

# Desarrollo de código con GIT

Tutorial de Git, GitHub y GitKraken

José Luis González Sánchez





# ¿Qué vamos a aprender?

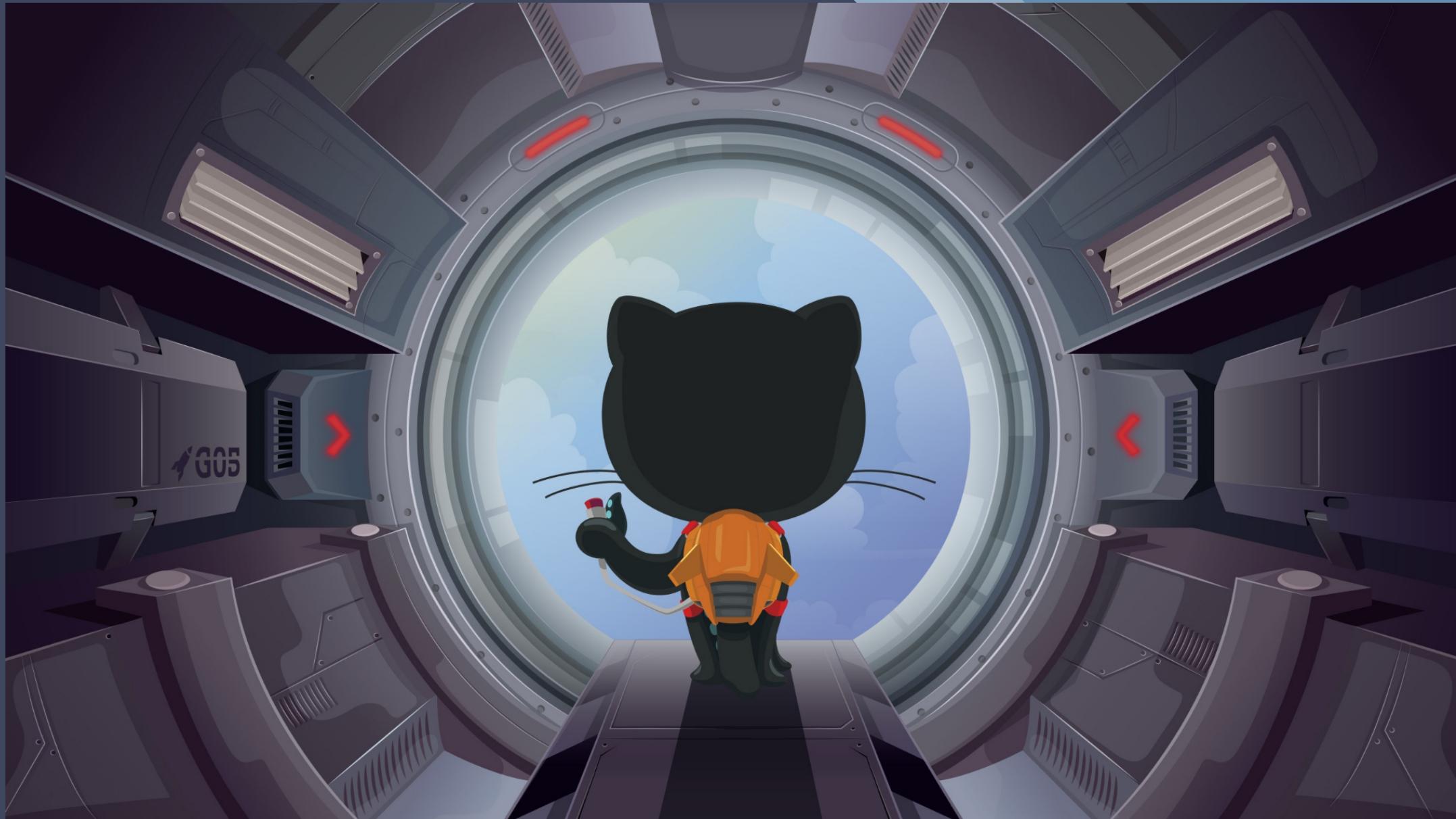
- Aprenderemos a mejorar el desarrollo de código de manera avanzada y colaborativa haciendo uso de un sistema de control de versiones.



git

Github





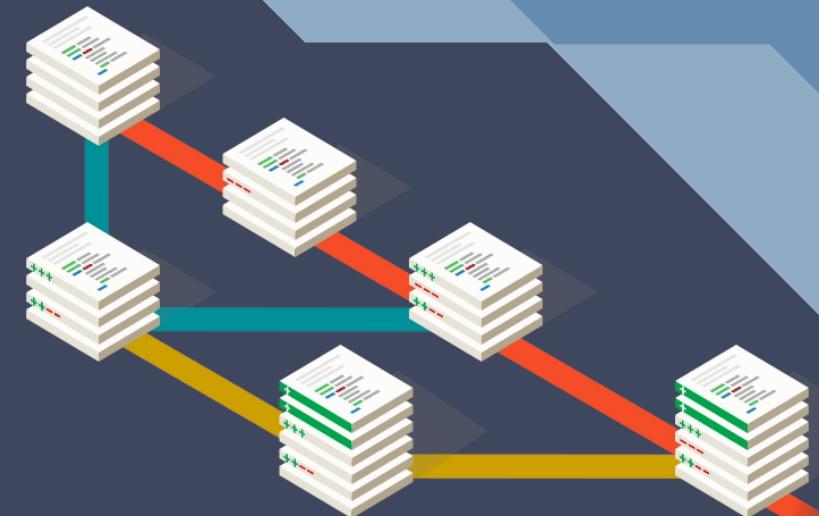
# Sistemas de Control de Versiones

¿Qué es y para qué se utiliza?

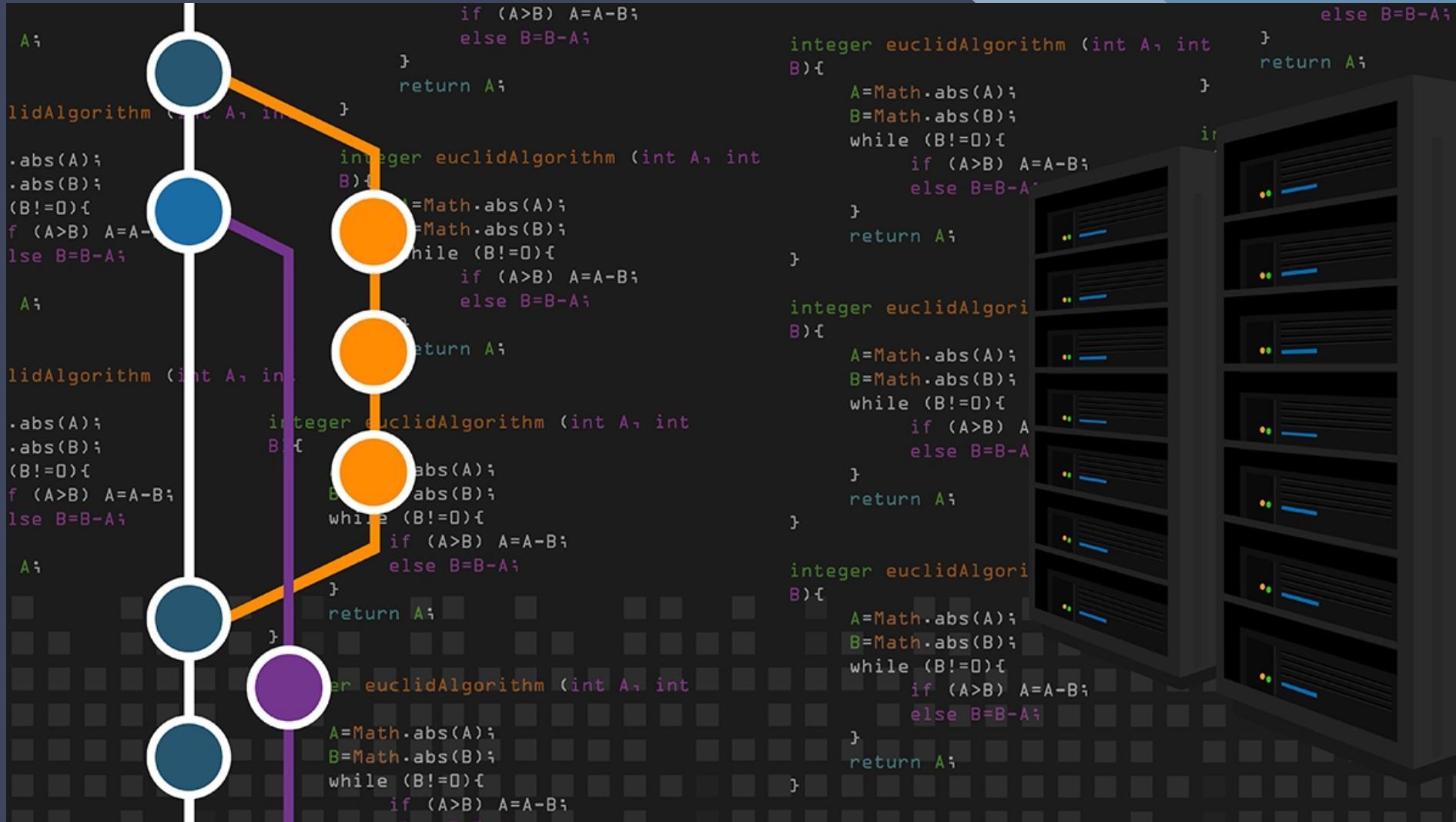


# Sistemas de Control de Versiones

- Un **Sistema de Control de Versiones** (VCS) nos permite guardar un registro de las modificaciones que realizamos sobre un fichero o conjunto de ficheros a lo largo del tiempo de tal manera que sea posible recuperar versiones específicas más adelante. Habitualmente se utiliza en entornos de desarrollo de software, pero puede resultar de gran utilidad para cualquier persona que necesite un control robusto sobre la tarea que está realizando.
- Es una valiosa herramienta con numerosos beneficios para un flujo de trabajo de equipos de software de colaboración. Cualquier proyecto de software que tiene más de un desarrollador manteniendo archivos de código fuente debe usar un VCS. Además, los proyectos mantenidos por una sola persona se beneficiarán enormemente de su uso. Se puede decir que no hay una razón válida para privarse del uso de un VCS en cualquier proyecto moderno de desarrollo de software.



# Sistemas de Control de Versiones





# Características

- **Resolución de conflictos**
  - Es muy probable que los miembros del equipo tengan la necesidad de realizar cambios en el mismo archivo de código fuente al mismo tiempo. Un VCS monitoriza y ayuda a poder resolver los conflictos entre varios desarrolladores.
- **Revertir y deshacer los cambios en el código fuente**
  - Al empezar a monitorizar un sistema de archivos de códigos fuente, existe la posibilidad de revertir y deshacer rápidamente a una versión estable conocida.
- **Copia de seguridad externa del código fuente**
  - Se debe crear una instancia remota del VCS que se puede alojar de forma externa con un tercero de confianza y con ello, se conservará una copia del código fuente.



# Algunos VCS

- **Git:** es una de las mejores herramientas de control de versiones disponible en el mercado actual. Es un modelo de repositorio distribuido compatible con sistemas y protocolos existentes como HTTP, FTP, SSH y es capaz de manejar eficientemente proyectos pequeños a grandes.
- **CVS:** es otro sistema de control de versiones muy popular. Es un modelo de repositorio cliente-servidor donde varios desarrolladores pueden trabajar en el mismo proyecto en paralelo. El cliente CVS mantendrá actualizada la copia de trabajo del archivo y requiere intervención manual sólo cuando ocurre un conflicto de edición.
- **Apache Subversion (SVN):** abreviado como SVN, apunta a ser el sucesor más adecuado. Es un modelo de repositorio cliente-servidor donde los directorios están versionados junto con las operaciones de copia, eliminación, movimiento y cambio de nombre.
- **Mercurial:** es una herramienta distribuida de control de versiones que está escrita en Python y destinada a desarrolladores de software. Los sistemas operativos que admite son similares a Unix, Windows y macOS. Tiene un alto rendimiento y escalabilidad con capacidades avanzadas de ramificación y fusión y un desarrollo colaborativo totalmente distribuido. Además, posee una interfaz web integrada.
- **Monotone:** está escrito en C ++ y es una herramienta para el control de versiones distribuido. El sistema operativo que admite incluye Unix, Linux, BSD, Mac OS X y Windows. Brinda un buen apoyo para la internacionalización y localización. Además, utiliza un protocolo personalizado muy eficiente y robusto llamado Netsync.



# Conceptos clave

- **Repositorio:** Lugar en el que se almacenan los datos actualizados e históricos de cambios, a menudo en un servidor. A veces se le denomina depósito o depot. Puede ser un sistema de archivos en un disco duro, un banco de datos, etc..
- **Módulo:** Conjunto de directorios y/o archivos dentro del repositorio que pertenecen a un proyecto común.
- **Revisión ("version"):** es una versión determinada de la información que se gestiona. Hay sistemas que identifican las revisiones con un contador (Ej. subversion). Hay otros sistemas que identifican las revisiones mediante un código de detección de modificaciones (Ej. Git usa SHA1). A la última versión se le suele identificar de forma especial con el nombre de **HEAD**. Para marcar una revisión concreta se usan los rótulos o tags.



# Conceptos clave

**Rotular ("tag"):** Darle a alguna versión de cada uno de los ficheros del módulo en desarrollo en un momento preciso un nombre común ("etiqueta" o "rótulo") para asegurarse de reencontrar ese estado de desarrollo posteriormente bajo ese nombre.

En la práctica se rotula a todos los archivos en un momento determinado. Para eso el módulo se "congela" durante el rotulado para imponer una versión coherente. Pero bajo ciertas circunstancias puede ser necesario utilizar versiones de algunos ficheros que no coinciden temporalmente con las de los otros ficheros del módulo.

Los tags permiten identificar de forma fácil revisiones importantes en el proyecto. Por ejemplo se suelen usar tags para identificar el contenido de las versiones publicadas del proyecto. En algunos sistemas se considera un tag como una rama en la que los ficheros no evolucionan, están congelados.



# Conceptos clave

- **Línea base ("baseline"):** Una revisión aprobada de un documento o fichero fuente, a partir del cual se pueden realizar cambios subsiguientes.
- **Abrir rama ("branch") o ramificar:** Un módulo puede ser branched o bifurcado en un instante de tiempo de forma que, desde ese momento en adelante se tienen dos copias (ramas) que evolucionan de forma independiente siguiendo su propia línea de desarrollo. El módulo tiene entonces 2 (o más) "ramas". La ventaja es que se puede hacer un "merge" de las modificaciones de ambas ramas, posibilitando la creación de "ramas de prueba" que contengan código para evaluación, si se decide que las modificaciones realizadas en la "rama de prueba" sean preservadas, se hace un "merge" con la rama principal. Son motivos habituales para la creación de ramas la creación de nuevas funcionalidades o la corrección de errores.
- **Desplegar ("Check-out", "checkout", "co"):** Un despliegue crea una copia de trabajo local desde el repositorio. Se puede especificar una revisión concreta, y predeterminadamente se suele obtener la última.



# Conceptos clave

**"Publicar" o "Enviar" ("commit", "check-in", "ci", "install", "submit"):** Un commit sucede cuando una copia de los cambios hechos a una copia local es escrita o integrada sobre el repositorio.

- **Conflicto:** Un conflicto ocurre cuando el sistema no puede manejar adecuadamente cambios realizados por dos o más usuarios en un mismo archivo. Por ejemplo, si se da esta secuencia de circunstancias:

- Los usuarios X e Y despliegan versiones del archivo A en que las líneas n1 hasta n2 son comunes.
- El usuario X envía cambios entre las líneas n1 y n2 al archivo A.
- El usuario Y no actualiza el archivo A tras el envío del usuario X.
- El usuario Y realiza cambios entre las líneas n1 y n2.
- El usuario Y intenta posteriormente enviar esos cambios al archivo A.
- El sistema es incapaz de fusionar los cambios. El usuario Y debe resolver el conflicto combinando los cambios, o eligiendo uno de ellos para descartar el otro.



# Conceptos clave

- **Resolver:** El acto de la intervención del usuario para atender un conflicto entre diferentes cambios al mismo archivo.
- **Cambio ("change", "diff", "delta"):** 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.
- **Lista de cambios ("changelist", "change set", "patch"):** En muchos sistemas de control de versiones con commits multi-cambio atómicos, una lista de cambios identifica el conjunto de cambios hechos en un único commit. Esto también puede representar una vista secuencial del código fuente, permitiendo que el fuente sea examinado a partir de cualquier identificador de lista de cambios particular.
- **Exportación ("export"):** Una exportación es similar a un check-out, salvo porque crea un árbol de directorios limpio sin los metadatos de control de versiones presentes en la copia de trabajo. Se utiliza a menudo de forma previa a la publicación de los contenidos.



# Conceptos clave

- **Importación ("import"):** Una importación es la acción de copiar un árbol de directorios local (que no es en ese momento una copia de trabajo) en el repositorio por primera vez.
- **Integración o fusión ("merge"):** Una integración o fusión une dos conjuntos de cambios sobre un fichero o un conjunto de ficheros en una revisión unificada de dicho fichero o ficheros.
  - Esto puede suceder cuando un usuario, trabajando en esos ficheros, actualiza su copia local con los cambios realizados, y añadidos al repositorio, por otros usuarios.
  - Análogamente, este mismo proceso puede ocurrir en el repositorio cuando un usuario intenta check-in sus cambios. Puede suceder después de que el código haya sido branched, y un problema anterior al branching sea arreglado en una rama, y se necesite incorporar dicho arreglo en la otra.
  - Puede suceder después de que los ficheros hayan sido branched, desarrollados de forma independiente por un tiempo, y que entonces se haya requerido que fueran fundidos de nuevo en un único trunk unificado.



# Conceptos clave

- **Integración inversa:** El proceso de fundir ramas de diferentes equipos en el trunk principal del sistema de versiones.
- **Actualización ("sync" o "update"):** Una actualización integra los cambios que han sido hechos en el repositorio (por ejemplo por otras personas) en la copia de trabajo local.
- **Copia de trabajo ("workspace"):** La copia de trabajo es la copia local de los ficheros de un repositorio, en un momento del tiempo o revisión específicos. Todo el trabajo realizado sobre los ficheros en un repositorio se realiza inicialmente sobre una copia de trabajo, de ahí su nombre. Conceptualmente, es un cajón de arena o sandbox.
- **Congelar:** Significa permitir los últimos cambios (commits) para solucionar las fallas a resolver en una entrega (release) y suspender cualquier otro cambio antes de una entrega, con el fin de obtener una versión consistente. Si no se congela el repositorio, un desarrollador podría comenzar a resolver una falla cuya resolución no está prevista y cuya solución dé lugar a efectos colaterales imprevistos.

# GIT

Nuestro gran amigo

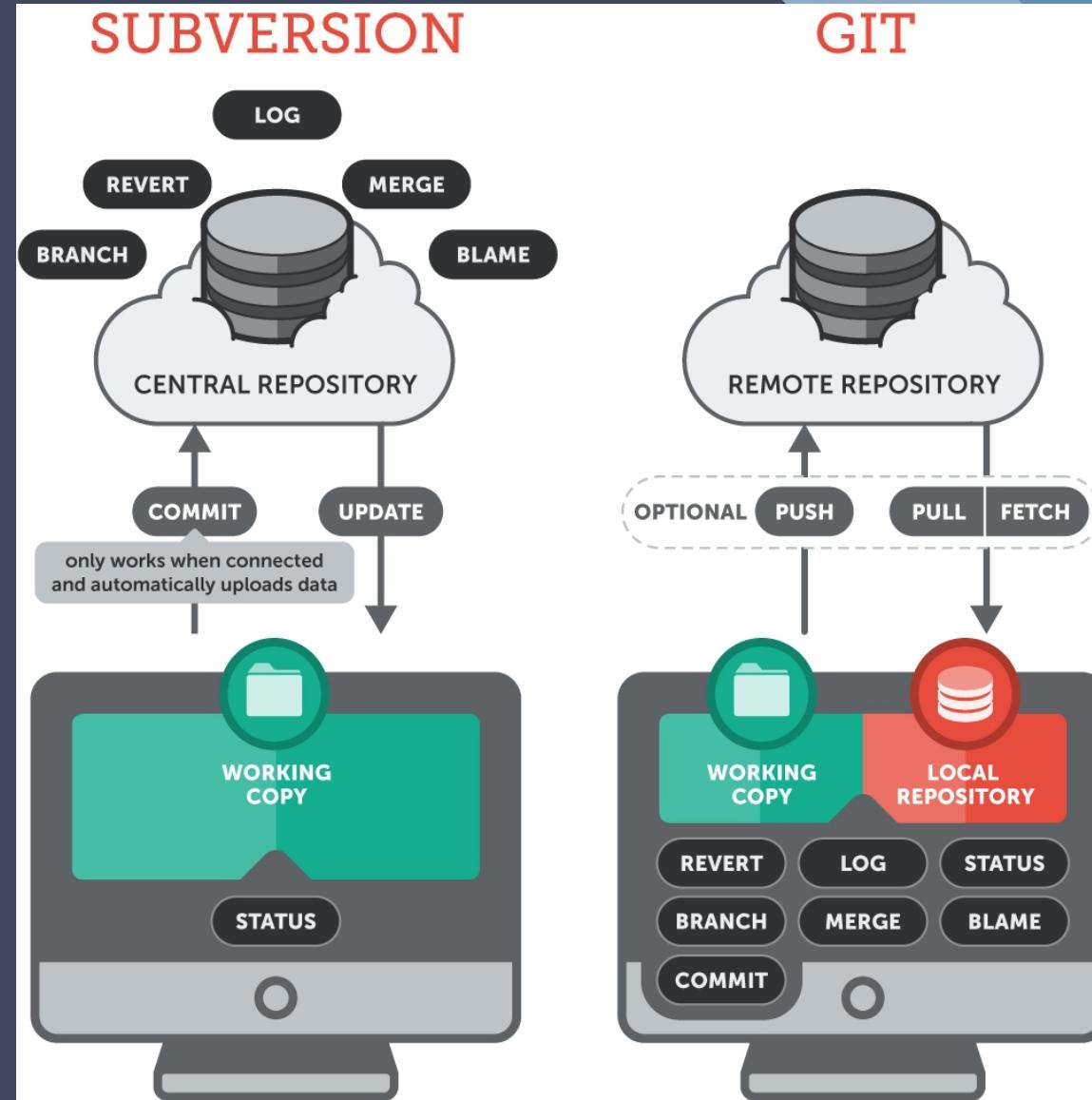


# Git

- Un sistema distribuido de control de versiones
- Muy potente
- No depende de un repositorio central
- Creado por Linus Torvalds
- **Es software libre**
- Disponemos de un historial de revisiones completo.
- Trabajar con ramas (branches) diferentes de código y fusiones (merges) de ramas decódigo es un proceso ágil.
- Gestión distribuida; Los cambios se importan como ramas adicionales y pueden ser fusionados de la misma manera como se hace en la rama local.

