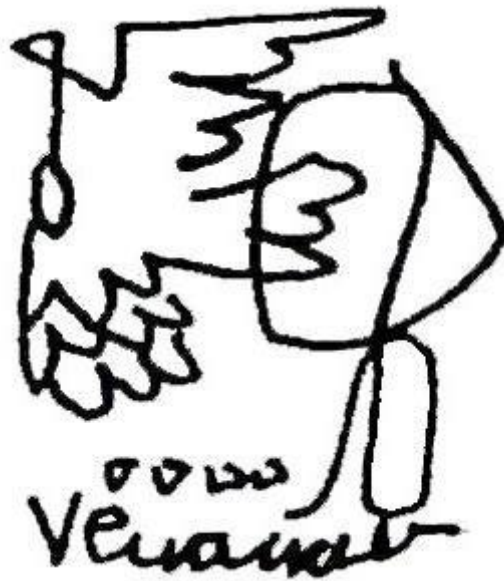


ENTORNOS DE DESARROLLO

TEMA 3:

DOCUMENTACIÓN Y SISTEMAS DE CONTROL DE VERSIONES

Desarrollo de Aplicaciones Multiplataforma



Curso 2021/2022

Profesor:

Marcos Unzueta Puente

Índice

CONTENIDO BOCYL (nº 140 20-julio-2011)	¡Error! Marcador no definido.
Índice	2
Índice de ilustraciones	3
1. Sistemas de control de versiones	4
1.1. Tipos de sistemas de control de versiones	4
1.2. Git.....	5
1.2.1. Instalación de git.....	7
1.3. Comenzando en Git	7
1.4. Estados de archivos Git	9
1.5. Comenzando a operar sobre el repositorio.....	9
1.6. Ignorando ficheros	14
1.7. Trabajando con ramas	14
1.8. Etiquetas	17
1.9. Usar git en NetBeans	19
1.10. Operar con la nube.....	19
1.11. Uso de Markdown	24
1.12. Flujo de trabajo de GitHub	24
1.13. Gestión de ramas remoto-local.....	26
1.14. Colaboradores e issues.....	27
1.15. Proyectos de GitHub	28
1.16. Otras funcionalidades de GitHub	29
1.17. Coordinando con GitHub desde NetBeans	29
1.18. GitKraken.....	29
2. Documentación del código fuente	31
2.1. JavaDoc	31
REFERENCIAS	33

Índice de ilustraciones

No se encuentran elementos de tabla de ilustraciones.

1. Sistemas de control de versiones

El sistema de control de versiones se encarga de registrar y gestionar los cambios realizados en un archivo o conjunto de archivos a lo largo del tiempo.

Un sistema de control de versiones:

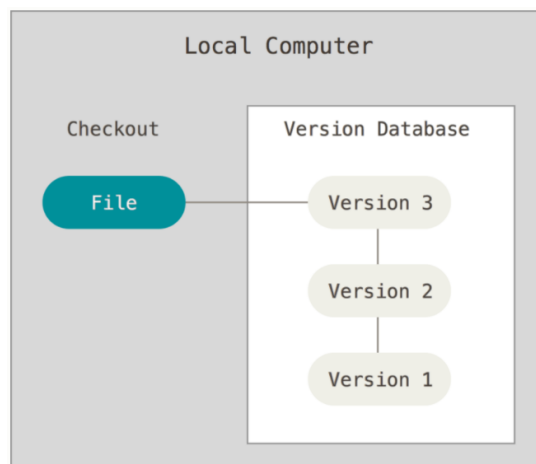
- Genera un historial con las versiones del proyecto (con la fecha de cada versión y el desarrollador que la ha publicado).
- Permite la comunicación directa entre desarrolladores.
- Facilita el trabajo paralelo.
- Da la opción de revertir el estado del proyecto al de una versión anterior.
- Dificulta la pérdida o el extravío de ficheros del proyecto.

Un **repositorio** en git es un espacio de almacenamiento del proyecto en el que se almacenan todas las versiones de los archivos y ficheros.

1.1. Tipos de sistemas de control de versiones

- **Sistemas de control de versiones locales**

Consiste en copiar los archivos a otro directorio que tan solo se guardará de manera local.



Es el método más antiguo, más propenso a fallos y más sencillo de usar.

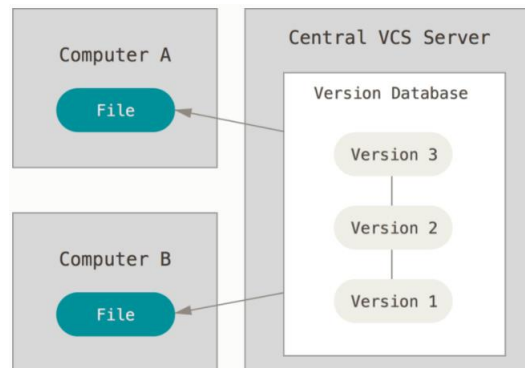
Este método solo puede ser utilizado si el proyecto lo desarrolla un solo programador.

- **Sistemas de control de versiones centralizados**

Si queremos colaborar con otros desarrolladores, será necesario cambiar de paradigma.

Debemos contar con un servidor en el que se guardará el repositorio y al que han de poder acceder todos los desarrolladores del proyecto.

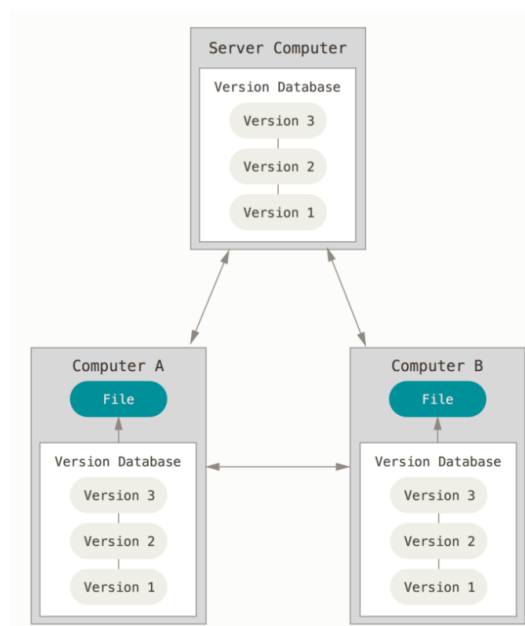
Para operar con sistemas de control de versiones de este tipo, hay que descargar en local el proyecto o la parte con la que se va a trabajar, se realizan los cambios oportunos y se publican en el repositorio remoto. El problema está en que existe un único punto de fallo.



- **Sistemas de control de versiones distribuidos**

Existe un servidor remoto en el que se encuentra el repositorio, pero este se replicará en cada computadora con la que se pretenda trabajar.

De esta manera, si falla el repositorio del servidor, el riesgo de pérdida de datos es mucho menor.

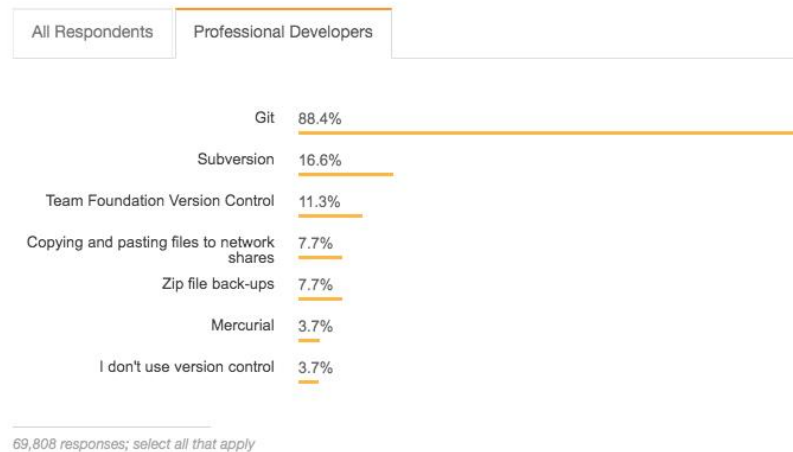


1.2. Git

Git es un sistema de control de versiones

- Es el más utilizado en todo el mundo.

Version Control

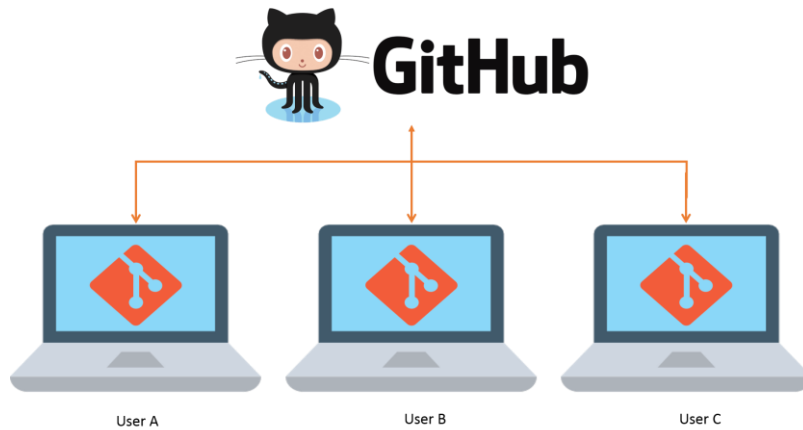


Git is the dominant choice for version control for developers today, with almost 90% of developers checking in their code via Git.

