Curso de GIT y control de versiones

Marco Teran

29 de agosto de 2022

Índice

Resumen

Domina los fundamentos del desarrollo de hardware. Aplica elementos de robótica y conecta tus dispositivos a internet. Dale vida a todo lo que te rodea con el Curso de IoT de Platzi.

- Conocer diversos protocolos de comunicación
- Desarrollar un proyecto de IoT
- Dominar el proceso de desarrollo desde la elección de componentes hasta su implementación

1. ¿Qué es Git?

Git es un software de control de versiones diseñado por **Linus Torvalds**, pensando en la eficiencia y la confiabilidad del mantenimiento de versiones de aplicaciones cuando estas tienen un gran número de archivos de código fuente.
Página oficial de Git: git-scm.com/

1.1. GitHub

En su lugar GitHub es una forja para alojar proyectos utilizando el sistema de control de versiones Git. GitHub sería la red social de código para los programadores, tu propio curriculum vitae.

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

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

- El comando para iniciar nuestro repositorio, o sea, indicarle a Git que queremos usar su sistema de control de versiones en nuestro proyecto, es `git init`.
- El comando para que nuestro repositorio sepa de la existencia de un archivo o sus últimos cambios es `git add`. Este comando no almacena las actualizaciones de forma definitiva, solo las guarda en algo que conocemos como **staging area**.
- El comando para almacenar definitivamente todos los cambios que por ahora viven en el **staging area** es `git commit`. También podemos guardar un mensaje para recordar muy bien qué cambios hicimos en este *commit* con el argumento `-m "Mensaje del commit"`.
- Por último, si queremos mandar nuestros *commits* a un servidor remoto, un lugar donde todos podamos conectar nuestros proyectos, usamos el comando git push.

2. Instalando Git y GitBash

Windows y Linux tienen comandos diferentes, graban el enter de formas diferentes y tienen muchas otras diferencias. Cuando instales Git Bash en Windows debes elegir si prefieres trabajar con la forma de Windows o la forma de UNIX (Linux y Mac) . Ten en cuenta que, normalmente, los entornos de desarrollo profesionales tienen personas que usan sistemas operativos diferentes. Esto significa que, si todos podemos usar los mismos comandos, el trabajo resultará más fácil para todos en el equipo. Los comandos de UNIX son los más comunes entre los equipos de desarrollo. Así que, a menos que trabajes con tecnologías nativas de Microsoft (por ejemplo, .NET), la recomendación es que elijas la opción de la terminal tipo UNIX para obtener una mejor compatibilidad con todo tu equipo.

2.1. Instalando Git en Linux

Cada distribución de Linux tiene un comando especial para instalar herramientas y actualizar el sistema. En las distribuciones derivadas de Debian (como Ubuntu) el comando especial es `apt-get`, en Red Hat es yum y en ArchLinux es `pacman`. Cada distribución tiene su comando especial y debes averiguar cómo funciona para poder instalar Git. Antes de hacer la instalación, debemos hacer una actualización del sistema. En nuestro caso, los comandos para hacerlo son `sudo apt-get update` y `sudo apt-get upgrade`. Con el sistema actualizado, ahora sí podemos instalar Git y, en este caso, el comando para hacerlo es `sudo apt-get install git`. También puedes verificar que Git fue instalado correctamente con el comando `git --version`.

3. Introducción a la terminal y línea de comandos

Diferencias entre la estructura de archivos de Windows, Mac o Linux: La ruta principal en Windows es `C:\`, en UNIX es solo `/`. Windows no hace diferencia entre mayúsculas y minúsculas pero UNIX sí. Recuerda que GitBash usa la ruta `/c` para dirigirse a `C:\` (o `/d` para dirigirse a `D:\`) en Windows. Por lo tanto, la ruta del usuario con el que estás trabajando es `/c/Users/Nombre de tu usuario` Comandos básicos en la terminal:

- `pwd`: Nos muestra la ruta de carpetas en la que te encuentras ahora mismo.
- `mkdir`: Nos permite crear carpetas (por ejemplo, `mkdir Carpeta-Importante`).
- `touch`: Nos permite crear archivos (por ejemplo, `touch archivo.txt`).
- `rm`: Nos permite borrar un archivo o carpeta (por ejemplo, `rm archivo.txt`). Mucho cuidado con este comando, puedes borrar todo tu disco duro.

