

UD00: Ejercicios Git



1. Áreas de un repositorio GIT

2. Configurar nuestro git

3. Inicializar repositorio local

3. 1. Creamos una carpeta para alojar el proyecto.
3. 2. Comprobamos que tenemos la carpeta vacía
3. 3. Inicializamos el repositorio
3. 4. Comprobamos que se ha creado una carpeta `.git`.
3. 5. Creamos/editamos un archivo `README.md`
3. 6. Registramos cambios en el repositorio
3. 7. Realizar los puntos 3.5, 3.6 y 3.7 otras dos veces
3. 8. Por último vemos cambios realizados

4. Revisar commits realizados

4. 1. Ver el contenido de `README.md` en commit actual
4. 2. Vamos a movernos al primer commit
4. 3. En qué posición de la rama nos encontramos
4. 4. Movernos al segundo commit
4. 5. Vuelve a hacer
4. 6. Volver al commit master

5. Etiquetar commits y ver diferencias

5. 1. Etiquetamos el commit primero y el tercero.
5. 2. Usando etiquetas para movernos
5. 3. Examinar cambios de un commit respecto al anterior
5. 4. Examinar cambios de un commit respecto anteriores
5. 5. Diferencia entre `git show` y `git diff`

6. Crear repositorio remoto y subir commits locales

6. 1. Creamos un repositorio totalmente vacío en GitHub
6. 2. Asociar repositorio local con repositorio remoto
6. 3. Subir todos los commits locales al repositorio remoto
6. 4. Comprobando la subida
6. 5. Examinando commits y releases en GitHub

7. Deshacer cambios en repositorio local

7. 1. Deshacer cambios en el directorio de trabajo
7. 2. ¿Y para deshacer el área de preparación?

7. 3. ¿Y qué pasa si ya realicé un commit?

8. Archivo `.gitignore`

- 8. 1. Creamos una aplicación HolaMundo en Java con nuestro IDE
- 8. 2. Añadiendo archivos al repositorio local
- 8. 3. Subir cambios de repositorio local a repositorio remoto

9. Usando un par de claves SSH

- 9. 1. Generamos un par de claves SSH
- 9. 2. Añadimos clave ssh pública a github.
- 9. 3. Comprobamos que se ha creado bien
- 9. 4. Obteniendo URL SSH del repositorio
- 9. 5. Asociando nuestro repositorio local mediante SSH
- 9. 6. Creamos un commit y subimos a GitHub

10. Resolviendo conflictos

- 10. 1. Modificamos archivo README.md remoto
- 10. 2. Modificamos archivo README.md local
- 10. 3. Intentamos subir el commit local
- 10. 4. Se produce conflicto
- 10. 5. Arreglamos conflicto

11. Creación de ramas

- 11. 1. Crear rama mediante `git checkout`
- 11. 2. Crear ramas con `git branch` ...
- 11. 3. Subir ramas a repositorio remoto

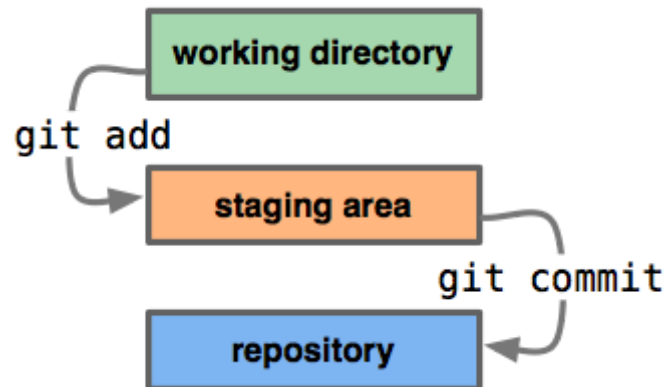
12. Fusión y eliminación de ramas

- 12. 1. Eliminando una rama local
- 12. 2. Fusionando ramas locales
 - 12. 2. 1. Cambiamos a rama `master`
 - 12. 2. 2. Fusionamos rama `licencia`
 - 12. 2. 3. Fusionamos rama autor
 - 12. 2. 4. Fusionamos rama rama1
- 12. 3. Subiendo cambios a repositorio remoto
- 12. 4. Eliminando apuntadores a ramas locales
- 12. 5. Eliminando apuntadores a ramas remotas
- 12. 6. Comprobando cambios en repositorio remoto
- 12. 7. Tarea propuesta para el alumno/a

1. Áreas de un repositorio GIT

En esta actividad deberás buscar información y explicar las 3 áreas de un proyecto Git:

- Directorio de trabajo (*Working directory*)
- Área de preparación (*Staging area*)
- Repositorio (*Directorio .git*)



Subir a la plataforma [AULES](#) un documento PDF de nombre [actividad01tunombre](#) con las capturas de pantalla y explicaciones pertinentes.

2. Configurar nuestro git

Antes de comenzar a utilizar git, debemos configurarlo con los valores que tendrá a partir de ahora (*nombre, correo electrónico, ...*).

Para ello, establecemos:

- el **nombre** de usuario:

```
1 | git config --global user.name "tu_nombre_completo"
```

- el **correo** de usuario:

```
1 | git config --global user.email "tu_direccion_de_correo_electronico"
```

- el **coloreado** de la salida:

```
1 | git config --global color.ui auto
```

- el **estado original en los conflictos**:

```
1 | git config --global merge.conflictstyle diff3
```

Para mostrar la configuración, que ya hemos establecido:

```
1 | git config --list
```

3. Inicializar repositorio local

En la actividad siguiente, vamos a crear un repositorio local, es decir en nuestro PC personal. Luego añadiremos y modificaremos algunos archivos y registraremos los cambios. Trabajaremos desde el terminal de texto.

Debes recopilar la información y capturas necesarias para generar el documento final.

Seguiremos el siguiente proceso:

3.1. Creamos una carpeta para alojar el proyecto.

Por ejemplo, podemos poner nuestro nombre:

```
1 | mkdir pruebas-arturo
```

Y, acto seguido, entrar en dicha carpeta:

```
1 | cd pruebas-arturo
```

3.2. Comprobamos que tenemos la carpeta vacía

```
1 | ls -la
```

3.3. Inicializamos el repositorio

Para inicializar el repositorio se debe ejecutar la siguiente orden **dentro de la carpeta**:

```
1 | git init
```

Puedes observar por el texto `Initialized empty Git repository in ...` que se acaba de crear un repositorio local:

```
$ git init
Initialized empty Git repository in D:/5IES/1DAW-PROGRAMACIO/pruebas-arturo/.git
```

3.4. Comprobamos que se ha creado una carpeta `.git`.

Esta es la carpeta donde se registrarán todos los cambios que vayamos realizando.

```
1 | ls -la
```

```
$ ls -la
total 8
drwxr-xr-x 1 abc 197121 0 Aug 31 07:11 ./
drwxr-xr-x 1 abc 197121 0 Aug 31 07:11 ../
drwxr-xr-x 1 abc 197121 0 Aug 31 07:11 .git/
```

Comprueba el contenido de esta nueva carpeta, ¿por qué **.git** tiene un punto delante?

3.5. Creamos/editamos un archivo README.md

Creamos un fichero (con el editor **nano**, por ejemplo):

```
1 | nano README.md
```

Acto seguido, añadimos a dicho archivo una línea con nuestro *nombre y apellidos*. Guardamos archivo (en nano con **Ctrl+X**):

```
Save modified buffer? |
Y Yes
N No                  ^C Cancel
```

Guardar con el mismo nombre (pulsa **INTRO**):

```
File Name to Write: README.md
^G Help          ^M-D DOS Format    ^M-A Append       ^M-B Backup File
^C Cancel        ^M-M Mac Format    ^M-P Prepend      ^T Browse
```

3.6. Registramos cambios en el repositorio

Para ello deberemos realizar 2 pasos:

Paso 1. Añadimos al área de preparación con la orden `git add ...`:

```
1 | git add README.md
```

Si existen varios ficheros que queremos pasar preparación podemos ejecutar la orden `git add .`:

Paso 2. Añadimos al repositorio local con la orden `git commit -m "mensaje"`:

```
1 | git commit -m "primer cambio Arturo"
```

3.7. Realizar los puntos 3.5, 3.6 y 3.7 otras dos veces

La primera vez añadimos una segunda línea con la *fecha actual* y luego volvemos a hacer `git add ...` y `git commit ...` correspondientes.

La segunda vez añadimos una tercera línea con el *nombre del IES* y luego volvemos a hacer `git add ...` y `git commit ...` correspondientes.

3.8. Por último vemos cambios realizados

Para ver los commit realizados ejecutamos:

```
1 | git log
2 | git log --oneline
```

```
38428b1 (HEAD -> master) tercer cambio Arturo
15cb1e6 segundo cambio Arturo
5e178c8 primer cambio Arturo
```

Deberían aparecer 3 commits.

NOTA: No borrar el repositorio local. Lo volveremos a utilizar en la siguiente actividad.

Subir a la plataforma **AULES** un documento PDF de nombre **actividad02tunombre** con las capturas de pantalla y explicaciones pertinentes.

4. Revisar commits realizados

En esta actividad, haremos uso del comando `git checkout` para movernos por los distintos commits.

Antes de nada comprueba que tienes al menos 3 commits realizados. Para ello ejecuta:

```
1 | git log --oneline --all
```

La opción `--oneline`, nos muestra la información de cada commit en una línea.

La opción `--all`, nos muestra todos los commits.

Debería aparecerte algo semejante a la siguiente imagen:

```
38428b1 (HEAD -> master) tercer cambio Arturo
15cb1e6 segundo cambio Arturo
5e178c8 primer cambio Arturo
```

La primera columna es un **hash**, un identificador.

Los números no están ordenados. En mi caso, el primer commit tiene un hash `5e178c8`. El último commit es el `38428b1`. **Tú deberías tener otros hash distintos**. No te preocupes, es así.

La segunda columna es el **mensaje** que pusimos cuando se hizo el commit.

Fíjate también que en el último commit, en mi caso `38428b1`, existe un **identificador HEAD**. Ésta es una referencia que apunta al commit en el que estamos situados en el momento actual. Además aparece otro **identificador master**, que indica en la rama en la que estamos. Por defecto, siempre es master.

El identificador master siempre apunta al último commit de la rama. Sin embargo el identificador **HEAD** podemos moverlo y desplazarnos entre distintos commit y ver cómo estaban los archivos en cada momento.

Para mover el identificador HEAD utilizamos el comando `git checkout numero_hash`.

Realiza los siguientes pasos y crea las capturas correspondientes:

4.1. Ver el contenido de README.md en commit actual

Para ello:

```
1 | cat README.md
```

Deben aparecer 3 líneas de texto: *tu nombre, la fecha y el IES*.

4.2. Vamos a movernos al primer commit

Para ello hacemos:

```
1 | git checkout 5e178c8
```

Tú deberás poner el hash que tengas en el primer commit.

Te aparecerá un mensaje que contiene "*Te encuentras en estado 'detached HEAD'....*". Esto indica que la referencia HEAD no está al final de la rama. No te preocupes por ello.

Ahora veamos el contenido del archivo `README.md`.

```
1 | cat README.md
```

Debe aparecer sólo una línea con tu nombre. Es el contenido que tenía dicho archivo en ese commit.

4.3. En qué posición de la rama nos encontramos

Para ello ejecutamos:

```
1 | git log --oneline --all
```

Debería aparecer algo semejante a la siguiente imagen:

```
$ git log --oneline --all
38428b1 (master) tercer cambio Arturo
15cb1e6 primer cambio Arturo
5e178c8 (HEAD) primer cambio Arturo
```

Fíjate donde apunta la referencia HEAD en este momento.

Algo que quizás te haya pasado desapercibido pero que es extremadamente **IMPORTANTE** es que cada vez que nos movemos de un commit a otro, el contenido del directorio de trabajo cambia. Esto lo hace git de forma automática.

NO REALIZAREMOS ningún cambio a los archivos, sólo vamos a echar un vistazo.

4.4. Movernos al segundo commit

Para ello hacemos:

```
1 | git checkout 15cb
```

En este caso deberás poner el hash que tengas en tu repositorio como segundo commit. No es necesario poner todos los dígitos, podemos acortar el hash.

Ejecuta:

```
1 | cat README.md
```

y haz una captura de pantalla.

Deberían aparecer 2 líneas: *tu nombre y la fecha*.

4.5. Vuelve a hacer

```
1 | git log --oneline --all
```

Y comprueba que `HEAD` está en el segundo commit.

4.6. Volver al commit master

Para volver al último commit de la rama master, simplemente hacemos:

```
1 | git checkout master
```

Podemos ver que todo está en su sitio haciendo:

```
1 | git log --oneline --all
```

Haz una captura de pantalla.

NOTA: No borrar el repositorio local. Lo volveremos a utilizar en la siguiente actividad.

Subir a la plataforma **AULES** un documento PDF de nombre **actividad03tunombre** con las capturas de pantalla y explicaciones pertinentes.

5. Etiquetar commits y ver diferencias

En esta actividad vamos a ver 3 comandos:

- `git tag`
- `git show`
- `git diff`

El primer comando (`git tag`) nos permite poner etiquetas a los commits.

No se etiquetan todos los commits, sólo las releases que deseemos.

Los 2 siguientes (`git show` y `git diff`) son para ver los cambios realizados entre distintos commits. Son muy parecidos aunque con pequeñas diferencias.

Básicamente `git show` nos permite ver los cambios de un commit respecto al anterior, mientras que `git diff` nos permite ver cambios en un rango de commits.

De todas formas tanto `git show` como `git diff` tienen tantas opciones que aquí sólo nos centraremos en las esenciales.

Empecemos.

5.1. Etiquetamos el commit primero y el tercero.

El primer commit será la versión 1 de nuestro proyecto. La etiqueta será `v1`.

El segundo commit no será etiquetado.

El tercer commit será la versión 2 de nuestro proyecto. La etiqueta será `v2`.

