

Despliegue de aplicaciones web

UD 02. Control de versiones



Actualizado Septiembre 2024

Licencia



Reconocimiento – NoComercial - Compartir Igual (BY-NC-SA): No se permite un uso comercial de la obra original ni de las posibles obras derivadas, la distribución de las cuales se debe hacer con una licencia igual a la que regula la obra original.

Nomenclatura

A lo largo de este tema se utilizarán distintos símbolos para distinguir elementos importantes dentro del contenido. Estos símbolos son:

 Importante

 Atención

 Interesante

ÍNDICE DE CONTENIDO

1. Introducción	4
2. Control de versiones	4
2.1 Instalación de Git	5
2.2 Configuración	5
2.3 Ciclo de vida	6
2.4 Comandos básicos de Git	7
2.5 Ignorar archivos	9
3. Ramas en GIT	11
3.1 Comandos específicos de ramas	14
3.2 Conflictos	15
3.3 Eliminar commits	16
4. Remotes en GIT	17
4.1 Merge	20
4.2 GitHub	21
4.2.1 Pull Request	21
5. Bibliografía	22
6. Autores (en orden alfabético)	22

1. INTRODUCCIÓN

En esta unidad, vamos a centrarnos en el manejo del control de versiones, concretamente Git, para después hablar de una de las plataformas de control de versiones online más utilizadas que es GitHub.

2. CONTROL DE VERSIONES

En resumen, Git es un sistema de control de versiones, o lo que es lo mismo, una herramienta con la que podremos tener nuestro proyecto catalogado por cambios y organizado para posibles modificaciones anteriores o posteriores.

Git fue diseñado por Linus Tolvard, ya que como sabrás para desarrollar un sistema operativo como linux se dispone de miles de archivos de código fuente y una amplia comunidad de desarrolladores, lo que en gran medida dificulta llevar un “control” sobre los cambios en lo archivos y la coordinación entre varias personas modificando código todas a la vez.

Git es un Software cuyo propósito principal es llevar un registro de los cambios realizados en archivos del ordenador y coordinar el trabajo que varias personas realizan sobre archivos compartidos.

Utilizando este software podremos llevar un registro de los cambios de los archivos de un proyecto en el que estemos trabajando, independientemente el lenguaje y también nos permitirá trabajar junto a otras personas que quieran sumarse a nuestro proyecto. O nos permitirá sumarnos a trabajar en los proyectos de otros programadores de una manera muy sencilla.

¿Qué es un repositorio?

Un repositorio de Git es un almacenamiento virtual de tu proyecto. Te permite guardar versiones del código a las que puedes acceder cuando lo necesites. Todos los participantes descargan una copia local del mismo en su propio dispositivo. Cada una de estas copias constituye una copia completa de todo el contenido del repositorio. Además, estos archivos sirven como copia de seguridad en caso de que el repositorio principal falle o resulte dañado. Los cambios en los archivos pueden intercambiarse con todos los demás participantes del proyecto en cualquier momento.

Antes de continuar, te recomiendo que visualices este vídeo en el cual se da una introducción más visual de GIT: [¿Qué es Git y cómo funciona?](#)

2.1 Instalación de Git

<https://git-scm.com/download>

- **Linux:** Para realizar la instalación usaremos el instalador de paquetes con las siguientes instrucciones:
 - o `sudo apt update`
 - o `sudo apt install git`
 - o `git --version`
- **Windows:** deberemos descargar el instalador y, una vez instalado, ejecutar git bash.

2.2 Configuración

<https://git-scm.com/book/es/v2/Inicio---Sobre-el-Control-de-Versiones-Configurando-Git-por-primera-vez>

```
# Opciones obligatorias (nombre y correo)
git config --global user.name "Nombre y apellido"
git config --global user.email CORREO@ELECTRONICO

# Editor de preferencia: elegir solo una opción

# Opción 1: Editor de preferencia. Notepad++ en Windows
git config --global core.editor "'C:/Program Files/Notepad++/notepad++.exe'
-multiInst -notabbar -nosession -noPlugin"

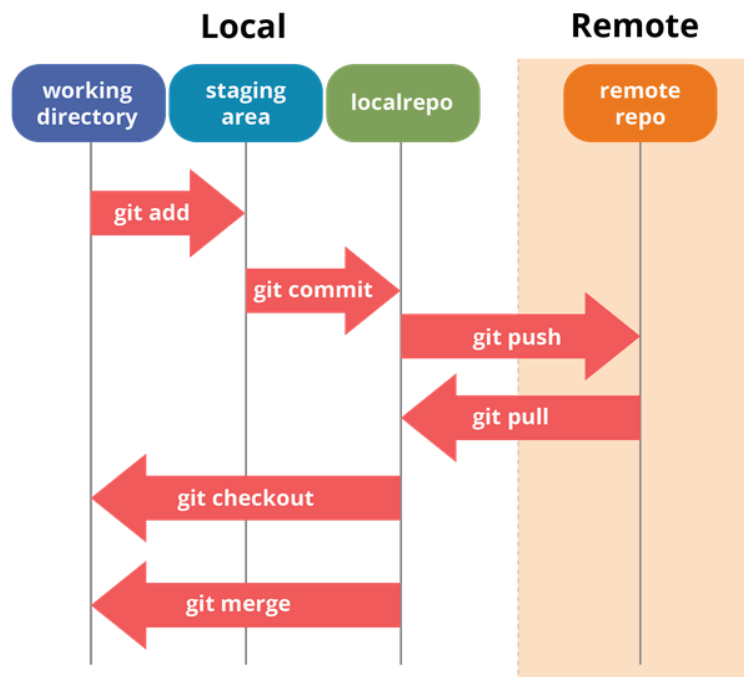
# Opción 2: Editor de preferencia. Visual Studio Code
git config --global core.editor "code --wait"

# Referencia:
https://stackoverflow.com/questions/30024353/how-to-use-visual-studio-code-as-default-editor-for-git
```

IMPORTANTE: Si no se indica un editor de preferencia git utilizará el editor vim cuando tenga que solicitar la intervención del usuario (al hacer un merge, o si el usuario ejecuta git commit sin indicar el mensaje). Este editor es complicado de utilizar para alguien no iniciado, por lo que es muy recomendable cambiar el editor por defecto.

