

Informe Taller Git Parte 1

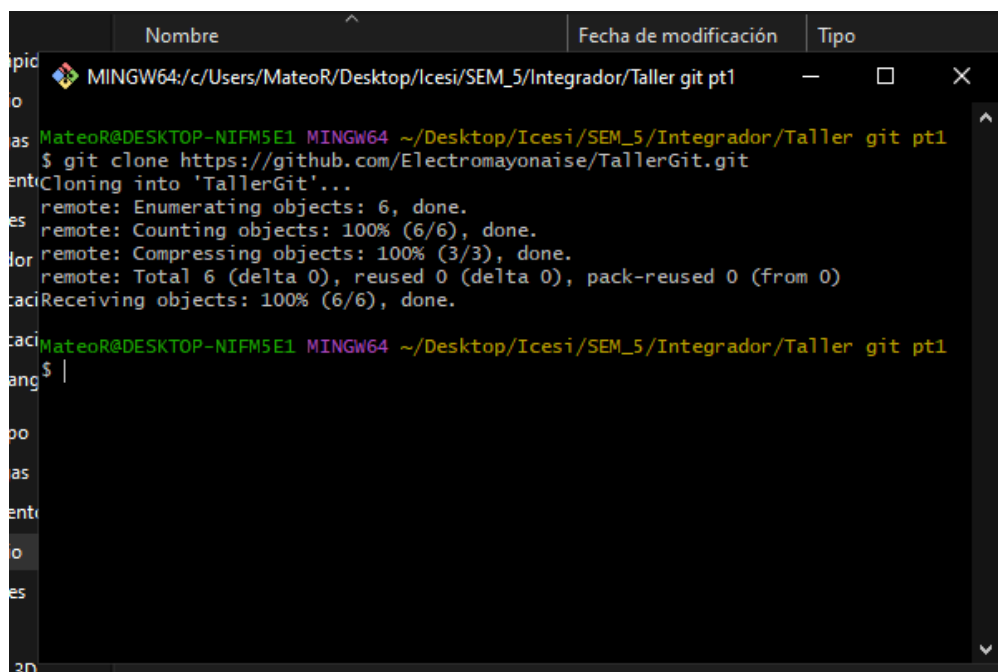
Usuario A - Mateo Rubio - A00400104
Usuario B - Martin Gomez - A00399958
Usuario C - Santiago Escobar - A00382203
Usuario D - Santiago Angel - A00379822
Usuario E - Julio Prado - A00399637