En la captura se muestra un error que más tarde corregiremos en la etiqueta de la v2.

```
abc@ABC MINGW64 /d/5IES/1DAW-PROGRAMACIO/pruebas-arturo (master)
$ git tag -a v1 -m "versión 1" 5e17

abc@ABC MINGW64 /d/5IES/1DAW-PROGRAMACIO/pruebas-arturo (master)
$ git tag -a v2 -m "versión 1" 3842

abc@ABC MINGW64 /d/5IES/1DAW-PROGRAMACIO/pruebas-arturo (master)
$ git log --oneline --all
38428b1 (HEAD -> master, tag: v2) tercer cambio Arturo
15cb1e6 primer cambio Arturo
5e178c8 (tag: v1) primer cambio Arturo
```

Para etiquetar utilizamos el comando

```
1 | git tag -a nombre_etiqueta -m "Mensaje" commit_a_etiquetar
```

Por ejemplo, en mi caso:

```
1 | git tag -a v1 -m "Versión 1" 5e17
2 | git tag -a v2 -m "Versión 2" 3842
```

La opción `-a` significa annotate.

La opción `-m` nos permite poner un mensaje.

Finalmente debemos poner el commit al que deseamos aplicar la etiqueta.

Si por cualquier motivo nos equivocamos al crear la etiqueta podemos eliminarla con:

```
1 | git tag -d nombre_etiqueta
```

Por ejemplo, en el caso anterior nos hemos equivocado en el mensaje de v2, así que:

```
abc@ABC MINGW64 /d/5IES/1DAW-PROGRAMACIO/pruebas-arturo (master)
$ git tag -d v2
Deleted tag 'v2' (was 88cc957)

abc@ABC MINGW64 /d/5IES/1DAW-PROGRAMACIO/pruebas-arturo (master)
$ git tag -a v2 -m "versión 2" 3842
```

5.2. Usando etiquetas para movernos

Las etiquetas nos permiten referenciar commits de una forma más cómoda que usando el identificador de hash.

Por ejemplo es más cómodo usar:

```
1 | git checkout v1
```

que:

```
1 | git checkout 8b67
```

Para volver al último commit:

```
1 | git checkout master
```

5.3. Examinar cambios de un commit respecto al anterior

```
$ git log --oneline --all
38428b1 (HEAD -> master, tag: v2) tercer cambio Arturo
15cb1e6 primer cambio Arturo
5e178c8 (tag: v1) primer cambio Arturo
```

Para ver los cambios introducidos respecto al commit anterior hacemos:

```
1 | git show
```

En este caso, al coincidir todos los apuntadores (HEAD, master, v2 y 3842) al mismo sitio, el comando anterior es equivalente a

```
1 | git show HEAD
2 | git show master
3 | git show 3842
4 | git show v2
```

```
$ git show v2
tag v2
Tagger: Arturo B <arturoblasco@iesmre.com>
Date:   Wed Aug 31 12:03:15 2022 +0200

versión 2

commit 38428b1edb36ba78b1ed3483cbb8dc02664d9332 (HEAD -> master, tag: v2)
Author: Arturo B <arturoblasco@iesmre.com>
Date:   Wed Aug 31 11:41:07 2022 +0200

    tercer cambio Arturo

diff --git a/README.md b/README.md
index 53eef61..d2fa124 100644
--- a/README.md
+++ b/README.md
@@ -1,2 +1,3 @@
   Arturo BC
   05/08/2022
+IES Mestre Ramon Esteve
```

Como podemos observar, se añadió una línea, la que contiene el IES.

Las líneas añadidas aparecen en verde y con un signo +.

Las líneas eliminadas aparecen en rojo y con un signo - (En este caso sólo hemos realizado operaciones de adición).

Para ver el cambio realizado en el commit segundo respecto al primero:

```
1 | git show 15cb
```

Debe aparecer añadida la línea con la fecha.

```
$ git show 15cb
commit 15cble6c4225f6c92341adf549a74952efc13969
Author: Arturo B <arturoblasco@iesmre.com>
Date:   Wed Aug 31 11:38:26 2022 +0200

    primer cambio Arturo

diff --git a/README.md b/README.md
index adc071e..53eef61 100644
--- a/README.md
+++ b/README.md
@@ -1 +1,2 @@
   Arturo BC
+05/08/2022
```

Y para ver el cambio realizado en el commit primero respecto al repositorio vacío:

```
1 | git show v1
```

Debe aparecer añadida la línea con el nombre.

```
commit 5e178c8f08b8faf8e3bbe33532eb14c8eeb0115a (tag: v1)
Author: Arturo B <arturoblasco@iesmre.com>
Date:   Wed Aug 31 07:24:54 2022 +0200

    primer cambio Arturo

diff --git a/README.md b/README.md
new file mode 100644
index 0000000..adc071e
--- /dev/null
+++ b/README.md
@@ -0,0 +1 @@
+Arturo BC
```

5.4. Examinar cambios de un commit respecto anteriores

Si deseamos ver todos los cambios realizados a lo largo de varios commits, haremos uso de `git diff`.

La forma de uso es

```
1 | git diff commit1..commit2
```

Por ejemplo, para ver los cambios entre la versión 1 y la versión 2, hacemos

```
1 | git diff v1..v2
```

```
$ git diff v1 v2
diff --git a/README.md b/README.md
index adc071e..d2fa124 100644
--- a/README.md
+++ b/README.md
@@ -1,3 @@
 Arturo BC
+05/08/2022
+IES Mestre Ramon Esteve
```

Podemos ver que se han añadido 2 líneas desde el commit v1.

Es muy **aconsejable poner primero el commit más antiguo y después el commit más moderno**. Si lo hacemos al contrario, el resultado en lugar de aparecer en color verde aparecerá en color rojo, y su interpretación será más confusa.

5.5. Diferencia entre `git show` y `git diff`

También podemos hacer:

```
1 | git show v1..v2
```

Ejecuta dicho comando y haz una captura de pantalla. Explica brevemente la diferencia respecto a `git diff v1..v2`

NOTA: No borrar el repositorio local. Lo volveremos a utilizar en la siguiente actividad.

Subir a la plataforma **AULES** un documento PDF de nombre **actividad04tunombre** con las capturas de pantalla y explicaciones pertinentes.

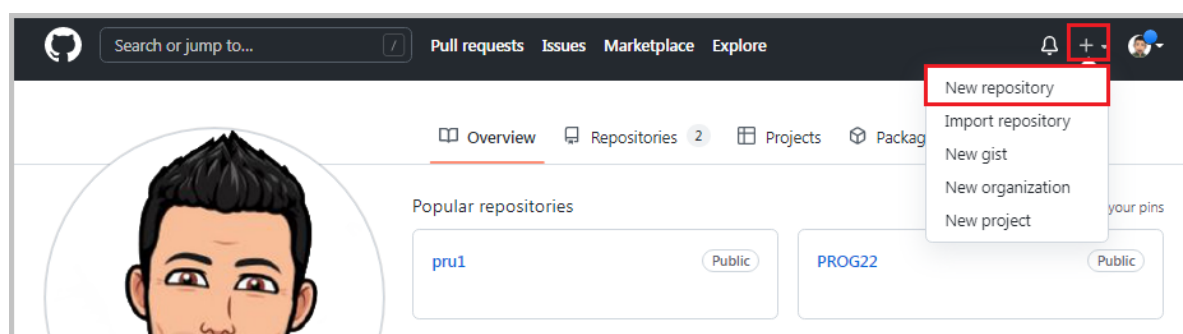
6. Crear repositorio remoto y subir commits locales

En esta actividad crearemos un repositorio vacío en GitHub y subiremos el contenido de nuestro repositorio local.

6.1. Creamos un repositorio totalmente vacío en GitHub

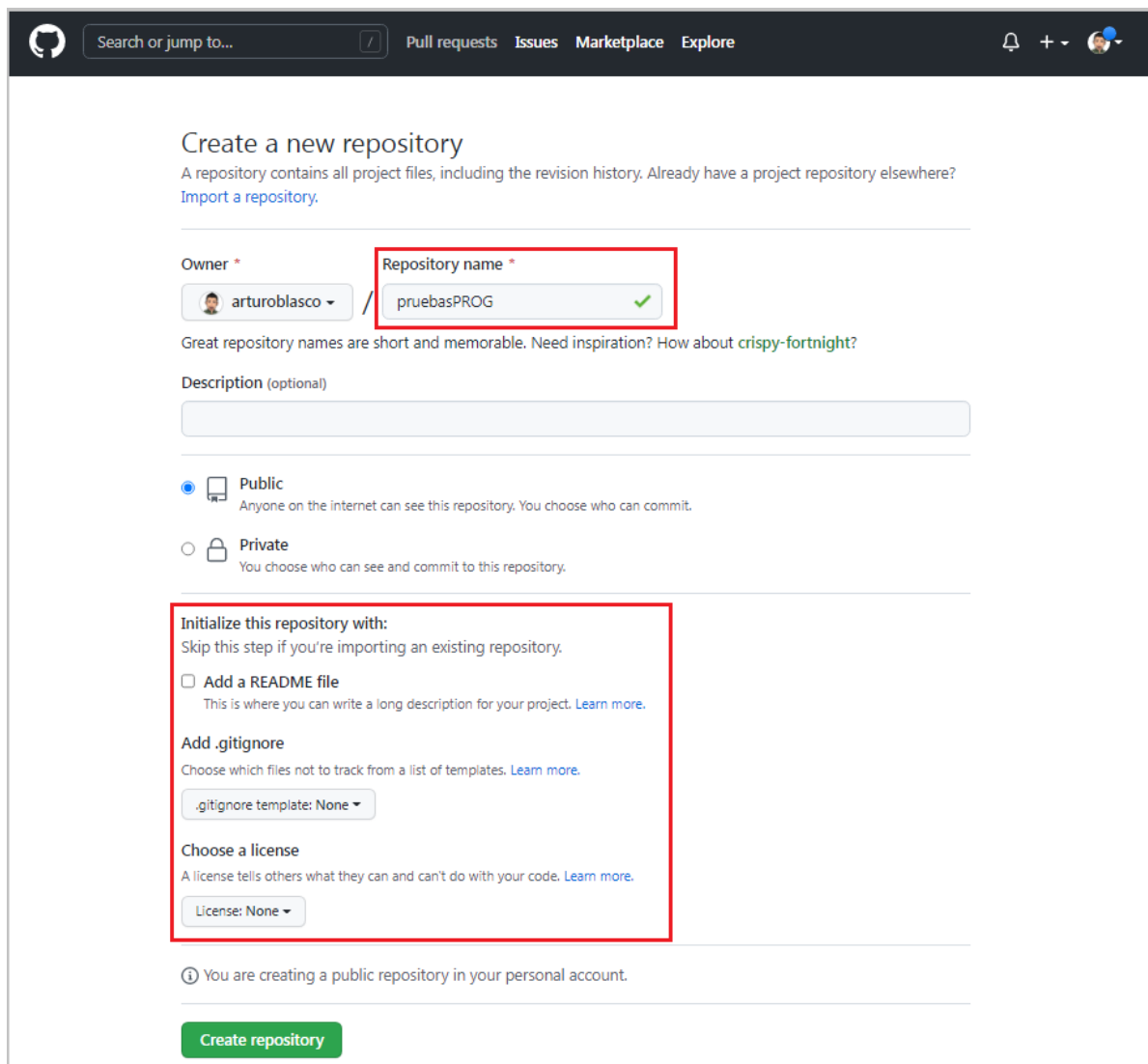
Accedemos a nuestra cuenta de GitHub.

En la **esquina superior derecha**, pulsamos en el signo + y luego en **New repository**



Escogemos el nombre del repositorio. No tiene porqué coincidir con el nombre del repositorio local, aunque es lo aconsejable para no hacernos un lío.


En lugar de *pruebasPROG* pon tu nombre.



Search or jump to... Pull requests 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 *  arturoblasco / Repository name * pruebasPROG ✓

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

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.](#)
.gitignore template: None ▾

Choose a license
A license tells others what they can and can't do with your code. [Learn more.](#)
License: None ▾

 You are creating a public repository in your personal account.

[Create repository](#)

Puedes elegir a tu gusto si el repositorio es público o privado, esto no afectará al resto de secciones.

Es muy importante que **NO INICIALICES EL REPOSITORIO**. Si el repositorio no estuviese vacío podría darnos un conflicto.

En una actividad posterior crearemos conflictos y veremos como resolverlos. Pero en esta actividad, sólo vamos a trabajar lo básico.

Pulsaremos en **Create Repository** y nos aparecerá una página como la siguiente:

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

Set up in Desktop or **SSH** `git@github.com:arturoblasco/pruebasPROG.git`

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 "# pruebasPROG" >> README.md
git init
git add README.md
git commit -m "first commit"
git branch -M main
git remote add origin git@github.com:arturoblasco/pruebasPROG.git
git push -u origin main
```

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

```
git remote add origin git@github.com:arturoblasco/pruebasPROG.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.

Ahí podemos ver la URL del repositorio remoto. Hay 2 formas de acceso:

- mediante HTTPS
- mediante SSH

Usaremos SSH ya que es más seguro y nos permite utilizar cifrado público-privado debido a que recientemente github ha deshabilitado el acceso mediante usuario y contraseña. En el punto 2.1 Configuración con clave pública/privada del archivo UD00_anexo_ES.pdf tienes detallada la configuración y pasos a seguir, si todavía no has configurado tu PC de este modo... **debes hacerlo antes de seguir.**

Más abajo se indican los comandos a ejecutar en nuestro repositorio local. Lo vemos en el siguiente punto.

Para tu comodidad, no cierras la página. Más adelante volveremos a ella.

6.2. Asociar repositorio local con repositorio remoto

En nuestro repositorio local, para asociarlo con el repositorio remoto, hacemos:

```
1 | git remote add origin git@github.com:arturoblasco/pruebasPROG.git
```

Nuestro repositorio remoto será identificado como **origin**. Podemos ponerle otro nombre, pero no debemos. Es una convención ampliamente aceptada poner este nombre al repositorio remoto de GitHub.

