo con ramas.

Por un lado, si lo ejecutamos tal cuál, nos dice la rama en la que estamos en ese momento, y lista el resto de ramas disponibles.

```
javip@Javier MINGW64 ~/OneDrive/Codenotch/fundamentos/mygrit (master)
$ git branch
* master
```

Como vemos, GIT nos indica que estamos en la rama master (en color verde) y además es la única rama que tenemos, pues evidentemente aún no hemos creado ninguna nueva.

codenotch

DEFINICIÓN

GIT BRANCH

Ahora, para crear una nueva rama, debemos ejecutar lo siguiente:

```
javip@Javier MINGW64 ~/OneDrive/Codenotch/fundamentos/mygrit (master)
$ git branch my_first_branch
```

Usamos git branch + el nombre de la rama que vamos a crear y con esto ya estaría creada. Ahora, para ver esta rama en nuestro listado, volveríamos a usar GIT BRANCH de la siguiente manera:

```
javip@Javier MINGW64 ~/OneDrive/Codenotch/fundamentos/mygrit (master)
$ git branch
* master
  my_first_branch
```

Vemos que ahora GIT nos muestra la nueva rama que hemos creado, aunque nos indica que seguimos en la rama “master” (asterisco y color verde).

codenotch

DEFINICIÓN

GIT CHECKOUT

Lo que toca a continuación es cambiarnos de rama, de la principal del proyecto a la que acabamos de crear. Para existe GIT CHECKOUT.

Este comando, seguido del nombre de la rama que queramos (git checkout nombrerama), nos cambia a esa rama en concreto del proyecto, y los archivos estarán en el estado en el que estaban cuando se creó la rama.

Veamos como es esto:

```
javip@Javier MINGW64 ~/OneDrive/Codenotch/fundamentos/mygrit (master)
$ git checkout my_first_branch
Switched to branch 'my_first_branch'

javip@Javier MINGW64 ~/OneDrive/Codenotch/fundamentos/mygrit (my_first_branch)
$
```

Vemos que GIT nos informa que ha cambiado a la rama “my_first_branch”, y además podemos verlo en la siguiente línea en color azul, ya que nos indica en la rama actual en la que nos encontramos.

codenotch

DEFINICIÓN

GIT CHECKOUT

Además, si utilizamos el ya conocido comando GIT BRANCH, que nos muestra la lista de todas las ramas disponibles y nos marca en cuál estamos, podemos encontrar lo siguiente:

```
javip@Javier MINGW64 ~/OneDrive/Codenotch/fundamentos/mygrit (my_first_branch)
$ git branch
  master
* my_first_branch
```

Vemos que ahora con asterisco y color verde, tenemos la rama “my_first_branch”, indicándonos GIT de esta manera que es la rama actual en la que nos encontramos.

codenotch

DEFINICIÓN

GIT MERGE

Cuando hemos realizado los cambios en nuestra rama, y los queremos incorporar a la rama “master” o principal, lo haremos utilizando el comando git merge.

Este comando fusionará nuestra rama con la principal, mirando los cambios hechos e incorporándolos al proyecto global. En el caso de que no se pudiera por encontrar algún conflicto, nos lo haría saber, y el proceso no se completaría hasta que corrigiéramos los conflictos, poniendo en común nuestros cambios con los que haya podido hacer otro colaborador.

Esta es la manera en la que GIT evita problemas entre códigos de varios colaboradores del proyecto.

codenotch

DEFINICIÓN

GIT MERGE

Ahora veamos cómo se ejecuta el comando GIT MERGE:

```
javip@Javier MINGW64 ~/OneDrive/Codenotch/fundamentos/mygrit (master)
$ git merge my_first_branch
Updating b1eba7a..bf89cf5
Fast-forward
 benchmarks.txt | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Sobre la rama principal hacemos GIT MERGE seguido del nombre de la rama que queremos unir a la rama principal, y esta, al no encontrar conflictos, incorpora todos los cambios realizados.

codenotch

DEFINICIÓN

GIT MERGE

Si ahora vemos los cambios con GIT STATUS sobre la rama master, veremos que hay dos, el del commit que hicimos sobre esta rama y el del merge de la otra rama que acabamos de fusionar:

