Git

Daniel Rojo Pérez

Índice

Breve historia de Git

Git es un sistema de control de versiones, si ponéis “man git” en el terminal, el nombre completo es: **git - the stupid content tracker**

Eso da una idea de cómo comenzó la historia de git. Hasta abril de 2005, el método de control de versiones que usaba el Kernel de Linux era un sistema propietario de la empresa BitKeeper. Esta empresa alegó que Andrew Tridgell estaba haciendo ingeniería inversa a su protocolo propietario, de modo que dejaba de prestar el servicio gratuito al Kernel de Linux.

Breve historia de Git

En este punto Linus Torvalds, que no tenía que estar muy contento con la forma en la que hacían el control de versiones con BitKeeper, se negó a pagar por el servicio y comenzó el desarrollo de git en abril de 2005. En julio, ya tenían la primera RC de la versión 0.99, hasta la versión definitiva 0.99.9n en diciembre de 2005. La versión 1.0.13 salió a finales de enero de 2006.

La versión estable más actual es la 2.35.1, publicada en enero de 2022

Instalación

Es necesario instalar un cliente de Git

Vamos a usar visual studio code

Podemos instalar algún cliente gráfico de Git como GitKraken

Podemos crear nuestro propio servidor de Git muy fácilmente en Linux o usar algún servicio de Git en la red, los más populares son: Github, Bitbucket, Gitlab

Vamos a hacernos una cuenta en gitlab

Configurando una clave ssh

Con ssh-keygen generamos una clave ssh

➢ ssh-keygen -o -t rsa -b 4096 -C "email@example.com"

Añadir clave que se genera en el archivo .pub a las claves de ssh de la cuenta de gitlab, dentro del apartado de settings.

Comprobar que todo está correcto

➢ `ssh -T git@gitlab.com`

Configuración general de Git en local

Configuramos nuestro nombre de usuario

```
➢  git config --global user.name “[firstname lastname]”
```

Configuramos una cuenta de email para nuestro usuario

```
➢  git config --global user.email “[valid-email]”
```

Configuramos el coloreado de las respuestas de git

```
➢  git config --global color.ui auto
```

Configuración general de Git en local

Se puede elegir el editor de texto por defecto cuando Git necesite que introduzcas un mensaje

```
➢  git config --global core.editor emacs
```

Para mostrar todas las propiedades que Git ha configurado

```
➢  git config --list
```

Para ver la página del manual para el comando config

```
➢  git help config
```

Crear un repositorio

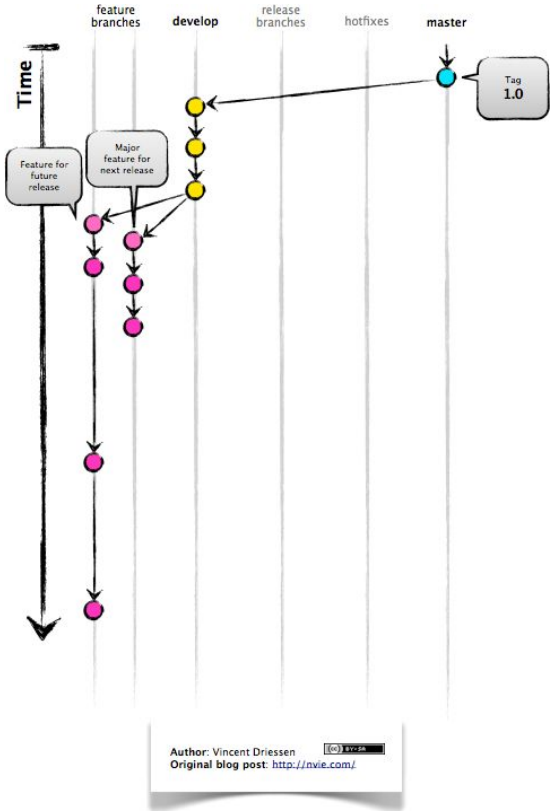
Podemos crear un repositorio desde cero

```
➢  git init [project-name]
```

