Untracked files: Son archivos que están en nuestro Working Directory, lo que aparezca en rojo es lo que se ha modificado y hay que pasarlo al Staging.

git add [archivo]: Nos agrega al archivo al staging area(el limbo) \*\* -A\*\* nos agrega todos los archivos

git rm --cached: Nos devolvemos un paso

git rm -f [file]: Elimina el archivo por completo.

git add -n [file]: Que no haga nada

# Edit hello.py and main.py git add hello.py git commit # Realize you forgot to add the changes from main.py git add main.py git commit --amend --no-edit

**git tag: nos permite agregar etiquetas a nuestros cambios.**

-a para la anotación

-m para el mensaje

-l nos muestra la lista de etiquetas

-f para renombrar

-d para borrar

\*Tipos de tag  
–tag ligera

git **tag** **1**.0 + "SHA"

git **tag** **1**.0 02512454212dfd2s...

\*\*tag anotada

git **tag** **-a** 1.0 -m '**version** estable' 0251245421...

git **tag** **-a** "version" -m "mensaje" "sha"

\*Opciones  
\*\*Borrar tags

git **tag** **-d** 1.0

git **tag** **-d** "versíon"

\*\*Renombrar tags

git tag -f -a 0.1 'iniciando el proyecto' + sha

**git log**

git config --global alias.superlog "log --graph --abbrev-commit --decorate --date=relative --format=format:'%C(bold blue)%h%C(reset) - %C(bold green)(%ar)%C(reset) %C(white)%s%C(reset) %C(dim white)- %an%C(reset)%C(bold yellow)%d%C(reset)' --all"

**git diff** sha1 sha2

git diff también funciona con los tags:

git diff 0.5 0.3

git reset –soft

Hay que tener en cuenta que si usas **git reset --soft**  
Tienes 10 commits y borras el numero 7, los commits 8, 9 y 10 también se borran, y tendras un nuevo commit que tomara el numero 8.

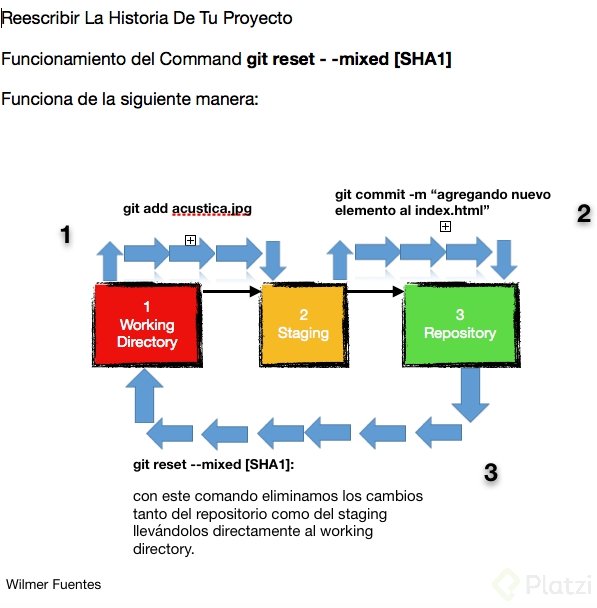
Quita cambios desde el commit indicado y coloca esos archivos con sus cambios en el staging. Listos para hacer nuevamente cambios y un commit.

git reset –mixed

Descarta cambios y quita commit pero no deja archivos en el staging y los deja en el working directory

Git reset –hard

https://stackoverflow.com/questions/3528245/whats-the-difference-between-git-reset-mixed-soft-and-hard,



git config –global core.editor “atom –wait”

RAMAS

Las ramas son muy importantes si quieres trabajar con un equipo y no quieres tocar la rama master para no crear conflictos,

**git branch [nombre]** se crea una nueva rama  
**-l**: listamos las ramas  
**-d/-D [nombre]**: borramos rama  
**-m [nombre] [nombre\_nuevo]**: para renombrar ramas

Las ramas ayudan a gestionar el flujo de trabajo en un equipo de desarrollo.