Para ver si se ha añadido bien:

```
1 | git remote -v
```

```
$ git remote -v
origin  git@github.com:arturoblasco/pruebasPROG.git (fetch)
origin  git@github.com:arturoblasco/pruebasPROG.git (push)
```

Deben aparecer 2 entradas, una para bajada (fetch) y otra para subida (push)

NOTA: Si por cualquier motivo nos equivocamos y escribimos mal el nombre o la URL, podemos borrar la asociación con

```
1 | git remote remove origin
```

y luego volver a crear la asociación.

6.3. Subir todos los commits locales al repositorio remoto

Para subir el contenido de nuestro repositorio local al repositorio remoto hacemos:

```
1 | git push -u origin master
```

El identificador **origin** es el nombre que dimos a nuestro vínculo. El identificador **master** se refiere a la rama principal.

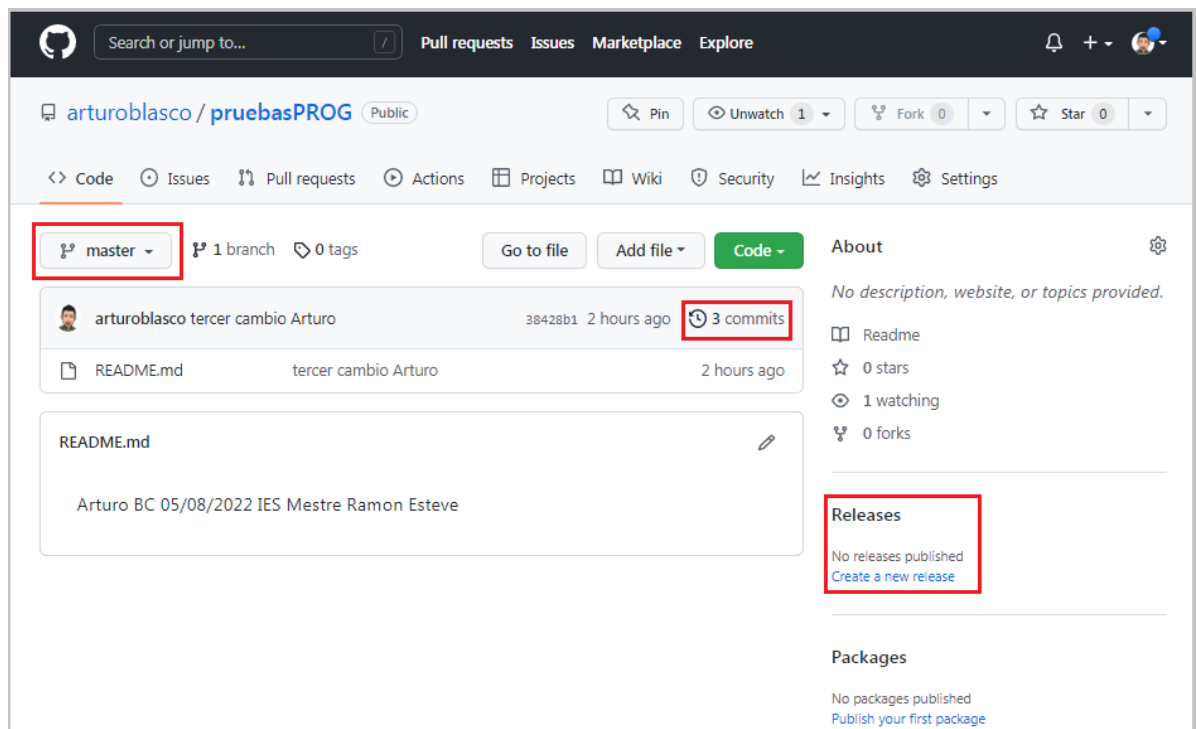
Es una convención ampliamente seguida, así que respétala.

```
$ git push origin master
Enumerating objects: 9, done.
Counting objects: 100% (9/9), done.
Delta compression using up to 4 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (9/9), 714 bytes | 119.00 KiB/s, done.
Total 9 (delta 0), reused 0 (delta 0), pack-reused 0
To github.com:arturoblasco/pruebasPROG.git
* [new branch]      master -> master
```

Si hemos realizado correctamente la configuración de git en nuestro PC se deberían enviar los cambios de nuestro PC al repositorio remoto sin pedir contraseña ya que estamos usando la llave que tenemos configurada en nuestro sistema.

6.4. Comprobando la subida

Volvemos a la página de GitHub y la actualizamos. Nos aparecerá algo semejante a esto:



GitHub ofrece muchas funcionalidades.

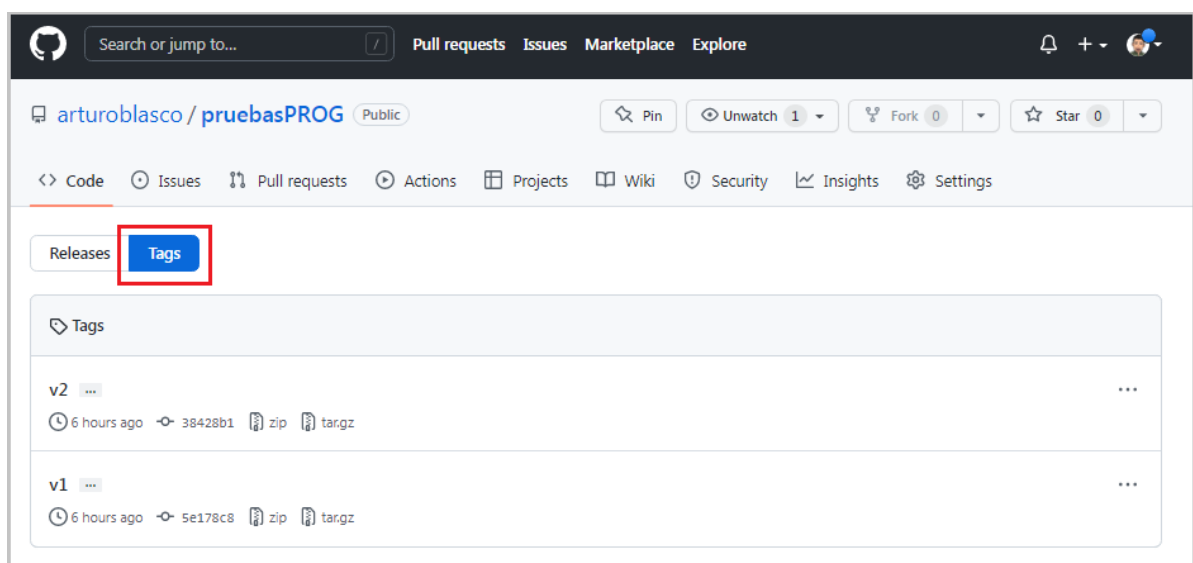
Así que nos centraremos ahora mismo en las *releases*. Estas se corresponden con el etiquetado que realizamos en la actividad anterior con `git tag`.

Teníamos 2 releases, etiquetadas como v1 y v2, pero sin embargo aquí no aparece ninguna.

El motivo, es que debemos subir las etiquetas por separado con el siguiente comando:

```
1 | git push --tags
```

Así que ejecutaremos dicho comando desde nuestro repositorio local. Refrescaremos la página. Et voilà !



6.5. Examinando commits y releases en GitHub

Pulsa en commits y haz una captura de pantalla. Por tu cuenta puedes examinar cada uno de los commits.

Pulsa en Tags y haz una captura de pantalla. Observa que se han creado archivos comprimidos con el código fuente para descargar.

NOTA: No borrar los repositorio local ni repositorio remoto. Los volveremos a utilizar en la siguiente actividad.

Subir a la plataforma AULES un documento PDF de nombre actividad05tunombre con las capturas de pantalla y explicaciones pertinentes.

7. Deshacer cambios en repositorio local

En esta actividad, veremos qué podemos hacer cuando cometemos errores.

Si realizamos algún cambio y hemos "metido la pata", podemos deshacer el "entuerto".

Vamos a verlo de forma práctica haciendo uso del comando `git reset --hard`

7.1. Deshacer cambios en el directorio de trabajo

Estando en el último commit de la rama master, modificamos el archivo `README.md`

Vamos a eliminar las 2 últimas líneas.

```
1 | nano README.md
```

Editamos. Debe quedar una sola línea con *nuestro nombre*.

Para ver los cambios que hemos introducido ejecutamos:

```
1 | git diff HEAD
```

Es decir vamos a ver las diferencias que existen en nuestro directorio de trabajo respecto al commit HEAD, o sea, el último commit confirmado.

NOTA: Si quisiésemos ver las diferencias de nuestro directorio de trabajo respecto al commit de la Versión 1, haríamos `git diff v1`.

Observa que estamos viendo las diferencias hacia el pasado. Esta forma de uso de git diff es diferente a la que vimos en la última actividad, en la cual veíamos las diferencias hacia el futuro.

```
abc@ABC MINGW64 /d/5IES/1DAW-PROGRAMACIO/pruebas-arturo (master)
$ nano README.md

abc@ABC MINGW64 /d/5IES/1DAW-PROGRAMACIO/pruebas-arturo (master)
$ git diff HEAD
diff --git a/README.md b/README.md
index d2fa124..adc071e 100644
--- a/README.md
+++ b/README.md
@@ -1,3 +1 @@
 Arturo BC
-05/08/2022
-IES Mestre Ramon Esteve
```

Se ve claramente que hemos eliminado las 2 últimas líneas.

Para volver el estado de este archivo y de CUALQUIER OTRO de nuestro directorio de trabajo que hayamos modificado, ejecutamos:

```
1 | git reset --hard
```

```

abc@ABC MINGW64 /d/5IES/1DAW-PROGRAMACIO/pruebas-arturo (master)
$ git reset --hard
HEAD is now at 38428b1 tercer cambio Arturo

abc@ABC MINGW64 /d/5IES/1DAW-PROGRAMACIO/pruebas-arturo (master)
$ cat README.md
Arturo BC
05/08/2022
IES Mestre Ramon Esteve

```

7.2. ¿Y para deshacer el área de preparación?

Imaginemos que hemos ido un poco más lejos, y que además de modificar el directorio de trabajo, hemos añadido los cambios al *Staging Area*. Es decir hemos hecho:

```
1 | nano README.md
```

Borrado las 2 últimas líneas.

Y luego hemos añadido al área de preparación mediante

```
1 | git add README.md
```

No te preocupes en este caso puede también aplicarse el comando anterior:

```
1 | git reset --hard
```

Dicho comando coge el contenido que hay en nuestro commit confirmado y recupera ambos: el directorio de trabajo y el área de preparación.

7.3. ¿Y qué pasa si ya realicé un commit?

Imaginemos que hemos ido todavía un poco más lejos, y que además de modificar el directorio de trabajo y añadir los cambios al Staging Area, hemos realizado un commit. Es decir hemos hecho

```
1 | nano README.md
```

Borrado las 2 últimas líneas.

Y luego hemos añadido al área de preparación mediante

```
1 | git add README.md
```

Y además hemos hecho

```
1 | git commit -m "Borras líneas de README.md"
```



```

abc@ABC MINGW64 /d/5IES/1DAW-PROGRAMACIO/pruebas-arturo (master)
$ nano README.md

abc@ABC MINGW64 /d/5IES/1DAW-PROGRAMACIO/pruebas-arturo (master)
$ git add README.md

abc@ABC MINGW64 /d/5IES/1DAW-PROGRAMACIO/pruebas-arturo (master)
$ git commit -m "borrado líneas README.md"
[master 1838184] borrado líneas README.md
1 file changed, 2 deletions(-)

abc@ABC MINGW64 /d/5IES/1DAW-PROGRAMACIO/pruebas-arturo (master)
$ git log --oneline --all
1838184 (HEAD -> master) borrado líneas README.md
38428b1 (tag: v2, origin/master) tercer cambio Arturo
15cb1e6 primer cambio Arturo
5e178c8 (tag: v1) primer cambio Arturo

```

Pues en este caso también podemos usar el comando `git reset --hard` de la siguiente forma:

```
1 | git reset --hard HEAD~1
```

```

abc@ABC MINGW64 /d/5IES/1DAW-PROGRAMACIO/pruebas-arturo (master)
$ git reset --hard HEAD~1
HEAD is now at 38428b1 tercer cambio Arturo

abc@ABC MINGW64 /d/5IES/1DAW-PROGRAMACIO/pruebas-arturo (master)
$ git log --oneline --all
38428b1 (HEAD -> master, tag: v2, origin/master) tercer cambio Arturo
15cb1e6 primer cambio Arturo
5e178c8 (tag: v1) primer cambio Arturo

```

HEAD~1 significa el commit anterior al actual. Es decir **un commit hacia atrás**.

HEAD~2 significa **2 commits hacia atrás**.

HEAD~n significa **n commits hacia atrás**, sustituyendo n por un número.

NOTA: Usar `git reset --hard` de esta última forma es peligroso, porque perdemos el último o últimos commits. Así que hay que asegurarse muy bien de que es eso lo que queremos.

NOTA: No borrar los repositorio local ni el remoto. Los volveremos a utilizar en la siguiente actividad.

Subir a la plataforma **AULES** un documento PDF de nombre **actividad06tunombre** con las capturas de pantalla y explicaciones pertinentes.

8. Archivo .gitignore

En esta actividad empezaremos a trabajar con algo más real. Por ejemplo, una sencilla aplicación de Java. Esta actividad también es práctica.

Vamos a seguir utilizando el repositorio que estábamos usando en las actividades anteriores.

