I.E.S. DOÑANA

GIT & GITHUB

Entornos de desarrollo

1º Desarrollo de aplicaciones web

Tabla de contenido

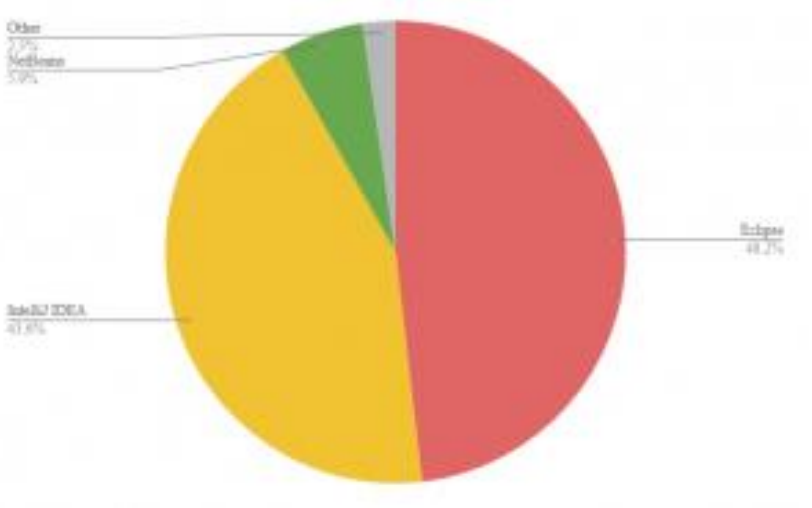
HERRAMIENTAS DE DESARROLLO 3
Entornos de Desarrollo Integrados (IDE) 3
Componentes de un IDE 3
Frameworks 4
Sistemas de Control de Versiones 5
Arquitectura Centralizada 5
Arquitectura Distribuida 6
GIT 7
Servidores de Repositorios 7
Funcionamiento de Git 7
Instalación de Git 9
Configuración inicial de Git 9
Manejo de Git 9
Comandos básicos de Git 10
Primer ejemplo 10
Trabajo en repositorio local 10
Modificación de archivos 14
Deshacer cambios en archivos modificados 15
Añadir archivos modificados 15
Obtener versiones anteriores 16
Añadir archivos a la última versión sin crear una nueva 18
Comparativa entre git reset y git checkout 18
Ignorar archivos en las versiones 18
Trabajar con ramas 19
Listado de ramas existentes 20
Crear una nueva rama o versión alternativa 20
Cambiar de rama 20
Volver a rama master 21
Fusionar ramas 21
Eliminar ramas 21
Etiquetar revisiones 21
Otros comandos 22
GIT Y GITHUB 23

Registro y creación de nuestro primer repositorio 23
Exportar a GitHub nuestro proyecto desde Git..... 24
Importar nuestro proyecto desde GitHub..... 26
Actualizar nuestro repositorio remoto desde nuestro repositorio local 27
Actualizar tu repositorio local al commit más nuevo de GitHub 27
Volver a la versión del repositorio remoto en nuestro repositorio local..... 27
Eliminar un repositorio en GitHub 27
Cómo colaborar en un proyecto en GitHub sin ser colaborador 28
Desconectar nuestro repositorio local del repositorio remoto 29
Invitar colaboradores a un repositorio personal..... 30
Eliminar un colaborador de un repositorio personal 30

HERRAMIENTAS DE DESARROLLO

Entornos de Desarrollo Integrados (IDE)

Un entorno de desarrollo integrado es una aplicación software que ayuda en la tarea de desarrollo de software integrando una serie de herramientas enfocadas en dicho fin. La cantidad de herramientas incluidas es configurable dependiendo de las características de las aplicaciones que se quieran crear.



Componentes de un IDE

Por lo general los entornos de desarrollo actuales ofrecen:

- **Editor de texto**: facilita la escritura organizando las instrucciones atendiendo a un formato, colores, permite la escritura automática, autocompletar, además de las clásicas funciones de buscar, reemplazar cortar, pegar, etc.
- **Compilador/Interprete**: normalmente los IDEs incluyen herramientas de compilación y ejecución propias, sin tener necesidad de un kit de herramientas de desarrollo como JDK.
- **Depurador**: Herramienta que permite la ejecución de una aplicación instrucción a instrucción, y nos ayuda a examinar las distintas situaciones y cambios que se produzcan en las variables del programa.
- **Asistente para GUI**: (GUI - Interfaz gráfica de usuario). Normalmente es una paleta de componentes que me permite arrastrar y soltar (drag and drop) componentes gráficos a un panel que corresponde a la ventana que vamos a crear.
- **Control de versiones**: permite almacenar cambios en nuestro código y compartirlo con otros desarrolladores.
- **Exportar programas**: permitirá exportar nuestra aplicación completa en algún formato de fichero.

Frameworks

Una aplicación informática requiere que su información esté estructurada de un modo que entienda para poder gestionarla. Cuando los desarrolladores queremos crear una aplicación usamos y generamos distinto tipo de información: código fuente, ficheros de configuración, ficheros binarios de datos, librerías, ficheros ejecutables, etc. A veces nos centramos únicamente en el tipo de tecnología que vamos a usar pero nos despreocupamos a la hora de crear infinidad de recursos propios, de ficheros y de código fuente para crear nuestra aplicación pero sin ningún orden u organización.

Un framework es un esquema (una estructura, un patrón) para el desarrollo de una aplicación. Se puede entender como una estructura conceptual y tecnológica que nos ayuda a través de un conjunto de herramientas, librerías, módulos, para crear aplicaciones de un tipo concreto. No están ligados a un lenguaje concreto, pero cuando más detallado es el framework, más necesidad tendrá de vincularse a un lenguaje concreto.

Puede ser algo muy sencillo como el patrón MVC, que indica que separemos las clases de nuestro programa en las capas de presentación (vista), los datos (modelo) y las operaciones (controlador). Pero también tenemos otros frameworks que llegan a definir los nombres de los ficheros, su estructura, las convenciones de programación, etc. También es posible que el framework defina una estructura para una aplicación completa, o bien sólo se centre en un aspecto de ella.

Ejemplos de frameworks:

- JUnit es un framework para el desarrollo de pruebas unitarias de software. Está formado por una serie de librerías.
- Java Server Pages es un framework para crear aplicaciones web con Java, principalmente la interfaz de usuario.
- Xamarin es un framework para crear aplicaciones móviles para Android, IOS en el lenguaje C#.
- Hibernate es un framework para Java para facilitar el trabajo con bases de datos relacionales desde la programación orientada a objetos.
- .NET es un framework con soporte para múltiples lenguajes de programación y está enfocado en la creación de aplicaciones para cualquier sistema Windows.

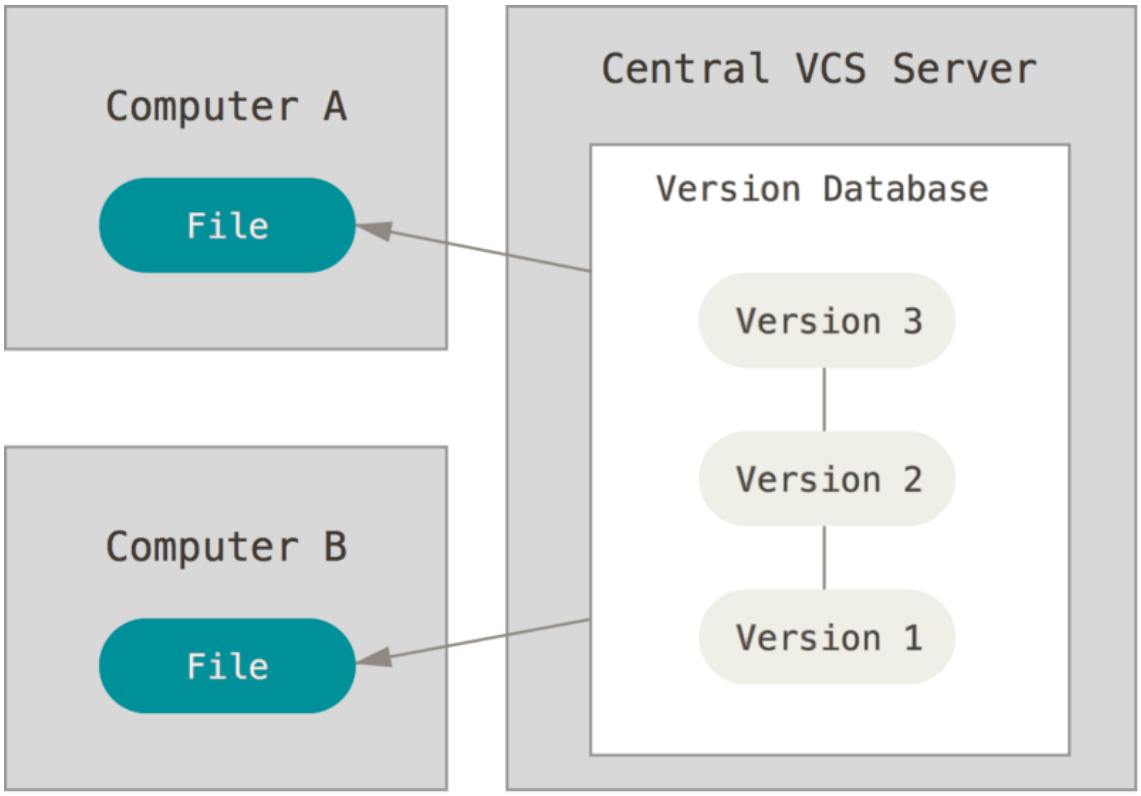
Sistemas de Control de Versiones

El control de versiones es un sistema que registra los cambios realizados sobre un archivo o conjunto de archivos a lo largo del tiempo, de modo que puedas recuperar versiones específicas más adelante.