* git branch **nombrerama** : comando para crear la rama
* git branch -l : listar ramas
* git branch -d **namebranch** : eliminar una rama
* git branch -D **namebranch**: eliminar branch’s con commits
* git branch -m **namebranchedit** **newnamebranch** : editar nombre de la rama

Movernos entre ramas

**git checkout [nombre/sha1]**: Nos permite mover entre ramas y entre commits, no vamos a borrar nada. Acá es donde podemos movernos en el tiempo.

git checkout -b [nombre\_rama]: Nos permite crear una nueva rama sin necesidad de usar *branch*

Git Merge

**git merge [rama]**: Nos permite mezclar los cambios realizados en dicha rama con la rama en la que estamos.

**fast-forward**: los mezcla automáticamente  
**recursive/auto-merging**: ambas ramas salieron al mismo tiempo y hay algo nuevo en la rama que la otra no recuerda, por eso hace la mezcla recursiva.  
**manual merge**: nos va a tocar decirle a git específicamente los cambios que queremos mezclar

**git rebase:** hace prácticamente lo mismo que *merge*, cambiamos la historia de nuestro proyecto sin crear bifurcaciones del proyecto. Es mejor usar **merge**  
Usar solo git rebase de manera local.

-i: de manera interactiva, nos abrira el editor que tengamos definido en la configuración de git.

**Git Stash**

**git stash**: es otro de los limbos, como el staging area. Para agregar los cambios estos deben estar en el staging area.  
**git stash list**: nos muestra la lista de stash que tengamos.  
**git stash drop stash@{numero}**: nos permite borrar un stash.  
**git stash apply**: aplicamos el último cambio

Solo se pueden aplicar si sobre cambios estageados

**Git cherry-pick**

**git cherry-pick [SHA1]** nos permite cambiar un commit a otra rama para salvarnos la vida.

Tomar el sha del commit, pasarse a la rama donde debe estar y tipear

Git cherry-pick sha1

Referencias relativas

Moverse por git usando los hashes de los commits puede volverse un tanto tedioso. En el mundo real no vas a tener una visualización de commits tan linda en la terminal, así que vas a tener que usar git log para ver los hashes.

Peor aún, los hashes en general son mucho más largos en el git real, también. Por ejemplo, el hash del commit que introduje en el nivel anterior es fed2da64c0efc5293610bdd892f82a58e8cbc5d8. No es algo particularmente fácil de nombrar...

Lo copado es que git es bastante astuto con los hashes. Sólo requiere que especifiques una cantidad de caracteres suficientes para identificar unívocamente al commit. Entonces, yo podría simplemente tipear fed2 en lugar de esa cadena larga de arriba.

Como ya dije, especificar los commits por su hash no es la manera más conveniente, y por eso git tiene referencias relativas. ¡Son geniales!

Con las referencias relativas podés arrancar de algún lugar recordable (como la rama bugFix, o HEAD) y manejarte desde ahí.

Los commits relativos son poderosos, pero ahora vamos a presentar sólo dos formas simples:

Moverse un commit atrás con ^

Moverse una cantidad de commits atrás con ~<num>

**El operador "~"**

Digamos que querés moverte un montón de niveles atrás en tu árbol de commits. Podría ser tedioso tipear ^ muchas veces, por lo que git tiene el operador ~.

El operador ~ (opcionalmente) toma una cantidad que especifica la cantidad de padres que querés volver hacia atrás. Veámoslo en acción

Ejemplo: git checkout HEAD~4

**Forzando los branches**

Ahora que sos un experto en las referencias relativas, usémoslas para algo.

Una de las formas más comunes en que uso las referencias relativas es para mover las ramas. Podés reasignar directamente una rama a un commit usando la opción -f. Así que algo como:

git branch -f main HEAD~3

Mueve (forzadamente) la rama main tres padres atrás de HEAD.

GITHUB

el \*\*fork \*\*en github nos hace una copia del proyecto en nuestro perfil/repositorios para poder hacerle lo que queramos sin afectar el proyecto original.

git clone [https/ssh]: siempre es recomendable en lo que se puede de usar ssh([git@github.com](mailto:git@github.com)…) en vez de usar https ya que la primera es mucho más segura.