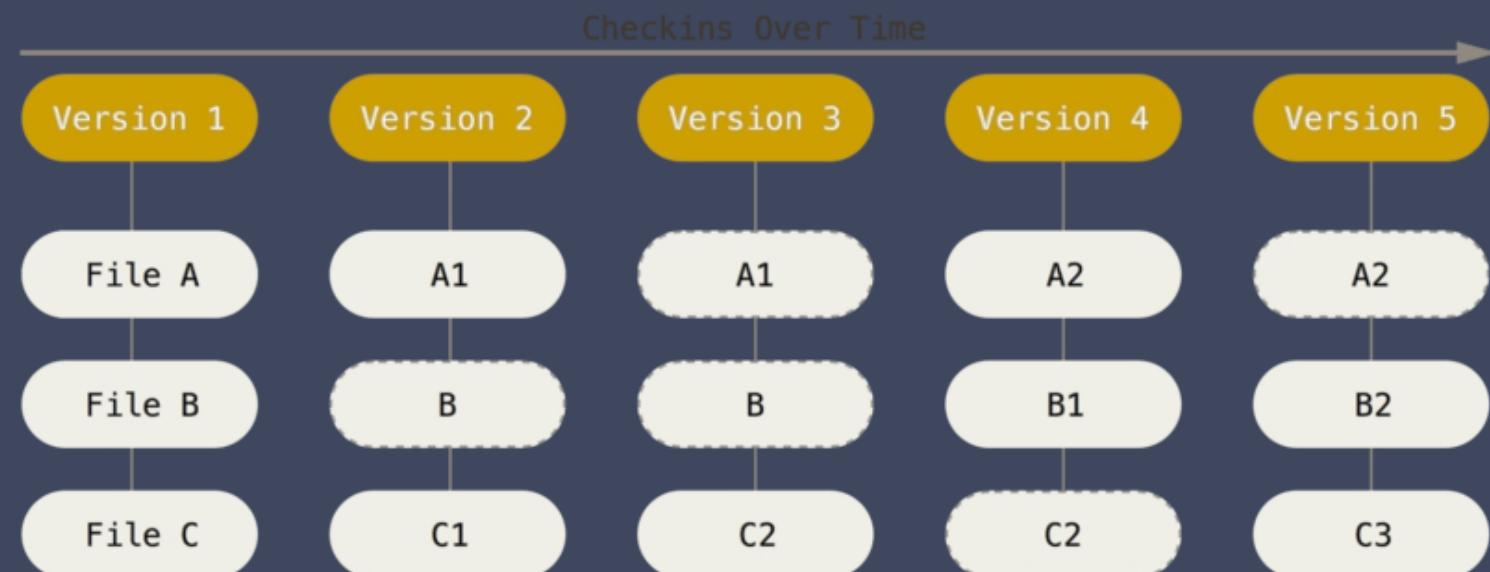


# Git



# Git

- Git maneja sus datos como un conjunto de **copias instantáneas** de un sistema de archivos miniatura. Cada vez que confirmas un cambio, o guardas el estado de tu proyecto en Git, él básicamente **toma una foto del aspecto de todos tus 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 sus datos como una secuencia de copias instantáneas.



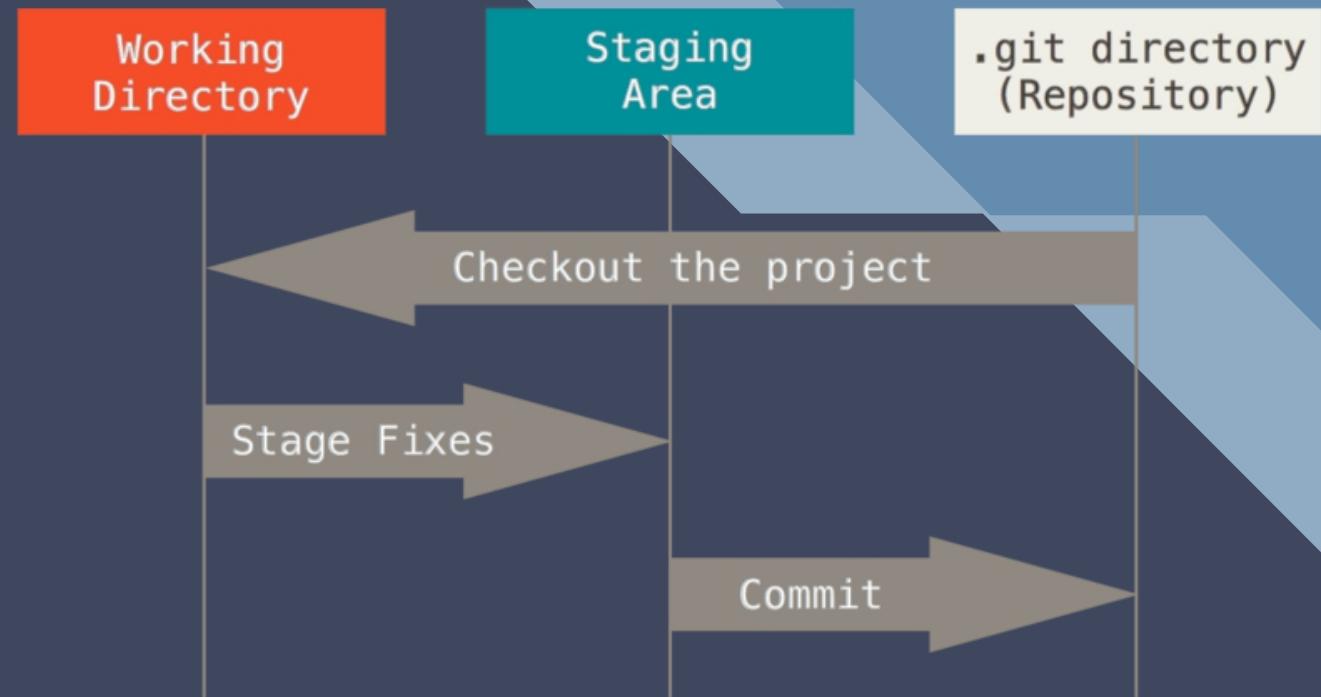


# Git

- Git tiene tres estados principales en los que se pueden encontrar tus archivos: confirmado (committed), modificado (modified), y preparado (staged).
  - **Confirmado:** significa que los datos están almacenados de manera segura en tu base de datos local.
  - **Modificado:** significa que has modificado el archivo pero todavía no lo has confirmado a tu base de datos.
  - **Preparado:** significa que has marcado un archivo modificado en su versión actual para que vaya en tu próxima confirmación.
- Esto nos lleva a las **tres secciones principales de un proyecto de Git:**
  - **El directorio de Git o Repositorio (Git directory)**
  - **El directorio de trabajo (working directory)**
  - **El área de preparación o de almacenamiento intermedio local (staging area).**

# Git

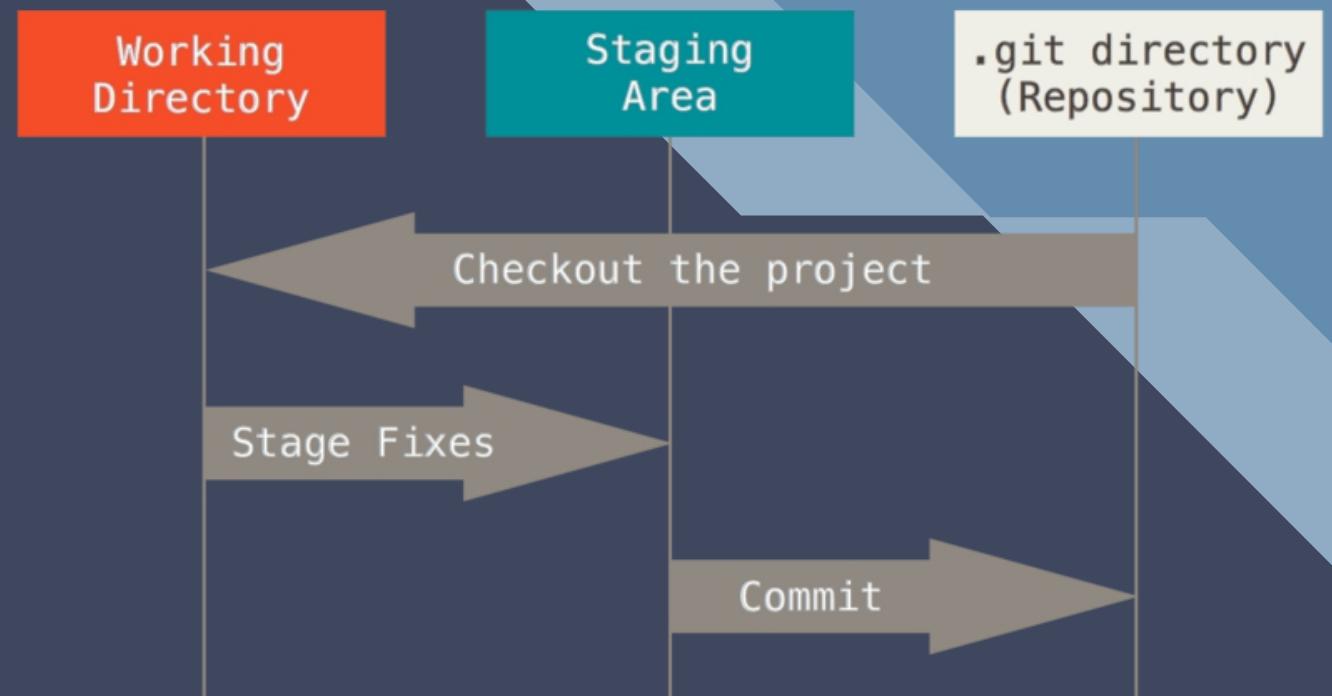
- El **directorio de Git** es donde se almacenan los metadatos y la base de datos de objetos para tu proyecto. Es la parte más importante de Git, y es lo que se copia cuando clonas un repositorio desde otra computadora.
- El **directorio de trabajo** es una copia de una versión del proyecto. Estos archivos 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.
- El **área de preparación es un archivo o almacenamiento**, 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.





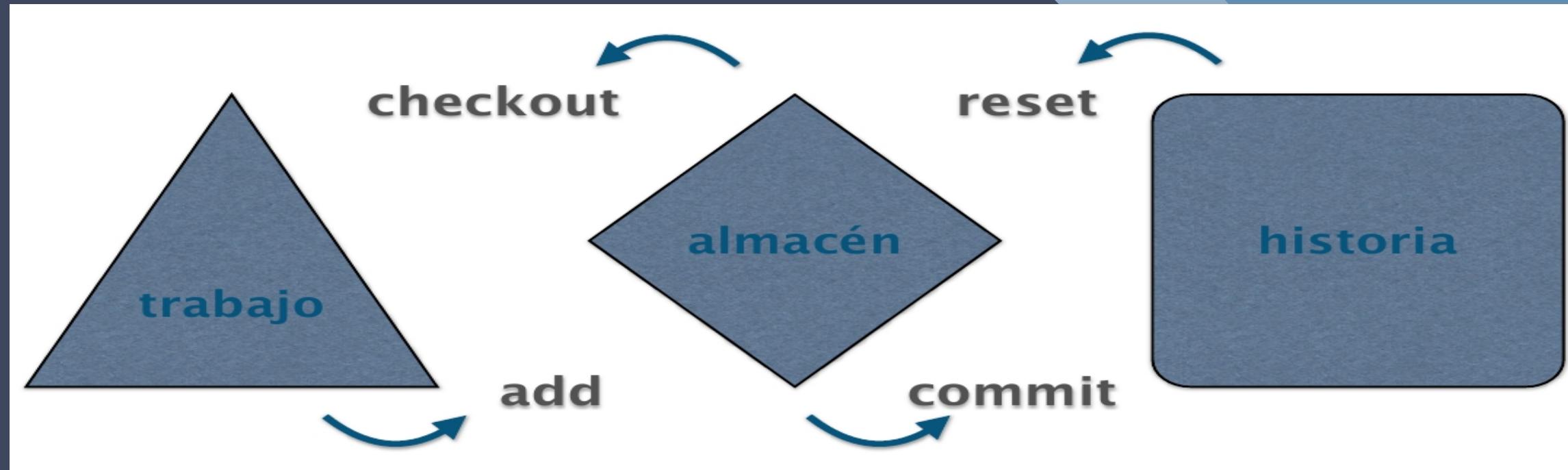
# Git

- El flujo de trabajo básico en Git es algo así:
  - **Modificas** una serie de archivos en tu directorio de trabajo.
  - **Preparas los archivos**, añadiéndolos a tu área de preparación.
  - **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.
  - 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).





# Git (Esquema mental)





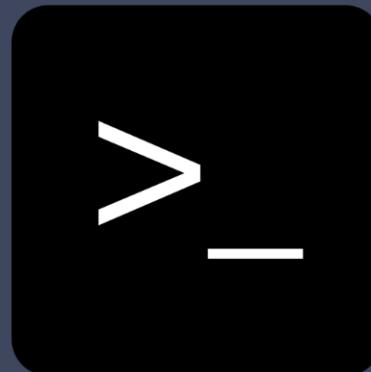
# Nuestro entorno de trabajo



git



GitHub





# Puesta en marcha

- Instalación: `sudo apt install git`
- Comprobación: `git --version`
- Ayuda: `git --help` o `git help nombreComando` o `git nombreComando --help`
- Configuración:
  - `git config --global user.name "John Doe"`
  - `git config --global user.email johndoe@example.com`
  - `git config --global core.editor emacs`
- Comprobar tu configuración: `git config --list`



# Guías de supervivencia

## Create a Repository

From scratch -- Create a new local repository

```
$ git init [project name]
```

Download from an existing repository

```
$ git clone my_url
```

## Observe your Repository

List new or modified files not yet committed

```
$ git status
```

Show the changes to files not yet staged

```
$ git diff
```

Show the changes to staged files

```
$ git diff --cached
```

Show all staged and unstaged file changes

```
$ git diff HEAD
```

Show the changes between two commit ids

```
$ git diff commit1 commit2
```

List the change dates and authors for a file

```
$ git blame [file]
```

Show the file changes for a commit id and/or file

```
$ git show [commit]:[file]
```

Show full change history

```
$ git log
```

Show change history for file/directory including diffs

```
$ git log -p [file/directory]
```

## Working with Branches

List all local branches

```
$ git branch
```

List all branches, local and remote

```
$ git branch -av
```

Switch to a branch, my\_branch, and update working directory

```
$ git checkout my_branch
```

Create a new branch called new\_branch

```
$ git branch new_branch
```

Delete the branch called my\_branch

```
$ git branch -d my_branch
```

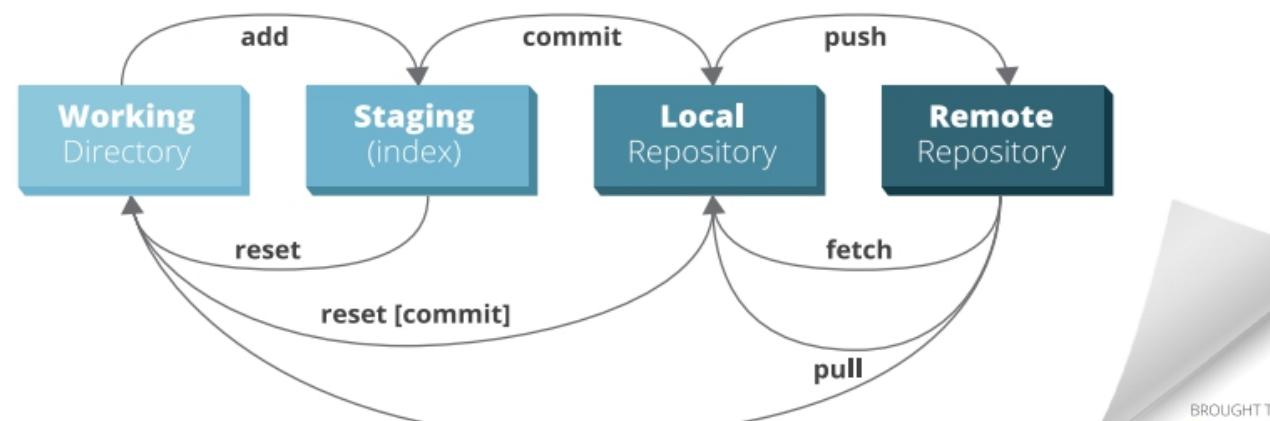
Merge branch\_a into branch\_b

```
$ git checkout branch_b
```

```
$ git merge branch_a
```

Tag the current commit

```
$ git tag my_tag
```



<https://ndpsoftware.com/git-cheatsheet.html>

## Make a change

Stages the file, ready for commit

```
$ git add [file]
```

Stage all changed files, ready for commit

```
$ git add .
```

Commit all staged files to versioned history

```
$ git commit -m "commit message"
```

Commit all your tracked files to versioned history

```
$ git commit -am "commit message"
```

Unstages file, keeping the file changes

```
$ git reset [file]
```

Revert everything to the last commit

```
$ git reset --hard
```

## Synchronize

Get the latest changes from origin (no merge)

```
$ git fetch
```

Fetch the latest changes from origin and merge

```
$ git pull
```

Fetch the latest changes from origin and rebase

```
$ git pull --rebase
```

Push local changes to the origin

```
$ git push
```

## Finally!

When in doubt, use git help

```
$ git command --help
```

Or visit <https://training.github.com/> for official GitHub training.



# Guías de supervivencia

Básico	Rescribiendo el historial
<code>git init &lt;carpeta&gt;</code>	Crea un repositorio vacío en <code>&lt;carpeta&gt;</code> . Sin argumentos permite inicializar la carpeta actual como un repositorio.
<code>git clone &lt;repositorio&gt;</code>	Clona un <code>&lt;repositorio&gt;</code> en nuestra máquina local. El <code>&lt;repositorio&gt;</code> puede ser local o remoto mediante <a href="#">HTTP</a> o <a href="#">SSH</a> .
<code>git config user.name &lt;autor&gt;</code>	Define al <code>&lt;autor&gt;</code> para todas las confirmaciones en el repositorio actual.
<code>git add &lt;carpeta archivo&gt;</code>	Agrega los cambios realizados en dicha <code>&lt;carpeta/archivo&gt;</code> para la siguiente confirmación. Acepta expresiones regulares.
<code>git commit -m "&lt;mensaje&gt;"</code>	Confirma los cambios en esa instancia, pero sin abrir o usar un editor de texto, usa el <code>&lt;mensaje&gt;</code> como mensaje de confirmación.
<code>git status</code>	Muestra la lista de archivos de los cambios montados, no montados y sin seguimiento.
<code>git log</code>	Muestra el historial de confirmaciones (formato por defecto). Para personalizar ver la sección de opciones adicionales.
<code>git diff</code>	Muestra los cambios no marcados entre el índice y la carpeta de trabajo.
Deshaciendo cambios	Ramas
<code>git revert &lt;confirmacion&gt;</code>	Crea una confirmación que deshace todos los cambios realizado en la <code>&lt;confirmacion&gt;</code> y lo aplica a la rama actual.
<code>git reset &lt;archivo&gt;</code>	Elimina el <code>&lt;archivo&gt;</code> del área de montaje, dejando el directorio de trabajo sin cambios por montar, sin sobreescribir ningún cambio.
<code>git clean -n</code>	Muestra los archivos que van a ser eliminados de la carpeta de trabajo. Para eliminar dichos archivos se usa <code>-f</code> en vez de <code>-n</code> .
Rescribiendo el historial	Repositorios remotos
<code>git commit --amend</code>	Reemplaza la última confirmación y/o cambios montados. Usarlo sin cambios montados edita el mensaje de la última confirmación.
<code>git rebase &lt;base&gt;</code>	Elimina el <code>&lt;archivo&gt;</code> del área de montaje, dejando el directorio de trabajo sin cambios sin sobreescribir ningún cambio.
<code>git reflog</code>	Muestra un log de los cambios en el <a href="#">HEAD</a> del repositorio local. Agrega <code>--relative-date</code> para mostrar la fecha o <code>--all</code> para todos.
Ramas	Repositorios remotos
<code>git branch &lt;rama&gt;</code>	Crea una nueva <code>&lt;rama&gt;</code> . Sin argumentos muestra la lista de ramas del repositorio.
<code>git checkout -b &lt;rama&gt;</code>	Crea una nueva <code>&lt;rama&gt;</code> en base a la rama actual. Si quitamos <code>-b</code> permite posicionarse en la <code>&lt;rama&gt;</code> .
<code>git merge &lt;rama&gt;</code>	Mezcla la <code>&lt;rama&gt;</code> con la rama actual.
Repositorios remotos	
<code>git remote add &lt;nombre&gt; &lt;enlace&gt;</code>	Crea una nueva conexión a un repositorio remoto. El <code>&lt;nombre&gt;</code> es el alias del <code>&lt;enlace&gt;</code> al cual se comunica.
<code>git fetch &lt;remoto&gt; &lt;rama&gt;</code>	Obtiene la <code>&lt;rama&gt;</code> del repositorio <code>&lt;remoto&gt;</code> . Si no se especifica la rama trae toda todas las ramas.
<code>git pull &lt;remoto&gt;</code>	Obtiene una copia del repositorio <code>&lt;remoto&gt;</code> y la mezcla al instante con el repositorio local.
<code>git push &lt;remoto&gt; &lt;rama&gt;</code>	Envía la <code>&lt;rama&gt;</code> al repositorio <code>&lt;remoto&gt;</code> actualizándola si existe y sino creandola.