Se usan para registrar cualquier tipo de archivos, y comparar cambios a lo largo del tiempo, revertir a un estado anterior del proyecto, ver en qué momento se introdujo un error, etc.

Tenemos dos arquitecturas principales a través de la que podemos clasificar distintos SCVs:

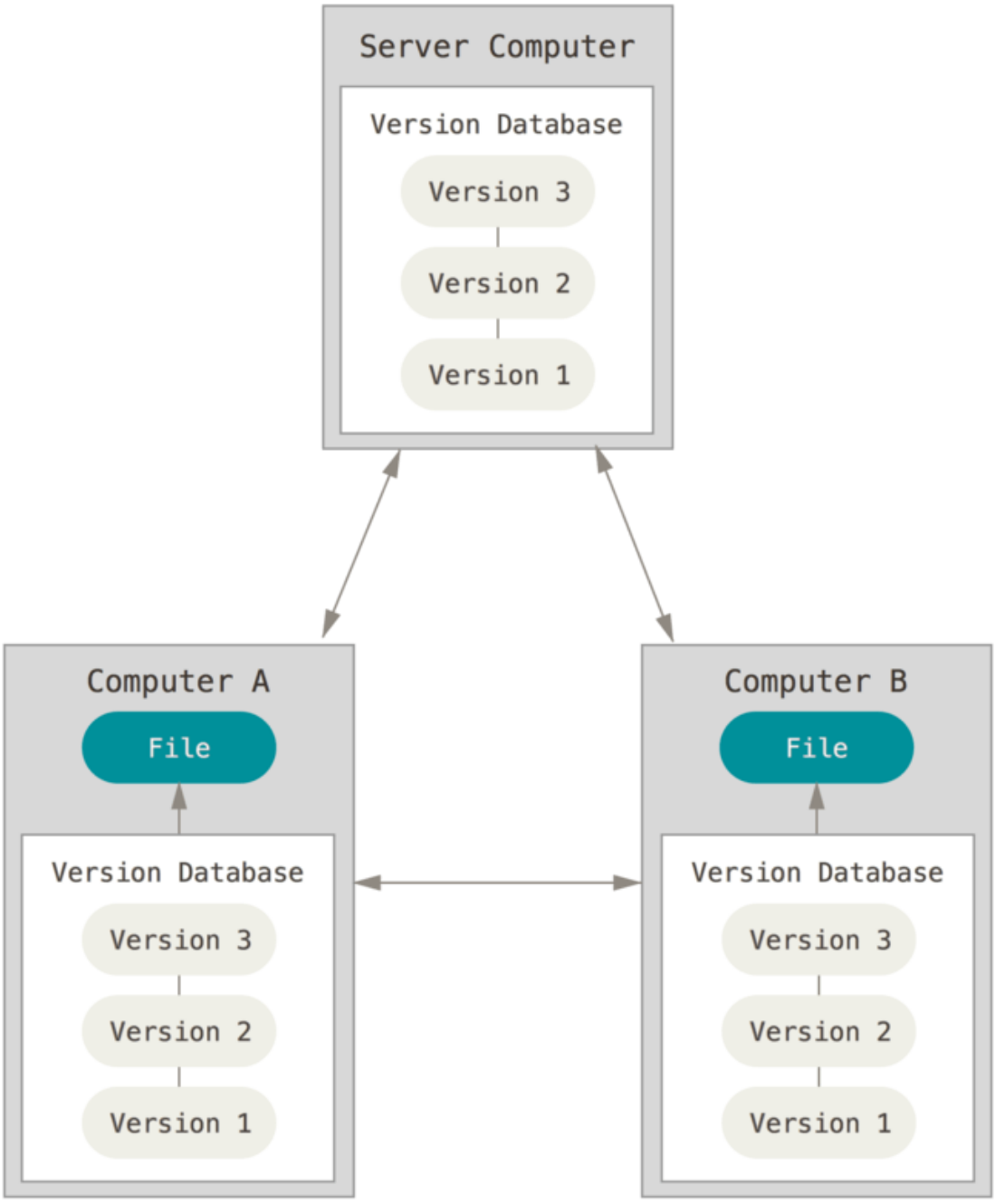
Arquitectura Centralizada



Los desarrolladores usan un repositorio central al que acceden mediante un cliente en su máquina. Tienen un único servidor que contiene todos los archivos versionados, y varios clientes que descargan los archivos desde ese lugar central. Durante muchos años éste ha sido el estándar para el control de versiones. La desventaja más obvia es el servidor centralizado como único punto de fallo. Si ese servidor se cae o pierde los datos, se pierde el trabajo.

- Subversion SVN (Código Abierto)
- Visual SourceSafe (Propietario)

Arquitectura Distribuida



En un SCV distribuido los clientes no sólo descargan la última instantánea de los archivos: suben completamente el repositorio. Cada vez que se descarga una instantánea (versión del proyecto), en realidad se hace una copia de seguridad completa de todos los datos. Así, si un servidor muere, cualquiera de los repositorios de los clientes puede copiarse en el servidor para restaurarlo.

- GIT (Código Abierto)
- Mercurial (Código Abierto)

GIT

Es un SCV (sistema de control de versiones) distribuido creado como necesidad durante el desarrollo de núcleo de Linux. El creador es Linus Torvalds, creador también del núcleo de Linux.

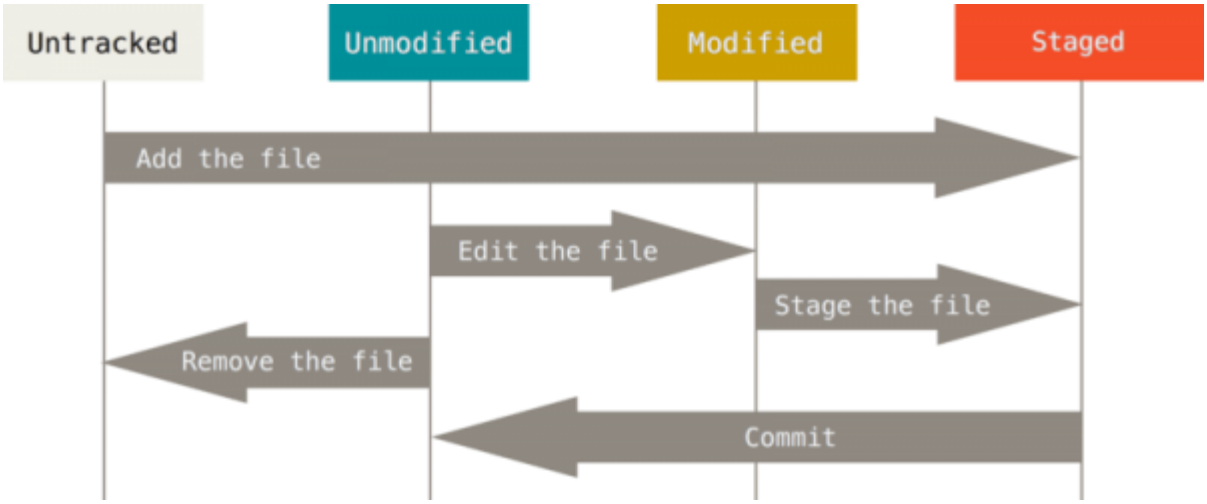
Servidores de Repositorios

Para almacenar las versiones de nuestro código necesitamos un servidor de control de versiones basado en Git. Actualmente hay distintas opciones que ofrecen servicios comerciales y también servicios gratuitos, con algunas limitaciones, pero perfectamente válidos para trabajar en nuestros pequeños proyectos. Las versiones gratuitas de los siguientes servicios ofrecen:

- GitHub: repositorios públicos ilimitados.
- BitBucket: repositorios públicos y privados ilimitados.
- GitLab: repositorios públicos y privados ilimitados.

Funcionamiento de Git

Git guarda una copia completa del estado de los ficheros de nuestro proyecto en cada versión (commit). Esta información se almacena en la base de datos local (repositorio local), y posteriormente se puede sincronizar con otros repositorios remotos.



En Git, los ficheros tienen 2 estados principales:

1. Ficheros fuera de seguimiento (*untracked*)
2. Ficheros bajo seguimiento (*tracked*)

- **Ficheros Untracked**. Cuando creo o añado ficheros nuevos a mi repositorio local estos están fuera de seguimiento. También están fuera de seguimiento los que no estaban bajo seguimiento en la última versión; último *commit* (confirmación).

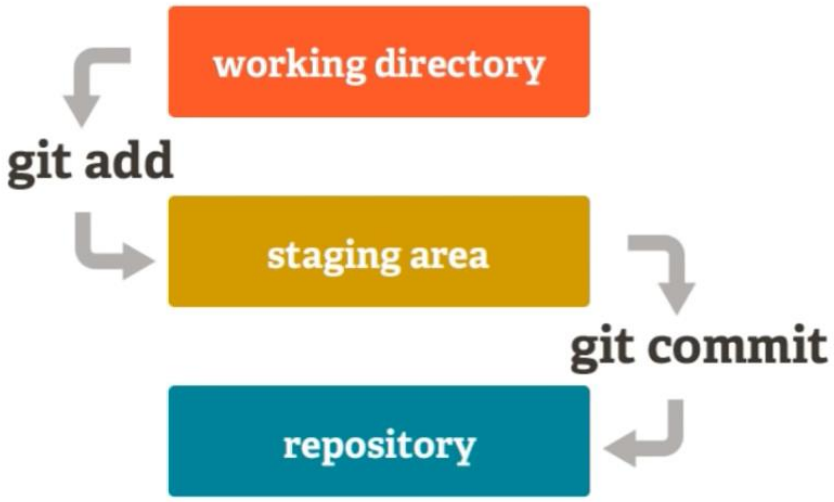
Los ficheros que están bajo seguimiento pueden tener 3 estados (*unmodified, staged, modified*)

- **Ficheros Staged (Preparados para confirmación):** Cuando añado ficheros *untracked* mediante el comando (git add) para que formen parte de la siguiente confirmación, estos pasan a estar pendientes de confirmación, preparados, (*staged*). Estos cambios no estarán confirmados hasta que haga *commit*.
- **Ficheros unmodified (Sin modificar)**: Justo después de hacer un *commit* (crear una versión), todos los ficheros que estaban pendientes de confirmación (*staged*), se almacenan en la base de datos local de Git. En ese momento todos los ficheros están sin ninguna modificación, que quiere decir que no hay modificaciones desde la última versión.
- **Ficheros modified (Modificados)**: Si modifico un fichero *unmodified*, este pasará a estado *modified* indicando que debo de volver a añadirlo (*git add*) parar prepararlo (*staged*) para la siguiente confirmación (*commit*).

