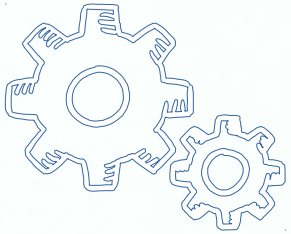
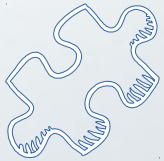


GIT... Y TODO ESO DEL CONTROL DE VERSIONES

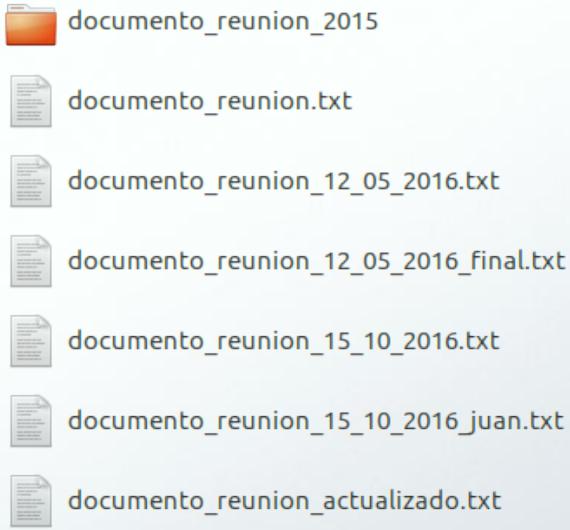


¿Por qué sistemas de control de versiones?

+ Grabar cambios y poder volver a versiones anteriores
+ Comparar cambios
+ Permite recuperar después de un cambio desastroso
+ Facilita el trabajo colaborativo

¿Por qué sistemas de control de versiones?

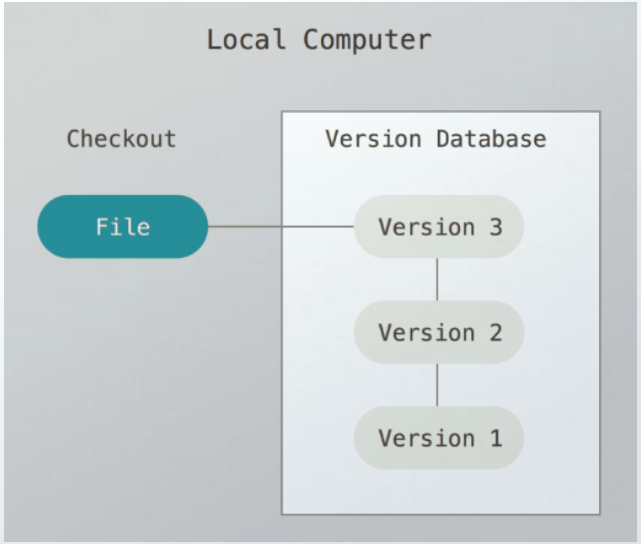
Para no hacer más esto:



Sistemas de control de versiones (VCS's)

Local (VCS)

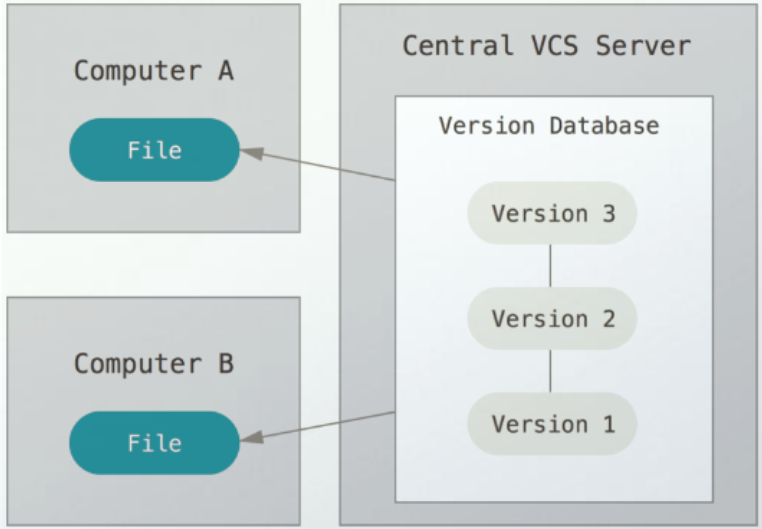
+ Base datos de cambios
+ Mejor que mantener copias manualmente
+ No colaborativo
+ No es robusto a fallos



Centralizado (CVCS)

+ Colaborativo
+ Es más fácil de administrar
+ Mejor división de trabajo
+ No es robusto frente a fallas
+ Dependencia de servidor
+ Dependencia de conectividad

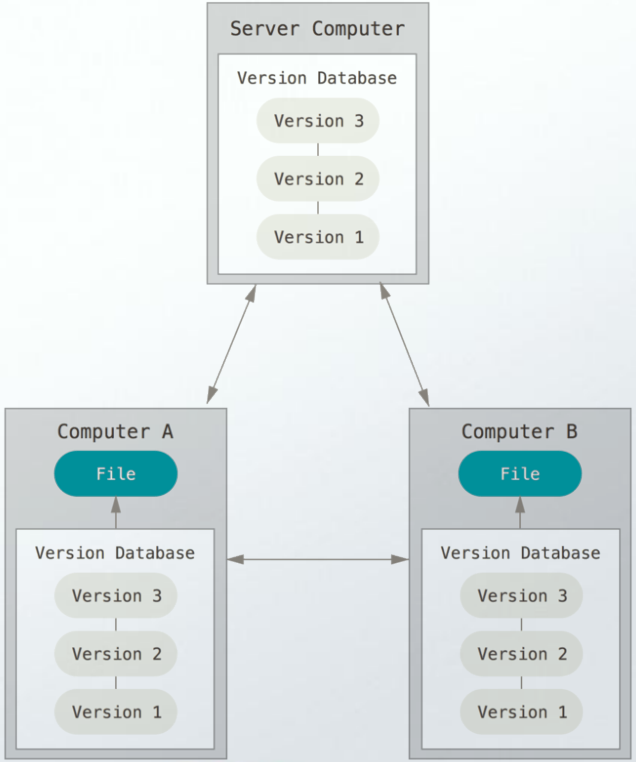
CVS, Subversion, Perforce



Distribuido (DVCS)

+ Colaborativo
+ Fácil administración
+ Robusto frente a fallas
+ Repositorio central opcional

Git, Mercurial, Bazaar, Darcs



GIT!

- Libre y de código abierto
- Rápido y eficiente
- Totalmente distribuido
- Gran capacidad de *branching*
- Seguro y robusto

Empezando con git

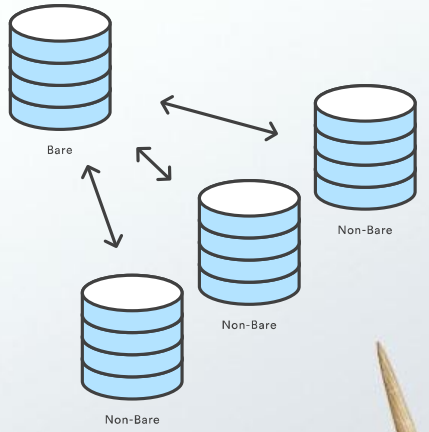
Los primeros comandos en git

Creando un repo en el server

```
git init --bare <nombre_de_repo>
```

inicializa un repositorio para compartir
– no crea el 'directorio de trabajo'

Por convención, se pone .git como extensión a repos *bare*



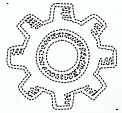
Clonando un repo en nuestra máquina

`git clone <ubicacion_del_repo>`

Inicializa un repositorio para trabajar
- Se crea un directorio de trabajo y un directorio .git con el repositorio

Notar como el sufijo '.git' desaparece en el nombre del directorio creado

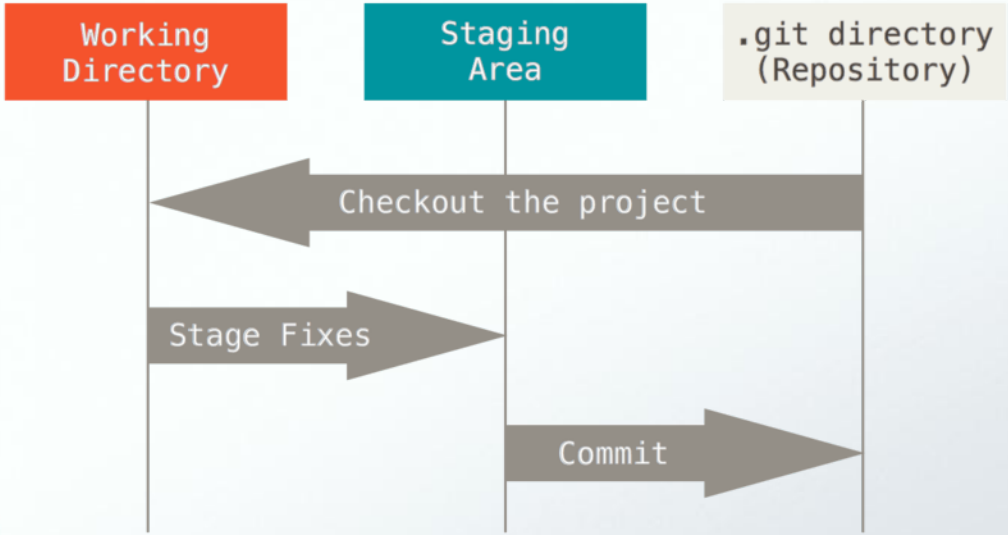
Configurando git



```
git config --global user.name <nombre>
git config --global user.email <email>

git config --system core.editor <editor>
```