# Guías de supervivencia

git config	git diff
<code>git config --global user.name &lt;autor&gt;</code> Define el nombre del <code>&lt;autor&gt;</code> de todas las confirmaciones para el usuario actual.	<code>git diff HEAD</code> Muestra diferencias de la ult. confirmación y la carpeta trabajo. <code>git diff --cached</code> Muestra diferencias de la ult. confirmación y los cambios marcados.
<code>git config --global user.email &lt;correo&gt;</code> Define el <code>&lt;correo&gt;</code> del autor de todas las confirmaciones para el usuario actual.	
<code>git config --global alias.&lt;alias&gt; &lt;comando&gt;</code> Crea un atajo de <code>&lt;comando&gt;</code> en Git que se usara mediante el <code>&lt;alias&gt;</code> . Ejm. alias.glog log --graph equivalente. glog -> git log --graph.	
<code>git config --system core.editor &lt;editor&gt;</code> Define el <code>&lt;editor&gt;</code> usado por los comandos para todos los usuarios. Por ejemplo <code>&lt;editor&gt;</code> puede ser vim.	
<code>git config --global --edit</code> Abre el archivo de configuración global para que pueda ser editado manualmente. Usa el editor por defecto de la maquina. Ejm. nano.	
git log	git reset
<code>git log --&lt;limite&gt;</code> Limita el número de logs usando el <code>&lt;limite&gt;</code> . Ejemplo: git log -5 muestra los últimos 5 logs.	<code>git reset</code> Reinicia el área de marcado para coincidir con la última confirmación, pero deja la carpeta de trabajo sin cambios.
<code>git log --online</code> Condensa cada una de las confirmaciones a una sola linea.	<code>git reset --hard</code> Reinicia el área de marcado y carpeta de trabajo para coincidir con la ultima confirmación, sobreescribiendo todos los cambios.
<code>git log -p</code> Muestra todos los cambios de cada confirmación.	<code>git reset &lt;confirmacion&gt;</code> Mueve la rama actual hacia la <code>&lt;confirmacion&gt;</code> reiniciando el área de marcado, pero deja la carpeta de trabajo.
<code>git log --stat</code> Incluye que archivos se modificaron y el número relativo de líneas que fueron añadidas o borradas de cada uno de ellos.	<code>git reset --hard &lt;confirmacion&gt;</code> Lo mismo que <code>git reset --hard</code> pero todos los cambios no confirmados y todas las confirmaciones despues de la <code>&lt;confirmacion&gt;</code> .
<code>git log --author="&lt;autor&gt;"</code> Busca las confirmaciones por un <code>&lt;autor&gt;</code> en particular.	
<code>git log --grep="&lt;patron&gt;"</code> Busca confirmaciones cuyo mensaje de confirmación coincide con el <code>&lt;patron&gt;</code> asignado.	git rebase
<code>git log &lt;desde&gt;..&lt;hasta&gt;</code> Muestra las confirmaciones entre <code>&lt;desde&gt;</code> y <code>&lt;hasta&gt;</code> . Se puede usar hash de confirmación, rama, <code>HEAD</code> o cualquier punto de revisión.	<code>git rebase -i &lt;base&gt;</code> Un rebase interactivo que abre un editor para ingresar los comandos para cada confirmación que sera transferida a la nueva <code>&lt;base&gt;</code> .
<code>git log --&lt;archivo&gt;</code> Muestra solo las confirmaciones del <code>&lt;archivo&gt;</code> especificado.	git pull
<code>git log --graph --decorate</code> El <code>--graph</code> muestra un gráfico en texto al lado izquierdo de los mensajes de confirmación y <code>--decorate</code> el nombre de la rama o tag.	<code>git pull --rebase &lt;remoto&gt;</code> Obtiene una copia de <code>&lt;remoto&gt;</code> de la rama actual pero en vez de realizar una mezcla realiza un rebase de las ramas.
git push	git push
	<code>git push --force &lt;remoto&gt;</code> Fuerza a Git a enviar la rama al <code>&lt;remoto&gt;</code> ignorando si hay cambios e incompatibilidades. Solo usar cuando estas seguro de lo que haces.
	<code>git push --all &lt;remoto&gt;</code> Envía todas las ramas del repositorio local al repositorio <code>&lt;remoto&gt;</code> .
	<code>git push --tags &lt;remoto&gt;</code> Las etiquetas no son enviadas automaticamente por lo que es necesario usar <code>--tags</code> para enviarlas al repositorio <code>&lt;remoto&gt;</code> .



# Ejercicio

- **Instalar Git en tu maquina Linux y configurarlo teniendo en cuenta:**
  - Tu nombre y tu correo electrónico
  - Que el editor predefinido sea vs code
  - Activar el coloreado de la salida
  - Mostrar el estado original en los conflictos
  - Mostrar la configuración final
- **Abrir una cuenta en GitHub**
- **Instalar GitKraken**
- **Instalar VS Code o tu editor favorito y configurarlo para que se integre con Git/GitHub**



# GIT Básico

Manejo básico de Git



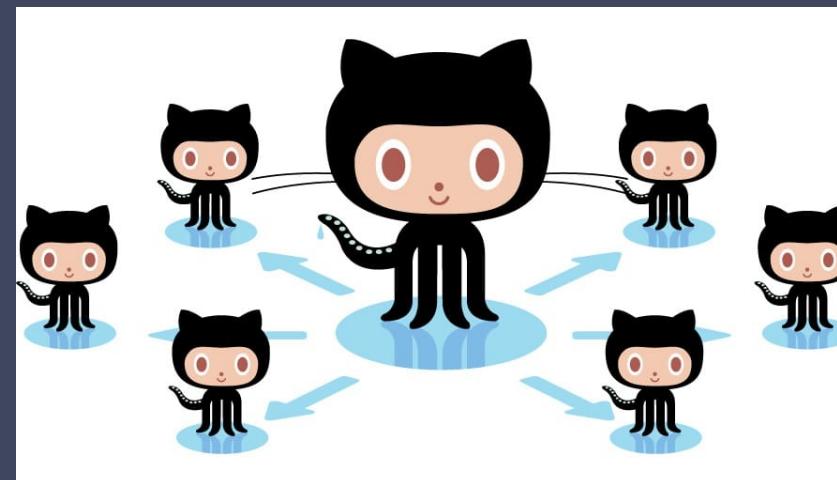
# Creación de repositorios

- **git init <nombre-repositorio>** crea un repositorio nuevo con el nombre <nombre-repositorio>.
- Este comando crea una nueva carpeta con el nombre del repositorio, que a su vez contiene otra carpeta oculta llamada .git que contiene la base de datos donde se registran los cambios en el repositorio.



# Clonación de repositorios

- **git clone <url-repositorio>** crea una copia local del repositorio ubicado en la dirección <url-repositorio>.
- A partir de que se hace la copia, los dos repositorios, el original y la copia, son independientes, es decir, cualquier cambio en uno de ellos no se verá reflejado en el otro.





# Ejercicio

- Crea un repositorio llamado git-local. Visualiza la estructuras de carpetas ocultas
- Clona el repositorio <https://github.com/joseluisgs/bootstrap-sass-init>
- Repite con VS Code y GitKraken indicando los pasos que has seguido



# Estado del repositorio

- **git status.**
- Muestra el estado de los cambios en el repositorio desde la última versión guardada. En particular, muestra los cheros con cambios en el directorio de trabajo que no se han añadido a la zona de intercambio temporal y los cheros en la zona de intercambio temporal que no se han añadido al repositorio.





# Añadir ficheros a la zona de intercambio temporal

- **git add <fichero>** añade/actualiza los cambios en el chero <fichero> del directorio de trabajo a la zona de intercambio temporal.
- **git add <carpeta>** añade/actualiza los cambios en todos los cheros de la carpeta <carpeta> del directorio de trabajo a la zona de intercambio temporal.
- **git add .** añade/actualiza todos los cambios de todos losaún en la zona de intercambio temporal.





# Eliminar ficheros

- **git rm <file>**. Elimina el fichero de la zona de trabajo
- **git rm --cached <file>**. Elimina el fichero de la zona de intercambio temporal



# Mover el fichero

- **git mv <file> <new\_file>**. Renombra el fichero o lo mueve de un path a otro. Cambia el nombre se entiende como un movimiento



# Descartar cambios

- **git restore <file>**. Elimina los cambios realizados en el fichero en tu directorio de trabajo
- **git restore --staged <file>**. Elimina los cambios realizados en el fichero de la zona de almacenamiento
- **OJO: Nos puede servir para recuperar ficheros eliminados por error**



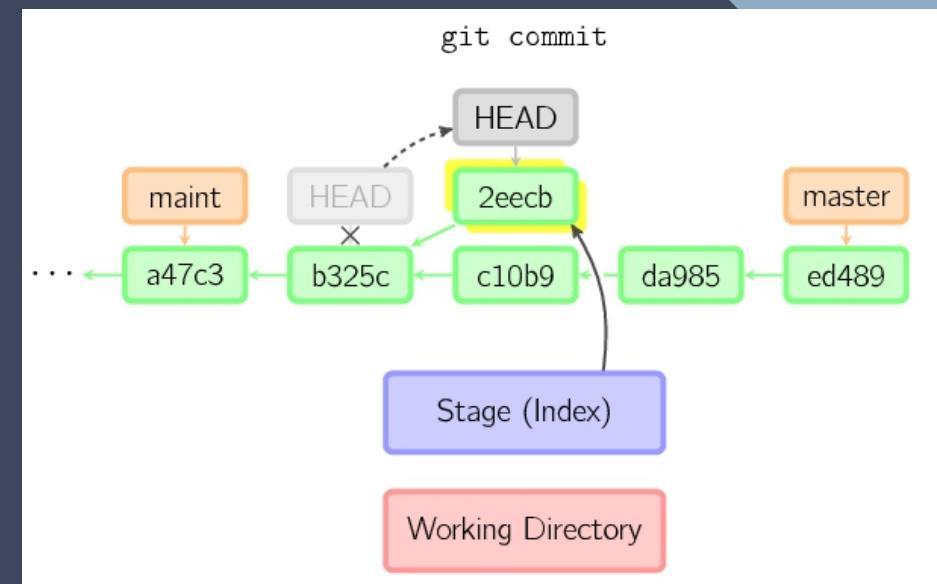
# Añadir cambios al repositorio

- **git commit -m "mensaje"** confirma todos los cambios de la zona de intercambio temporal añadiéndolos al repositorio y creando una nueva versión del proyecto. "mensaje" es un breve mensaje describiendo los cambios realizados que se asociará a la nueva versión del proyecto.
- **git commit --amend -m "mensaje"** cambia el mensaje del último commit por el nuevo mensaje "mensaje".

Cada commit tiene asociado un código hash de 40 caracteres hexadecimales que lo identifica de manera única. Hay dos formas de referirse a un commit:

**Nombre absoluto:** Se utiliza su código hash (basta indicar los 4 o 5 primeros dígitos).

**Nombre relativo:** Se utiliza la palabra HEAD para referirse siempre al último commit. Para referirse al penúltimo commit se utiliza HEAD~1, al antepenúltimo HEAD~2, etc.





# Historial de versiones de un repositorio

- **git log** muestra el historial de commits de un repositorio ordenado cronológicamente. Para cada commit muestra su código hash, el autor, la fecha, la hora y el mensaje asociado.
- Este comando es muy versátil y muestra la historia del repositorio en distintos formatos dependiendo de los parámetros que se le den. Los más comunes son:
  - **--oneline**: muestra cada commit en una línea produciendo una salida más compacta.
  - **--graph**: muestra la historia en forma de grafo.
  - **--pretty**: muestra la salida en formato más legible
  - **-n X**: muestra los X últimos commits
  - **--stat**: muestra información sobre los archivos que se han cambiado y las líneas afectadas
  - **--author =”XX”**: muestra los commits que ha realizado XX



# Mostrar los datos de un commit

- **git show** muestra el usuario, el día, la hora y el mensaje del último commit, así como las diferencias con el anterior.
- **git show <commit>** muestra el usuario, el día, la hora y el mensaje del commit indicado, así como las diferencias con el anterior.
- **git show HEAD~X** muestra el commit X anterior al último.



# Mostrar el historial de cambios de un fichero

- **git annotate <fichero>** o **git blame <fichero>** muestra el contenido de un fichero anotando cada línea con información del commit en el que se introdujo. Cada línea de la salida contiene los 8 primeros dígitos del código hash del commit correspondiente al cambio, el autor de los cambio, la fecha, el número de línea del fichero y el contenido de la línea.



# Ejercicio

- **En tu repositorio git-local, realiza.**

- Crea un fichero llamado git.html con un contenido que diga Hola Git
- Visualiza el estado del repositorio
- Añadelo a la zona de intercambio o almacenamiento
- Modificalo y añade una nueva línea con tu nombre.
- Visualiza el estado
- Descarta los cambios en el fichero en la zona de trabajo
- Vuelve a añadir otra línea con tus apellidos
- Visualiza el estado del repositorio.
- Confirma en el repositorio
- Actualiza a la zona de intercambio
- Modifica el fichero y añade el grupo.
- Visualiza el estado del repositorio
- Actaliza el fichero en la zona de intercambios
- Descarta los cambios en la zona de trabajo.
- Confirma los cambios en el repositorio
- Muetra los confirmaciones realizadas en directorio
- Muetra los cambios realizados en el fichero de dos maneras distintas
- **Repite usando VS Code y GitKraken**





# Mostrar las diferencias entre versiones

- **git diff** muestra las diferencias entre el directorio de trabajo y la zona de intercambio temporal.
- **git diff --cached** muestra las diferencias entre la zona de intercambio temporal y el último commit.
- **git diff HEAD** muestra la diferencia entre el directorio de trabajo y el último commit.





# Eliminar los cambios del directorio de trabajo o volver a la versión anterior

- `git checkout <commit> -- <file>` actualiza el fichero `<file>` a la versión correspondiente al commit `<commit>`.
- Suele utilizarse para eliminar los cambios en un fichero que no han sido guardados aún en la zona de intercambio temporal, mediante el comando actualiza el fichero a la ultima versión
- Es importante entender que `git checkout -- [archivo]` **es un comando peligroso**. Cualquier cambio que le hayas hecho a ese archivo desaparecerá - acabas de sobreescribirlo con otro archivo. Nunca utilices este comando a menos que estés absolutamente seguro de que ya no quieres el archivo
- Este comando extrae el contenido del repositorio y lo coloca en su árbol de trabajo. También puede tener otros efectos, dependiendo de cómo se invocó el comando. Por ejemplo, también puede cambiar la rama en la que está trabajando actualmente. Este comando no hace ningún cambio en el historial.



# Eliminar cambios de la zona de intercambio temporal

- **git reset <fichero>** elimina los cambios del fichero <fichero> de la zona de intercambio temporal, pero preserva los cambios en el directorio de trabajo.
- Para eliminar por completo los cambios de un fichero que han sido guardados en la zona de intercambio temporal hay que aplicar este comando y después **git checkout HEAD -- <fichero>**.



# Eliminar cambios de un commit

**git reset --hard <commit>** elimina todos los cambios desde el commit <commit> y actualiza el HEAD este commit.

- ¡Ojo! Usar con cuidado este comando pues los cambios posteriores al commit indicado se pierden por completo.
- Suele usarse para eliminar todos los cambios en el directorio de trabajo desde el último commit mediante el comando `git reset --hard HEAD`.
- **git reset <commit>** actualiza el HEAD al commit <commit>, es decir, elimina todos los commits posteriores a este commit, pero no elimina los cambios del directorio de trabajo.
- Este comando es un poco más complicado. En realidad, hace un par de cosas diferentes dependiendo de cómo se invoca. Modifica el índice (el llamado "área de preparación"). O bien, los cambios a los que está comprometido un encabezado de rama están apuntando actualmente. Este comando puede alterar el historial existente (cambiando la confirmación que hace referencia una rama).



# Eliminar cambios de un commit

- **git revert <file>** Este comando crea una nueva confirmación que deshace los cambios de una confirmación anterior. Este comando agrega un nuevo historial al proyecto (no modifica el historial existente).



# Ignorar ficheros

- `.gitignore` pondremos la lista de ficheros, directorios y patrones que no queremos realizar el control de versiones.



# Tags

- 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).
- **git tag**
- **git tag -a v1.4 -m 'my version 1.4'**, crea una etiqueta anotada
- **git show v1.4**, muestra una etiqueta



# Alias

Git no deduce automáticamente tu comando si lo tecleas parcialmente. Si no quieres teclear el nombre completo de cada comando de Git, puedes establecer fácilmente un alias para cada comando mediante git config. Aquí tienes algunos ejemplos que te pueden interesar:

- `git config --global alias.co checkout`
  - `git config --global alias.br branch`
  - `git config --global alias.ci commit`
  - `git config --global alias.st status`
- Esto significa que, por ejemplo, en lugar de teclear `git commit`, solo necesitas teclear `git ci`. A medida que uses Git, probablemente también utilizarás otros comandos con frecuencia; no dudes en crear nuevos alias para ellos.
- Esta técnica también puede resultar útil para crear comandos que en tu opinión deberían existir. Por ejemplo, para corregir el problema de usabilidad que encontraste al quitar del área de preparación un archivo, puedes añadir tu propio alias a Git:
- `git config --global alias.unstage 'reset HEAD --'`
  - Esto hace que los dos comandos siguientes sean equivalentes:
  - `git unstage fileA`
  - `git reset HEAD fileA`
- Esto parece un poco más claro. También es frecuente añadir un comando `last`, de este modo:
- `git config --global alias.last 'log -1 HEAD'`



# Trabajando con los repositorios

- Las herramientas más utilizadas para deshacer acciones son **git restore**, **git checkout**, **git revert** y **git reset**.
- Estos son algunos puntos clave que debes recordar:
  - Una vez confirmados los cambios, por lo general son permanentes. Así que aunque no es una regla 100% aplicable sigue esta regla.
  - Si hemos hecho un **add**, podemos deshacerlo con un **restore**
  - Utiliza **git checkout** para desplazarte por el historial de confirmaciones y consultarlos, sin perder el árbol de la historia
  - Git **revert** es la mejor herramienta para deshacer cambios públicos compartidos. Puedes eliminar parte de la historia.
  - El comando **git reset** es más adecuado para deshacer cambios privados locales. Cuidado entre las opciones hard y soft.
  - Además de los comandos principales para deshacer acciones, echa vistazo a otras utilidades de Git: git log para encontrar confirmaciones perdidas, git clean para deshacer cambios no confirmados y git add para modificar el índice del entorno de ensayo.
- Cada uno de estos comandos tiene su propia documentación detallada. Para obtener más información sobre un comando concreto de los que hemos mencionado aquí, visita los enlaces correspondientes.