Todo este trabajo se realiza en el repositorio local y se almacena en la base de datos local de Git. Independientemente de esto, puedo sincronizar el estado de mi repositorio local (directorio en mi equipo), con un repositorio remoto (fuera de mi equipo).

Las áreas o estados principales donde se pueden encontrar mis archivos en Git son los siguientes:

- **Working directory (directorio de trabajo):** que será donde trabajaremos con todos nuestros archivos
- **Staging area:** aquí se agregarán todos los archivos que vamos a preparar para el guardado de una versión, es decir, añadiremos (git add) archivos para que formen parte de la siguiente confirmación, estos pasan a estar pendientes de confirmación, preparados, (*staged*).
- **Repository:** Una vez nos hayamos decidido a guardar los cambios realizaremos un *commit* (crear una versión), todos los ficheros que estaban pendientes de confirmación (*staged*), se almacenarán en este repositorio.



Instalación de Git

En el siguiente video-tutorial podemos ver como instalar y configurar GIT.

https://www.youtube.com/watch?time_continue=3&v=IcHAKwakopQ&feature=emb_logo

Instalaremos la herramienta Git que usaremos como **cliente de nuestros repositorios** almacenados en BitBucket o GitHub.

1. https://git-scm.com/, descargamos el instalador y ejecutamos.
2. Libro https://git-scm.com/book/es/v2

Configuración inicial de Git

Al configurar GIT por **primera vez** (apartado 1.6), como mínimo debemos configurar nuestra identidad para realizar *commits* (Confirmar una versión). Esto se debe hacer una sola vez, después de instalar Git. Para ello abrimos un terminal y ejecutamos:

```
git config --global user.name "José Reyes Gil Rubio"
git config --global user.email josereyesgil@gmail.com

git config --list   #Nos permite ver todas las propiedades de configuración de Git.
```

Manejo de Git

Para entender completamente el funcionamiento de Git debemos leer los capítulos 2 y 3 del libro Pro Git Book. Podemos descargarlo desde su web oficial en inglés o leerlo online en español.

Git ofrece tanto integración en un terminal de Windows (cmd), como en un terminal de Linux (Bash). Adicionalmente, Git instala su propia consola basada en sistemas operativos Linux.

Los siguientes comandos están centrados en el manejo del terminal de Linux (bash):

Comandos Bash	Función
ls	Muestra el contenido del directorio actual
ls -a	Muestra el contenido del directorio actual con los fichero ocultos
pwd	Muestra la ruta del directorio en el que estoy
cd [directorio]	Me permita cambiar de directorio
directorios . y ..	Representan el directorio actual y su padre, respectivamente
rm [nombre_archivo]	Elimina un archivo
rm -r [nombre directorio]	Elimina un directorio con su contenido
touch [nombre_fichero]	Crea un fichero de texto plano con dicho nombre]
mkdir [nombre_directorio]	Crea un directorio vacío con dicho nombre]
clear	Limpia el terminal de comandos

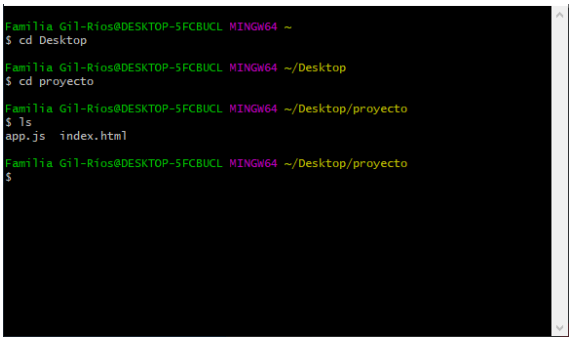
Comandos básicos de Git

- **git init**: indica a git que vamos a crear un proyecto nuevo o que vamos a "cargar" o indicar que vamos a trabajar con un proyecto ya existente.
- **git add <file>:** pasa los archivos del working directory al staging area.
- **git status:** se usa para ver en qué estado están tus archivos, si están en el working directory, en el staging area,…
- **git commit:** pasa del staging area al repository, es decir, para crear un snapshot.
- **git push:** sube nuestros archivos a un repositorio remoto para que pueda ser accedido por los demás desarrolladores que tienen permiso para modificar este código.
- **git pull:** en caso de trabajar con múltiples desarrolladores, te permite "traer" los cambios que han hecho dichos desarrolladores.
- **git clone:** hace una copia desde el servidor donde se encuentra el código a tu ordenador para que puedas comenzar a trabajar.

Primer ejemplo

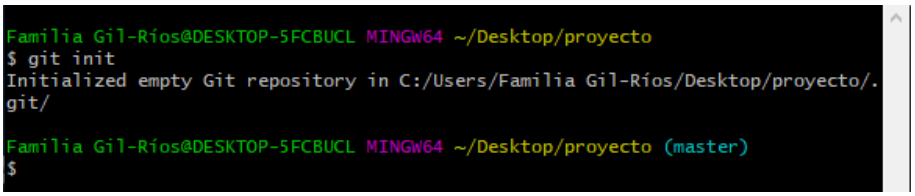
Trabajo en repositorio local

1. En primer lugar crearemos una carpeta para guardar nuestro proyecto.
2. Arrastrándola dentro de nuestro editor de código, por ejemplo, Visual Studio Code, generamos un proyecto, creamos nuestros archivos *(Ej. index.html y app.js)*.
3. Accedemos al repositorio de código usando alguno de los siguientes métodos:
    1. Desde CMD accediendo a nuestra carpeta de proyecto y usando el comando ***git init***.
    2. Pulsando sobre la carpeta de nuestro proyecto con el botón derecho y seleccionando la opción "Git Bash Here"
    3. Abriendo la consola de Git Bash y accediendo a nuestra carpeta de proyecto.



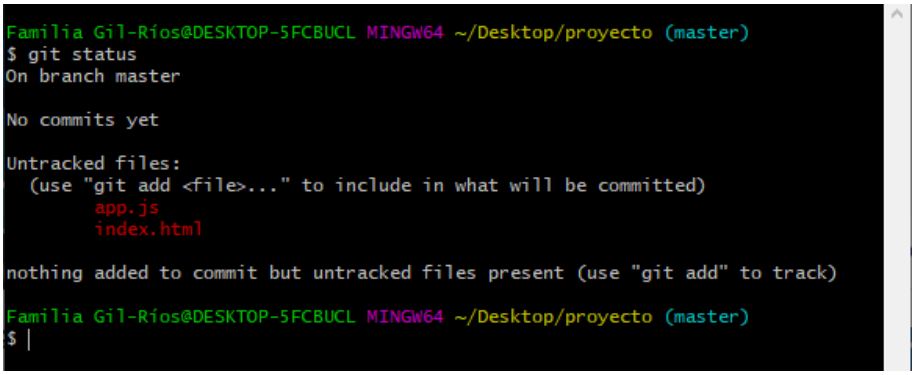
4. **Crear un repositorio local**. Se ejecuta solo una vez, indicando que en dicho directorio se llevará un control de versiones (se convierte en repositorio). Para ello crea un directorio oculto (.git) que contiene la información que Git necesita para registrar las versiones (no tendremos que tocarla para nada).

```
git init
```



5. **Estado actual del repositorio**. Indica los cambios que ha habido en el directorio (repositorio) desde la última versión realizada. Muestra los nuevos ficheros añadidos al repositorio o fuera de seguimiento, los modificados y los eliminados, desde el último commit.

```
git status
```

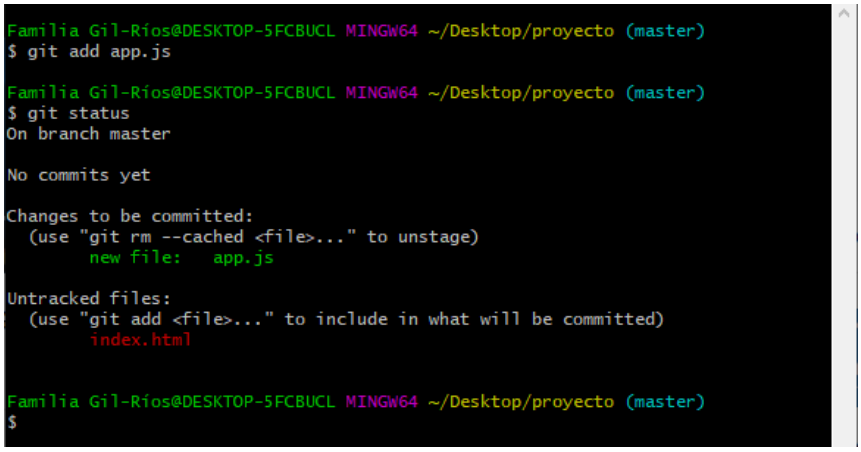


6. **Incluir archivos o directorios para la siguiente versión o revisión**. Permite añadir archivos que están fuera de seguimiento (untracked) o que han sido modificados (modified) y prepararlos para el siguiente commit.