Con git config --global --edit podemos editar manualmente el archivo global de configuración

"Son 3 cosas"


Working Directory
Staging Area
.git directory (Repository)
Checkout the project
Stage Fixes
Commit

Add y commit

Nuestro primer commit

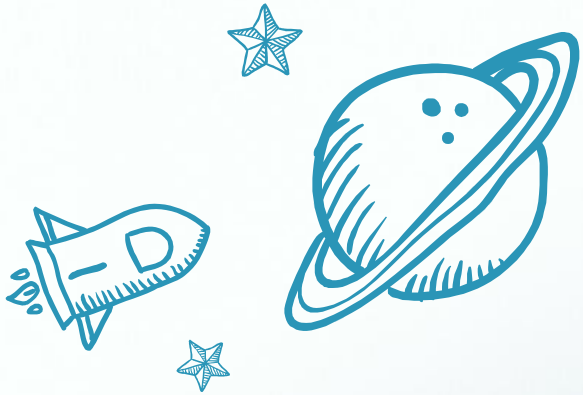
`git add <file>`

Agrega <file> a la "staging area". Esto nos permite armar el siguiente snapshot

`git commit`

Todo lo que estaba en el "staging area" se registra en el repositorio

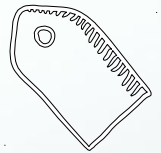
git no hace diferencia entre agregar archivos nuevos y archivos modificados que ya forman parte del repo



Los commits son únicos

Los commits no cambian

ID = CONTENIDO + AUTOR + FECHA + MENSAJE + ID DE COMMIT ANTERIOR +



24b9da6552252987aa493b52f8696cd6d3b00373

ATAJOS

`git commit -m <message>`

No abre el editor, el mensaje se escribe inline

`git commit -a`

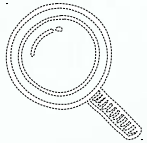
Commitea un snapshot de todos los cambios en el directorio de trabajo. Sólo se incluyen las modificaciones a los archivos "trackeados" (aquellos que fueron agregados con git add en algún punto en la historia)

Generalmente, conviene hacer commits atómicos (de cambios relacionados entre sí)

Metodología de trabajo: commits

- Recordar: commits son *locales*
- Commits diferentes para cambios lógicamente separados
- Limpiar el trabajo previo a publicarlo
- Mensajes con cierta estructura

Ignorando archivos



Tipos de archivos en el directorio de trabajo:

- **Tracked** – archivos que fueron agregados o comiteados previamente
- **Untracked** – archivos que NO fueron agregados o comiteados nunca
- **Ignorados** – archivos que fueron indicados especialmente para que git ignore

¿Qué es útil ignorar?

- Código compilado, como archivos `.o`, `.pyc` y `.class`
- Directorios completos como `/out`, `/bin`, `/target`
- Archivos generados en tiempo de ejecución:
- `.lock`, `.tmp` y `.log`
- Archivos ocultos como: `Thumbs.db`
- Etc.

.gitignore ← ¡¡ Archivo especial !!

```
# Byte-compiled / optimized / DLL files
__pycache__/
*.py[cod]
*$py.class
```

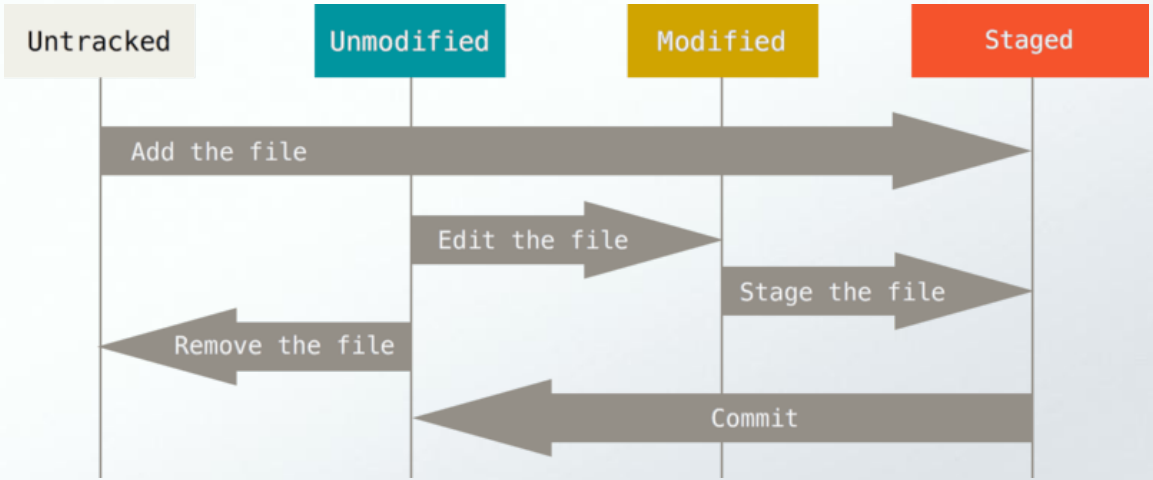
```
# Logs fuera. Mantener important, salvo trace
*.log
!important/*.log
trace.*
```

Más info: https://git-cm.com/docs/gitignore

Explorando los cambios: git status

`git status`

Muestra archivos modificados, staged y untracked



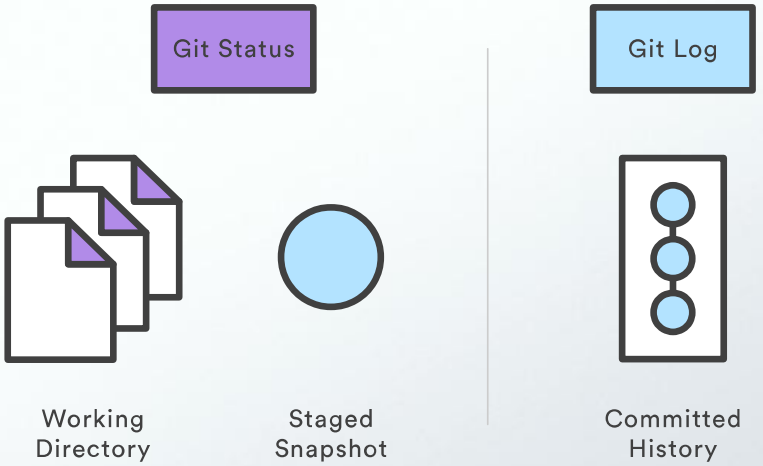
La práctica hace al maestro

1. Crear un archivo nuevo en el directorio de trabajo: `resumen.txt`
2. Mirar el status del repo (probar `git status --short` también)
3. Agregar el archivo `resumen.txt` al staging area
4. Mirar el status del repo
5. Agregar `hello.py` al staging area
6. Mirar el status del repo
7. Hacer una ligera modificación a `hello.py`
8. Mirar el status del repo
9. Agregar `hello.py` al staging area
10. Finalmente commitear cambios al repo

Explorando el repo: git log

`git log`

Muestra el historial de commits en el repo



git log

`git log -n <limit>`
Limita el número de commits a mostrar

`git log --oneline`
Cada commit ocupa una sola línea

`git log --stat`
Incluye estadísticas de líneas agregadas y removidas

`git log -p`
Incluye los diffs de cada commit. Mucha información

`git log --grep="<pattern>"`
Commits con mensajes siguiendo pattern

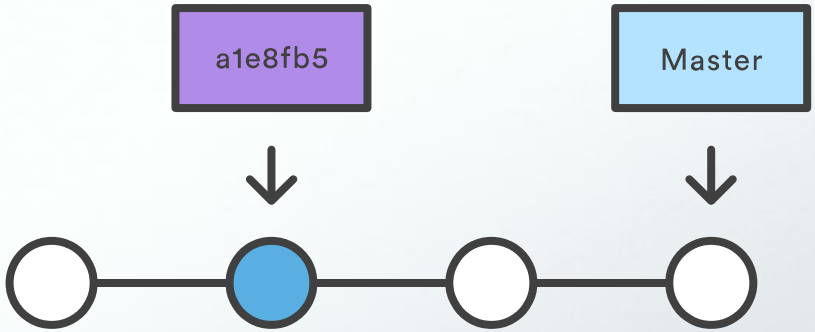
`git log <file>`
Para ver el historial de un archivo en particular

