# Capacitación Git
# Módulo 4

Branches

por Gonzalo Barro Gil, fuente https://git-scm.com/book/es/v2

https://github.com/GonzaloBarroGil/git-course

# ¿Qué es un "branch"?

- Desde el punto de vista de Git una "rama" o **branch** es un archivo conteniendo el id de un **commit**.

  ./.git/refs/heads/master

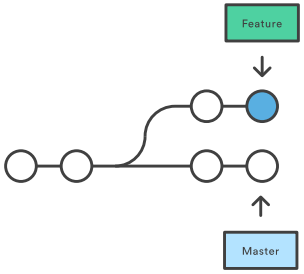
- En esto difiere completamente de otros objetos (commit, tree, blob, tag) que se identifican también con un id. En este caso la integridad viene dada por el FS, en el que no pueden haber dos archivos con el mismo nombre.

- De la misma manera, el **HEAD**, al que definimos previamente como un "apuntador", es un archivo conteniendo la ruta de un archivo de **branch**.

  ./.git/HEAD

- Estas características son las que destacan en Git su soporte para múltiples ramas y la velocidad de ramificación. Esto nos permite conceptualizar mejor un **branch**. Desde este punto de vista, un **branch** puede representar:
  - Una línea o flujo de trabajo.


  - Una versión paralela



# Cambiar de versión

- La interfaz que nos ofrece Git para cambiar de versión es el comando **checkout**. Se aplica sobre tres entidades distintas:

  - **archivo**: Deshace los cambios no preparados del mismo.

    ```
    git checkout index.html
    ```

    - El directorio de trabajo estará alineado con el directorio Git.

  - **branch**: Cambia de rama:

    ```
    git checkout development
    ```

    - ahora .git/HEAD contendrá '.git/refs/heads/development'.

  - **commit**: Obtener versión desde commit:

    ```
    git checkout 0645d25
    ```

    - ahora .git/HEAD contendrá '0645d25 ...' (40 caracteres). El HEAD estará "desprendido" o detached.

- En este momento nos interesa particularmente el aplicado sobre **branch**.

# Manipular branch

- El el comando **branch** nos permite crear, listar o eliminar branches:
  - `git branch`
    Lista los branches existentes.
  - `git branch -d customerApi`
    Elimina un **branch** local llamado customerApi.
  - `git branch homeStyles`
    Crea un **branch** llamado homeStyles
- Otra forma de crear un **branch** es mediante **checkout**:
  - `git checkout -b newBranch`
    Crea un nuevo **branch** y cambia al mismo mediante el modificador **-b**.
    Es la versión combinada de:
    ```
    git branch newBranch
    git checkout newBranch
    ```

# Comparar branches

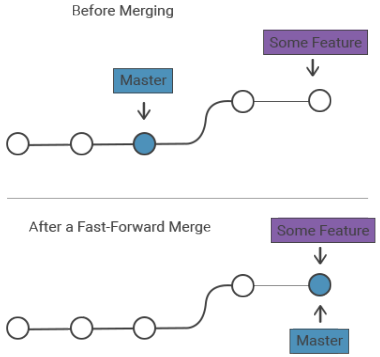
- Si comparamos dos branches por el commit al que apuntan, pueden darse tres situaciones:
  - Ambos apuntan al mismo commit:
    - Es decir, comparten la misma historia de commits; son **equivalentes**.
  - Apuntan a distintos commits:
    - Sólo uno tiene commits adicionales; están **desactualizados**.
    - Ambos tienen commits adicionales; son **divergentes**.
- Los branches pueden ser equivalentes en dos situaciones:
  - Al crear un branch. Un nuevo branch apuntará al commit del branch al que apuntaba el HEAD al momento de crearlo.
  - Al fusionar (merge) dos branches.
- Dos branches se pueden fusionar cuando no son equivalentes:

```
git merge branchAdelantadoODivergente
```

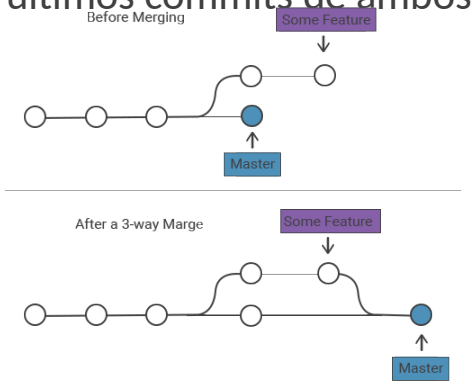
- Git aplicará distintos métodos de fusión entre el branch actual y branchAdelantadoODivergente, dependiendo de las diferencias entre ambos.