Link: <https://github.com/Electromayonaise/TallerGit.git>

1. Configuración Inicial y Creación de Repositorio

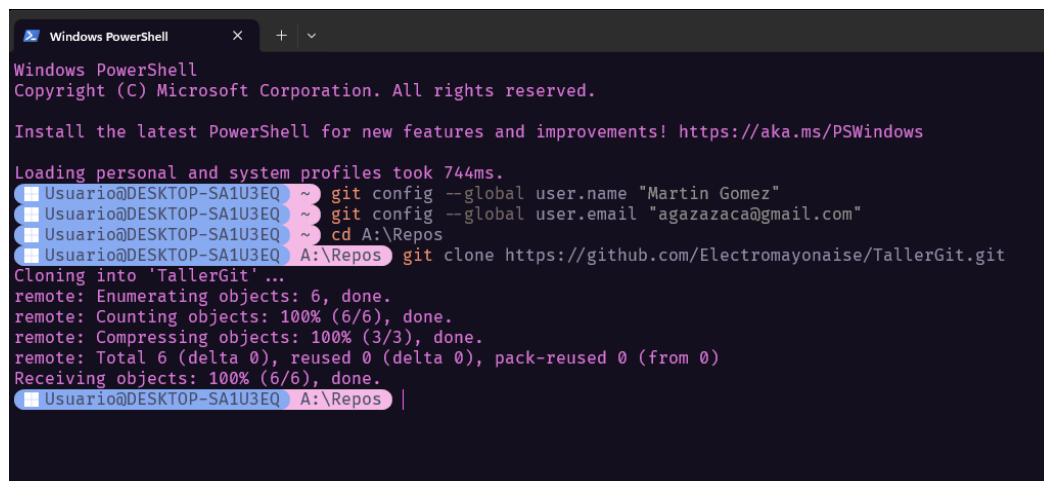
Mateo

En mi caso ya tenía configurado el usuario de Git por lo que solo clone el repositorio



```
Nombre ^ Fecha de modificación Tipo
MINGW64: c:/Users/MateoR/Desktop/Icesi/SEM_5/Integrador/Taller git pt1
MateoR@DESKTOP-NIFM5E1 MINGW64 ~/Desktop/Icesi/SEM_5/Integrador/Taller git pt1
$ git clone https://github.com/Electromayonaise/TallerGit.git
Cloning into 'TallerGit'...
remote: Enumerating objects: 6, done.
remote: Counting objects: 100% (6/6), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 6 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
Receiving objects: 100% (6/6), done.
MateoR@DESKTOP-NIFM5E1 MINGW64 ~/Desktop/Icesi/SEM_5/Integrador/Taller git pt1
$ |
```

Martin



```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

Loading personal and system profiles took 744ms.
Usuario@DESKTOP-SA1U3EQ ~ git config --global user.name "Martin Gomez"
Usuario@DESKTOP-SA1U3EQ ~ git config --global user.email "agazazaca@gmail.com"
Usuario@DESKTOP-SA1U3EQ ~ cd A:\Repos
Usuario@DESKTOP-SA1U3EQ A:\Repos git clone https://github.com/Electromayonaise/TallerGit.git
Cloning into 'TallerGit' ...
remote: Enumerating objects: 6, done.
remote: Counting objects: 100% (6/6), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 6 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
Receiving objects: 100% (6/6), done.
Usuario@DESKTOP-SA1U3EQ A:\Repos |
```

Julio

```
Julio@fedora:~/repos$ git clone https://github.com/Electromayonaise/TallerGit.git
Cloning into 'TallerGit'...
remote: Enumerating objects: 6, done.
remote: Counting objects: 100% (6/6), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 6 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
Receiving objects: 100% (6/6), done.
Julio@fedora:~/repos$
```

Santiago Escobar

```
MINGW64:/c/Users/SANTIAGO/Desktop

SANTIAGO@DESKTOP-ORSU30Q MINGW64 ~/Desktop (master-slave)
$ git clone https://github.com/Electromayonaise/TallerGit.git
Cloning into 'TallerGit'...
remote: Enumerating objects: 6, done.
remote: Counting objects: 100% (6/6), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 6 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
Receiving objects: 100% (6/6), done.

SANTIAGO@DESKTOP-ORSU30Q MINGW64 ~/Desktop (master-slave)
$
```

Santiago Angel Ordoñez

```
MINGW64:/c/Users/USUARIO/Documents/6toSemestre/Integrador I/TallerGithub

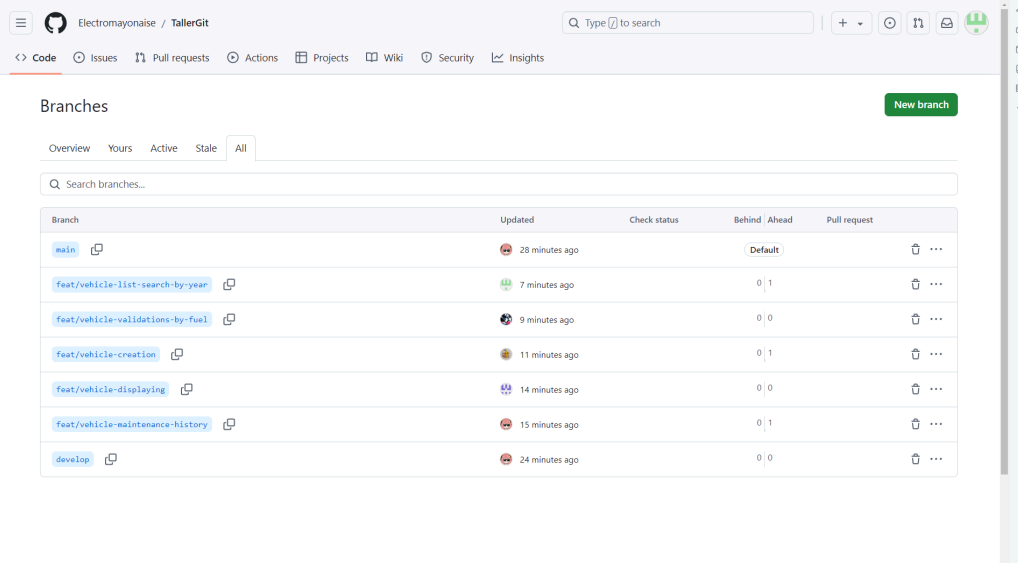
USUARIO@LAPTOP-2DDPU884 MINGW64 ~/Documents/6toSemestre/Integrador I/TallerGithub
$ git clone https://github.com/Electromayonaise/TallerGit.git
Cloning into 'TallerGit'...
remote: Enumerating objects: 6, done.
remote: Counting objects: 100% (6/6), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 6 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
Receiving objects: 100% (6/6), done.

USUARIO@LAPTOP-2DDPU884 MINGW64 ~/Documents/6toSemestre/Integrador I/TallerGithub
$ |
```

