

Curso de GIT y control de versiones v0.5

Marco Teran

Índice

1. ¿Qué es Git?	2
1.1. GitHub	2
1.2. ¿Por qué usar un sistema de control de versiones como Git?	2
2. Instalando Git y GitBash	2
2.1. Instalando Git en Linux	2
3. Introducción a la terminal y línea de comandos	3
4. ¿Qué es el <i>staging</i> y los repositorios? Ciclo básico de trabajo en Git	3
4.1. Ciclo de vida o estados de los archivos en Git:	4
5. Comandos para mover archivos entre los estados de Git	4
6. ¿Qué es un <i>Branch</i> (rama) y cómo funciona un <i>Merge</i> en Git?	4
7. Crea un repositorio de Git y haz tu primer commit	5
8. Comandos Git utilizados	6
9. Volver en el tiempo en nuestro repositorio utilizando <i>reset</i> y <i>checkout</i>	6
9.1. Git reset vs. Git rm	7
10. Introducción a las ramas o <i>branches</i> de Git	8
11. Fusión de ramas con Git merge	8
12. Resolución de conflictos al hacer un merge	9
13. Laboratorio: Introducción a Git y control de versiones en un proyecto de IoT	9

Resumen

Este manual proporcionará a los usuarios una guía básica para trabajar con Git y control de versiones en proyectos de desarrollo de IoT. Los conceptos y prácticas esenciales de Git se presentarán de manera clara y sencilla para ayudar a los usuarios a comprender cómo utilizar Git para el control de versiones y el seguimiento de cambios en sus proyectos.

- Aprender los conceptos fundamentales del control de versiones con Git y su utilidad en proyectos de IoT.
- Comprender cómo utilizar Git para crear y trabajar con repositorios y ramas, realizar seguimiento de cambios y revertir cambios no deseados.
- Adquirir las habilidades necesarias para colaborar en equipo y resolver conflictos en proyectos de IoT utilizando Git.

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`.

El modelo SVN (Subversion) es un sistema centralizado de control de versiones que mantiene un solo repositorio central donde se almacena todo el historial de cambios. Las principales diferencias entre Git y SVN son que Git es un sistema de control de versiones distribuido, mientras que SVN es centralizado; Git es más rápido y flexible, y permite trabajar sin conexión a internet y con múltiples ramas locales, mientras que SVN es más limitado en cuanto a la gestión de ramas y fusiones.

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, observas características, *features* entre otros. Para finalmente unirlas a la rama *master* cuando estén listas.
- **Bugfixing o Hotfix:** Esta rama se encarga de reparar 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 llevas 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:

```
1 git config --global user.email "tu@email.com"
2 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. Borrarnos la historia y la debemos sobrescribir. 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`: Borrarnos 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`).

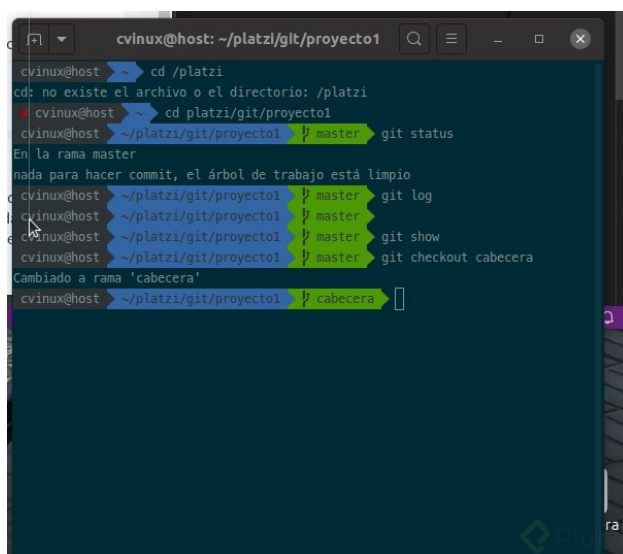
Un comando super útil que estuve utilizando mucho últimamente es:

```
1 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:



```
cvlinux@host: ~/platzi/git/proyecto1
cvlinux@host ➤ cd /platzi
cd: no existe el archivo o el directorio: /platzi
cvlinux@host ➤ cd platzi/git/proyecto1
cvlinux@host ➤ ~/platzi/git/proyecto1 master git status
En la rama master
nada para hacer commit, el árbol de trabajo está limpio
cvlinux@host ➤ ~/platzi/git/proyecto1 master git log
cvlinux@host ➤ ~/platzi/git/proyecto1 master git show
cvlinux@host ➤ ~/platzi/git/proyecto1 master git checkout cabecera
Cambiado a rama 'cabecera'
cvlinux@host ➤ ~/platzi/git/proyecto1 cabecera
```

```
1 sudo apt-get install zsh
2 sudo apt-get install git-core
3 wget https://github.com/robbyrussell/oh-my-zsh/raw/master/tools/install.sh -O - | zsh
4 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).

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

Ahora,

```
1 # Crear un nuevo commit en la rama cabecera combinando
2 # los cambios de cualquier otra rama:
3 git checkout cabecera
4 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 **VSCo** 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:

```
1 <<<<<< HEAD
2 =====
3 >>>>>> 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

```
1 git add .
2 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:

```
1 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`.

13. Laboratorio: Introducción a Git y control de versiones en un proyecto de IoT

Laboratorio paso a paso: Control de versiones con Git y Arduino

En este laboratorio, aprenderás los conceptos básicos de Git y cómo aplicarlos en el desarrollo de un proyecto de IoT utilizando un web server sencillo en Arduino para una ESP32. En este laboratorio, aprenderás

a utilizar Git para el control de versiones en un proyecto de Arduino. Crearás un repositorio de Git y realizarás las configuraciones necesarias. Desarrollarás un código que leerá e imprimirá los datos de humedad y temperatura de un DHT-11 a través del puerto serial. Luego, crearás una nueva rama para agregar un servidor web que mostrará los datos del DHT-11 en una página web. Finalmente, fusionarás ambas ramas y resolverás cualquier conflicto que surja en el proceso. En este laboratorio aprenderás los conceptos básicos de Git y cómo aplicarlos en un proyecto de Arduino. El proyecto consiste en un web server sencillo en arduino para una ESP32 para mostrar la humedad y temperatura de un DHT-11. Se mostrarán los datos en la web y el puerto serial.

Objetivos Crear un repositorio de Git para el proyecto de Arduino. Configurar el repositorio para el seguimiento de archivos y la solución de conflictos. Utilizar las funcionalidades básicas de Git para crear y fusionar ramas. Solucionar conflictos en la fusión de ramas. Trabajar en equipo en un proyecto de Arduino utilizando Git.

Requisitos previos Conocimientos básicos de programación en Arduino Git instalado en tu computadora

Materiales: Una placa ESP32 con conexión WiFi Un sensor DHT-11 Un cable USB para programar la ESP32 Una computadora con el software de Arduino IDE y Git instalados Computadora con Git y Arduino IDE instalados. ESP32 con un sensor DHT11.

Pasos: Crear un repositorio de Git para el proyecto en la computadora. Configurar el repositorio de Git para trabajar con Arduino. Escribir el código para leer e imprimir los datos del DHT-11 a través del puerto serial. Realizar los commits necesarios para guardar los cambios en el repositorio. Crear una nueva rama para agregar el servidor web. Escribir el código para mostrar los datos del DHT-11 en una página web. Realizar los commits necesarios para guardar los cambios en la nueva rama. Fusionar ambas ramas y resolver cualquier conflicto que surja en el proceso. Probar el código completo y hacer los ajustes necesarios.

A lo largo del laboratorio, seguirás los siguientes pasos:

1. Creación del repositorio en GitHub: Entra en GitHub y crea un nuevo repositorio. Nombra el repositorio como "web-server-esp32". 2. Clonar el repositorio: Clona el repositorio recién creado en tu equipo con el siguiente comando:

```
1 git clone https://github.com/TU_USUARIO/web-server-esp32.git
```

3. Configurar el repositorio: Configura tus credenciales en Git con los siguientes comandos:

```
1 git config --global user.name "TU_NOMBRE"
2 git config --global user.email "TU_EMAIL"
```

Agregar y confirmar los cambios: Agrega y confirma los cambios realizados en el proyecto utilizando los siguientes comandos:

```
1 git add .
2 git commit -m "Mensaje de confirmación de cambios"
```

Verificación de cambios: Verifica los cambios realizados en el repositorio utilizando el siguiente comando:

```
1 git status
```

Pull: Si trabajas en equipo y alguien más ha subido cambios al repositorio, realiza un pull para sincronizar tu repositorio local con el remoto:

```
1 git pull
```

Crear una rama: Crea una nueva rama para trabajar en ella con el siguiente comando:

```
1 git branch NOMBRE_DE_LA_RAMIFICACIÓN
```

Cambiar de rama: Cambia a la rama creada con el siguiente comando:

```
1 git checkout NOMBRE_DE_LA_RAMIFICACIÓN
```

9. Realiza cambios en la rama creada: Realiza cambios en el código en la rama creada. 10. Fusionar ramas: Fusiona la rama creada con la rama principal utilizando el siguiente comando:

```
1 git merge NOMBRE_DE_LA_RAMIFICACIÓN
```

Solución de conflictos: Si hay conflictos entre los cambios realizados en diferentes ramas, resuélvelos utilizando el siguiente comando:

```
1 git mergetool
```

Devolver a estado anterior: Si cometes un error y necesitas volver a un estado anterior, utiliza el siguiente comando:

```
1 git checkout -- NOMBRE_DEL_ARCHIVO
```

Realiza un push: Sube los cambios realizados al repositorio remoto utilizando el siguiente comando:

```
1 git push origin NOMBRE_DE_LA_RAMIFICACIÓN
```

Conclusión: En este laboratorio, aprendiste los conceptos básicos de Git y cómo aplicarlos en un proyecto de IoT utilizando un web server sencillo en Arduino para una ESP32. Aprendiste a crear un repositorio, clonarlo, realizar cambios, verificar cambios, hacer un pull, trabajar en ramas, fusionar ramas, solucionar conflictos, volver a un estado anterior y hacer push. Estas son habilidades importantes para cualquier desarrollador de IoT que trabaje en proyectos colaborativos y necesite controlar y gestionar el versionado de su código. Resultado: Al final de este laboratorio, tendrás un proyecto de Arduino que leerá e imprimirá los datos de humedad y temperatura de un DHT-11 a través del puerto serial, y mostrará estos datos en una página web utilizando un servidor web. Además, habrás aprendido a utilizar Git para el control de versiones de tu proyecto, lo que te permitirá mantener un registro de todos los cambios y revertirlos si es necesario.

Paso 1: Crear el repositorio de Git Abrir la línea de comandos y crear un directorio para el proyecto. Dentro del directorio, ejecutar git init para crear el repositorio. Crear un archivo .gitignore y agregar las carpetas y archivos que no se deben seguir (por ejemplo, archivos compilados o temporales). Paso 1: Crear un repositorio en Git Crea un nuevo repositorio en Git y nómbralo .arduino-dht11-web-server Clona el repositorio en tu computadora

Paso 2: Configurar el repositorio de Git Configurar el nombre y correo electrónico del usuario con los siguientes comandos:

```
1 git config --global user.name "Nombre del usuario"
2 git config --global user.email "Correo electrónico del usuario"
```

Agregar los archivos del proyecto al repositorio con git add . Hacer el primer commit con git commit -m "Primer commit". Paso 2: Configurar el repositorio Abre la carpeta del repositorio en tu terminal Configura tu nombre de usuario y correo electrónico con los siguientes comandos:

```
1 git config --global user.name "Tu nombre"
2 git config --global user.email "tu_correo@ejemplo.com"
```

Paso 3: Crear la rama para el sensor DHT11 Crear una nueva rama con git branch sensor_dht11. Cambiar a la rama creada con git checkout sensor_dht11. Escribir el código para leer e imprimir los datos del sensor en un archivo llamado sensor_dht11.ino. Agregar el archivo creado al repositorio con git add sensor_dht11.ino. Hacer el commit con git commit -m .^Agregado código para el sensor DHT11".

Paso 3: Crear el código de la función del DHT11 Crea una nueva rama llamada "dht11-funcion Crea un archivo llamado "dht11_funcion.ino" escribe el código para leer e imprimir los datos del DHT11 en el puerto serial:

```
1 #include "DHT.h"
2
3 #define DHTPIN 4
4 #define DHTTYPE DHT11
5
6 DHT dht(DHTPIN, DHTTYPE);
7
8 void setup() {
9   Serial.begin(9600);
10  dht.begin();
11 }
12
13 void loop() {
14   delay(2000);
15
16   float h = dht.readHumidity();
17   float t = dht.readTemperature();
18
```

```

19 Serial.print("Humedad: ");
20 Serial.print(h);
21 Serial.print(" %\t");
22 Serial.print("Temperatura: ");
23 Serial.print(t);
24 Serial.println(" *C");
25 }

```

Agrega los cambios y haz un commit con el mensaje Creación de función para lectura e impresión de datos del DHT11. Realiza un push a la rama "dht11-funcion" en el repositorio remoto.

```

1 #include "WiFi.h"
2 #include "WebServer.h"
3 #include "DHT.h"
4
5 #define DHTPIN 4
6 #define DHTTYPE DHT11
7
8 DHT dht(DHTPIN, DHTTYPE);
9
10 WebServer server(80);
11
12 const char* ssid = "nombre_de_tu_red";
13 const char* password = "contraseña_de_tu_red";
14
15 void handleRoot() {
16     float h = dht.readHumidity();
17     float t = dht.readTemperature();
18
19     String html = "<html><body>";
20     html += "<p>Humedad: " + String(h) + " %</p>";
21     html += "<p>Temperatura: " + String(t) + "

```

Paso 4: Crear la rama para el web server. Cambiar a la rama principal con git checkout master. Crear una nueva rama con git branch web_server. Cambiar a la rama creada con git checkout web_server. Escribir el código para el web server en un archivo llamado web_server.ino. Agregar el archivo creado al repositorio con git add web_server.ino. Hacer el commit con git commit -m "Agregado código para el web server". Paso 4: Crear el código del web server. Cambia a la rama "master" con el comando git checkout master. Crea una nueva rama llamada "web-server". Crea un archivo llamado "web_server.ino" y escribe el código para crear el web server y mostrar los datos del DHT11:

Paso 5: Fusionar las ramas. Cambiar a la rama principal con git checkout master. Fusionar la rama del sensor DHT11 con la rama principal con git merge sensor_dht11. Fusionar la rama del web server con la rama principal con git merge web_server.

Paso 6: Solucionar conflictos. Simular un conflicto editando la misma línea en ambas ramas. Cambiar a la rama principal con git checkout master. Fusionar la rama del sensor DHT11 con la rama principal con git merge sensor_dht11. Resolver el conflicto manualmente editando el archivo en conflicto. Agregar los cambios al crear la rama "web-server": En la terminal de Git, escribir el comando "git branch web-server" luego "git checkout web-server" para cambiar a la rama "web-server". Crear el código del web server: Crear un nuevo archivo llamado "web_server.ino" en el directorio del proyecto y agregar el siguiente código:

```

1 #include <WiFi.h>
2 #include <WebServer.h>
3 #include "DHT.h"
4
5 #define DHTPIN 4
6 #define DHTTYPE DHT11
7
8 const char* ssid = "TU_SSID";
9 const char* password = "TU_CLAVE_WIFI";
10
11 DHT dht(DHTPIN, DHTTYPE);
12 WebServer server(80);
13
14 void handleRoot() {
15     float hum = dht.readHumidity();

```