NOTA: en las prácticas de GIT os pediré que hagáis capturas y expliquéis cómo habéis instalado GIT y habéis realizado la configuración inicial, por lo tanto, si vas a realizar ya estos pasos antes de ponerte con la práctica, te recomiendo que saques ya las capturas para quitarte trabajo.

2.3 Ciclo de vida



Para empezar a utilizar git principalmente tenemos que definir las 3 áreas en las que nos vamos a mover.

1. **Working directory:** Será el directorio en el que trabajaremos, de manera local.
2. **Staging area:** Área de preparación. Será donde se almacenen los datos antes de ser enviados al repositorio.
3. **Local repository:** Al realizar cada *commit* quedarán guardados los datos que hayamos añadido en *Staging Area*.
4. **Remote repository:** Repositorio al cual podremos acceder de manera remota, indispensable para trabajar en grupo.
 - Este es opcional, es decir, podríamos trabajar sin necesidad de tener una copia del repositorio en remoto, pero perdemos la opción de trabajar en equipo y también el backup que nos ofrece (ten en cuenta que si tienes un repositorio remoto, si el tuyo en local se rompe o incluso tu ordenador se estropea, puedes recuperar el proyecto, por lo tanto, aunque no se trabaje en equipo es totalmente recomendable trabajar con GIT).
 - Para trabajar con un repositorio remoto necesitamos un lugar en la nube para poder alojarlo, esto lo podemos hacer con GitHub, Bitbucket o GitLab (nosotros utilizaremos GitHub).

2.4 Comandos básicos de Git

- **git init**: iniciar un nuevo repositorio.
- **git config**: establecer parámetros de configuración del repositorio (nombre, correo...)
 - o `git config --global user.name "<mi nombre>"`
 - o `git config --global user.email <mi correo>`
 - o `git config --list` mostrará todas las propiedades que se han configurado.
- **git status**: comprobar el estado de los ficheros del proyecto.
 - o **verde**: añadidos al *Staging area*
 - o **rojo**: no añadidos, presentan cambios. Si son carpetas, solo se mostrará el nombre de la carpeta. Si se desea ver el contenido de nuevas carpetas se deberá ejecutar `git status -u`
- **git add**: añadir elementos del *Working directory* a *Staging area*
 - o `git add .` : para añadir todos los archivos
 - o `git add <nombre archivo>` para archivos individuales
- **git log**: listar historial de versiones de la rama actual. Para salir hay que pulsar la tecla q.
 - o `git log --oneline`: muestra el log en una línea con lo necesario para trabajar
 - o `git log`
 - o `git log --graph`
- **git diff**: mostrar los cambios de los archivos en commits
 - o `git diff <nombre archivo>`: Diff para un archivo
 - o Si queremos visualizar cambios de los archivos en el área de preparación usaremos: `git diff --staged` o `git diff --staged <nombrearchivo>`

```
diff --git a/src/main.c b/src/main.c
index 3ffea1a..f7ac387 100644
--- a/src/main.c
+++ b/src/main.c
@@ -3,9 +3,9 @@

int main ( int argc, char* argv[] )
{
-     if (argc != 1)
+     if (argc > 2)
     {
-         printf("Hola Mundo!\n");
+         printf("Chau Luna?\n");
     } else {
```

- git commit: crea una nueva versión en el repositorio. Cada commit tiene un identificador único denominado hash. Los commits están relacionados entre sí mediante una red de tipo grafo.
 - `git commit -m "mensaje"`
- git show <commit>: mostrar los cambios que se han realizado en los commits. Este comando nos permite mostrar los cambios que se introdujeron en un determinado commit. En primer lugar se puede ejecutar `git log` para buscar el hash del commit que nos interese y a continuación ejecutar `git show` indicando después el hash del commit correspondiente. Los hash de los commits tienen 40 caracteres, pero no es necesario copiarlos enteros: basta con indicar entre los 8 y 10 primeros caracteres para identificar un commit correctamente.
- git tag NOMBRE_TAG: Este comando crea un tag en el commit en que nos encontremos en este momento. Un tag es un **alias** que se utiliza para **hacer referencia a un commit** sin necesidad de saber su hash. Normalmente se utiliza para **indicar números o nombres de versiones** asociadas a un determinado commit. De esta manera podemos **identificar una versión de una manera más amable**. El nombre de los tag se puede utilizar con los comandos de git: por ejemplo, `git show`.
 - git tag <nombre tag> <commit>: etiqueta el commit especificado.
- git checkout: mover el **HEAD** tanto para *commits* como *branches* (ramas).
 - `git checkout -- <archivo>` : Este comando es peligroso, ya que **elimina todos los cambios del archivo** que no hayan sido guardados en el repositorio. Es decir, si el archivo tiene cambios y está en color **rojo**, se perderán dichos cambios. Este comando puede ser útil para dejar un archivo tal como estaba en la última versión guardada del repositorio.
 - `git checkout HEAD~4`: este comando mueve el head 4 posiciones atrás
- git push: subir el estado del proyecto desde *Local repository* a *Remote repository*.
- git pull: bajar el último estado del proyecto guardado en *Remote repository* a *Local repository* para la rama en la que estamos situados.
- git clone <dirección repositorio>: obtener todo el proyecto desde *Remote directory*
- git stash: En ocasiones se hacen cambios que se desea preservar para más adelante: por ejemplo, trabajamos en una modificación de un fichero y de repente nos avisan de que hay un bug en otro fichero que tiene que ser resuelto inmediatamente. Para no trabajar en ambas tareas a la vez podemos ejecutar `git stash`: los cambios que tenemos en ese momento y que no están en el área de preparación (es decir, los cambios que están en color rojo) se guardan en un área temporal; al ejecutar `git status` veremos que no hay ninguna modificación, el directorio de trabajo está limpio. A continuación trabajamos en el bug, hacemos cambios y al terminar ejecutamos `git add` y `git commit` para resolverlo. Una vez resuelto, ejecutamos `git stash pop` y recuperamos los cambios que estábamos realizando antes de ser interrumpidos: veremos que `git status` nos muestra en color rojo los archivos que habíamos modificado al principio.
 - # Guardado temporal de cambios no añadidos al área de preparación
 - `git stash`
 - # Restaurar cambios guardados mediante git stash
 - `git stash pop`

2.5 Ignorar archivos

- Archivo .gitignore
- Plantillas de archivos [.gitignore](#).