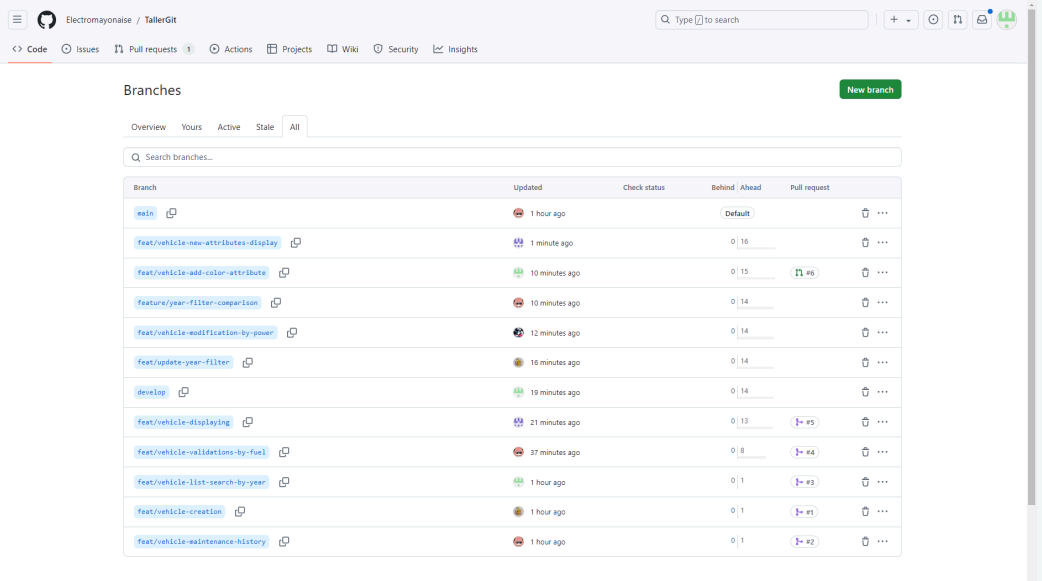
¿Cuál es la diferencia entre clonar un repositorio y hacer un fork? ¿En qué situaciones utilizarías cada uno?

Clonar viene a ser descargar todos los archivos existentes en un repositorio de tal forma que el proyecto acabe estando disponible en mi máquina física. Por su parte, un fork es crear un nuevo repositorio que contiene un proyecto determinado. El repositorio resultante del fork será completamente independiente del repositorio original y todos los cambios realizados se guardarán en este nuevo repositorio y no afectarán al original. Clonamos un repositorio cuando queremos descargar los archivos de un proyecto y hacer cambios que solamente se reflejan en nuestra máquina. Hacemos fork cuando queremos crear un nuevo repositorio que tome como base otro ya existente.