- Es un proyecto de código abierto. (<https://github.com/git/git>)
- Fue desarrollado originalmente por Linus Torvalds.
- Proyectos de desarrollo software muy importantes usan esta tecnología.



- Es un sistema distribuido.



- Es la alternativa más potente.
- Difícil de utilizar.

1.2.1. Instalación de git

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

Para comprobar si la instalación se ha realizado correctamente usamos el comando **git --version**, que nos indica la versión del programa.

```
Símbolo del sistema
Microsoft Windows [Versión 10.0.19042.1348]
(c) Microsoft Corporation. Todos los derechos reservados.

C:\Users\srunz>git --version
git version 2.33.1.windows.1

C:\Users\srunz>
```

1.3. Comenzando en Git

(-- cuando va a ir una palabra concreta, - cuando van a ir abreviaturas)

- **git --version** : Para saber la versión de git actual. <https://git-scm.com/docs/git-version>
- **git help**: Para poder ver la ayuda de git. <https://git-scm.com/docs/git-help>
- **git config**: Permite configurar git. <https://git-scm.com/docs/git-config>
 - **Local**: Ajustes específicos del repositorio.
 - **Global**: Ajustes específicos del usuario.

- **Sistema:** Ajustes de todo el sistema.

Ejercicio 1: Comprueba que Git está bien instalado, configúralo con tu nombre y correo electrónico y, posteriormente, consulta el archivo de configuración.

- **git init:** Crea un repositorio local. <https://git-scm.com/docs/git-init>

```
PS C:\Users\srunz\Desktop\Curso Git\01-bases> git init
Initialized empty Git repository in C:/Users/srunz/Desktop/Curso Git/01-bases/.git/
```

Este comando creará en el proyecto la carpeta **.git** que contendrá toda la información necesaria por el control de versiones.

Ejercicio 2: Descarga el proyecto adjuntado por el profesor y conviértelo en un repositorio.

- **git status:** Muestra el estado del espacio de trabajo. <https://git-scm.com/docs/git-status>

Indica

- La rama en la que nos encontramos.
- Permite ver los cambios que tiene el proyecto respecto a la **versión activa** en el repositorio.
- Permite ver los ficheros sobre los que Git no hace seguimiento.
- Permite ver los cambios que formarán parte del commit y los que no.

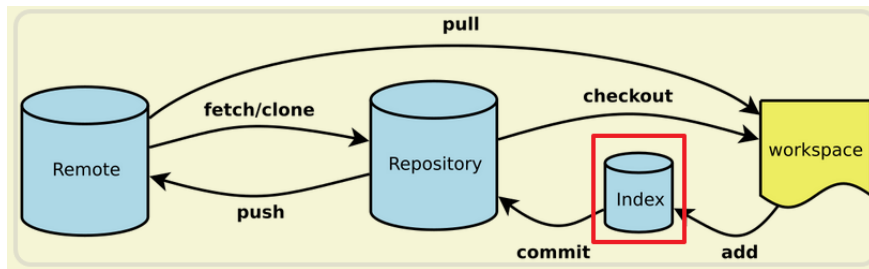
```
PS C:\Users\srunz\Desktop\Curso Git\01-bases> git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        .DS_Store
        css/
        fonts/
        images/
        index.html
        js/
        main.html
        scss/

nothing added to commit but untracked files present (use "git add" to track)
PS C:\Users\srunz\Desktop\Curso Git\01-bases>
```

- **git add:** Añade los ficheros y carpetas indicadas al área de **stage**.
 - **<dirección>:** Añade un archivo específico o directorio.
 - **..:** Añade todos los ficheros y carpetas del repositorio que no se encuentren incluidos en el **.gitignore**.



1.4. Estados de archivos Git

- **Tracked:** Son archivos sobre los que Git tiene control, no tienen modificaciones pendientes y su estado actual está almacenado en el repositorio.
- **Staged:** Son archivos sobre los que Git tiene control. Estos archivos han sido modificados, pero estas modificaciones no han sido todavía almacenadas en el repositorio.
Estos archivos se encuentran en el stage, por tanto, cuando se realice un commit, sus cambios serán guardados en el repositorio.
- **Unstaged:** Son archivos sobre los que Git tiene control. Estos archivos han sido modificados, pero estas modificaciones no han sido todavía almacenadas en el repositorio.
Estos archivos no se encuentran todavía en el stage, por tanto, si inmediatamente se realiza un commit, los cambios de estos ficheros no serán almacenados en el repositorio.
- **Untracked:** Son archivos sobre los que git no tiene ningún tipo de control, por tanto, no se encuentran en el repositorio.

Ateniéndonos a estas definiciones, **git add** permite pasar a los archivos de Untracked o Unstaged a Staged.

- **git reset:** Si añadimos por error un archivo al Stage y queremos sacarlo, podemos usar este comando. Este comando se usará posteriormente para muchas más cosas.

Ejercicio 2: Añade dos archivos al Stage, comprueba el output de git status e interprétalo. Saca los archivos del Stage.

Ejercicio 3: Añade todos los archivos al Stage, comprueba el output de git status e interprétalo. Saca los archivos del Stage.

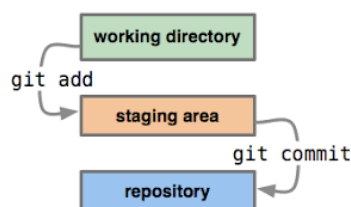
1.5. Comenzando a operar sobre el repositorio

Un **commit** captura el estado del proyecto en un momento determinado. El commit se registrará en el repositorio generando una instantánea.

Es conveniente hacer un commit cada vez que se programa una funcionalidad importante.

- **git commit:** Crea una instantánea del estado del proyecto utilizando las modificaciones que se encuentran en el Stage y añade estas modificaciones al repositorio local.
 - **-m "Mensaje del commit":** Permite introducir directamente el nombre del commit a realizar.
 - **-am "Mensaje del commit":** Añade los ficheros que se encuentran en estado de Unstaged al Stage y permite introducir directamente el nombre del commit a realizar.

Siempre que hacemos un commit es obligatorio ponerle un nombre y este ha de ser una breve descripción de los cambios realizados, de otra manera, el control de versiones perderá eficacia.



- **git log:** Permite consultar el histórico de commits. <https://git-scm.com/docs/git-log>
- **git show hash-del-commit:** Permite consultar información ampliada sobre un commit. Puede usarse también para ramas, etiquetas, etc. <https://git-scm.com/docs/git-show>

Ejercicio 4: Haz un commit de un archivo. Interpreta el status y el log.

Ejercicio 5: ¿Qué elementos contiene el status de cada commit?.

Ejercicio 6: Haz un commit que contenga todos los archivos html. Interpreta el status y el log.

Ejercicio 7: Haz un commit de todos los archivos que quedan en estado Untracked. Interpreta el status y el log.

Ejercicio 8: Si ahora modificamos un fichero ¿En qué estado se encontrará? Compruébalo.

Ejercicio 9: Modifica un fichero y haz un commit. Interpreta el status y el log.

Ejercicio 10: Modifica dos ficheros y haz otro commit. Interpreta el status y el log.

Ejercicio 11: Crea un README.md y haz un commit con él. Interpreta el status y el log.

Ejercicio 12: Crea una carpeta vacía y haz un commit con ella. Interpreta el status y el log.

En función de la versión de git en la que nos encontremos, la rama en la que nos encontraremos trabajando se llamará de una manera u otra.

Las ramas son versiones del repositorio que se suelen utilizar en la creación de funcionalidades o en la creación de errores y genera una bifurcación del estado del proyecto, creando una nueva ruta para el desarrollo del mismo.

Lo recomendable es que los desarrolladores trabajen en ramas y en la principal solo esté la versión que va para producción.

En una primera instancia, la rama principal se llamaba master, no obstante, git ha decidido sustituir este nombre por main, ya que se considera tan entendible como el anterior y menos ofensivo.

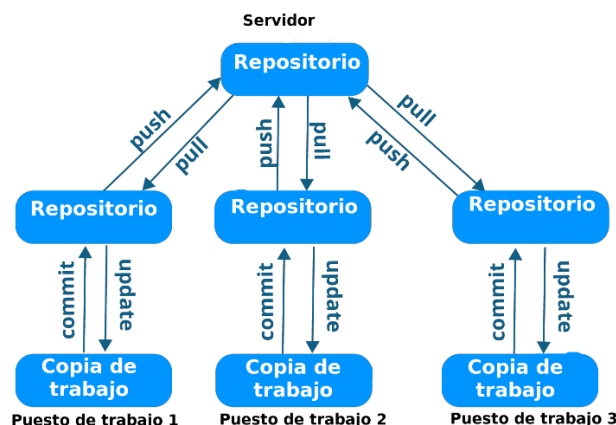
El concepto de rama se explicará más en profundidad posteriormente.