**git remote (Añadiendo un repositorio remoto a uno local) - enlazando**

* git remote add [origin] [SSH/HTTPS] Conecta un repositorio con nuestro equipo local.
* git remote -v Lista las conexiones existentes.
* git remote remove [origin] Elimina una conexión con algún repositorio.

git remote add origin [**git@github.com:remyor09/practica\_git.git**](mailto:git@github.com:remyor09/practica_git.git)

$ git remote -v

origin git@github.com:remyor09/practica\_git.git (fetch)

origin git@github.com:remyor09/practica\_git.git (push)

git remote remove origin (borra el repositorio enlazado en remoto)

# git pull/fetch (Trayendo cambios desde el repositorio remoto)

# Vamos a compartir nuestro código con el repositorio remoto. Antes de poder enviar cambios tengo que tener los mismos cambios en el repositorio remoto que en el repositorio local.

# Para traer los cambios del repositorios remoto vamos a usar el comando git fetch. Este comando recibe dos parámetros 1. desde donde y hacia donde van.

# Luego de traer los cambios, debemos mezclarlos con la rama que estamos usando en local. Para esto usamos git merge.

**2** formas de traer tus cambios de github a tu repo local:

* **git fetch**: descarga los cambios a una rama generalmente llamada origin/master, seguido de esto tienes que fuisonar esa rama

$ git fetch origin master

# origin= remoto | master= rama del remoto

# ahora tienes una rama llamada origin/master

# que pueedes ver con 'git branch -a'

$ git merge origin/master

# si obtienes un error usar la siguiente linea

# git merge origin/master --allow-unrelated-histories

* **git pull**: es como ejecutar los 2 pasos anteriores(fetch y merge) en uno solo

$ git pull origin master

Debemos mezclar los cambios

git merge origin/master

Error: fatal: refusing to merge unrelated histories

Solución: git merge origin/merge –allow-unrelated-histories

# git push (Enviando cambios al repositorio remoto)

Para enviar nuestro código al repositorio remoto vamos a usar el comando git push. Así logramos que otras personas tengan acceso a nuestro proyecto.  
Cuando esto termine tendremos acceso a nuestro trabajo desde la plataforma de **GitHub.**  
Lo primero que debemos hacer es enviar nuestros tags en el proyecto.  
también recordemos que podemos enviar ramas, lo que también podemos hacer a través del uso del comando git push [nombre de la rama].

Asi subimos nuestros cambios a github:

git push origin master

Tambien podemos enviar los tags:

git push origin master --tags

Podemos enviar otras ramas:

git push origin [otra\_rama]

Unwatch (Nos permite que nos lleguen notificaciones si hay cambios en el repositorio)

Not watching (Se te notifica cuando participas o mencionas)

Ignoring (Nunca ser notificado)

Star (Asi como en facebook se vive a base de likes, aqui se vive a base de estrellas, cuanto mas estrellas mas valorado es ese proyecto, y a su misma vez tiene muchos forks)

Fork (Indica la cantidad de gente que hizo copia de su repositorio para poderles enviar cambios)

Podemos ver las estadísticas de nuestro proyecto en Insights

contributors (Nos permite ver el avance de nuestro proyecto)

traffic (Nos muestra el trafico de nuestro proyecto)

punch card (Muestra los dias que a estado trabajando)

network (Vemos el flujo de ramas)

members (Nos muestra la gente que le a hecho fork a nuestro repositorio)

dependents (Nos muestra si nuestro repositorio depende de otro, es decir de un super repositorio)

‘‘LeonidasEsteban’’ Para proyectos complejos se puede hacer un “dependecia” de repositorios o sub repositorios. Es como un proyecto dentro de otro pero es muy raro usarlos y de preferencia yo solo recomiendo un solo reposotorio’.

Settings:

collaborators (Aqui se añade a mas gente para que colabore en este proyecto)

branches (Podemos cambiar la rama principal, y proteger ramas)

CONSEJO: Protege siempre la rama master (Una vez protegida ningun cambio va a ir directamente a master tenemos que pasar por cierto proceso para que un codigo vaya a master) (Este proceso lo vemos dentro del curso)

pull request (Es una forma de tomar una rama alterna por ejemplo la rama RD y no mezclarla directamente con master. Si no enviar una solicitud de mezcla desde la rama RD a la master, asi alguien mas puede ver esa solicitud, algun colaborador, etc…Y se haga un code review)

Webhooks (Es la forma de poderle agregar git code a otros servicios para que hagan un auto deploy)

Integrations & servicies (Incluir otros servicios como amazon, y que pueden hacer uso de nuestro repositorio)

Deploy keys (Son las llaves publicas ssh de tu servidor para poder hacer git pull desde el servidor, y tener tu código en producción.)