Las rutas y nombres de archivo que aparezcan en el fichero .gitignore serán ignoradas por git **siempre que no hayan sido añadidas previamente al área de preparación o al repositorio**. Por ejemplo, si añadimos un archivo al área de preparación mediante git add y a continuación lo añadimos al fichero .gitignore, git lo seguirá manteniendo en el área de preparación, por lo que será incluido en el repositorio si ejecutamos un git commit.

De igual manera, si previamente hemos guardado un archivo en el repositorio mediante git commit y a continuación lo incluimos en el fichero .gitignore, git no lo borrará: será necesario borrarlo del sistema de ficheros (a través de la consola o el navegador de archivos) y añadir los cambios (git add y git commit) para que se borre del repositorio. Una vez borrado, si lo volvemos a crear veremos que git sí que lo ignora si está incluido en el fichero .gitignore.

O también podemos utilizar los siguientes comandos:

- Para directorios: `git rm -r --cached <directorio>`
- Para ficheros: `git rm --cached <fichero>`

Ejemplo de .gitignore:

```
$ cat .gitignore
*.oa
*~
```

La primera línea le indica a Git que ignore cualquier archivo que termine en “.o” o “.a” - archivos de objeto o librerías que pueden ser producto de compilar tu código. La segunda línea le indica a Git que ignore todos los archivos que terminen con una tilde (~), la cual es usada por varios editores de texto como Emacs para marcar archivos temporales.

También puedes incluir cosas como trazas, temporales, o pid directamente; documentación generada automáticamente; etc.

Crear un archivo .gitignore antes de comenzar a trabajar es generalmente una buena idea, pues así evitas confirmar accidentalmente archivos que en realidad no quieres incluir en tu repositorio Git.

- Cuando creas un nuevo repositorio en GitHub, se te añade automáticamente un .gitignore el cual posteriormente podrás modificar.
- Por otro lado, lo más normal es que creéis proyectos haciendo uso de frameworks, éstos por lo general tienen un asistente para crear un nuevo proyecto, lo que incluye la creación de un .gitignore, el cual se adapta a las necesidades del framework.
- En resumen, lo más normal es que ya partáis de un archivo .gitignore el cual simplemente tendréis que modificarlo a vuestras necesidades.

Las reglas sobre los patrones que puedes incluir en el archivo `.gitignore` son las siguientes:

Los patrones glob son una especie de expresión regular simplificada usada por los terminales. Un asterisco (*) corresponde a cero o más caracteres; [abc] corresponde a cualquier carácter dentro de los corchetes (en este caso a, b o c); el signo de interrogación (?) corresponde a un carácter cualquiera; y los corchetes sobre caracteres separados por un guión ([0-9]) corresponde a cualquier carácter entre ellos (en este caso del 0 al 9). También puedes usar dos asteriscos para indicar directorios anidados; `a/**/z` coincide con `a/z`, `a/b/z`, `a/b/c/z`, etc.

Aquí puedes ver otro ejemplo de un archivo `.gitignore`:

```
# ignora los archivos terminados en .a
*.a

# pero no lib.a, aun cuando había ignorado los archivos terminados en .a en
# la línea anterior
!lib.a

# ignora únicamente el archivo TODO de la raíz, no subdir/TODO
/TODO

# ignora todos los archivos del directorio build/
build/

# ignora doc/notes.txt, pero no este: doc/server/arch.txt
doc/*.txt

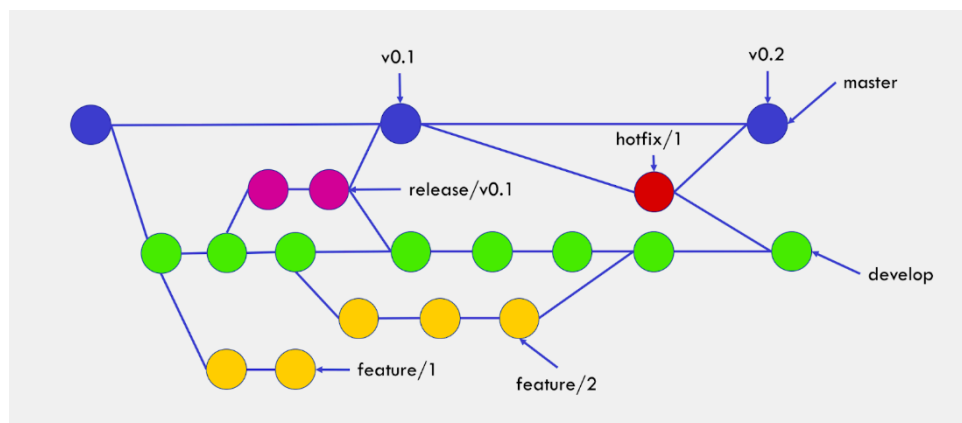
# ignora todos los archivos .txt del directorio doc/
doc/**/*.txt
```

