Sistemas de control de versiones

Rubén Gómez Olivencia

2024-09-08

- Github: https://github.com/yuki

Puedes encontrar la última versión de este libro en formato **HTML** en el siguiente link, así como otros libros que he creado. Para descargar el código fuente en formato **Markdown** visita el repositorio en GitHub.

Información

Por favor, ponte en contacto conmigo si encuentras algún fallo, falta de ortografía o quieres mejorar de alguna manera este libro. Gracias.

I Sistemas de control de versiones

II Introducción a Git

III GitHub como servidor remoto

IV Ramas, merges y conflictos

Sistemas de control de versiones

1. Introducción

Un sistema de control de versiones es un sistema que nos permite tener un histórico de las modificaciones que sufre un fichero a lo largo del tiempo.

Si pensamos en un documento, un ciclo de vida con modificaciones puede ser:

1. Crear el documento.
2. Correcciones.
3. Cambios gramaticales.
4. Modificar colores de los encabezados.
5. Añadir logo de la compañía.
6. Modificaciones finales.

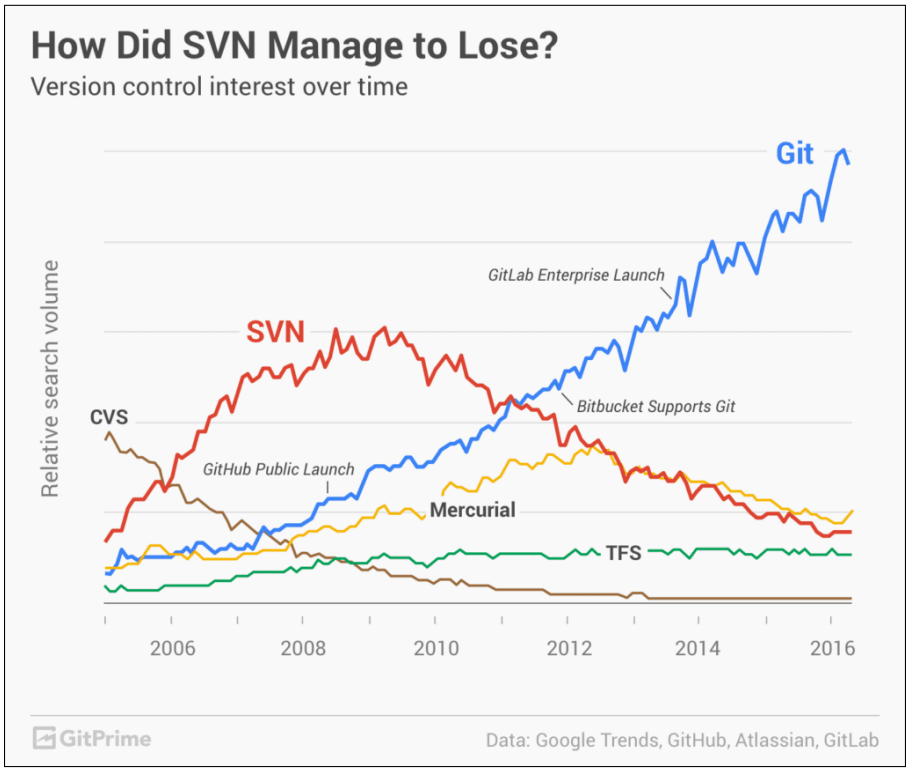
Los sistemas de control de versiones pueden controlar las modificaciones de cualquier tipo de fichero, pero **son especialmente útiles cuando se trata con ficheros de tipo texto**, como código fuente, documentos tipo texto/markdown, imágenes de tipo vectorial svg...

2. Un poco de historia

Aunque existen muchos sistemas de control, vamos a enumerar unos pocos que han tenido cierta relevancia en el mundo del software:

- **CVS**: En inglés *concurrent versions systems*, creado en el año 1990, comenzó como un *frontend* de un sistema de versiones anterior (llamado RCS). Añadió funcionalidades sobre RCS hasta que se consideró un sistema propio. En el mundo del software libre ganó muchos adeptos a pesar de que era complicado de utilizar y tenía ciertas carencias y fallos.
- **Subversion**: Apareció en el año 2000 con la intención de ser parecido a CVS pero tratando de corregir fallos del anterior y añadirle características que carecía. Hace uso de un sistema **basado en un repositorio central** al que se envían los cambios. Debido a las mejoras que tenía, y la aparición del portal Sourceforge, se vuelve muy utilizado y prácticamente como el sistema principal del Sofware Libre.
- **Bitkeeper**: Es un sistema de control de versiones **distribuido** originalmente como software privativo, pero que permitió hacer uso de manera gratuita a los desarrolladores del kernel Linux (hasta el año 2005). Los desarrolladores del kernel Linux adoptaron esta solución porque ya desde 1998 estaban teniendo problemas, y **no podían adoptar un sistema centralizado**.
- **GNU Bazaar**: Creado en 2005 por la empresa Canonical (creadores de Ubuntu), es un sistema de control de versiones distribuido. La idea era impulsarlo como gestor del código utilizado en Ubuntu.

- **Git**: Creado por Linus Torvalds en 2005 debido al cambio de licencia de Bitkeeper. Decidió crearlo debido a que quería un sistema distribuido tal como usaba Bitkeeper, pero ninguno de los que había en ese momento le satisfacía.



Interés por los distintos sistemas de control de versiones. Fuente.

Wikipedia tiene una página donde se puede visualizar una comparativa de distintos sistemas de control de versiones. En ella se comparan información general, licencias, características, ... Es una buena manera de conocer otros sistemas aparte de los nombrados previamente.

3. Características habituales

Aunque cada sistema de control de versiones es diferente, todos tienen las siguientes características:

- Comprobar el estado de los ficheros: si han sido modificados.
- Identificar cada cambio de una manera única.
- Conocer quién ha realizado las modificaciones que han sufrido los ficheros.
- Visualizar la diferencias que ha habido entre versiones en los ficheros.
- Volver atrás a una versión concreta del fichero, o de todo el proyecto.
- Tener un sistema de etiquetas para nombrar un estado concreto del proyecto.

Más adelante, con el ejemplo de Git, veremos qué significa cada una de esas características y cómo se realiza en Git.

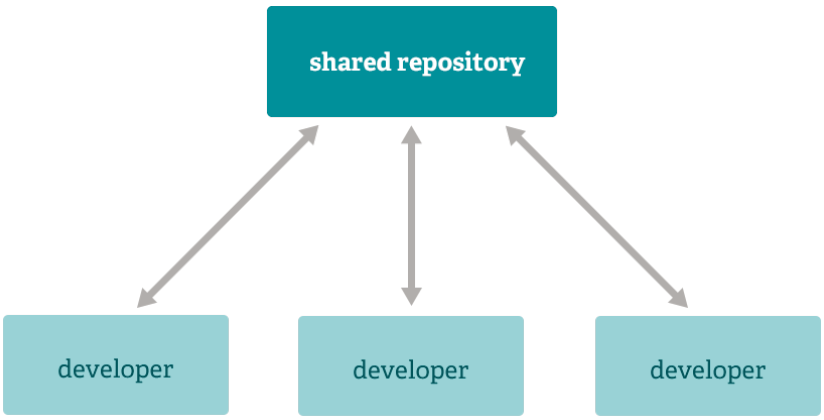
4. Tipos de sistemas de control de versiones

Dentro de los sistemas de control de versiones se pueden diferenciar dos tipos cuando hablamos de cómo se interactúa y de cómo se almacena el histórico de todos los cambios.

4.1. Centralizado

El sistema centralizado era el más utilizado hasta la llegada y expansión de Git. Sistemas centralizados son CVS y Subversion.

El histórico de las modificaciones de nuestro repositorio se encontraba centralizado en un servidor. Cada vez que un usuario quería comprobar los cambios que había realizado, crear un commit, o volver a una versión anterior del proyecto **necesitaba realizar una conexión con el servidor central**.



Arquitectura centralizada. Fuente.

Esto suponía que era necesario tener siempre acceso a internet, y aparte, que para realizar pequeñas acciones cotidianas necesitases esperar a la respuesta del servidor.

Tampoco se podía saber si alguien había realizado modificaciones en el código hasta que no se intentasen subir nuevas modificaciones. De haber modificaciones y no tenerlas en la copia de trabajo local, había que resolver el conflicto, pudiendo dejar el servidor central en estado bloqueado.

¡Cuidado!

Los conflictos bloquean el servidor, por lo que hasta que no se resuelvan, nadie puede subir cambios.