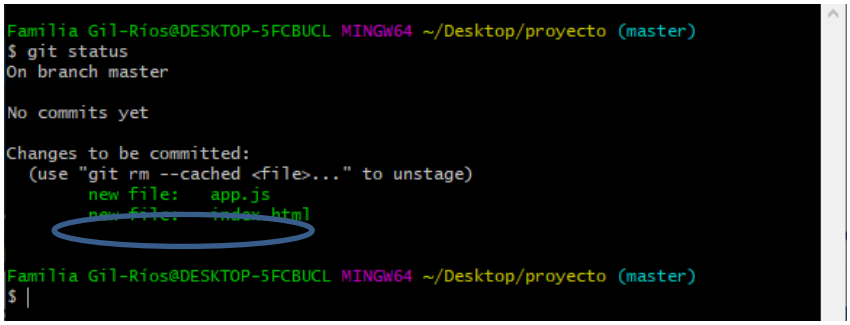
```
git add [nombre_fichero_o_directorio]
# O si quiero añadir todo el contenido de mi repositorio local:
git add .
```

7. **Volver a estado anterior.** Mientras que no se confirmen (*commit*) los archivos, se puede restaurar el ***estado*** inicial con el comando ***git reset*** .
    - En seguimiento (staging area) – Modified → Modificado
    - En seguimiento (staging area) – New file → Untracked

    Por ejemplo, creamos un nuevo archivo index.html en nuestro directorio de trabajo (estado untracked) y lo añadimos al staging area (estado staged).

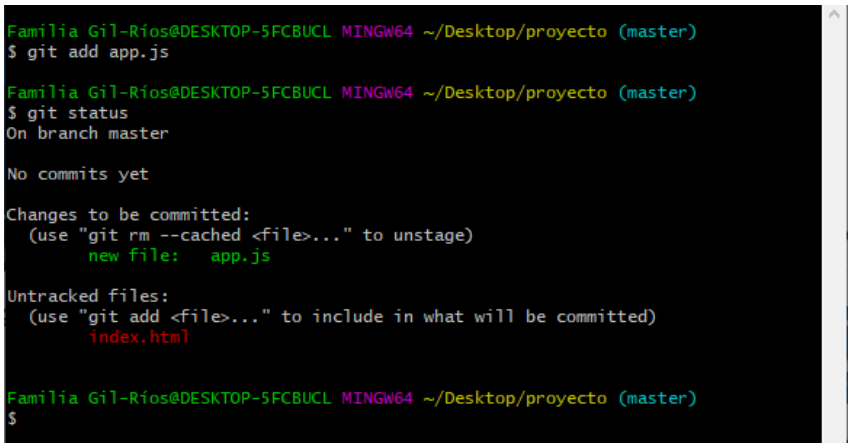


Index.html creado en el directorio de trabajo



Añadidos index.html al staging area

```
$ git reset .
o
$ git reset index.html
```



Restauramos index.html al estado anterior, siempre y cuando no hayamos hecho commit

8. **Deshacer modificación de contenido**. Modificamos el contenido del fichero prueba.css

```
Familia Gil-Ríos@DESKTOP-5FCBUCL MINGW64 ~/Desktop/git-curso (master)
$ git status
On branch master
Your branch and 'origin/master' have diverged,
and have 1 and 1 different commits each, respectively.
  (use "git pull" to merge the remote branch into yours)

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   prueba.css

no changes added to commit (use "git add" and/or "git commit -a")

Familia Gil-Ríos@DESKTOP-5FCBUCL MINGW64 ~/Desktop/git-curso (master)
$
```

```
$ git checkout .
o
$ git checkout prueba.css
```

```
Familia Gil-Ríos@DESKTOP-5FCBUCL MINGW64 ~/Desktop/git-curso (master)
$ git checkout prueba.css
Updated 1 path from the index

Familia Gil-Ríos@DESKTOP-5FCBUCL MINGW64 ~/Desktop/git-curso (master)
$ git status
On branch master
Your branch and 'origin/master' have diverged,
and have 1 and 1 different commits each, respectively.
  (use "git pull" to merge the remote branch into yours)

nothing to commit, working tree clean

Familia Gil-Ríos@DESKTOP-5FCBUCL MINGW64 ~/Desktop/git-curso (master)
$
```

9. Añadimos desde nuestro editor de código un nuevo archivo *(style.css)* y volvemos a realizar un git status.

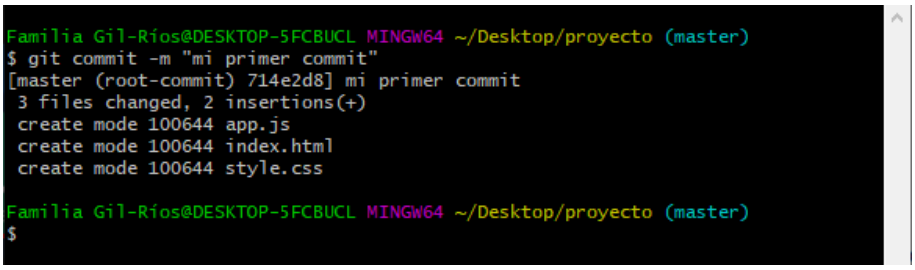
```
Familia Gil-Ríos@DESKTOP-5FCBUCL MINGW64 ~/Desktop/proyecto (master)
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   app.js
        new file:   index.html

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        style.css


Familia Gil-Ríos@DESKTOP-5FCBUCL MINGW64 ~/Desktop/proyecto (master)
$
```

10. Añadimos el fichero: *git add style.css*

11. **Confirmar las modificaciones**. Crea una nueva versión con los archivos que están bajo seguimiento dentro de mi repositorio. Debemos indicar siempre un mensaje descriptivo.

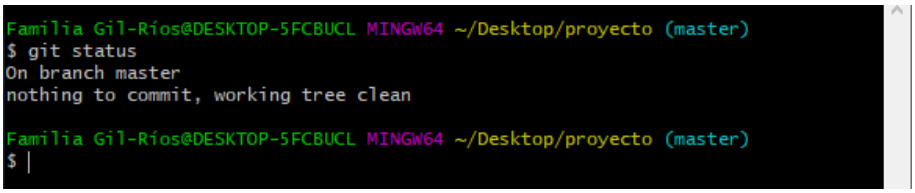
```
git commit -m "Descripción del commit"
```

12. En caso de que no nos hayamos identificado en la configuración de git (nombre y email) nos indicará que debemos hacerlo previamente (véase configuración inicial de Git).
13. En caso de no usar el parámetro –m “Descripción del commit” se nos abrirá un editor (Vim) para añadir en la cabecera (primera línea del editor) dicha descripción. Para salir y guardar cambios pulsamos la tecla Esc y escribimos :wq



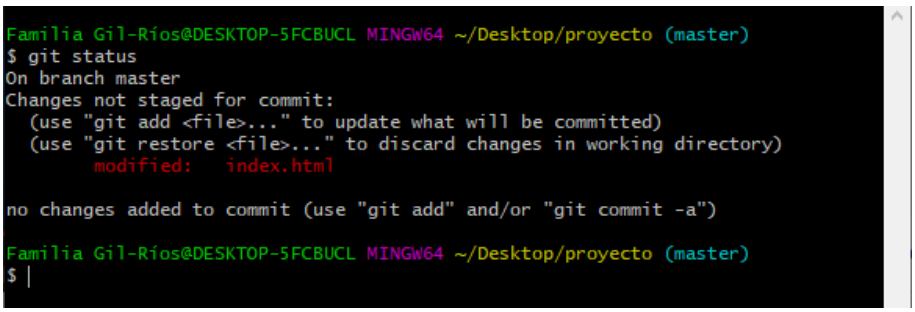
Con este primer commit hemos creado un punto de partida para nuestro proyecto.

14. Si a continuación hacemos un git status nos indica que “no tenemos nada para enviar, nuestro árbol de trabajo está limpio” es decir, todos los cambios están guardados y enviados.

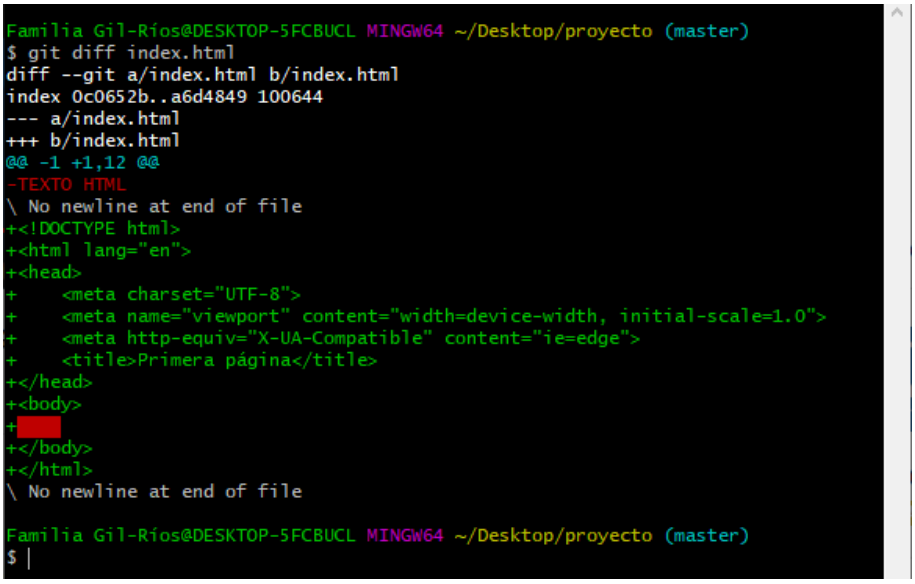


Modificación de archivos

1. Una vez preparada nuestra primera versión. Vamos a ver qué ocurre si modificamos los archivos que componen nuestro proyecto. Accedemos a un archivo desde nuestro editor de código (VSC), lo modificamos y lo guardamos.
2. *git status* nos indicará que el archivo ha sido modificado



3. **Ver diferencias**. Para ver las diferencias entre las diferentes versiones del archivo podemos hacer uso del comando ***git diff <nombre de fichero>***



Deshacer cambios en archivos modificados

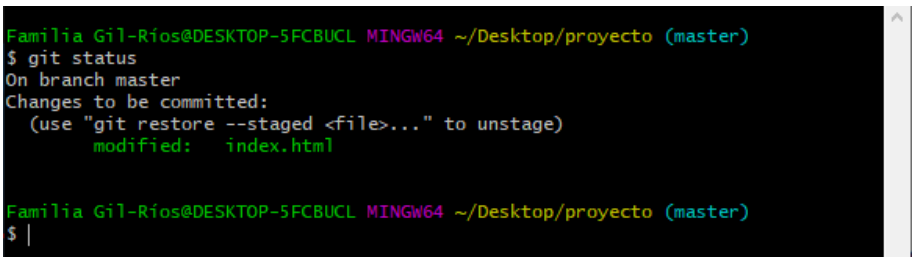
Podemos restaurar todos los archivos modificados:

```
$ git checkout .
o
$ git checkout index.html
```

Añadir archivos modificados

1. Simplemente añadimos el fichero modificado con git add

```
git add [nombre_fichero_o_directorio]
```



Git status nos muestra que existe un fichero preparado para hacerle commit y que es un fichero modificado.