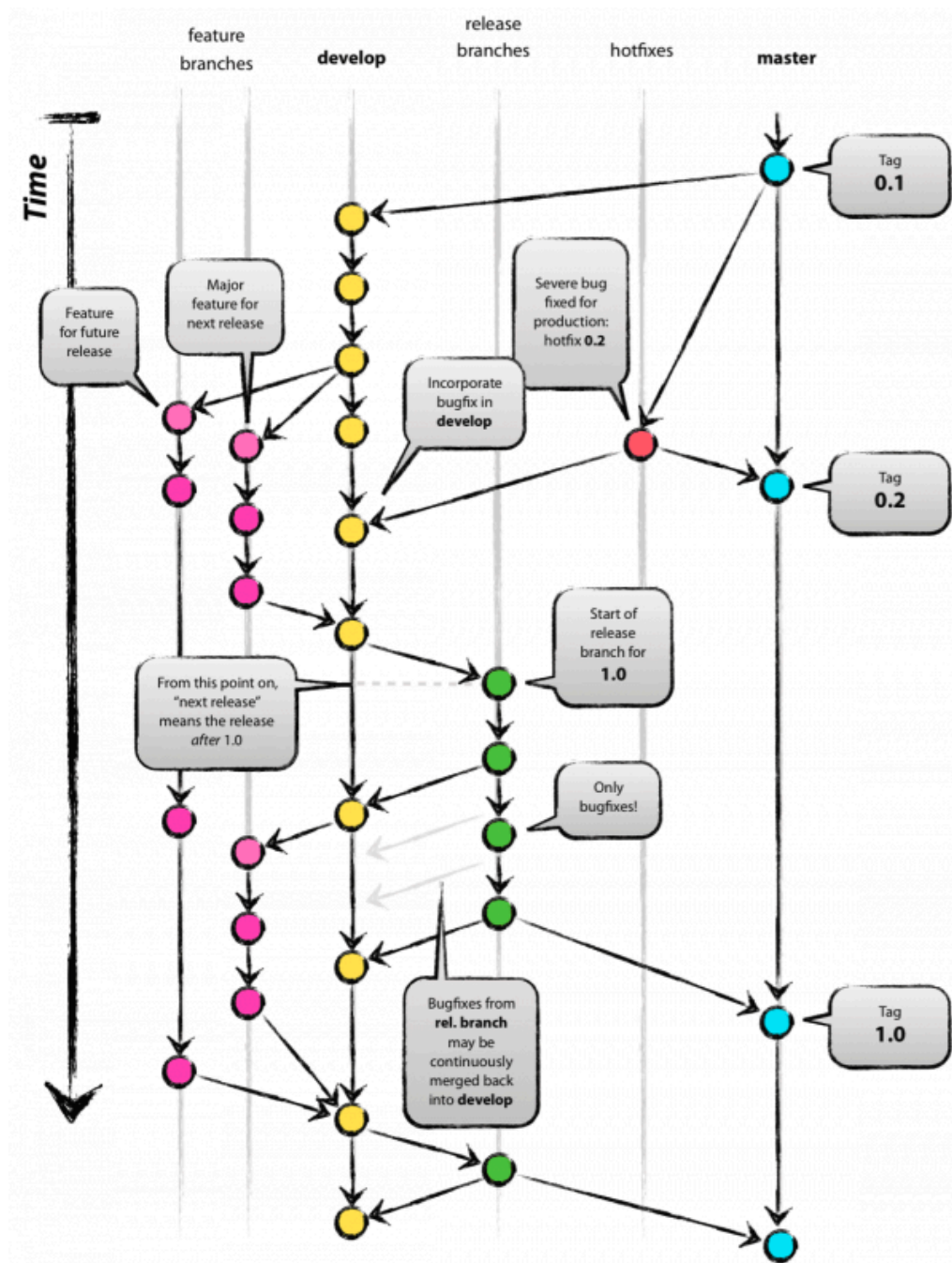
3. RAMAS EN GIT

Las ramas son bifurcaciones de un proyecto, esto permite tener varias copias del proyecto, cada cual sigue un camino distinto hasta unirse de nuevo.

- Una rama es un puntero que apunta a un determinado commit.
- Un repositorio debe tener una rama como mínimo.
- El nombre de la rama que se crea por defecto es **master**.
 - Este nombre no es especial ni tiene una función o significado especial. En GitHub la rama que se crea por defecto si se inicializa un repositorio a través de su interfaz web (lo veremos en la sesión siguiente) se llama main.
- Existe un puntero especial llamado **HEAD** que apunta a la rama en la que estamos en ese momento.
- Al cambiar de rama se modifica el contenido del directorio de trabajo: éste se muestra tal como estaba en la rama a la que hemos saltado.
- La creación y el cambio de ramas se realizan de forma instantánea: no tienen apenas coste.
- El trabajo con ramas es muy interesante por los siguientes motivos:
 - Se pueden hacer pruebas sin modificar el código en producción.
 - Se puede separar el trabajo en tareas o subproyectos que no afecten unos a otros.
 - Cada miembro del equipo puede trabajar sin ser interferido por los demás.

Al crear el repositorio con git nuestro proyecto se encontrará en la rama **master**. Esta rama será la principal de nuestro proyecto. En proyectos individuales que no requieran de diferentes líneas de trabajo podría ser suficiente, pero ¿Qué hacer cuando se trabaja de manera cooperativa? La solución, crear más **ramas**.





Hacer uso de técnicas de trabajo en el que se utilizan varias ramas según las tareas a realizar se le denomina GitFlow, no hay por así decirlo un GitFlow estándar, os vais a encontrar empresas que apliquen GitFlows distintos.

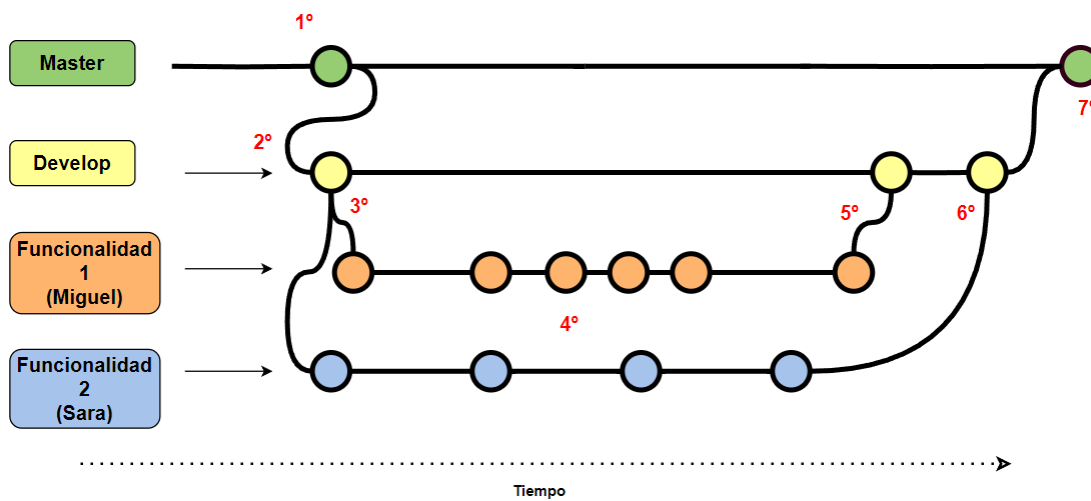
Ejemplo de Gitflow:

Partir de la rama **master**, será la rama principal, la cuál contendrá una versión final y estable del proyecto para salir a producción. La rama master no puede tener código que se encuentre en desarrollo.