- **git branch:** Indica la rama en la que nos encontramos y muestra las ramas existentes en local. <https://git-scm.com/docs/git-branch> . Esta función tiene muchas más funcionalidades que se explicarán posteriormente.
 - **-m antiguo-nombre nuevo-nombre:** Para cambiar el nombre a una rama existente.
- **git config --global init.defaultBranch nombre-rama:** Para que siempre que se cree un repositorio la rama principal se llame como deseemos.

Ejercicio 13: Consulta el nombre de la rama sobre la que estás trabajando y, si se llama master, cámbiale el nombre a main.

Posteriormente cambia la configuración para que la rama principal, al crear un repositorio, siempre se llame main y consulta el archivo de configuración para comprobar que este cambio se ha realizado correctamente.

Ejercicio 14: Realmente, ¿Hasta qué punto de esta imagen hemos llegado?



Hay algunos comandos de git que usamos muy a menudo, o que necesitamos mejorar constantemente añadiendo parámetros. Para este tipo de situaciones, existen los alias.

Por ejemplo, el comando **status**, en muchas ocasiones, da más información de la que deseamos. Existe una versión de este comando que nos brinda una información más escueta, pero que se puede ajustar mejor a lo que buscamos en un momento determinado: **status --short**.

Por otra parte, el log es muy tosco, nos interesaría que fuera algo más visual. Añadiéndole una serie de parámetros conseguimos un efecto más agradable e intuitivo ([URL del alias](#)) : **log --graph --abbrev-commit --decorate --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**

Ejercicio 15: Crea un alias para cada uno de los comandos indicados (se necesita usar el archivo de configuración). Comprueba que se ha creado el alias correctamente.

- **git diff**: Permite comprobar los cambios que presenta el proyecto en el espacio de trabajo respecto a la versión de commit que se encuentra activa en el repositorio. <https://git-scm.com/docs/git-diff>

Ejercicio 16: Modifica dos ficheros y compararlos con la última versión almacenada en el repositorio.

- **git checkout --** .: Git checkout permite hacer muchas cosas, pero, de momento, vamos a darle un uso muy simple. Con los parámetros -- . permite devolver al proyecto al estado en el que se encuentra la versión activa del commit (siempre que los cambios del proyecto en el espacio de trabajo estén en estado Unstaged).

Ejercicio 17: Usando un comando devuelve el proyecto al estado activo del commit. Analiza el status y el log.

Ejercicio 18: Traduce al español el texto de index.html y haz un commit (deja una frase por traducir).

- **git commit --amend**: Permite modificar el commit más reciente.
 - **-m**: Cambia el mensaje del commit.
 - Si hay archivos modificados en el stage, los añade al último commit.

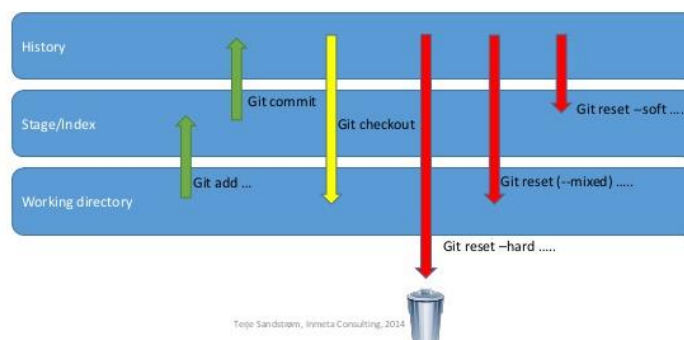
Una vez se ha subido un commit a GitHub NUNCA se debe de modificar de esta manera.

Es más normal hacer varios commits que modificar el contenido de uno (cambiar el nombre es más normal).

Ejercicio 19: Cambia el nombre del commit anterior y haz que la frase que quedaba por traducir esté traducida en ese mismo commit.

- **git reset:** Git reset es una herramienta que nos permite movernos entre versiones del proyecto.
 - -- **hard:** Los cambios realizados se eliminan del espacio de trabajo. (Los cambios que no hayan sido commiteados se pierden para siempre)
 - -- **soft:** Los cambios realizados no se eliminan de nuestro espacio de trabajo y los deja como staged.
 - -- **mixed:** Los cambios realizados no se eliminan de nuestro espacio de trabajo y los deja como unstaged. Es la opción por defecto.

Git tree movements visualized



Ejercicio 20: Vuelve al commit anterior a traducir el index con los tres tipos de reset y explica las cosas diferentes que ha hecho cada uno. Al final vuelve al estado en el que te encontrabas antes de este ejercicio con un comando. Analiza el status y el log.

Ejercicio 21: Traduce el main.html (no hagas commit) y vuelve al commit anterior a traducir el index con los tres tipos de reset y explica las cosas diferentes que ha hecho cada uno. Al final vuelve al estado en el que te encontrabas antes de este ejercicio con un comando. Analiza el status y el log.

- **git reflog:** Contiene un registro de referencias.

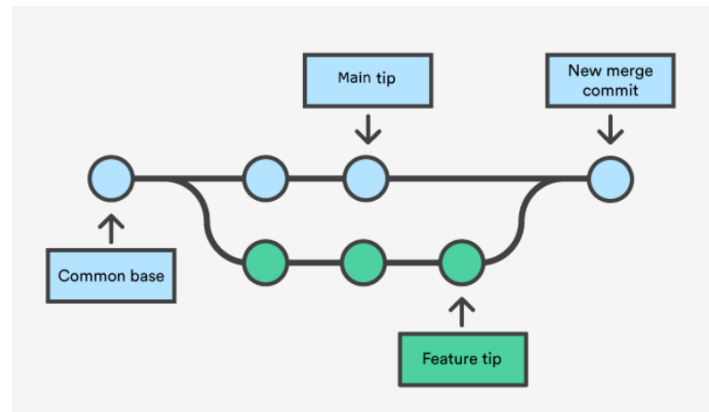
Ejercicio 22: ¿Qué pasa cuándo haces un reset y luego un commit? Ponlo a prueba.

Ejercicio 23:

- 1º Crea un nuevo fichero en el proyecto y añádele un lorem ipsum.
- 2º Haz un commit del mismo.
- 3º Modifica el nombre. Analiza el status. ¿Qué pasa? ¿Por qué? Haz un commit.
- 4º Analiza el status y el log.
- 5º Ahora modifica el nombre y el contenido. Haz un commit.
- 6º Analiza el status y el log. ¿Qué ha pasado? ¿por qué?
- 7º Elimina el fichero. Haz commit.
- 8º Analiza el status y el log.

Cuando unimos una rama a otra decimos que las **fusionamos**. Realmente, lo que estamos haciendo, es volcar sobre una rama todos los cambios que se realizaron en otra.

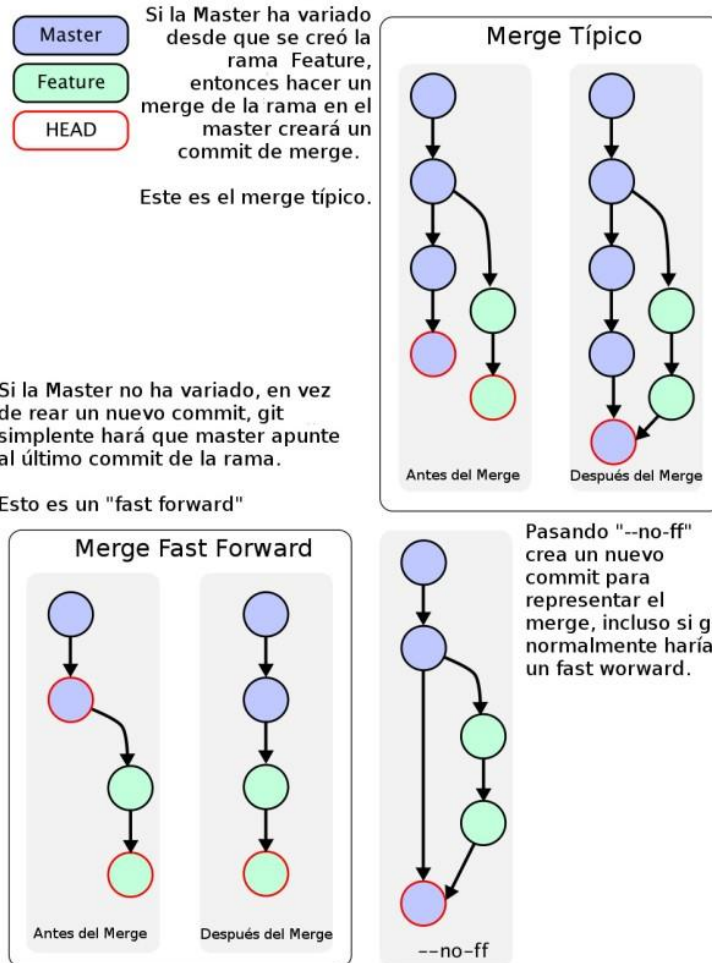
Cuando trabajamos en Git todo el trabajo (menos el commit inicial) ha de realizarse en ramas, no se debe trabajar directamente sobre la rama main.