4.2. Distribuido

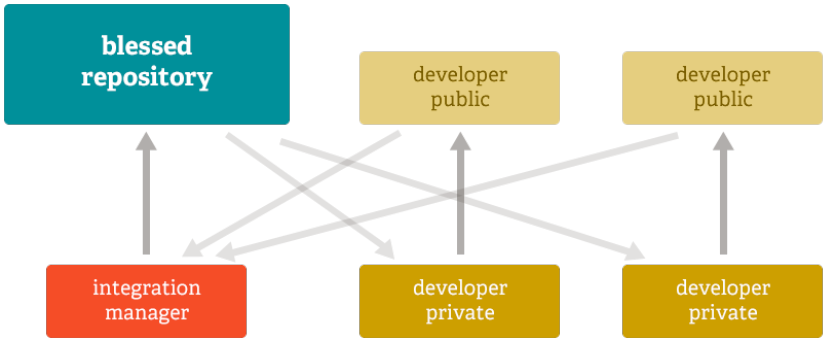
Los sistemas de control de versiones distribuidos siguen la filosofía de que **en cada copia de trabajo existen todos los datos, metadatos y el histórico completo de modificaciones** que ha tenido el proyecto desde el inicio de los tiempos.

Gracias a eso, permite hacer uso del trabajo *offline*, no necesitando la conexión a internet hasta que no nos interese subir los cambios realizados a un repositorio donde el resto de desarrolladores puedan acceder.

También es posible crear ramas locales, comprobar cómo ha evolucionado el proyecto, realizar *diffs* … **sin necesidad de realizar ningún tipo de conexión**, por lo que estos cambios se realizan en local **mejorando la velocidad de trabajo**.

Trabajando de esta manera, también **nos permite centrarnos en las características que estamos realizando**, dejando para más adelante la posibilidad de que existan conflictos.

Por otro lado, el flujo de trabajo puede variar entre proyectos, y dentro de nuestro repositorio podemos tener distintos orígenes de los que obtener cambios. Un ejemplo de sistema distribuido:



Workflow distribuido. Fuente.

Información

En los sistemas distribuidos podemos hacer que el sistema de control de versiones se adapte a nuestra manera de trabajar, y no al revés.

5. Glosario

A la hora de utilizar un sistema de control de versiones tenemos que conocer cierta nomenclatura que es habitual. Es importante conocer estas palabras para identificar a qué términos nos estamos refiriendo.

Como son palabras utilizadas por los propios sistemas de control de versiones en sus clientes de línea de comandos (o en interfaces gráficas), se ha decidido mantener en inglés.

Blame Comprobar el autor y la revisión de la última vez que se modificó una línea de un documento. En inglés *blame* significa culpar/acusar.

Branch Los sistemas de control de versiones nos permiten crear ramificaciones (temporales o perpetuas) partiendo de una versión concreta.

Checkout Es utilizado para poner el repositorio local en una revisión concreta del histórico.

Clone Clonar significa crear un repositorio obteniendo todas las revisiones de otro repositorio.

Commit Puede tener dos acepciones:

- Como nombre: Un "***commit***" (o revisión), es el conjunto de modificaciones que se empaquetan conjuntamente. Un "commit" muestra las modificaciones realizadas respecto al commit anterior.

¡Atención!

Los commits deberían llevar modificaciones que tengan que ver entre sí y tratando de que sea código válido.

- Como verbo: Hacer un "commit", o "***commitear***", es la acción de empaquetar modificaciones de nuestra copia de trabajo, para crear un "*commit*", o revisión, que va a pertenecer al histórico del proyecto.

Origin En Git, es el nombre por defecto del repositorio remoto del que nos hemos clonado el repositorio.

Pull Obtener y aplicar en el repositorio local todos los commits desde un repositorio remoto. En caso de existir cambios en local y en los commits, puede surgir conflictos.

Push Enviar los commits creados en local que no están en el repositorio remoto "central".

Repository Puede tener distintas acepciones dependiendo del sistema de control de versiones que estemos utilizando. Es la estructura de directorios y ficheros (junto con su metadata) en el que guardamos el proyecto que nos interesa versionar. En sistemas distribuidos, tanto la copia local como la remota se denominan repositorios.

Stage Area Es el área donde están los cambios que pueden ir en el siguiente commit.

Tag Una etiqueta es la manera de darle un nombre que podemos identificar de manera "humana" a una revisión concreta. Normalmente se usa para indicar las versiones del software: v1.0, v1.5, …

Working copy Es la copia local de un repositorio, en un momento específico del tiempo o revisión.

II

Introducción a Git

1. Introducción

Git es un sistema de control de versiones distribuido creado en 2005 por Linus Torvalds, el mismo creador del kernel Linux. En tres días el sistema de control de versiones ya estaba versionando su propio código, y en dos semanas ya tenía gestión de ramas.

El proyecto se inició debido a que el sistema que utilizaban para la gestión del kernel Linux (Bitkeeper, en ese momento software privativo) decidió dejar de dar licencias gratuitas a los desarrolladores de Software Libre.

Linus hizo un análisis de los sistemas de control que existían en ese momento, y al ver que ninguno se adaptaba a las necesidades de un sistema tan complejo como el proyecto Linux, decidió crear uno propio.

Para el 16 de junio de 2005, Git manejaba el código fuente completo del kernel Linux, siendo el sistema utilizado a partir de ese momento. No sólo los cambios a partir de ese momento, si no que habían portado todo el histórico de cambios de los últimos 14 años.

2. Características

Git es un proyecto que ha crecido y ha añadido nuevas características, pero desde el inicio las más importantes fueron:

- Sistema **distribuido**, por lo que cada desarrollador tiene una copia completa de todo el histórico y los cambios sin necesidad de necesitar acceso a internet.
- Compatible con los sistemas y protocolos actuales, como HTTPS, FTP y SSH.
- Debe permitir **desarrollos no lineales**, donde se permita crear ramas y unirlas de manera rápida y eficiente.
- **Eficiente** con proyectos grandes y gran cantidad de ficheros y desarrolladores. Al final, el propósito inicial era usarlo para el kernel Linux donde había miles de ficheros y desarrolladores.
- Los identificadores de los commits están basados en criptografía (SHA1). De esta manera no puedan existir dos commits con el mismo ID, y un ID representa única y exclusivamente un commit.

3. Instalación

Git está presente en todos los sistemas operativos actuales. Dependiendo del sistema operativo que utilicemos, podremos instalarlo de distintas maneras. En la web oficial están las últimas versiones.

Podemos hacer uso de Git a través de sistemas de consola o de aplicaciones gráficas. Hoy en día los entornos de desarrollo más conocidos también lo tienen integrado, por lo que es posible hacer uso de Git desde ellos.

- Windows:
    - Git Bash

- MacOS:
  - MacOS tiene integrado Git dentro de las herramientas de desarrollador de Xcode. Para instalar únicamente Git desde un terminal debemos ejecutar: `xcode-select -install`
  - Para tener la última versión se recomienda usar Brew.sh e instalarlo a través de él.
- GNU/Linux:
  - Hoy en día todas las distribuciones tienen en sus repositorios Git, por lo que lo recomendable es hacer uso del sistema de instalación propio (apt, yast, ...). También es probable que ya esté instalado.

4. Primeros pasos

Una vez instalado, debemos realizar una pequeña configuración que después se utilizará cada vez que realicemos un commit: crear una identidad.

Añadiendo nuestro nombre y e-mail

```
ruben@vega:~$ git config --global user.name "Ruben Gomez"
ruben@vega:~$ git config --global user.email ruben@example.com
```

Esta configuración se guarda dentro de un fichero de configuración en la carpeta raíz de nuestro usuario. El fichero es `.gitconfig`. En este fichero podremos añadir configuraciones que se aplicarán a los comandos que realicemos.

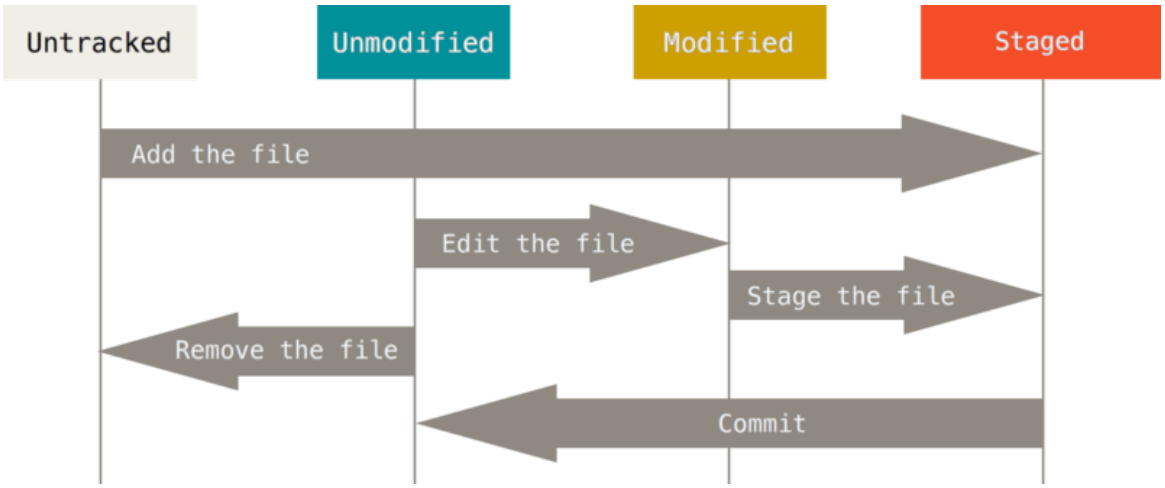
A nivel de configuración podemos llegar a tener configuraciones específicas globales, del sistema, del repositorio y del área de trabajo, pero no entraremos en ello.