```

16 float temp = dht.readTemperature();
17
18 String html = "<html><body>";
19 html += "<h1>Temperatura y Humedad</h1>";
20 html += "<p>Temperatura: " + String(temp) + " °C</p>";
21 html += "<p>Humedad: " + String(hum) + " %</p>";
22 html += "</body></html>";
23
24 server.send(200, "text/html", html);
25 }
26
27 void setup() {
28   Serial.begin(115200);
29   dht.begin();
30
31   WiFi.begin(ssid, password);
32
33   while (WiFi.status() != WL_CONNECTED) {
34     delay(1000);
35     Serial.println("Connecting to WiFi...");
36   }
37
38   Serial.println("WiFi connected");
39   Serial.println("IP address: ");
40   Serial.println(WiFi.localIP());
41
42   server.on("/", handleRoot);
43
44   server.begin();
45 }
46
47 void loop() {
48   server.handleClient();
49 }

```

Hacer commit de los cambios en la rama "web-server": En la terminal de Git, escribir el comando "git add ." para agregar todos los cambios al stage, luego "git commit -m 'Agregado el código del web server'" para hacer commit de los cambios en la rama "web-server".

Fusionar la rama "web-server" con la rama "main": En la terminal de Git, escribir el comando "git checkout main" para cambiar a la rama "main", luego "git merge web-server" para fusionar la rama "web-server" con la rama "main".

Resolver conflictos en el merge: Si hay conflictos en el merge, Git mostrará un mensaje indicando que debe resolver los conflictos manualmente. Abrir los archivos en conflicto, buscar las secciones en conflicto y decidir cuál es la versión correcta. Eliminar las marcas de conflicto y guardar los cambios.

Hacer commit de la fusión: En la terminal de Git, escribir el comando "git commit -m 'Fusionada la rama web-server con la rama main'" para hacer commit de la fusión.

Verificar que todo funciona correctamente: Subir el código a la ESP32 y verificar que el servidor web funciona correctamente y muestra la temperatura y humedad.

Este es un ejemplo básico de cómo utilizar Git para controlar versiones de un proyecto de hardware. Hay muchas otras funcionalidades y comandos que se pueden utilizar en Git, pero este laboratorio proporciona una introducción sólida a los conceptos básicos y su aplicación práctica.

Laboratorio: Control de versiones con Git y Arduino

Objetivos Crear un repositorio Git y realizar un seguimiento de los cambios realizados en el código de Arduino para el proyecto del web server con la ESP32. Crear una rama para la creación de la función que lee e imprime datos con el DHT11 y otra para el web server que muestra los datos. Fusionar las dos ramas y solucionar cualquier conflicto que surja durante el merge.

Materiales Computadora con acceso a Internet. Placa ESP32. Sensor de temperatura y humedad DHT11. Cables de conexión. Arduino IDE instalado. Git instalado.

Pasos Crear un repositorio de Git para el proyecto.

```
1 $ git init
```

Crear un archivo .gitignore para ignorar los archivos de compilación y otros archivos innecesarios.

```

1 *.bin
2 *.ino~
3 *.DS_Store
4 .vscode

```

Crear una rama para la creación de la función que lee e imprime datos con el DHT11.

```
1 $ git checkout -b dht11
```

Crear un archivo dht11.ino y agregar el código para leer e imprimir los datos del sensor DHT11

```

1 #include <DHT.h>
2
3 #define DHTPIN 4
4 #define DHTTYPE DHT11
5
6 DHT dht(DHTPIN, DHTTYPE);
7
8 void setup() {
9     Serial.begin(115200);
10    dht.begin();
11 }
12
13 void loop() {
14     delay(2000);
15     float temperature = dht.readTemperature();
16     float humidity = dht.readHumidity();
17     Serial.print("Temperature: ");
18     Serial.print(temperature);
19     Serial.print(" °C");
20     Serial.print(" Humidity: ");
21     Serial.print(humidity);
22     Serial.println(" %");
23 }

```

Agregar los cambios y hacer commit en la rama dht11.

```

1 $ git add dht11.ino
2 $ git commit -m "Agregada la función para leer e imprimir los datos del DHT11"

```

Cambiar a la rama principal master.

```
1 $ git checkout master
```

Crear un archivo webserver.ino y agregar el código para crear un servidor web que muestra los datos del sensor DHT11.