Tipos de fusiones (merge):

- **Fast forward:** Cuando no hay ningún cambio en la rama principal y los cambios pueden ser integrados de manera transparente. Cada uno de los commits de la otra rama formará parte de la rama principal como si nunca se hubieran separado.
- **Uniones automáticas:** Git detecta que en la rama principal hubo algún cambio que la rama secundaria desconoce, pero git no detecta conflictos.
- **Uniones manuales:** Git detecta cambios en las mismas líneas de los mismos archivos en ambas ramas.

¿Qué es un Merge Fast Forwarded?



- **git branch:** Enumera las ramas del repositorio local y muestra las activas. (<https://git-scm.com/docs/git-branch>)
 - **nombre-de-rama:** Crea una nueva rama con el nombre indicado como una bifurcación de la existente.
 - **-d nombre-de-rama:** Elimina la rama con el nombre indicado.
 - **-m antiguo-nombre nuevo-nombre:** Para cambiar el nombre a una rama existente.
 - **-a:** Enumera también las ramas remotas.
- **git checkout** (<https://git-scm.com/docs/git-checkout>)
 - **-b <nueva-rama>:** Crea la rama indicada y se mueve a esta.
 - **nombre-de-rama:** Pasa de una rama a otra.
 -
- **git merge nombre-de-rama:** Se utiliza para unir dos ramas. (<https://git-scm.com/docs/git-merge>). Hay que usarlo en la rama sobre la que se quieren volcar los cambios.

Ejercicio 26: Provoca un fast forward y haz todos los status y logs necesarios para justificarlo. Cuando acabes de utilizar la nueva rama bórrala.

Ejercicio 27: Provoca una unión automática y haz todos los status y logs necesarios para justificarlo. Cuando acabes de utilizar la nueva rama bórrala.

Ejercicio 28: Provoca una unión manual y haz todos los status y logs necesarios para justificarlo. Cuando acabes de utilizar la nueva rama bórrala. ¿Cómo se ha solucionado el conflicto?

Ejercicio 29: Vamos a simular la elaboración de un proyecto (los ficheros que se han de hacer estarán vacíos).

- Empieza un proyecto nuevo.
- Haz el README.
- Haz la sección de login (primero se hace la vista, luego el controlador y luego el modelo).
- Haz la sección de registro (primero se hace la vista, luego el controlador y luego el modelo).
- Haz la sección de olvido clave (primero se hace la vista, luego el controlador y luego el modelo).
- Haz la sección de splash screen (primero se hace la vista, luego el controlador y luego el modelo).

Haz todos los status y logs necesarios para justificarlo.

1.8. Etiquetas

Las etiquetas son referencias que apuntan a puntos concretos del historial de git.

Se utilizan para capturar un punto en el historial que se considera como importante e indica una versión del proyecto, correspondiendo esta con un commit específico.

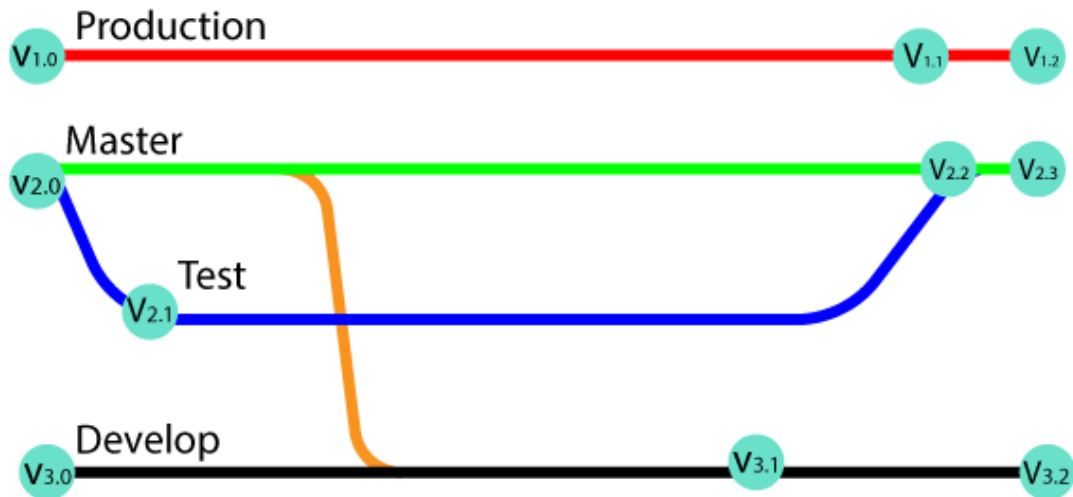
Para crear versiones por número se utiliza la siguiente nomenclatura: X.Y.Z

- **X:** Se conoce como versión mayor y nos indica la versión principal del software.
- **Y:** Se conoce como versión menor y nos indica nuevas funcionalidades.
- **Z:** Se conoce como revisión e indica arreglos de código por algún fallo.

Los tags se usan, principalmente, indicar versiones y le dan un nombre a un commit específico.

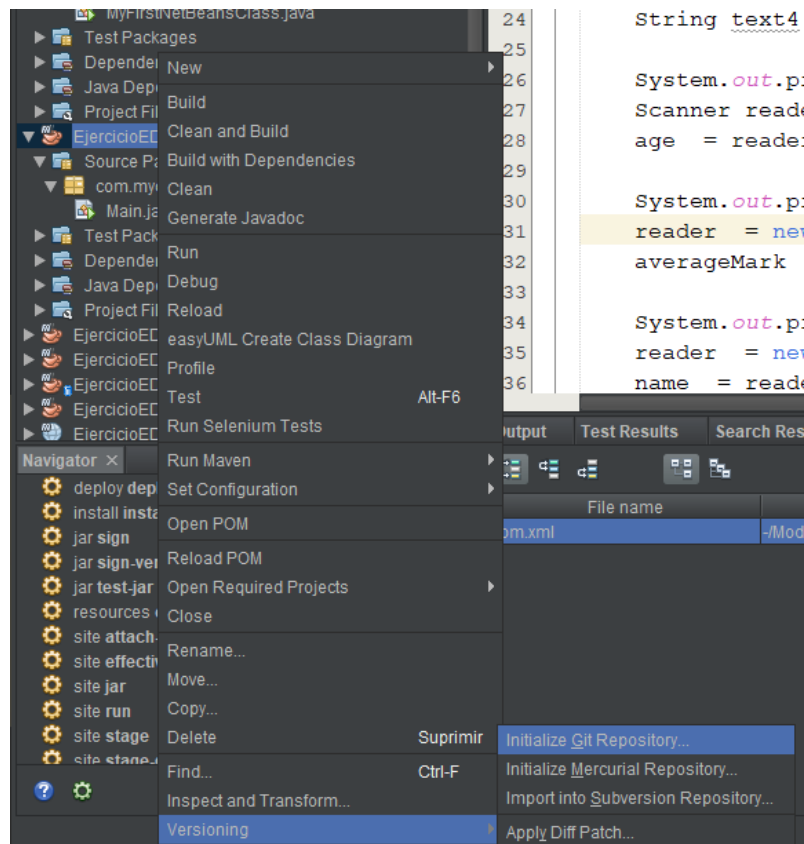
- **git tag:** Lista las etiquetas almacenadas en un repositorio. (<https://git-scm.com/book/en/v2/Git-Basics-Tagging>)
 - **-a versión -m "mensaje-del-tag":** Para crear una nueva etiqueta correspondiente al estado del repositorio en el momento en que se creó la etiqueta con un mensaje.

- **-a versión hash:** Para crear una nueva etiqueta correspondiente al estado del repositorio cuando se hizo el commit al que referencia el hash.
- **-d nombre-del-tag:** Borra la etiqueta indicada.



Ejercicio 28: Crea tres etiquetas, una en el estado actual del repositorio y dos correspondientes a estados anteriores. Borra una de las etiquetas. Muestra en todo momento las comprobaciones realizadas para asegurar el buen funcionamiento del ejercicio.

1.9. Usar git en NetBeans



Ejercicio 29: Utilizando NetBeans (y comprobando con la cmd):

- Inicializa un repositorio.
- Haz un commit inicial.
- Realiza modificaciones y cambios e indica cómo se visualiza en el entorno.
Haz commit
- Elimina un fichero y analiza cómo se marca. Haz commit.
- Crea un fichero y haz que git lo ignore.
- Revierte el proyecto a un estado anterior y devuélvelo al estado actual.
- Crea una rama y provoca un conflicto con el merge. Soluciónalo. ¿Cómo se soluciona en NetBeans?.
- Crea una etiqueta.