Podemos clonar uno ya existente

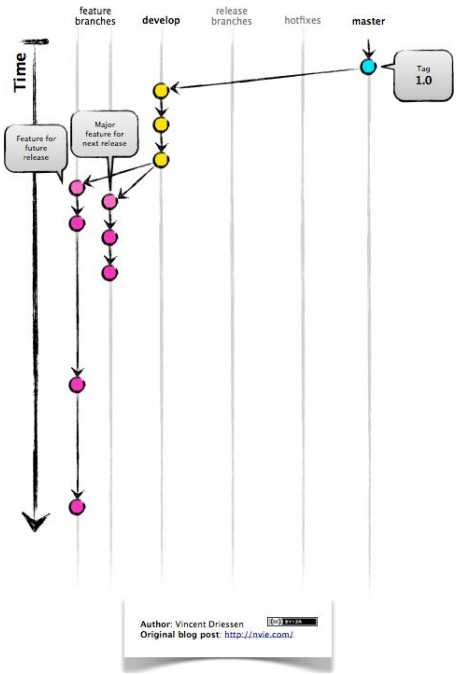
```
➢  git clone [url]
```

Aspecto de un repositorio Git

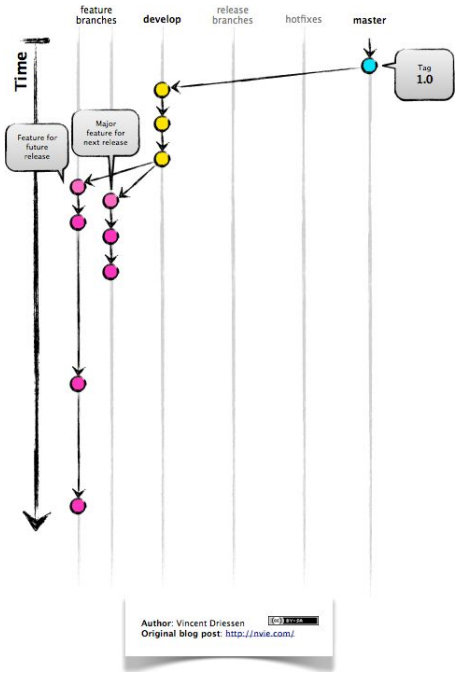


Remoto y local en Git

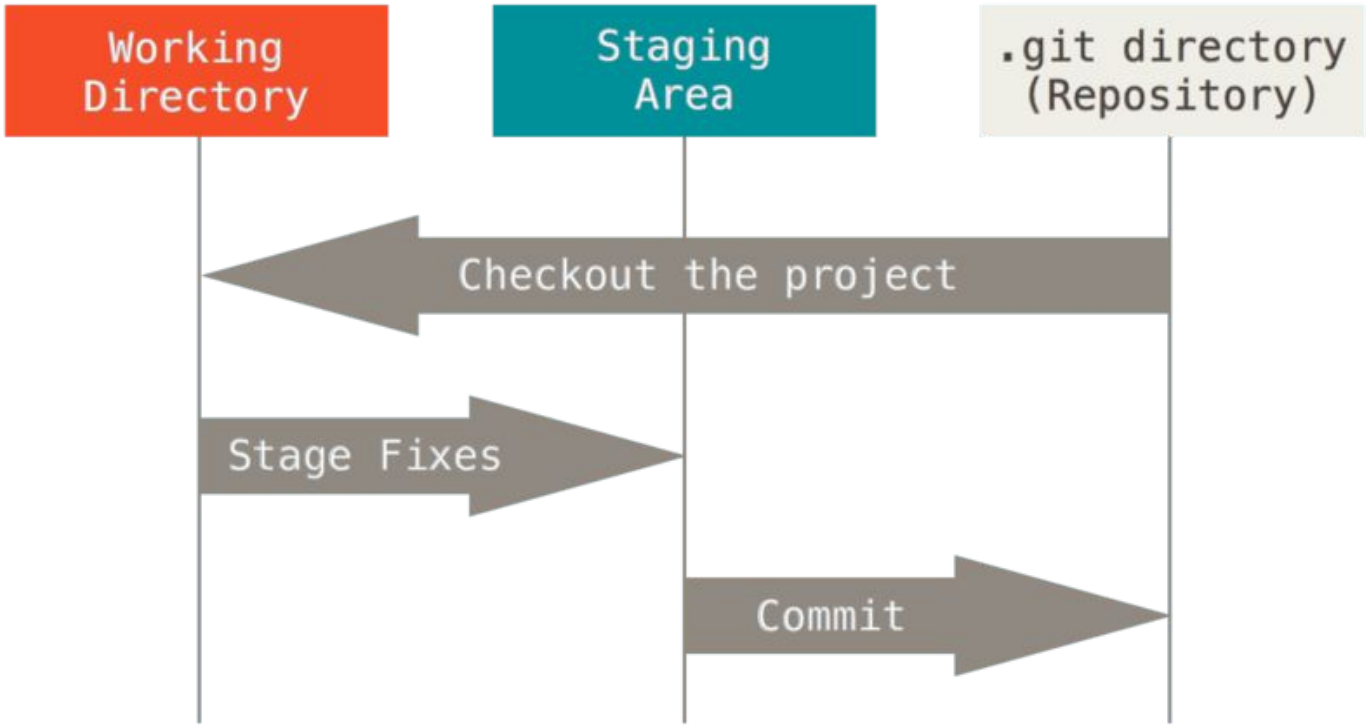
Local



Remoto



Los tres estados de git



.gitignore

Es un archivo de configuración que establece archivos o directorios que deben ignorarse para ser incluidos en el stage de git

- **directorio/ignorado/** Ignora el contenido de un directorio completo
- ***.ext** Ignora todos los archivos con extensión .ext
- **doc/**/*.txt** Ignora todo archivo .txt dentro de cualquier subdirectorio de /doc
- **!archivo.ext** Establece una excepción para el archivo con nombre archivo.ext
- **!*.dat** Establece una excepción para todos los archivos con extensión .dat
- **#** Marca una línea de comentario

Trabajando con commits

Nos muestra los cambios en los archivos del repositorio

➢ `git status`

Añade a stage un archivo

➢ `git add [file]`

Elimina de stage un archivo (no se borra del directorio)

➢ `git reset [file]`

Nos muestra los archivos que están en stage que no se han añadido a un commit

➢ `git diff --staged`

Crea un commit con una **descripción de los cambios introducidos**

➢ `git commit -m “[descriptive message]”`

Mostrar cambios e historial en un repositorio

Podemos listar los commits en un repositorio

```
➢  git log
```

Podemos ver los cambios en un archivo

```
➢  git log --follow [file]
```

Podemos ver los cambios en archivos de nuestro repositorio

```
➢  git diff
```

Con este comando vemos los cambios de un commit

```
➢  git show [commit]
```

Git diff

Para ver qué ha cambiado pero no está en stage

➢ `git diff`

Si quieres ver lo que hay en stage y será incluido en el commit (--cached)