`git log --graph --decorate --oneline`

Moviéndonos por el repo: git checkout

`git checkout`

Sirve para restaurar un commit, un archivo o un branch



git checkout

`git checkout <branch>`

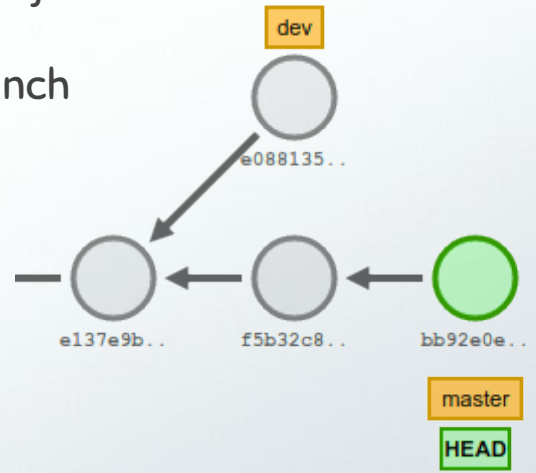
Restaura branch al directorio de trabajo

`git checkout -b <branch>`

Crea branch y mueve HEAD a ese branch

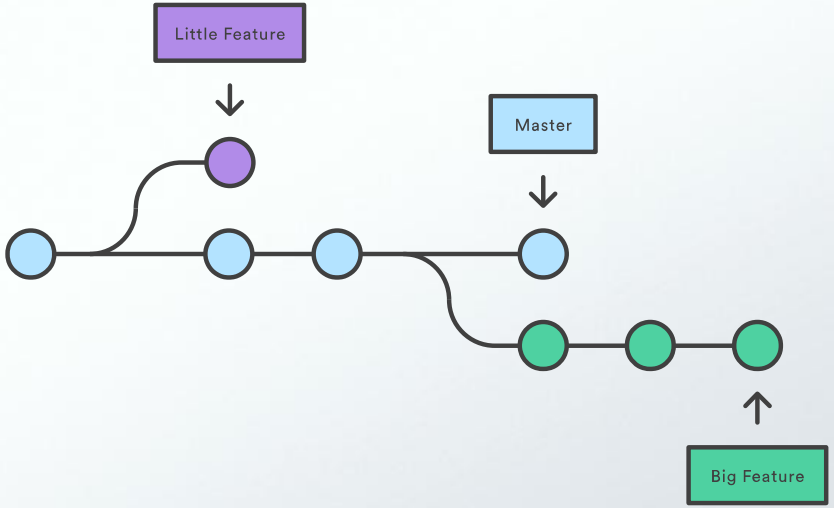
Un momento...¿Qué es HEAD?
¿Y master, branch?

HEAD es una referencia al último commit del branch actual



Branchs

Un branch es una línea independiente de desarrollo en la historia del repositorio



git branch

`git branch`

Lista los branchs existentes

`git branch <branch>`

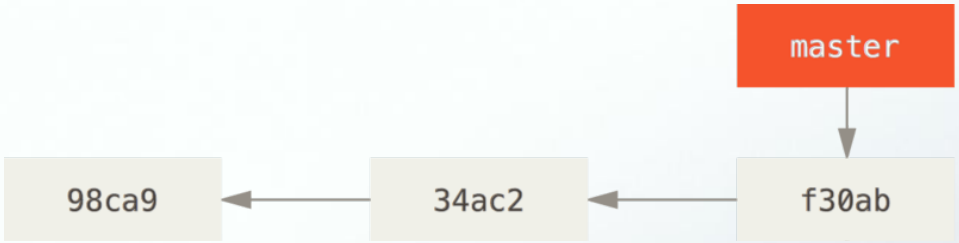
Crea un branch. No hace un checkout

`git branch -d <branch>`

Borra un branch si toda historia ha sido completamente mergeada.
Es una operación segura en ese sentido.

Branchs: ¿Cómo funcionan?

Tenemos nuestro querido master. Queremos implementar opciones de testing, sin modificar master



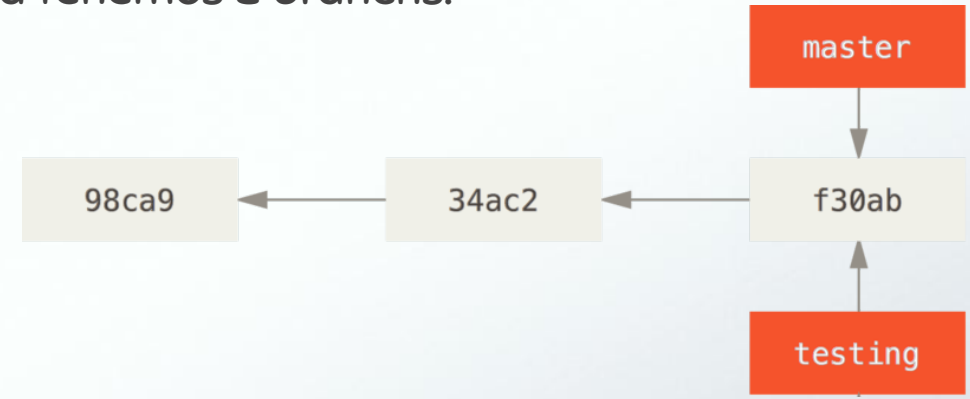
Tipeamos

```
git branch testing
```

Branchs: ¿Cómo funcionan?

¡Apareció testing!

Ahora tenemos 2 branchs:



Pregunta: ¿En qué branch estamos?

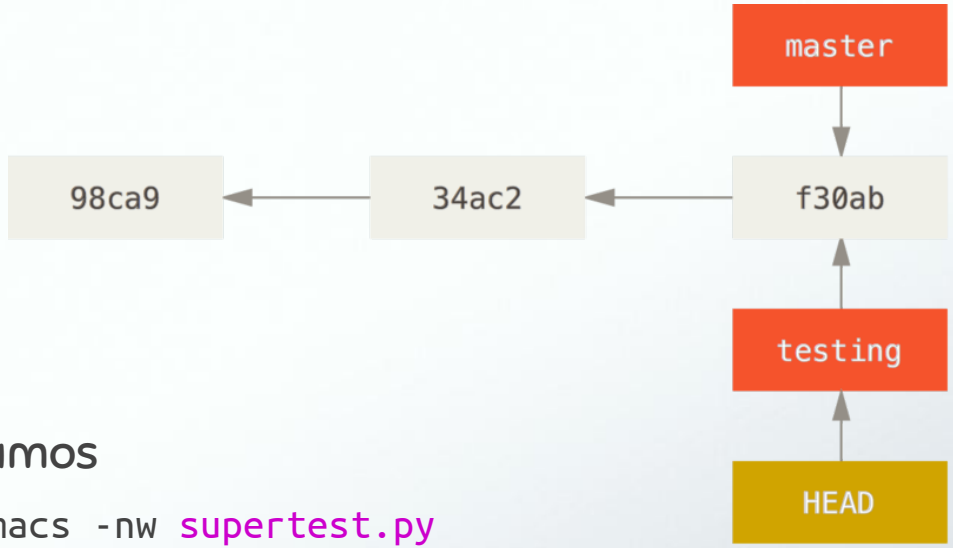
Branchs: ¿Cómo funcionan?