- `cat`: Ver el contenido de un archivo (por ejemplo, `cat nombre-archivo.txt`).
- `ls`: Nos permite cambiar ver los archivos de la carpeta donde estamos ahora mismo. Podemos usar uno o más argumentos para ver más información sobre estos archivos (los argumentos pueden ser `--` + *el nombre del argumento* o `-` + *una sola letra* o *shortcut* por cada argumento).
  - `ls -a`: Mostrar todos los archivos, incluso los ocultos.
  - `ls -l`: Ver todos los archivos como una lista.
- `cd`: Nos permite navegar entre carpetas.
  - `cd /`: Ir a la ruta principal:
  - `cd` o `cd ~`: Ir a la ruta de tu usuario
  - `cd carpeta/subcarpeta`: Navegar a una ruta dentro de la carpeta donde estamos ahora mismo.
  - `cd ..` (`cd` + *dos puntos*): Regresar una carpeta hacia atrás.
  - Si quieres referirte al directorio en el que te encuentras ahora mismo puedes usar `cd .` (`cd` + *un punto*).
- `history`: Ver los últimos comandos que ejecutamos y un número especial con el que podemos repetir su ejecución.
- `!` + *número*: Ejecutar algún comando con el número que nos muestra el comando `history` (por ejemplo, `!72`).
- `clear`: Para limpiar la terminal. También podemos usar los atajos de teclado `Ctrl + L` o `Command + L`.

Todos estos comandos tiene una función de autocompletado, o sea, puedes escribir la primera parte y presionar la tecla `Tab\verb` para que la terminal nos muestre todas las posibles carpetas o comandos que podemos ejecutar. Si presionas la tecla Arriba puedes ver el último comando que ejecutamos. Recuerda que podemos descubrir todos los argumentos de un comando con el argumento `--help` (por ejemplo, `cat --help`).

4. ¿Qué es el *staging* y los repositorios? Ciclo básico de trabajo en Git

Para iniciar un repositorio, o sea, activar el sistema de control de versiones de Git en tu proyecto, solo debes ejecutar el comando `git init`.
Este comando se encargará de dos cosas:

1. Crear una carpeta .git, donde se guardará toda la base de datos con cambios atómicos de nuestro proyecto; y segundo, crear un área que conocemos como Staging, que guardará temporalmente nuestros archivos (cuando ejecutemos un comando especial para eso)

2. Nos permitirá, más adelante, guardar estos cambios en el repositorio (también con un comando especial).

4.1. Ciclo de vida o estados de los archivos en Git:

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

1. **Archivos *Tracked*:** son los archivos que viven dentro de Git, no tienen cambios pendientes y sus últimas actualizaciones han sido guardadas en el repositorio gracias a los comandos `git add` y `git commit`.

2. **Archivos *Staged*:** son archivos en *Staging*. Viven dentro de Git y hay registro de ellos porque han sido afectados por el comando `git add`, aunque no sus últimos cambios. Git ya sabe de la existencia de estos últimos cambios, pero todavía no han sido guardados definitivamente en el repositorio porque falta ejecutar el comando `git commit`.

3. **Archivos *Unstaged*:** entiéndelos como archivos *"Tracked pero Unstaged"*. Son archivos que viven dentro de Git pero no han sido afectados por el comando `git add` ni mucho menos por `git commit`. Git tiene un registro de estos archivos, pero está desactualizado, sus últimas versiones solo están guardadas en el disco duro.

4. **Archivos *Untracked*:** son archivos que NO viven dentro de Git, solo en el disco duro. Nunca han sido afectados por git add, así que Git no tiene registros de su existencia.

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

5. Comandos para mover archivos entre los estados de Git

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

6. ¿Qué es un *Branch* (rama) y cómo funciona un *Merge* en Git?

Git es una base de datos muy precisa con todos los cambios y crecimiento que ha tenido nuestro proyecto. Los *commits* son la única forma de tener un registro de los cambios. Pero las ramas amplifican mucho más el potencial de Git. Todos los *commits* se aplican sobre una rama. Por defecto, siempre empezamos en la rama master (pero puedes cambiarle el nombre si no te gusta) y creamos nuevas ramas, a partir de esta, para crear flujos de trabajo independientes. Crear una nueva rama se trata de copiar un *commit* (de cualquier rama), pasarlo a otro lado (a otra rama) y continuar el trabajo de una parte específica de nuestro proyecto sin afectar el flujo de trabajo principal (que continúa en la rama *master* o la rama *principal*). **Master:** Esta es la rama donde se inicia siempre por defecto (se le puede asignar cualquier nombre), a partir de esta rama se crean todas las ramas, en esta rama estan todos los cambios de tus archivos. Cada vez que haces una nueva rama lo único que se hace es que copias un *commit*, que pasa a la rama de a lado, en la cual puedes continuar trabajando sin afectar el flujo de trabajo principal que esta en la rama *master*.