1.10. Operar con la nube

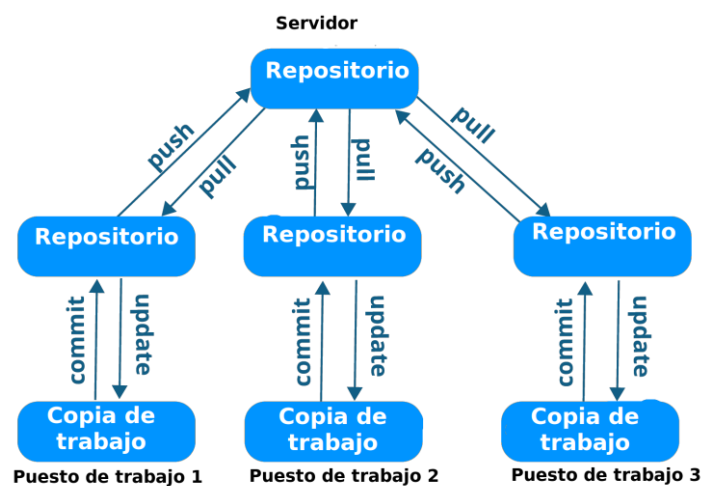
Hasta ahora nos hemos quedado en el trabajo local, por tanto, si nuestro ordenador se estropea perderíamos todo el trabajo. Además de momento no habríamos podido trabajar en equipo.



GitHub, inicialmente, era un servidor remoto que trabaja con Git y nos permite almacenar los repositorios en la nube y administrarlos.

Actualmente sigue siendo esto, pero debido a su éxito, ha crecido mucho y ofrece más posibilidades convirtiéndose una plataforma de desarrollo colaborativo de software de manera remota.

La versión gratuita de GitHub es muy potente.







Ejercicio 30: Crea una cuenta de GitHub.

Para poder explicar este apartado es necesario que hagamos un repositorio local. (La temática no tiene por qué ser la misma). El repositorio contendrá la información de un equipo de fútbol:

- Un fichero jugadores.
- Un fichero entrenadores.
- Un fichero con los médicos.
- Un fichero con los utilleros.
- Un fichero con los títulos.
- Una carpeta con una breve descripción de los hitos históricos del club.

Es necesario hacer, al menos, 20 commits, 6 ramas que tengan sus propios commits y se acaben fusionando en la rama principal y 3 tags.

Ejercicio 31: Crea el repositorio local que se acaba de indicar.


 Search or jump to... [/](#) [Pulls](#) [Issues](#) [Marketplace](#) [Explore](#)   

Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)


Owner *


Repository name *

 srunzu15jcy1 /

Great repository names are short and memorable. Need inspiration? How about [studious-chainsaw?](#)

Description (optional)

☒  **Public**
Anyone on the internet can see this repository. You choose who can commit.

☐  **Private**
You choose who can see and commit to this repository.

Initialize this repository with:

Skip this step if you're importing an existing repository.


☐ **Add a README file**
This is where you can write a long description for your project. [Learn more.](#)

☐ **Add .gitignore**
Choose which files not to track from a list of templates. [Learn more.](#)

☐ **Choose a license**
A license tells others what they can and can't do with your code. [Learn more.](#)

Create repository

Quick setup — if you've done this kind of thing before

 Set up in Desktop or [HTTPS](#) [SSH](#)

Get started by [creating a new file](#) or [uploading an existing file](#). We recommend every repository include a [README](#), [LICENSE](#), and [.gitignore](#).

...or create a new repository on the command line

```
echo "# repositorio-prueba" >> README.md
git init
git add README.md
git commit -m "first commit"
git branch -M main
git remote add origin https://github.com/srunzu15jcy1/repositorio-prueba.git
git push -u origin main
```

...or push an existing repository from the command line

```
git remote add origin https://github.com/srunzu15jcy1/repositorio-prueba.git
git branch -M main
git push -u origin main
```

...or import code from another repository

You can initialize this repository with code from a Subversion, Mercurial, or TFS project.

[Import code](#)

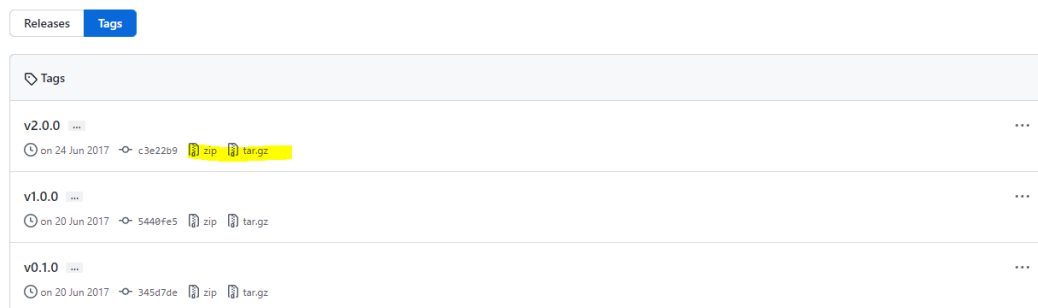
 **ProTip!** Use the URL for this page when adding GitHub as a remote.

- **git remote** : Enumera las conexiones existentes con otros repositorios. (<https://git-scm.com/docs/git-remote>)
 - **-v**: Muestra la url de cada conexión.
 - **add nombre-origen url-repositorio**: Añade una nueva conexión a un repositorio remoto.
 - **rm nombre-origen**: Elimina el repositorio indicado.
 - **rename nombre-antiguo nuevo-nombre**: Renombra un repositorio.

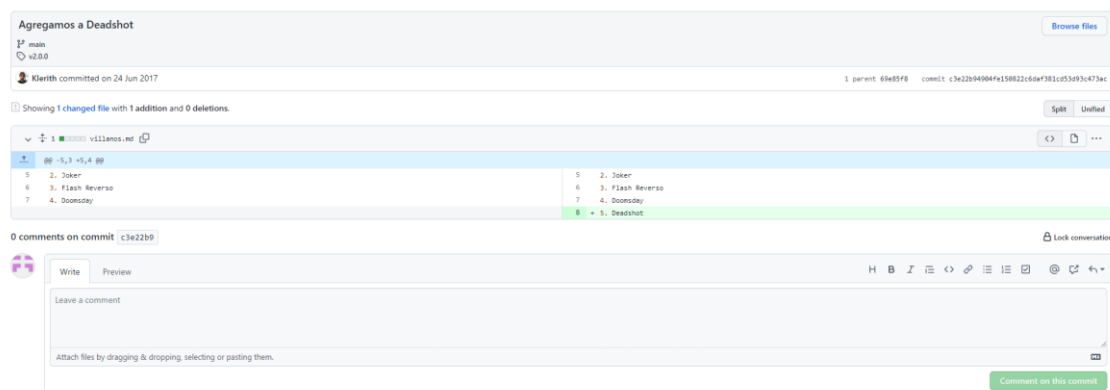
Ejercicio 32: Importa el repositorio creado en GitHub.

Ejercicio 33: Comprueba que el origen remoto se ha importado correctamente.

Ejercicio 34: Añade los tags a GitHub. ¿Qué hace la siguiente opción?



Si pulsamos el hash del commit o en un tag podremos ver los cambios que implementó.



Ejercicio 35: Añade un comentario al commit.

Un release permite prestarles a los usuarios una versión del software en forma de código. Incluye detalles de la versión y se basan en las etiquetas.

Los releases tienen una importancia mayor que las etiquetas.

Ejercicio 36: Crea un release.

Ejercicio 37: Personaliza tu perfil.

- **git clone:** Clona un repositorio en el directorio indicado. (<https://git-scm.com/docs/git-clone>)

Ejercicio 38: Elimina la carpeta del proyecto en local. Importa el proyecto desde GitHub.

- **git push:** Permite subir los commits desde el repositorio local al remoto. (<https://git-scm.com/docs/git-push>)
 - **nombre-repositorio-remoto nombre-rama:** Sube la rama indicada al repositorio de destino.
 - **nombre-repositorio-remoto --all:** Sube todas las ramas al repositorio remoto indicado. (pero no crea tracking)
 - **nombre-repositorio-remoto --tags:** Envía todas la etiquetas locales al repositorio remoto indicado.
 - **-u:** Para crear un tracking de las ramas.
- **git branch -vv:** Muestra el tracking de las ramas.

Ejercicio 39: Modifica el proyecto en el espacio de trabajo local y haz un commit. Analiza el log. Envía este cambio a la nube.

Cuando creamos ramas en local que no se han trackeado en remoto tenemos que sincronizarlas.