# Trabajando con los repositorios

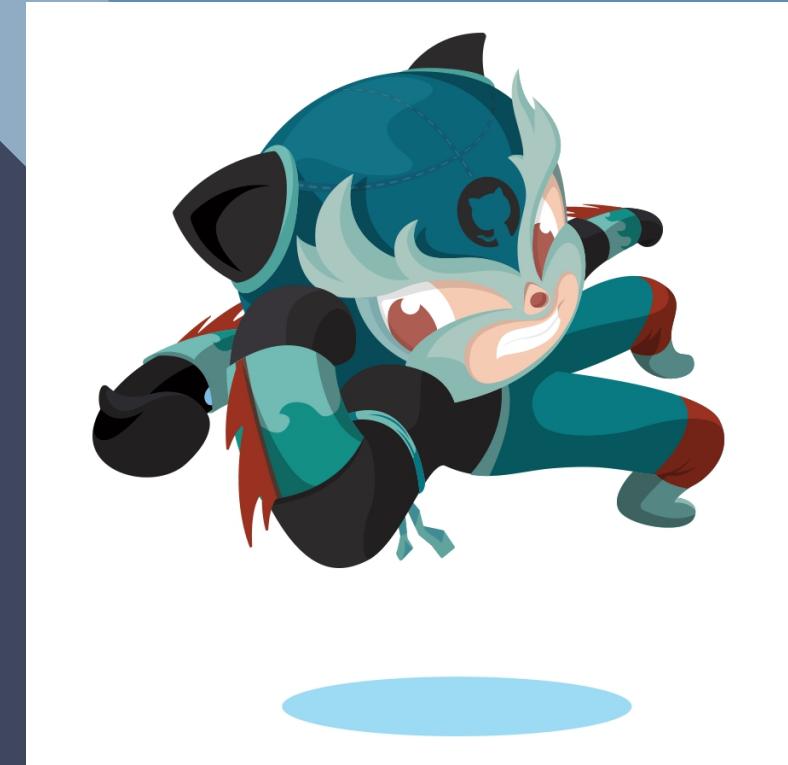




# Ejercicio

- **Sobre tu repositorio de clase**

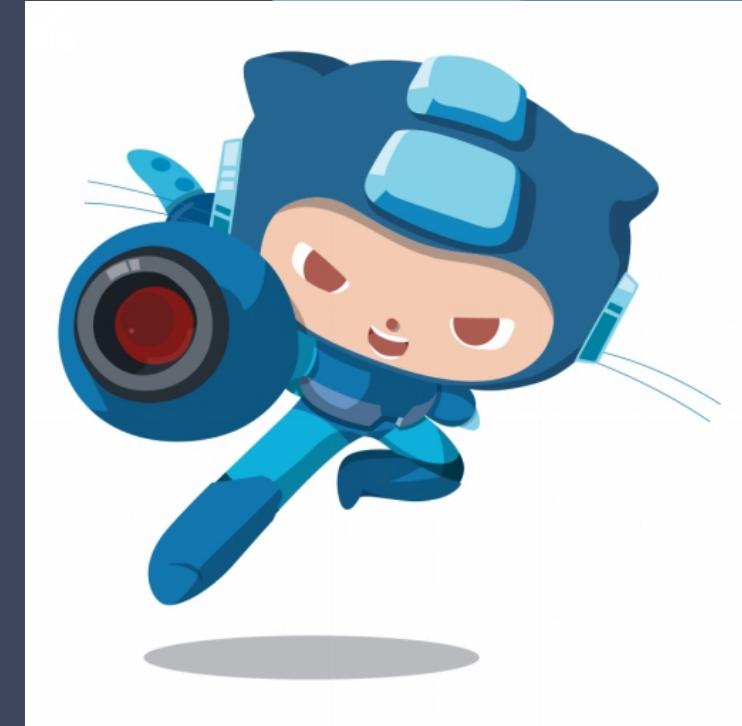
- Creo un nuevo fichero llamado otro.html y añade el nombre del instituto.
- Añadelo a tu zona de intercambio.
- Bórralo con la consola o desde el explorador
- Recupéralo.
- Cambiale el nombre a otro2.html
- Confirma los cambios en el repositorio
- Modifica los dos ficheros ya añade el nombre del módulo que estas cursando a ellos.
- Confirma esos cambios en el repositorio
- a tu fichero git.html elimina la primera línea líneas, a tu fichero otro.htm
- añádele tu dirección de email
- Visualiza las diferencias
- Confirma los cambios el el repositorio
- **Repite con VS Code y GitKraken**





# Ejercicio

- **Crea un nuevo repositorio**
  - Crea un fichero llamado prueba.html y añade una línea con tu nombre
  - Añádelo a tu área de intercambio.
  - Descarta los cambios
  - Vuelve a añadir tu nombre y tus apellidos
  - Confirma el repositorio
  - Añade el módulo que estas estudiando
  - Visualiza las diferencias
  - Revierte el fichero al estado anterior a estos cambios usando el almacenado en el repositorio (último commit)
  - Vuelve a introducir en el fichero los nombres del instituto
  - Actualiza el fichero en la área de intercambio.
  - Vuelve al estado anterior
  - Añade al fichero tu dirección
  - Confirma los cambios
  - Etiqueta la nueva versión
  - Vuelve al estado del primer commit.
  - **Repite con VS Code y GitKraken**



¡¡La he cagado!!





# ¡¡La he cagado!!

## Si quieres arreglar el último commit (y no has hecho push)

A veces no quieres tirar atrás el último commit que has hecho si no que simplemente quieres arreglarlo. Aquí hay dos opciones:

Sólo quieres arreglar el mensaje que has usado para el último commit:

```
git commit --amend -m "Este es el mensaje correcto"
```

Quieres añadir más cambios al último commit:

```
# Añade los archivos con modificaciones que quieras añadir al commit anterior
```

```
git add src/archivo-con-cambios.js
```

```
# Vuelve a hacer el commit con el parámetro amend
```

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

Ya sea que sólo quieras cambiar el mensaje de commit o que además quieras añadir modificaciones en el último commit, lo importante es que esto NO va a crear un nuevo commit si no que va a solucionar el anterior.

**Importante:** El parámetro de --amend es muy útil pero sólo funciona con el último commit y siempre y cuando NO esté publicado. Si ya has hecho push de ese commit, esto no va a funcionar. Deberías hacer un git revert en su lugar



# ¡¡La he cagado!!

## Deshacer el último commit (no publicado)

- A veces queremos tirar para atrás el último commit que hemos hecho porque hemos añadido más archivos de la cuenta, queremos hacer commit de otra cosa o, simplemente, porque ahora no tocaba. Si todavía no has hecho push de tus cambios tienes dos formas de hacer esto que dependerá de si quieres, o no, mantener los cambios del commit.
- Si quieres mantener los cambios:
  - **git reset --soft HEAD~1**
  - Con el comando reset hacemos que la rama actual retroceda a la revisión que le indicamos. En este caso le decimos HEAD~1 que significa: queremos volver a la versión inmediatamente anterior a la que estamos ahora. El parámetro --soft es el que va a hacer que los cambios que habíamos hecho en el commit, en lugar de eliminarlos, nos los mantenga como cambios locales en nuestro repositorio. Es decir nos trae del repositorio a la área de intercambio sin tocar el directorio de trabajo
- Si NO quieres mantener los cambios:
  - **git reset --hard HEAD~1**
  - Es simplemente el mismo comando pero cambiamos --soft por --hard. Esto eliminará los cambios de los que habíamos hecho commit anteriormente. ¡⚠ Asegúrate que eso es lo que quieras antes de hacerlo! Machaca todo en el directorio de trabajo.



# ¡¡La he cagado!!

## Deshacer un commit (ya publicado)

- A veces es demasiado tarde y no sólo has hecho commit, si no que además has publicado los cambios. Peeero, todavía hay esperanza. Puedes hacer un revert de tus cambios indicando el commit que quieras deshacer.

```
git revert 74a1092
```

Esto creará un nuevo commit que deshará todos los cambios de ese commit en concreto. Es una forma interesante de mantener en el historial de Git ese cambio (quién sabe si lo puedes necesitar más adelante).

A veces el problema es que puedes haber añadido otros commits posteriormente que entren en conflicto con ese pero eso es una historia que igual abordamos en futuro.

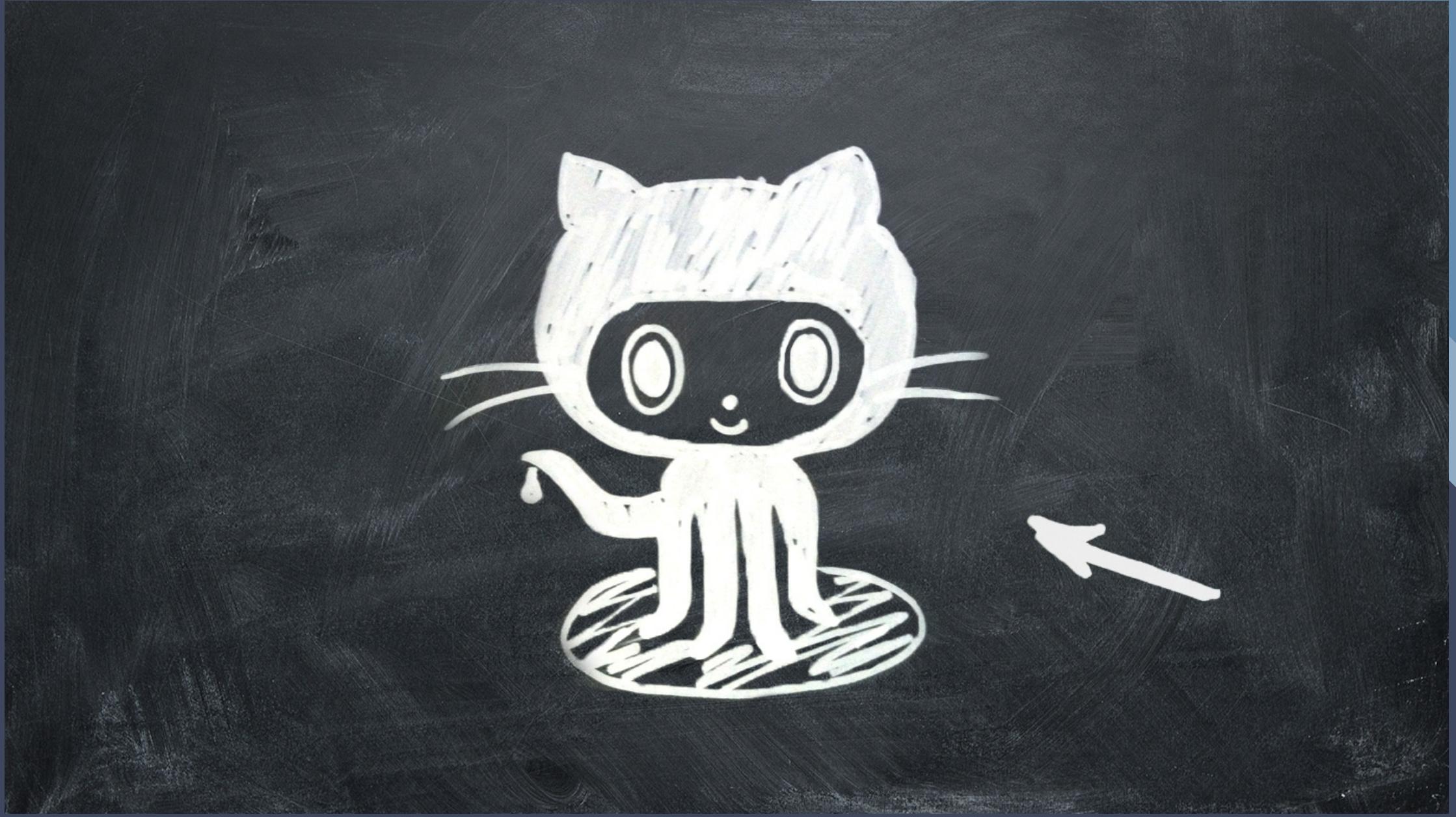


# ¡¡La he cagado!!

- Un consejo antes de cargarte los cambios y todo, este alias ayuda, pero debes entender lo que hace. Repasa las transparencias anteriores

```
# Añade un alias a .gitconfig con este comando
git config --global alias.undo 'reset --soft HEAD^'

# Ahora lo tienes disponible con el alias undo
git undo
```



# GIT Avanzado

Trabajando con ramas y flujos de trabajo avanzados



# Trabajando con ramas

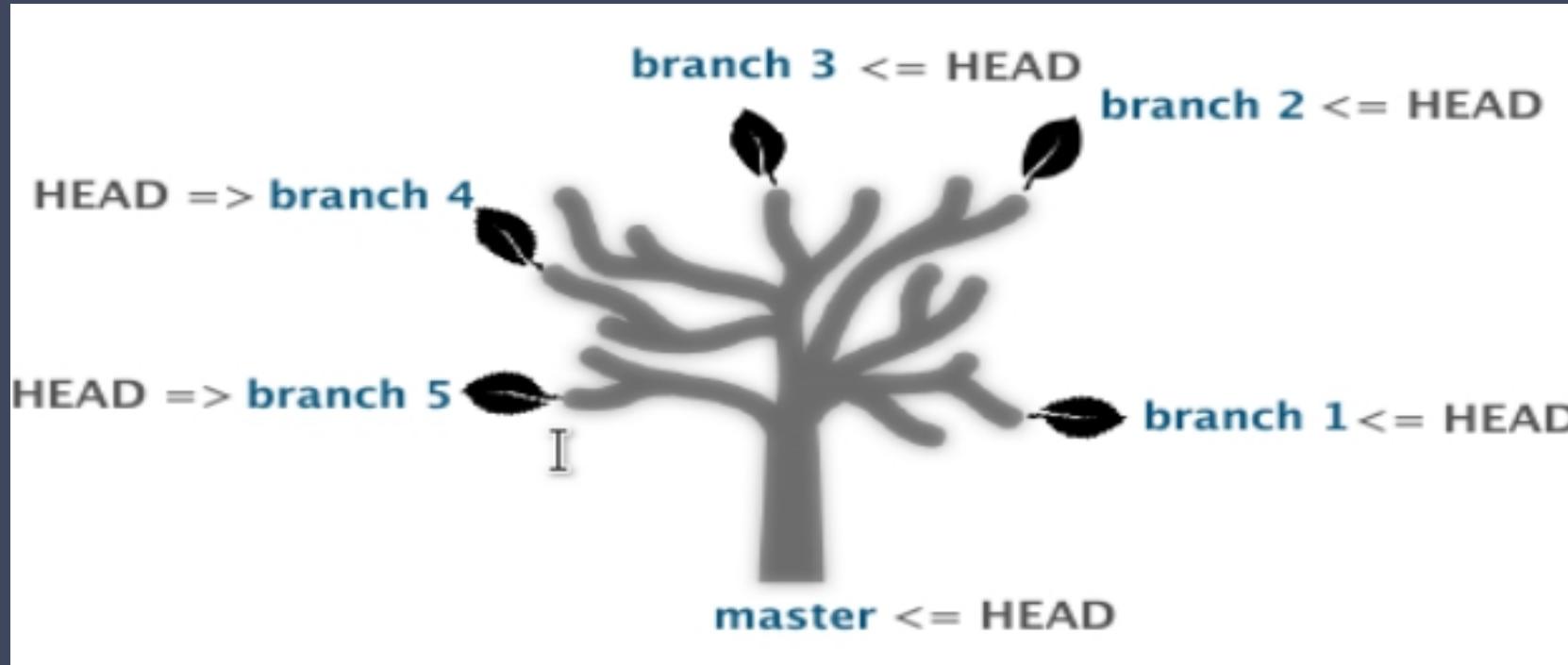
- Aquí empieza una de las partes más potentes que tiene Git y es el uso de ramas para el trabajo compartido y o colaborativo.
- Inicialmente cualquier repositorio tiene una única rama llamada **master** donde se van sucediendo todos los commits de manera lineal.
- Una de las características más útiles de Git es que permite la creación de ramas para trabajar en distintas versiones de un proyecto a la vez. Es muy importante que al comenzar un proyecto creamos una rama, con el objetivo de dejar la rama principal lo más estable posible.
- Al crear una nueva rama siempre se clona la rama en la que estamos actualmente.
- Esto es muy útil si, por ejemplo, se quieren añadir nuevas funcionalidades al proyecto sin que interfieran con lo desarrollado hasta ahora. Cuando se termina el desarrollo de las nuevas funcionalidades las ramas se pueden fusionar para incorporar los cambios al proyecto principal.
- De esta manera siempre podemos asegurar que nuestro proyecto está en perfectas condiciones.





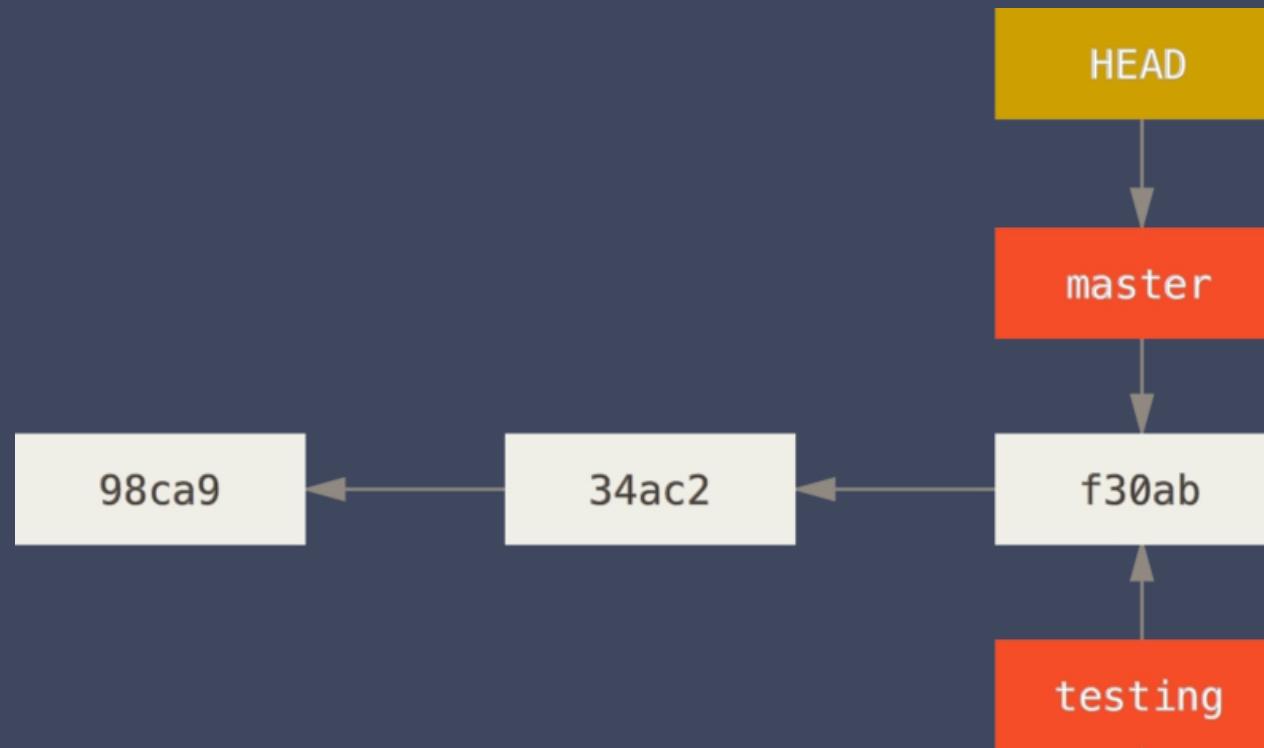
# Trabajando con ramas

- Por cada commit Git guarda un HEAD independiente por rama, pudiendo volver a cualquier momento histórico de una rama, teniendo control para funcionar, restaurar, etc.
- Es importante tener en cuenta que se puede generar una maraña en el flujo de trabajo, por lo que es conveniente ser ordenado y seguir unos patrones de trabajo común dentro del equipo.



# Crear ramas

- **git branch <rama>** crea una nueva rama con el nombre <rama> en el repositorio a partir del último commit, es decir, donde apunte HEAD.
- Al crear una rama a partir de un commit, el flujo de commits se bifurca en dos de manera que se pueden desarrollar dos versiones del proyecto en paralelo.





# Listado de ramas

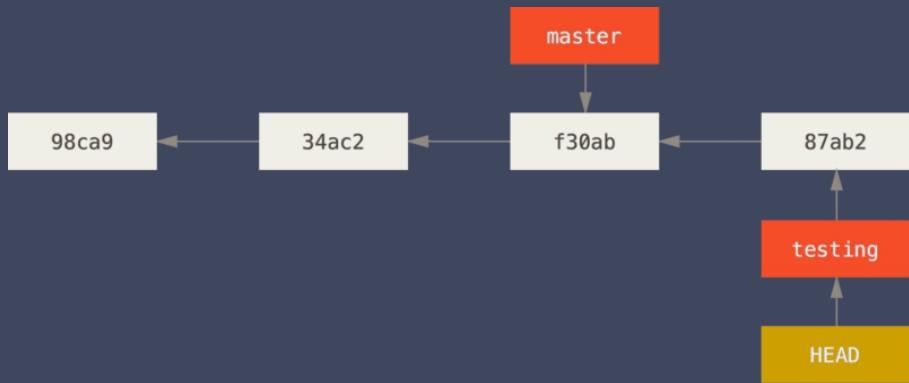
- **git branch** muestra las ramas activas de un repositorio indicando con \* la rama activa en ese momento.
- **git log --graph --oneline** muestra la historia del repositorio en forma de grafo (--graph) incluyendo todas las ramas (--all).





# Cambio de ramas

- **git checkout <rama>** actualiza los archivos del directorio de trabajo a la última versión del repositorio correspondiente a la rama <rama>, y la activa, es decir, HEAD pasa a apuntar al último commit de esta rama.
- **git checkout -b <rama>** crea una nueva rama con el nombre <rama> y la activa, es decir, HEAD pasa a apuntar al último commit de esta rama. Este comando es equivalente aplicar los comandos git branch <rama> y después y después git checkout <rama>.



**Saltar entre ramas cambia archivos en tu directorio de trabajo.**

Es importante destacar que cuando saltas a una rama en Git, los archivos de tu directorio de trabajo cambian. Si saltas a una rama antigua, tu directorio de trabajo retrocederá para verse como lo hacía la última vez que confirmaste un cambio en dicha rama. Si Git no puede hacer el cambio limpiamente, no te dejará saltar.