Los equipos de desarrollo tienen un estándar: Todo lo que esté en la rama master va a producción, las nuevas *features*, características y experimentos van en una rama "*development*" (para unirse a *master* cuando estén definitivamente listas) y los *issues* o errores se solucionan en una rama "*hotfix*" para unirse a *master* tan pronto como sea posible.

- **Development (o rama experimental):** Esta es una versión copiada de la rama *master*, donde experimentas, observws características, *features* entre otros. Para finalmente unirlas a la rama master cuando estén listas.
- **Bugfixing o Hotfix:** Esta rama se encarga de repara cualquier *bug*, *issues* (errores) o fallas que surjan en la rama experimental, aquí puedes hacer cualquier clase de cambios para después probarlo en tu rama *master*.

Crear una nueva rama lo conocemos como **Checkout**. Unir dos ramas lo conocemos como **Merge**. *Merge* es una brecha que une dos ramas. Podemos crear todas las ramas y *commits* que queramos. De hecho, podemos aprovechar el registro de cambios de Git para crear ramas, traer versiones viejas del código, arreglarlas y

combinarlas de nuevo para mejorar el proyecto. Solo ten en cuenta que combinar estas ramas (sí, hacer "*merge*") puede generar conflictos. Algunos archivos pueden ser diferentes en ambas ramas. Git es muy inteligente y puede intentar unir estos cambios automáticamente, pero no siempre funciona. En algunos casos, somos nosotros los que debemos resolver estos conflictos "*a mano*". **Head:** Es a donde llevás todo cuando ya tienes tu versión final.
Estándar común entre desarrolladores de git:

- **Master** – Para producción.
- **Feature** – Para las nuevas características.
- **Development** – Para las pruebas o experimentos.
- **Hotfix** – Para solucionar los errores y unirlas a la rama master.

El flujo de **Gitflow** es así:

- En la rama *master* tendremos solo lo que se ha liberado.
    1. Se crea la rama *develop*, es la rama en la que estamos trabajando (lo que vamos a liberar).
    2. Liberar a producción con tu equipo de trabajo se crea una *release* desde *develop*. No se pasa directo de *develop* a *master*, **Git Flow** crea la nueva rama de *release*.
    3. Por cada petición o tarea se genera una rama llamada *feature* a partir de *develop*.
    4. Por ejemplo una pantalla nueva, se crea y está completa el *feature* de pantalla se cierra y se afusiona con *develop*.
    5. Cuando tienes la rama *release* terminada, fusionas con *develop* y *master*.
    6. Si hay problema en master se crea *hotfix* que son los cambios sobre algo que está en producción.
    7. Se crea una nueva rama se trabaja y se reintegra. Una vez que *hotfix* se completa, se fusiona a ambos *develop* y *master*.

La rama/repositorio por defecto de git es *master*. Cada *commit* es una versión en git, y cada versión tiene su código identificador.

7. Crea un repositorio de Git y haz tu primer commit

Le indicaremos a Git que queremos crear un nuevo repositorio para utilizar su sistema de control de versiones. Solo debemos posicionarnos en la carpeta raíz de nuestro proyecto y ejecutar el comando `git init`. Recuerda que al ejecutar este comando (y de aquí en adelante) vamos a tener una nueva carpeta oculta llamada `.git` con toda la base de datos con cambios atómicos en nuestro proyecto. Recuerda que Git está optimizado para trabajar en equipo, por lo tanto, debemos darle un poco de información sobre nosotros. No debemos hacerlo todas las veces que ejecutamos un comando, basta con ejecutar solo una sola vez los siguientes comandos con tu información:

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

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

8. Comandos Git utilizados

- `git init` Inicia Git dentro de la carpeta raíz de nuestro proyecto. Inicia el repositorio y crea la carpeta (`.git`). En la carpeta en la estamos ubicados. lo usamos para determinar la carpeta en la que vamos a trabajar.
- `git status` lo usamos para saber si tenemos un archivo añadido o borrado en nuestro proyecto, para saber en la rama en la que estamos y si tenemos *commits*.
- `git add myfile.txt` Agrega el archivo en *staging*. No añade el archivo a la rama, solo lo añade a la base de datos de `git commit` si lo añade a la rama. es para añadir un archivo a nuestra rama seguidamente ponemos entre comillas el nombre de nuestro archivo o poner un punto para añadir todos los archivos de nuestra carpeta.