Crear la rama **develop**, la cuál se utilizará por los programadores para crear nuevas funcionalidades, a partir de develop cada programador creará una nueva rama para cada funcionalidad, una vez la funcionalidad esté terminada, se unificarán los cambios a develop. De esta forma, develop es el punto de encuentro de las funcionalidades de

distintos miembros del equipo. Hasta que una funcionalidad no se termine, no se podrá unificar a develop, por lo tanto, en develop no se desarrolla directamente.

Vamos a verlo de forma gráfica:



1º Se crea el repositorio y por lo tanto, tenemos la primera rama master (puede llamarse main, es indiferente, pero siempre tiene que haber una principal).

2º Creamos la rama de desarrollo (develop) a partir de master.

3º Los trabajadores Miguel y Sara van a empezar a desarrollar sus funcionalidades, por lo tanto, NO trabajan directamente en develop, si no que se crean sus propias ramas.

4º Miguel y Sara van haciendo commits conforme van desarrollando sus funcionalidades, en este punto, si Miguel rompe el proyecto, con eliminar su rama sería suficiente, y volvería a lo que había anteriormente, de esta forma protegemos el proyecto.

5º Miguel termina su funcionalidad y la unifica con develop.

6º Sara termina su funcionalidad y la unifica con develop, en este punto podrían haber conflictos si Miguel y Sara han modificado las mismas líneas de un mismo archivo, por lo tanto, deberían de comunicarse para solucionar el conflicto ya que develop no puede dejar de estar estable.

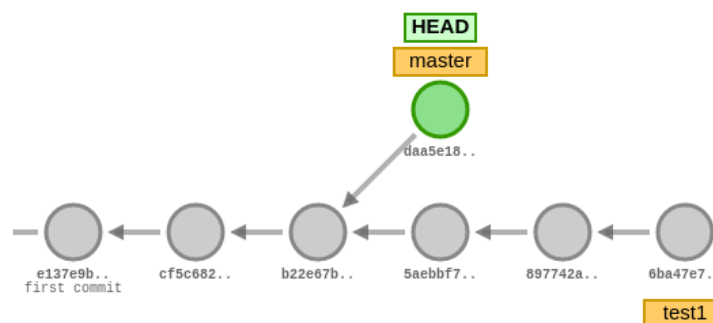
7º Se comprueba que develop está estable con ambas funcionalidades y se unifica con master, de esta forma ya tenemos las funcionalidades 1 y 2 preparadas para llevarlas a producción (*En este paso veremos más adelante que es más especial y se hace de una forma concreta, con Pull Requests*).

3.1 Comandos específicos de ramas

- git branch: Listar todas las ramas locales que tenemos en nuestro proyecto. Si queremos, además, mostrar las remotas, se puede ejecutar `git branch --all`
- git branch <nombre rama>: crear una nueva rama.
 - `git checkout -b <nombre rama>`: crear y cambiarnos a esta nueva rama.
- git checkout <nombre rama>: Este comando cambiará a la rama indicada. Es posible que el comando falle **si hay cambios en el directorio de trabajo** que no estén integrados en ningún commit: si dichos cambios pueden ser sobreescritos por los cambios de la rama a la que se desea cambiar, git abortará el cambio de rama y nos indicará el problema. En ese caso deberemos crear un commit con los cambios que estemos realizando (o bien guardarlos mediante `git stash`) y a continuación volver a ejecutar el comando de cambio de rama.
- git merge <nombre rama>: fusionar 2 o más ramas. Además, funciona como *commit*. Fusionar una rama, en inglés merge, consiste en incorporar los cambios presentes en una rama a la rama en la que nos encontramos actualmente. Para realizar una fusión hay que realizar las siguientes acciones:
 - Primero nos posicionamos en la rama sobre la que se va a realizar la fusión (la rama que va a recibir los cambios).
 - Para realizar la fusión ejecutar:
`git merge <nombre_rama_a_fusionar>`

Si por ejemplo queremos integrar en la rama desarrollo (develop) los cambios presentes en la rama funcionalidad1, nos cambiaremos a la rama develop (si no estamos ya en ella) mediante `git checkout develop` y a continuación, ejecutamos `git merge funcionalidad1`.

- git branch -d <nombre rama>: Este comando eliminará la rama local indicada. **IMPORTANTE:** la eliminación de una rama supone la eliminación del puntero que hace referencia a un determinado commit. Si al eliminar una rama se quedan commits sin referenciar, dichos commits se perderán: pueden recuperarse durante un tiempo solo si se conocen sus *hash*; git realiza también tareas de limpieza al realizar algunas acciones y procede a limpiar los commits “huérfanos”, por lo que pasado un tiempo ni siquiera se podrán recuperar a través de sus *hash*. En el ejemplo siguiente, si se elimina la rama test1 se perderán los commits 5aebbf7, 897742a y 6ba47e7, ya que no habrá ninguna rama que haga referencia a ellos.



3.2 Conflictos

Al fusionar una rama pueden producirse **conflictos**. Un conflicto se produce cuando **diferentes commits introducen cambios en las mismas líneas de los mismos archivos**.

Al producirse un conflicto, git no sabe qué cambios deben prevalecer: los de la rama A, los de la rama B, los dos, ninguno, algo totalmente distinto,... En este caso **es necesaria la intervención humana**. Git modificará los ficheros afectados incluyendo **delimitadores** para indicar los cambios que vienen de una rama y los que vienen de HEAD, es decir, de la rama en la que nos encontremos.

Es importante recalcar que **git no perderá información**: la incluirá toda, junto con los delimitadores para identificar la procedencia de los cambios.

Si se produce un conflicto **git quedará en un estado intermedio**: añadirá al área de preparación (color verde) los archivos que no presenten conflictos e **indicará los archivos en conflicto**, en color rojo, para que el usuario los edite y resuelva los conflictos.

Resolver los conflictos pasa por **editar el archivo**, localizar los **delimitadores** y **dejar el archivo como queremos que quede**. Normalmente esta última acción consistirá en decidir **qué cambios son los que queremos dejar y eliminar los delimitadores**. Al final, **el fichero debe quedar tal como queremos que quede**: en ocasiones una de las versiones será la correcta; en otras, la otra versión; en otras, ninguna; en otras, ambas; en otras, algo totalmente distinto.