HEAD marca en que branch estamos

Tipeamos

```
git checkout testing
```

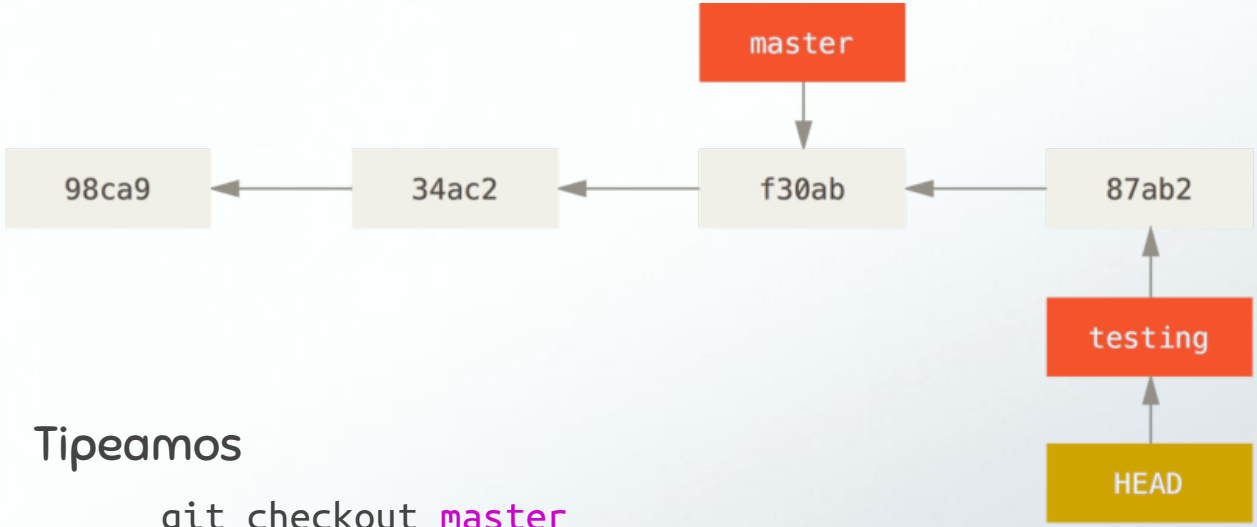
Branchs: ¿Cómo funcionan?



Tipeamos

```
emacs -nw supertest.py
git add supertest.py
git commit -m "Creado super test"
```

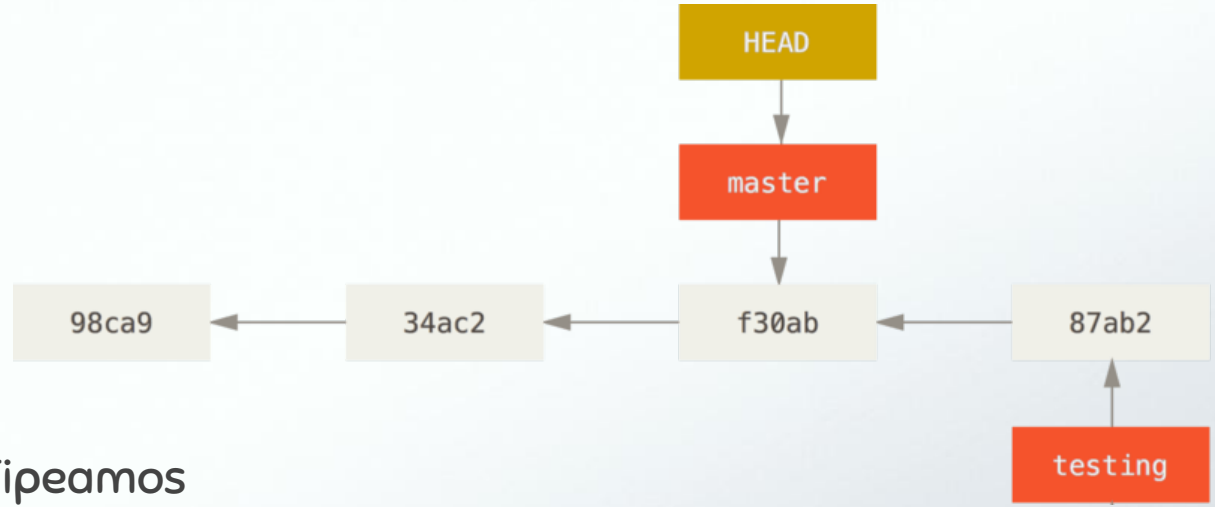
Branchs: ¿Cómo funcionan?



Tipeamos

```
git checkout master
```

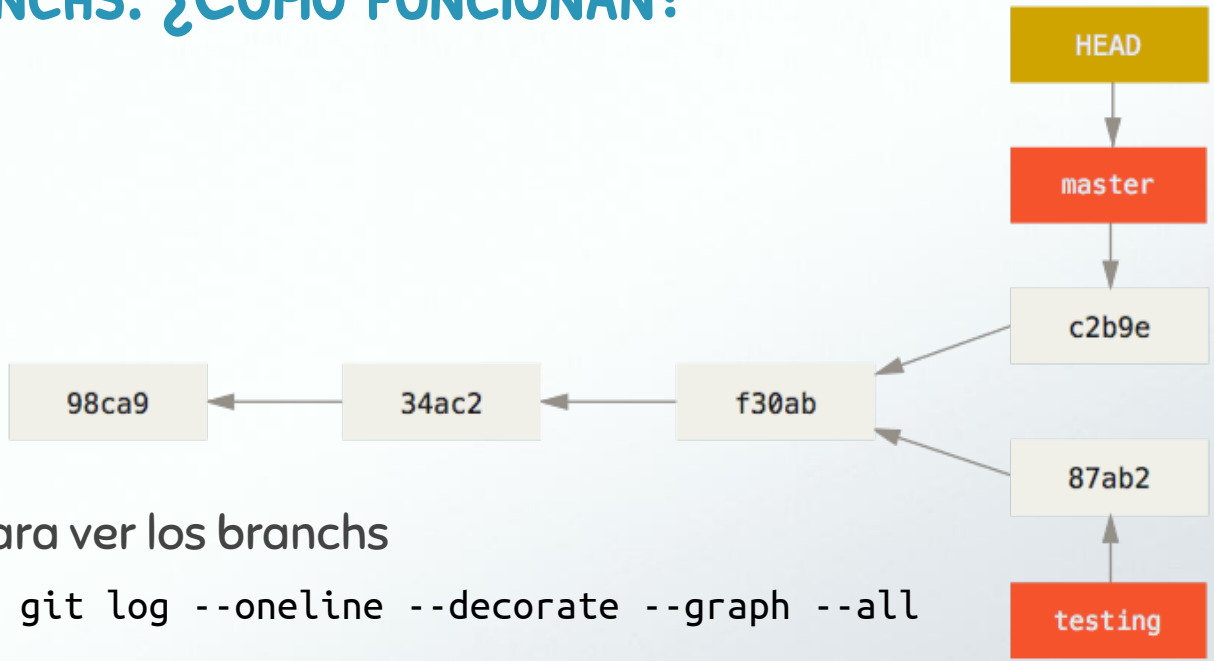
Branchs: ¿Cómo funcionan?



Tipeamos

```
emacs -nw VERSION
git add VERSION
git commit -m ”Se agrega archivo de VERSION”
```

Branchs: ¿Cómo funcionan?



Para ver los branchs

```
git log --oneline --decorate --graph --all
```

Práctica

- Clonamos el repo:

  https://github.com/onlywei/explain-git-with-d3.git

- Abrimos index.html y en zen mode, practicamos crear branches y ver cómo se actualiza el árbol

- Modo git master: usamos el repo existente y visualizamos los cambios con git log

Mirando viejos commits...

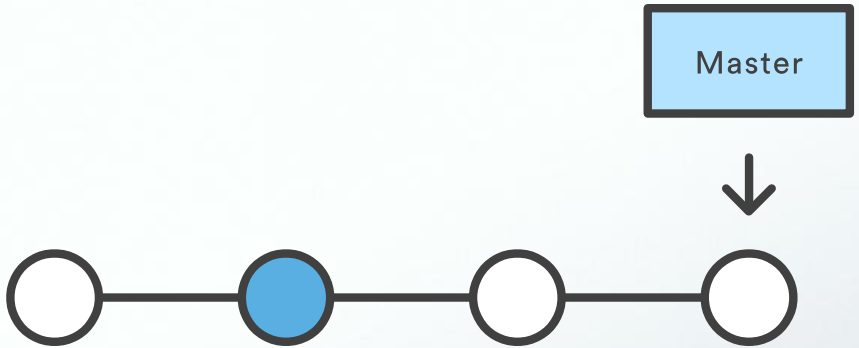
Para restaurar ese commit hacemos:

```
git checkout fa3f1
```

git nos responde:

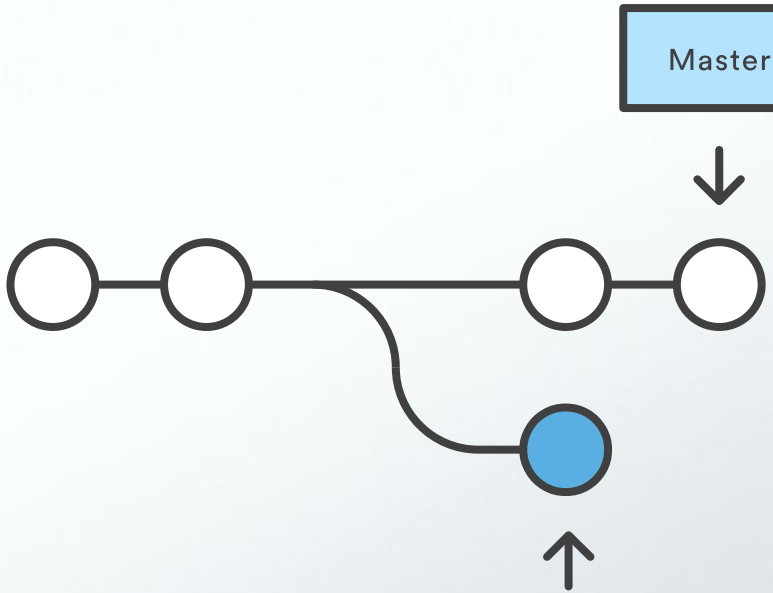
```
You are in 'detached HEAD' state. You can look around, make
experimental changes and commit them, and you can discard any
commits you make in this state without impacting any branches by
performing another checkout.

If you want to create a new branch to retain commits you create,
you may do so (now or later) by using -b with the checkout command
again. Example:

  git checkout -b <new-branch-name>
```