➢ `git diff --staged`

Mostrar los archivos con conflictos

➢ `git diff --name-only --diff-filter=U`

Mostrar los cambios en stage y fuera de stage

➢ `git diff HEAD`

Mostrar los cambios en un commit

➢ `git diff-tree --no-commit-id --name-only -r <commit-id>`

Git log

Podemos listar los commits en un repositorio

➢ `git log`

Con el parámetro -p muestra las diferencias introducidas en cada commit

➢ `git log -p`

Para ver la información extendida de cada commit

➢ `git log --stat`

En un intervalo de tiempo --since, --until: minutes, hours, weeks, months, years

➢ `git log --since=2.hours`

Muestra commits con un formato personalizado

➢ `git log --pretty=format:"%h %s" --graph`

➢ `git log --pretty=oneline`

Commits

Crear un commit:

➢ `git commit -m 'initial commit'`

Para modificar el último commit, puedes hacerlo con la opción --amend:

➢ `git commit --amend`

Para sacar un archivo de stage:

➢ `git reset HEAD <archivo>`

Para descartar los cambios de un archivo fuera de stage:

➢ `git checkout -- <archivo>`

Eliminación o cambios de archivos

Borra el archivo del directorio y deja instancia en stage de la eliminación

➢ `git rm [file]`

Conserva el archivo del directorio pero deja de pertenecer al repositorio

➢ `git rm --cached [file]`

Cambia el nombre del archivo

➢ `git mv [file-original] [file-renamed]`

Elimina los commits posteriores al especificado, se mantienen los archivos

➢ `git reset [commit]`

Trabajando con stash

Guardar los cambios en stash

➢ `git stash`

Listar los cambios almacenados en stash

➢ `git stash list`

Recuperar los archivos guardados en stash

➢ `git stash pop`

Descartar los cambios en stash

➢ `git stash drop`

Alias de Git

Para crear un alias en global

```
➢  git config --global alias.unstage 'reset HEAD --'
```