2. Colaboración en Equipo usando Ramas



Branch	Updated	Check status	Behind	Ahead	Pull request
main	28 minutes ago		Default		
feat/vehicle-list-search-by-year	7 minutes ago		0	1	
feat/vehicle-validations-by-fuel	9 minutes ago		0	0	
feat/vehicle-creation	11 minutes ago		0	1	
feat/vehicle-displaying	14 minutes ago		0	0	
feat/vehicle-maintenance-history	15 minutes ago		0	1	
develop	24 minutes ago		0	0	

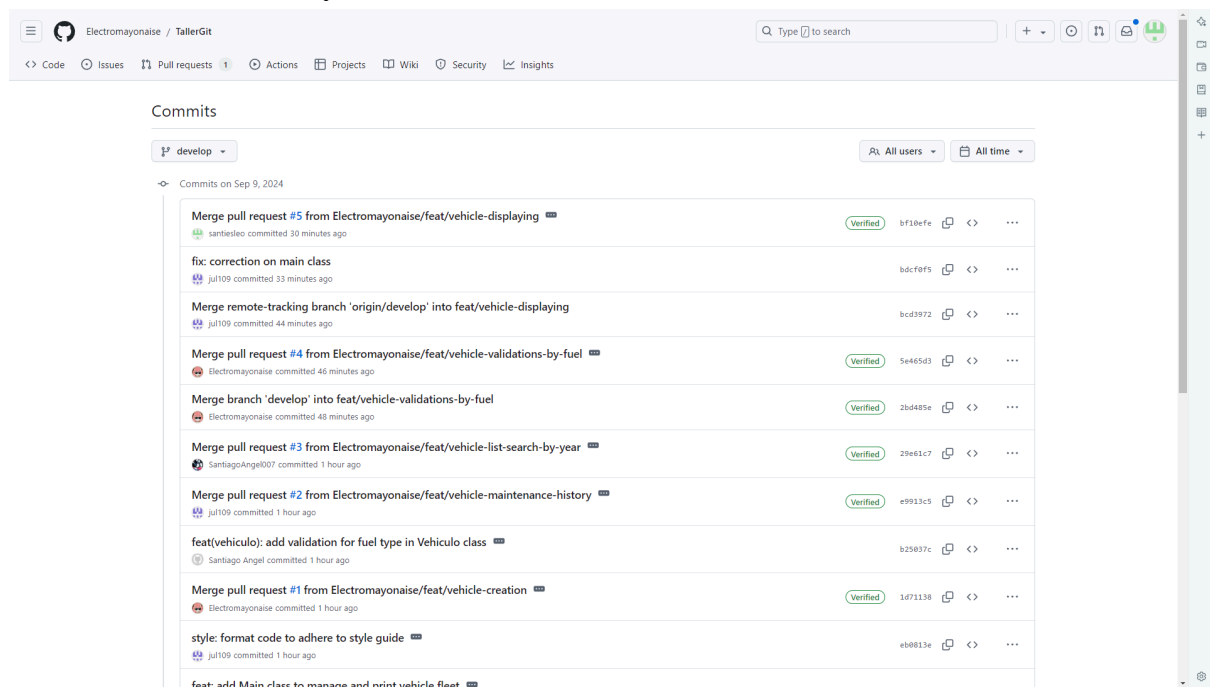


Branch	Updated	Check status	Behind	Ahead	Pull request
main	1 hour ago		Default		
feat/vehicle-new-attributes-display	1 minute ago		0	16	
feat/vehicle-add-color-attribute	10 minutes ago		0	13	#6
feature/year-filter-comparison	10 minutes ago		0	14	
feat/vehicle-modification-by-power	12 minutes ago		0	14	
feat/update-year-filter	16 minutes ago		0	14	
develop	19 minutes ago		0	14	
feat/vehicle-displaying	21 minutes ago		0	13	#5
feat/vehicle-validations-by-fuel	27 minutes ago		0	8	#4
feat/vehicle-list-search-by-year	1 hour ago		0	1	#3
feat/vehicle-creation	1 hour ago		0	1	#1
feat/vehicle-maintenance-history	1 hour ago		0	1	#2