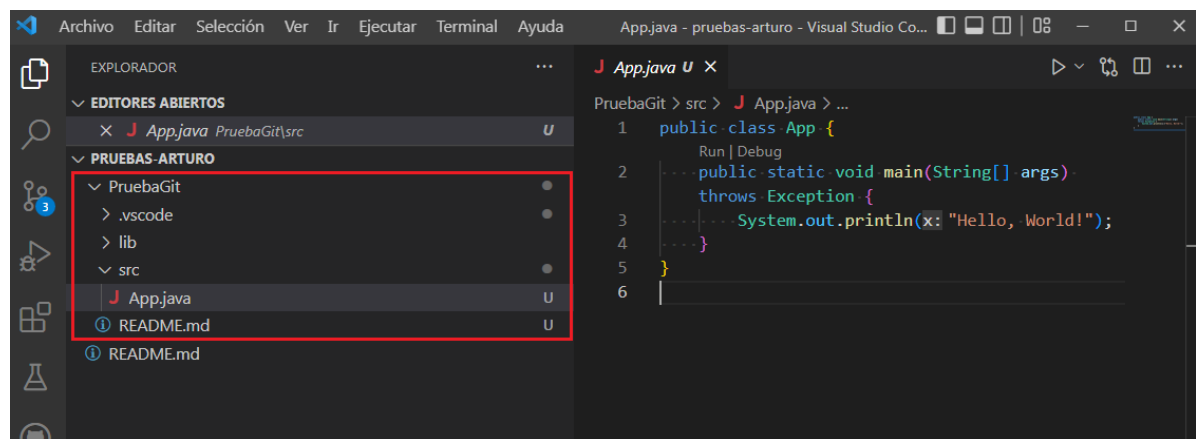
```
1 | git log --oneline --all
```

```
$ git log --oneline --all
38428b1 (HEAD -> master, tag: v2, origin/master) tercer cambio Arturo
15cb1e6 primer cambio Arturo
5e178c8 (tag: v1) primer cambio Arturo
```

8.1. Creamos una aplicación HolaMundo en Java con nuestro IDE

Para ello abriremos nuestro IDE favorito (en mi caso Visual Studio Code) crearemos un nuevo proyecto (en mi caso *PruebasGit*) basado en la misma carpeta en la que tenemos nuestro repositorio local de GIT. Creamos la clase principal, y la modificamos para que pueda imprimir el típico "Hola mundo".

Nuestra estructura de carpetas debería ser algo similar a esto:



8.2. Añadiendo archivos al repositorio local

Como vimos en la actividad anterior, si ahora ejecutamos `git diff HEAD`, esperaríamos ver los cambios de nuestro directorio de trabajo respecto al último commit.

Sin embargo esto no es lo que ocurre. **NO SE MUESTRA NADA.** ¿Por qué es esto?

Esto es porque `git diff HEAD` funciona siempre teniendo en cuenta los archivos que ya habían sido añadidos previamente al repositorio. Es decir sólo tiene en cuenta los archivos con seguimiento.

Los archivos nuevos son archivos sin seguimiento. En este caso debemos usar `git status` para ver esta circunstancia.

```

abc@ABC MINGW64 /d/5IES/1DAW-PROGRAMACIO/pruebas-arturo (master)
$ git diff HEAD

abc@ABC MINGW64 /d/5IES/1DAW-PROGRAMACIO/pruebas-arturo (master)
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    PruebaGit/

nothing added to commit but untracked files present (use "git add" to track)

```

Ahora debemos añadir todos estos archivos al área de preparación (*Staging Area*) y luego realizar un commit.

PERO ESPERA UN MOMENTO. Voy a explicarte algo.

Cuando se trabaja con proyectos de código fuente existen algunos archivos que no interesa añadir al repositorio, puesto que no aportan nada. En el repositorio, como norma general, no debe haber archivos ejecutables, ni bytecode, ni código objeto, y muchas veces tampoco .zip, .rar, .jar, .war, etc. Estos archivos inflan el repositorio y, cuando llevamos muchos commits, hacen crecer demasiado el repositorio y además pueden ralentizar el trabajo de descarga y subida.

Para cada lenguaje y para cada entorno de desarrollo se recomienda no incluir ciertos tipos de archivos. Son los **archivos a ignorar**. Cada programador puede añadir o eliminar de la lista los que considere adecuados. Los archivos y carpetas a ignorar deben indicarse en el archivo `.gitignore`. En cada línea se pone un archivo, una carpeta o una expresión regular indicando varios tipos de archivos o carpetas.

En el repositorio <https://github.com/github/gitignore> tienes muchos ejemplos para distintos lenguajes, herramientas de construcción y entornos.

Para el lenguaje Java: <https://github.com/github/gitignore/blob/master/Java.gitignore>

Para la herramienta Gradle: <https://github.com/github/gitignore/blob/master/Gradle.gitignore>

Para el entorno VsCode: <https://github.com/github/gitignore/tree/main/Global/VisualStudioCode.gitignore>

Nosotros, siguiendo las indicaciones de este último enlace vamos a ignorar las carpetas y archivos sugeridos. Entonces, el archivo `.gitignore` debe tener el siguiente contenido:

```

1  .vscode/*
2  !.vscode/settings.json
3  !.vscode/tasks.json
4  !.vscode/launch.json
5  !.vscode/extensions.json
6  !.vscode/*.code-snippets
7
8  # Local History for Visual Studio Code
9  .history/
10
11 # Built Visual Studio Code Extensions
12 *.vsix

```

La barra final es opcional, pero a mí me gusta ponerla cuando me refiero a carpetas, para así saber cuando se trata de un archivo y cuando de una carpeta.

Crea el archivo `.gitignore` con dicho contenido y haz una captura de pantalla.

Ahora si, hacemos:

```
1 | git add .
2 | git status
```

veremos que no nos aparecen las carpetas `dist`, `build` ni `nbproject/private`, ni ninguno de los archivos omitidos en `.gitignore`.

```
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   .gitignore
    new file:   PruebaGit/.vscode/settings.json
    new file:   PruebaGit/README.md
    new file:   PruebaGit/src/App.java
```

Ahora ya podemos ejecutar

```
1 | git commit -m "Código fuente inicial"
```

Fíjate que he escrito `git add .`. El punto indica el directorio actual, y es una forma de indicar que incluya en el área de preparación todos los archivos del directorio en el que me encuentro (salvo los archivos y carpetas indicados en `.gitignore`) Se utiliza bastante esta forma de `git add` cuando no queremos añadir los archivos uno a uno.

8.3. Subir cambios de repositorio local a repositorio remoto

Ya sólo nos queda subir los cambios realizados al repositorio remoto con `git push`

```
$ git push origin master
Enter passphrase for key '/c/Users/abc/.ssh/id_ed25519':
Enter passphrase for key '/c/Users/abc/.ssh/id_ed25519':
Enumerating objects: 10, done.
Counting objects: 100% (10/10), done.
Delta compression using up to 4 threads
Compressing objects: 100% (7/7), done.
Writing objects: 100% (9/9), 1.28 KiB | 436.00 KiB/s, done.
Total 9 (delta 0), reused 0 (delta 0), pack-reused 0
To github.com:arturoblasco/pruebasPROG.git
 38428b1..e59fde1  master -> master
```

Para hacer algo más interesante este apartado, vamos a crear una etiqueta en el commit actual y subirla a github para que éste cree una nueva *release*.

```
1 | git tag v3
2 | git push --tags
```

```
$ git push --tags
Total 0 (delta 0), reused 0 (delta 0), pack-reused 0
To github.com:arturoblasco/pruebasPROG.git
 * [new tag]          v3 -> v3
```

En este caso, podríamos también haber ejecutado:

```
1 | git push origin v3
```

Y la historia de nuestro repositorio local nos quedaría así de bonita:

```
$ git log --oneline
e59fde1 (HEAD -> master, tag: v3, origin/master) codigo fuente inicial
38428b1 (tag: v2) tercer cambio Arturo
15cb1e6 primer cambio Arturo
5e178c8 (tag: v1) primer cambio Arturo
```

Accede a tu repositorio en GitHub y haz una captura de pantalla de las *Tags*.

Haz otra captura de los archivos y carpetas de código subidas a GitHub. No deberían aparecer la carpeta `lib`. Y sí debería aparecer el archivo `.gitignore`.

NOTA: La carpeta `.git` nunca se muestra en GitHub.

NOTA: No borrar los repositorio local ni repositorio remoto. Los volveremos a utilizar en la siguiente actividad.

Subir a la plataforma **AULES** un documento PDF de nombre **actividad07tunombre** con las capturas de pantalla y explicaciones pertinentes.

9. Usando un par de claves SSH

GitHub ya no permite las conexiones por HTTP, solo por SSH, y esto ya lo hicimos al comenzar los ejercicios, así que te puedes saltar el paso 8 e ir directamente al punto 9. Lo dejo aquí como referencia y consulta.

Como habréis observado, cada vez que hacemos un `git push` nos pide el usuario y contraseña. Esto es bastante molesto.

Una forma de evitar esto es mediante un **par de claves SSH** (una clave privada y una clave pública). Ambas se complementa. La una sin la otra no sirve de nada.

Este método evita que nuestro usuario y contraseña de GitHub se guarde en un archivo de disco. Por tanto es muy seguro. En caso de que alguien haga login en nuestro PC podría acceder a nuestras claves. En dicho caso eliminaríamos el par de claves y volveríamos a crear unas nuevas y nuestro usuario y contraseña de GitHub nunca se verían comprometidos.

Vamos a seguir los siguientes pasos:

9.1. Generamos un par de claves SSH

Es muy sencillo. Como usuario normal (sin ser root) ejecutamos el comando

```
1 | ssh-keygen
```

```
~
ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/jose/.ssh/id_rsa):
/home/jose/.ssh/id_rsa already exists.
Overwrite (y/n)? y
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/jose/.ssh/id_rsa.
Your public key has been saved in /home/jose/.ssh/id_rsa.pub.
The key fingerprint is:
SHA256:n8cPswqt0QaBtQMsHnd7IAUV4DoC7r067pE/gADsrPc jose@lenovo
The key's randomart image is:
+---[RSA 2048]---+
| . .O=+. |
| ..O.= + |
|=. +.* + |
|+O... = . |
|=.O +S |
|+O+. . O O |
| =.O .. + = |
| .O.E O+ . = |
|O+OO. .O.... |
+---[SHA256]---+
```

Pulsamos Intro a todo. Salvo que ya exista un par de claves previo. En ese caso nos preguntará si deseamos sobrescribir (Override (y/n)?) En este caso, en esta pregunta respondemos y . Luego todo Intro.

Esto nos creará una carpeta `~/.ssh` y dentro al menos 2 archivos:

- `id_rsa`
- `id_rsa.pub`

El primero archivo corresponde a la clave privada y el segundo a la clave pública.

Copiamos el contenido de la clave pública en un editor de texto. Nos hará falta más adelante.

```
ls .ssh
id_rsa id_rsa.pub known_hosts known_hosts.old

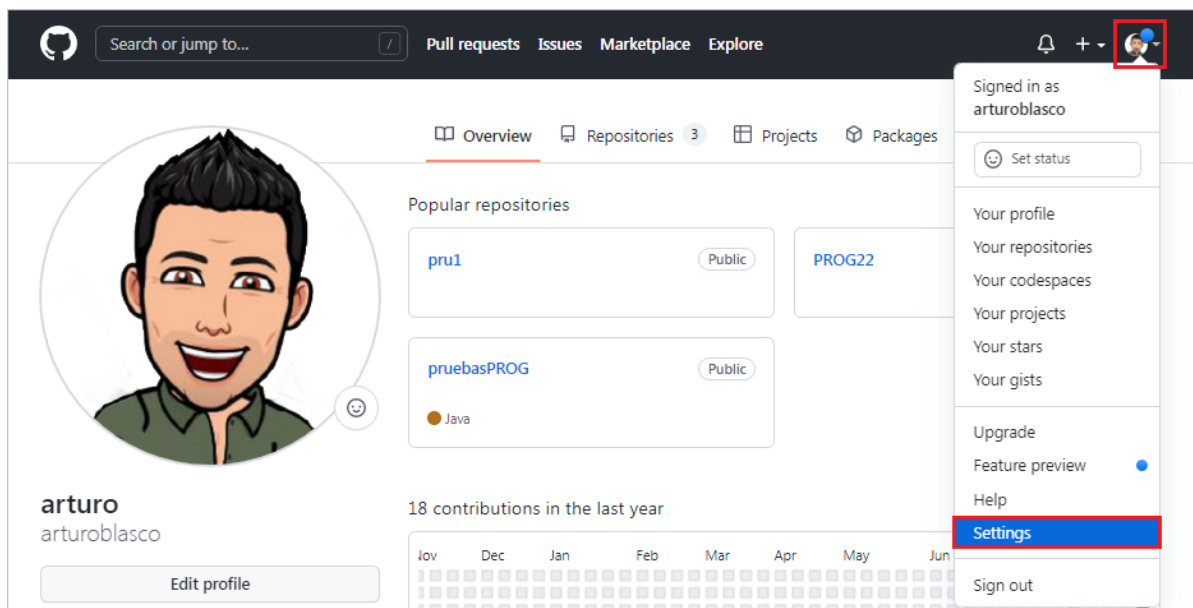
cat .ssh/id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAQDPBICjppHYLyMpCXsmr88v15GvuJ3+TSXiIWQKF0ZwybiznsX81L3JuL/Id2H1
8ZT35J2pG/X1jMtun+UrsMMPMRj3LT87ZVUv0J//0FenmeBZ1nHg+uLYwMP8PhR9W0NSAnllQGeylt53dfW7xUfWLMUia7D0Ll9
8KLnP0ASyjf66L7nhFiCTRumY8P0gfn4qkIz7xiwxHtILOXmzwGhvtArr9k6tvRlpl6IxX6Z2m3YHpbj6Ry5WRG/OMXX8lcu5DP
Q9+/3F0fvN0TLau26MVUfh4G8PdrIZ1NEHz2RL4v2Ij2sfb0shTVmyIBD8XWrFLho9I+2V6sPRUsejdH jose@lenovo
```