Ejercicio 40: Crea 3 ramas en el espacio de trabajo local, haz (al menos) dos commits en cada una de ellas y también dos commits en la rama main. Sube todos los cambios al repositorio remoto y trackea las ramas (comprueba que se ha hecho correctamente). Analiza en todo momento lo que está sucediendo.

- **git fetch:** Extrae y descarga el contenido de un repositorio remoto y lo vuelca en el repositorio local, no obstante, no actualiza el espacio de trabajo.
 - **repositorio-remoto:** Actualiza todas las ramas en el repositorio local conforme su estado en el repositorio remoto.
 - **repositorio-remoto nombre-rama:** Actualiza la rama especificada en el repositorio local conforme su estado en el repositorio remoto.

Una vez hecho el fetch, hay que fusionar los cambios descargados respecto al estado actual del espacio de trabajo local. Para eso hay que ir rama a rama haciendo la fusión.

Ejercicio 41: En el repositorio de GitHub, haz al menos un commit en la rama Main. Descarga estos cambios al repositorio local usando fetch.

- **git pull:** Extrae y descarga el contenido de un repositorio remoto y actualiza el repositorio local para reflejar este contenido. El **pull**, realmente, es una combinación de **git fetch** y **git merge** (<https://git-scm.com/docs/git-clone>)

Ejercicio 42: En el repositorio de GitHub, haz al menos un commit en cada una de las ramas. Descarga estos cambios al repositorio local usando pull. Analiza en todo momento lo que está sucediendo (pista: es necesario hacer el pull rama a rama para que los cambios apliquen al espacio de trabajo local).

Ejercicio 43: Genera un conflicto entre el repositorio del hub y el local y resuélvelo. Analiza en todo momento lo que está sucediendo.

1.11. Uso de Markdown

Markdown es un lenguaje de marcado que facilita la aplicación de formato a una serie de caracteres escritos de una manera especial.

El archivo de README que usa GitHub es un archivo de Markdown.

Ejercicio 44: ¿Cómo se ponen encabezados en Markdown?.

Ejercicio 45: ¿Cómo se crea una lista en Markdown?.

Ejercicio 46: ¿Cómo se pone una imagen en Markdown?.

Ejercicio 47: ¿Cómo se pone una negrita en Markdown? ¿Y una cursiva?.

Ejercicio 48: ¿Cómo se pone una url en Markdown?.

Ejercicio 49: ¿Cómo se pone una cita en bloque en Markdown?.

Ejercicio 50: ¿Cómo se mencionan usuarios en Markdown?.

Ejercicio 51: ¿Cómo se referencia un pull request o una issue en Markdown?.

Ejercicio 52: ¿Cómo se crea una lista de tareas en Markdown?.

Ejercicio 53: ¿Cómo se crea una tabla en Markdown?.

Ejercicio 54: ¿Cómo se añade un emoji en Markdown?.

[Ejemplo de buen uso de Markdown](#)

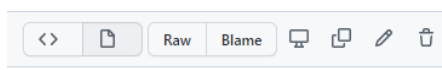
Ejercicio 55: Haz el tutorial de markdown propuesto por Github [Tutorial de Markdown](#)

Ejercicio 56: Modifica el readme del proyecto y haz que sea atractivo.

1.12. Flujo de trabajo de GitHub

Ejercicio 57: ¿Cómo se busca un fichero por el nombre?

Ejercicio 58: ¿Qué significan estas opciones?



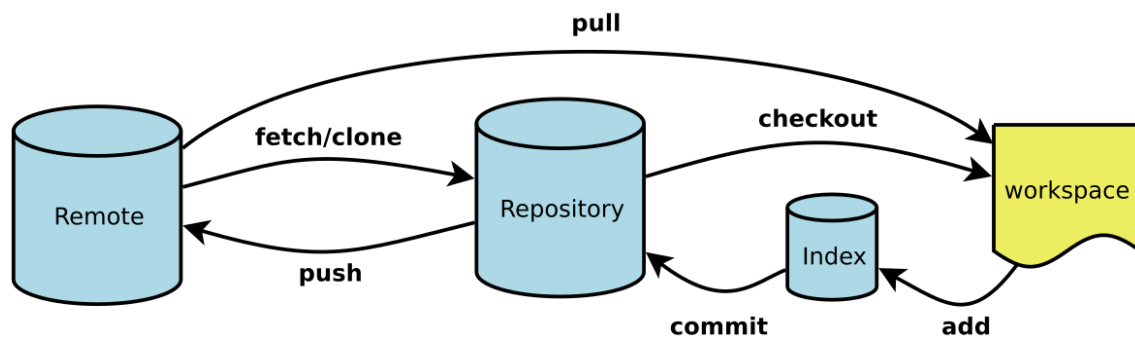
Un pull request es una solicitud de incorporación de cambios-

Normalmente, se utilizan en los flujos de desarrollo cuando un programador termina de realizar la funcionalidad que deseaba en una rama y solicita que se una a la principal. De esta manera, todos los desarrolladores tienen la posibilidad de revisar el código y debatir sobre los cambios. Si el código tiene algún problema se puede solicitar modificaciones al desarrollador.

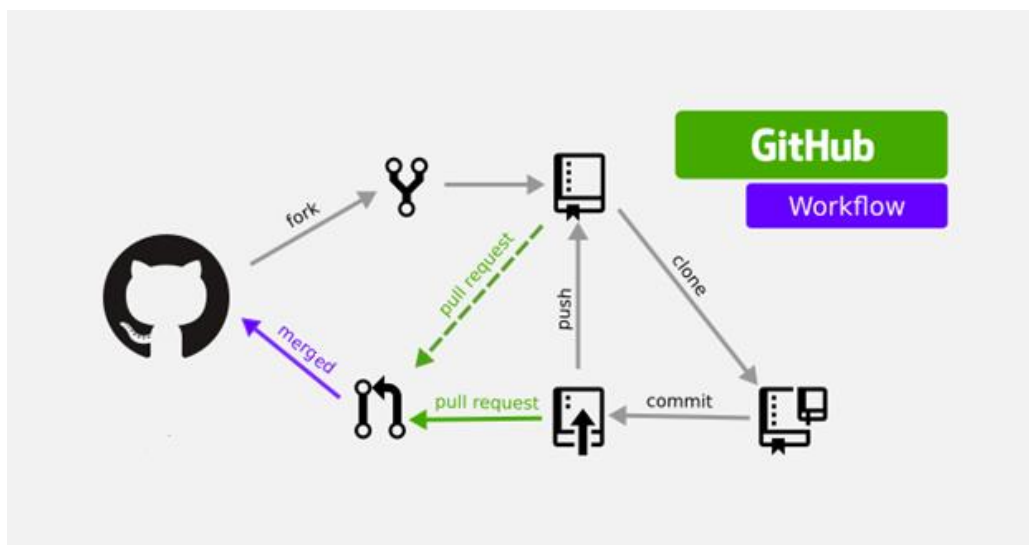
Ejercicio 59: Crea un pull request, coméntalo, modifícalo y acéptalo.

El flujo de uso de GitHub.

- 1º Copiamos o descargamos el repositorio.
- 2º Hacemos una nueva rama.
- 3º Hacemos las modificaciones pertinentes.
- 4º Hacemos un pull request con estas modificaciones.
- 5º Se abre el debate sobre esas modificaciones.
- 6º Se van arreglando los problemas que van surgiendo hasta que la rama esté lista.
- 7º La rama se fusiona con el main.



Un usuario que no es colaborador ni dueño de un repositorio de GitHub no podrá subir sus modificaciones directamente (por mucho que sea público), no obstante, podrá crear una copia de ese repositorio en su cuenta de GitHub, realizar las modificaciones y hacer un pull request para que el propietario del repositorio lo analice y decida si fusionar esa nueva versión del proyecto en su repositorio.

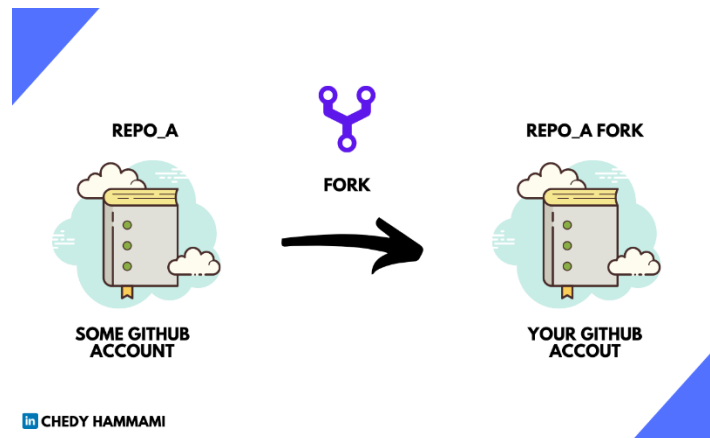


Para crear esta copia de un repositorio se utiliza la herramienta **fork**.

Esta crea una copia del repositorio original en nuestra cuenta de GitHub de la que seremos propietarios, con todo lo que ello conlleva.