```
javip@Javier MINGW64 ~/OneDrive/Codenotch/fundamentos/mygrit (master)
$ git status
On branch master
Your branch is ahead of 'origin/master' by 2 commits.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
```

codenotch

DEFINICIÓN

GIT IGNORE

Una herramienta importante cuando estamos en proyectos grandes es la del archivo ".gitignore".

Este archivo se ubica a nivel de directorio y contiene un listado de todos los archivos que **NO** queremos que GIT haga un seguimiento, es decir, que no van a ser incluidos en ninguno de los estados del proyecto, hagamos lo que hagamos.

Esto es útil cuando en el mismo directorio del proyecto o sus subdirectorios, ponemos archivos de metainformación, guías de uso o documentaciones, que evidentemente son útiles, pero no necesitan estar online o que otros colaboradores vean.

codenotch

DEFINICIÓN

GIT IGNORE

Lo único que tenemos que hacer es crear el archivo ".gitignore" en el directorio que queramos y añadir los archivos que no queremos que se incluyan. Veamos un ejemplo del contenido de un archivo ".gitignore":

```
javip@Javier MINGW64 ~/OneDrive/Codenotch/fundamentos/mygrit (master)
$ cat .gitignore
pkg
.DS_Store
*.exe
*.zip
```

Vemos que al pintar el contenido nos encontramos con:

- Los archivo "pkg" y ".DS_Store" no serán incluidos.
- Cualquier archivo que acabe con la extensión ".exe" o ".zip" no será incluido.

La wildcard "*" nos permite decir "lo que sea" en cuanto al nombre de los archivos, por tanto, nos ahorra escribir en algunos casos muchos nombres.

codenotch

Glosario

- GIT
- GIT INIT, CLONE, STATUS
- GIT ADD, COMMIT, PUSH
- GIT PULL
- GIT BRANCH
- GIT CHECKOUT
- GIT MERGE
- GIT IGNORE

codenotch

#Retos

codenotch

Reto 1

1. Crea un repositorio de GitHub con el fichero Readme.md.
2. Clonar el repositorio en vuestro equipo.
3. Dentro de la carpeta del repositorio crear un proyecto node.
4. Copia en dicha carpeta el reto 2 del día anterior (día 5).
5. Añade a este archivo de funciones, una nueva función que haga el cuadrado de un numero.
6. Sube el proyecto a tu repositorio git creado.

codenotch

Reto 2

1. En una carpeta fuera del repositorio del reto anterior. Crear un proyecto node y crea un archivo Readme.md que describa los pasos que debes realizar para subir un proyecto a Git.
2. Crear un repositorio local con git init dentro de la carpeta creada en el punto anterior.
3. Crear un nuevo repositorio en Github
4. Enlaza el repositorio local al remoto.
5. Añade el fichero Readme.md al “stage”.
6. Comprueba el estado.
7. Hemos subido el archivo al “stage”, pero ahora nos damos cuenta que no es el archivo correcto y queremos sacarlo del “stage”. (Es decir dar marcha atrás al git add).
8. Haz capturas de todos los pasos que has ido realizando y guárdalas en una carpeta llamada “reto_2”.

codenotch

Reto 3

1. Añade el archivo .md creado anteriormente de nuevo al “stage”.
2. Comprueba el estado.
3. Haz el commit de este archivo.
4. Comprueba de nuevo el estado.
5. Nuevamente nos damos cuenta de que hemos vuelto ha añadir el archivo equivocado. Ahora debemos revertir el “git commit”.
6. Haz capturas de todos los pasos realizados y guárdalas como archivos individuales .png

Reto 4

1. Crea un archivo Git ignore, y añade en él la carpeta “reto_2” y todas las imágenes.
2. Ahora sube los cambios realizados a tu repositorio.
3. ¿Qué ha ocurrido? Explica lo ocurrido en el archivo .md.

codenotch

Reto 5

1. Hasta ahora hemos trabajado sobre la rama “main”/”master”. Crea ahora dos ramas en el proyecto del reto 1, “rama1” y “rama2” .
2. Una vez creadas las ramas, cámbiate a la “rama1” y borra desde ella la función suma.
3. Sube los cambios al repositorio y fusiónalos con el proyecto principal.
4. Cámbiate a la “rama2” y modifica la función suma, para que ahora reciba tres números por parámetro y los sume.
5. Sube los cambios al repositorio y fusiónalos con el proyecto principal.
6. ¿Qué ha ocurrido? Explica lo ocurrido en el archivo .md.

Reto 6

1. Sube los últimos cambios a tu repositorio Git.
2. Crea una versión del proyecto actualizado.


codenotch