Pull requests (Vemos una lista de pull requests)

Issues (Son la forma en que manejamos nuestros problemas, bugs en github. Por ejemplo: hay un bug, este no va a ser reportado si no que lo que vamos a hacer es crear un issue comentando donde hay un problema, se hace de esta manera, ocurre en tal sistema operativo, etc…)

Proyects (Nos aparece nuestros proyectos)

Al crear un “issue” o un “pull request”, github te permite incluir etiquetas en los mismos. Entre las etiquetas que nos pueden interesar a los que estamos comenzando a usar la plataforma se encuentran “help wanted” y “good first issue”. Github anima a los desarrolladores a que marquen los “issue” y los “pull request” que son más aptos para los que inician en este mundo con estas etiquetas y adicionalmente se pide a la comunidad que den un apoyo especial para estos issue/pull request. ¿Quiéres comenzar a colaborar en un proyecto real en Github pero aún te sientes inseguro con tus conocimientos? Haz una búsqueda de estás etiquetas y encontrarás miles de oportunidades para hacer tu primer aporte.

En la primera parte no es realmente correcto. Es mejor decir:

De las opciones:

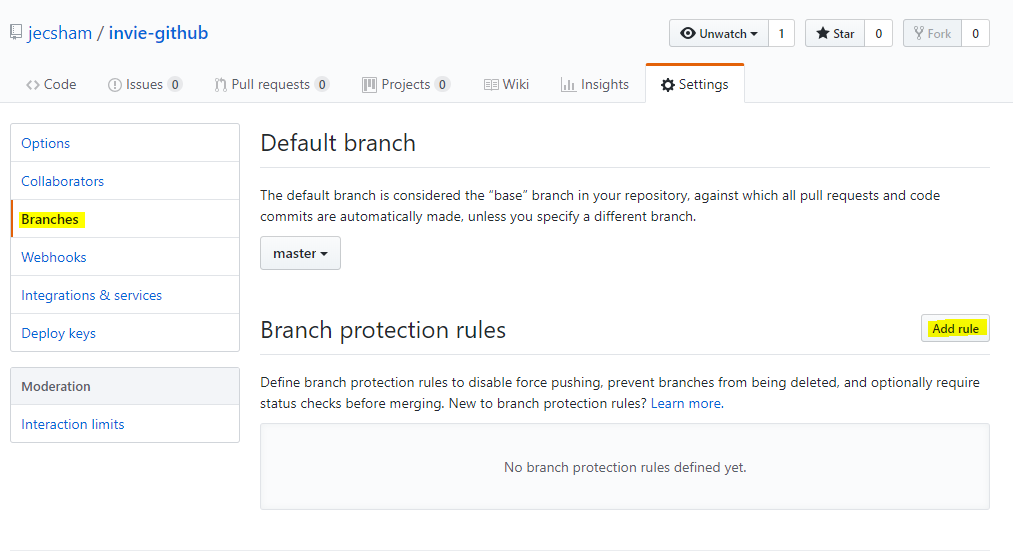
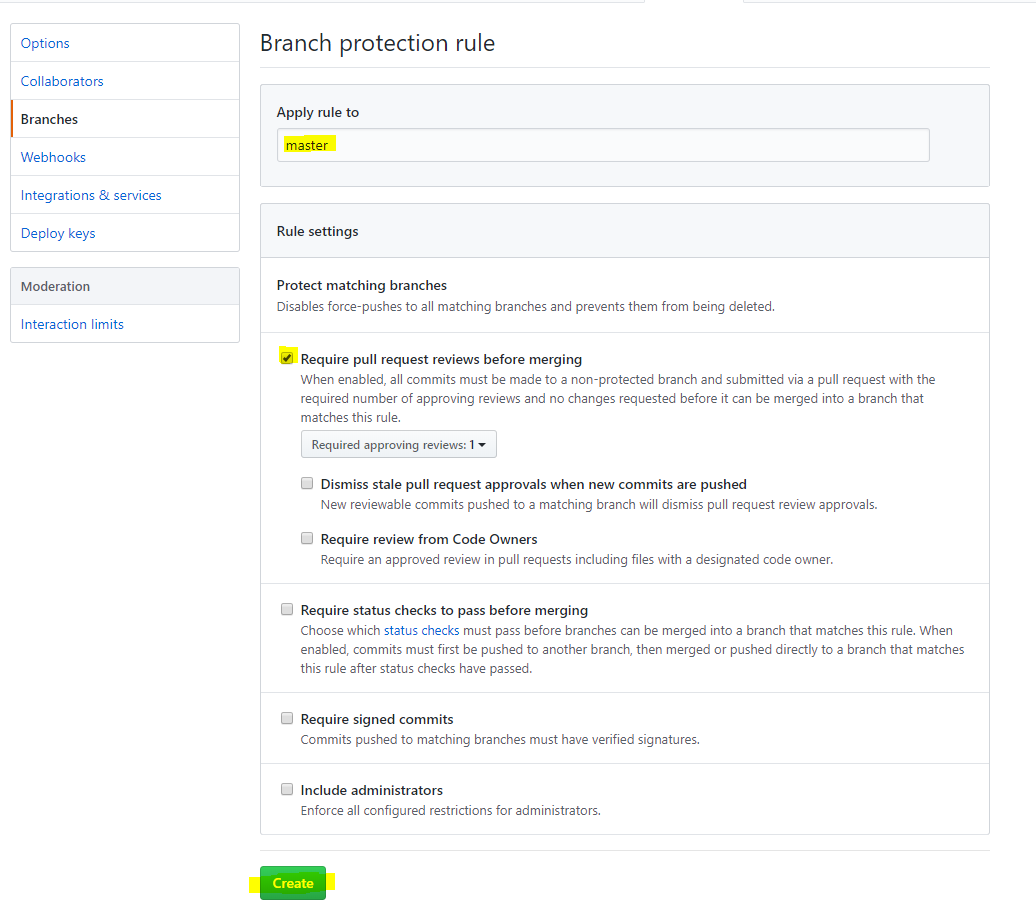
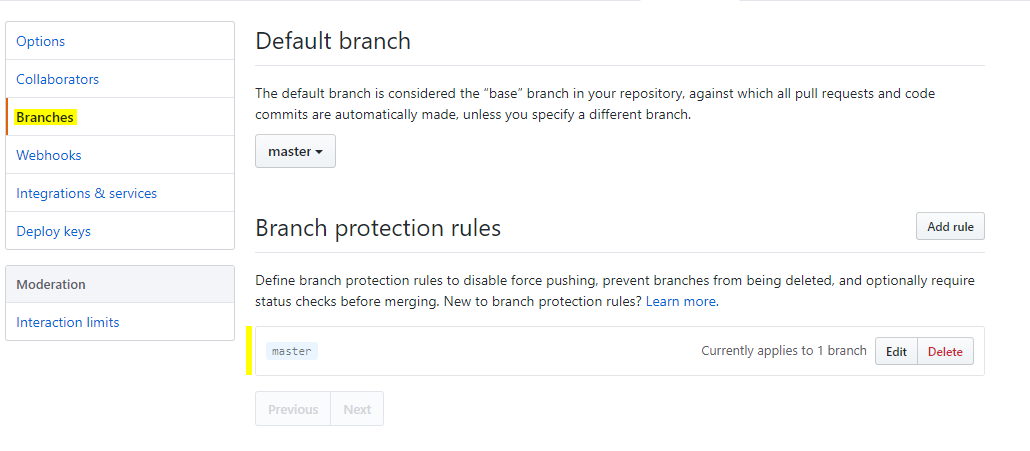
Not watching: ser notificado cuando seas mencionado o cuando estes participando (en una conversacion).

Releases only: ser notificado cuando haya un nuevo release, seas mencionado o estes participando.

Watching: ser notificado de todas las conversaciones.

Ignoring: nunca ser notificado.

**Aporte para la nueva interfaz gráfica de GitHub, sigan estos pasos para proteger sus branches:**  
Presten atención a las selecciones amarillas.