Para crear un alias en local

```
➢  git config --local alias.unstage 'reset HEAD --'
```

Para ejecutar un comando externo se puede comenzar el comando con un carácter !

```
➢  git config --global alias.visual "!gitk"
```

Tags

Mostrar los tags del repositorio

```
➢  git tag
```

Mostrar los tags a partir de una etiqueta

```
➢  git tag -l 'v1.8.5*'
```

Crear un tag con una nota

```
➢  git tag -a v1.4 -m 'my version 1.4'
```

Etiquetar un commit

```
➢  git tag -a <tag> <commit_id>
```

Tags

Por defecto, el comando git push no transfiere las etiquetas a los servidores remotos. Hay que enviar las etiquetas de forma explícita al servidor.

Este proceso es similar al de compartir ramas remotas - puedes ejecutar git push origin [etiqueta].

➢ `git push <nombre_remoto> <tag>`

Para enviar todas las etiquetas al remoto

➢ `git push <nombre_remoto> --tags`

Tags

En Git, no puedes sacar una etiqueta, pues no es algo que puedas mover. Si quieres colocar en tu directorio de trabajo una versión de tu repositorio que coincida con alguna etiqueta, debes crear una rama nueva con esa etiqueta

➢ `git checkout -b <nombre_rama> <version>`

Ramas en Git

Lista todas las ramas del repositorio

➢ `git branch`

Crea una nueva rama

➢ `git branch [branch-name]`

Cambia a una rama que especifiquemos

➢ `git checkout [branch-name]`

Une la rama actual con la que especificamos, modificamos siempre la actual

➢ `git merge [branch]`

Trabajar con repositorio remoto

Mostrar remotos:

➢ `git remote -v`

Añadir un repositorio remoto:

➢ `git remote add <nombre_repo> <url_repo>`

Traer la información de remoto:

➢ `git fetch <nombre_repo>`

Enviar de local a remoto:

➢ `git push <nombre_repo> <rama_repo>`

Eliminar un remoto:

➢ `git remote rm <nombre_repo>`

Sincronización de cambios en un repositorio

Descarga la historia del repositorio desde la localización remota

```
➢  git fetch [bookmark]
```

Combina la rama remota con la rama local

```
➢  git merge [bookmark]/[branch]
```

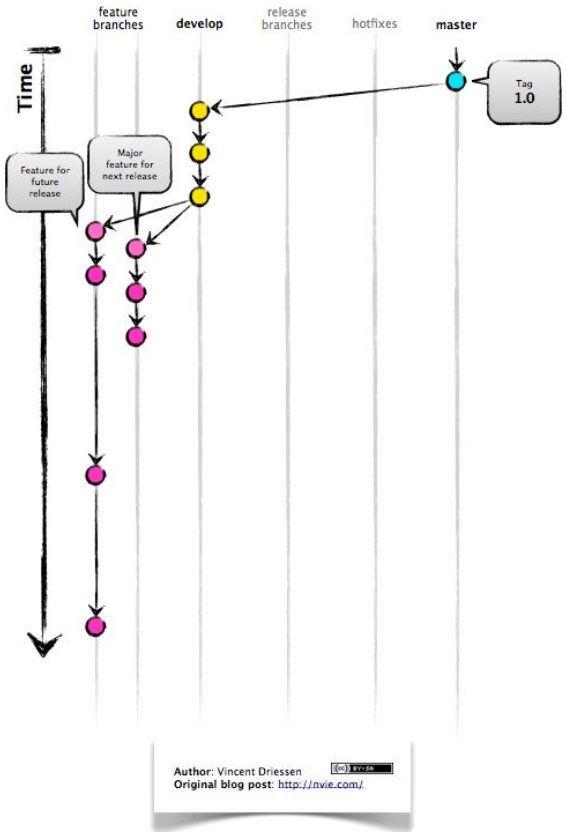
Sube la rama actual a remoto, especificando alias de remoto y rama