¿Por qué es importante seguir una convención para nombrar las ramas? ¿Qué beneficios tiene en un equipo grande

Seguir una convención para nombrar ramas en un equipo grande ofrece numerosos beneficios, como mejorar la claridad y la organización del proyecto. Permite que todos los miembros del equipo entiendan rápidamente el propósito de cada rama, lo que facilita la colaboración y reduce el tiempo dedicado a buscar o preguntar sobre el trabajo de otros. Además, ayuda a minimizar conflictos al evitar colisiones de nombres y mejorar la eficiencia en la revisión de código, ya que los revisores pueden comprender el contexto de los cambios con facilidad. Las convenciones de nombres también contribuyen a mantener un historial de proyecto claro y ordenado, lo que es esencial para rastrear cambios, gestionar versiones y simplificar la depuración. En definitiva, una convención bien definida para el nombramiento de ramas mejora la eficiencia, la comunicación y la calidad del desarrollo en equipo.

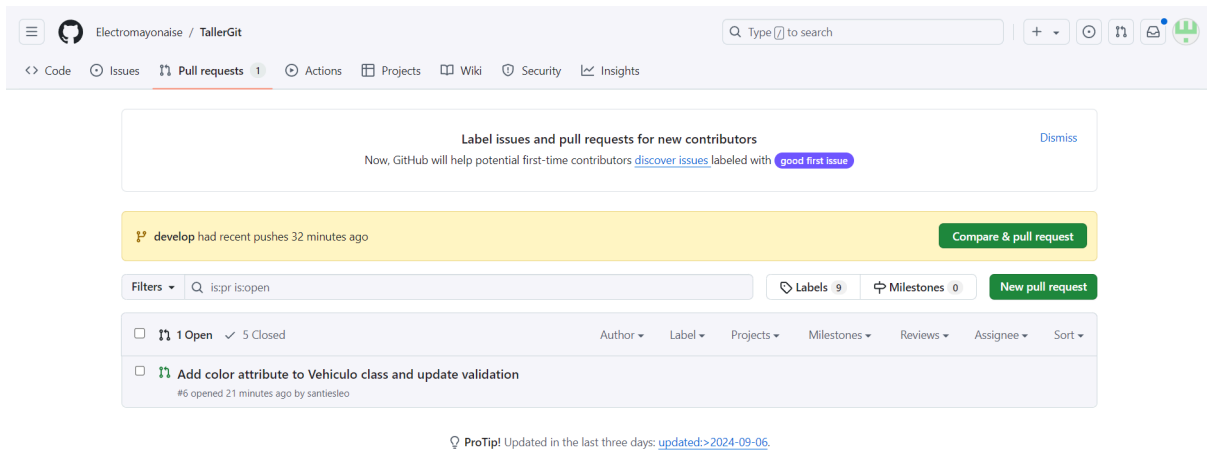
3. Gestión de Commits y Estándares de Codificación



¿Qué diferencia hay entre un commit estándar y uno amend? ¿Cuándo usarías cada uno?

Un commit estándar y uno con `--amend` se diferencian principalmente en cómo afectan al historial de cambios en un repositorio Git. Un commit estándar crea un nuevo registro, añadiendo un punto en el historial que refleja los cambios realizados, mientras que `git commit --amend` permite modificar el último commit, ya sea para corregir el mensaje o añadir cambios olvidados. El commit estándar se utiliza para registrar modificaciones regularmente y mantener un historial detallado, siendo la práctica habitual durante el desarrollo. Por otro lado, `--amend` es útil cuando necesitas hacer pequeñas correcciones al último commit y aún no lo has compartido con otros, ya que altera el historial

4. Merge y Resolución de Conflictos



¿Qué estrategias de resolución de conflictos podrías aplicar en un proyecto con múltiples colaboradores?