5. Estado de los ficheros

Dentro del repositorio, los ficheros que vayamos creando y/o modificando pueden estar en distintos estados. Esto es lo que se denomina “ciclo de vida” o *lifecycle* de un fichero. Los ficheros pueden estar en los siguientes estados:

- **Sin seguimiento**: Es un fichero nuevo que no está en seguimiento por el sistema de control de versiones. Aunque se realicen cambios en él, no podremos volver a versiones previas. En caso de usar un repositorio remoto, este fichero no estará en él.
- **Con seguimiento**: Estos ficheros pueden estar en los siguientes estados:
  - **Sin modificar**: El fichero no ha sufrido modificaciones desde el último commit.
  - **Modificado**: El fichero tiene modificaciones.
  - **Preparado**: En inglés “staged”. Es un área donde se encuentran los ficheros que van a ir en el siguiente commit.

En esta imagen se puede ver cómo los ficheros pueden cambiar de estado, y desde qué estado pasar a otro:



Estado de los ficheros.Fuente.

Para entender esto mejor lo veremos a medida que vayamos haciendo uso de los comandos y creando/modificando ficheros.

6. Comandos básicos

Vamos a crear un repositorio para empezar a entender qué es lo que sucede con los ficheros que vamos creando en él, y tratar de entender los comandos más básicos.

6.1. Crear repositorio local

Vamos a crear un repositorio local dentro de un directorio nuevo, donde crearemos un proyecto que queremos versionar. Todo ello lo vamos a hacer dentro de un directorio nuevo llamado `pruebas`.

Crear un repositorio git

```
ruben@vega:~$ mkdir pruebas
ruben@vega:~$ cd pruebas
ruben@vega:~/pruebas$ git init
Inicializado repositorio Git vacío en /home/ruben/pruebas/.git/

ruben@vega:~/pruebas$ ls -a
.  ..  .git/
```

Tal como se puede ver, ejecutando `git init` dentro del directorio, nos inicializa el repositorio. Podemos ver que nos ha creado un directorio `.git`, que es un directorio oculto donde dentro se guarda la configuración y los commits que iremos realizando.

¡Cuidado!

No hagas cambios (ni borres nada) dentro del directorio .git

6.2. Crear primer commit

Con nuestro editor de textos favoritos, vamos a crear un fichero `README.md`. Normalmente este fichero es creado para indicar (en formato Markdown) información acerca del contenido del proyecto, qué es, para qué sirve, cómo compilarlo/usarlo...

Una vez creado el fichero vamos a entender qué es lo que está sucediendo dentro del repositorio:

Comprobar estado del repositorio

```
ruben@vega:~/pruebas$ git status
En la rama main
No hay commits todavía

Archivos sin seguimiento:
(usa "git add <archivo>..." para incluirlo a lo que será confirmado)
README.md

no hay nada agregado al commit pero hay archivos sin seguimiento
presentes (usa "git add" para hacerles seguimiento)
```

Vemos que el estado del único fichero que hemos creado es “sin seguimiento” (tal como hemos explicado en el punto anterior). Es momento de pasar nuestro fichero a estado “preparado”, para ello:

Preparamos el fichero para hacer commit

```
ruben@vega:~/pruebas$ git add README.md
ruben@vega:~/pruebas$ git status
En la rama main
No hay commits todavía

Cambios a ser confirmados:
(usa "git rm --cached <archivo>..." para sacar del área de stage)
nuevos archivos: README.md
```

El fichero `README.md` está en el “stage area”, por lo que ahora es el momento en el que podemos hacer nuestro primer commit con las modificaciones realizadas:

Hacemos el commit

```
ruben@vega:~/pruebas$ git commit -m "Primer commit"
[main (commit-raíz) 45900ae] Primer commit
1 file changed, 3 insertions(+)
create mode 100644 README.md

guruben@vega:~/pruebas$ git status
En la rama main
nada para hacer commit, el árbol de trabajo está limpio
```

Con `git commit -m ``Primer commit''` lo que estamos es "encapsulando" todas las modificaciones de todos los ficheros que están en el área "stage" (en este caso un único fichero), y vamos a generar un commit al que le hemos puesto el mensaje "Primer commit".

6.3. Ver histórico de cambios

Si realizamos varios cambios al fichero, o si añadimos un fichero nuevo y realizamos una serie de commits, nos puede interesar saber cómo está yendo el histórico de commits.

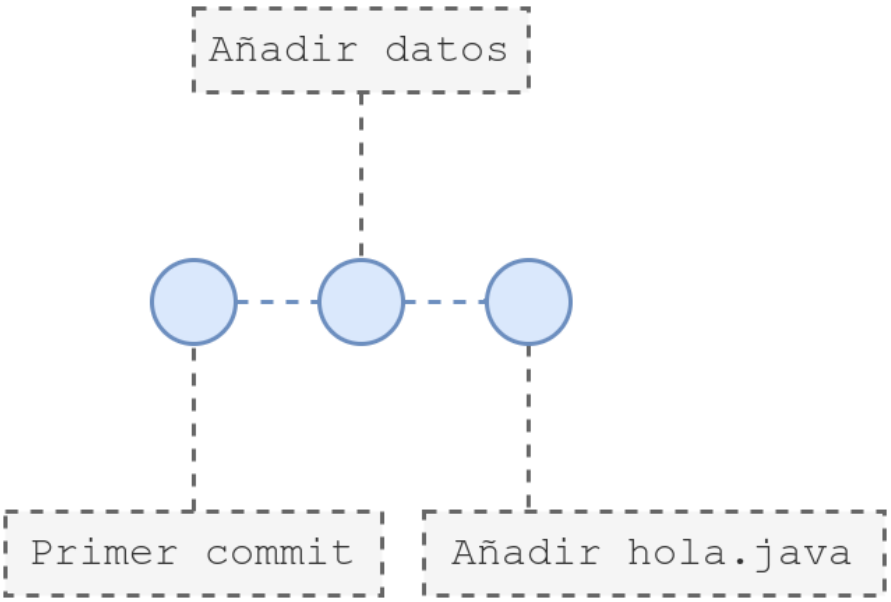
Para ello podemos hacer uso del comando `git log`, que nos mostrará en orden por fecha descendente, todos los commits que ha tenido nuestro repositorio:

Ver histórico de commits

```
ruben@vega:~/pruebas$ git log

commit ac00ce47dcdcda01bf33d162890bd98cc4f36ead (HEAD -> main)
Author: Rubén Gómez <ruben@example.com>
Date:   Sun Sep 17 10:52:26 2023 +0200

Añadir hola.java

commit a3f9554e0917dfdd2ce6ccccb5957d44d63c7f6f
Author: Rubén Gómez <ruben@examplel.com>
Date:   Sun Sep 17 10:52:03 2023 +0200

Añadir datos a README

commit 45900aef0300bf88c3a2939b8cf2f6b05de572dc
Author: Rubén Gómez <ruben@example.com>
Date:   Sun Sep 17 10:46:12 2023 +0200
```

```
Primer commit
```

De manera gráfica, el histórico de los *commits* podríamos representarlo de la siguiente manera, empezando por la izquierda el commit más antiguo:



Estado tras varios commits

6.4. Ver diferencias

Si estamos realizando modificaciones y queremos saber qué modificaciones hemos realizado en los ficheros, podemos realizar lo siguiente:

Ver cambios locales

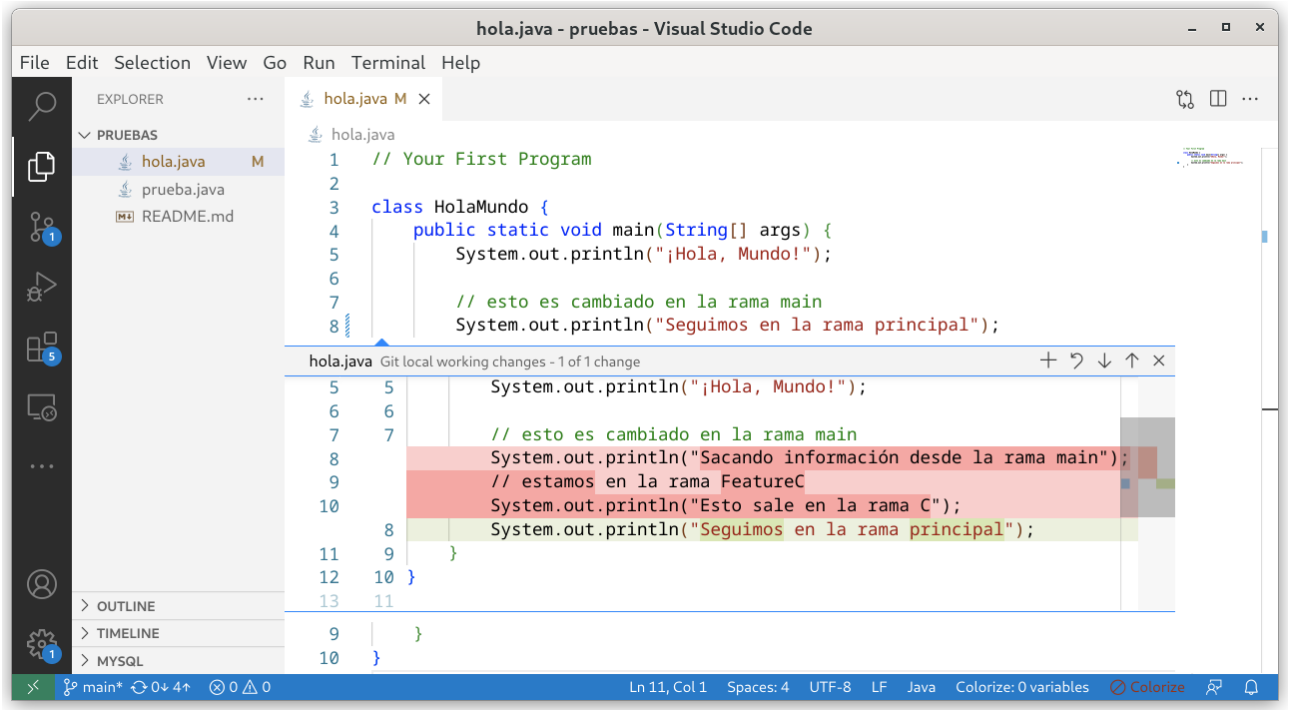
```
ruben@vega:~/pruebas$ git diff
```

Y obtendríamos algo como esto, donde las líneas que empiezan con "**-**" han sido borradas y las que empiezan con "**+**" son añadidos.

```
diff --git a/hola.java b/hola.java
index 903bffe..3e46c29 100644
--- a/hola.java
+++ b/hola.java
@@ -5,8 +5,6 @@ class HolaMundo {
         System.out.println("¡Hola, Mundo!");

         // esto es cambiado en la rama main
-        System.out.println("Sacando información desde la rama main");
-        // estamos en la rama FeatureC
-        System.out.println("Esto sale en la rama C");
+        System.out.println("Seguimos en la rama principal");
     }
 }
```

Cambios respecto al commit anterior

Esta diferencia se aprecia de la siguiente manera en entornos de desarrollo como Visual Studio Code.



Diff en Visual Studio Code

III

GitHub como servidor remoto

1. Usar GitHub como repositorio remoto

GitHub es una web donde podemos crear repositorios y usarlo como sistema centralizado de nuestros proyectos.

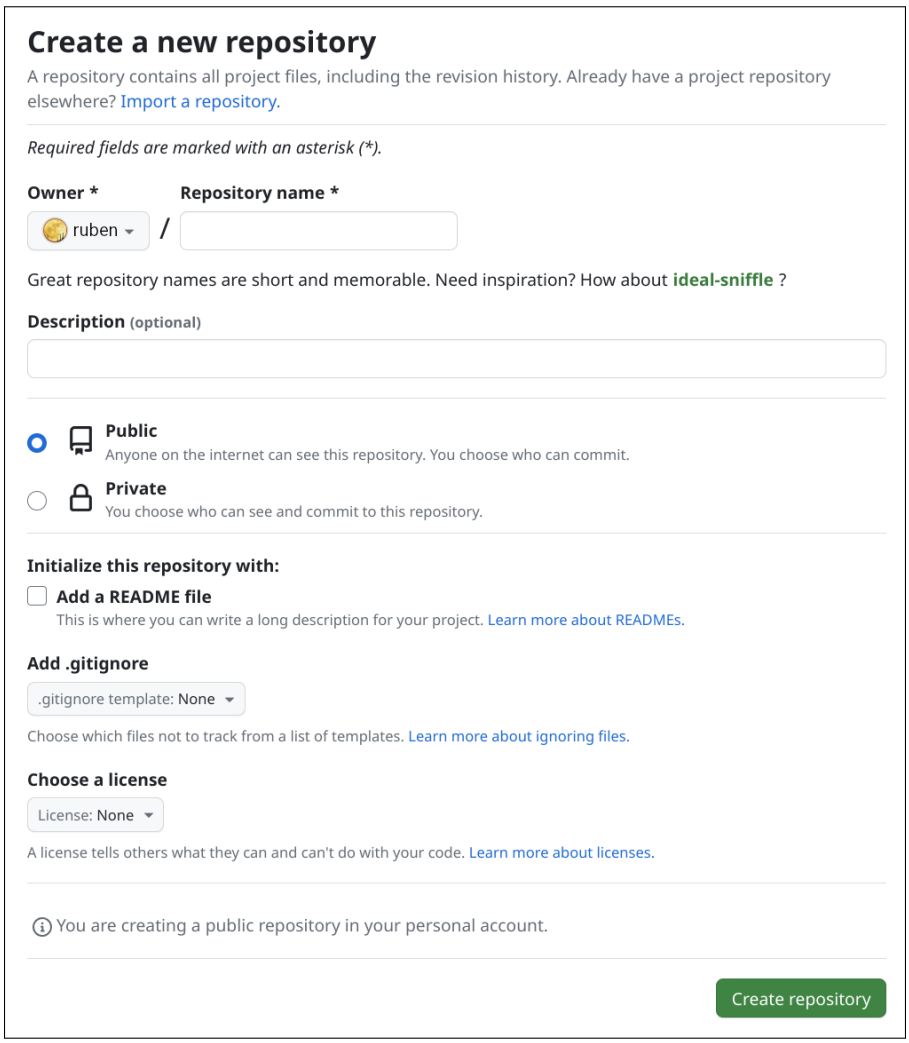
Entre las característica que tiene, se pueden destacar:

- Entorno gráfico para controlar el repositorio. Se puede ver el histórico del repositorio, quién ha realizado los cambios, cuándo, ramas creadas...
- Control de incidencias. Para poder crear "*issues*" del proyecto a medida que encontremos errores.
- Generar documentación por proyecto en formato Wiki.
- Gestión de "*pull requests*" para integrar cambios en la rama principal.
- Sistema de acciones automatizadas, que ayudan para el sistema de CI/CD. Con estas acciones podemos generar "*releases*", compilar el código y comprobar si hay errores, pasar tests, ... Hay mucha documentación al respecto.

1.1. Crear repositorio

Una vez hemos creado una cuenta, podremos crear un nuevo repositorio en la plataforma. Al crearlo, podemos elegir distintas configuraciones:

- **Nombre del repositorio**, para poder acceder a él. Es recomendable darle un nombre significativo al proyecto.
- **Descripción**, donde podremos indicar un poco de texto para entender de qué trata el proyecto.
- **Visibilidad**. Podemos hacer que el repositorio sea **público** (cualquier persona puede ver el contenido del proyecto) o **privado** (sólo nuestro usuario puede verlo).
- **Inicializar el proyecto con**: Podemos hacer que cuando Github inicialice el proyecto le añada ciertos ficheros:
  - **README**: Fichero donde indicar de qué trata el fichero, cómo compilarlo, ...
  - **.gitignore**: Un fichero que nos permite ignorar ficheros dentro de nuestro "área de trabajo". Podemos elegir de una plantilla para distintos lenguajes de programación.
  - **Licencia**: Podemos elegir entre distintas licencias libres para nuestro proyecto.



Opciones al crear un nuevo repositorio en GitHub

En este caso se va a crear un repositorio público llamado **pruebas**, sin ningún tipo de fichero. De esta manera "enlazaremos" el repositorio local de los pasos anteriores con este repositorio.