# Eliminar ramas

- **git branch -d <rama>** elimina la rama de nombre <rama> siempre y cuando haya sido fusionada previamente.
- **git branch -D <rama>** elimina la rama de nombre <rama> incluso si no ha sido fusionada. Si la rama no ha sido fusionada previamente se perderán todos los cambios de esa rama.
- **OJO, desde la rama activa nos traemos los cambios de la rama que indiquemos y es esa la que desaparece. ¡CUIDADO! que podeis perder una rama que no queríais, por ejemplo master. Comprobar siempre la rama activa con git branch**

# Ejercicio

- **Crea un nuevo repositorio**

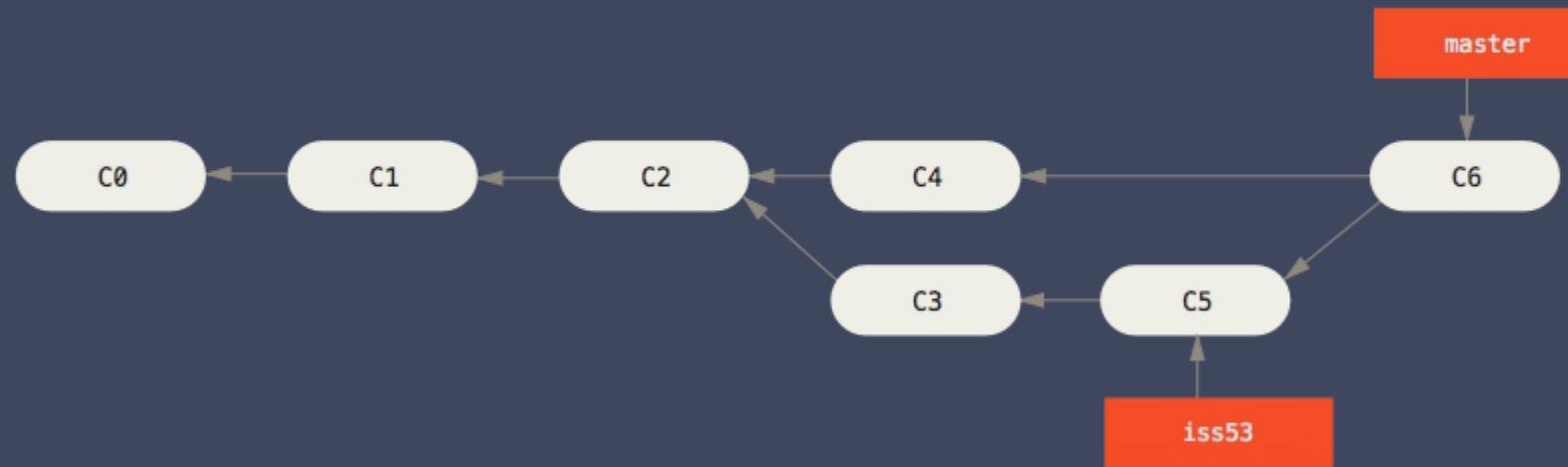
- Sobre este repositorio crea y confirma un fichero llamado nombre.txt con tu nombre
- Confirma los cambios
- Crea una nueva rama llamada test
- En la rama test, modifica el fichero nombre.txt y
- añade tus apellidos.
- Crea un nuevo fichero llamado curso.txt y
- añade el curso que estas.
- Visualiza las ramas
- Vuelve a la rama primera y crea un fichero
- llamado modulo.txt
- **Repite con VS Code y GitKraken**





# Fusionar ramas

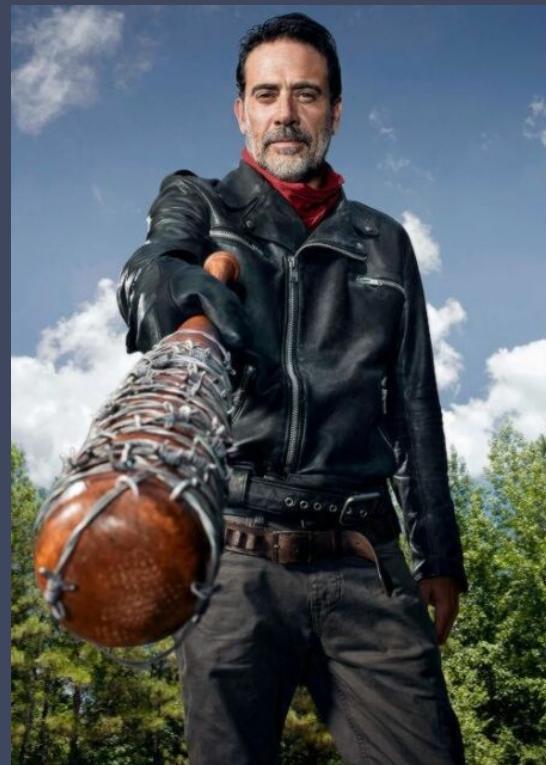
- **git merge <rama>** integra los cambios de la rama <rama> en la rama actual a la que apunta HEAD.
- Es decir, imaginemos que queremos fusionar los cambios de la rama testing en master.
  - Nos vamos a master: **git checkout master**, esto nos hace que el HEAD activo sea el de master,
  - **git merge testing**, con esto fusionamos los cambios





# ¡¡La cagaste!!

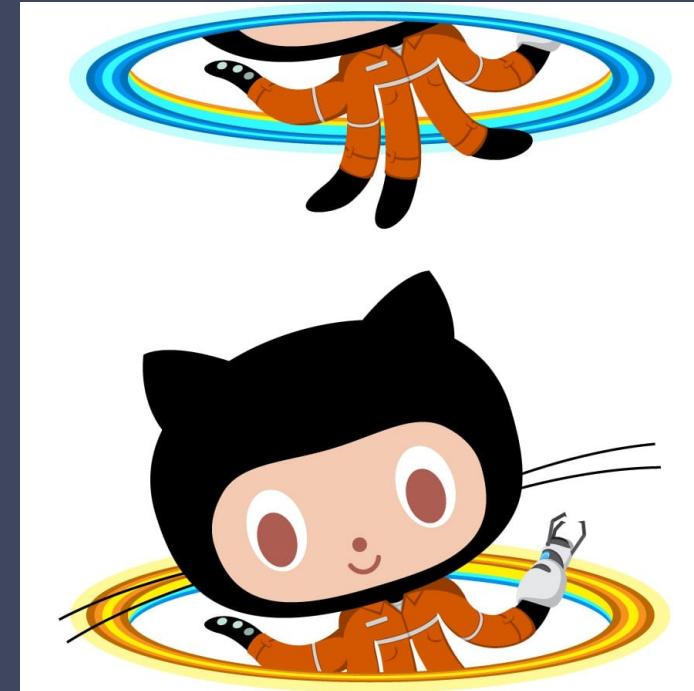
- Has hecho una fusión y no querías hacerlo...
- Fácil, tienes que localizar el commit a donde quieras ir
  - **git reset --hard <commit>**
  - Esto ya lo habíamos visto
  - Repasa... y Presta atención
  - Las ramas son importantes





# Ejercicio

- **Sobre el repositorio anterior**
  - Fusiona todas las ramas para que todos los ficheros estén en master
  - Borra la rama test
  - Repite con VS Code y GitKraken





# Resolución de conflictos

- Para fusionar dos ramas es necesario que no haya conflictos entre los cambios realizados a las dos versiones del proyecto.
- Si en ambas versiones se han hecho cambios sobre la misma parte de un fichero, entonces se produce un conflicto y es necesario resolverlo antes depoder fusionar las ramas.
- La resolución debe hacerse manualmente observando los cambios que interfieren y decidiendo cuales deben prevalecer, aunque existen herramientas como KDiff3 o meld que facilitan el proceso.
- Todo aquello que sea conflictivo y no se haya podido resolver, se marca como "sin fusionar" (unmerged). Git añade a los archivos conflictivos **unos marcadores especiales de resolución de conflictos** que te guiarán cuando abras manualmente los archivos implicados y los edites para corregirlos.
- Si en lugar de resolver directamente prefieres utilizar una herramienta gráfica, puedes usar el comando **git mergetool**, el cual arrancará la correspondiente herramienta de visualización y te permitirá ir resolviendo conflictos con ella



# Resolución de conflictos

- Imagina que en dos ramas (master y test) hemos trabajado el mismo fichero index.html. Tendríamos
  - git status
    - On branch master
    - You have unmerged paths.
    - (fix conflicts and run "git commit")
  - Unmerged paths:
    - (use "git add <file>..." to mark resolution)
    - both modified: index.html
  - no changes added to commit (use "git add" and/or "git commit -a")



# Resolución de conflictos

- Al abrir el fichero conflictivo, git no los ha marcado con unos caracteres especiales. Abrimos el fichero index.html y observamos

```
<<<<< HEAD:index.html
```

```
<div id="footer">contact : email.support@github.com</div>
```

```
=====
```

```
<div id="footer">  
    please contact us at support@github.com
```

```
</div>
```

```
>>>>> test:index.html
```

- Donde nos dice que la versión en **HEAD** (la rama master, la que habías activado antes de lanzar el comando de fusión) contiene lo indicado en la parte superior del bloque (todo lo que está encima de =====) y que la versión en **test** contiene el resto, lo indicado en la parte inferior del bloque.



# Resolución de conflictos

- Para resolver el conflicto, has de elegir manualmente el contenido de uno o de otro lado. Por ejemplo, puedes optar por cambiar el bloque, dejándolo así:
  - <div id="footer">
  - please contact us at email.support@github.com
  - </div>
- Esta corrección contiene un poco de ambas partes y **se han eliminado completamente las líneas <<<<< , ===== y >>>>>**. Tras resolver todos los bloques conflictivos, has de lanzar comandos git add para marcar cada archivo modificado. Marcar archivos como preparados (staged) indica a Git que sus conflictos han sido resueltos y haz tu commit.
- Si deseas usar una herramienta distinta de la escogida por defecto, puedes escogerla entre la lista de herramientas soportadas mostradas al principio ("merge tool candidates") tecleando el nombre de dicha herramienta.



# Reorganización de ramas

- **git rebase <rama-1> <rama-2>** replica los cambios de la rama <rama-2> en la rama <rama-1> partiendo del ancestro común de ambas ramas. El resultado es el mismo que la fusión de las dos ramas pero la bifurcación de la <rama-2> desaparece ya que sus commits pasan a estar en la <rama-1>.



# Cherry Pick

- **git cherry-pick** es un potente comando que permite que las confirmaciones arbitrarias de Git se elijan por referencia y se añadan al actual HEAD de trabajo.
- La ejecución de cherry-pick es el acto de elegir una confirmación de una rama y aplicarla a otra. git cherry-pick puede ser útil para deshacer cambios.
- Por ejemplo, supongamos que una confirmación se aplica accidentalmente en la rama equivocada. Puedes cambiar a la rama correcta y ejecutar cherry-pick en la confirmación para aplicarla a donde pertenece.
- git cherry-pick es una herramienta útil, pero **no siempre es una práctica recomendada**. La ejecución de cherry-pick puede generar duplicaciones de confirmaciones y, en muchos casos en los que su ejecución sí funcionaría, se prefieren las fusiones tradicionales. Por lo tanto, git cherry-pick es una herramienta útil solo en algunos casos...



# Cherry Pick

- **git cherry-pick** es un potente comando que permite que las confirmaciones arbitrarias de Git se elijan por referencia y se añadan al actual HEAD de trabajo.
- La ejecución de cherry-pick es el acto de elegir una confirmación de una rama y aplicarla a otra. git cherry-pick puede ser útil para deshacer cambios.
- Por ejemplo, supongamos que una confirmación se aplica accidentalmente en la rama equivocada. Puedes cambiar a la rama correcta y ejecutar cherry-pick en la confirmación para aplicarla a donde pertenece.
- git cherry-pick es una herramienta útil, pero **no siempre es una práctica recomendada**. La ejecución de cherry-pick puede generar duplicaciones de confirmaciones y, en muchos casos en los que su ejecución sí funcionaría, se prefieren las fusiones tradicionales. Por lo tanto, git cherry-pick es una herramienta útil solo en algunos casos...



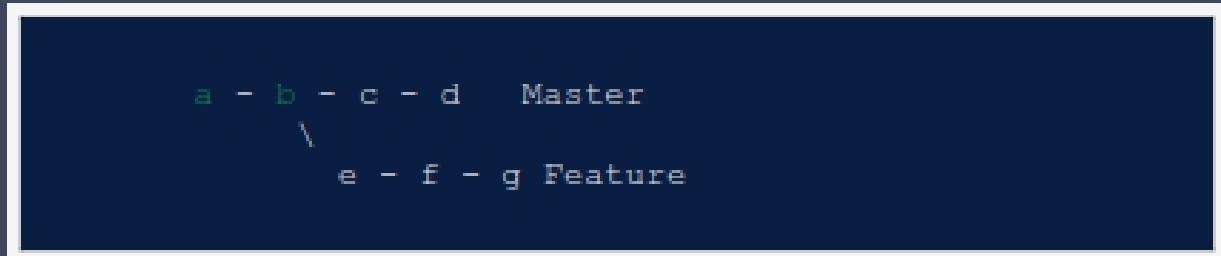
# Cherry Pick

- **Podemos usarlo:**
  - **Colaboración en equipo.** Incoprar funcionalidad que se ha confirmado en una rama en otra.
  - **Corrección de errores.** Si detectamos un error en unuestra rama principal que ha sido solucionado en otra rama, podemos incorporarlo.
  - **Recuperar confirmaciones perdidas.** Que existan en otra rama. Pero hay que tener cuidado



# Cherry Pick

- Para demostrar cómo utilizar git cherry-pick, supongamos que tenemos un repositorio con el siguiente estado de rama:

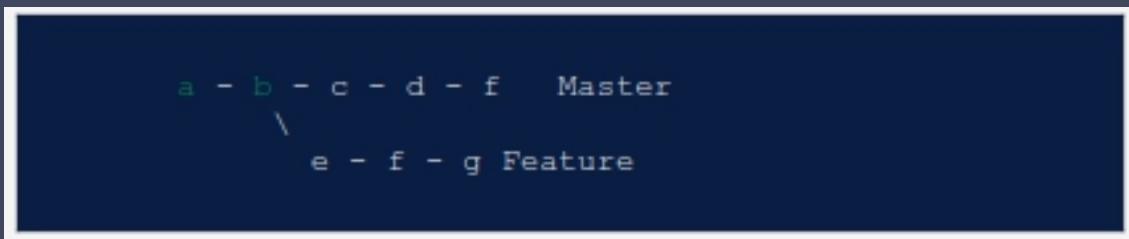


- El uso de git cherry-pick es sencillo y se puede ejecutar de la siguiente manera:
  - git cherry-pick commitSha
- En este ejemplo, commitSha es una referencia de confirmación. Puedes encontrar una referencia de confirmación utilizando el comando **git log**.



# Cherry Pick

- En este caso, imaginemos que queremos utilizar la confirmación 'f' en la rama master. Para ello, primero debemos asegurarnos de que estamos trabajando con esa rama master.
  - **git checkout master**
- A continuación, ejecutamos cherry-pick con el siguiente comando:
  - **git cherry-pick f** (f es el commit que queremos incorporar)
- Una vez ejecutado, el historial de Git se verá así:



- La confirmación f se ha sido introducido con éxito en la rama de funcionalidad



# ¡¡La cagaste!! ... ¡Y mucho!

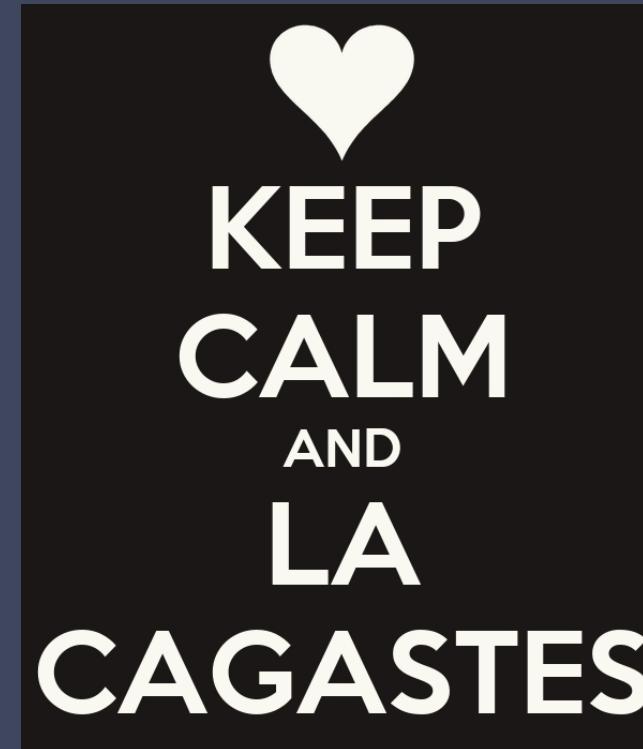
¿Alguna vez has hecho commits en la rama que no era?

## SOLUCION

- \$ git log
- \$ git checkout <rama\_correcta>
- \$ git cherry-pick <commit\_id>

Para dejar limpia la otra rama:

- \$ git checkout <rama\_incorrecta>
- \$ git reset --hard HEAD^1





# Ejercicio

- **Crea un repositorio y sobre él:**
  - Crea el fichero nombre.txt y añade tu nombre
  - Confirma
  - Crea una nueva rama llamada test
  - Crea un nuevo fichero llamado curso.txt
  - y añade tu curso
  - Abre el fichero nombre.txt y añade tus apellidos.
  - Confirma los cambios
  - Cambia a la rama principal y abre el fichero
  - nombre.txt
  - Añade tu dirección.
  - Fusiona la rama test en la rama master
  - Resuleve los conflictos que puedan darse
  - **Repite con VS Code y GitKraken**





# Ejercicio

- Repite el ejercicio anterior usando rebase, podrías explicar las diferencias y como se puede apreciar usando los comandos de Git
  - Repite Usando VS Code y GitKraken.
- 
- Es muy importante que adquieras una visión abstrancta de las ramas en tu cabeza, si no usa un entorno gráfico



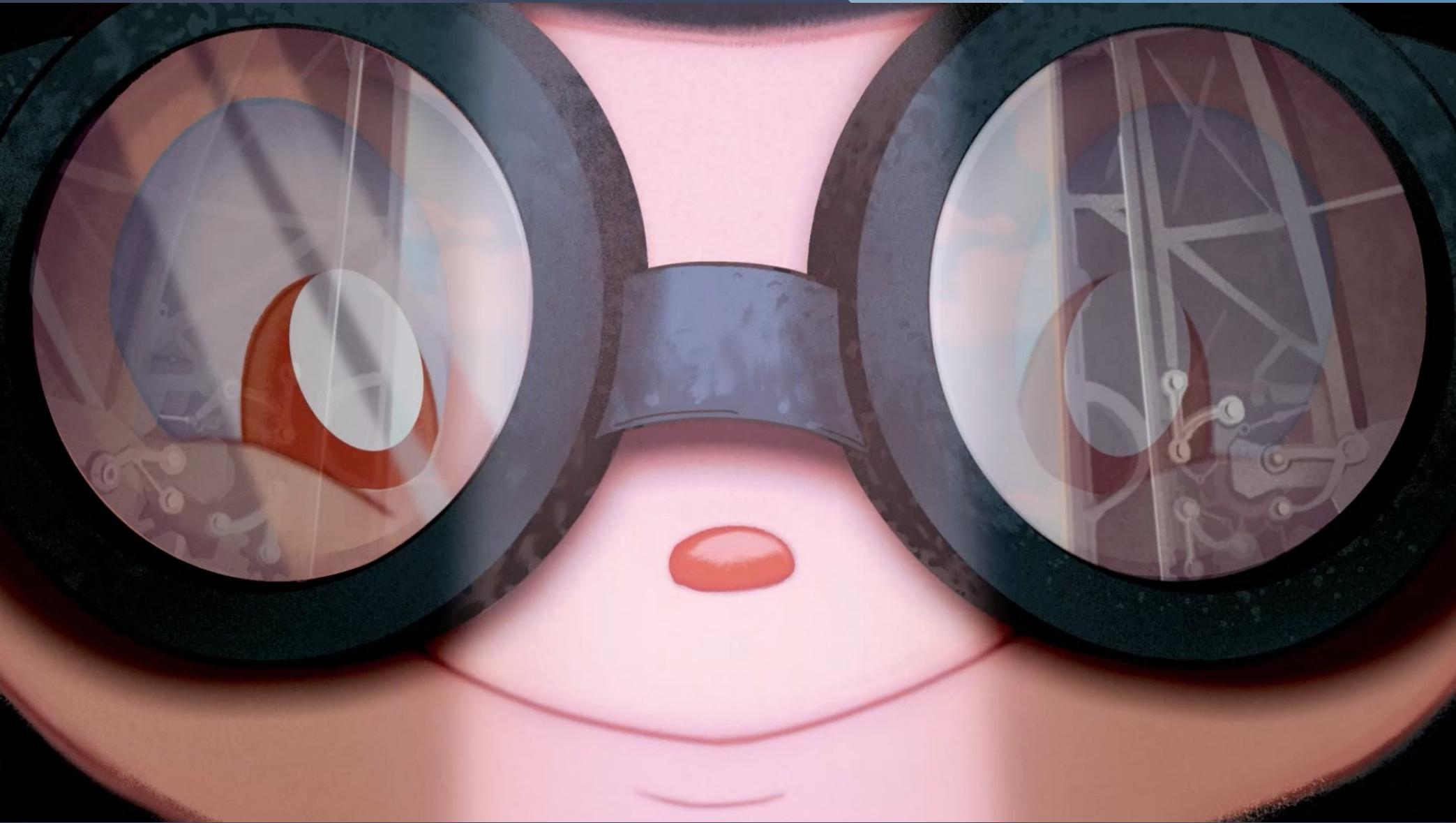


# Ejercicio

- **Crea un repositorio y sobre él:**
  - Crea el fichero nombre.txt y añade tu nombre
  - Confirma
  - Crea una nueva rama llamada test
  - Crea un nuevo fichero llamado curso.txt y añade tu curso
  - Abre el fichero nombre.txt y añade tus apellidos.
  - Confirma los cambios
  - Cambia a la rama principal y añade los nuevos cambios
  - a ella usando cherry pick
- **Repite con VS Code y GitKraken**

We Can Do It!