Detached HEAD



No es conveniente hacer commits en este estado

Detached HEAD

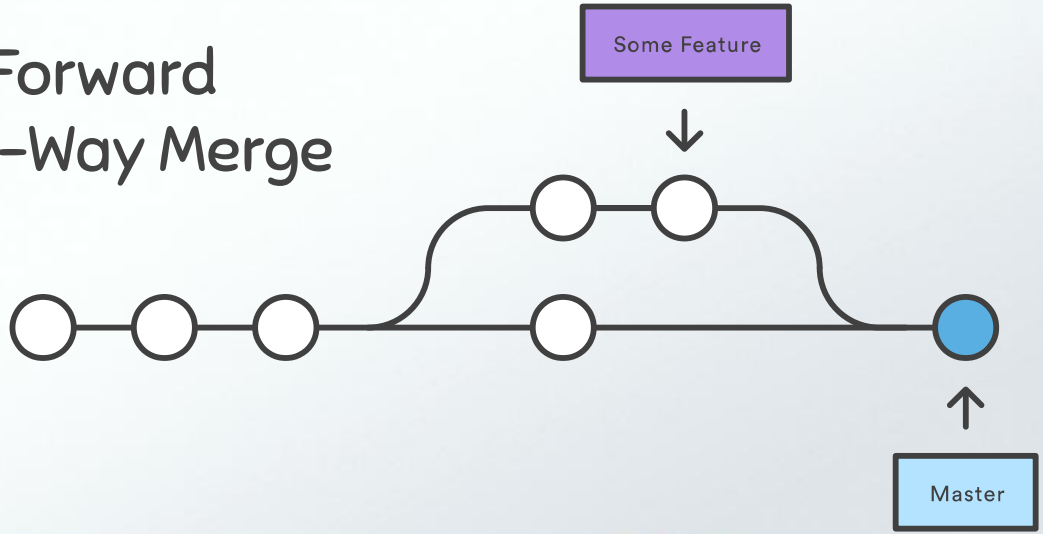


No es posible mergear este commit

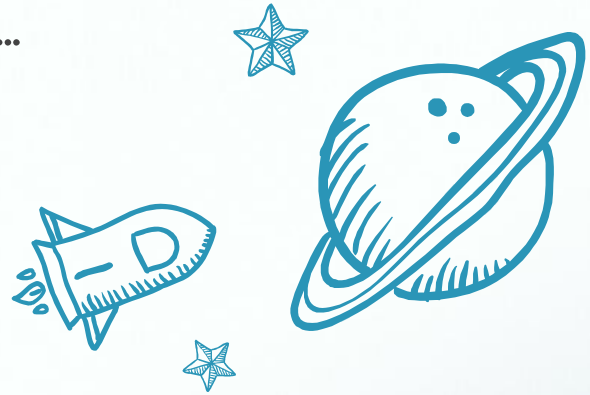
GIT MERGE

Nos permite integrar ramas de desarrollo independiente

- Fast-Forward
- Three-Way Merge



Antes de empezar...

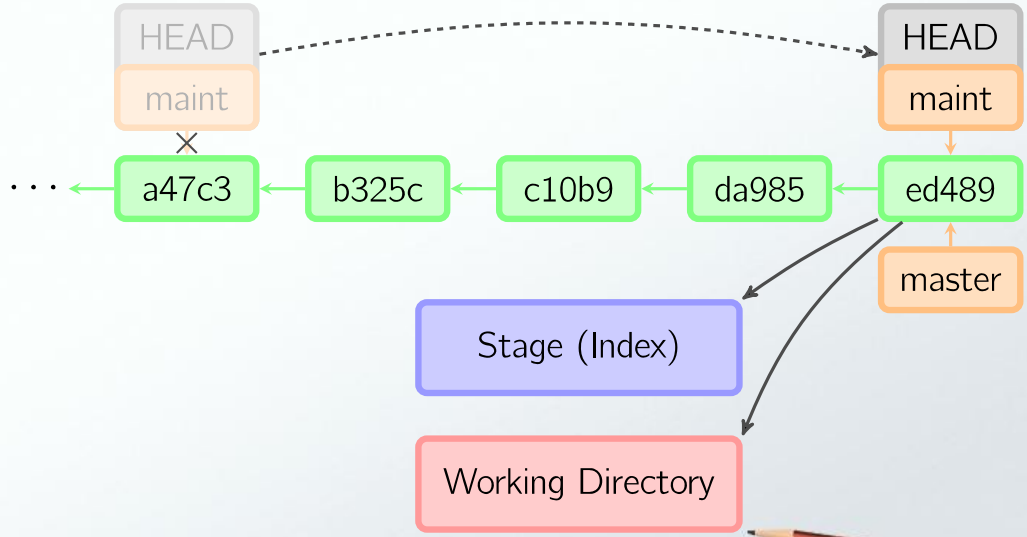


Al hacer un merge, se hace HACIA el branch donde estamos

FAST-FORWARD

Se da cuando el commit actual es un ancestro directo del otro commit

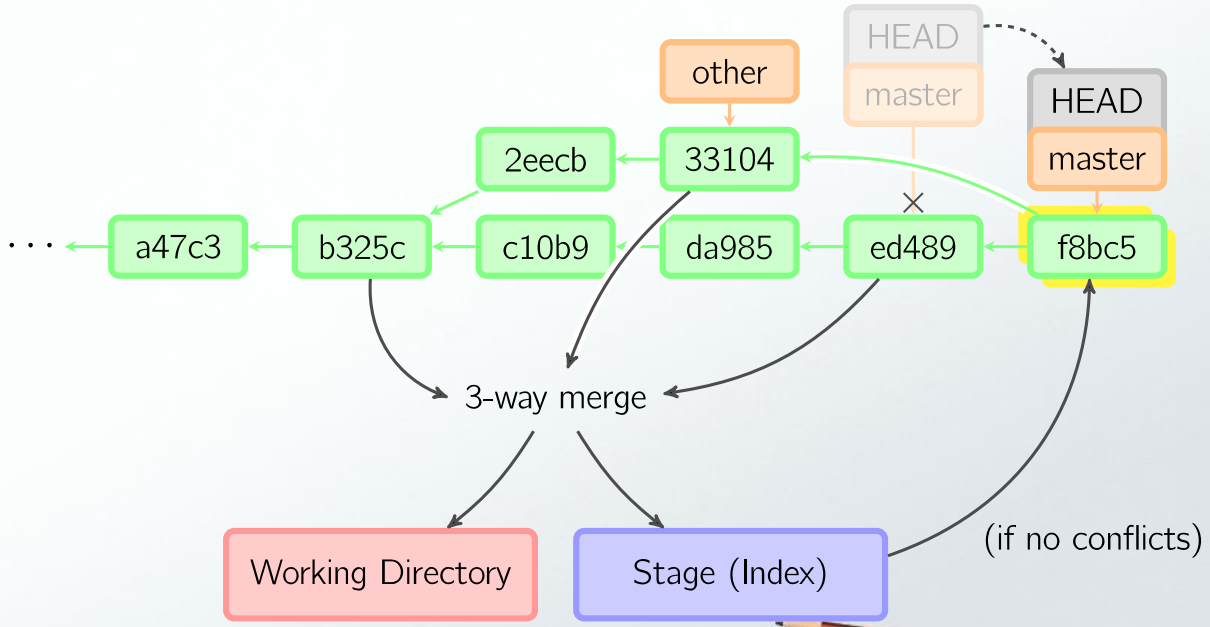
```
git merge master
```



THREE-WAY MERGE

Se toman el commit actual, el otro y ancestro común entre ambos

```
git merge other
```