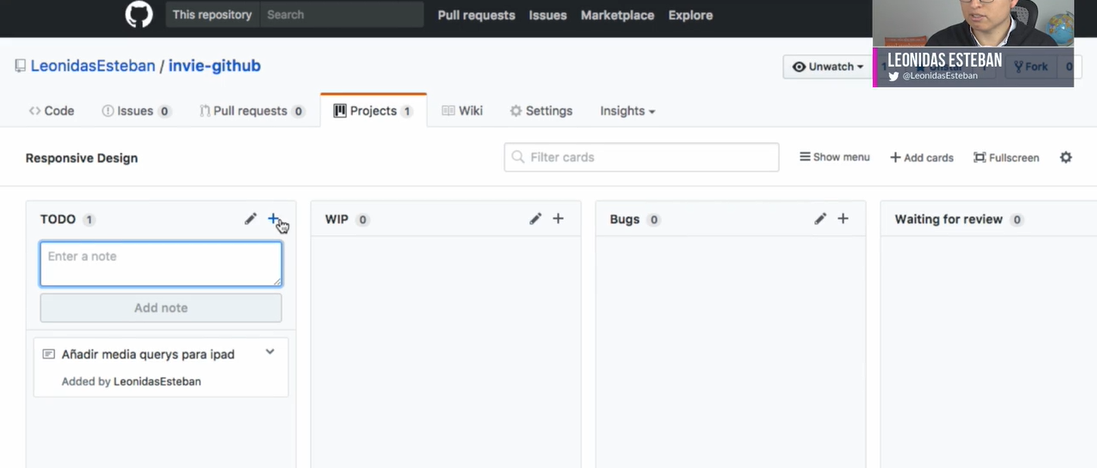
1. Ya en la pestaña de “Branches”, le damos click al botón *Add rule*.  
   
2. En el campo *Apply rule to* colocamos nuestra rama a proteger, en este caso, *master*. También seleccionamos los checks que @LeonidasEsteban nos indica en la clase.  
   Por último le damos click al botón *Create* que se encuentra al final.  
   
3. Así quedaría la pestaña de *Branches* cuando volvamos a entrar.  
   

**GitHub Projects, el SCRUM de GitHub**

GitHub tiene unas herramientas increíbles para nuestros proyectos. Una de estas herramientas es el projects de GitHub, la cual nos da la posibilidad de tener un mayor control del desarrollo de nuestro proyecto.

Aquí como en la metodología de SCRUM tenemos la posibilidad de saber qué tenemos pendiente por hacer, cuáles son las features que se encuentran en desarrollo, identificar los bugs que emergen de nuestro proyecto, y los features que debemos revisar.

Proyectos por features grandes.



**Columnas Basicas en tu proyecto:**

TODO: Son las cosas por hacer.

WIP: Work in Progress. En que estamos trabajando.

Bugs: Cosas que hay que arreglar de manera prioritaria.

Waiting for review: nos sirve para cuando trabajamos con los pulls requests, ya que se espera que alguien revise nustro codigo.

Done: Cuando la tarea ha finalizado.

**Creando un template para issues**

Lo ideal es que siempre que creemos un proyecto tengamos un template para enviar pull requests o cuando tengamos issues.

En GitHub puedes crear un template para los issues, basta con crear un archivo llamado issue\_template.md y definir los lineamientos para reportar un issue, ahora al generar un nuevo issue se mostrará el nuevo template.

Recuerda que los archivos .md por dentro contienen sintaxis markdown para dar estilos a tu texto, puedes usar esta guía

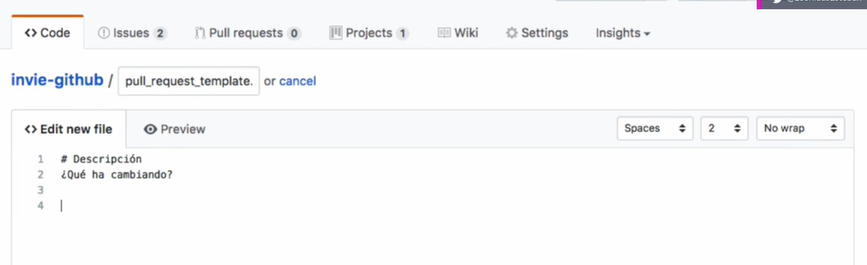
**markdown:** <https://github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet>

**Dar estilos al texto:** <https://guides.github.com/pdfs/markdown-cheatsheet-online.pdf>

**Creando un template para Pull Request**

GitHub permite usar templates a la hora de generar Pull Request

1. Crear archivo llamado pull\_request\_template.md
2. Definir los lineamientos usando sintaxis markdown
3. Hacer commit
4. Ahora todo será mas ordenado cuando generen contribuciones a tu proyecto.