A partir de este punto, cada repositorio podría evolucionar de una manera completamente diferente.

El propietario de la copia puede proponer modificaciones al dueño usando **pull request**.



Ejercicio 60: Haz un fork de un proyecto cualquiera de los realizados y publicados por un compañero (lo suyo es que tenga varias ramas). Es importante que este ejercicio se haga uno a uno (el alumno A hará un fork del proyecto del alumno B y viceversa). Llévalo al espacio de trabajo local.

Ejercicio 61: Realiza varios cambios en el repositorio copia y commítealos (realiza esta parte en local y trabajando en otra rama como hemos aprendido). Propón los cambios usando pull request. El compañero debe revisar los cambios y comentarlos solicitando alguna modificación. Realiza la modificación solicitada. El compañero ha de aceptar las modificaciones.

Si el proyecto sobre el que hemos hecho fork se actualiza, tendremos que descargar esos cambios a nuestra copia. Para eso se utiliza la herramienta **fetch upstream**.

Ejercicio 62: Descarga los cambios realizados por el compañero en el repositorio original cuando aceptó tu pull request. Ahora el compañero ha de realizar un cambio en varias ramas. Descarga esos cambios y llévalos al local.

1.13. Gestión de ramas remoto-local

Volvemos a nuestro proyecto.

Para poder hacer tracking de una rama que hemos hecho en local hay que usar el comando **-u** en el push.

Ejercicio 63: ¿Qué pasa si creo una rama en local hago un cambio y lo commito si usar el parámetro -u? ¿Qué solución nos propone Git? .

Ejercicio 64: Fusiona los cambios realizados en todas las ramas que tengas con el main.

Ejercicio 65: Borra ahora todas las ramas en el proyecto remoto y actualiza el proyecto en local. ¿Se han eliminado en local? Si no es así, elimínalas.

Ejercicio 66: Crea una rama en el hub y haz un commit. Descarga las modificaciones al local. Haz un cambio en esa rama en local y publícalo. Acepta el cambio en remoto. Elimina la rama tanto en remoto como en local.

- **git prune origin:** Elimina del repositorio local las ramas remotas que ya no existan en el hub.

Ejercicio 67: ¿Qué ramas se muestran si haces un branch -a? Elimina las que no se usen.

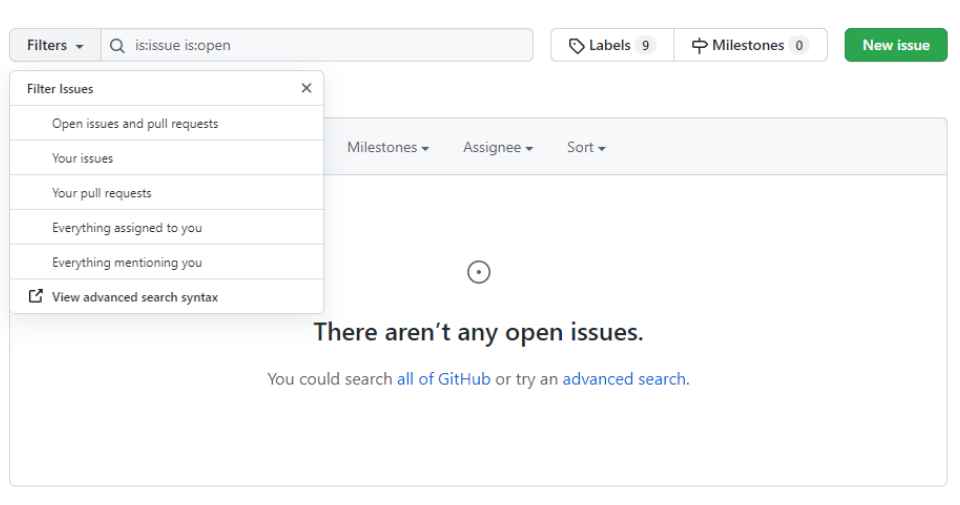
1.14. Colaboradores e issues

Los **colaboradores** son usuarios que tienen acceso de lectura, escritura y administración en el repositorio.

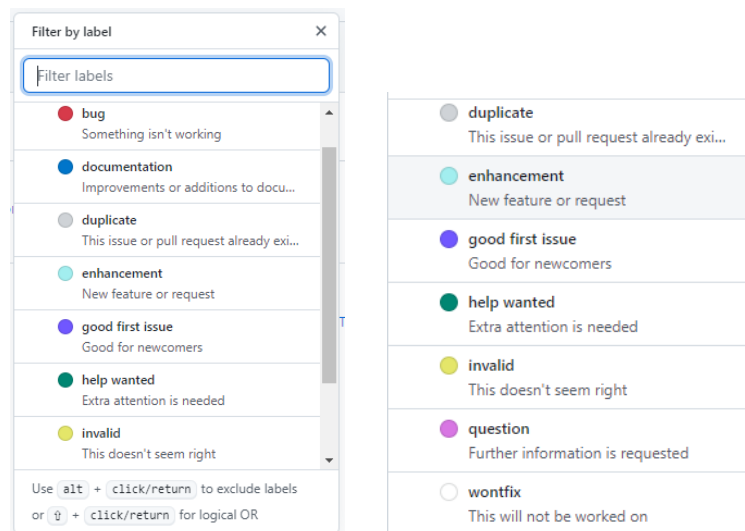
Un repositorio público es visible por todo el mundo, todos pueden hacer forks, pull request, etc.

El concepto **issues**, en GitHub, abarca un gran abanico de posibilidades. Realmente se engloba desde fallos, mejoras, dudas, o cualquier cuestión relativa al repositorio o al proyecto que contiene.

Se utiliza como una especie de foro en el que los usuarios con alguna implicación en el repositorio (ya sea de uso o desarrollo) publican cuestiones referentes a este.



Las issues se agrupan por etiquetas que especifican un poco más su fin.



Un milestone es un agrupador de issues de un proyecto.

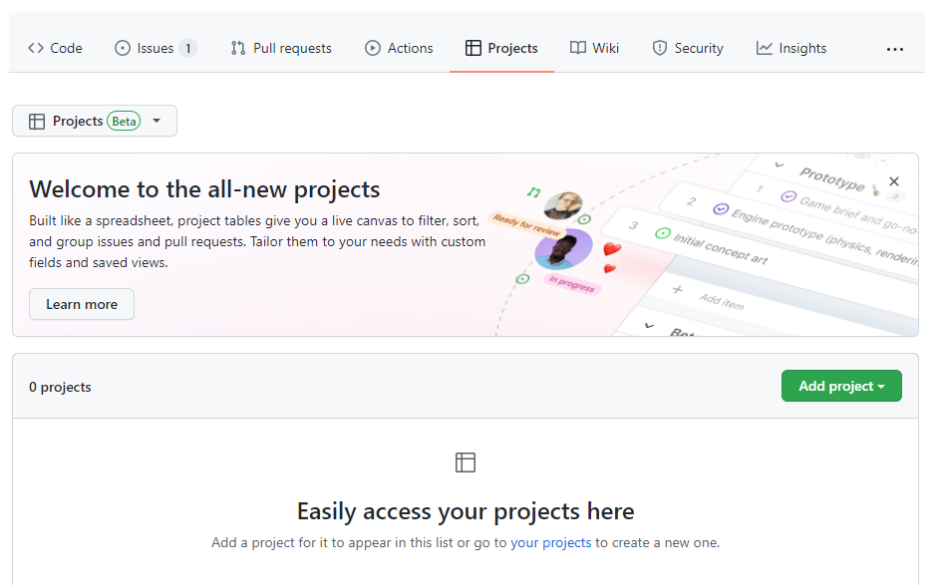
Ejercicio 68: Un compañero ha de crear dos issues (una de cada tipo) en tu repositorio (este ejercicio ha de realizarse uno a uno). Una ha de contener una lista (con 3 cosas más o menos) que a su vez se conviertan en issues. Todas las issues tienen que pertenecer a un milestone llamado “Lanzamiento Beta”. “Solúcialas” en local y publica un commit que “resuelva” cada issue. Cierra las issues indicando el commit que las resuelve.

Ejercicio 69: Añade dos colaboradores al repositorio.

1.15. Proyectos de GitHub

Para poder realizar este apartado es importante crear varias issues correspondientes a nuevas implementaciones.

Ahora creamos un proyecto.