# Merging

- **Fast-Forward** merge: se aplica cuando los branches sólo están **desactualizados**.
  - El resultado es que el branch sobre el que se aplica el merge apuntará al mismo commit que el branch que estaba adelantado.




- **3-way** merge: se aplica cuando los  branches son **divergentes**.
  - El resultado implica que en el branch sobre el que aplica el merge se creará un nuevo commit que incluirá los cambios de los últimos commits de ambos branches respecto a su antecesor común.



# Conflictos

- Los **conflictos** se dan si al intentar fusionar dos versiones se dan dos situaciones:
  - Los branches **divergen**.
  - Los branches implican cambios en el mismo sector de archivos.

- Al hacer git merge obtendremos una salida como esta:

```
...
Auto-merging <archivo>
CONFLICT (content): Merge conflict in <archivo>
Automatic merge failed; fix conflicts and then commit the result.
```

- Si consultamos el estado:

```
On branch <branch>
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)

    both modified:   <archivo>
```

- Esto representa dos cosas:
  - El merge no se puede completar porque hay divergencia entre branches y existen conflictos en archivo.
  - Git expone una versión modificada del archivo en un estado intermedio que combina cambios de ambos branches.

# Resolver conflictos

- Archivos binarios: No se pueden combinar los dos cambios; el usuario debe optar por uno:
  - `git checkout --ours archivo`
    En el archivo con conflicto se dejan sólo los cambios del branch **en** donde efectúo el merge (HEAD).
  - `git checkout --theirs archivo`
    En el archivo con conflicto se dejan sólo los cambios del branch **desde** donde efectúo el merge.
- Archivos de texto plano: Al ser de tipo secuencial, Git puede marcar como líneas de texto los sectores del archivo que corresponden a cada versión. Al editar el archivo veremos tres líneas que delimitan dos sectores:
  - archivo con conflicto:

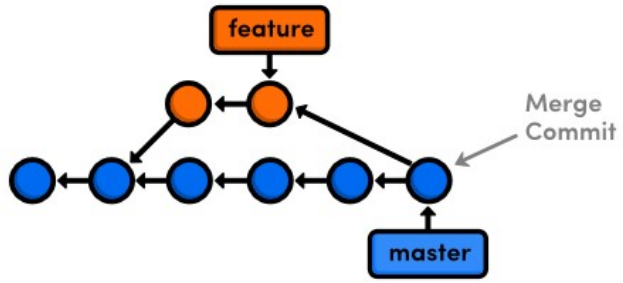
```
...
<<<<<<< HEAD
# Líneas modificadas en el branch actual
=======
# Líneas modificadas en el branch que voy a fusionar en el actual
>>>>>>> otroBranch
...
```

  - Esto nos permite no sólo optar por una u otra versión, en la misma forma que con los archivos binarios, sino también mantener ambas, o modificar cualquiera de las dos. La única condición es que al final deben quitarse las líneas con las marcas delimitadoras.
- Para ambos tipos de archivos, independientemente de la resolución que se haya tomado, se debe salir del estado intermedio del merge:
  - `git add`
  - `git commit`

# Rebasing

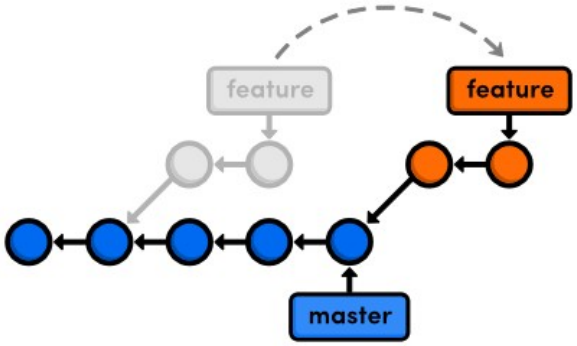
- Este método de integración sólo se aplica a branches **divergentes** y presenta diferencias respecto al merging:
  - **Merging**: toma los cambios del commit del HEAD y los del último commit del branch divergente y combina los cambios de ambos commits en uno nuevo que tendrá a los dos como padres.

git merge master




  - **Rebasing**: parte desde el commit del HEAD hacia atras hasta el primer hijo del ancestro común, lo transforma en hijo del último commit del branch divergente, y vuelve hacia adelante modificando uno por uno los commits con el mismo hasta llegar de nuevo al HEAD.

git rebase master



# Resolver conflictos

- En el caso de un rebase los conflictos se resuelven en la forma correpondiente al tipo de archivo (binario o texto plano) al igual que el caso del merge.