Como equipo, para manejar conflictos en un proyecto con múltiples colaboradores, podríamos aplicar las siguientes estrategias:

1. **Establecer un flujo de trabajo claro:** Definir y documentar un flujo de trabajo para la gestión de ramas y merges puede reducir la frecuencia de conflictos. Por ejemplo, podríamos usar una estrategia como Git Flow o GitHub Flow, donde se establecen ramas específicas para características, correcciones de errores y lanzamientos.
2. **Comunicación abierta y frecuente:** Mantener una comunicación constante entre los miembros del equipo es clave. Utilizar herramientas de comunicación como Slack o Teams para discutir cambios importantes y coordinar el trabajo puede ayudar a prevenir conflictos antes de que ocurran.
3. **Realizar commits y pushes frecuentes:** Hacer commits y pushes regulares asegura que los cambios se integren con frecuencia en la rama principal, lo que reduce la posibilidad de grandes conflictos cuando se realiza un merge.
4. **Resolver conflictos lo antes posible:** Abordar los conflictos tan pronto como se detecten es fundamental. Hacerlo en etapas tempranas minimiza el riesgo de conflictos complicados que pueden surgir cuando los cambios se acumulan.
5. **Utilizar herramientas de fusión y comparación:** Emplear herramientas gráficas de fusión y comparación, como `git mergetool` o herramientas de terceros como Meld o Beyond Compare, puede facilitar la resolución de conflictos al proporcionar una visualización clara de las diferencias.
6. **Asignar responsabilidades para la resolución de conflictos:** Designar a un miembro del equipo o un pequeño grupo para manejar la resolución de conflictos puede garantizar que se aborden de manera sistemática y eficiente.
7. **Documentar resoluciones de conflictos:** Registrar cómo se resolvieron los conflictos y las decisiones tomadas en el proceso puede servir como referencia futura. Esto ayuda a los nuevos colaboradores a entender el contexto y las decisiones previas.
8. **Implementar revisiones de código (code reviews):** Realizar revisiones de código antes de fusionar ramas puede ayudar a identificar y discutir posibles conflictos antes de que se conviertan en problemas mayores.

5. Trabajo Final y Documentación del Proyecto

¿Cómo puedes asegurarte de que la documentación de tu proyecto sea útil para futuros colaboradores?

Como equipo, podemos asegurarnos de que la documentación de nuestro proyecto sea útil para futuros colaboradores siguiendo estos pasos:

1. **Estructura clara y organizada:** Es importante que la documentación siga una estructura lógica, comenzando con una introducción al proyecto, seguido por guías detalladas sobre la instalación, configuración y uso del proyecto. Esto ayudará a nuevos colaboradores a entender rápidamente de qué se trata el proyecto y cómo empezar a trabajar en él.
2. **Actualización constante:** Debemos revisar y actualizar la documentación regularmente para reflejar los cambios que se hagan en el proyecto. Esto evitará que la documentación se quede obsoleta y garantice que siempre esté alineada con el código actual.
3. **Ejemplos y casos de uso:** Incluir ejemplos prácticos y casos de uso comunes facilita la comprensión de cómo funciona el proyecto en diferentes escenarios. Esto también puede ayudar a los colaboradores a implementar nuevas funcionalidades de manera más eficiente.
4. **Estándares y convenciones:** Definir estándares claros para el código (como convenciones de nombres, estilo de código, estructura de carpetas, etc.) en la documentación asegura consistencia en el desarrollo. Esto facilita que los colaboradores sepan cómo contribuir de manera coherente con el proyecto.
5. **Guía de contribución:** Incluir una guía de cómo contribuir al proyecto es esencial. Esta guía debe detallar el proceso para hacer pull requests, cómo reportar errores y sugerir mejoras, así como las herramientas o scripts necesarios para ejecutar pruebas.
6. **Documentación del código:** Asegurarnos de que el código esté bien comentado y con descripciones claras de las funciones y métodos principales ayuda a que sea más fácil de entender y mantener. Además, generar documentación automática a partir del código puede ser muy útil para seguir las actualizaciones.
7. **Preguntas frecuentes (FAQ):** Incluir una sección de preguntas frecuentes puede ayudar a futuros colaboradores a resolver dudas comunes rápidamente sin tener que buscar demasiada información o consultar a los autores.