2. Para enviar esta revisión usamos de nuevo el comando:
    - *git commit o git commit –m "descripción"*

```
Familia Gil-Rios@DESKTOP-5FCBUCL MINGW64 ~/Desktop/proyecto (master)
$ git commit
[master b121de9] Index.html incluye estructura básica
 1 file changed, 12 insertions(+), 1 deletion(-)

Familia Gil-Rios@DESKTOP-5FCBUCL MINGW64 ~/Desktop/proyecto (master)
$
```

3. **Mostrar información sobre los diferentes commits realizados**. Si ahora hacemos un ***git log*** podremos observar que tenemos dos versiones:

```
Familia Gil-Rios@DESKTOP-5FCBUCL MINGW64 ~/Desktop/proyecto (master)
$ git log
commit b121de9bd82a3ef0c71da84715dd657263160a74 (HEAD -> master)
Author: Jose Reyes Gil Rubio <josereyesgil@gmail.com>
Date:   Fri Jan 24 13:35:39 2020 +0100

    Index.html incluye estructura básica

commit 714e2d8377aa052fafaef5702da22356f8e2c0c6
Author: Jose Reyes Gil Rubio <josereyesgil@gmail.com>
Date:   Fri Jan 24 12:00:02 2020 +0100

    mi primer commit

Familia Gil-Rios@DESKTOP-5FCBUCL MINGW64 ~/Desktop/proyecto (master)
$
```

La versión de trabajo sería la que nos indica HEAD

Obtener versiones anteriores

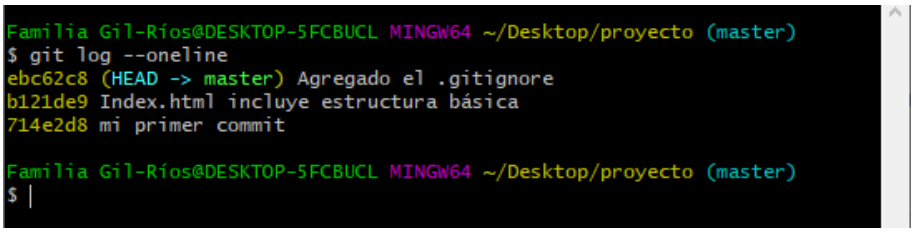
Cuando queremos obtener una versión anterior de nuestro proyecto podemos consultar el log de nuestro repositorio local. Cada confirmación (commit) crea una versión. Al restaurar a una versión anterior, el estado (ficheros y directorios) de mi repositorio local se cambia por lo que había en el momento de crear dicha versión.

```
git log #Muestra un historial con las versiones

git log --oneline #Muestra un historial resumido
```

Obtenemos la siguiente salida, que representa todos los commits realizados en mi repositorio local y el mensaje que se incluyó.

```
Familia Gil-Rios@DESKTOP-5FCBUCL MINGW64 ~/Desktop/proyecto (master)
$ git log
commit ebc62c860f7d92b75305e0439b20427dde440aca (HEAD -> master)
Author: Jose Reyes Gil Rubio <josereyesgil@gmail.com>
Date:   Fri Jan 24 15:39:30 2020 +0100

    Agregado el .gitignore

commit b121de9bd82a3ef0c71da84715dd657263160a74
Author: Jose Reyes Gil Rubio <josereyesgil@gmail.com>
Date:   Fri Jan 24 13:35:39 2020 +0100

    Index.html incluye estructura básica

commit 714e2d8377aa052fafaef5702da22356f8e2c0c6
Author: Jose Reyes Gil Rubio <josereyesgil@gmail.com>
Date:   Fri Jan 24 12:00:02 2020 +0100

    mi primer commit

Familia Gil-Rios@DESKTOP-5FCBUCL MINGW64 ~/Desktop/proyecto (master)
$
```

git log, nos muestra hash, autor, fecha, descripción del commit

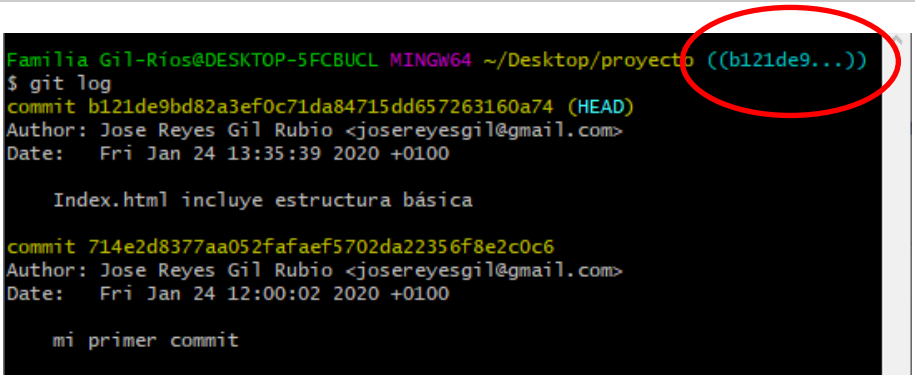


git log –oneline

Para volver a una versión anterior

$ git checkout <número de id del commit> por ejemplo

```
$ git checkout b121de9
```



Para volver a la versión actual

```
$ git checkout master
```

Para eliminar una versión completamente

Eliminamos el último/s commit de la historia, dejando la carpeta en el punto anterior en el que estaba antes de hacer los cambios correspondientes a dicho commit. Perderás todos los cambios que habías realizado en este commit.

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

O

$ git reset HEAD^ --hard
```

Donde el número 1 indica el número de commits que deseamos eliminar a partir de HEAD.

Para eliminar un commit sin perder los cambios posteriores

Con la siguiente opción retrocedemos en las versiones y no perdemos los cambios de los commits posteriores. **Todos los cambios aparecerán como pendientes para realizar un commit**.

```
$ git reset --soft HEAD~1

O

$ git reset HEAD^ --soft
```

Añadir archivos a la última versión sin crear una nueva

- Creamos los archivos que faltan o modificamos los ya existentes
- Los añadimos con git add
- Ejecutamos el siguiente comando:

```
$ git commit --amend
```

Comparativa entre git reset y git checkout

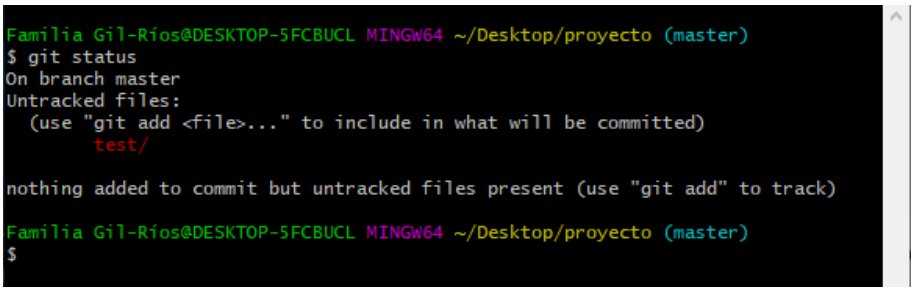
Command	Scope	Common use cases
git reset	A nivel de commit	Descarta commit
git reset	A nivel de ficheros	Restaura etapa anterior de un fichero
git checkout	A nivel de commit	Cambia entre commit para inspeccionar
git checkout	A nivel de ficheros	Descarta cambios en el directorio de trabajo

Ignorar archivos en las versiones

Es bastante frecuente que en los proyectos que tengo en mi repositorio local tenga archivos o directorios que no quiero tener en seguimiento (versiones). Es el caso del directorio /bin o de los ficheros .class de Java, o directorios de ejemplos o tests.

Por ejemplo, creamos en nuestro proyecto una carpeta llamada test y dentro creamos un archivo .js, por ejemplo.

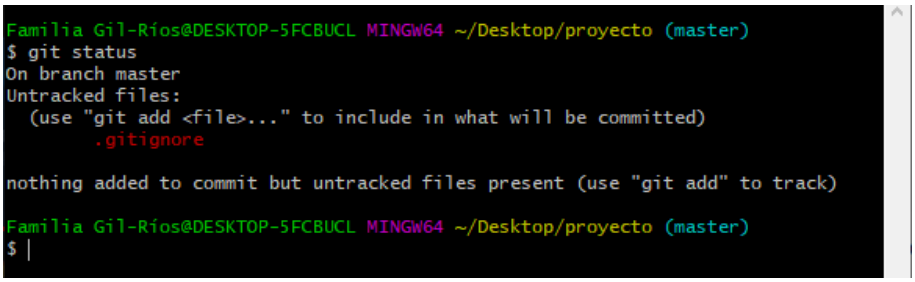
Al hacer un git status nos indicará que tenemos una nueva carpeta, pero ¿qué hacemos si queremos ignorar esta carpeta en nuestro git?.



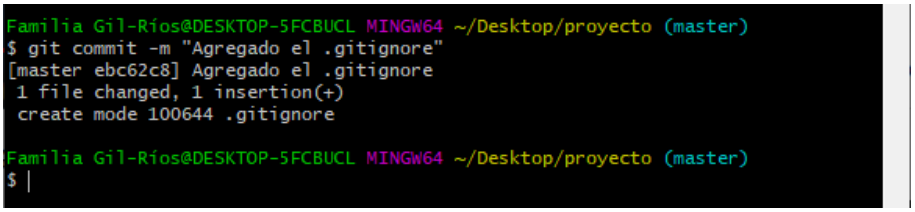
1. Crearemos un fichero en nuestro proyecto llamado *.gitignore*. El fichero .gitignore es un fichero de texto que podemos crear en nuestro repositorio local y que indica qué ficheros se excluirán del seguimiento.

   .gitignore