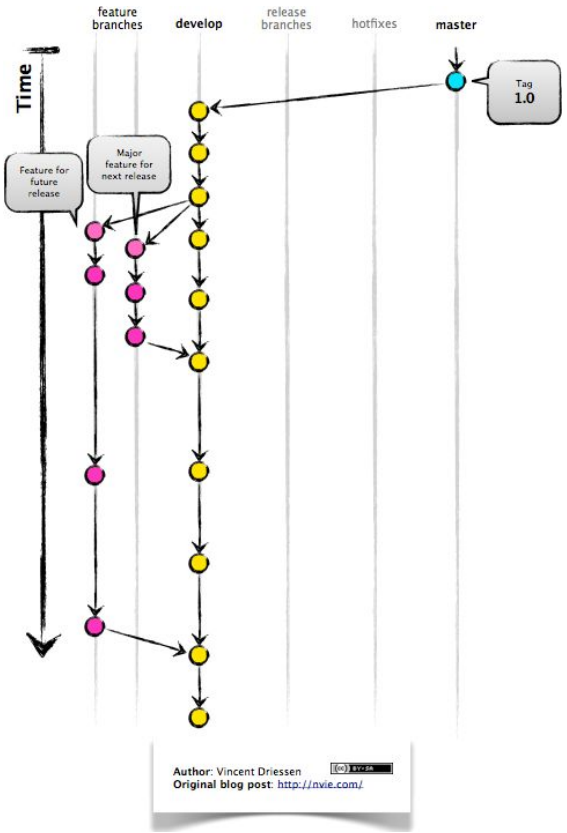
```
➢  git push [alias] [branch]
```

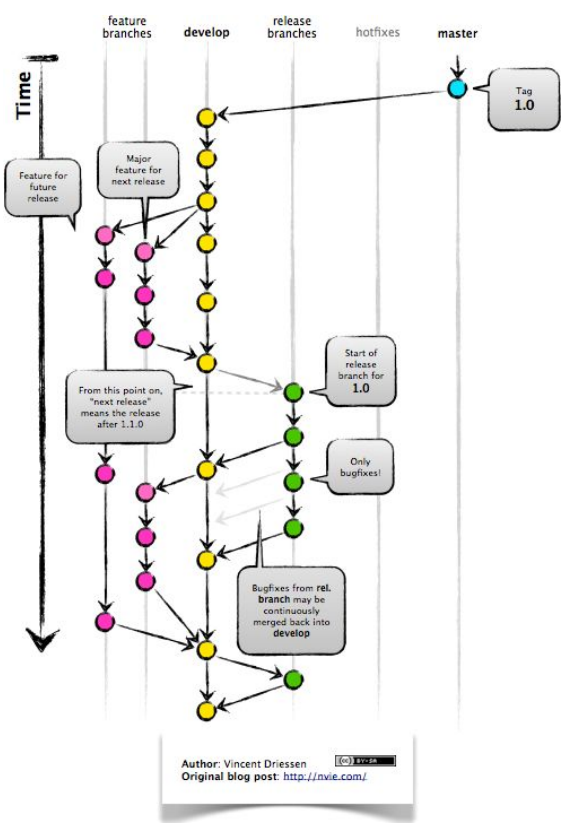
Combina fetch y merge

```
➢  git pull
```