2. Enlazar repositorio local con remoto

GitHub nos muestra cuáles son los pasos para enlazar un repositorio existente con el que acabamos de crear en su plataforma, a través de la línea de comandos:

Enlazando repositorio local con uno en GitHub

```
ruben@vega:~/pruebas$ git remote add origin git@github.com:yuki/pruebas.git
ruben@vega:~/pruebas$ git branch -M main
ruben@vega:~/pruebas$ git push -u origin main
```

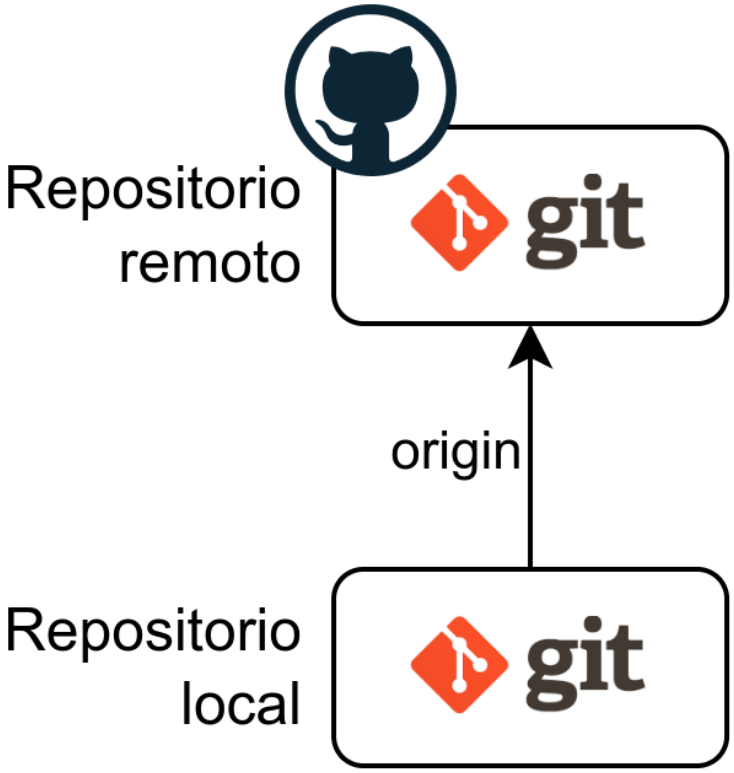
Vamos a tratar de entender qué es lo que hace cada uno de los comandos, ya que es importante.

- `git remote add origin git@github.com:yuki/pruebas.git`

  Con este comando se añade un repositorio remoto con el nombre "**origin**". Básicamente estamos diciéndole al repositorio local que cuando realicemos algo sobre el repositorio remoto "origin" haga

uso de esa URL.

El nombre "origin" se puede cambiar, pero es el nombre que han decidido estandarizar:



Enlazamos repositorio local con remoto de nombre "origin"

- `>_ git branch -M main`

  Este comando lo que hace es cambiar el nombre de la rama principal para que se llame "**main**. Originalmente la rama principal se llamaba "master", pero en 2020 decidieron cambiarlo a"main".

  En las nuevas versiones de git "main" ya es el nombre por defecto, por lo que este comando puede no ser necesario (si se ejecuta no hace nada).

- `>_ git push -u origin main`

  Este comando se puede separar en dos partes:

  - `>_ git push`

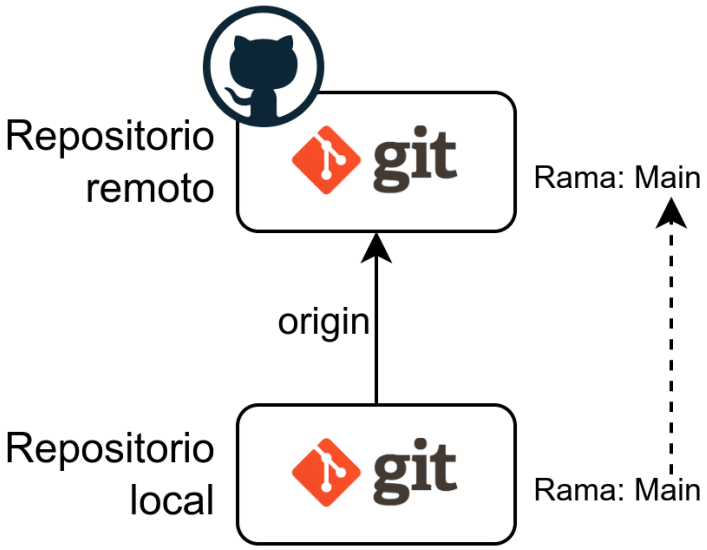
    Este es el comando que utilizaremos para enviar al repositorio remoto los commits que hemos realizado en el repositorio local.

  - `>_ -u origin main`

    Estos parámetros sólo los usaremos para realizar el primer envío. Estos indican al repositorio local que haga uso del servidor remoto "origin" para enlazar la rama en la que nos encontramos ("main") con la rama remota "main" a la hora de enviar las revisiones.

¡Atención!

Los nombres de las ramas no tienen por qué coincidir, pero que sean iguales nos va a facilitar identificar ambas.



Enlazamos rama local con rama remota

Al ejecutar el último comando, se va a tratar de enviar información a GitHub, por lo que deberemos hacer uso de un sistema de autenticación para asegurar que tenemos acceso al repositorio.

2.1. Sistemas de autenticación con GitHub

Para asegurar que quien manda información al servidor remoto es quien dice ser, GitHub cuenta distintos sistemas de autenticación.

A continuación se detallan los utilizados para línea de comandos

2.1.1. Autenticación SSH (certificados de clave pública/clave privada)

A la hora de utilizar la conexión/autenticación SSH con GitHub, para que no nos pida el usuario y la contraseña es **hacer uso de los certificados de clave pública y clave privada**. Este concepto de "clave pública y clave privada" viene de la **criptografía asimétrica**.

Este sistema de criptografía asimétrica hace uso de dos claves que están asociadas entre sí:

- **Clave privada**: Es la base del sistema criptográfico, y como su nombre indica, se debe de mantener en privado. **Nunca se debe de compartir**, ya que entonces se podrían hacer pasar por nosotros.
- **Clave pública**: Asociada a la clave privada, la clave pública puede ser compartida y enviada a otros ordenadores para poder realizar la conexión.

Para generar el par de claves se realiza con el siguiente comando (**que funciona tanto en Windows como en Linux**):

Crear par de claves pública/privada

```
ruben@vega:~$  ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/ruben/.ssh/id_rsa):

Enter passphrase (empty for no passphrase):
Enter same passphrase again:

Your identification has been saved in /home/ruben/.ssh/id_rsa
Your public key has been saved in /home/ruben/.ssh/id_rsa.pub
The key fingerprint is:
SHA256:SPqPOYBmPb8PCFhcZgqcWZPZzaL5RNfMeKmHqebvC7E ruben@vega
The key's randomart image is:
+---[RSA 3072]----+
|o +oB o = .      |
| * B.+ = *       |
|  + + + =        |
| o o + = .       |
|. .o+.o S        |
|  +.+*o          |
| o  +Eo          |
|     .+=         |
|      *B+        |
+----[SHA256]-----+
```

El comando muestra los siguientes pasos:

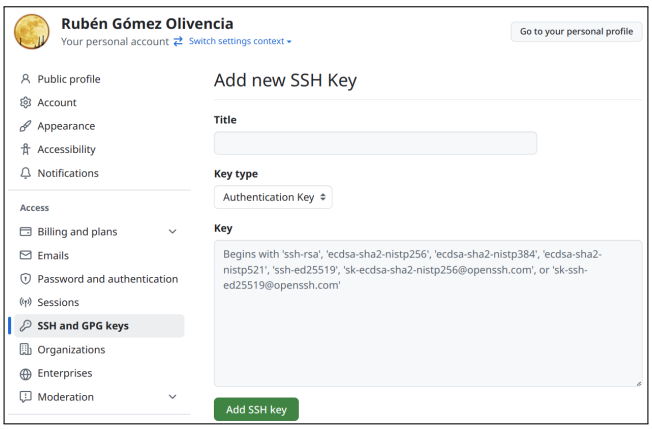
1. Creación de la pareja de claves pública/privada haciendo uso, en este caso, del sistema criptográfico **RSA**. En otros caso puede ser Ed25519.

2. Lugar donde se va a guardar la clave privada. En este caso en `.ssh/id\_rsa`.

3. Contraseña para securizar la clave privada. De esta manera, para poder usarla habrá que introducir dicha contraseña. Dado que nosotros queremos evitar introducir contraseñas, lo dejaremos en blanco.

4. Lugar donde se va a guardar la clave pública. En este caso en `.ssh/id\_rsa.pub`.

Una vez tenemos nuestro par de claves, podemos copiar la clave pública en nuestro perfil de GitHub, apartado “SSH and GPG Keys”.