Debe copiarse `ssh-rsa jose@lenovo`

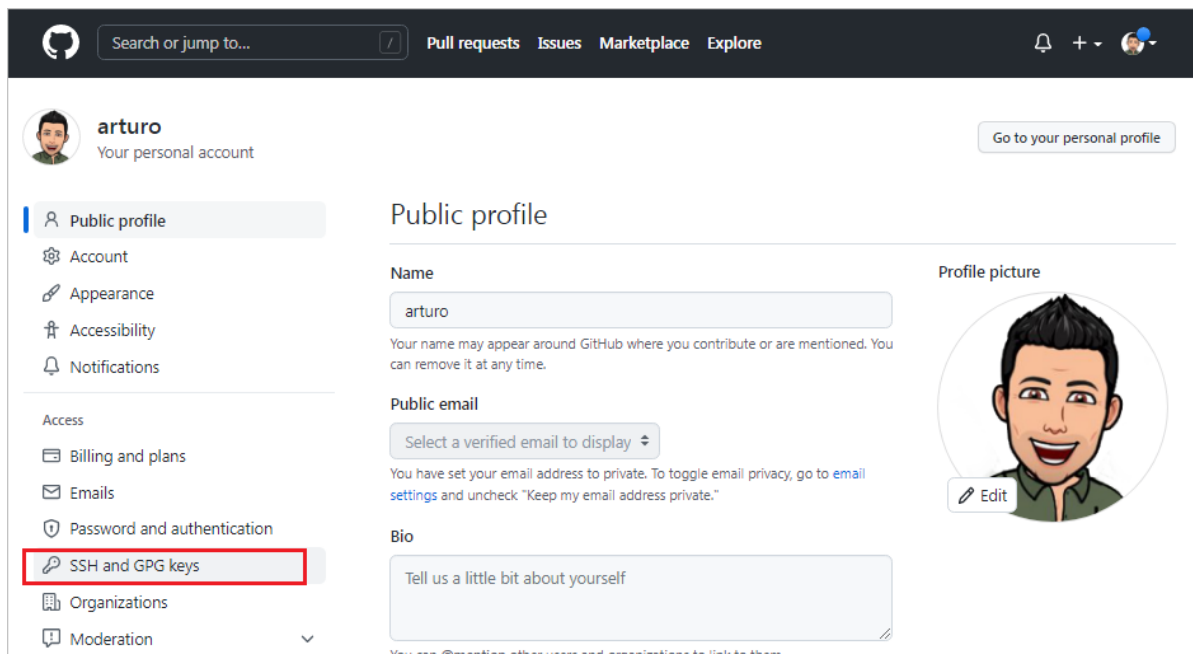
En vuestro caso, en lugar de `jose@lenovo` aparecerá otro usuario y pc.

9.2. Añadimos clave ssh pública a github.

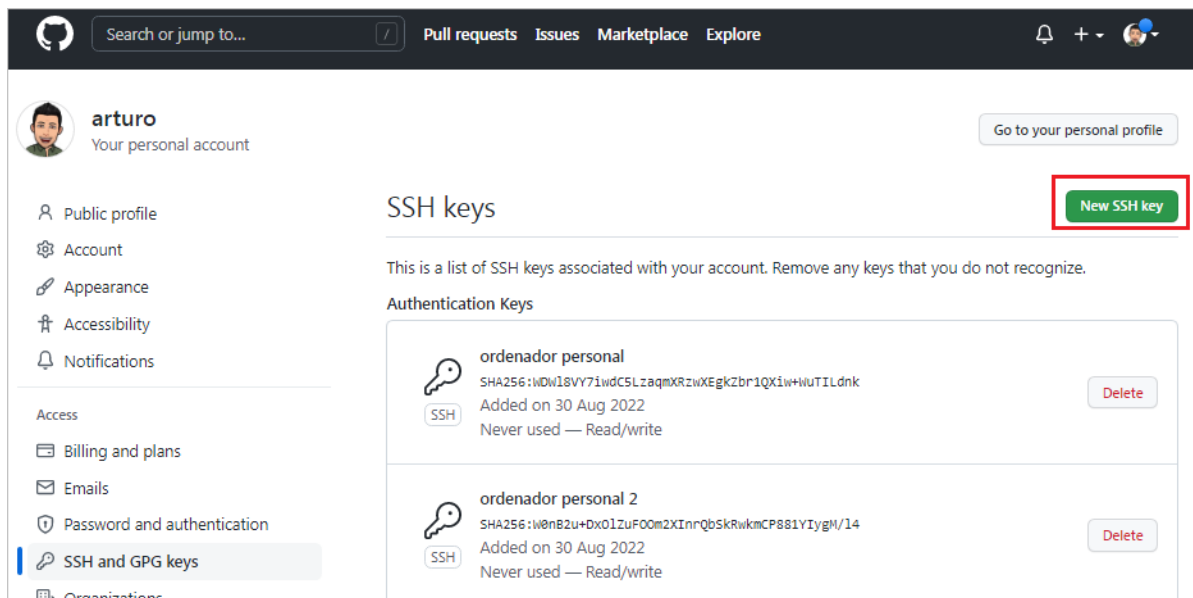
Iniciamos sesión de GitHub y en el menú general (esquina superior derecha) seleccionamos la opción **Settings**.



Luego, en la parte izquierda, elegimos la opción **SSH y GPG keys**



A continuación, a la derecha, pulsamos en el botón **New SSH key**



Luego ponemos un nombre a la clave, por ejemplo pc-casa. Y copiamos el contenido de la clave pública. Finalmente, pulsamos en el botón **Add SSH key**

The screenshot shows the GitHub interface for user 'arturo'. The left sidebar contains navigation links: Public profile, Account, Appearance, Accessibility, Notifications, Access (Billing and plans, Emails, Password and authentication, SSH and GPG keys, Organizations, Moderation), and Code, planning, and automation (Repositories, Packages, GitHub Copilot). The main content area is titled 'SSH keys / Add new'. It contains a form with the following fields:

- Title:** A text input field containing 'pc-personal-casa'.
- Key type:** A dropdown menu set to 'Authentication Key'.
- Key:** A large text area containing the SSH public key: `ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIPXZ5Xuz35VSwTUUE6dgReggOVz7Ksz/68TYd47K89E6 arturoblasco@iesmre.com`.
- Add SSH key:** A green button at the bottom of the form.

La clave anterior puede usarse para cualquiera de nuestros repositorios. Para hacer uso de ella, lo único que necesitamos es la URL en formato SSH de cada repositorio.

9.3. Comprobamos que se ha creado bien

Si, por cualquier motivo, alguien accediera a nuestro PC y cogiera la clave privada, bastaría con eliminar esta clave pública de GitHub y al ladrón no le serviría de nada nuestra clave privada.

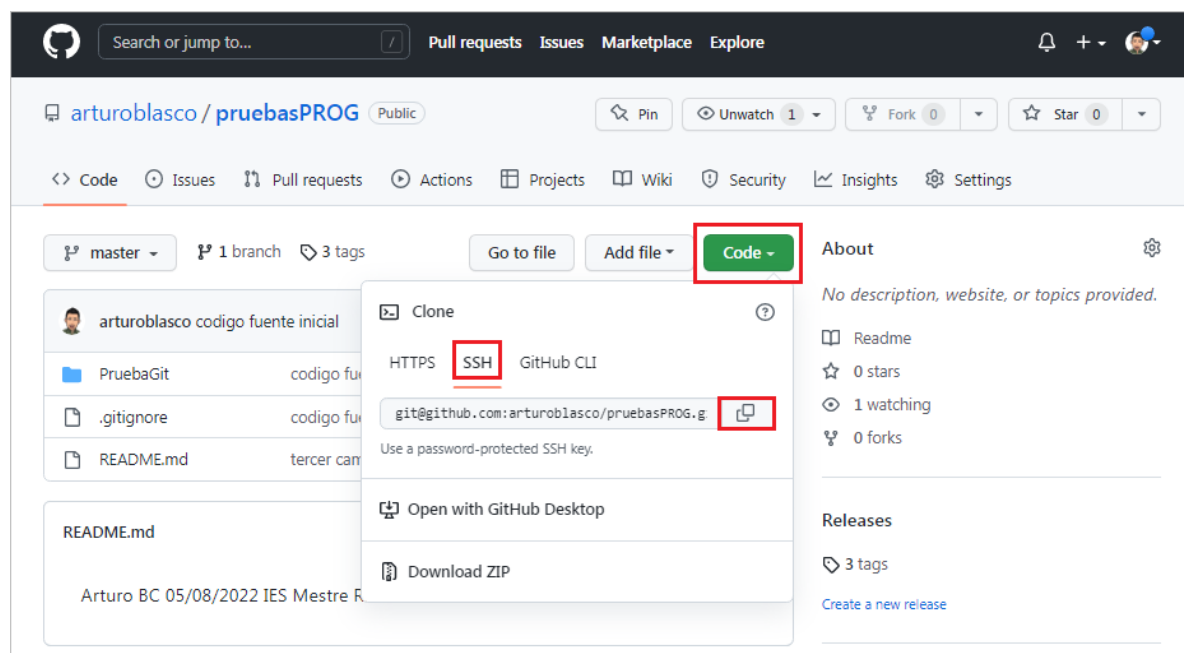
The screenshot shows the GitHub 'SSH keys' page for user 'arturo'. The left sidebar is identical to the previous screenshot. The main content area is titled 'SSH keys' and includes a 'New SSH key' button. Below the title, it states: 'This is a list of SSH keys associated with your account. Remove any keys that you do not recognize.' The page lists three 'Authentication Keys':

Icon	Name	SHA256	Added	Status	Action
SSH	ordenador personal	SHA256:WdW18VY7iWdCSLzaqmXRZwXegkZbr1QXiW+uUTILdnk	Added on 30 Aug 2022	Never used — Read/write	Delete
SSH	ordenador personal 2	SHA256:W8nB2u+Dx0LzuFOm2XInrQbSkRwkmCP881YIyGw/14	Added on 30 Aug 2022	Never used — Read/write	Delete
SSH	ordenador personal 3	SHA256:bIS5qR8+m7H+sRapfZLeftjTKAl1qIUIW8SyTh1+bM	Added on 30 Aug 2022	Last used within the last week — Read/write	Delete

9.4. Obteniendo URL SSH del repositorio

Botón **Clone or download, Use SSH**

Copiamos URL en formato SSH. Su formato es relativamente fácil de memorizar. Siempre [git@github.com](https://github.com) seguido de dos puntos : y luego el **nombre de usuario / nombre de repositorio**.

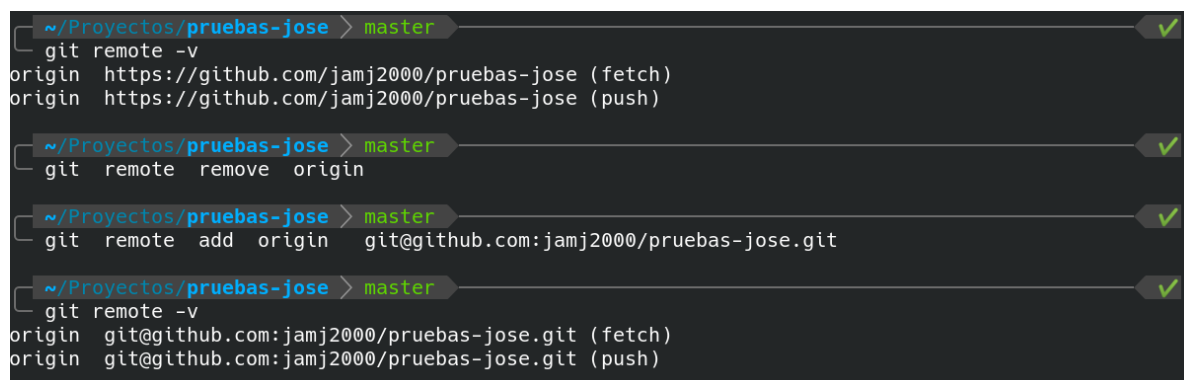


9.5. Asociando nuestro repositorio local mediante SSH

Nuestro repositorio local estaba asociado a origen mediante HTTPS. Debemos dar de baja dicho enlace y crear uno nuevo que haga uso del protocolo SSH.

Ejecutamos:

```
1 git remote remove origin
2 git remote add origin git@github.com:tu_usuario/tu_repositorio
```



9.6. Creamos un commit y subimos a GitHub

Para comprobar que no nos pide usuario y contraseña cuando hagamos git push, vamos a modificar el archivo README.md, crear un commit y subir a GitHub.

Pondremos al principio de cada línea el símbolo > y un espacio. El archivo README.md quedaría más o menos así:

```
1 > Arturo BC
2 > 12 Noviembre 2022
3 > IES Mestre Ramón Esteve
```

Luego guardamos. Ejecutamos:

```
1 git add README.md
2 git commit -m "Añadida cita"
3 git push -u origin master
```

Al ejecutar el último comando, se realizará una conexión SSH con GitHub.

```
~/Proyectos/pruebas-jose > master
git push -u origin master
The authenticity of host 'github.com (140.82.118.3)' can't be established.
RSA key fingerprint is SHA256:nThbg6kXUpJWGl7E1IGOCspRomTxdCARLviKw6E5SY8.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'github.com,140.82.118.3' (RSA) to the list of known hosts.
```

Cuando se realiza una conexión SSH con una nueva clave, la primera vez se pide confirmación y deberás escribir **yes**. Después de ello, quedará registrado el host remoto en el archivo `.ssh/known_hosts`. Las siguientes veces ya no se pide confirmación, siempre que el archivo `.ssh/known_hosts` contenga dichos registros.

NOTA: No borrar los repositorio local ni repositorio remoto. Los volveremos a utilizar en la siguiente actividad.

Subir a la plataforma **AULES** un documento PDF de nombre **actividad08tunombre** con las capturas de pantalla y explicaciones pertinentes.

10. Resolviendo conflictos

En esta actividad veremos qué se entiende por conflicto, cuándo se produce y cómo resolverlo.

Como sabéis un mismo repositorio puede tener copias en distintos sitios. Ahora mismo tenemos una copia en GitHub y otra local en nuestro PC. Pero podrían existir más copias locales en otros PC.