```
# Ficheros generados Java
*.class
# Carpetas que contienen ficheros generados
bin/
out/
# Carpetas que contienen ficheros de test o pruebas
test/
```

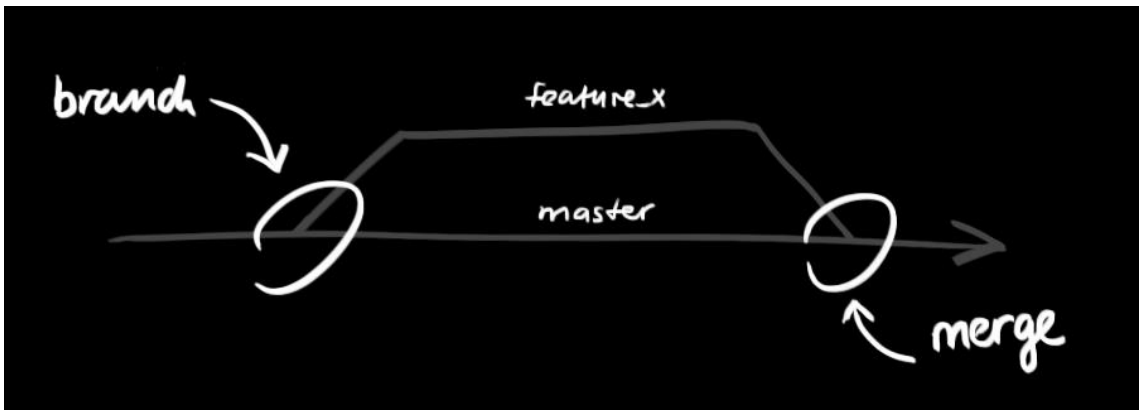


2. El fichero ***.gitignore*** debería ser añadidio a nuestro git con el resto de ficheros que conforman nuestro proyecto. Por lo tanto *git add .gitignore*.
3. Realizamos el tercer commint *git commit –m "Agregado el .gitignore"*.



Trabajar con ramas

Git nos permite trabajar con versiones alternativas a nuestro proyecto, para esto hacemos uso de las "ramas".



Listado de ramas existentes

```
git branch
```

```
Familia Gil-Ríos@DESKTOP-5FCBUCL MINGW64 ~/Desktop/proyecto (master)
$ git branch
* master
```

Crear una nueva rama o versión alternativa

```
git branch login
```

```
Familia Gil-Ríos@DESKTOP-5FCBUCL MINGW64 ~/Desktop/proyecto (master)
$ git branch login

Familia Gil-Ríos@DESKTOP-5FCBUCL MINGW64 ~/Desktop/proyecto (master)
$ git branch
  login
* master
```

Cambiar de rama

```
git checkout login
```

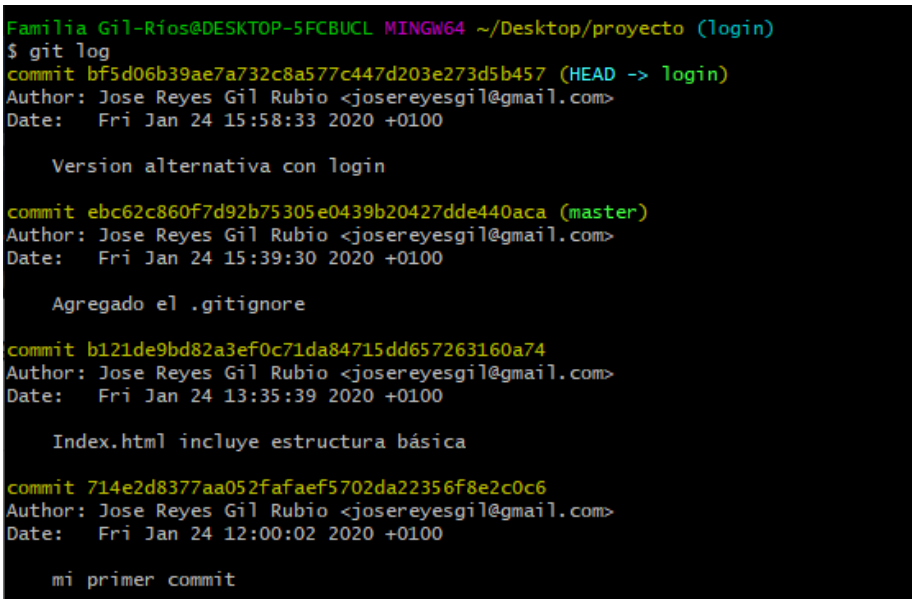
```
Familia Gil-Ríos@DESKTOP-5FCBUCL MINGW64 ~/Desktop/proyecto (master)
$ git checkout login
Switched to branch 'login'

Familia Gil-Ríos@DESKTOP-5FCBUCL MINGW64 ~/Desktop/proyecto (login)
$ git branch
* login
  master
```

Como me encuentro en la rama login, puedo cambiar elementos de mi proyecto, por ejemplo, creo carpetas login, authentication,… (NOTA: Las carpetas sólo aparecerán en git status si contienen algún archivo).

```
Familia Gil-Ríos@DESKTOP-5FCBUCL MINGW64 ~/Desktop/proyecto (login)
$ git status
On branch login
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        authentication/
        index.php
        login/

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

Añadimos todos los archivos *git add .* (punto)

Creamos nueva versión ***git commit –m "Versión alternativa con login"***, comprobamos con ***git log***.



Volver a rama master

Para volver a la rama master ***git checkout master***, una vez de nuevo en la rama master podremos observar que los archivos creados en la otra rama (login) no aparecen en la rama master.

Además si hacemos un git log veremos que no aparece el último commit, porque este último commit no pertenece a esta versión.

Fusionar ramas

```
git checkout master

git merge nombre-rama
```

Eliminar ramas

Para eliminar una rama de nuestro repositorio local usaremos el comando:

```
$ git branch -d nombre_rama
```

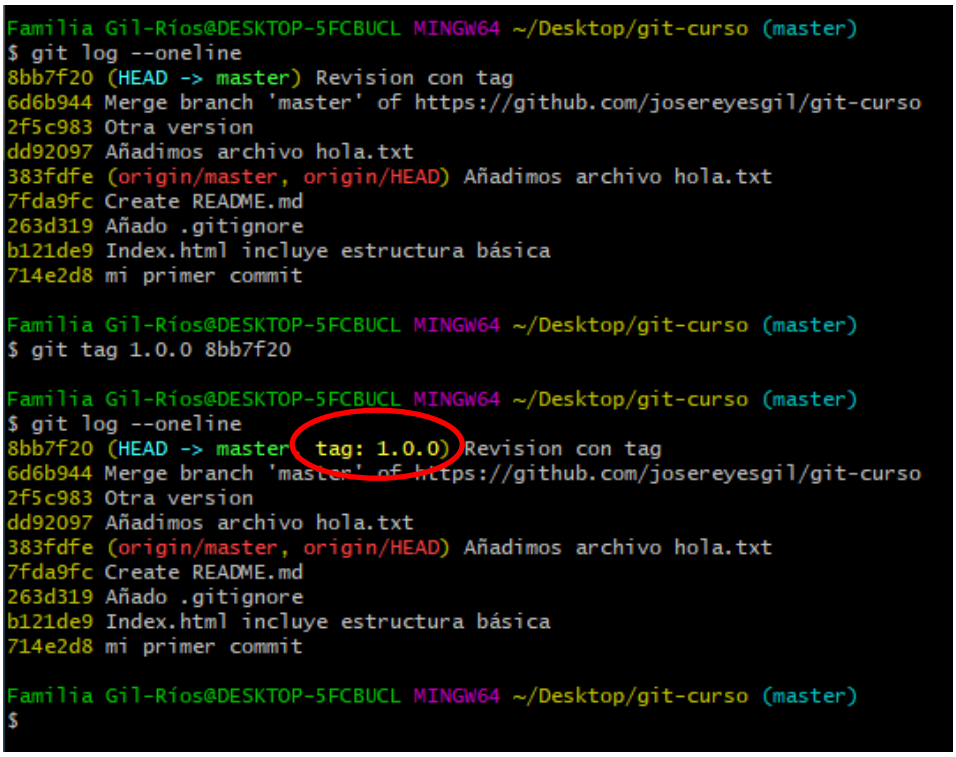
En el caso de que esa rama contenga trabajos sin fusionar, el comando anterior nos devolverá un error. Si aun así queremos eliminar esa rama, se puede forzar el borrado de la siguiente manera:

```
$ git branch -D nombre_rama
```

Etiquetar revisiones

Se recomienda crear etiquetas para cada nueva versión publicada de un software.