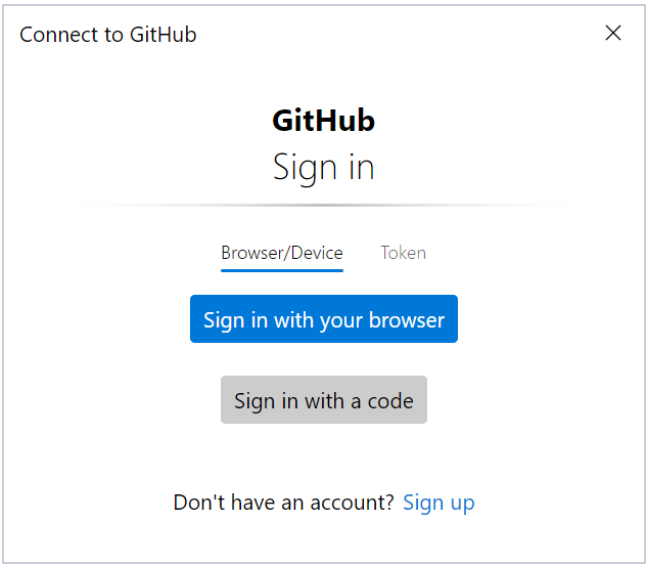


Añadir clave pública en GitHub

De esta manera, ya podremos hacer uso del protocolo SSH para el sistema de autenticación contra GitHub.

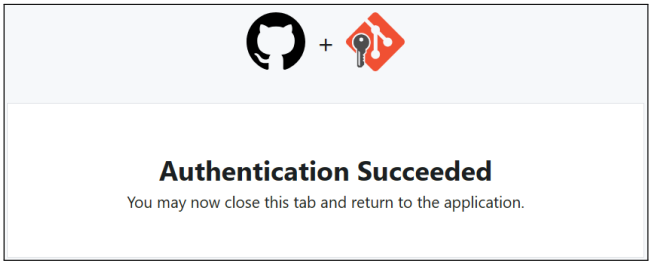
2.1.2. Añadiendo credenciales de acceso en Windows

En el caso no utilizar el sistema de autenticación anterior, nos aparecerá una ventana para que realicemos el login de usuario.



Ventana para realizar el login en Github

Elegiremos la opción marcada en azul, lo que nos abrirá el navegador web para realizar el login en la web de GitHub. Una vez hayamos introducido bien los credenciales de acceso, veremos la confirmación. Esto nos creará una autenticación "OAuth de aplicación" en nuestro perfil de GitHub (Usuario → Settings → Applications), pestaña "*Authorized OAuth Apps*".



Login realizado correctamente

Y para que siga funcionando, en el **Administrador de Credenciales** de Windows ("Panel de Control → Cuentas de Usuario → Administrador de Credenciales → Credenciales de Windows"), también nos creará una entrada nueva de credenciales genéricas:

Credenciales de GitHub en Windows

Una vez realizados estos pasos, nos funcionará el comando y enlazará la rama y enviará los commits realizados al servidor remoto.

Enlazando la rama y enviando los cambios

```
ruben@vega:~/pruebas$ git push -u origin main
Enumerando objetos: 10, listo.
Contando objetos: 100% (10/10), listo.
Compresión delta usando hasta 6 hilos
Comprimiendo objetos: 100% (7/7), listo.
Escribiendo objetos: 100% (10/10), 889 bytes | 222.00 KiB/s, listo.
Total 10 (delta 2), reusados 0 (delta 0), pack-reusados 0
remote: Resolving deltas: 100% (2/2), done.
To github.com:yuki/pruebas.git
* [new branch]      main -> main
rama 'main' configurada para rastrear 'origin/main'.
```

3. Enviar modificaciones locales

A partir de ahora, cualquier modificación que hayamos realizado en local **deberemos enviarla al servidor remoto**. No tenemos por qué hacerlo por cada commit, ya que cuando realicemos el envío se enviarán todos los que no estén en remoto.

Enlazando la rama y enviando los cambios

```
ruben@vega:~/pruebas$ git push
git push
Enumerando objetos: 5, listo.
Contando objetos: 100% (5/5), listo.
Compresión delta usando hasta 6 hilos
Comprimiendo objetos: 100% (3/3), listo.
Escribiendo objetos: 100% (3/3), 324 bytes | 324.00 KiB/s, listo.
Total 3 (delta 1), reusados 0 (delta 0), pack-reusados 0
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To github.com:yuki/pruebas.git
cefb314..2cac944  main -> main
```

4. Clonar repositorio remoto

Imaginemos que una vez subido los cambios locales a GitHub queremos hacer uso del repositorio en otro ordenador. Para ello, debemos realizar un “clonado” del repositorio en cuestión.

Clonar repositorio remoto en repositorio local

```
ruben@vega:~/pruebas$ git clone https://github.com/yuki/pruebas.git
Clonando en 'pruebas'...
remote: Enumerating objects: 20, done.
remote: Counting objects: 100% (20/20), done.
remote: Compressing objects: 100% (12/12), done.
remote: Total 20 (delta 4), reused 20 (delta 4), pack-reused 0
Recibiendo objetos: 100% (20/20), listo.
Resolviendo deltas: 100% (4/4), listo.
```

5. Obtener últimos commits

Si alguien ha realizado commits en nuestro repositorio (o los hemos realizado nosotros desde otro ordenador), es posible que nuestro repositorio local no esté actualizado. Para actualizarlo tenemos que entender dos comandos.

5.1. git fetch

`git fetch` obtiene los commits del repositorio remoto, **pero no los aplica sobre nuestra copia de trabajo actual**. De esta manera, no se aplican los cambios y mientras tanto podemos seguir trabajando.

Obtener últimos cambios

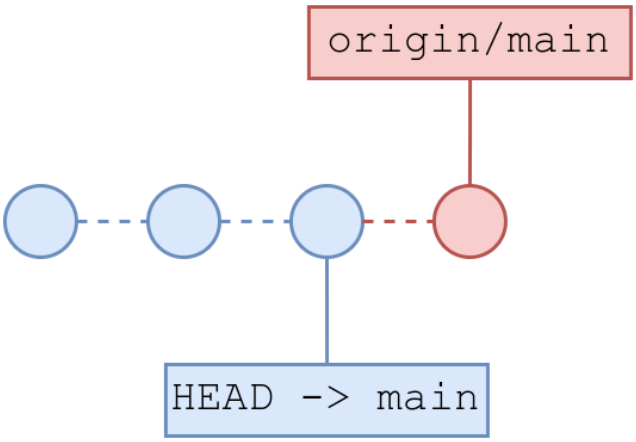
```
ruben@vega:~/pruebas$ git fetch
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 3 (delta 0), pack-reused 0
Desempaquetando objetos: 100% (3/3), 326 bytes | 27.00 KiB/s, listo.
Desde github.com:yuki/pruebas
2cac944..97f5359  main       -> origin/main

ruben@vega:~/pruebas$ git status
En la rama main
Tu rama está detrás de 'origin/main' por 1 commit, y puede ser
avanzada rápido. (usa "git pull" para actualizar tu rama local)
```

Tal como se puede ver el primer comando nos descarga objetos nuevos, y al ver el estado nos avisa que **nuestra rama está por detrás de "origin/main"** (la rama remota). También podemos ver todos los commits de la siguiente manera:

Ver logs de los commits

```
ruben@vega:~/pruebas$ git log --all
commit 97f5359058551cfbe1d61c7b3db1bd648ac496ba (origin/main)
Author: Rubén Gómez <pruebas@example.com>
Date:   Sun Sep 17 19:40:18 2023 +0200

Pequeño cambio en el README

commit 2cac94467ca7c50e2cf2125ce11e21eda8461cc1 (HEAD -> main)
Author: Rubén Gómez <pruebas@example.com>
Date:   Sun Sep 17 18:40:37 2023 +0200

Añadiendo "adios" en hola.java

commit cefb3143e3f298dc8fe200d6a2804165f58bab69
Author: Rubén Gómez <pruebas@example.com>
Date:   Sun Sep 17 18:40:02 2023 +0200

Corregido error en hola.java
```

El primer commit nos indica que está en **origin/main**, la rama del repositorio remoto, mientras que el segundo nos aparece "**HEAD -> main**", que es la copia de trabajo local. El resumen es el siguiente.

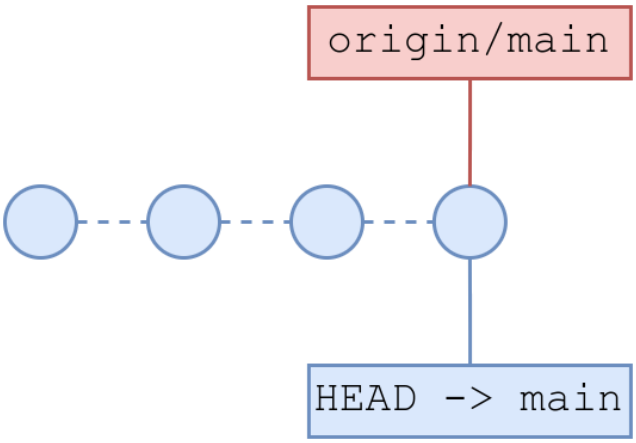


Estado de los commits tras el "fetch"

5.2. git pull

En este caso se obtienen los commits del repositorio **y se aplican sobre la rama de trabajo actual**. Si tenemos cambios realizados en local, los cambios que obtenemos del repositorio **pueden entrar en conflicto con lo que tenemos**. Más adelante hablaremos de ello.