```

1 #include <WiFi.h>
2 #include <WiFiClient.h>
3 #include <WebServer.h>
4 #include <DHT.h>
5
6 #define DHTPIN 4
7 #define DHTTYPE DHT11
8
9 DHT dht(DHTPIN, DHTTYPE);
10 WebServer server(80);
11
12 const char* ssid = "nombre_de_la_red";
13 const char* password = "contraseña_de_la_red";
14
15 void handleRoot() {
16     float temperature = dht.readTemperature();
17     float humidity = dht.readHumidity();
18     String html = "<html><body>";
19     html += "<h1>ESP32 Web Server</h1>";
20     html += "<h

```

Laboratorio de Git y Arduino: Creación de un web server con ESP32 y DHT11 En este laboratorio se explorarán los conceptos básicos de Git, desde la creación de un repositorio hasta la solución de conflictos.

Además, se trabajará en un proyecto de Arduino utilizando una placa ESP32 y un sensor DHT11 para crear un web server que muestre la humedad y temperatura en una página web y en el puerto serial. Requisitos Placa ESP32 Sensor DHT11 Cable USB IDE de Arduino instalado Librerías de DHT11 y ESP32 instaladas en Arduino IDE Git instalado en la computadora Paso 1: Crear un repositorio en Git Abrir la terminal o línea de comandos en tu computadora. Navegar a la carpeta donde se creará el repositorio. Ejecutar el comando git init para crear un nuevo repositorio. Paso 2: Crear el archivo principal del proyecto Abrir Arduino IDE y crear un nuevo sketch. Agregar las librerías de DHT11 y ESP32 al sketch. Definir las constantes necesarias para el pin del sensor y el pin del web server. Crear una función readDHT11() que lea los datos del sensor y los imprima en el puerto serial. Crear una función setup() que inicialice el sensor y el web server. Crear una función loop() que llame a la función readDHT11() y actualice los datos en la página web. El código completo del archivo main.ino debe verse así:

```

1 #include <WiFi.h>
2 #include "DHT.h"
3
4 // Definir constantes
5 #define DHTPIN 23      // Pin del sensor
6 #define WEB_SERVER_PORT 80 // Puerto del web server
7
8 // Definir variables
9 WiFiServer server(WEB_SERVER_PORT);
10 DHT dht(DHTPIN, DHT11);
11
12 // Función para leer el sensor y enviar datos al puerto serial
13 void readDHT11() {
14     float temp = dht.readTemperature();
15     float hum = dht.readHumidity();
16
17     Serial.print("Temperatura: ");
18     Serial.print(temp);
19     Serial.print("°C Humedad: ");
20     Serial.print(hum);
21     Serial.println("%");
22 }
23
24 // Configuración inicial
25 void setup() {
26     Serial.begin(9600);
27
28     // Inicializar el sensor
29     dht.begin();
30
31     // Inicializar el web server
32     WiFi.begin("SSID", "PASSWORD");
33     while (WiFi.status() != WL_CONNECTED) {
34         delay(1000);
35         Serial.println("Connecting to WiFi...");
36     }
37
38     Serial.println("WiFi connected");
39     server.begin();
40 }
41
42 // Loop principal
43 void loop() {
44     // Leer y enviar datos del sensor al puerto serial
45     readDHT11();
46
47     // Actualizar los datos en la página web
48     WiFiClient client = server.available();
49     if (client) {
50         String response = "HTTP/1.1 200 OK\nContent-Type: text/html\n\n<!DOCTYPE HTML>\n<html>\n";
51         response += "<h1>Humedad y temperatura</h1>\n";
52         response += "<p>Temperatura: ";
53         response += dht.readTemperature();
54         response += "°C</p>\n";

```

Laboratorio: Control de versiones con Git y Arduino En este laboratorio, aprenderás cómo utilizar Git para controlar versiones de un proyecto de hardware utilizando una placa ESP32 de Arduino. Crearás un repositorio, realizarás cambios en el código, agregarás y confirmarás esos cambios en Git y trabajarás con ramas y fusiones. También aprenderás a solucionar conflictos de fusión utilizando Git. Prerrequisitos Tener conocimientos básicos de programación en Arduino y el lenguaje C/C++. Tener instalado el IDE de Arduino. Tener instalado Git en tu computadora. Paso 1: Crear el repositorio Crea una carpeta para el proyecto en tu computadora y ábrela en el terminal o línea de comandos. Inicializa un repositorio de Git en la carpeta con el comando `git init`. Crea un archivo llamado `README.md` en la carpeta y agrega una breve descripción del proyecto. Agrega el archivo `README.md` al repositorio con el comando `git add README.md`. Confirma los cambios con el comando `git commit -m "Agrega el archivo README.md"`. Paso 2: Configuración del proyecto Arduino Crea un nuevo proyecto en el IDE de Arduino y guarda el archivo en la carpeta del proyecto. Abre el archivo y escribe el código para leer los datos de un sensor de temperatura y humedad DHT11 y mostrarlos en la pantalla serial.