git tab v.v.v hash_revisión ***P.ej.: git tag 1.0.0 1b2e1d6***



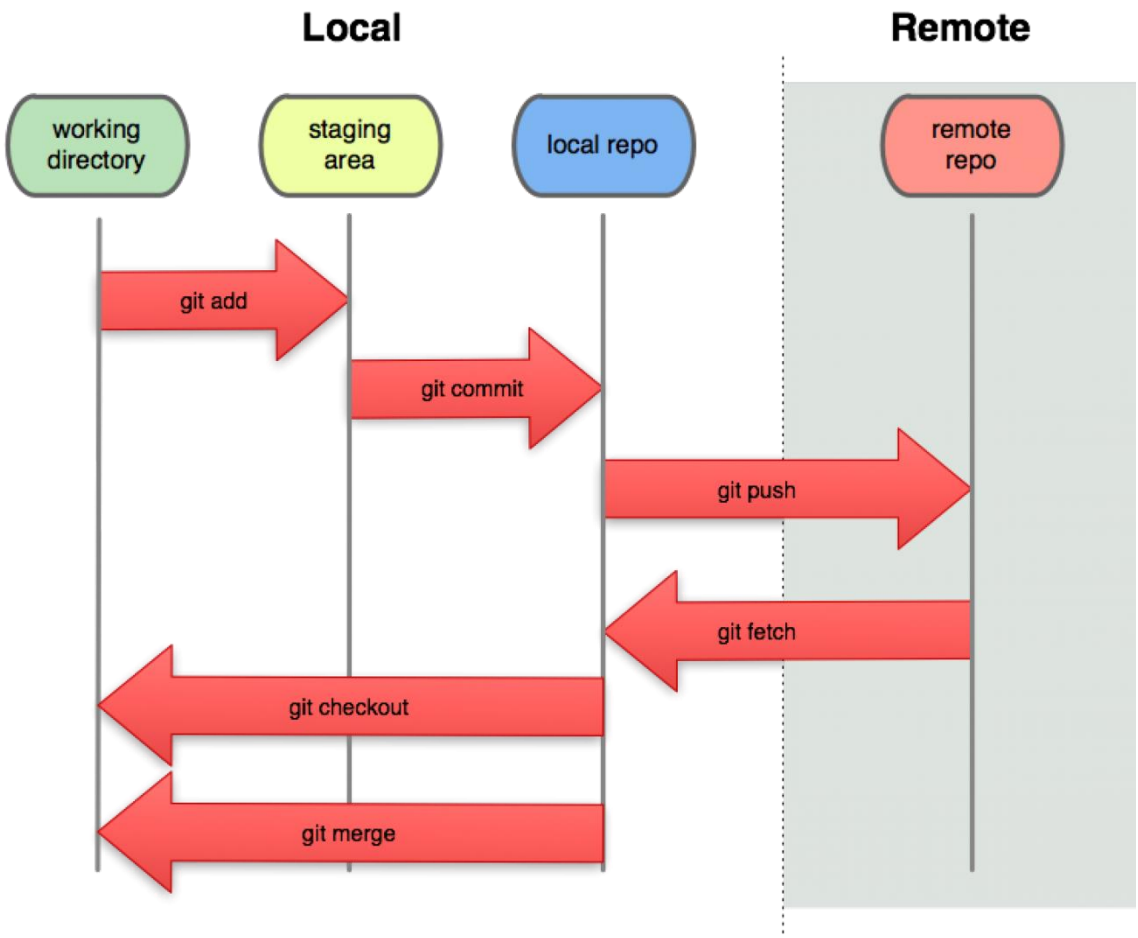
Otros comandos

- git add –i → Agregar archivos de forma interactiva
- gitk → Interfaz gráfica por defecto

GIT Y GITHUB

Ya conocemos los comandos básicos. Ahora, cuando nosotros trabajamos en un proyecto real deberemos subir nuestro proyecto a un repositorio de código remoto para que múltiples desarrolladores tengan acceso a él, participando en su desarrollo o ayudándonos a modificarlo.

Básicamente, el diagrama de funcionamiento sería algo similar a lo siguiente:

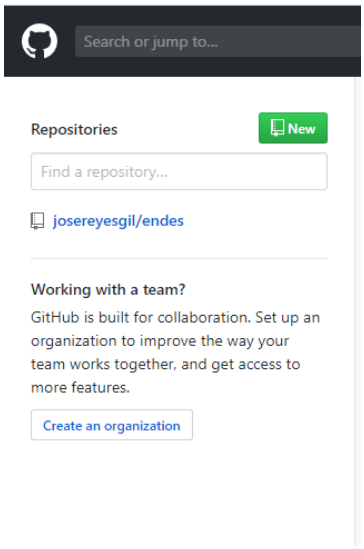


GitHub es un sistema de gestión de proyectos y control de versiones de código, así como una plataforma de red social diseñada para desarrolladores. ¿Pero para qué se usa GitHub? Bueno, en general, permite trabajar en colaboración con otras personas de todo el mundo, planificar proyectos y realizar un seguimiento del trabajo.

GitHub es también uno de los repositorios online más grandes de trabajo colaborativo en todo el mundo.

Registro y creación de nuestro primer repositorio

1. El primer paso por el que debemos pasar para trabajar con GitHub es **registrarnos (**https://github.com/**)**.
2. A continuación agregaremos un **nuevo repositorio (New)** o https://github.com/new.



3. Le damos nombre, descripción y seleccionamos si es público o privado.
4. Pulsamos en Create repository.
5. Ya tenemos el repositorio creado con una dirección URL propia.
   https://github.com/nombre-usuario/nombre-repositorio
6. En la pantalla inicial del repositorio nos informa de una serie de comandos.

Exportar a GitHub nuestro proyecto desde Git

1. Desde nuestra consola de Git ***(Git Bash)*** podemos hacer uso de estos comandos para añadir un proyecto de Git ya existente a GitHub. Usaremos los comandos:

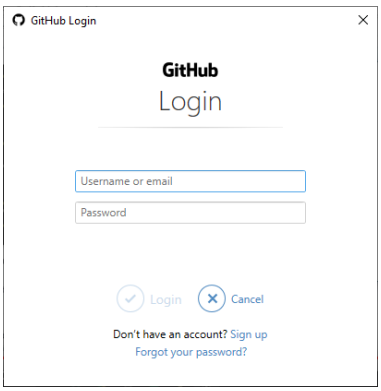
   git remote add origin https://github.com/josereyesgil/git-curso.git

   (Conecta nuestro repositorio local a un repositorio remoto que debe estar vacío)

   Con él indicaremos cual es la dirección del lugar donde almacenaremos el código.

   git push -u origin master

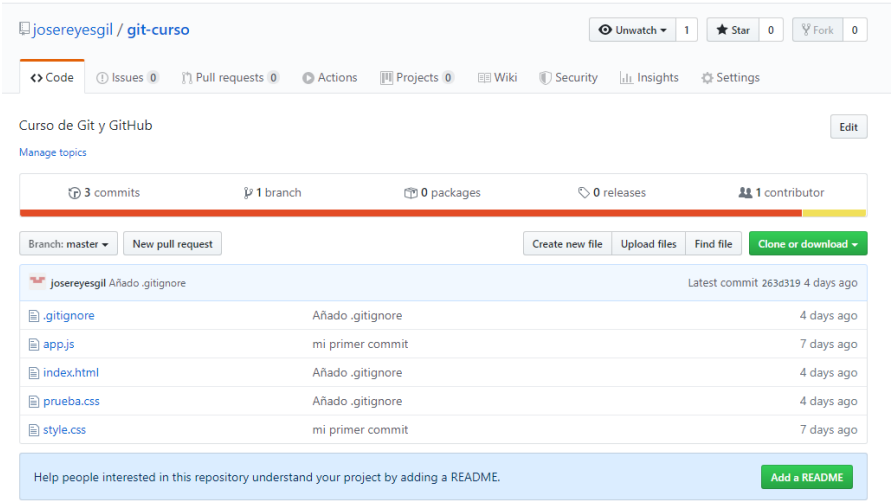
   Incluimos los archivos de nuestro proyecto en este repositorio.



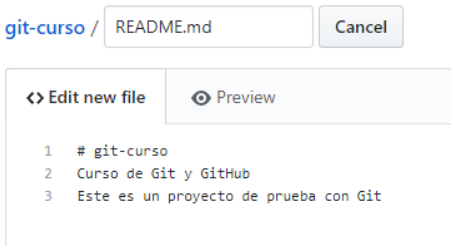
2. Nos logamos en nuestra cuenta de GitHub con usuario y contraseña en caso necesario.

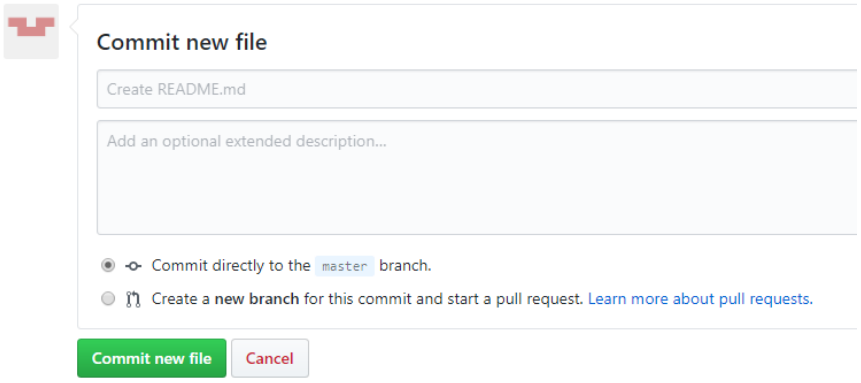
```
Familia Gil-Ríos@DESKTOP-5FCBUCL MINGW64 ~/Desktop/proyecto (master)
$ git remote add origin https://github.com/josereyesgil/git-curso.git

Familia Gil-Ríos@DESKTOP-5FCBUCL MINGW64 ~/Desktop/proyecto (master)
$ git push -u origin master
Enumerating objects: 12, done.
Counting objects: 100% (12/12), done.
Delta compression using up to 2 threads
Compressing objects: 100% (8/8), done.
Writing objects: 100% (12/12), 1.22 KiB | 78.00 KiB/s, done.
Total 12 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), done.
To https://github.com/josereyesgil/git-curso.git
 * [new branch]      master -> master
Branch 'master' set up to track remote branch 'master' from 'origin'.

Familia Gil-Ríos@DESKTOP-5FCBUCL MINGW64 ~/Desktop/proyecto (master)
$
```

3. Si volvemos a nuestro repositorio en GitHub podremos comprobar que nos ha subido todo nuestro repositorio local a nuestro repositorio remoto en GitHub.