De no existir conflictos, los cambios se aplican y el estado quedaría en ambos repositorios en el mismo punto exacto:



Estado de los commits tras el "pull"

IV

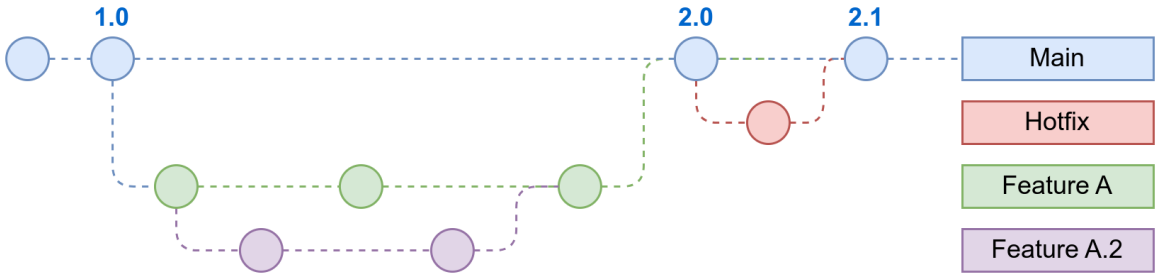
Ramas, merges y conflictos

1. Usar ramas en git

La creación de ramas (llamadas ***branches***) en un repositorio nos permite realizar pruebas, añadir características nuevas, o cambiar de ámbito sin perjudicar el flujo de trabajo principal.

Una rama es una bifurcación del camino principal del desarrollo de una aplicación (o de un commit concreto). Esta rama puede ser una rama pública (existir en GitHub) o ser privada (sólo existir en nuestro repositorio local).

La creación de ramas en Git es instantáneo (al contrario de lo que sucedía con sistemas anteriores), por lo que crearla no supone un esfuerzo ni una pérdida de tiempo para el desarrollo.



Un desarrollo con ramas

En el dibujo aparecen ramas que posteriormente se unen a la rama *main* principal, pero esto no tiene por qué ser así, y una rama puede mantener un desarrollo paralelo y nunca unirse.

1.1. Crear rama

Para crear una rama en el desarrollo, desde el punto en el que nos encontramos, se puede hacer de dos maneras. El resultado es el mismo, pero conviene entender qué es lo que sucede en ambos casos.

- **Crear rama y luego movernos a ella**: Este caso consta de dos pasos.

Crear rama “featureA” y movernos a ella

```
ruben@vega:~/pruebas$ git branch featureA
ruben@vega:~/pruebas$ git switch featureA
Cambiado a rama 'feature1'

ruben@vega:~/pruebas$ git log
commit 170f9ce8c214b82f... (HEAD -> featureA, origin/main, main)
Author: Rubén Gómez <ruben@example.com>
Date:   Sun Sep 17 18:40:37 2023 +0200
...
```

Tal como se puede ver, se ha creado la rama con nombre “featureA”, para posteriormente con el comando `git switch featureA` cambiarnos a dicha rama.

Con `>_ git log` podemos comprobar cómo en ese *commit* ahora mismo se encuentran tres puntos de nuestro sistema de repositorios:

- **HEAD -> featureA**: que es la rama en la que nos encontramos ahora.
- **origin/main**: la rama "main" del repositorio remoto.
- **main**: la rama local "main".

Los tres puntos coinciden porque no se han realizado todavía ningún cambio en ninguna rama.

- **Crear rama y movernos a ella automáticamente**: en este caso los dos pasos se convierten en uno, pero el resultado es el mismo.

>_ Crear rama "featureB" y movernos a ella directamente

```
ruben@vega:~/pruebas$ git switch -c featureB
Cambiado a nueva rama 'featureB'
```

1.2. Cambiar entre ramas

Ahora que ya sabemos cómo crear ramas, hay que entender cómo podemos cambiar entre ellas, aunque el comando lo acabamos de ver en el punto anterior: `>_ git switch branch`, donde "branch" es el nombre de la rama a la que queremos ir.

Siguiendo con el ejemplo anterior, si queremos volver a la rama "main", deberíamos hacer:

>_ Volver a la rama "main"

```
ruben@vega:~/pruebas$ git switch main
Cambiado a rama 'main'
Tu rama está actualizada con 'origin/main'.
```

Si queremos volver a la rama "featureA":

>_ Volver a la rama "featureA"

```
ruben@vega:~/pruebas$ git switch featureA
Cambiado a rama 'featureA'
```

1.3. Ver estado de las ramas

Para comprobar cuál es el estado de las ramas, vamos a realizar dos *commits* distintos en la rama "main" y otro en la rama "featureA". Para ver cómo se encuentra ahora el estado de nuestro repositorio, podemos hacer uso del siguiente comando:

>_ Volver a la rama "featureA"

```
ruben@vega:~/pruebas$ git log --oneline --decorate --graph --all
```

Obtendríamos el siguiente resultado:

```
* 1e30067 - (featureA) Añadida featureaA
| * 4218bca - (HEAD -> main) Nueva función
| * e98a802 - Pequeños cambios en hola.java
|/
* 170f9ce - (origin/main, featureB) Nuevos añadidos a README
* cefb314 - Modificaciones
* ac00ce4 - Añadimos hola.java
* a3f9554 - Añadimos datos a README
* 45900ae - Primer commit
```

Ramas actuales

Tal como se puede ver, la bifurcación sucede en el commit “170f9ce”, que es donde está situado el último commit del servidor remoto y la rama “featureB” creada previamente (y que no se ha tocado).

Por otro lado, desde ese punto surgen dos ramas:

- **featureA**, con el único commit 1e30067.
- **main**, que tiene 2 commits.

Se puede también ver que existe una rama “featureB” que se mantiene en el mismo punto que antes, ya que no se ha decidido añadir nada todavía en esa rama. Y ese mismo punto es el que coincide con el repositorio remoto “origin/main”.

Tras realizar estas modificaciones, a continuación vamos a ver cómo podemos fusionar los cambios de una rama en la otra.

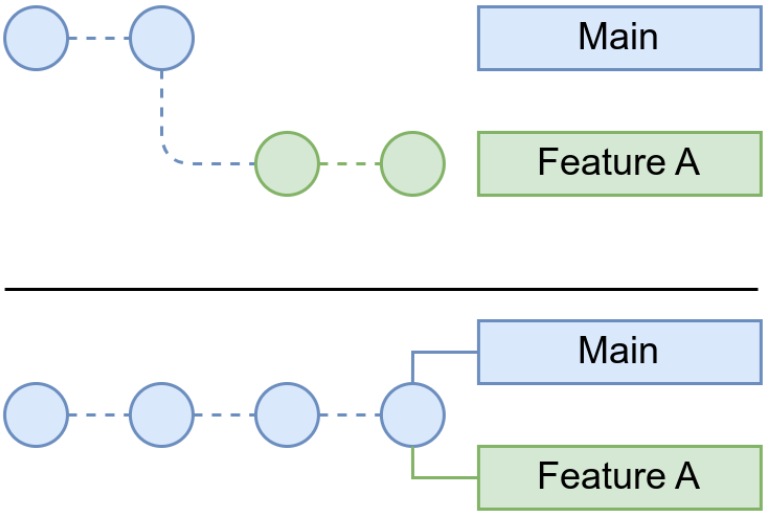
2. Fusionar ramas

La fusión de ramas (en inglés ***merge***) sucede cuando queremos obtener los cambios realizados en una rama y fusionar dichos cambios con la rama en la que nos encontramos actualmente. Ahora que tenemos distintas ramas creadas, con sus correspondientes commits, es buen momento para realizar la fusión de las ramas.

Información

Merge es coger los cambios de una rama y fusionarlos con la actual.