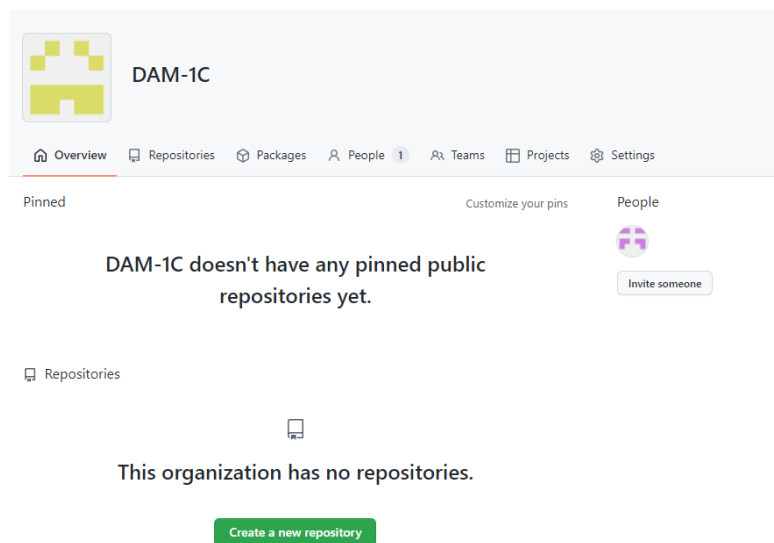
Los proyectos se utilizan para crear flujos de trabajo.

Ejercicio 70: Crea un nuevo proyecto asociado al repositorio con el que hemos venido trabajando y añádele las columnas: Por hacer, urgente, en progreso, en revisión, terminado, cancelado.

Ejercicio 71: Crea varias tareas en cada columna, importa las issues y asígnalas a colaboradores.

1.16. Otras funcionalidades de GitHub

Para publicar código html css y javascript en un servidor se puede usar GitHub pages.



Se pueden crear organizaciones de trabajo.

Si quieres que tu repositorio pase a ser propiedad de otra persona puedes transferírselo.

1.17. Coordinando con GitHub desde NetBeans

Ejercicio 72: Haz un push y un pull desde NetBeans (hay que usar un token como contraseña).

1.18. GitKraken

GitKraken es un cliente de Git que nos permite hacer más amigable el trabajo con esta tecnología.

Descarga GitKraken y configura tu usuario. <https://www.gitkraken.com/>

Ejercicio 73: Crea un repositorio con GitKraken y haz varios commits.

Ejercicio 74: Crea al menos tres ramas con GitKraken y haz varios commits en cada una de ellas. Finalmente fusionalas. Cada rama ha de tener un tipo de fusión distinta (ff, automática y manual).

Ejercicio 75: Añade tu cuenta de GitHub y sube tu nuevo repositorio.

Ejercicio 76: Haz cambios en local (en varias ramas) y súbelos al hub. Haz cambios en el hub y descárgalos usando GitKraken.

Ejercicio 77: Abre el repositorio sobre el que has estado trabajando y analiza el output que brinda GitKraken.

2. Documentación del código fuente

El **código fuente** es el escrito por los programadores en algún editor de texto. Se escribe usando algún lenguaje de programación y contiene el conjunto de instrucciones necesarias para compilar programas.

Es importante documentarlo correctamente para que el equipo de desarrollo sepa **qué se está haciendo y por qué** en todo momento.

Es mucho más sencillo el mantenimiento y desarrollo en un software bien documentado.

Existen muchas herramientas para documentar proyectos como PHPDoc, phpDocumentor, Javadoc o JSDoc.

2.1. JavaDoc

JavaDoc es la herramienta de Java para extraer y generar documentación del código en formato HTML.

Comentar el código siguiendo las recomendaciones de JavaDoc aumentará la utilidad y efectividad.

Los comentarios y el código se van a incluir en el mismo fichero.

Hay tres tipos de comentarios:

- **Comentarios en línea:** Comienzan con `/**` y acaban con la línea.
- **Comentarios tipo C:** Se encuentran dentro de los caracteres `/*` y `*/`. Pueden agrupar varias líneas.
- **Comentarios de documentación Javadoc:** Son comentarios que se colocan dentro de los caracteres `/**` y `*/`. Deben colocarse antes de la declaración de una clase, un campo, un método o un constructor. Dentro de los delimitadores se pueden escribir etiquetas HTML. Están formados por una descripción seguida de un bloque de **etiquetas (tags)**.

ETIQUETA	DESCRIPCIÓN
@autor	Autor de la clase. Solo para las clases.
@version	Versión de la clase. Solo para clases.
@see	Referencia a otra clase, ya sea del API, del mismo proyecto o de otro. Por ejemplo: @see cadena @see paquete.clase#miembro @see enlace
@param	Descripción del parámetro. Una etiqueta por cada parámetro.

@return	Descripción de lo que devuelve. Solo si no es void. Podrá describir valores de retorno especiales según las condiciones que se den, dependiendo del tipo de dato
@throws	Descripción de la excepción que puede propagar. Habrá una etiqueta throws por cada tipo de excepción
@deprecated	Marca el método como obsoleto. Solo se mantiene por compatibilidad.
@since	Indica el nº de versión desde la que existe el método.

```

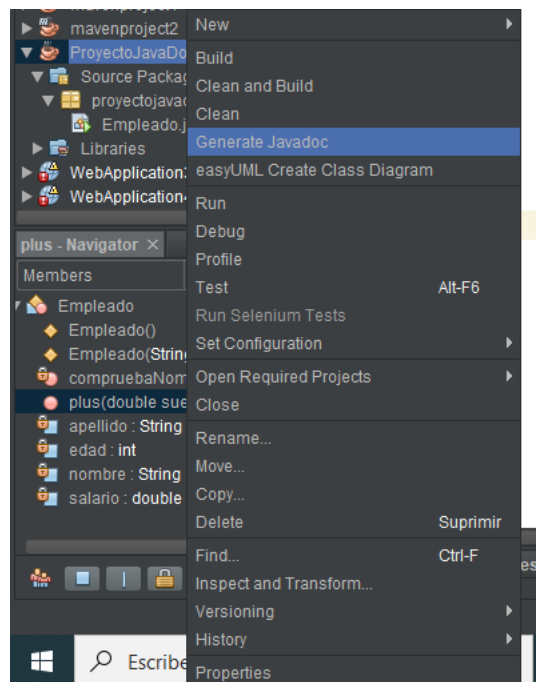
1  /**
2   * Inserta un título en la clase descripción.
3   * Al ser el título obligatorio, si es nulo o vacío se lanzará
4   * una excepción.
5   *
6   * @param titulo El nuevo título de la descripción.
7   * @throws IllegalArgumentException Si titulo es null, está vacío o contiene sólo espacios.
8   */
9
10 public void setTitulo (String titulo) throws IllegalArgumentException
11 {
12     if (titulo == null || titulo.trim().equals(""))
13     {
14         throw new IllegalArgumentException("El título no puede ser nulo o vacío");
15     }
16     else
17     {
18         this.titulo = titulo;
19     }
20 }

```

Ilustración 1: Ejemplo de un método comentado con Javadoc

Vamos a crear un proyecto y generar la documentación.

Se puede generar el html de javadoc desde el entorno de desarrollo.



Ejercicio 78: Genera la documentación de tres de los programas realizados en programación. Es conveniente que estos programas tengan más de dos clases.

REFERENCIAS

<https://github.com/statickidz/TemarioDAW/blob/master/DAW/DAW06.pdf>

<https://git-scm.com/figures/18333fig0103-tn.png>

<https://www.opentix.es/sistema-de-control-de-versiones/>

https://docs.microsoft.com/es-es/devops/_img/branching_line.png

<https://rubensa.wordpress.com/2013/06/25/entendiendo-el-workflow-de-git/>

<https://programmerclick.com/images/645/306c7f60cf9a46c9fd4c744e31ca6775.png>

https://miro.medium.com/max/1400/1*gzc-13rEKx-WWrZaWUsmTw.png

https://miro.medium.com/max/624/1*IeAxduwS_YtpsrRe1d0Q.png

<https://www.atlassian.com/es/git/tutorials/what-is-git>

<https://git-scm.com/book/es/v2/Inicio---Sobre-el-Control-de-Versiones-Acerca-del-Control-de-Versiones>

<https://www.sivsa.com/site/ventajas-controlador-versiones-git-en-programacion/>

<https://image.slidesharecdn.com/wsoeg7n2qhaw2w6ey2nj-signature-af14b15233db106d69af9e7b35aa1fcf254d509b8d654cc7700f025138bab5f6-poli-150612203119-lva1-app6892/95/git-with-scrum-en-espaol-17-638.jpg?cb=1434141138>

<https://d1jnx9ba8s6j9r.cloudfront.net/blog/wp-content/uploads/2017/12/gitHub.png>

<https://platzi.com/clases/1557-git-github/19946-que-es-el-staging-y-los-repositorios-ciclo-basico-/>

https://encrypted-tbn0.gstatic.com/images?q=tbn:ANd9GcQtdzF_jrEoXIHNV5fRpxKD8xY_3VZDbv9pVjMe_k7AdP5kKY2YTCYZ-Xe6jYHcLoLQodS8&usqp=CAU

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

<https://ed.team/blog/como-se-deciden-las-versiones-del-software>

<https://www.javatpoint.com/git-tags>

<https://www.discoduroderoer.es/como-utilizar-javadoc/>