```
1 #include <DHT.h>
2
3 #define DHTPIN 2          // Pin de datos del sensor
4 #define DHTTYPE DHT11    // Tipo de sensor
5
6 DHT dht(DHTPIN, DHTTYPE); // Instancia del sensor DHT11
7
8 void setup() {
9   Serial.begin(9600);
10  dht.begin();
11 }
12
13 void loop() {
14   delay(2000);
15   float temperature = dht.readTemperature();
16   float humidity = dht.readHumidity();
17   Serial.print("Temperatura: ");
18   Serial.print(temperature);
19   Serial.print(" °C\tHumedad: ");
20   Serial.print(humidity);
21   Serial.println("%");
22 }
```

Guarda el archivo y comprueba que se pueden leer los datos en la pantalla serial.

Paso 3: Creación de una nueva rama Crea una nueva rama en Git para el código del sensor DHT11 con el comando `git branch dht-sensor`. Cambia a la nueva rama con el comando `git checkout dht-sensor`. Agrega el archivo de código del sensor DHT11 al repositorio con el comando `git add <nombre del archivo>`. Confirma los cambios con el comando `git commit -m "Agrega el código del sensor DHT11"`.

Paso 4: Creación de una nueva rama para el servidor web Crea una nueva rama en Git para el servidor web con el comando `git branch web-server`. Cambia a la nueva rama con el comando `git checkout web-server`. Crea un nuevo archivo llamado `web_server.ino` en el IDE de Arduino y escribe el código para mostrar los datos del sensor DHT11 en una página web.

Creación de una nueva rama para el servidor web:

Ahora crearemos una nueva rama en la que trabajaremos en el servidor web. Esta rama será una copia de la rama "main". En la línea de comando, escriba:

```
1 git checkout -b server-web
```

Esto crea una nueva rama llamada "server-web" lo mueve a ella.

Agregue el servidor web:

Ahora, crearemos un servidor web para mostrar la temperatura y la humedad en una página web. Cree un nuevo archivo llamado `server.ino` y pegue el siguiente código:

```
1 // Incluir librerías
2 #include <WiFi.h>
3 #include <WebServer.h>
4
5 // Definir el SSID y la contraseña de la red WiFi
6 const char* ssid = "nombre_red";
```