4. Podemos añadir un archivo README a nuestro repositorio. *"Help people interested in this repository understand your project by adding a README"*. Pulsando sobre "Add a README que nos permitirá añadir ese archivo a nuestro proyecto.

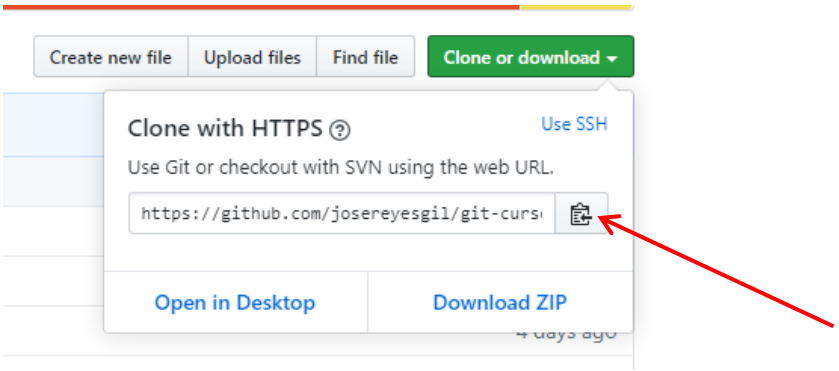




En este caso sólo pulsaremos en el botón verde "***Commit new file***"

Importar nuestro proyecto desde GitHub

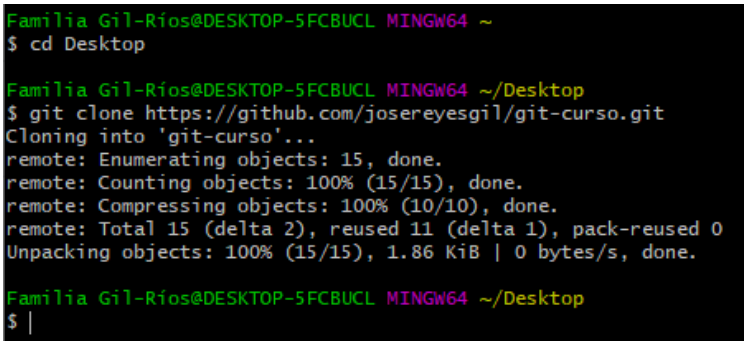
1. Para demostrar esta acción vamos a eliminar nuestro repositorio local (carpeta proyecto) de nuestro equipo.
2. Desde nuestro repositorio remoto en GitHub copiamos su dirección URL. Para ello, hacemos uso del botón "***Clone or download***"



*NOTA: git clone nos permite hacer clonados de nuestro repositorio local a otra carpeta de nuestro equipo (**git clone /path/to/repository**)*

3. Abrimos Git Bash, nos dirigimos al Escritorio (ubicación de nuestro repositorio local) e insertamos el siguiente comando:

$ ***git clone <dirección de repositorio remoto>***



4. Hemos "descargado" todo nuestro proyecto a un repositorio local.

Actualizar nuestro repositorio remoto desde nuestro repositorio local

1. Añadimos un archivo a nuestra carpeta de proyecto. Ej. hola.txt con el texto “hola mundo”
2. Desde consola y desde nuestro directorio de trabajo: ***git status***. Comprobamos que tenemos pendiente un archivo.
3. Ahora añadimos el archivo con: ***git add .*** (o nombre de archivo) pasándolo a la zona de staging area.
4. Si a continuación, modificamos el archivo añadiendo, por ejemplo, más texto y haciendo un ***git diff***, nos mostrará las diferencias entre el archivo original y el modificado.
5. ***git commit –m “Añadimos archivo hola.txt”***
6. ***git push –u origin master*** (podría pedirnos nombre de usuario y contraseña), con esto subimos los archivos nuevos a nuestro repositorio en GitHub.
7. Si vamos a nuestro GitHub podemos comprobar que, efectivamente, aparece nuestro nuevo commit con el archivo hola.txt.

Actualizar tu repositorio local al commit más nuevo de GitHub

Si se han realizado modificaciones en nuestro proyecto por parte de otros colaboradores o por nosotros mismos podemos actualizar nuestro repositorio local desde GitHub con el siguiente comando:

git pull

Volver a la versión del repositorio remoto en nuestro repositorio local

Si quieres deshacer todos los cambios locales y commits, puedes traer la última versión del servidor y apuntar a tu copia local principal de esta forma:

1. ***git reset --hard HEAD~5*** (Eliminamos varios commit para comprobar)
2. ***git reset --hard origin/master***

o

1. ***git reset --hard HEAD~5*** (Eliminamos varios commit para comprobar)
2. ***git fetch*** (tan sólo recupera la información del repositorio remoto y la ubica en una rama oculta de tu repositorio local (FETCH_HEAD, podemos verla con ***git branch -a***), por lo que no la fusionará automáticamente con tu repositorio local. En este caso tenemos que saber que por cada repositorio remoto que tengamos configurado también tendremos una rama oculta de este.)
3. ***git merge origin/master*** (Fusionamos la rama oculta con la rama local en la que estamos trabajando).

Eliminar un repositorio en GitHub

1. Entramos en el repositorio a eliminar
2. Clic sobre el botón Settings
3. Dentro de settings hacemos scroll hasta el final.
4. Clic sobre Delete this repository

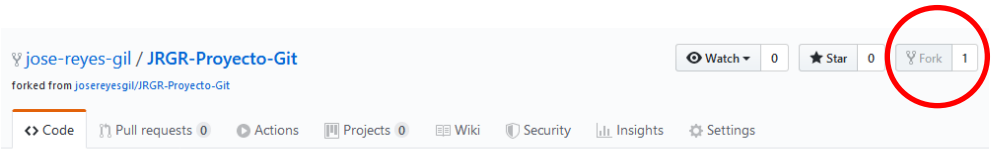
Cómo colaborar en un proyecto en GitHub sin ser colaborador

Acceso al repositorio

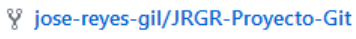
Accedemos desde nuestra cuenta de GitHub al repositorio con el que queremos colaborar a través de su URL.

Fork del repositorio

A continuación hacemos un Fork del repositorio con el que queremos colaborar. Esto quiere decir, hacer una copia que nos permita experimentar libremente sin afectar al proyecto original.



En nuestra lista de repositorios aparecerá este repositorio como forkeado.



Clonar el repositorio

Después de tener el repositorio en nuestra cuenta, seleccionamos la dirección del repositorio en el botón Clone or download y la copiamos.

Nos vamos a nuestra consola de git y clonamos.

$ git clone <dirección de repositorio forkeado>

Dentro de la carpeta que genera, comprobamos la URL del repositorio original del proyecto.

$ git remote -v

Antes de realizar modificaciones agregar la URL del repositorio original del proyecto:

$ git remote add upstream <dirección del repositorio original>

Comprobar

$ git remote –v

Actualizar la rama master

Como la utilidad de nuestra rama master es mantener todos los cambios del repositorio original, debemos actualizarla antes de comenzar a trabajar, así tendremos los cambios que se hagan a último minuto.

$ git pull –r upstream master

Crear una rama

Se recomienda siempre hacer una rama desde master para realizar cambios y proponer su inclusión en el repositorio oficial.

$ git branch <nombre_rama>
$ git checkout <nombre_rama>

O

$ git checkout –b <nombre_rama>

Hacer cambios

Realizar todos los cambios que se desea hacer al proyecto.

Agregar los archivos y hacer un commit

Después de realizar el commit hacer el push hacia nuestro repositorio indicando la rama que hemos creado.

$ git push origin <nombre_rama>

Hacer un Pull Request

Haremos clic en “Compare & Pull Request”

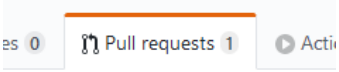
Añadimos el contenido de los cambios para notificarle al propietario

Pulsamos sobre “Create Pull Request”

Esperar a que el dueño del repositorio lo revise, acepte y mezcle en la rama correspondiente.

Por parte del propietario del repositorio

El propietario recibirá la solicitud de modificación en la solapa “***Pull requests***”.



que podrá aceptarla pulsando en “***Merge pull request***” y “***Confirm merge***”

Desconectar nuestro repositorio local del repositorio remoto

Para desconectar nuestro repositorio local del repositorio remoto basta con utilizar el siguiente comando:

$ git remote remove origin

Invitar colaboradores a un repositorio personal

Si usas GitHub en su versión gratuita, podrás agregar colaboradores ilimitados en repositorios públicos, y hasta tres colaboradores en repositorios privados.

GitHub limita la cantidad de personas que se pueden invitar a un repositorio 2n 24 horas. Si excedes ese límite, tendrás que esperar 24 horas para colaborar con más personas.

1. Solicita el nombre de usuario de la persona que estás invitando como colaboradora.
2. En GitHub visita la página principal del repositorio.
3. Pulsa en Settings.
4. En la barra lateral izuiqerda, haz clic en Collaborators (Colaboradores).
5. Empieza a escribir el nombre del usuario colaborador.
6. Selecciónalo de la lista y pulsa en Add Collaborator.
7. El usuario recibirá un correo electrónico invitándolo al repositorio. Una vez que acepte la invitación, tendrá acceso de colaborador a tu repositorio.

You now have push access to the <nombre_repositorio> repository.

Eliminar un colaborador de un repositorio personal

1. En GitHub, visita la página principal del repositorio.
2. Abre Settings.
3. En la barra lateral izquierda haz clic sobre Collaborators (Colaboradores)
4. Al lado del colaborador que deseas eliminar, haz clic en el icono X.