La práctica hace al maestro

1. Sobre un repo existente, crear un branch llamado `hotfix` y commitear algún cambio ahi
2. Mirar el log del repo y luego hacer un merge de `master` y `hotfix` ¿Qué tipo de merge es?
3. Ahora, comittear un cambio en archivos distintos en cada branch, para crear una bifurcación. Repetir 2
4. Por último, comittear un cambio en archivos iguales en cada branch, para crear una bifurcación. Modificar la misma sección del archivo. Repetir 2

Trabajando colaborativamente

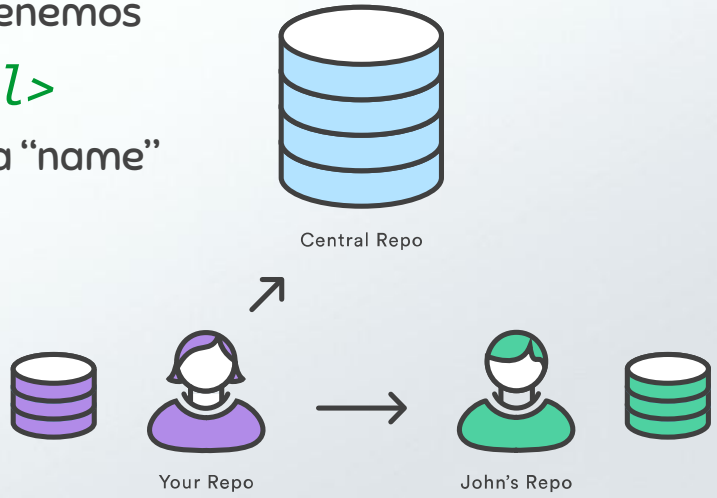
`git remote -v`

Lista las conexiones a remotos que tenemos

`git remote add <name> <url>`

Agrega un remoto a la lista y lo llama "name"

¿Que es origin?



¿Que es origin?

- Es el nombre que git le da al repositorio remoto después de hacer un `git clone`.
- origin queda como remoto automáticamente después de un `git clone`

No hay nada especial en el nombre origin

```
git clone -o server
```

Tipos de branchs ← Muy importante

- Local branches
  - Non-tracking branches
  - Tracking branches
- Remote-tracking branches

- Remote branches

Local branches

- Non-tracking branches

  No están asociados a otro branch

  ```
  git branch <branch>
  ```

- Tracking branches

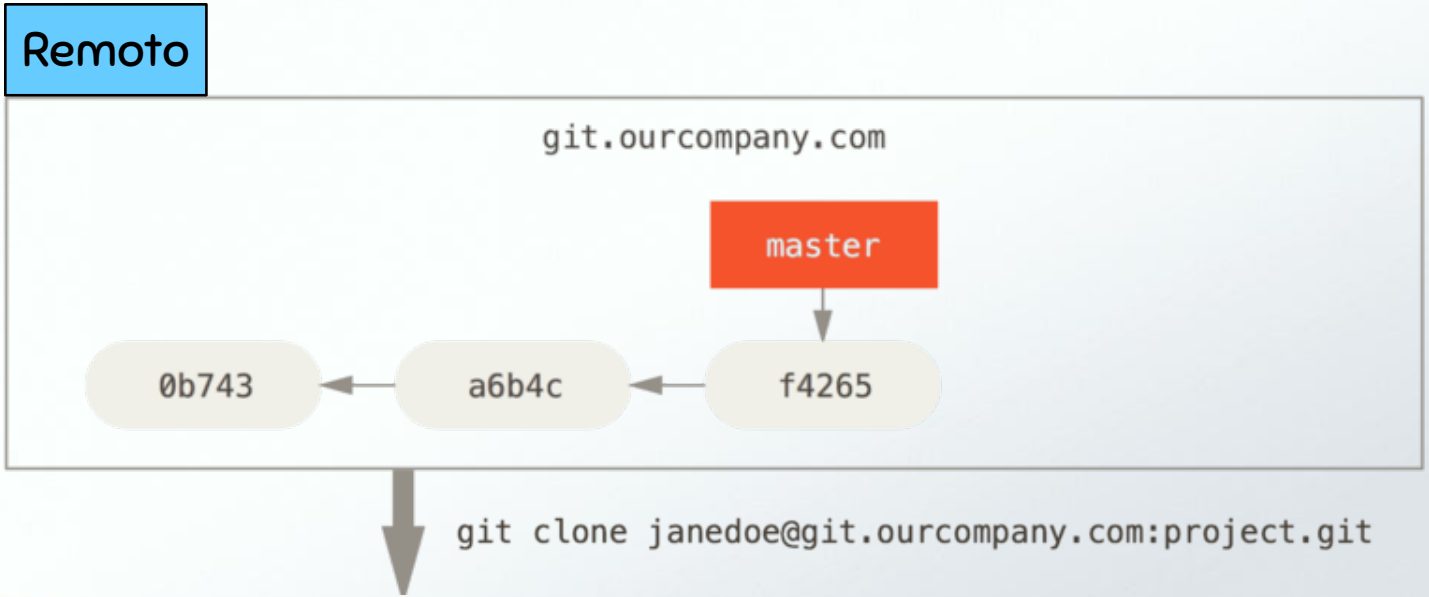
  Están asociados con otro branch,
  usualmente con un remote-tracking-branch

  ```
  git branch --track <branch> [<start-point>]
  ```

Para ver local branches
(tracking y non-tracking)

`git branch -vv`

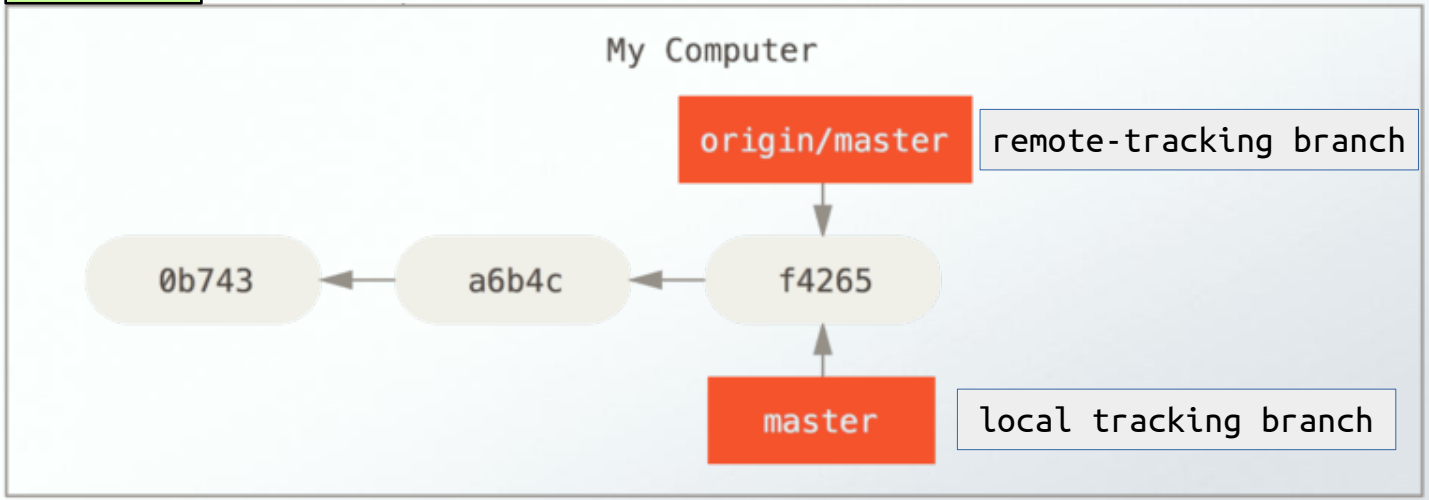

Remote-tracking branches



Remote-tracking branches

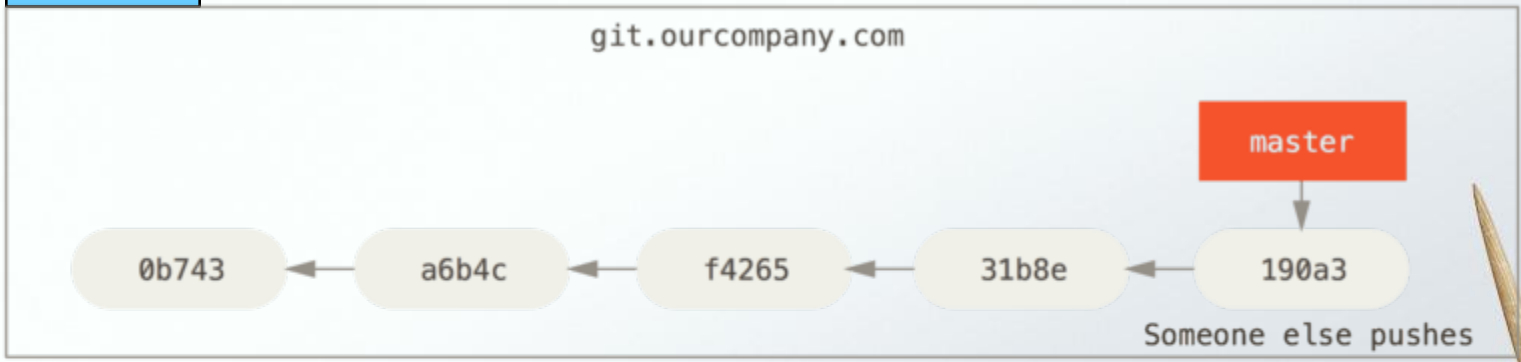
Local

origin y master no tienen nada de especial. Son nombres por default



Hago algo de trabajo y lo commiteo a mi repo local. Mientras tanto, un colaborador pushea algo al server...