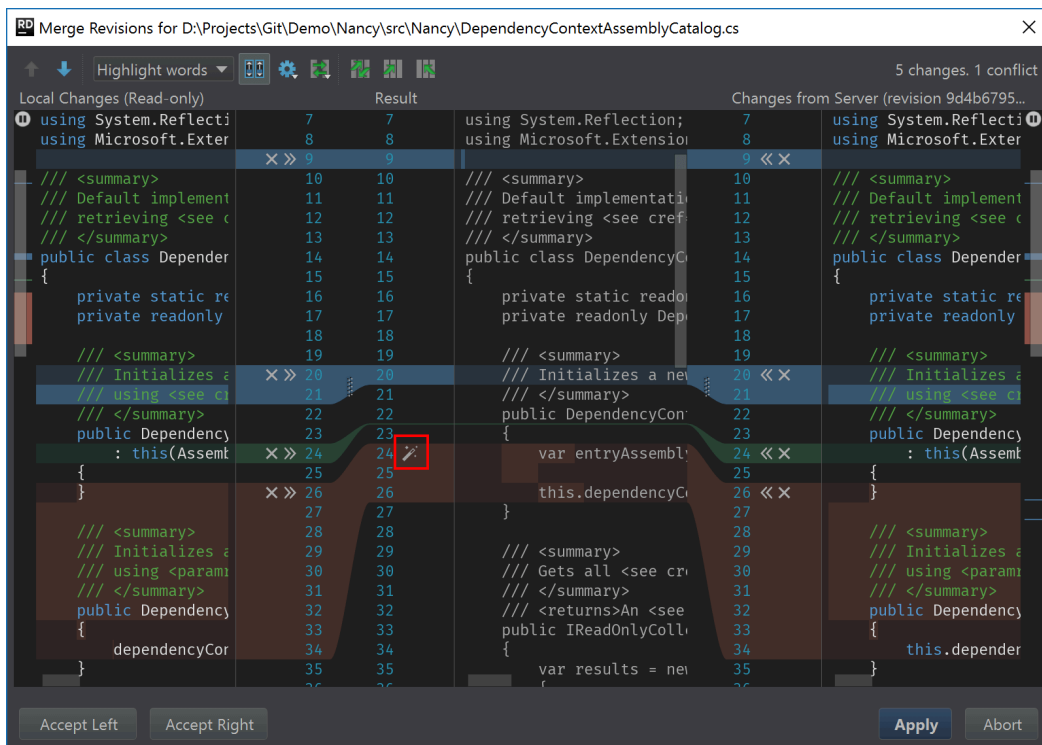
Una vez resuelto el conflicto en todos los archivos habrá que ejecutar los comandos git add y git commit para **crear un commit que resuelva el conflicto**.

En resumen, los pasos a seguir para solucionar el problema son los siguientes:

- git status (lista los archivos conflictivos).
- Se accede a los archivos y se opta por una de las modificaciones (o se hace una nueva).
- git add <archivos> .
- git commit -m "<mensaje>"

Otra ventaja de git es que los comandos utilizados para solucionar estos conflictos son los mismos que los que se utilizan normalmente, por lo que no altera el flujo de trabajo.

Existen herramientas gráficas que permiten solucionar los conflictos de una forma más visual, por ejemplo los IDE de JetBrains incorporan una resolución de conflictos muy recomendable, aquí puedes ver un ejemplo, en el cual, te muestra 3 visiones, la de la izquierda es una rama, la de la derecha es la otra y el centro es cómo quieres que quede el archivo final, en rojo marca las líneas que causan el conflicto:



3.3 Eliminar commits

Hay momentos en los que por equivocarnos en los archivos a incluir, o por un error en el código que no queremos ver reflejados en nuestros commits. Para solucionar estos problemas podremos seguir las siguientes estrategias.

- **git reset:** Eliminación por completo de un commit (sólo aconsejable cuando se tiene claro que no generará ningún conflicto con otras ramas de desarrollo)

```
git reset --hard [commit hasta donde resetear]:
```

Borrará además los cambios de Staging Area y Working Directory

```
git reset --soft [commit hasta donde resetear]:
```

Permanecerán los cambios de Staging Area y Working Directory

PRECAUCIÓN

git reset es uno de los pocos comandos con los que se puede perder información de manera irreversible!

- **git revert:** Creación de un commit con información de un commit posterior. Es el más recomendable para evitar conflictos con otras ramas y desarrolladores.

```
git revert [commit] [-m]: revertiremos los cambios indicando el commit al cual queremos llegar.
```

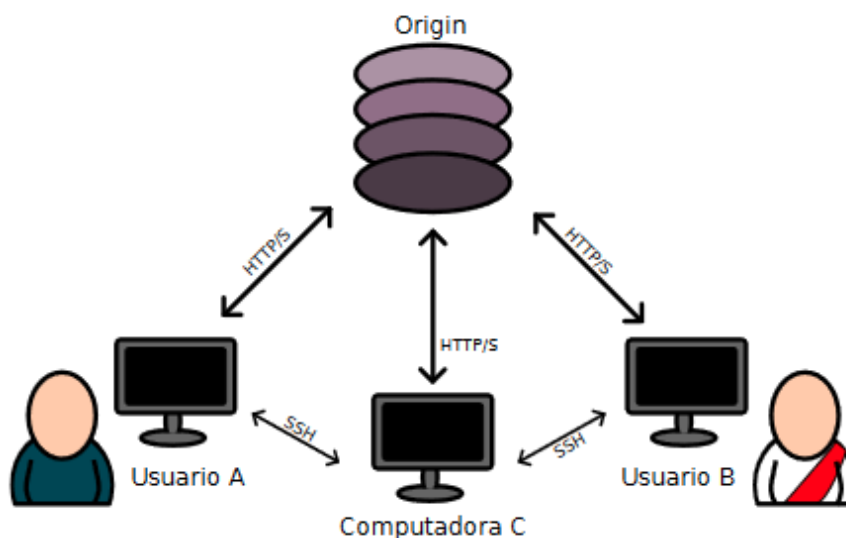
- Ejemplos:

```
git revert HEAD~1. realizaremos el revert a partir del commit anterior a nuestro HEAD.
```

```
git revert commit. realizaremos el revert del commit especificado
```


4. REMOTES EN GIT

Git permite establecer conexiones con diversos repositorios remotos (repositorios hospedados en internet o en una red privada) llamados remotes que sirven tanto como back-up del proyecto así como para compartirlo entre personas o computadoras.