# GitFlow

Flujo de Trabajo con Git

# Git Flow

- Gitflow es una especie de idea abstracta de un flujo de trabajo de Git. Esto quiere decir que ordena qué tipo de ramas se deben configurar y cómo fusionarlas.





# Git Flow

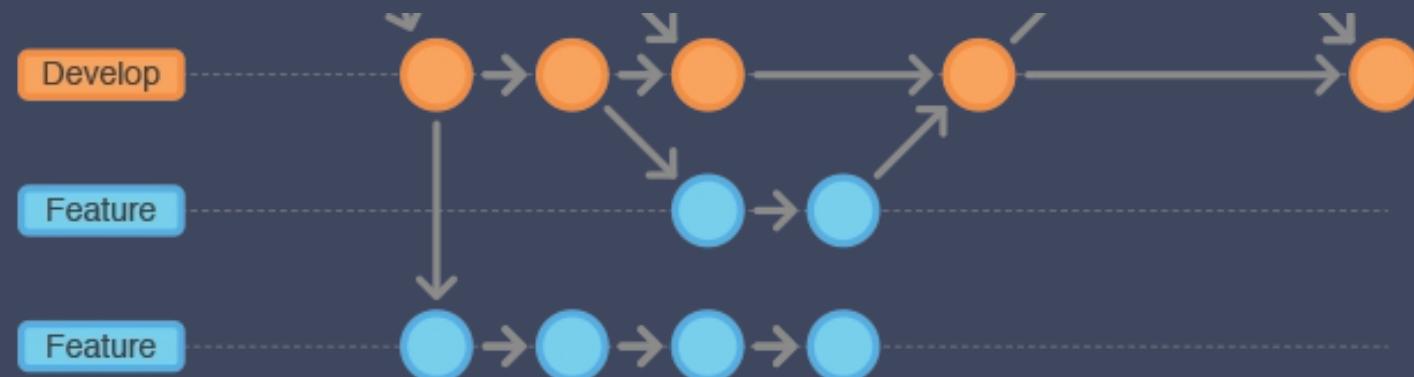
- En vez de una única rama maestra, este flujo de trabajo utiliza dos ramas para registrar el historial del proyecto. La **rama maestra** almacena el historial de publicación oficial y la **rama de desarrollo** sirve como rama de integración para funciones. Asimismo, conviene etiquetar todas las confirmaciones de la rama maestra con un número de versión.
- **Rama master:** cualquier commit que pongamos en esta rama debe estar preparado para subir a producción
- **Rama develop:** rama en la que está el código que conformará la siguiente versión planificada del proyecto





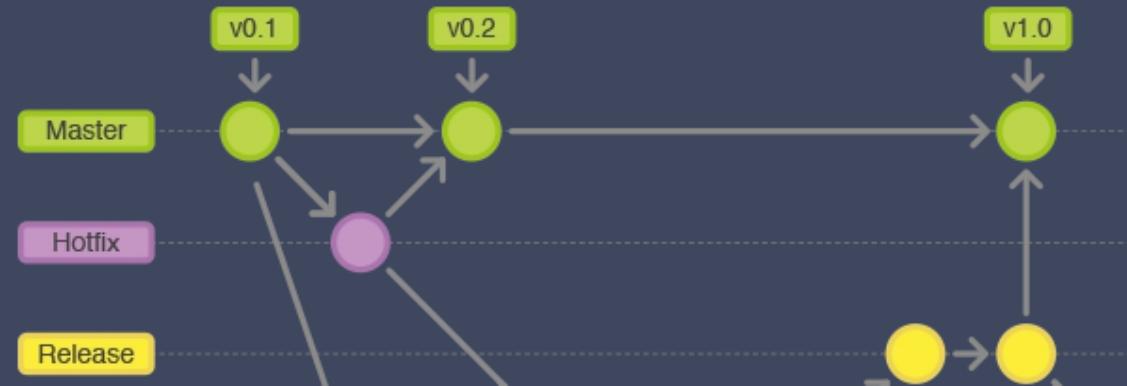
# Git Flow

- Cada **nueva función debe estar en su propia rama**, que se puede enviar al repositorio central para copia de seguridad/colaboración. Sin embargo, en vez de ramificarse de la maestra, las **ramas de función utilizan la de desarrollo como rama primaria**. Cuando una función está completa, se vuelve a fusionar en la de desarrollo. Las funciones nunca deben interactuar directamente con la maestra.
- Se originan a partir de la rama develop.
- Se incorporan siempre a la rama develop.
- Nombre: cualquiera que no sea master, develop, hotfix-\* o release-\*
- Estas ramas se utilizan para desarrollar nuevas características de la aplicación que, una vez terminadas, se incorporan a la rama develop



# Git Flow

- Una vez que el desarrollo ha adquirido suficientes funciones para **una publicación** (o se está acercando una fecha de publicación predeterminada), bifurca una rama de versión a partir de una de desarrollo. Al crear esta rama, se inicia el siguiente ciclo de publicación, por lo que no pueden añadirse nuevas funciones tras este punto; solo las soluciones de errores, la generación de documentación y otras tareas orientadas a la publicación deben ir en esta rama. Una vez que esté lista para el lanzamiento, la rama de publicación se fusiona en la maestra y se etiqueta con un número de versión. Además, **debería volver a fusionarse en la de desarrollo, que podría haber progresado desde que se inició la publicación.**
- Se originan a partir de la rama develop
- Se incorporan a master y develop
- Nombre: release-\*
- Estas ramas se utilizan para preparar el siguiente código en producción. En estas ramas se hacen los últimos ajustes y se corregen los últimos bugs antes de pasar el código a producción incorporándolo a la rama master.



# Git Flow

- Las **ramas de mantenimiento o "corrección" (hotfix)** se utilizan para reparar rápidamente las publicaciones de producción. Las ramas de corrección son muy similares a las ramas de publicación y a las de función, salvo porque se basan en la maestra en vez de la de desarrollo. Es la única rama que debería bifurcarse directamente a partir de la maestra. Una vez que la solución esté completa, debería fusionarse en la maestra y la de desarrollo (o la rama de publicación actual), y la maestra debería etiquetarse con un número de versión actualizado.
- Se origina a partir de la rama master
- Se incorporan a la master y develop
- Nombre: hotfix-\*
- Esas ramas se utilizan para corregir errores y bugs en el código en producción. Funcionan de forma parecida a las Releases Branches, siendo la principal diferencia que los hotfixes no se planifican





# Git Flow

- **Git Flow** (como herramienta) existe como paquete especial que te ayuda a combinar comandos git para poder llevar esta organización a cabo o trabajar con ella. También los puedes usar en GitKraken o VS Code.
- Ver:
  - <https://www.atlassian.com/es/git/tutorials/comparing-workflows/gitflow-workflow>
  - [https://danielkummer.github.io/git-flow-cheatsheet/index.es\\_ES.html](https://danielkummer.github.io/git-flow-cheatsheet/index.es_ES.html)
- Pero debemos entender que es solo una filosofía de trabajo. Además, como herramienta, todo se puede hacer con tus comandos de git como siempre. Tu equipo puede adoptar esta u otras similares o variaciones de la misma. En definitiva, si trabajas con Git intenta que tu trabajo siga unas directrices parecidas a estas:
  1. Se crea una rama de desarrollo a partir de la maestra.
  2. Una rama de publicación se crea a partir de la de desarrollo.
  3. Las ramas de función se crean a partir de la de desarrollo.
  4. Cuando una función está completa, se fusiona en la rama de desarrollo.
  5. Cuando la rama de publicación está lista, se fusiona en la de desarrollo y la maestra.
  6. Si se detecta una incidencia en la maestra, se crea una rama de corrección a partir de la maestra.
  7. Una vez que la corrección está completa, se fusiona tanto con la de desarrollo como con la maestra.



# GitHub

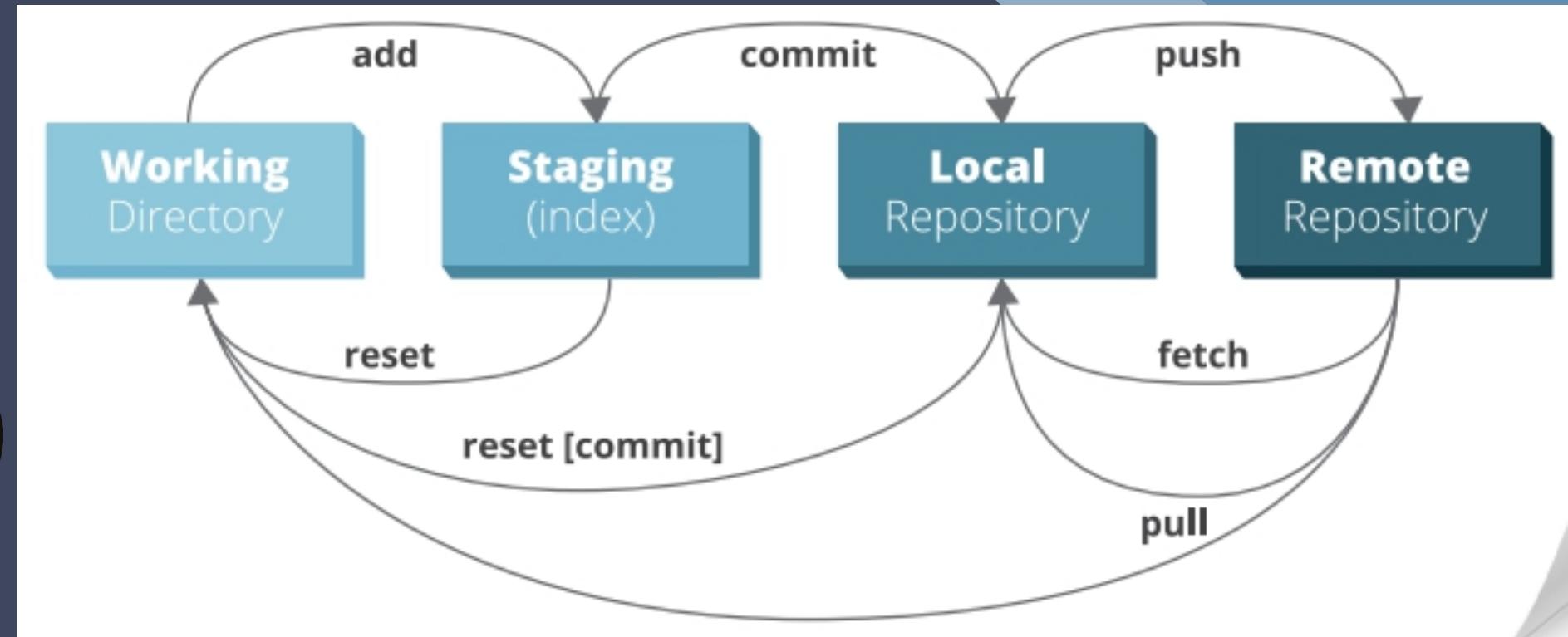
Nuestro repositorio remoto



# Trabajando con un servidor remoto GIT



# GitHub





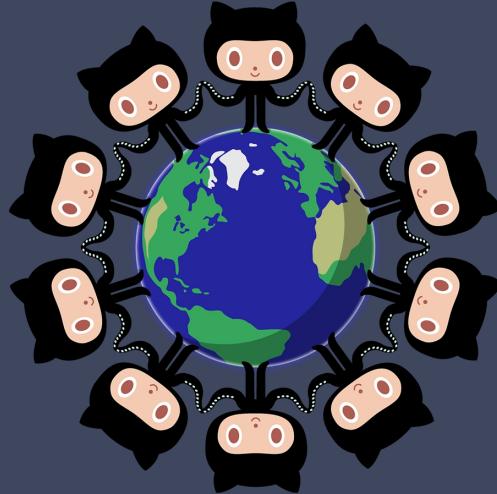
# Añadir un repositorio remoto

- **git remote add <repositorio-remoto> <url>** crea un enlace con el nombre <repositorio-remoto> a un repositorio remoto ubicado en la dirección <url>.
  - **git remote add origin <server>**
- Cuando se añade un repositorio remoto a un repositorio, Git seguirá también los cambios del repositorio remoto de manera que se pueden descargar los cambios del repositorio remoto al local y se pueden subir los cambios del repositorio local al remoto.
  - Lectura: <https://www.atlassian.com/es/git/tutorials/syncing>



# Lista de repositorios remotos

- **git remote** muestra un listado con todos los enlaces a repositorios remotos definidos en un repositorio local.
- **git remote -v** muestra además las direcciones url para cada repositorio remoto.





# Descargar cambios de un repositorio remoto

- **git pull <remoto> <rama>** descarga los cambios de la rama <rama> del repositorio remoto <remoto> y los integra en la última versión del repositorio local, es decir, en el HEAD.
  - Lectura: <https://www.atlassian.com/es/git/tutorials/syncing/git-pull>
  - **git pull origin master**
- **git fetch <remoto>** descarga los cambios del repositorio remoto <remoto> pero no los integra en la última versión del repositorio local: git fetch origin master o git fetch origin master
  - **git fetch origin**
  - Lectura: <https://www.atlassian.com/es/git/tutorials/syncing/git-fetch>



# Subir cambios a un repositorio remoto

- **git push <remoto> <rama>** sube al repositorio remoto <remoto> los cambios de la rama <rama> en el repositorio local.
  - **git push origin master**
  - Lectura: <https://www.atlassian.com/es/git/tutorials/syncing/git-push>





# Eliminar sincronización con un repositorio remoto

- **git remote remove <name>** donde name es el nombre del repositorio remoto





# Subir etiquetas a GitHub

- Aunque hayamos creado Tags en nuestro repositorio éstas no se sincronizan con Git si no se lo indicamos. Se deben subir las etiquetas que consideremos importantes.
  - `git push origin <tag_name>` sube la etiqueta indicada al repositorio
  - `git push --tags` sube todas las etiquetas. No es recomendado, solo debemos subir las etiquetas que consideremos importantes para compartir.
- Desde Git 2.4 se ha añadido la opción `push.followTags` para activar la opción de sincronizar las tags por defecto. Sincronizaría todas:
  - `git config --global push.followTags true`



# Colaborando con GitHub

Existen dos formas de colaborar en un repositorio alojado en GitHub:

## **Ser colaborador del repositorio.**

1. Recibir autorización de colaborador por parte de la persona propietaria del proyecto.
2. Clonar el repositorio en local.
3. Hacer cambios en el repositorio local.
4. Subir los cambios al repositorio remoto. Primero hacer git pull para integrar los cambios remotos en el repositorio local y luego git push para subir los cambios del repositorio local al remoto.



# Colaborando con GitHub

**Replicar el repositorio y solicitar integración de cambios.**

1. Replicar el repositorio remoto en nuestra cuenta de GitHub mediante un fork.
2. Hacer cambios en nuestro repositorio y sincronizarlos con nuestro remoto.
3. Solicitar a la persona propietaria del repositorio original que integre nuestros cambios en su repositorio mediante un **pull request**.





# Colaborando con GitHub

## Pasos para actualizar la copia (fork) de un repositorio de Git:

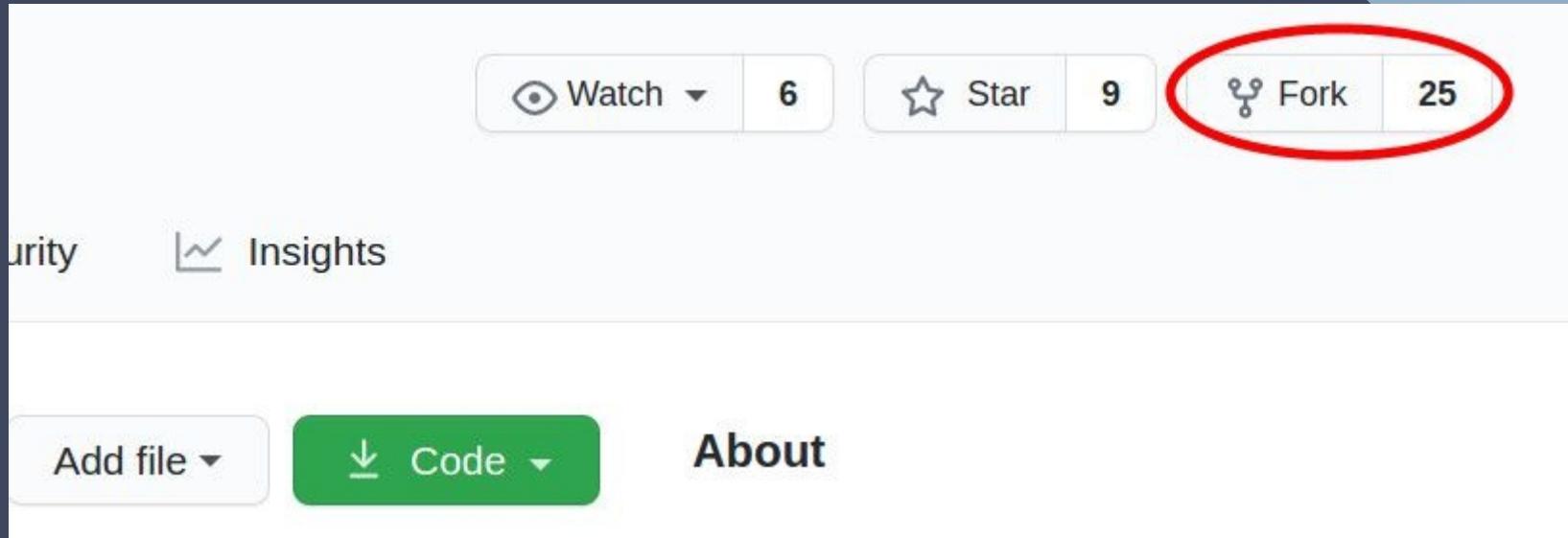
- Agregar la referencia al repositorio remoto original, al cual llamaremos «upstream», esto lo logramos con el comando:`git remote add upstream https://github.com/whoever/whatever.git`
- Traernos todas las ramas de dicho repositorio remoto con: `git fetch upstream`
- Irnos a la rama que queremos actualizar, por ejemplo master: `git checkout master`
- Reescribir nuestra rama master con los commits nuevos de la rama master del repositorio original con:
- `git rebase upstream/master`
- Finalmente si queremos actualizar nuestro fork remoto, lo haremos ejecutando `git push -f origin master`

**Recuerda que estos cambios no afectarán al repositorio original, sólo a tu fork (sobretodo si no tienes acceso de escritura al mismo).**



# Pull Request en GitHub

1. **Realicemos un fork del repositorio.** Realiza un fork del repositorio haciendo un clic en el botón fork de la parte superior de la página. Esto creará una instancia del repositorio completo en tu cuenta.

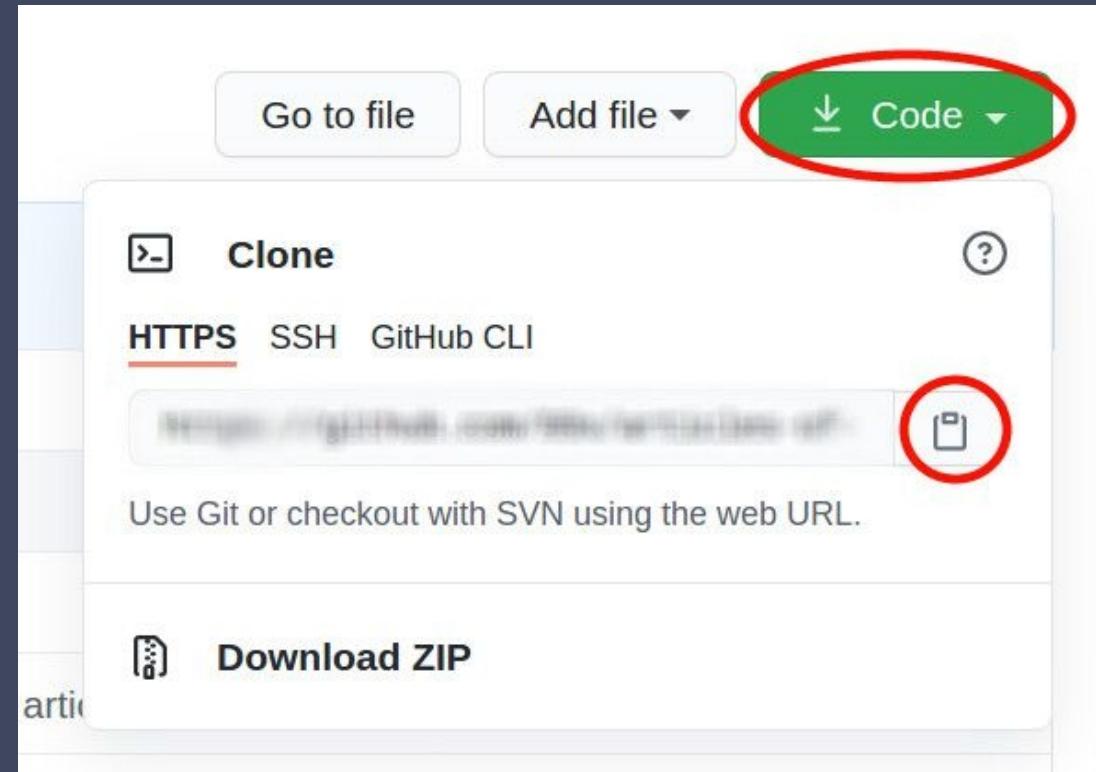




# Pull Request en GitHub

**2 Clona el repositorio.** Una vez que el repositorio esté en tu cuenta, clónalo a tu computador para trabajar lo localmente. Para clonarlo, has clic en el botón "Code" y copia el link.