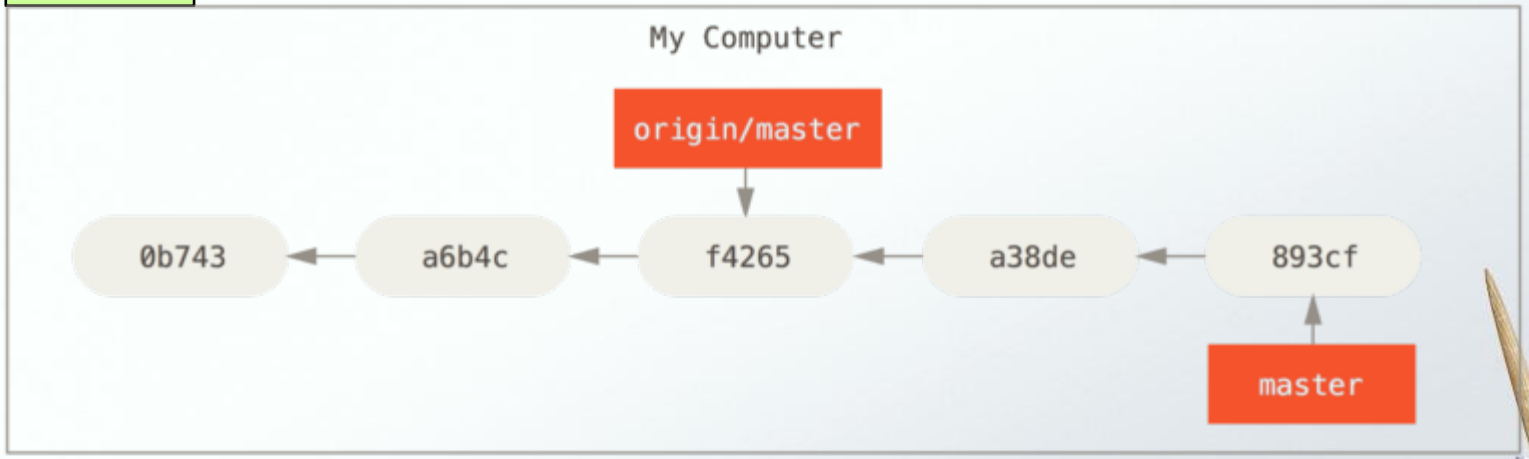
Remoto



¿Cómo queda nuestro repo local?

Sólo se mueve nuestro branch local: master

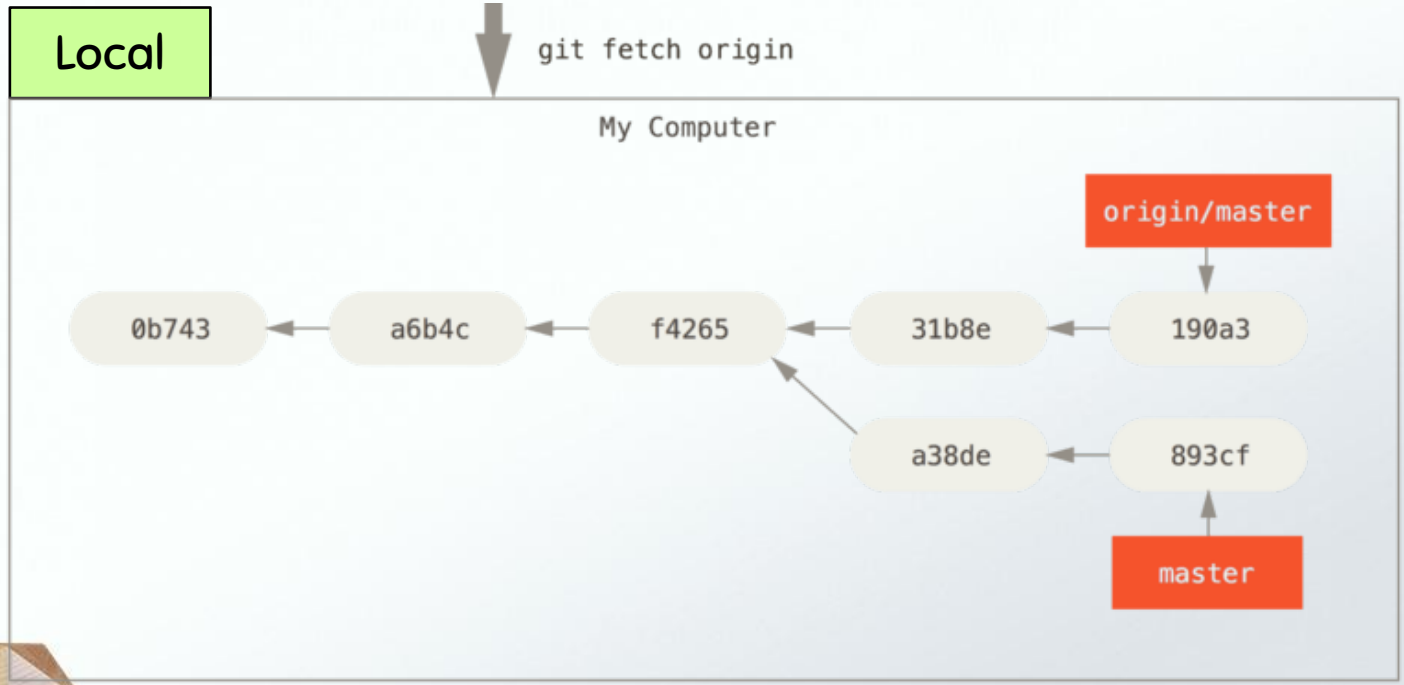
Local



Ahora hacemos:

```
git fetch origin
```

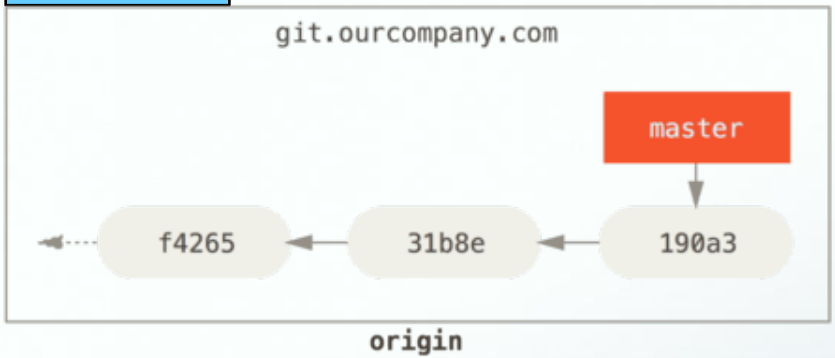
El remote-tracking branch se actualiza:



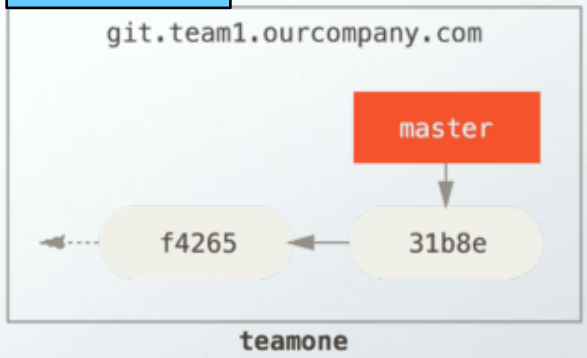
¡Se agrega otro remoto a la escena!

```
git remote add teamone git://git.team1.ourcompany.com
```