    - `git add .`] Sube todo los archivos de la carpeta actual
- `git rm` lo usamos para borrar un archivo que hayamos añadido, para eliminarlo por completo de nuestra rama usamos `git rm --cached`.
    - `git rm -- cached _myfile.txt_`] Elimina el archivo de la *staging* sin eliminarlo de la carpeta
- `git commit -m "Primer commit de este archivo"`] se usa para añadir un *commit* a nuestra rama, también podemos ponerle un `-m` seguidamente ponemos entre comillas nuestro ensaje. Sube el archivo al repositorio.
- `git config`] ver la configuración actual de git. Muestra configuraciones de git también podemos usar `-list` para mostrar la configuración por defecto de nuestro git y si añadimos `--show-origin` nos muestra las configuraciones guardadas y su ubicación.
    - `git config --global user.name "Tu Nombre"` cambia de manera global el nombre del usuario, seguidamente ponemos entre comillas nuestro nombre.
    - `git config --global user.email "tu@email.com"` cambia de manera global el email del usuario, seguidamente ponemos entre comillas nuestro email.
- `git log` se usa para ver la historia de nuestros archivos, los *commits*, el usuario que lo cambió, cuando se realizaron los cambios etc., seguidamente ponemos el nombre de nuestro archivo.
- `git code` Abre VSCode en Windows
- `git status` Muestra el estado actual de la base de datos
- `git log _my file.txt_` Puede ver todas las modificaciones del archivo así como ver quien las hizo
    - `git log --oneline -decorate` Puedes ver los *commits* y los *merge* de manera gráfica.
- `git show` este comando muestra:
    - SHA (Secure Hash Algorithm) de último commit
    - Quién y cuándo realizó el último commit.
    - Mensaje de último *commit*
    - Hace un `diff` (comando) de la versión anterior **(a)** con la versión nueva **(b)**, para mostrar las diferencias entre versión **a** y versión **b**

9. Volver en el tiempo en nuestro repositorio utilizando *reset* y *checkout*

El comando `git checkout` + *ID* del *commit* nos permite viajar en el tiempo. Podemos volver a cualquier versión anterior de un archivo específico o incluso del proyecto entero. Esta también es la forma de crear ramas y movernos entre ellas.

- También hay una forma de hacerlo un poco más **"ruda"**: usando el comando `git reset`. En este caso, no solo *"volvemos en el tiempo"*, sino que borramos los cambios que hicimos después de este *commit*. Hay dos formas de usar git reset:

    1. Con el argumento `--hard`, borrando toda la información que tengamos en el área de *staging* (y perdiendo todo para siempre).
    2. Un poco más seguro, con el argumento `--soft`, que mantiene allí los archivos del área de *staging* para que podamos aplicar nuestros últimos cambios pero desde un *commit* anterior.

9.1. Git reset vs. Git rm

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

- `git rm`: Este comando nos ayuda a eliminar archivos de Git sin eliminar su historial del sistema de versiones. Esto quiere decir que si necesitamos recuperar el archivo solo debemos *viajar en el tiempo* y recuperar el último *commit* antes de borrar el archivo en cuestión. Recuerda que `git rm` no puede usarse así no más. Debemos usar uno de los *flags* para indicarle a Git cómo eliminar los archivos que ya no necesitamos en la última versión del proyecto:
    - `git rm --cached`: Elimina los archivos del área de *Staging* y del próximo *commit* pero los mantiene en nuestro disco duro.

    - `git rm --force`: Elimina los archivos de Git y del disco duro. Git siempre guarda todo, por lo que podemos acceder al registro de la existencia de los archivos, de modo que podremos recuperarlos si es necesario (pero debemos usar comandos más avanzados).
- `git reset`: Este comando nos ayuda a volver en el tiempo. Pero no como `git checkout` que nos deja ir, mirar, pasear y volver. Con `git reset` volvemos al pasado sin la posibilidad de volver al futuro. Borramos la historia y la debemos sobreescribir. No hay vuelta atrás.