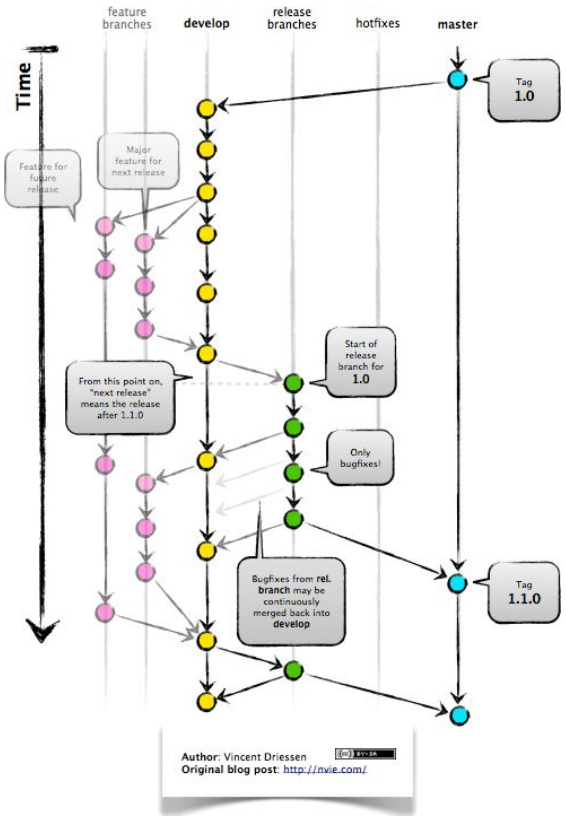
Gitflow

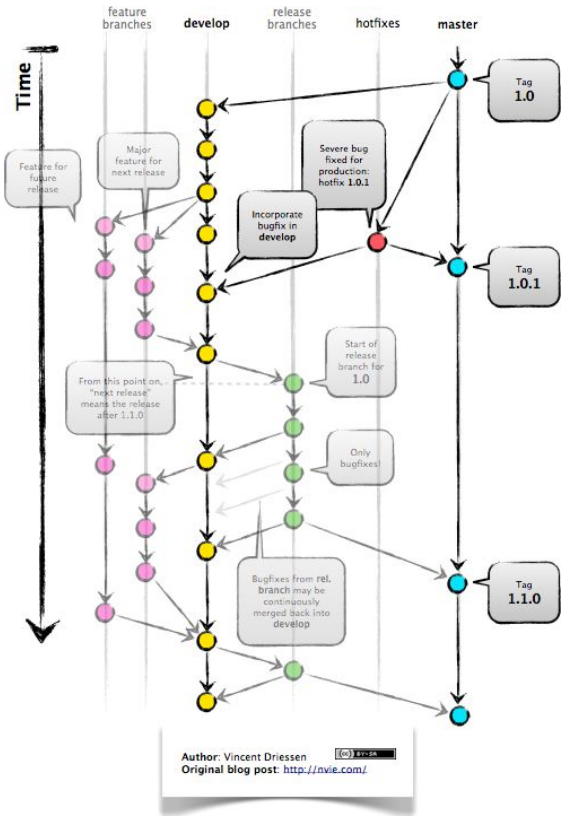

feature branches
develop
release branches
hotfixes
master
Time
Tag 1.0
Feature for future release
Major feature for next release
Author: Vincent Driessen
Original blog post: http://nvie.com/


feature branches
develop
release branches
hotfixes
master
Time
Tag 1.0
Feature for future release
Major feature for next release
Author: Vincent Driessen
Original blog post: http://nvie.com/


feature branches
develop
release branches
hotfixes
master
Time
Tag 1.0
Feature for future release
Major feature for next release
From this point on, "next release" means the release after 1.1.0
Start of release branch for 1.0
Only bugfixes!
Bugfixes from rel. branch may be continuously merged back into develop
Author: Vincent Driessen
Original blog post: http://nvie.com/

Gitflow


feature branches
develop
release branches
hotfixes
master
Time
Tag 1.0
Feature for future release
Major feature for next release
Start of release branch for 1.0
From this point on, "next release" means the release after 1.1.0
Only bugfixes!
Bugfixes from rel. branch may be continuously merged back into develop
Tag 1.1.0
Author: Vincent Driessen
Original blog post: http://nvie.com/


feature branches
develop
release branches
hotfixes
master
Time
Tag 1.0
Feature for future release
Major feature for next release
Severe bug fixed for production: hotfix 1.0.1
Incorporate bugfix in develop
Tag 1.0.1
Start of release branch for 1.0
From this point on, "next release" means the release after 1.1.0
Only bugfixes!
Bugfixes from rel. branch may be continuously merged back into develop
Tag 1.1.0
Author: Vincent Driessen
Original blog post: http://nvie.com/

Submódulos git

A menudo ocurre que mientras trabaja en un proyecto, necesita usar otro proyecto desde dentro (una biblioteca externa o un desarrollo de otro equipo). Un problema común surge en este escenario: poder tratar los dos proyectos como separados y aún así poder usar uno desde el otro.