\$ git clone [DIRECCIÓN HTTPS]





# Pull Request en GitHub

**3. Crea una rama.** Es una buena práctica crear una rama (branch) nueva cuando trabajas con repositorios, ya sea que se trate de un proyecto pequeño o estés contribuyendo en un equipo de trabajo. El nombre de la rama debe ser breve y debe reflejar el trabajo que estamos haciendo.

Ahora crea una rama usando el comando git checkout:

```
$ git checkout -b [Nombre de la Rama]
```



# Pull Request en GitHub

**4. Realiza cambios y confímalos.** Has cambios esenciales al proyecto y guárdalos.

Luego ejecuta git status , y verás los cambios.

grega esos cambios a la rama recién creada usando el comando git add.

```
$ git add .
```

Ahora confirma esos cambios utilizando el comando git commit:

```
$ git commit -m "Actualizando la función de búsqueda recursiva en Personas"
```



# Pull Request en GitHub

**5. Envía los cambios a GitHub.** Para enviar los cambios a GitHub, debemos identificar el nombre del repositorio remoto.

```
$ git remote
```

Para este repositorio el nombre es "origin".

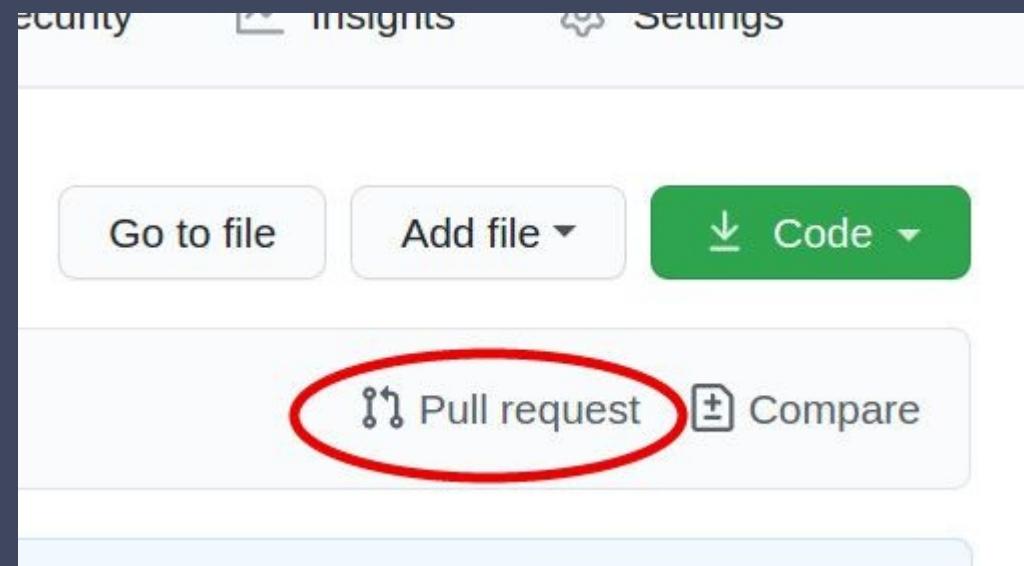
Luego de identificar el nombre podemos enviar en forma segura los cambios a GitHub.

```
git push origin [Nombre de la Rama]
```



# Pull Request en GitHub

6. Crea un pull request. Ve a tu repositorio en GitHub y verás un botón llamado "Pull request", has clic en él.





# Pull Request en GitHub

Por favor, provee todos los detalles necesarios de lo que has hecho (puedes referenciar problemas utilizando "#"). Ahora, envía el pull request.

¡Felicitaciones! Has hecho tu primer pull request. Si tu pull request es aceptado recibirás un mail.

The screenshot shows the GitHub interface for creating a pull request. At the top, it says "Open a pull request" and provides instructions: "Create a new pull request by comparing changes across two branches. If you need to, you can also compare across forks." Below this, there are dropdown menus for "base repository" (set to "99xt-incubator/articles-of-the..."), "base" (set to "master"), "head repository" (set to "ThanoshanMV/articles-of-the..."), and "compare" (set to "my-article"). A green success message "✓ Able to merge. These branches can be automatically merged." is displayed. The main area is a rich text editor with a placeholder "Adding an article to week 02 of articles of the week". Below the editor, there's a "Leave a comment" section and a note "Attach files by dragging & dropping, selecting or pasting them.". At the bottom, there are two checkboxes: "Allow edits from maintainers" (with a "Learn more" link) and "Create pull request". Below the editor, summary statistics are shown: "1 commit", "1 file changed", "0 commit comments", and "1 contributor".



# Pull Request en GitHub

**7. Sincroniza tu rama maestra con la del repositorio original.** Antes de enviar cualquier pull request al repositorio original debes sincronizar tu repositorio con aquel.

Incluso si no vas a enviar un pull request al repositorio original, es mejor efectuar la sincronización, ya que pueden haberse agregado algunas prestaciones o funciones adicionales y haberse corregido algunos errores desde la vez que realizaste un fork de aquel repositorio.

Sigue estos pasos para actualizar/sincronizar aquellos cambios con tu rama maestra:

1. evisa en que rama estás ubicado.

```
$ git branch
```

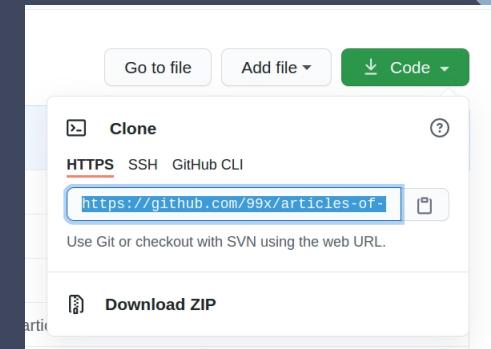
2. Cambia a la rama maestra.

```
$ git checkout master
```

3. Agrega el repositorio original como un repositorio upstream.

Para poder extraer los cambios desde el repositorio original a tu versión local, necesitas agregar el repositorio Git original como un repositorio upstream.

```
$ git remote add upstream [HTTPS]
```





# Pull Request en GitHub

4. Busca (fetch) el repositorio.

Busca todos los cambios del repositorio original. Las confirmaciones (commits) del repositorio original serán almacenadas en una rama local llamada upstream/master.

**\$ git fetch upstream**

5. Fusionala.

Fusiona los cambios de la rama upstream/master a tu rama maestra local. Esto hará que tu rama maestra se sincronice con el repositorio upstream sin perder tus cambios locales.

**\$ git merge upstream/master**

6. Envía (push) los cambios a GitHub

En este punto tu rama local está sincronizada con la rama maestra del repositorio original. Si deseas actualizar el repositorio de GitHub, necesitas enviar tus cambios.

**\$ git push origin master**

NOTA: Luego de sincronizar tu rama maestra puedes eliminar el repositorio upstream, si lo desea. Pero lo necesitará para actualizar/sincronizar tu repositorio en el futuro, por lo que es una buena práctica conservarlo.



# Pull Request en GitHub

## 8. Elimina ramas innecesarias

Las ramas son creadas para propósitos especiales. Una vez que ese propósito se cumple, aquellas ramas ya no son necesarias, por lo que puedes eliminarlas.

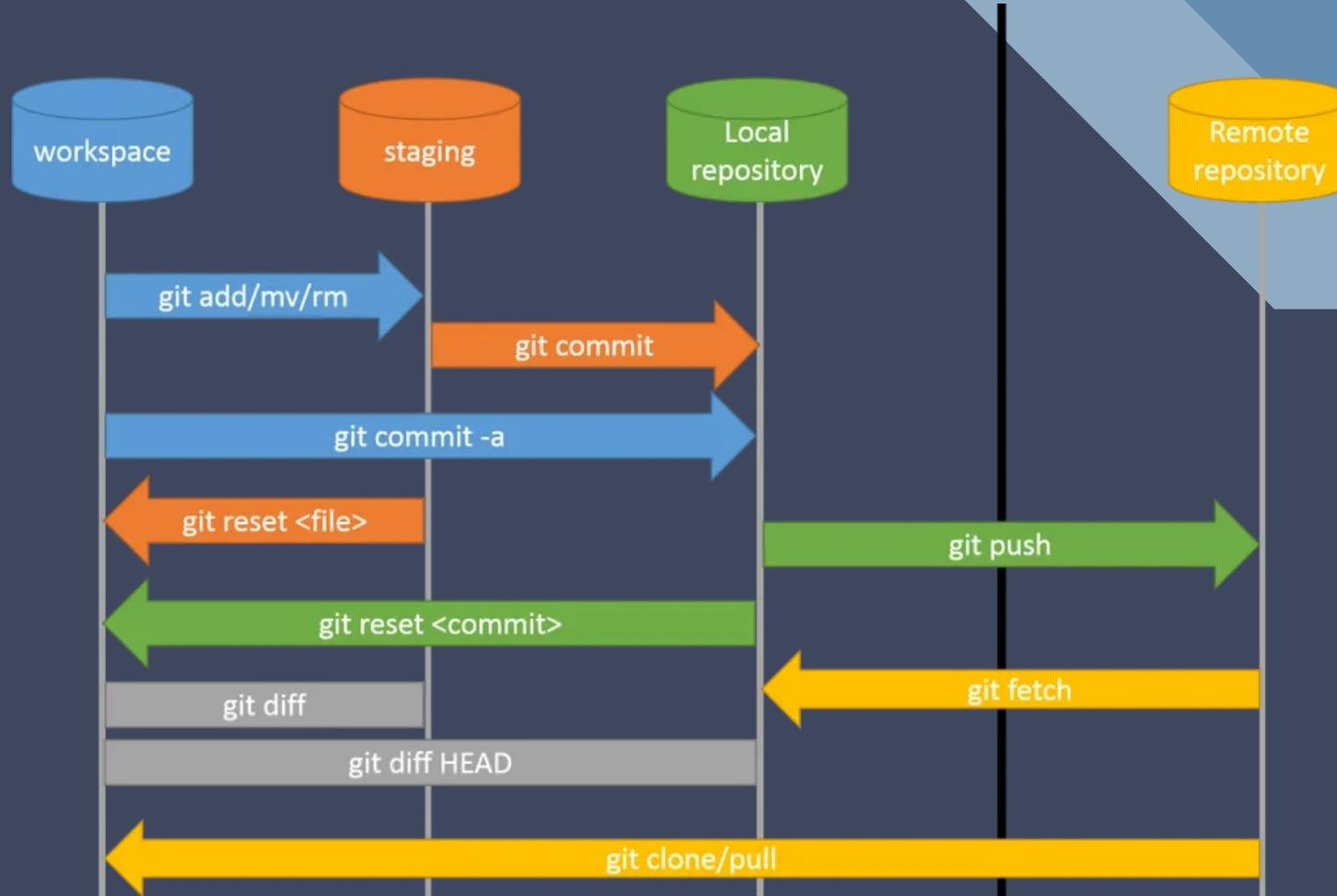
```
$ git branch -d [Nombre de la Rama]
```

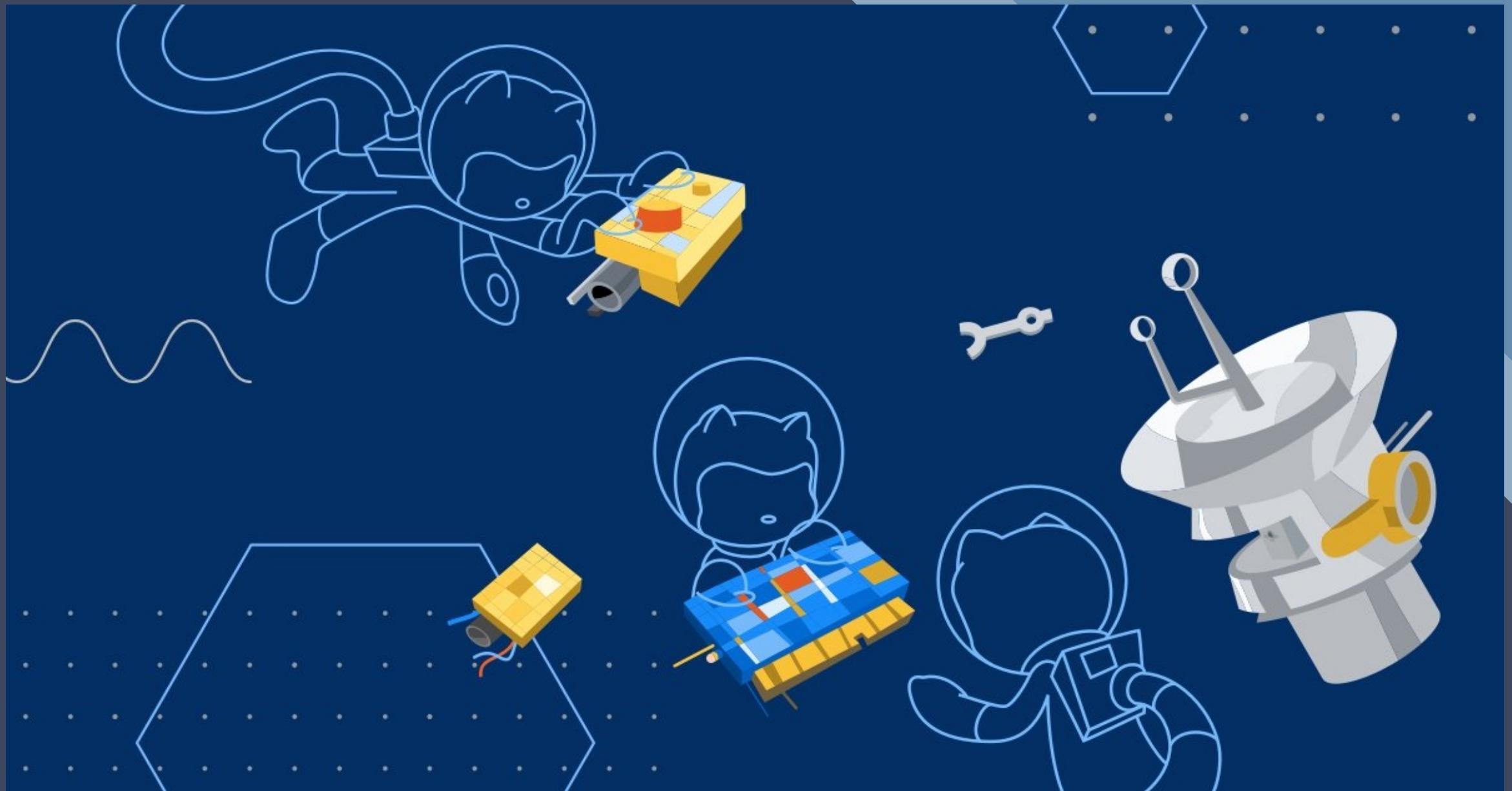
También, puedes eliminar su versión en GitHub.

```
git push origin --delete [Nombre de la Rama]
```



# Resumen Git y GitHub





# Herramientas

Mejorando nuestra productividad



# Git y VS Code

- Algunas extensiones interesantes
  - [Git Lens](#): Para manejar Blame e información.
  - [Git Graph](#): Para manejar ramas e información
  - [Gitflow Actions Sidebar](#): Para manejar el Flujo de GitFlow
  - [Git History](#): Historia de cambios
  - Hay más, pero lo básico lo tienes en el [tutorial](#) oficial de Microsoft.
  - También te recomiendo este [tutorial](#) de Lemoncode



# Git y VS Code

```
ts walkThroughPart.ts src/vs/workbench/parts/welcome/walkThrough/electron-browser

406 → → → → → → snippet: i
407 → → → → → → });
408 → → → → → });
409 → → → });
Accept Current Change | Accept Incoming Change | Accept Both Changes | Compare Changes
410 <<<<< HEAD (Current Change)
411 → → → → this.updateSizeClasses();
412 → → → → this.multiCursorModifier();
413 → → → → this.contentDisposables.push(this.configurationService.onDidU
414 =====
415 → → → → this.toggleSizeClasses();
416 >>>> Test (Incoming Change)
417 → → → → if (input.onReady)
418 → → → → input.onReady(i
419 → → → → }
420 → → → → this.scrollbar.scan
421 → → → → this.loadTextEditor
422 → → → → this.updatedScrollP
423 → → → });
424 }

● markdown.md docs\ticino\languages - Changes on working tree
14 # Markdown and VS Code
15
16 ## Markdown Preview
17 Open any Markdown file and press `kb(workbench.action.
18
19 ## Using your own CSS
20 By default, we use a CSS style for the preview that
```

Branches: Show All  Show Remote Branches

Graph	Description	Date	Author	Commit
	upstream/live Merge pull request #1419...	3 Sep 2019 23:27	Maira Wenzel	4e2d355b
	ardalis/scaling-scenarios origin upda...	3 Sep 2019 21:37	Steve Smith	5a3b41ec
	upstream/lateX Test double escape to re...	3 Sep 2019 20:47	Maira Wenzel	2ad84bf7
	upstream/mairaw-patch-1 update version...	3 Sep 2019 20:27	Maira Wenzel	3691754b
	Updating scaling chapter	3 Sep 2019 20:04	Steve Smith	a5fbc086
	upstream/master move article to tutorials...	3 Sep 2019 19:27	Maira Wenzel	06ba6e56
	turn off feedback for a few areas (#14187)	3 Sep 2019 19:26	Maira Wenzel	7fbe3c8f
	US 1583733 Add missing language IDs - 09 ...	3 Sep 2019 19:24	Tom Pratt	37c9da37
	Update erroreport-compiler-option.md (#14...	3 Sep 2019 19:19	Youssef Victor	57688c47
	Update bc30456.md (#14186)	3 Sep 2019 19:14	Youssef Victor	2b1732a4
	an Petrusha	943fe0c2		
	Juricio de los San...	a43af573		
	Maira Wenzel	4b647aed		
	Sirry Kim	59fc85db		
	Maira Wenzel	92cf5f0b		
	Maira Wenzel	9ecf85e1		
	Tom Dykstra	3e566025		
		43a6d908		

```
12
13 # Markdown and Visual Studio Code
14
15 Working with Markdown in Visual Studio Code is pretty
16
17 ## Markdown Preview
18 Open any Markdown file and press `kb(workbench.action.
19
20 Here is an example with a very simple file.
21
22 ! [Markdown Preview] (images/markdown/MDPreview.png)
23
24 > **Tip:** You can also click on the icons on the top
25
26
27 ## Using your own CSS
28 By default, we use a CSS style for the preview that
29
30 For instance in the screen shoot above we used a cus
```

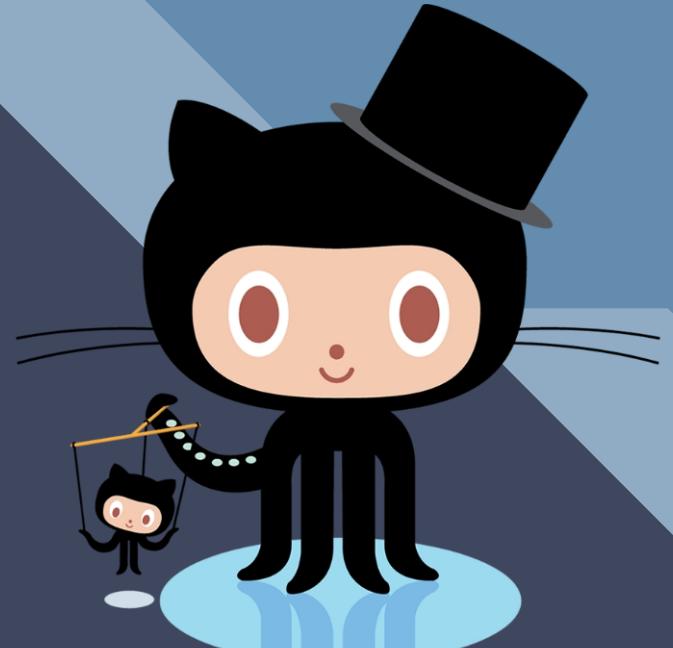


# Git y GitKraken

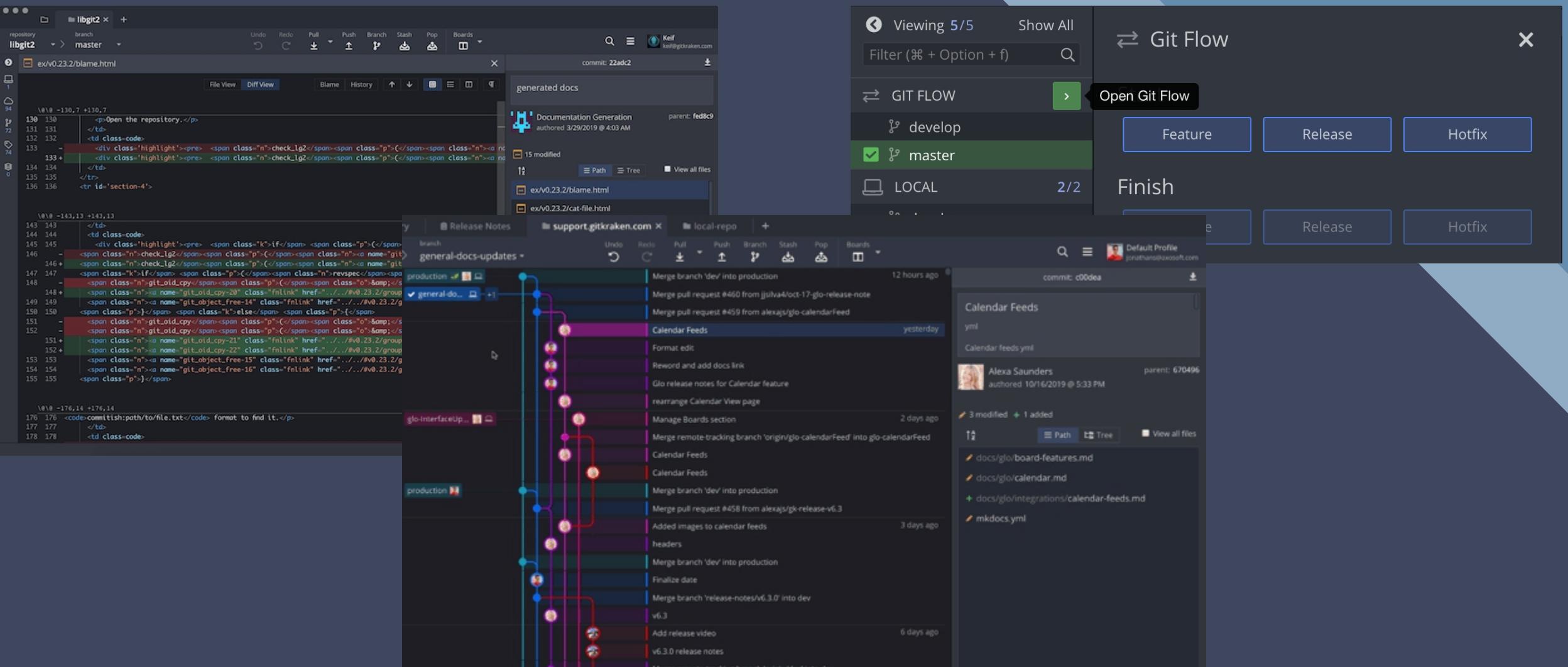
- Sin duda la mejor herramienta para manejar Git a toda potencia
- Te recomiendo estos recursos
  - [Documentación oficial](#)
  - Guías en PDF rápidas ([1](#) y [2](#))
  - [Soporte para Git Flow](#)
  - [Guia de usuario](#)