Este comando es muy peligroso y debemos usarlo solo en caso de emergencia. Recuerda que debemos usar alguna de estas dos opciones: Hay dos formas de usar `git reset`: con el argumento `--hard`, borrando toda la información que tengamos en el área de staging (y perdiendo todo para siempre). O, un poco más seguro, con el argumento `--soft`, que mantiene allí los archivos del área de staging para que podamos aplicar nuestros últimos cambios pero desde un *commit* anterior.

- `git reset --soft`: Borramos todo el historial y los registros de Git pero guardamos los cambios que tengamos en Staging, así podemos aplicar las últimas actualizaciones a un nuevo *commit*.
- `git reset --hard`: Borra todo. Todo todito, absolutamente todo. Toda la información de los *commits* y del área de staging se borra del historial.
- `git reset HEAD`: Este es el comando para sacar archivos del área de *Staging*. No para borrarlos ni nada de eso, solo para que los últimos cambios de estos archivos no se envíen al último *commit*, a menos que cambiemos de opinión y los incluyamos de nuevo en *staging* con `git add`, por supuesto.

  ¿Por qué esto es importante?: Imagina el siguiente caso:
  Hacemos cambios en los archivos de un proyecto para una nueva actualización. Todos los archivos con cambios se mueven al área de *staging* con el comando `git add`. Pero te das cuenta de que uno de esos archivos no está listo todavía. Actualizaste el archivo pero ese cambio no debe ir en el próximo *commit* por ahora. **¿Qué podemos hacer?** Bueno, todos los cambios están en el área de *Staging*, incluido el archivo con los cambios que no están listos. Esto significa que debemos sacar ese archivo de *Staging* para poder hacer *commit* de todos los demás. ¡Al usar `git rm` lo que haremos será eliminar este archivo completamente de git! Todavía tendremos el historial de cambios de este archivo, con la eliminación del archivo como su última actualización. Recuerda que en este caso no buscábamos eliminar un archivo, solo dejarlo como estaba y actualizarlo después, no en este *commit*. En cambio, si usamos `git reset HEAD`, lo único que haremos será mover estos cambios de *Staging* a *Unstaged*. Seguiremos teniendo los últimos cambios del archivo, el repositorio mantendrá el archivo (no con sus últimos cambios pero sí con los últimos en los que hicimos *commit*) y no habremos perdido nada.
  Conclusión: Lo mejor que puedes hacer para salvar tu puesto y evitar un incendio en tu trabajo es conocer muy bien la diferencia y los riesgos de todos los comandos de Git.

10. Introducción a las ramas o *branches* de Git

Las ramas son la forma de hacer cambios en nuestro proyecto sin afectar el flujo de trabajo de la rama principal. Esto porque queremos trabajar una parte muy específica de la aplicación o simplemente experimentar. La cabecera o **HEAD** representan la rama y el *commit* de esa rama donde estamos trabajando. Por defecto, esta cabecera aparecerá en el último *commit* de nuestra rama principal. Pero podemos cambiarlo al crear una rama (`git branch rama`, `git checkout -b rama`) o movernos en el tiempo a cualquier otro *commit* de cualquier otra rama con los comandos (`git reset id-commit`, `git checkout rama-o-id-commit`).

Un comando super útil que estuve utilizando mucho últimamente es:

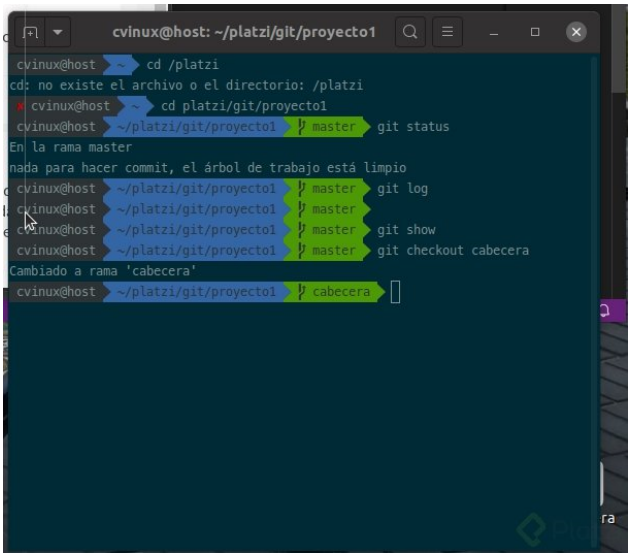
```
git checkout -b "nombre_rama"
```

Este comando es una fusión entre `git branch` y `git checkout`, y su funcionamiento es el siguiente:

- Crea una rama llamada `nombre_rama`
- Hace un checkout de la rama `nombre_rama`

En Ubuntu o distros Linux derivados de Debian, recomiendo usar **Oh My Zsh** para poder tener mejor visibilidad de los comando y ramas de Git Para instalar **Zsh** en Linux utilice los siguientes comandos:

```
sudo apt-get install zsh
sudo apt-get install git-core
```




```
wget https://github.com/robbyrussell/oh-my-zsh/raw/master/tools/install.sh -O - | zsh
chsh -s 'which zsh'
```

11. Fusión de ramas con Git merge

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

```
# Crear un nuevo commit en la rama master combinando
# los cambios de la rama cabecera:
git checkout master
git merge cabecera
```

Ahora,

```
# Crear un nuevo commit en la rama cabecera combinando
# los cambios de cualquier otra rama:
git checkout cabecera
git merge cualquier-otra-rama
```

Asombroso, ¿verdad? Es como si Git tuviera super poderes para saber qué cambios queremos conservar de una rama y qué otros de la otra. El problema es que no siempre puede adivinar, sobretodo en algunos casos donde dos ramas tienen actualizaciones diferentes en ciertas líneas en los archivos. Esto lo conocemos como un conflicto y aprenderemos a solucionarlos en la siguiente clase. Recuerda que al ejecutar el comando `git checkout` para cambiar de rama o commit puedes perder el trabajo que no hayas guardado. Guarda tus cambios antes de hacer `git checkout`.

12. Resolución de conflictos al hacer un merge

Git nunca borra nada a menos que nosotros se lo indiquemos. Cuando usamos los comandos `git merge` o `git checkout` estamos cambiando de rama o creando un nuevo commit, no borrando ramas ni *commits* (recuerda que puedes borrar *commits* con `git reset` y ramas con `git branch -d`). Git es muy inteligente y puede resolver algunos conflictos automáticamente: cambios, nuevas líneas, entre otros. Pero algunas veces no sabe cómo resolver estas diferencias, por ejemplo, cuando dos ramas diferentes hacen cambios distintos a una misma línea. Esto lo conocemos como conflicto y lo podemos resolver manualmente, solo debemos hacer el *merge*, ir a nuestro editor de código y elegir si queremos quedarnos con alguna de estas dos versiones o algo diferente. Algunos editores de código como **VSCode** nos ayudan a resolver estos conflictos sin necesidad de borrar o escribir líneas de texto, basta con hundir un botón y guardar el archivo. Recuerda que siempre

debemos crear un nuevo commit para aplicar los cambios del *merge*. Si Git puede resolver el conflicto hará commit automáticamente. Pero, en caso de no pueda resolverlo, debemos solucionarlo y hacer el *commit*. Los archivos con conflictos por el comando `git merge` entran en un nuevo estado que conocemos como *Unmerged*. Funcionan muy parecido a los archivos en estado *Unstaged*, algo así como un estado intermedio entre *Untracked* y *Unstaged*, solo debemos ejecutar `git add` para pasarlos al área de staging y `git commit` para aplicar los cambios en el repositorio. Hasta los mejores cometen errores, pero todo tiene solución. Se olvidó de guardar el archivo `.html`, y lo malo es que *commiteó* ese archivo todavía con el *merge conflict*. Con los:

```
<<<<<<< HEAD
=======
>>>>>>> cabecera
```

Si les pasa esto, no entren en pánico, mientras el *commit* siga en nuestro repositorio local. Tienen dos formas al menos de solucionarlo, ustedes eligen la que mas les convenga:

- Reemplazar el commit fallido: Pueden corregir el archivo (en este caso hubiera bastado con guardarlo) y luego usar

```
git add .
git commit --amend
```

El `--amend` *suma* al *commit* anterior el nuevo contenido en el *staging area*. Así que no se está creando un nuevo *commit*, si no reemplazando el último.

- Borrar el *commit*, sin perder los cambios, arreglar y *commitear* de nuevo. Bastaría con devolver el commit a la *staging area*, usando:

```
  git reset HEAD~1 --soft
```

y luego guardar el archivo y volver a hacer el *commit* (ya que esta vez con el *reset* si borraron el *commit* con el fallo). `commit --amend` resulta bastante útil, más de una vez habré *commiteado* algo, y luego me doy cuenta que me faltó algo, o algo está mal, y en vez de armar otro *commit* arreglando el error, podemos reemplazarlo con `amend`.