Vida de una Pull Request

Las pull request no son un recordatorio o un log de funcionalidades en desarrollo. Cuando se crea una pull request se debe tener la convicción de que el código cumple con la funcionalidad definida y se generá una discusión para su aprobación.

No deben de quedar desarrollos pendientes.

-Se puede trabajar sobre una pull request abierta para mejoras o correcciones.

-Si fuera necesario hacer cambios o seguir desarrollando, se deberá cancelar la pull request, realizar los cambios correspondientes y después volverla a crear.

-La aprobación de la pull request no significa que se deba mergear automáticamente.

La vida útil de una pull request no debe más de una semana, dependiendo del proyecto y sus periodos de aprobación/despliegue. Una pull request que se dilata en el tiempo demasiado pierde su sentido, al no estar su código actualizado y poder generar conflictos. En caso de no poder hacer el merge por cualquier razón, se deberá de cerrar y cuando vuelva a ser posible hacerlo, actualizar la rama y abrir una nueva pull request.

Una vez se tenga la aprobación, el tiempo que debe transcurrir entre su merge con integration y su merge con master y futuro despliegue debe ser el menor tiempo posible, con el fin de evitar colisiones y que estás funcionalidades queden fuera de contexto.

Yo uso este template para pull requests, por si le sirve a alguien.

<!-- Please fill out the title field according to our pull-requests conventions -->

### Description

Fixes a bug where '...'

resolves #XXXXX

review @alguien

### Checklist

<!-- Mark these as checked by replacing [ ] with [x] -->

- [ ] Easy to rollback to the previous state

- [ ] Works fine with the old state in parallel (backward compatible)

- [ ] It doesn't depend on other branches/PR

- [ ] Has tests.

y si es un issue de frontend se adjunta screenshots.

Aquí esta el texto para el archivo **pull\_request\_template.md**

# Descripció**n**

¿**Que** ha cambiado?

- [ ] Frontend

- [ ] Backend

- [ ] Configuració**n** del Server

# ¿Cómo puedo probar los cambios?

**En** **que** URL y **forma** puedo ver **Update**

En un proyecto colaborativo tu vas a tener mas personas poniendole codigo a tu proyecto. Nadie puede ponerle codigo a master en este caso, ya que la hemos protegido. Para que alguien le ponga codigo a master primero tiene que crear una rama y solicitar un cambio, un pull request.

Por ejemplo: Crearia la rama responsive design y solicitaria el request de que se mezcle con la rama master. Entonces esto debe pasar por la review de mas personas o tu mismo.

Para esto tambien podemos crear un template.

Creamos el archivo como la otra vez con nombre pull\_request\_template.md

Para agregar ckecks usamos [ ] Frontend, Por ejemplo.

RECUERDA: Haz los templates segun las necesidades de tu proyecto

Listo, ahora cuando queramos enviar un pull request nos va a salir ese template:

Middlestone: Son grupos de issues

En este apartado se sigue la misma temática de la clase anterior, dónde la se propone la creación de plantillas, con el fin agilizar procesos, está se hace con los pulls requests; para el momento en que estos ocurre, es por que hemos separado la rama principal de desarrollo, de las contribuciones externas, así si alguien quiere hacerlas, deberá pedir por medio de una solicitud, los cambios que quiere hacer hacer, en que mejora al proyecto existente, y una dirección electrónica dónde los pueda observar, el administrador del repositorio.

Un pull request es una petición que el propietario de un fork de un repositorio hace al propietario del repositorio original para que este último incorpore los commits que están en el fork. En el caso que nos ocupa, el usuario aprendegit-user1 le enviará la petición a aalbagarcia para que este último incorpore los commits que tiene en su fork.

**.gitignore (Ignorando archivos no deseados)**

Si tienes archivos que no pueden ser públicos, como archivos de configuración con contraseñas, lo ideal es que no los subas a tu repositorio, estos archivos los puedes poner en el archivo .gitignore.