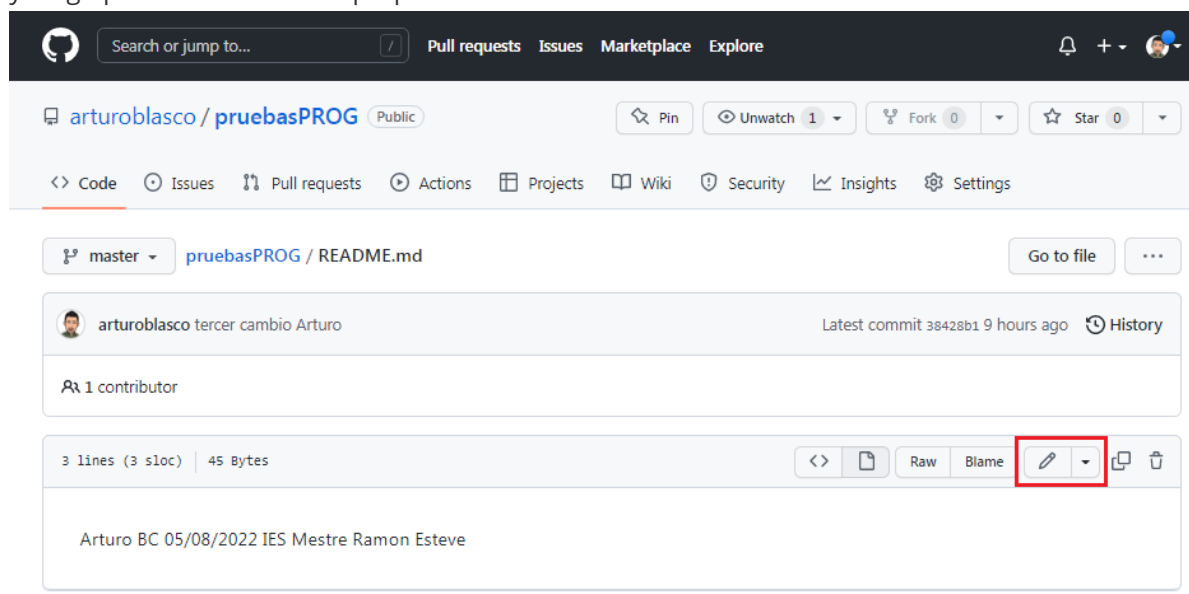
Siempre que realicemos cambios (es decir commits) en el mismo archivo en las mismas líneas pero en copias distintas, se producirá un conflicto.

Para ver esto, vamos a hacer un commit en nuestro repositorio en GitHub, y luego haremos un commit en nuestro repositorio local. Trabajaremos con el archivo `README.md` únicamente.

10.1. Modificamos archivo README.md remoto

En GitHub vamos a modificar el archivo `README.md` y registrar el cambio (commit).

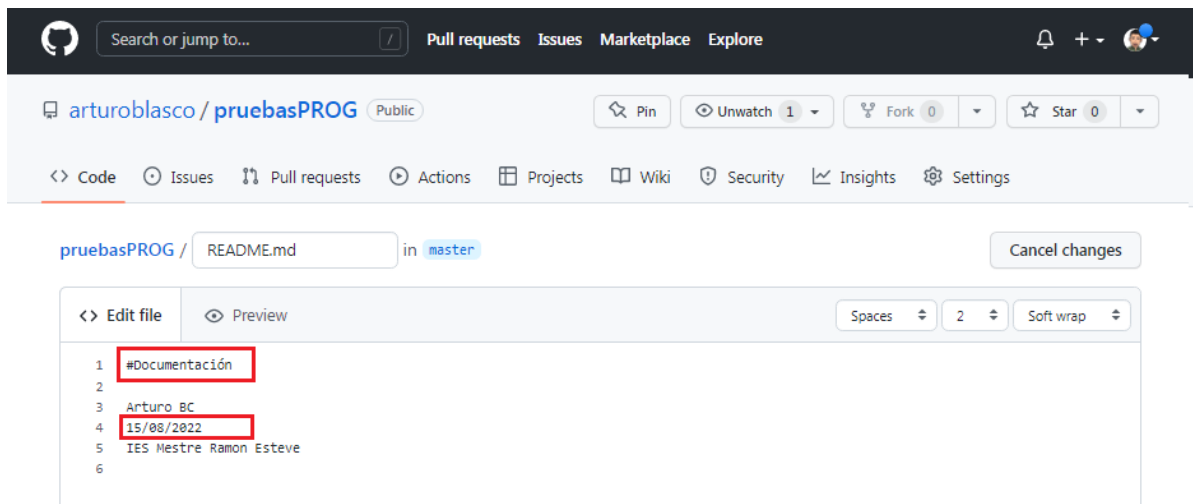
Para ello, entramos en nuestro repositorio remoto, pulsamos sobre el archivo `README.md` y luego pulsamos sobre el lápiz para editar.



Recientemente (mediados de agosto de 2021) gitHub añadió una funcionalidad interesante a todos sus repositorios, y es la posibilidad de abrir el editor vsCode online para cualquier repositorio simplemente usando la **hotkey** `."`.

Por tanto podemos hacer esta modificación tal y como se muestra en las capturas, o pulsar la tecla `."` (punto) y usar vsCode Online para hacer la modificación.

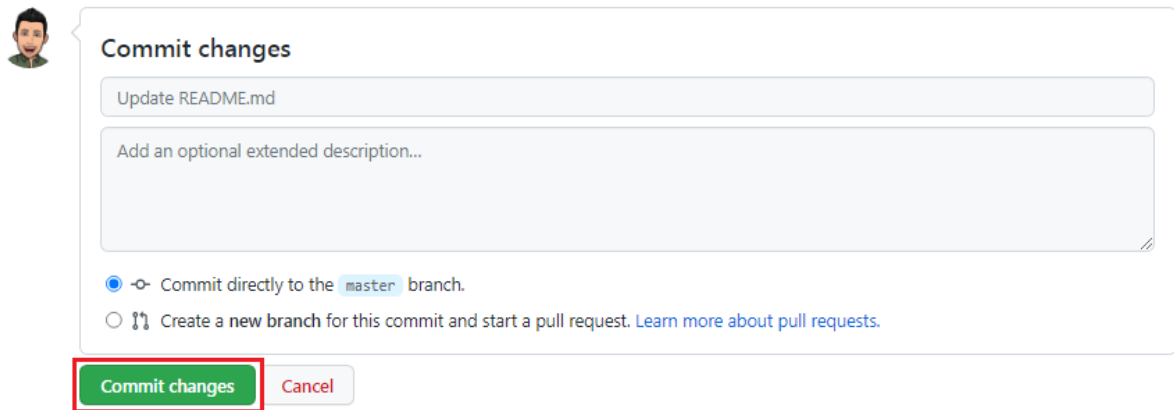
Insertamos una primera línea con título # y modificamos la línea de la fecha.



Registramos commit. Para ello pulsamos en **Commit changes**

Si lo deseamos, podemos poner un mensaje al commit y un descripción, aunque no es obligatorio.

GitHub pone una por defecto.



10.2. Modificamos archivo README.md local

En nuestro repositorio local, también vamos a modificar el archivo README.md.

En este caso añadiremos una línea al final del archivo y modificaremos la línea de la fecha.

```
1 | nano README.md
```

```
Arturo BC
25/04/2022
IES Mestre Ramon Esteve
Fin de documentación
```

Guardamos los cambios y registramos commit.

```
1 | git add README.md
2 | git commit -m "Actualización de README.md"
```

```

abc@ABC MINGW64 /d/5IES/1DAW-PROGRAMACIO/pruebas-arturo (master)
$ nano README.md

abc@ABC MINGW64 /d/5IES/1DAW-PROGRAMACIO/pruebas-arturo (master)
$ git add README.md

abc@ABC MINGW64 /d/5IES/1DAW-PROGRAMACIO/pruebas-arturo (master)
$ git commit -m "actualizacion README.md"
[master d5c9b0e] actualizacion README.md
1 file changed, 3 insertions(+), 1 deletion(-)

```

10.3. Intentamos subir el commit local

Al intentar subir nuestro commit local al repositorio remoto, se rechazará.

```
1 | git push
```

```

~/Proyectos/pruebas-jose > master ↑1
git push
To github.com:jamj2000/pruebas-jose.git
! [rejected]        master -> master (fetch first)
error: fallo el push de algunas referencias a 'git@github.com:jamj2000/pruebas-jose.git'
ayuda: Actualizaciones fueron rechazadas porque el remoto contiene trabajo que
ayuda: no existe localmente. Esto es causado usualmente por otro repositorio
ayuda: realizando push a la misma ref. Quizás quiera integrar primero los cambios
ayuda: remotos (ej. 'git pull ...') antes de volver a hacer push.
ayuda: Vea 'Notes about fast-forwards' en 'git push --help' para detalles.

```

Esto no es un conflicto.

Simplemente nos dice que debemos actualizar antes nuestro repositorio local con el contenido del repositorio remoto.

Si hemos realizado cambios en nuestro repositorio remoto, deberemos integrarlos en nuestro repositorio local antes de poder subir nuevos cambios locales.

10.4. Se produce conflicto

Así que ejecutamos:

```
1 | git pull origin master
```

para **bajar los commits del repositorio remoto** que no tenemos en local.

Esto no tendría por que provocar un conflicto.

Pero en este caso sí se produce, porque hemos modificado el mismo archivo (README.md) y además en la misma linea (la línea de la fecha).

Así que se realiza la fusión, pero nos avisa que hay conflicto en dicho archivo. Deberemos resolverlo manualmente.

```

$ git pull origin master
Enter passphrase for key '/c/Users/abc/.ssh/id_ed25519':
From github.com:arturoblasco/pruebasPROG
* branch          master      -> FETCH_HEAD
Auto-merging README.md
CONFLICT (content): Merge conflict in README.md
Automatic merge failed; fix conflicts and then commit the result.

```

10.5. Arreglamos conflicto

Para arreglar el conflicto, abrimos el archivo en cuestión y en la línea o líneas donde se ha producido el conflicto veremos unas marcas como las siguientes:

```
<<<<<<<
```

```
1 | <<<<<<<
2 | línea o líneas en commit local
3 | =====
4 | línea o líneas en commit remoto
5 | >>>>>>>
```

```
#Documentación
Arturo BC
<<<<<<< HEAD
25/04/2022
=====
15/08/2022
>>>>>>> 19eae3f832c48f6c37386c31e0ae0bdc000e6b1f
IES Mestre Ramon Esteve
Fin de documentación
```

Resolver el conflicto consiste en elegir una de las 2 opciones y eliminar las marcas anteriores.

Aunque también podemos no elegir ninguna de las opciones y escribir otra en su lugar. Esto es lo que yo he hecho aquí al poner fecha *11 agosto 2022*.

```
#Documentación
Arturo BC
11 Agosto 2022|
IES Mestre Ramon Esteve
Fin de documentación
```

A continuación, guardamos los cambios. Y registramos un nuevo commit.

```
1 | git add README.md
2 | git commit -m "Arreglado conflicto en README.md"
```

```
abc@ABC MINGW64 /d/5IES/1DAW-PROGRAMACIO/pruebas-arturo (master|MERGING)
$ git add README.md

abc@ABC MINGW64 /d/5IES/1DAW-PROGRAMACIO/pruebas-arturo (master|MERGING)
$ git commit -m "arreglado conflicto README.md"
[master c8d73b2] arreglado conflicto README.md
```

Ahora ya podremos subir nuestro commit con el conflicto solucionado.

```
git push origin master
```

```
$ git push origin master
Enter passphrase for key '/c/Users/abc/.ssh/id_ed25519':
Enumerating objects: 10, done.
Counting objects: 100% (10/10), done.
Delta compression using up to 4 threads
Compressing objects: 100% (6/6), done.
Writing objects: 100% (6/6), 671 bytes | 671.00 KiB/s, done.
Total 6 (delta 2), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (2/2), completed with 1 local object.
To github.com:arturoblasco/pruebasPROG.git
   19eae3f..c8d73b2  master -> master
```

NOTA: Para evitar situaciones como la anterior, es aconsejable **no realizar modificaciones en GitHub**, y si las hemos realizado o hemos subido commits desde otro repositorio local, lo primero que deberíamos hacer es `git pull`, resolver los conflictos que puedan darse, realizar los commits locales que deseemos y finalmente subir commits a GitHub.

Resumiendo, una buena estrategia puede ser la siguiente: al principio del día haremos `git pull`, y al final del día haremos `git push`.

NOTA: No borrar los repositorio local ni repositorio remoto. Los volveremos a utilizar en la siguiente actividad.

Subir a la plataforma **AULES** un documento PDF de nombre **actividad09tunombre** con las capturas de pantalla y explicaciones pertinentes.

11. Creación de ramas

En esta actividad vamos a empezar a trabajar con ramas. En concreto veremos cómo **crear nuevas ramas**.

Podemos definir una rama como un **desarrollo paralelo dentro del mismo repositorio**. Podemos iniciar dicho desarrollo paralelo en cualquier commit.

En esencia, las principales finalidades de las ramas son 2:

- **hacer cambios en el repositorio sin afectar a la rama master.** También aplicable a otras ramas.
- **hacer cambios en el repositorio e integrarlos posteriormente en la rama master.** También aplicable a otras ramas.

Por defecto cada repositorio de git dispone de una **rama master**. Ésta es la rama principal. Por motivos de seguridad, suele ser frecuente realizar los cambios en alguna otra rama y posteriormente integrarlos en la rama master.

Existen **flujos de trabajo** ([workflows](#)) en los que apenas se crean commits en la rama master, sólo se integran commits de otras ramas.

En esta actividad usaremos 2 métodos para trabajar con nuevas ramas:

- `git checkout -b nueva-rama`
- `git branch nueva-rama`, y luego `git checkout nueva-rama`

Comprobemos antes, el estado actual de nuestro repositorio. Con `git log ...` podemos ver que sólo tenemos la rama master.