Remoto



Remoto

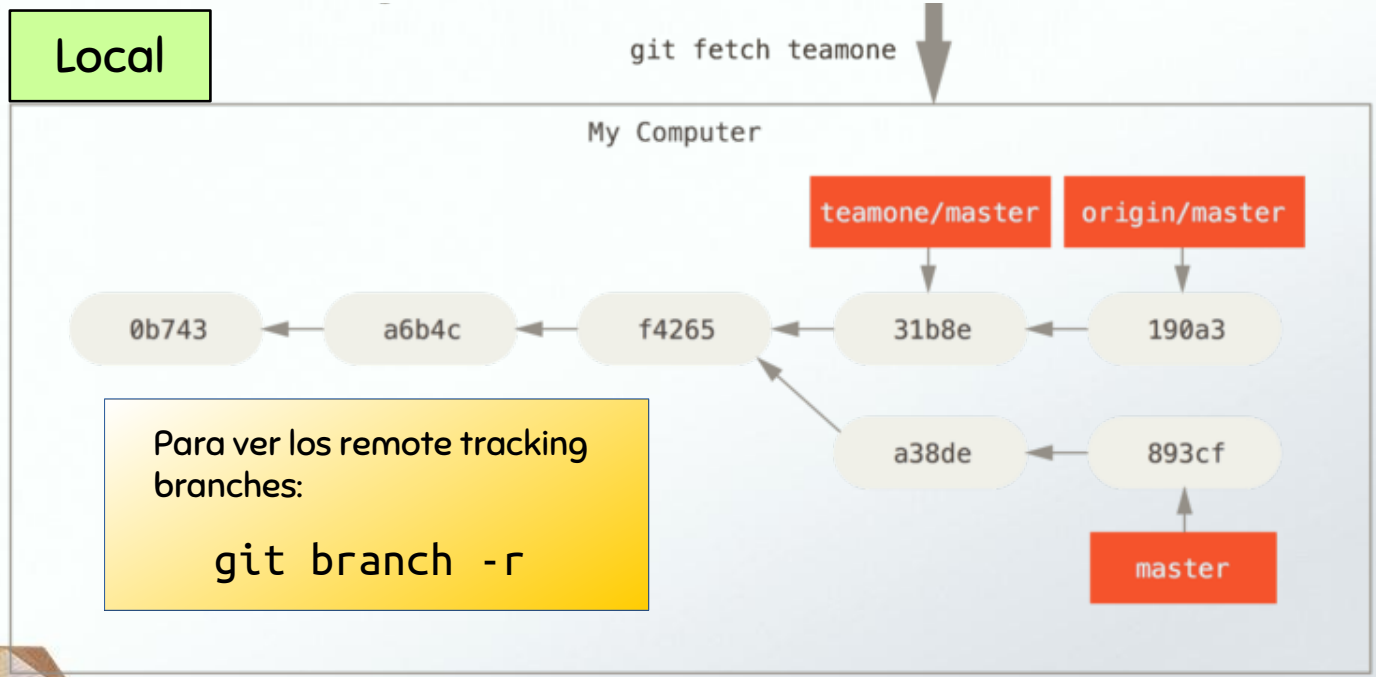


Ahora hacemos:

```
git fetch teamone
```

Se agrega un remote-tracking branch a nuestro repo

Local



Para ver los remote tracking branches:

```
git branch -r
```

Remote branches

Para listarlos:

```
git remote show <remote>
```

Este comando lista además un resumen de los remote-tracking branches

```
git branch origin/develop
```

Remote-tracking branch

```
git push --delete origin develop
```

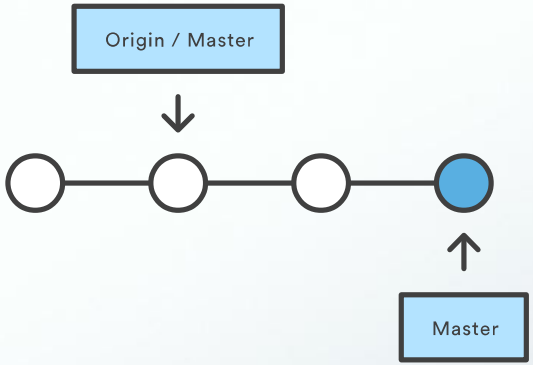
Remote branch

Publicando nuestros cambios: git push

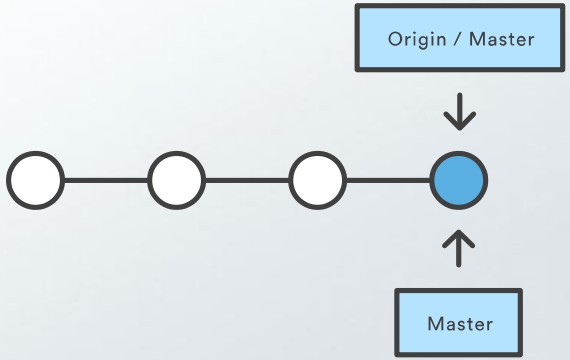
`git push <remote> <branch>`

Transfiere commits del repo local al repo remoto

Antes de push



Después de push



`git push <remote> <branch>`

Transfiere el branch especificado a remoto. Sólo fast-forward merge

`git push <remote> --force`

Tranfiere el branch al remoto y fuerza un merge, aunque no resulte en una operación fast-forward. Usar con precaución.

`git push <remote> --all`

Tranfiere todos los branchs locales al remoto

`git push <remote> --tags`

Transfiere los tags que de otra forma no son transferidos

Si la historia del remoto ha divergido con respecto a la nuestra, git no nos dejará hacer push. Necesitamos integrar los cambios a nuestro repo local antes de pushear

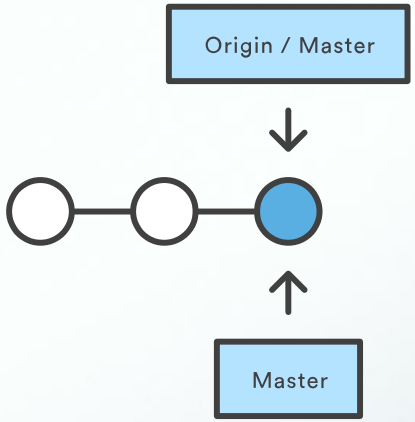
Siempre publicar

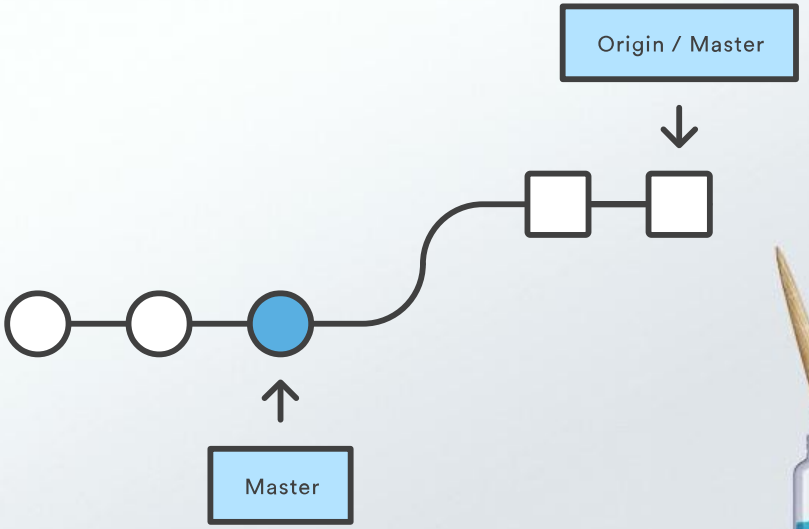
1. git commit
2. git push
3. leave building

THE FORCE IS STRONG WITH THIS ONE

Trayéndonos cambios de otros: git fetch

Transfiere commits del repo remoto al repo local





`git fetch <remote>`

Transfiere todos los branchs del repo remoto

`git fetch <remote> <branch>`

Transfiere el branch especificado del repo remoto

Hacer un fetch no influye sobre nuestro trabajo actual. Es sólo para ver en qué estuvo trabajando el resto

Un atajo: git pull

git pull <remote> =
git fetch <remote>
+
git merge <remote>/<current-branch>

Algunos desarrolladores prefieren hacer un rebase a un merge:

git pull --rebase <remote>


Para investigar

+ git stash
+ git rebase
+ git revert
+ git reset
+ Workflows
+ git commit --ammend
+ git reflog
+ git hooks


Gracias!

Alguna pregunta?

Mi dirección es:

+ cervetto@inti.gob.ar

Más información

- Excelente tutorial:
  https://www.atlassian.com/git/tutorials/
- Una guía visual rápida:
  http://marklodato.github.io/visual-git-guide/index-en.html
- Para probar algunos comandos:
  https://onlywei.github.io/explain-git-with-d3/
- La referencia absoluta:
  https://git-scm.com/doc


Citas

Esta presentación usa material de:

- https://www.atlassian.com/git/tutorials/
- In case of fire 1 git commit 2 git push 3 leave building by Marco Leong is licensed under a Creative Commons
- https://git-scm.com/book/en/v2
- http://rogerdudler.github.io/git-guide/
- https://onlywei.github.io/explain-git-with-d3/