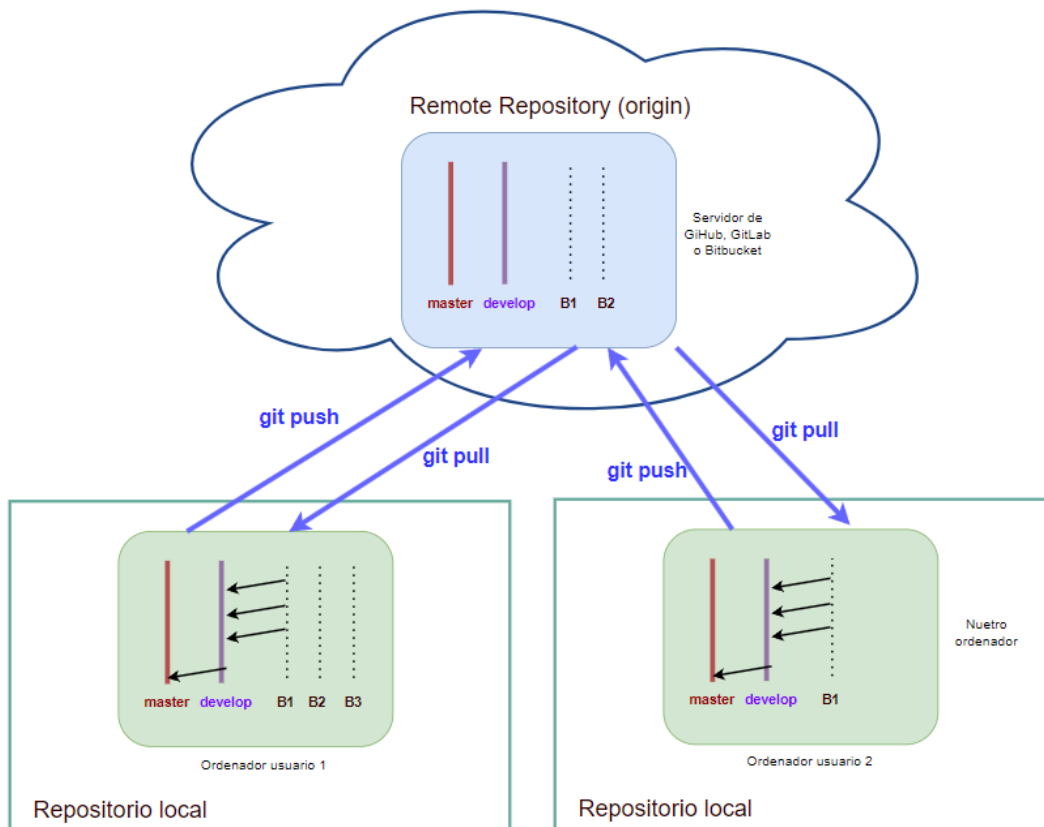
Existen varios servicios de hosting para repositorios online. Estos repositorios permiten interactuar de manera directa con el repositorio, modificando archivos, agregando, eliminando, viendo la historia de commits y los cambios de los mismos, todo a través de una interfaz gráfica más amigable que la consola de comandos.



GitLab



Una misma rama en git puede estar en varios sitios a la vez. El repositorio es el histórico donde están todos los *commits* que se han hecho.



Vamos a explicar cada uno de ellos:

- **Carpeta local o área de trabajo o árbol de trabajo:** Es donde están los ficheros con lo que trabajamos nosotros directamente mientras programamos.
- **Repositorio local:** es el repositorio que tenemos en nuestro ordenador, cada usuario del proyecto tiene su propio repositorio local.
 - En este repositorio podemos ir trabajando localmente haciendo cambios en ramas, commits, etc, pero todos los cambios se quedarían en este ordenador, si queremos subir cambios de cualquier rama, hay que ejecutar un comando que posteriormente veremos, de esa forma, haremos que estos cambios estén disponibles en el repositorio remoto y por lo tanto, accesibles por cualquier usuario que esté trabajando en el mismo proyecto.
 - **origin/<nombre_rama>** es el nombre que reciben las ramas remotas.
- **Repositorio remoto:** es el repositorio que se encontrará en el servidor hosting que estemos utilizando para este fin (GitHub, GitLab, Bitbucket, servidor propio...):
 - En este repositorio no se trabaja directamente, es decir, es un repositorio en el cuál solo debemos subir los cambios que vamos haciendo desde los repositorios locales.
 - Lo habitual es subir cambios de los repositorios locales, y descargar cambios del repositorio remoto a nuestro repositorio local para actualizar posibles cambios que hayan podido hacer otros usuarios del proyecto.

```
git clone <URL_REPOSITORIO>
```

Este comando permite clonar un repositorio remoto a partir de su URL.

- Por defecto, en nuestro repositorio local obtendremos únicamente la **rama principal** del repositorio remoto.
- El resto de ramas no aparecen como ramas locales, pero pueden obtenerse.

```
git checkout <nombre_rama>
```

Este comando ya lo hemos visto, sirve para cambiar de una rama a otra. Pero, en este caso, también lo podemos utilizar para descargar una rama, es decir, podemos cambiarnos a una rama que no tengamos en local, al cambiarnos se descarga.

```
git fetch
o
git fetch origin master
```

Este comando permite actualizar la información de los repositorios remotos:

- **git** se conecta al remoto y comprueba si hay nuevos cambios en las ramas remotas; por ejemplo que se ha creado alguna nueva, si es así, actualiza los punteros de las ramas remotas del repositorio local para reflejar dichos cambios.
- **Pero NO incorpora los cambios a las ramas locales.** Es decir, los cambios de ficheros que se hayan hecho. Ten en cuenta que si el usuario 1 crea una rama desde su repositorio local y lo sube al repositorio remoto, el repositorio local del usuario 2 no se sincronizará automáticamente, deberíamos manualmente ejecutar este comando para poder ver los cambios que se han hecho en el repositorio remoto.
- En resumen: git fetch es la única manera de que el repositorio local tenga conciencia de que ha habido cambios en el repositorio remoto.
- Se puede utilizar la opción --prune para eliminar referencias a ramas que ya han sido eliminadas.
- Este comando puede ser necesario para utilizarlo con el comando anterior, ya que si queremos descargar una rama, debemos tener actualizado el repositorio remoto.

```
git pull o git pull origin <rama>
```

Comando para copiar lo que hay en el repositorio remoto en nuestra rama local.