```

7  const char* password = "contraseña_red";
8
9  // Crear una instancia del servidor web
10 WebServer server(80);
11
12 void setup() {
13     // Iniciar la comunicación serial
14     Serial.begin(9600);
15
16     // Conectar a la red WiFi
17     WiFi.begin(ssid, password);
18
19     // Esperar a que se establezca la conexión
20     while (WiFi.status() != WL_CONNECTED) {
21         delay(1000);
22         Serial.println("Conectando a la red WiFi...");
23     }
24
25     // Imprimir la dirección IP en la consola
26     Serial.println("");
27     Serial.println("Conectado a la red WiFi");
28     Serial.print("Dirección IP: ");
29     Serial.println(WiFi.localIP());
30
31     // Manejar la petición GET en la ruta "/"
32     server.on("/", [](){
33         String html = "<html><body><h1>Temperatura: XX°C</h1><h1>Humedad: XX%</h1></body></html>";
34         server.send(200, "text/html", html);
35     });
36
37     // Iniciar el servidor
38     server.begin();
39 }
40
41 void loop() {
42     // Escuchar las peticiones entrantes
43     server.handleClient();
44
45     // Imprimir la temperatura y la humedad en la consola
46     Serial.print("Temperatura: ");
47     Serial.print(dht.readTemperature());
48     Serial.println(" °C");
49     Serial.print("Humedad: ");
50     Serial.print(dht.readHumidity());
51     Serial.println(" %");
52
53     // Esperar 2 segundos
54     delay(2000);
55 }

```

Este código crea un servidor web que muestra la temperatura y la humedad en una página web. Para hacer esto, usamos la librería WebServer. En la función setup(), conectamos a la red WiFi y configuramos el servidor web para manejar las peticiones GET en la ruta /. En la función loop(), leemos la temperatura y la humedad del DHT11 y las imprimimos en la consola. También escuchamos las peticiones entrantes del servidor web.

Confirme los cambios:

Ahora, confirme los cambios que ha realizado. En la línea de comando, escriba:

```

1 git add .
2 git commit -m "Agregado el servidor web"

```

Esto agrega todos los archivos nuevos y modificados al área de preparación y crea un nuevo commit con el mensaje .Agregado el servidor web". Cambie a la rama principal y fusionar los cambios:

```

1 git checkout main
2 git merge webserver

```

Resolver conflictos ¡Claro! Aquí te envió el paso a paso completo:

Cambie a la rama principal y fusionar los cambios con `git checkout main` y `git merge webserver`. Resolver conflictos. Es posible que aparezca un conflicto en esta fusión, ya que estamos fusionando cambios de dos ramas diferentes en la misma línea de código. Git nos pedirá que resolvamos este conflicto manualmente. Para hacer esto, abrimos el archivo en conflicto (en nuestro caso, `main.ino`) y buscamos las secciones que Git ha identificado como conflictivas.

```
1 <<<<<< HEAD
2 Serial.print("Humedad: ");
3 Serial.print(hum);
4 Serial.print("% - Temperatura: ");
5 Serial.print(temp);
6 Serial.println("°C");
7 =====
8 server.send(200, "text/plain", "Humedad: " + String(hum) + "% - Temperatura: " + String(temp) + "
   °C");
9 >>>>>> webserver
```

La sección encerrada por «“<HEAD y ===== es la versión que se encuentra en la rama actual (`main`), mientras que la sección encerrada por ===== y ””>webserver es la versión que se encuentra en la rama que estamos fusionando (`webserver`).

En este caso, queremos conservar los cambios de ambas ramas, por lo que podemos simplemente editar el archivo para combinar los cambios y eliminar las marcas de conflicto. Por ejemplo:

```
1 Serial.print("Humedad: ");
2 Serial.print(hum);
3 Serial.print("% - Temperatura: ");
4 Serial.print(temp);
5 Serial.println("°C");
6 server.send(200, "text/plain", "Humedad: " + String(hum) + "% - Temperatura: " + String(temp) + "
   °C");
```

Una vez resuelto el conflicto, guardamos el archivo y agregamos los cambios al índice de Git:

```
1 git add main.ino
```

Luego, podemos hacer el commit de la fusión:

```
1 git commit -m "Fusionar cambios de la rama webserver"
```

Volver a la rama del servidor web y eliminar la rama. Finalmente, volvemos a la rama `webserver` para asegurarnos de que todo funcione correctamente:

```
1 git checkout webserver
```

Si todo funciona como debería, podemos eliminar la rama `webserver` ya que ya no la necesitamos:

```
1 git branch -d webserver
```

¡Y listo! Ahora hemos creado un repositorio de Git, creado una nueva rama para trabajar en una nueva función, fusionado esa rama con la rama principal y resuelto cualquier conflicto que haya surgido en el proceso. Todo esto mientras trabajamos en un proyecto de Arduino con una ESP32 que lee la humedad y temperatura de un DHT11 y muestra los datos en un servidor web.