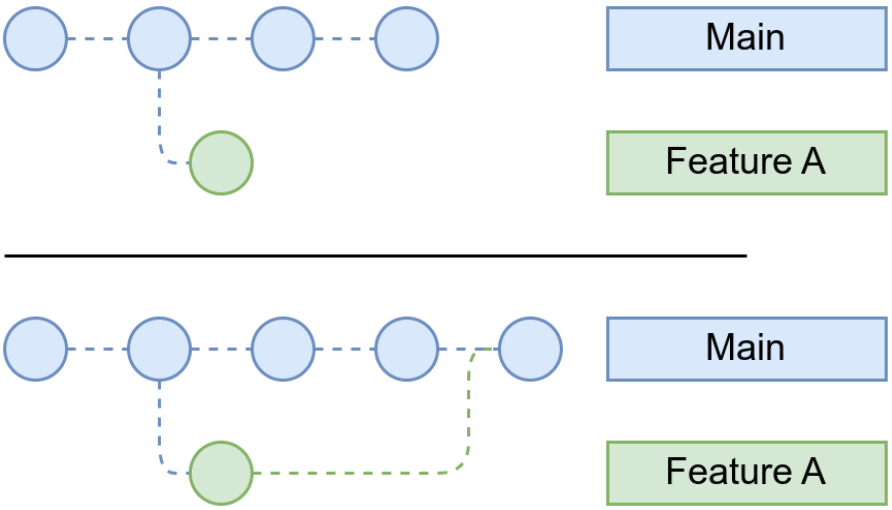
Dependiendo de cómo haya sido el desarrollo de las ramas, la fusión podrá terminar en un “dibujo” distinto. Por ejemplo, el caso más sencillo es que en la bifurcación sólo la rama nueva tiene nuevos commits, por lo que al realizar la fusión quedaría:



Merge de rama con commits sobre rama “main”

Tras realizar el merge, en este caso es como si la rama “FeatureA” no hubiese existido.

En cambio, si en la rama main, tal como se ha sugerido en el punto anterior, también se realizan modificaciones, el dibujo quedará tal que así:



Merge donde hay commits en ambas ramas

Para realizar la fusión, debemos seguir estos pasos:

- Colocarnos en la rama en la que queremos fusionar los cambios de otra rama. Normalmente, nos va a interesar añadir los cambios a la rama “main”.
- Realizar la fusión.

Volver a la rama “main”

```
ruben@vega:~/pruebas$ git switch main
ruben@vega:~/pruebas$ git merge featureA -m "Merge de FeatureA en main"
```

De esta manera, crearemos un nuevo commit con el texto “Merge de FeatureA en main”, que indicará la fusión de la rama “featureA” sobre la rama Main. El dibujo en la vida real queda de la siguiente manera:

```
*   053435d - (HEAD -> main) Merge de FeatureA en main
|\
| * 1e30067 - (featureA) Añadida featureaA
* | 4218bca - Nueva función
* | e98a802 - Pequeños cambios en hola.java
|/
* 170f9ce - (origin/main, featureB) Nuevos añadidos a README
* cefb314 - Modificaciones
```

Merge donde hay commits en ambas ramas

Una vez realizado el *merge*, podemos subir los cambios al repositorio central. La rama “featureA” es una rama local, por lo que a nivel de GitHub esa rama nunca ha existido, aunque podemos ver en el interfaz web que el gráfico sí ha sufrido una ramificación:

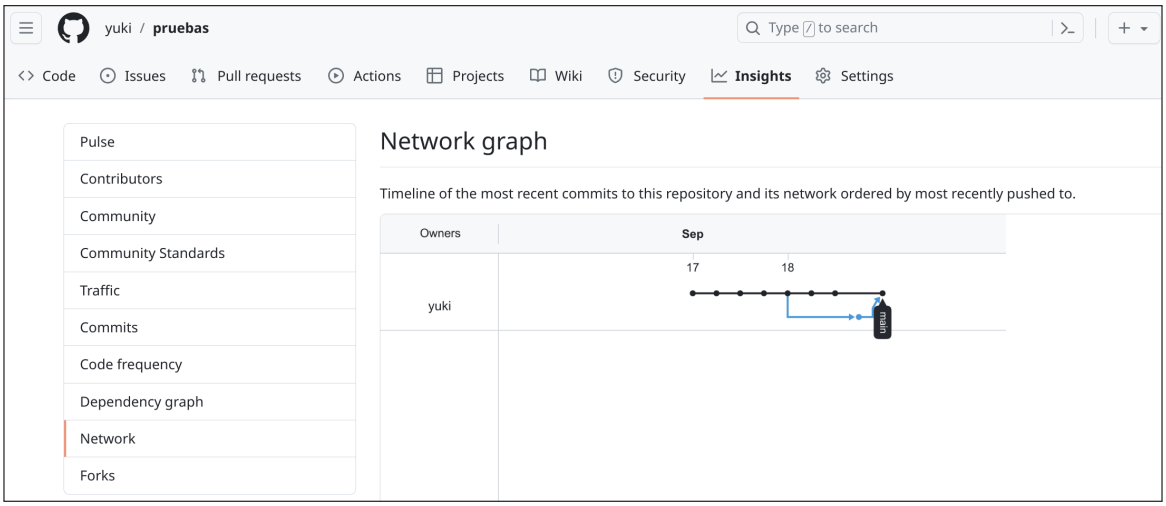


Gráfico en el interfaz de GitHub

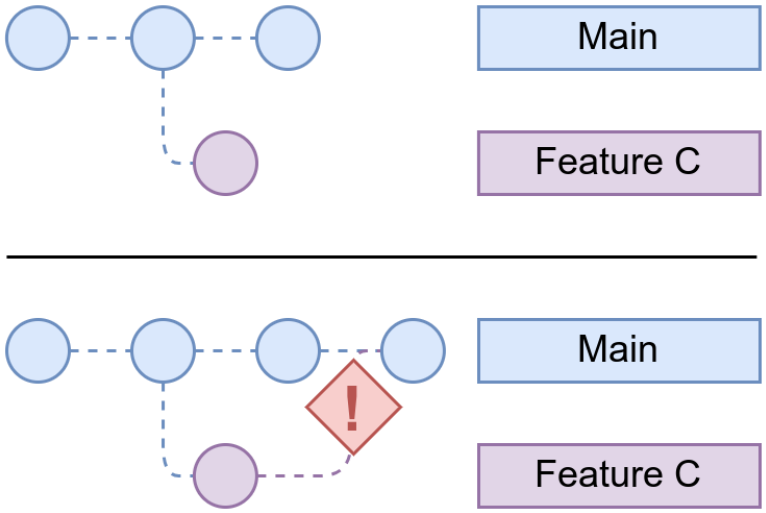
3. Resolver conflictos en un *merge*

Cuando se realiza un *merge* puede existir la posibilidad de crear un conflicto. Un conflicto sucede cuando al fusionar dos ramas en ambas se ha tocado las mismas líneas, y git no sabe qué hacer con los cambios. Para evitar problemas, se genera un conflicto que debe ser resuelto por el desarrollador.

Información

Un conflicto surge al fusionar dos ramas que tiene cambios en la misma porción de código. El desarrollador es el encargado de arreglar el conflicto.

Pongamos como ejemplo un caso similar al apartado anterior. Se ha creado la rama “featureC” y se ha modificado un par de líneas de una función y se ha hecho commit. En la rama “main” se ha hecho lo mismo.



Merge con conflicto

A la hora de ir a realizar el *merge*, git nos avisa que existe un conflicto en el fichero `hola.java` y que debemos resolverlo:

Volver a la rama "main"

```
ruben@vega:~/pruebas$ git merge FeatureC
Auto-fusionando hola.java
CONFLICTO (contenido): Conflicto de fusión en hola.java
Fusión automática falló; arregle los conflictos y luego realice
 un commit con el resultado.
```

Para resolver el conflicto deberemos editar el fichero que nos indica. En este caso, si lo editamos, veremos lo siguiente:

Editando el fichero con conflicto

```
class HolaMundo {
    public static void main(String[] args) {
        System.out.println("¡Hola, Mundo!");

        <<<<<<< HEAD
        // esto es cambiado en la rama main
        System.out.println("Sacando información desde la rama main");
        =======
        // estamos en la rama FeatureC
        System.out.println("Esto sale en la rama C");
        >>>>>>> FeatureC
    }
}
```

Tal como se puede ver, es una clase con una función en Java simulando el típico programa “Hola Mundo”, pero **aparecen una serie de líneas que nos están indicando dónde está el conflicto**:

- <<<<<< **HEAD**: Comienzan la parte en la que hay conflicto de la rama en la que nos encontramos. En este caso, la rama “main”.
- **=======**: Es la separación de los apartados que entran en conflicto.
- >>>>>> **FeatureC**: Es el final de la parte que entra en conflicto. Desde el punto anterior hasta este, en este caso, es de la rama “FeatureC”.

Para resolver el conflicto deberemos borrar esas líneas especiales, quedarnos con las partes del código que nos interesen y realizar un nuevo commit. Es por eso que para la resolución de los conflictos quizá tengan que participar los desarrolladores que han realizado los cambios que han creado el conflicto.

Información

Para resolver el conflicto deberemos borrar esas líneas especiales, quedarnos con las partes del código que nos interesen y realizar un nuevo commit

```
*   88d8c9d (HEAD -> main) Merge branch 'FeatureC'
|\
| * a0c0ab2 (FeatureC) Modificado desde featureC
* | 21cd5d9 Modificado hola mundo
|/
* e388fd4 Creando hola mundo
```

Merge con conflicto resuelto