Para ello ejecutamos

```
1 | git log --oneline --all --graph
```

La opción `--graph` nos permite ver las ramas de forma "gráfica".

```
$ git log --oneline --all --graph
* c8d73b2 (HEAD -> master, origin/master) arreglado conflicto README.md
|
* 19eae3f Update README.md
* | d5c9b0e actualizacion README.md
|/
* e59fde1 (tag: v3) codigo fuente inicial
* 38428b1 (tag: v2) tercer cambio Arturo
* 15cb1e6 primer cambio Arturo
* 5e178c8 (tag: v1) primer cambio Arturo
```

Podemos ver también "otra rama" sin nombre con el commit `19ea Update README.md`. En realidad éste es el commit que editamos en *GitHub* en una actividad anterior y que tuvimos que fusionar en la rama local *master*, antes de volver a subirlo a *GitHub*.

11.1. Crear rama mediante `git checkout`

El comando `git checkout -b nueva-rama` tiene esencialmente 2 formas:

1. `git checkout -b nueva-rama` (creamos una nueva rama a partir del commit actual, y nos pasamos a ella).
2. `git checkout -b nueva-rama commit-de-partida` (creamos una nueva rama a partir del commit indicado, y nos pasamos a ella).

En este apartado vamos a crear 2 ramas (las llamaremos `rama1` y `rama2`) a partir del primer commit, es decir el commit más antiguo, que tenemos etiquetado como `v1`.

Para crear `rama1` y movernos a ella, vamos a usar la forma más directa. Para ello hacemos:

```
1 | git checkout -b rama1 v1
```

En dicha `rama1`, creamos un nuevo commit:

```
abc@ABC MINGW64 /d/5IES/1DAW-PROGRAMACIO/pruebas-arturo (rama1)
$ nano rama1.txt

abc@ABC MINGW64 /d/5IES/1DAW-PROGRAMACIO/pruebas-arturo (rama1)
$ cat rama1.txt
Rama 1

abc@ABC MINGW64 /d/5IES/1DAW-PROGRAMACIO/pruebas-arturo (rama1)
$ git add .
warning: LF will be replaced by CRLF in rama1.txt.
The file will have its original line endings in your working directory

abc@ABC MINGW64 /d/5IES/1DAW-PROGRAMACIO/pruebas-arturo (rama1)
$ git commit -m "nuevo archivo rama1.txt"
[rama1 2ba620a] nuevo archivo rama1.txt
1 file changed, 1 insertion(+)
create mode 100644 rama1.txt
```

El resultado es:

```
$ git log --oneline --all --graph
* 2ba620a (HEAD -> rama1) nuevo archivo rama1.txt
| * c8d73b2 (origin/master, master) arreglado conflicto README.md
|/
| * 19eae3f Update README.md
| * | d5c9b0e actualizacion README.md
|/
* e59fde1 (tag: v3) codigo fuente inicial
* 38428b1 (tag: v2) tercer cambio Arturo
* 15cble6 primer cambio Arturo
|/
* 5e178c8 (tag: v1) primer cambio Arturo
```

Ahora hagamos otra rama llamada `rama2` a partir del commit `v1`, de una forma un poco distinta.

Imaginemos que, por despiste, nos hemos movido al commit `v1` con:

```
git checkout v1
```

```
$ git checkout v1
Note: switching to 'v1'.

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 switching back to a branch.

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

git switch -c <new-branch-name>

Or undo this operation with:

git switch -

Turn off this advice by setting config variable advice.detachedHead to false

HEAD is now at 5e178c8 primer cambio Arturo
```

Como se nos informa en el mensaje, ahora mismo estamos trabajando en modo *despegado* (detached HEAD). Esto nos permite realizar los cambios que deseemos creando commits sin afectar a la rama master.

Lo aconsejable es ejecutar ahora el comando `git checkout -b rama2`, porque después se nos podría olvidar, y al cambiar de rama perderíamos los commits realizados.

No obstante, vamos a simular que se nos olvida ejecutar el comando anterior. Empezamos a realizar commits. En este caso para simplificar, solo realizaremos un commit.

```
abc@ABC MINGW64 /d/5IES/1DAW-PROGRAMACIO/pruebas-arturo ((v1))
$ nano rama2.txt

abc@ABC MINGW64 /d/5IES/1DAW-PROGRAMACIO/pruebas-arturo ((v1))
$ cat rama2.txt
Rama 2

abc@ABC MINGW64 /d/5IES/1DAW-PROGRAMACIO/pruebas-arturo ((v1))
$ git add rama2.txt
warning: LF will be replaced by CRLF in rama2.txt.
The file will have its original line endings in your working directory

abc@ABC MINGW64 /d/5IES/1DAW-PROGRAMACIO/pruebas-arturo ((v1))
$ git commit -m "nuevo archivo rama2.txt"
[detached HEAD f4564be] nuevo archivo rama2.txt
1 file changed, 1 insertion(+)
create mode 100644 rama2.txt
```

Esto nos crea un nuevo commit. Ejecutamos una vez más

```
1 | git log --oneline --all --graph
```

```
$ git log --oneline --all --graph
* f4564be (HEAD) nuevo archivo rama2.txt
* 2ba620a (rama1) nuevo archivo rama1.txt
/
* c8d73b2 (origin/master, master) arreglado conflicto README.md
| \
| * 19eae3f Update README.md
* | d5c9b0e actualizacion README.md
| /
* e59fde1 (tag: v3) codigo fuente inicial
* 38428b1 (tag: v2) tercer cambio Arturo
* 15cb1e6 primer cambio Arturo
/
* 5e178c8 (tag: v1) primer cambio Arturo
```

Como se muestra en la captura, no existe ningún apuntador en forma de rama, así que si ahora, por ejemplo, ejecutásemos `git checkout master`, perderíamos todos los commits realizados (en este caso sólo uno, pero podrían ser muchos más).

Si no deseamos perder dichos commits, debemos ejecutar:

```
git checkout -b rama2
```

```
$ git log --oneline --all --graph
* f4564be (HEAD -> rama2) nuevo archivo rama2.txt
* 2ba620a (rama1) nuevo archivo rama1.txt
/
* c8d73b2 (origin/master, master) arreglado conflicto README.md
| \
| * 19eae3f Update README.md
* | d5c9b0e actualizacion README.md
| /
* e59fde1 (tag: v3) codigo fuente inicial
* 38428b1 (tag: v2) tercer cambio Arturo
* 15cb1e6 primer cambio Arturo
/
* 5e178c8 (tag: v1) primer cambio Arturo
```

Después de esto, ya podremos cambiar de rama con `git checkout` sin miedo a perder los commits realizados anteriormente.

Asegúrate de ejecutar el comando anterior antes de pasar al punto siguiente.

11.2. Crear ramas con `git branch` ...

El comando `git branch nueva-rama` tiene esencialmente 2 formas:

1. `git branch nueva-rama` (Creamos una nueva rama a partir del commit actual, pero NO nos pasamos a ella).
2. `git branch nueva-rama commit-de-partida` (Creamos una nueva rama a partir del commit indicado, pero NO nos pasamos a ella).

Después de ejecutar una de las formas anteriores, siempre deberemos hacer después un `git checkout` si queremos trabajar con la nueva rama.

Vamos a ver su uso, haciendo uso de la segunda forma. Desde la rama actual, es decir rama2, vamos a crear 2 ramas (llamadas licencia y autor) a partir de la rama master.

```
1 | git branch licencia master
2 | git branch autor master
```

```
abc@ABC MINGW64 /d/5IES/1DAW-PROGRAMACIO/pruebas-arturo (master)
$ git branch licencia master

abc@ABC MINGW64 /d/5IES/1DAW-PROGRAMACIO/pruebas-arturo (master)
$ git branch autor master

abc@ABC MINGW64 /d/5IES/1DAW-PROGRAMACIO/pruebas-arturo (master)
$ git checkout licencia
Switched to branch 'licencia'
```

Para empezar a trabajar con alguna de ellas, deberemos ejecutar `git checkout ...`. Por ejemplo:

```
1 | git checkout licencia
```

Con `git log --oneline --all --graph` podemos ver que el apuntador `HEAD` ahora apunta a la rama `licencia`.

```
$ git log --oneline --all --graph
* f4564be (rama2) nuevo archivo rama2.txt
| * 2ba620a (rama1) nuevo archivo ramal.txt
|/
| * c8d73b2 (HEAD -> licencia, origin/master, master, autor) arreglado conf
| licto README.md
| | \
| | * 19eae3f Update README.md
| | * | d5c9b0e actualizacion README.md
| | /
| | * e59fde1 (tag: v3) codigo fuente inicial
| | * 38428b1 (tag: v2) tercer cambio Arturo
| | * 15cble6 primer cambio Arturo
|/
* 5e178c8 (tag: v1) primer cambio Arturo
```

En esta rama crearemos un archivo nuevo llamado `LICENSE`.

Para ello:

```
1 | nano LICENSE
```

Escribimos dentro un línea con el texto siguiente: **GPL v3**

Y realizamos commit:

```
1 | git add LICENSE
2 | git commit -m "Nuevo archivo LICENSE"
```

Para trabajar con la rama `autor`, ejecutamos:

```
1 | git checkout autor
```

En esta rama vamos a crear un archivo `AUTHOR` y además vamos a modificar el archivo `README.md`.

Para ello:

```
1 | nano AUTHOR
```

Escribimos dentro un línea con el texto de vuestro nombre: **Arturo BC**

También modificaremos el archivo `README.md`.

En la línea donde aparece nuestro nombre, cambiaremos el texto para ponerlo todo en mayúsculas.

La finalidad es provocar un conflicto de fusión en un futuro, que resolveremos en la siguiente actividad.

Y realizamos commit:

```
1 | git add AUTHOR
2 | git commit -m "Nuevo archivo AUTHOR y editado README.md"
```

El resultado de `git log --oneline --all --graph` es:

```
$ git log --oneline --all --graph
* c1aeed3 (HEAD -> autor) nuevo archivo AUTHOR y edicion README.md
| * 48f5cbe (licencia) nuevo archivo LICENSE
|/
* c8d73b2 (origin/master, master) arreglado conflicto README.md
| \
| * 19eae3f Update README.md
| * | d5c9b0e actualizacion README.md
|/
* e59fde1 (tag: v3) codigo fuente inicial
* 38428b1 (tag: v2) tercer cambio Arturo
* 15cb1e6 primer cambio Arturo
| * f4564be (rama2) nuevo archivo rama2.txt
|/
| * 2ba620a (rama1) nuevo archivo rama1.txt
|/
* 5e178c8 (tag: v1) primer cambio Arturo
```

11.3. Subir ramas a repositorio remoto

Para subir todos los cambios realizados en todas las ramas:

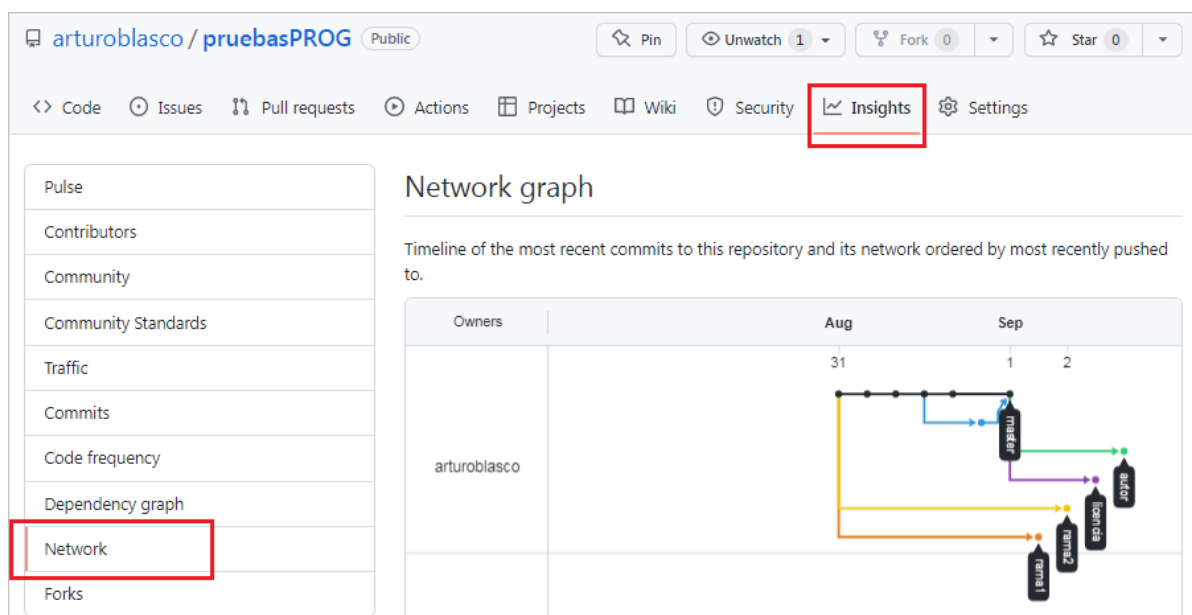
```
git push origin --all
```

```
$ git push origin --all
Enter passphrase for key '/c/Users/abc/.ssh/id_ed25519':
Enumerating objects: 17, done.
Counting objects: 100% (17/17), done.
Delta compression using up to 4 threads
Compressing objects: 100% (9/9), done.
Writing objects: 100% (13/13), 1.13 KiB | 387.00 KiB/s, done.
Total 13 (delta 2), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (2/2), done.
To github.com:arturoblasco/pruebasPROG.git
 * [new branch]      autor -> autor
 * [new branch]      licencia -> licencia
 * [new branch]      rama1 -> rama1
 * [new branch]      rama2 -> rama2
```

El resultado es que todos los apuntadores a ramas remotas se actualizan (aparecen en color rojo en la siguiente captura)

```
$ git log --oneline --all --graph
* c1aeed3 (HEAD -> autor, origin/autor) nuevo archivo AUTHOR y edicion README.md
|
* 48f5cbe (origin/licencia, licencia) nuevo archivo LICENSE
|
* c8d73b2 (origin/master, master) arreglado conflicto README.md
|
* 19eae3f Update README.md
* | d5c9b0e actualizacion README.md
|
* e59fde1 (tag: v3) codigo fuente inicial
* 38428b1 (tag: v2) tercer cambio Arturo
* 15cb1e6 primer cambio Arturo
|
* f4564be (origin/rama2, rama2) nuevo archivo rama2.txt
|
* 2ba620a (origin/rama1, rama1) nuevo archivo rama1.txt
|
* 5e178c8 (tag: v1) primer cambio Arturo
```

En **GitHub**, dentro del repositorio correspondiente, podemos ver un gráfico de las ramas pulsando en la pestaña **Insights** y luego en la opción **Network** (en la parte izquierda de la nueva página)



NOTA: No borrar los repositorio local ni repositorio remoto. Los volveremos a utilizar en la siguiente actividad.