Git aborda este problema utilizando submódulos. Los submódulos permiten mantener un repositorio de Git como un subdirectorio de otro repositorio de Git. Esto le permite clonar otro repositorio en su proyecto y mantener sus commits separados.

Submódulos git

Usa el comando git submodule add con la URL del proyecto que desea empezar a enlazar.

> git submodule add https://github.com/inventree/InvenTree.git

Por defecto, los submódulos agregarán el subproyecto a un directorio con el mismo nombre que el repositorio, en este caso “InvenTree”.

.gitmodules es un archivo de configuración que almacena la asignación entre la URL del proyecto y el subdirectorio local en el que lo ha insertado. Si tienes múltiples submódulos, tendrás múltiples entradas en este archivo

Submódulos git

Si clonamos un proyecto con un submódulo, por defecto se obtienen los directorios que contienen submódulos, pero ninguno de los archivos dentro de ellos aún. Hay que ejecutar dos comandos:

> git submodule init

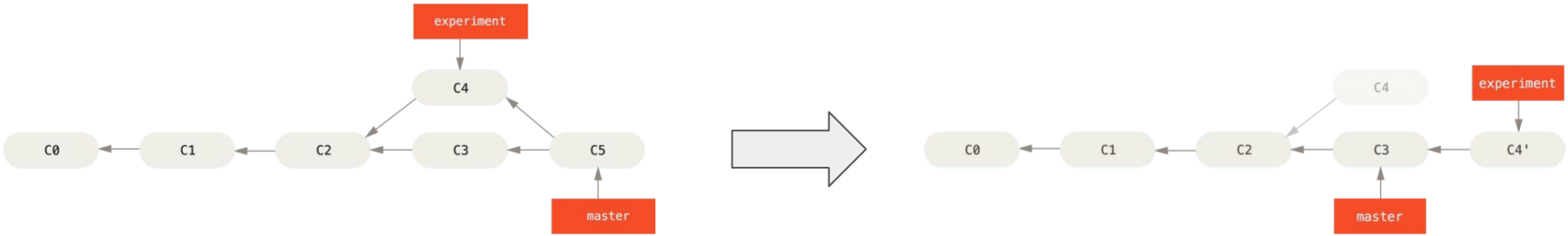
> git submodule update

hay otra manera de hacer esto que es un poco más simple. Se pasa --recursive al comando git clone:

> git clone --recursive https://github.com/chaconinc/MainProject

Git rebase

Con el comando git rebase, puedes capturar todos los cambios confirmados en una rama y reaplicarlos sobre otra.



Git squash

Se trata de agrupar varios commits en uno, se puede hacer proporcionando un argumento para el comando git rebase -i HEAD~N, siendo el número de commits atrás. Por ejemplo, para hacer un squash de los tres últimos commits:

```
> git rebase -i HEAD~3
```

Al ser un cambio de historia de tu rama, si has hecho push a remote antes de hacer squash, generas un conflicto con la historia de remoto.

Para confirmar el squash:

```
> git rebase --continue
```

Git cherry-pick

En algunas situaciones al estar trabajando en diferentes ramas de nuestro proyecto, nos hemos encontrado con situaciones en donde solo queremos capturar un commit en específico de una rama y pasarlo a la otra, ya que no queremos hacer un merge entre ambas ramas porque estarían pasando otros cambios que no deseamos. Para estos casos nos puede ayudar el comando “cherry-pick”.

```
> git cherry-pick <ID_commit>
```

Git cherry-pick

Es posible que al realizar un cherry-pick nos encontremos con conflictos, esto sucede cuando se ha editado el mismo archivo por otros usuarios.

Para este caso:

- Debemos resolver los conflictos: git mergetool
- Agregamos estos cambios: git add --all
- Aplicamos el comando: git cherry-pick --continue

Git blame

Git blame proporciona metadatos del autor adjuntos a líneas específicas en un commit de un archivo. Se usa para examinar puntos específicos del historial de un archivo y poner en contexto quién fue el último autor que modificó la línea y responder preguntas sobre qué código se ha añadido a un repositorio, cómo se ha añadido y por qué. Git blame se suele usar con una visualización de interfaz gráfica de usuario, pero se puede usar mediante comandos también. Ejemplos:

> git blame README.md <hash_commit>

> git blame -L 1,5 README.md

> git blame --since=3.weeks README.md