# GitKraken



# Git y GitKraken



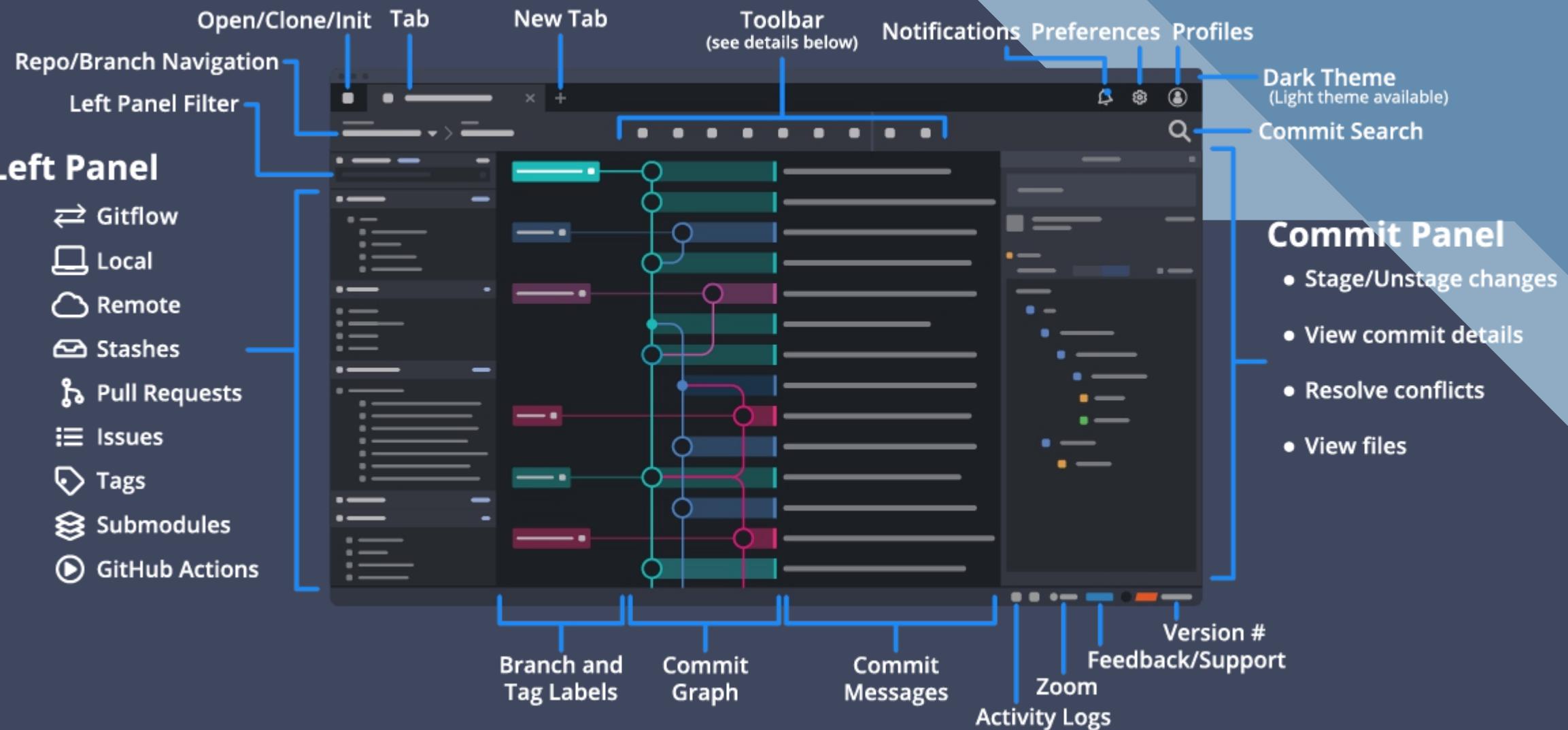


# GitKraken

El Mejor GUI para Git



# GitKraken





# GitKraken. Stage y Commit

The screenshot shows the GitKraken interface with the following details:

- Repository:** support.gitkraken.com
- Branch:** v7-5-docs-update
- Commit Message:** // WIP
- Unstaged Files:** 7 file changes on v7-5-docs-update
  - Stage all changes
  - docs/img/documentation/... /amend-checkbox.png
  - docs/img/documentation/... /amend-checkbox@2x.png
  - docs/img/documentation/working-w... /amend.png
  - docs/img/documentation/working-w... /amend@2x.png
  - docs/img/documentation/working-... /message.png
- Staged Files:** 1
  - content/committing-work/commits.md**Unstage all changes**
- Commit Message:** content/committing-work/commits.md
- Commit Tip:** Use CMD/Ctrl + Shift + Enter from here to stage all and commit
- Commit changes to 1 file**



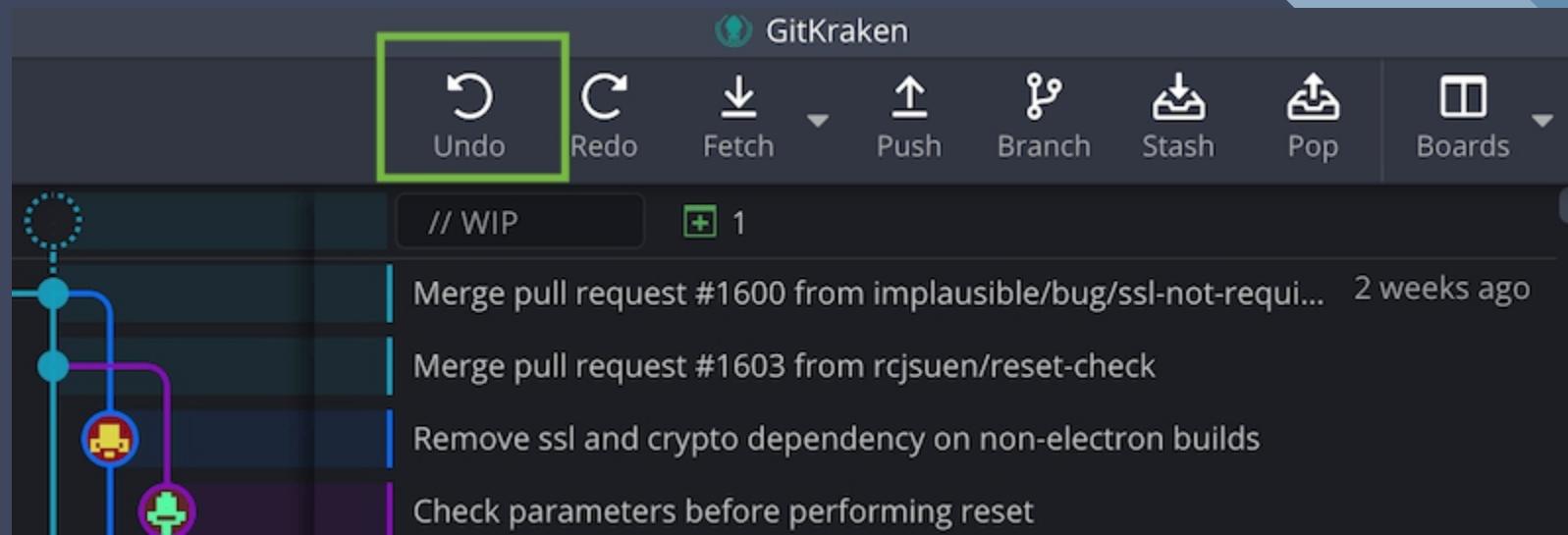
# GitKraken. Diff, Blame y History

The screenshot shows the GitKraken interface with the following details:

- Repository:** support.gitkraken.com
- Branch:** v7-5-docs-update
- File:** docs/account/faq.md
- Panels:**
  - Diff View:** Shows the diff between the current state and the previous commit.
  - Blame:** Shows the history of changes for each line of code.
  - History:** Shows the commit history for the file.
- Right Panel:** Displays the commit history for the file, showing 5 commits selected from 13 total changes in the working directory.
  - Viewing merged diff of 5 commits:**
    - Purchase FAQ change (bc8b7d)
    - v7.5 Trial page changes (0f8bb7)
    - Merge branch 'update-cta-links' into dev (d3910d)
    - Update CTA urls and fix formatting (12/31/2020 @ 12:45 PM by Jonathan Silva)
    - update logo file to no axosoft badge version (12/22/2020 @ 12:23 PM by Diane)
  - File Changes:** Shows the diff for the selected commit (d3910d).
    - Changes:** 8 modified, 18 added, 4 deleted.
    - Blame:** Shows the blame for each line of code, indicating which commit made each change.



# GitKraken. Hacer y Rehacer





# GitKraken. Manejando Ramas

The screenshot shows a terminal window with a context menu open over a file named 'glo/calendar.md'. The menu includes options like 'Pull (fast-forward if possible)', 'Push', 'Set Upstream', 'Rebase dev onto production', 'Fast-forward production to dev', 'Merge dev into production' (which is highlighted), 'Checkout production', 'Checkout origin/production', 'Create branch here', 'Cherrypick commit', 'Reset dev to this commit', and 'Revert commit'. The main area displays two conflicting commits, A and B, side-by-side. Commit A is on 'master' and Commit B is on 'MergeConflict'. Both commits contain similar documentation for date filters. The 'Output' section at the bottom shows the merged content where the date filter documentation from both commits has been combined. The status bar at the bottom right indicates 'conflict 1 of 1'.

```
⚠ glo/calendar.md (1 conflict)
A Commit edd642 on master
B Commit 938726 on MergeConflict
```

You can do some fancy footwork with due date filters. Here's what's available.

### Functional Requirements

Filterable fields: `created`, `createddate`, `due`, `duedate`.

Supported qualifiers: `>`, `>=`, `<`, `<=`, `after:`, `before:`

## Before/After

You can add an operator before the date. `after:` is an alias of `>` and `before:` is an alias of `<`.

- `due:>2018-12-31`
- `due:after:2018-12-31`
- `due:<2018-12-31`
- `due:<=2018-12-31`

## Date Range

More complex date ranges can be specified by combining the operators above. For example, `due:2018-12-31..2019-01-01` would select all dates between December 31, 2018, and January 1, 2019, inclusive.

## Filter by exact date

Use a date or one of the strings: `today`, `yesterday`, `tomorrow` .

- `due:2018-12-31`
- `due:today`
- `due:yesterday`
- `due:tomorrow`
- `due:at a point`

## Before/After

You can add an operator before the date. `after:` is an alias of `>` and `before:` is an alias of `<`.

conflict 1 of 1

```
77 You can do some fancy footwork with due date filters. Here's what's available.
78
79 ### Functional Requirements
80
81 Filterable fields: `created`, `createddate`, `due`, `duedate`.
82
83 Supported qualifiers: `>`, `>=`, `<`, `<=`, `after:`, `before:`
84
85 ## Before/After
86 You can add an operator before the date. `after:` is an alias of `>` and `before:` is an alias of `<`.
87
88 - `due:>2018-12-31`
89 - `due:after:2018-12-31`
90 - `due:<2018-12-31`
91 - `due:<=2018-12-31`
92
93 ## Date Range
94 More complex date ranges can be specified by combining the operators above. For example, `due:2018-12-31..2019-01-01` would select all dates between December 31, 2018, and January 1, 2019, inclusive.
95
96 ## Filter by exact date
97 Use a date or one of the strings: `today`, `yesterday`, `tomorrow` .
98
99 - `due:2018-12-31`
100 - `due:today`
101 - `due:yesterday`
102 - `due:tomorrow`
103 - `due:at a point`
104
105 ## Before/After
106 You can add an operator before the date. `after:` is an alias of `>` and `before:` is an alias of `<`.
107
108 - `due:>2018-12-31`
```



# GitKraken. Pull Request

The screenshot illustrates the GitKraken interface for creating a pull request. On the left, a sidebar shows repository statistics: LOCAL (9/9), REMOTE (21/21), PULL REQUESTS (31/31), TAGS (1/1), and SUBMODULES (0). The main area displays a "Create Pull Request" dialog for GitHub.com, Bitbucket.org, and GitLab.com. The dialog includes fields for "From Repo" (Select...), "To Repo" (Select...), "Branch" (Select...), and a "pull request title" input field. Below these are "Pull request description" and "Edit on GitHub" and "Create Pull Request" buttons. A modal at the bottom provides merge options: "Merge pull request" (selected), "Create a merge commit" (checkbox checked), "Squash and merge", and "Rebase and merge". On the right, another "Create Pull Request" dialog is shown for GitHub.com, GitLab.com, GitLab (self-hosted), Bitbucket.org, and VSTS. This dialog includes fields for "Title" ("Bump luna to next version") and "Description" ("to v2.6"). It also features sections for "Reviewers" (jesusmurillo, lhinsley), "Assignees" (jjsilva4, lhinsley), and "Labels" (enhancement, question). A note at the bottom states: "Your pull request reviewers will not be transferred to GitHub if you choose to Edit on GitHub".



# GitKraken. GitFlow

The screenshot shows the GitKraken application interface with the following components:

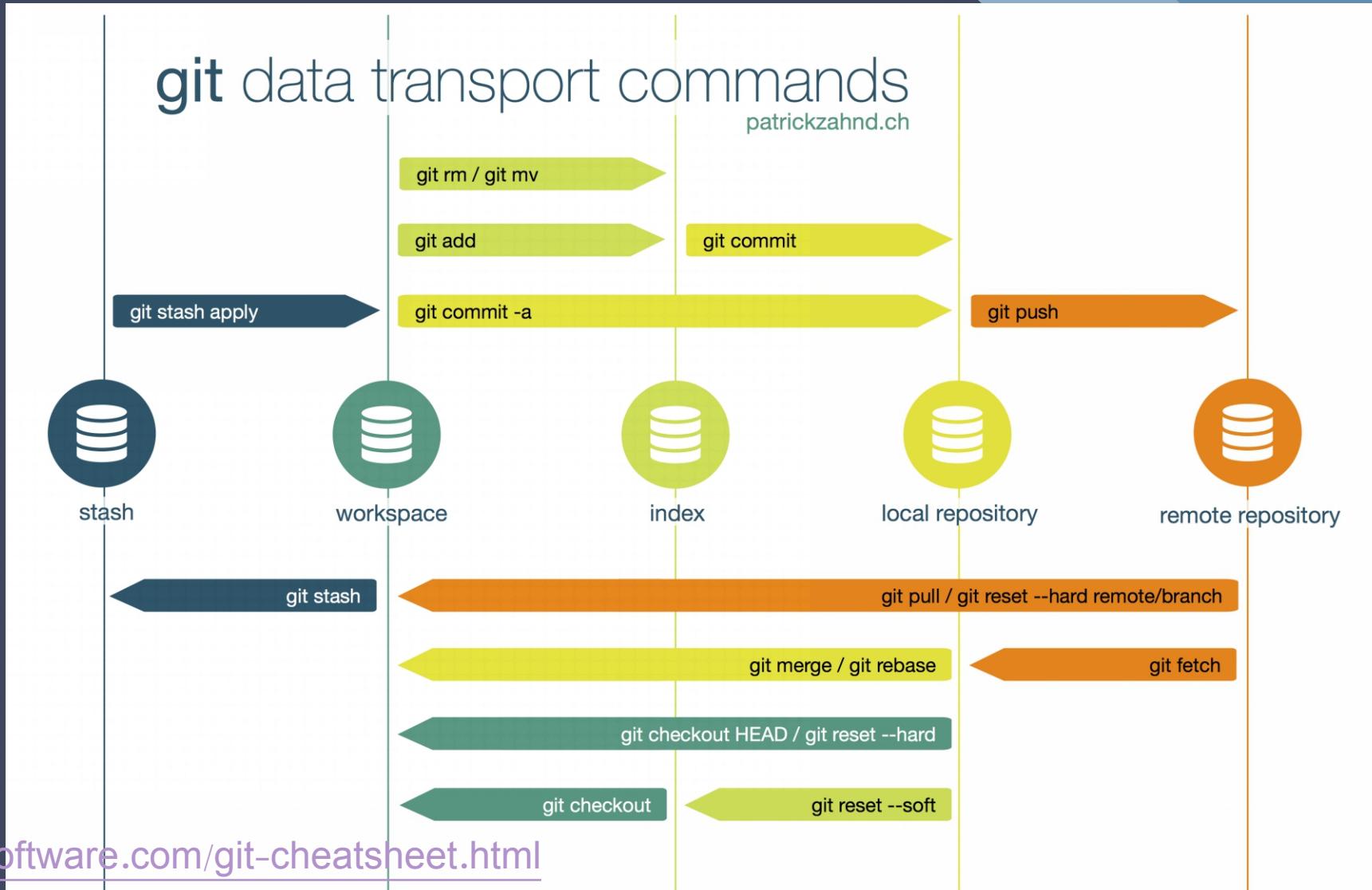
- Left Sidebar (Preferences):** Includes sections for General, Profiles, Authentication, UI Preferences, Editor Preferences, Repo-Specific Preferences (with a link to support.gitkraken.com), and Git Flow (which is selected).
- Main Area (Git Flow Configuration):**
  - Branches:** Shows Master (branch master) and Develop (branch develop).
  - Prefixes:** Shows Feature (prefix feature/), Release (prefix release/), Hotfix (prefix hotfix/), and Version Tag (prefix version/). An "Initialize Git Flow" button is located here.
- Top Right Overlay (GIT FLOW):** A dark overlay window titled "GIT FLOW" with a number "4" in the top right corner. It lists branches: develop, feature, doc-up, fix-typos (selected with a checkmark), and master.
- Bottom Center Modal (Git Flow):** A modal window titled "Git Flow". It shows a list of branches: develop, master (selected with a checkmark), LOCAL (2/2), develop, and master. It includes buttons for "Open Git Flow", "Feature", "Release", and "Hotfix".



# Resumen

Potencia tus desarrollos

# Resumen Git y GitHub



<https://ndpsoftware.com/git-cheatsheet.html>





# Referencias

- [Git](#). Sitio web de Git.
- [GitHub](#). Sitio web de GitHub.
- [Pro Git](#). Libro oficial de Git.
- [Ry's Git Tutorial](#). Tutorial de Git gratuito.
- [Gitcheats](#). Página de ayuda sobre los comandos de Git.
- [Comandos](#) usuales de Git
- [Aprende Git](#). Página web para dominar Git.
- [Tutoriales Altassian](#). Pagina web sobre uso de Git y distintos tutoriales
- [GitFlow](#). Página sobre Gitflow





“

‘Programa siempre tu código como si el tipo que va a tener que mantenerlo en el futuro fuera un violento psicópata que sabe dónde vives’.

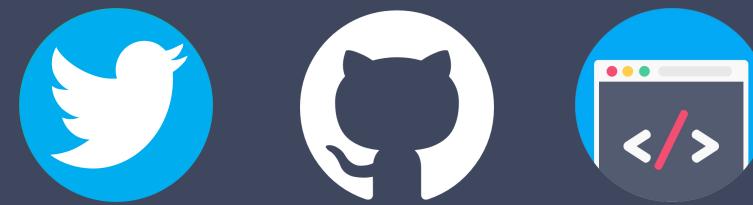
- Martin Goldin

”



# Recursos

- Twitter: <https://twitter.com/joseluisgonsan>
- GitHub: <https://github.com/joseluisgs>
- Web: <https://joseluisgs.github.io>



# Gracias

José Luis González Sánchez