- A diferencia del merge, una vez resueltos los conflictos, se sale del estado intermedio con:
  - `git add`
  - `git rebase --continue`

- Tomando en cuenta que los cambios se aplican a partir del ancestro común, recorriendo los commits siguientes, puede que se presenten conflictos en más de uno con lo cual se deba repetir el proceso varias veces.

- Se puede omitir alguno de los commits si resultan problemáticos con:
  - `git rebase --skip`

- Al llegar al último commit con conflictos resueltos, Git nos informará que el rebase fue aplicado.

# Branches remotos

- Cuando clonamos un repositorio nos crea una copia del remoto con los siguientes elementos:
  - El directorio .git conteniendo todos los objetos Git del remoto.
  - El branch principal del remoto (al que apunta el HEAD).
  - La referencia al repositorio remoto en la forma de ruta o url:
    - `git remote -v`

- Si consultamos los branches locales sólo veremos el branch al que apunta el HEAD, por ejemplo master:

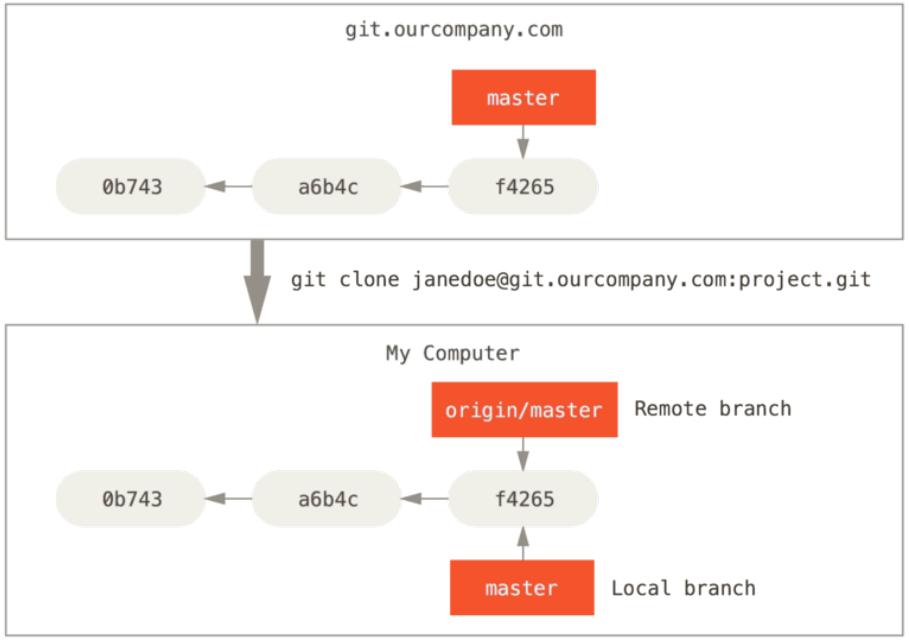
```
git branch
* master
```

- Sin embargo, si las consultamos con el modificador -a (all) veremos las referencias remotas al resto de los branches:

```
git branch -a
* master
remotes/origin/HEAD -> origin/master
remotes/origin/master
remotes/origin/staging
remotes/origin/development
```

# Branches remotos

- Esos branches remotos no son más que referencias apuntando a la copia local de los commits.




- Se puede hacer checkout a alguno de los branches remotos, dejando el HEAD desprendido, para crear el branch local a partir del mismo.

```
git checkout origin/staging
git branch staging
git checkout staging
```

- O bien:

```
git checkout -b staging origin/staging
```

# Branches remotos

- Ni este nuevo branch ni su referencia remota no aplicará cambios posteriores a menos que se le indique.
  - Para actualizar la rama local con los cambios aplicados en el remoto se utiliza:

```
git fetch origin staging
git merge origin/staging
```

    - o bien:

```
git pull origin staging
```

  - Para subir branches locales junto con sus commits y objetos asociados se utiliza:

```
git push origin staging
```

- Todas estas operaciones se hacen a través de la url definida para la referencia definida por defecto como **origin**.
  - Se pueden definir referencias nuevas con:

```
git remote add nombreRemoto urlORuta
```

  - Se pueden modificar referencias existentes con:

```
git remote set-url nombreRemoto nuevaUrlORuta
```