- En este caso, sí se hace la copia de todos los cambios que sean distintos entre nuestra rama local y la rama remota (ficheros, carpetas...).
- Al hacer un **pull**, realmente se está haciendo un merge entre la rama en la que estamos y la rama que especificamos, por ejemplo, si estamos en develop, al hacer pull mergeamos develop remoto con develop local. Un pull equivale a dos acciones: un fetch y un merge. Mediante fetch, git comprueba los cambios que hay en las ramas remotas y los refleja en los punteros de las ramas remotas en el repositorio local; a continuación, la acción merge incorpora los cambios de la rama remota a la rama local correspondiente.

```
git push
o
git push origin <rama>
```

Subir el estado del proyecto desde *Local repository* a *Remote repository*:

- Estaríamos subiendo el estado de únicamente la rama en la que nos encontremos.

```
git remote -v
originhttps://github.com/user/prueba.git (fetch)
originhttps://github.com/user/prueba.git (push)
```

Este comando permite ver los remotos configurados. Otra alternativa es ejecutar `git remote show <nombre_del_remoto>` para ver los detalles del repositorio remoto.

```
git remote add <nombre> <url>
```

Añadir el repositorio local al repositorio remoto especificado.

```
git remote rm <nombre>
```

Eliminar remoto.

```
git remote rename <nombre> <nuevo nombre>
```

Renombrar remoto.

Algunas consideraciones y recomendaciones para trabajar y aprovechar al máximo las características del control de versiones:

- Usar ramas, no trabajes únicamente con una rama y haz uso de gitflow.
- Ignorar archivos que se autogeneran (ejecutables, archivos objeto, log, pdf, etc).
- Todos los archivos deben ser trackeados o ignorados.
- Mensajes de commits claros.
- Commits cortos y específicos. No implementar (o arreglar) varias cosas en un único commit.
- **Todo código en la rama "Master" debe compilar.**
- **No desarrollar directamente en "Master", ni en "Develop" (si se utiliza).**

4.1 Merge

El merge es unir los commit de una rama en otra.

```
git merge <rama>
```

Para realizar una fusión hay que realizar las siguientes acciones:

- Primero nos posicionamos en la rama sobre la que se va a realizar la fusión (la rama que va a recibir los cambios).

- Para realizar la fusión ejecutar:
`git merge <nombre_rama_a_fusionar>`

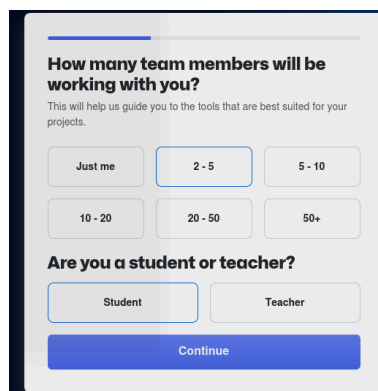
4.2 GitHub

GitHub es una plataforma para el alojamiento de proyectos git. Gracias a esta podremos ver proyectos de la comunidad (siempre que sean públicos) y subir nuestros proyectos creándonos una cuenta. En el resto de la asignatura utilizaremos esta plataforma para crear nuestros propios repositorios.

Es aconsejable usar el mismo correo con el que vamos a trabajar en git para darnos de alta nuestra cuenta de GitHub.

Para realizar este proceso:

- 1) Iremos a la página web de GitHub en <https://github.com/>
- 2) Clicamos en Sign Up e introduciremos nuestras credenciales
- 3) Confirmáis el correo que os mandan para verificar la cuenta ingresando el código.
- 4) Informáis del objetivo de vuestra cuenta:



The screenshot shows a GitHub sign-up form with the following sections:

- How many team members will be working with you?**
This will help us guide you to the tools that are best suited for your projects.
- Buttons for team size: Just me, 2 - 5 (selected), 5 - 10, 10 - 20, 20 - 50, 50+.
- Are you a student or teacher?**
- Buttons for user type: Student, Teacher.
- A blue **Continue** button at the bottom.

Y con esto ya tendréis acceso a vuestro panel de control en el cual administrar vuestros repositorios.

4.2.1 Pull Request

Un Pull Request es una función de GitHub que permite a tu equipo solicitar la revisión y aprobación de sus cambios antes de fusionarlos en la rama principal de desarrollo, denominada “master” o “main”.

Al crear un PR se genera una conversación en la que los demás miembros pueden seguir y comentar los cambios propuestos en el código del repositorio. Esto permite detectar cualquier error o problema potencial antes de integrarlos en la rama principal, lo que ayuda a mantener el proyecto más limpio y estable.

A continuación, realiza el tutorial que te propongo en Aules, en el cual trabajaremos de forma práctica todo lo que hemos visto en este documento.

5. BIBLIOGRAFÍA

- [1] <https://github.com/pedroprieto/curso-github/> - Gestión de Git para la tarea docente - curso de Pedro Prieto, profesor del IES Mare Nostrum de Alicante
- [2] <http://logongas.es/doku.php?id=clase:daw:daw:start>
- [3] <https://pedroprieto.github.io/categories/>
- [4] <https://git-scm.com/book/es/v2/Fundamentos-de-Git-Trabajar-con-Remotos>
- [5] Hoja de referencia de GitHub
https://training.github.com/downloads/es_ES/github-git-cheat-sheet.pdf
- [6] https://corriol.github.io/sxe/UD01/1_arquitectura_de_xarxa_tcpip.html
- [7] <https://www.blai.blog/2018/12/tipos-de-red-en-virtualbox.html>
- [8] <https://sites.google.com/site/wikiredespro/sistemas-operativos-ii/crear-red-entre-maquina-virtual-y-maquina-host-en-virtualbox>
- [9] Desplegament d'aplicacions web. Institut Obert de Catalunya
- [10] Libro de Git <https://git-scm.com/book/es/v2/>
- [11] Git CheatSheets <https://training.github.com/>

6. AUTORES (EN ORDEN ALFABÉTICO)

A continuación ofrecemos en orden alfabético el listado de autores que han hecho aportaciones a este documento.

- Silvia Amorós Hernández
- David Folgado De la Rosa
- Miguel Mira Flor