Subir a la plataforma **AULES** un documento PDF de nombre **actividad10tunombre** con las capturas de pantalla y explicaciones pertinentes.

12. Fusión y eliminación de ramas

Esta actividad es una continuación de la anterior. En ella veremos cómo realizar **fusión de ramas (merge)** y como eliminar apuntadores a ramas antiguas.

Vamos a suponer que hemos trabajado en las ramas de la actividad anterior `rama1`, `rama2`, `licencia` y `autor` añadiendo varios commits más, aunque realmente no ha sido así (las ramas con un único commit no suelen ser tan frecuentes).

Y llega el momento de desechar el trabajo realizado en alguna rama e integrar el contenido de otras en la rama `master`.

En esta actividad desecharemos el trabajo realizado en `rama2`, e integraremos en `master` las ramas `rama1`, `licencia` y `autor`.

Para realizar fusión (merge) de ramas se utiliza el comando:

```
git merge ...
```

12.1. Eliminando una rama local

Para eliminar una rama local se usa el comando:

```
1 | git branch -d rama
```

Por ejemplo, para borrar `rama2` hacemos

```
1 | git branch -d rama2
```

No se ejecuta la eliminación, puesto que los cambios no han sido integrados en `master`, ni en ninguna otra rama.

Para forzar la eliminación hacemos

```
1 | git branch -D rama2
```

De esta manera perdemos todas las modificaciones que hubiésemos realizado en dicha rama.

```
abc@ABC MINGW64 /d/5IES/1DAW-PROGRAMACIO/pruebas-arturo (autor)
$ git branch -d rama2
error: The branch 'rama2' is not fully merged.
If you are sure you want to delete it, run 'git branch -D rama2'.

abc@ABC MINGW64 /d/5IES/1DAW-PROGRAMACIO/pruebas-arturo (autor)
$ git branch -D rama2
Deleted branch rama2 (was f4564be).
```

12.2. Fusionando ramas locales

Vamos a integrar en la rama `master` los cambios realizados en `rama1`, `licencia` y `autor`.

Procederemos de la siguiente forma:

1. Cambiamos a rama `master`
2. Fusionamos rama `licencia`
3. Fusionamos rama `autor`
4. Fusionamos rama `rama1`

12.2.1. Cambiamos a rama `master`

Es **MUY IMPORTANTE** cambiar a la rama `master`. Si no hacemos el cambio, todas las fusiones se realizarían sobre la rama `autor` (la rama en la que actualmente estamos).

Debemos hacer

```
1 | git checkout master
```

12.2.2. Fusionamos rama `licencia`

Antes, fijémonos en la estructura de las ramas. Hacemos

```
1 | git log --oneline --all --graph
```

```
$ git log --oneline --all --graph
* c1aeed3 (origin/autor, autor) nuevo archivo AUTHOR y edicion README.md
| * 48f5cbe (origin/licencia, licencia) nuevo archivo LICENSE
|/
* c8d73b2 (HEAD -> master, origin/master) arreglado conflicto README.md
| \
| * 19eae3f Update README.md
* | d5c9b0e actualizacion README.md
|/
* e59fde1 (tag: v3) codigo fuente inicial
* 38428b1 (tag: v2) tercer cambio Arturo
* 15cble6 primer cambio Arturo
| * f4564be (origin/rama2) nuevo archivo rama2.txt
|/
| * 2ba620a (origin/rama1, rama1) nuevo archivo rama1.txt
|/
* 5e178c8 (tag: v1) primer cambio Arturo
```

Observa que fusionar la rama `licencia` en la rama `master` es equivalente a mover los apuntadores `HEAD` y `master` hacia arriba, es decir, hacerlos coincidir con el apuntador `licencia`.

Este tipo de fusión es el más sencillo y nunca da conflictos. Se conoce como **fast-forward merge** (abreviado **FF**) o **fusión con avance rápido**.

Para fusionar esta rama hacemos

```
1 | git merge licencia
```

```
$ git merge licencia
Updating c8d73b2..48f5cbe
Fast-forward
 LICENSE | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 LICENSE
```

Observa como queda tras la fusión. Únicamente se han movido los apuntadores `HEAD` y `master`.

```
$ git log --oneline --all --graph
* c1aeed3 (origin/autor, autor) nuevo archivo AUTHOR y edicion README.md
| * 48f5cbe (HEAD -> master, origin/licencia, licencia) nuevo archivo LICENSE
|/
* c8d73b2 (origin/master) arreglado conflicto README.md
| \
| * 19eae3f Update README.md
* | d5c9b0e actualizacion README.md
|/
* e59fde1 (tag: v3) codigo fuente inicial
* 38428b1 (tag: v2) tercer cambio Arturo
* 15cb1e6 primer cambio Arturo
| * f4564be (origin/rama2) nuevo archivo rama2.txt
|/
| * 2ba620a (origin/rama1, rama1) nuevo archivo rama1.txt
|/
* 5e178c8 (tag: v1) primer cambio Arturo
```

NOTA: No te preocupes ahora mismo por los apuntadores remotos (los que aparecen en color rojo). Más adelante los sincronizaremos con el repositorio remoto.

12.2.3. Fusionamos rama autor

Si en lugar de fusionar una rama que está adelantada respecto a `master`, lo que hacemos es fusionar una rama que está en paralelo con la rama `master`, entonces realizaremos una **fusión de 3 vías (3-way merge)**

Este tipo de fusión puede provocar conflictos. Si ambas ramas contienen modificaciones en las mismas líneas en los mismos archivos puede producirse un conflicto.

En este caso, el archivo `README.md` posee una línea con el nombre del autor, pero con líneas distintas en las ramas `master` y `autor` (todo en mayúsculas).

Para realizar la fusión:

```
1 | git merge autor
```

Cuando aparezca el editor con el mensaje asociado, aceptaremos el mensaje o lo editaremos a nuestro gusto.

```
$ git merge autor
MSG to Unix format...
Merge made by the 'recursive' strategy.
AUTHOR      | 1 +
README.md   | 2 +-
2 files changed, 2 insertions(+), 1 deletion(-)
create mode 100644 AUTHOR
```

En este caso no llegó a producirse el conflicto. Se resolvió automáticamente a favor del contenido de la rama `autor`.

Por tanto el autor en el archivo `README.md` aparecerá todo en mayúsculas.

Fíjate como se ha creado un nuevo commit resultado de unir la rama `autor` y la rama `master`.

Esto siempre sucede en la fusión de 3 vías.

```
$ git log --oneline --all --graph
* 79adc57 (HEAD -> master) Merge branch 'autor'
| \
| * c1aeed3 (origin/autor, autor) nuevo archivo AUTHOR y edicion README.md
* | 48f5cbe (origin/licencia, licencia) nuevo archivo LICENSE
| /
* c8d73b2 (origin/master) arreglado conflicto README.md
| \
| * 19eae3f Update README.md
* | d5c9b0e actualizacion README.md
| /
* e59fde1 (tag: v3) codigo fuente inicial
* 38428b1 (tag: v2) tercer cambio Arturo
* 15cble6 primer cambio Arturo
| * f4564be (origin/rama2) nuevo archivo rama2.txt
| /
| * 2ba620a (origin/rama1, rama1) nuevo archivo rama1.txt
| /
* 5e178c8 (tag: v1) primer cambio Arturo
```

12.2.4. Fusionamos rama rama1

Por último, integraremos en master los cambios realizados en la `rama1`.

Es un tipo de **fusión de 3 vías**, al igual que el anterior.

En este caso, no se producirá ningún conflicto, puesto que en esta rama sólo hemos realizado cambios sobre el archivo `rama1.txt`, el cual no existe en la rama `master`.

Para realizar la fusión:

```
1 | git merge rama1
```

Cuando aparezca el editor con el mensaje asociado, aceptaremos el mensaje o lo editaremos a nuestro gusto.

```
$ git merge rama1
MSG to Unix format...
Merge made by the 'recursive' strategy.
 rama1.txt | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 rama1.txt
```

12.3. Subiendo cambios a repositorio remoto

Para subir al repositorio remoto todos los cambios realizados en nuestro repositorio local, ejecutamos

```
1 | git push origin --all
```

```
$ git push origin --all
Enumerating objects: 7, done.
Counting objects: 100% (7/7), done.
Delta compression using up to 4 threads
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 537 bytes | 537.00 KiB/s, done.
Total 4 (delta 2), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (2/2), completed with 1 local object.
To github.com:arturoblasco/pruebasPROG.git
 c8d73b2..721ecfc master -> master
```

12.4. Eliminando apuntadores a ramas locales

Para eliminar los apuntadores locales:

```
1 | git branch -d rama1
```

Los apuntadores a `licencia` y `autor` no los eliminaremos, por si en el futuro deseamos seguir trabajando en dichas ramas.

```
$ git branch -d rama1
Deleted branch rama1 (was 2ba620a).
```

12.5. Eliminando apuntadores a ramas remotas

Para eliminar los apuntadores en el repositorio remoto:

```
1 | git push origin --delete rama1
2 | git push origin --delete rama2
```

```
$ git push origin --delete rama1
To github.com:arturoblasco/pruebasPROG.git
- [deleted]          rama1

abc@ABC MINGW64 /d/5IES/1DAW-PROGRAMACIO/pruebas-arturo (master)
$ git push origin --delete rama2
To github.com:arturoblasco/pruebasPROG.git
- [deleted]          rama2
```

Los apuntadores a `origin/licencia` y `origin/autor` no los eliminaremos, por si en el futuro deseamos seguir trabajando en dichas ramas.

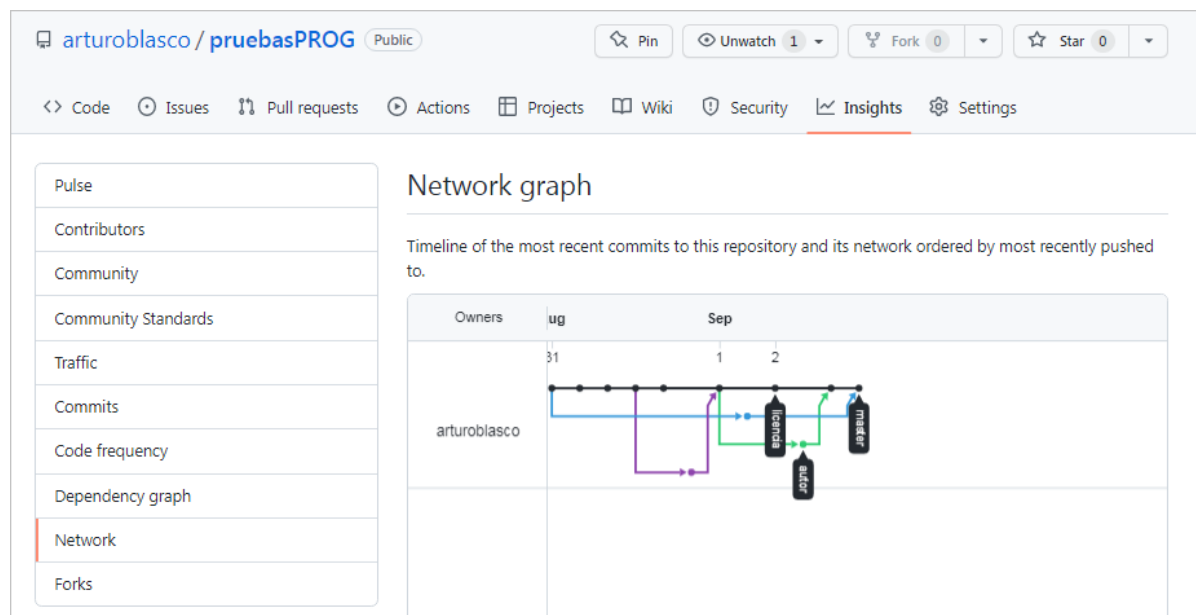
Para ver el estado ejecutamos `git log ...`

Observa como las ramas están actualizadas y sincronizadas con el repositorio remoto.

```
$ git log --oneline --all --graph
* 721ecfc (HEAD -> master, origin/master) Merge branch 'rama1'
| \
| * 2ba620a nuevo archivo rama1.txt
| * | 79adc57 Merge branch 'autor'
| \
| * | c1aeed3 (origin/autor, autor) nuevo archivo AUTHOR y edicion README.md
| * | 48f5cbe (origin/licencia, licencia) nuevo archivo LICENSE
| \
| * | c8d73b2 arreglado conflicto README.md
| \
| * | 19eae3f Update README.md
| * | d5c9b0e actualizacion README.md
| \
| * | e59fde1 (tag: v3) codigo fuente inicial
| * | 38428b1 (tag: v2) tercer cambio Arturo
| * | 15cble6 primer cambio Arturo
| \
| * 5e178c8 (tag: v1) primer cambio Arturo
```

12.6. Comprobando cambios en repositorio remoto

Para ver un gráfico de las ramas en el repositorio remoto pulsamos en **Insights, Network**.



12.7. Tarea propuesta para el alumno/a

Como tarea, se propone

- volver a la rama `licencia`, añadir contenido al archivo `LICENSE` y hacer commit.
- volver a la rama `autor`, añadir contenido al archivo `AUTHOR` y hacer commit.
- integrar los cambios de ambas ramas en la rama `master`.

NOTA: No borrar los repositorio local ni repositorio remoto. Los volveremos a utilizar en la siguiente actividad.

Subir a la plataforma **AULES** un documento PDF de nombre **actividad11tunombre** con las capturas de pantalla y explicaciones pertinentes.