UNIDAD DIDÁCTICA 8: Control de versiones

Contenido

En esta unidad aprenderás...

- Qué es un sistema de control de versiones
- Las diferencias entre un sistema de control de versiones centralizado y uno distribuido
- Qué es Git
- Los posibles estados de un fichero en un repositorio Git
- Qué es GitHub
- Cómo usar Git desde la terminal
- Crear un nuevo proyecto en GitHub
- Trabajar desde tu ordenador con un proyecto subido a GitHub
- Especificar que ficheros en local deberán ser ignorados por el servidor remoto
- Sincronizar el estado de tu proyecto en local con el estado en remoto
- Sincronizar el estado del proyecto en remoto con los cambios realizado en local
- Qué es una rama

- Cuántas ramas debería tener un proyecto y la función de cada una de ellas
- Cómo volver a una versión anterior del proyecto
- Cómo trabajar en grupo utilizando Git
- Qué es un fork y cómo hacerlos
- Qué es un pull request y cómo hacerlos

Temporización

Clase 1 → ¿Qué es un SCV? ¿Cuáles son los más usados?

Clase 2 → Uso de GitHub Desktop

Clase 3 → Uso de GitHub desde la terminal

Clase 4 → Uso avanzado de GitHub

Clase 5 → Trabajando en grupo

Clase 6 → fork, pull request, página web personal en GitHub y conclusiones

Clase 7 → Examen

Control de versiones

Un programa de control de versiones es aquel que permite monitorizar los cambios que se producen en un conjunto de ficheros. En principio los programas de control de versiones se asocian al control de cambios en ficheros de código fuente, pero su uso se puede llegar a extrapolar para el control de ficheros relativos a la documentación o incluso a ficheros multimedia.

Algunas de las funciones más relevantes de los programas de control de versiones:

- Tener una copia de seguridad del código del proyecto.
- Tener un histórico de los cambios realizados a los ficheros.
- Trabajar con un proyecto desde distintos equipos y que el código esté sincronizado en todos los equipos.
- Revertir un fichero o proyectos enteros, a un punto pasado determinado. Útil cuando tras algunos cambios el proyecto deja de funcionar.
- Si un fichero se pierde o se corrompe se puede recuperar volviendo a un estado anterior.

Hemos expuesto las ventajas que un sistema de control de versiones cuando trabajas tu solo con código tuyo, pero sin duda el principal objetivo de un programa de control de versiones es el de permitir a un grupo de personas trabajar en un mismo proyecto solucionando los problemas derivados del hecho de que varias personas estén trabajando a la vez sobre los mismos ficheros.

Control de versiones centralizado

Hay un único servidor que alberga el histórico del proyecto y el resto de clientes del sistema tienen el último estado del proyecto.

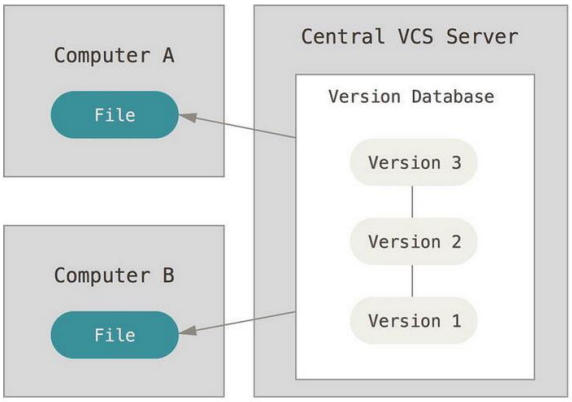


Ilustración 1. Esquema de un sistema de control de versiones centralizado

Este tipo de sistemas han sido los utilizados tradicionalmente, sin embargo tienen el problema de que un fallo en servidor central inhabilita al resto de clientes para trabajar.

Control de versiones distribuido

Nacen con el objetivo de solventar los problemas de los sistemas centralizados. Los clientes no sólo tienen la última versión del proyecto sino que disponen, en local, del histórico completo del proyecto, de esta forma aunque el servidor central falle el sistema puede volver a restituirse a partir de cualquier nodo.

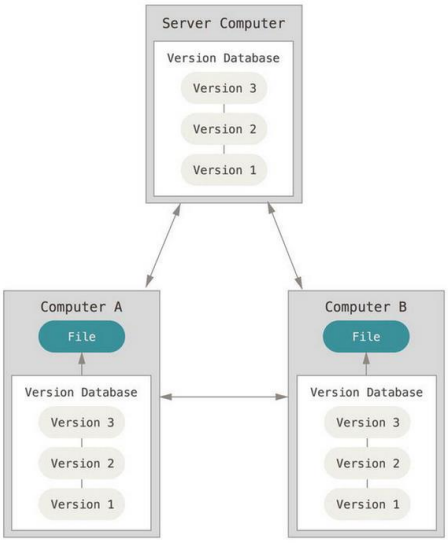


Ilustración 2. Esquema de un sistema de control de versiones distribuido

Git

Es un programa de control de versiones creado por la comunidad de desarrollo del kernel de Linux. Hasta 2005 utilizaron el programa BitKeeper, pero problemas de comunicación entre ambos equipos desembocaron en que parte de la comunidad de desarrolladores del kernel de Linux creasen un nuevo programa para el control de versiones distribuido: Git.

En la mayoría de los sistemas de control de versiones lo que se guarda son los cambios existente entre los ficheros, sin embargo en Git cada vez que un fichero se cambia se guarda una copia entera del fichero.

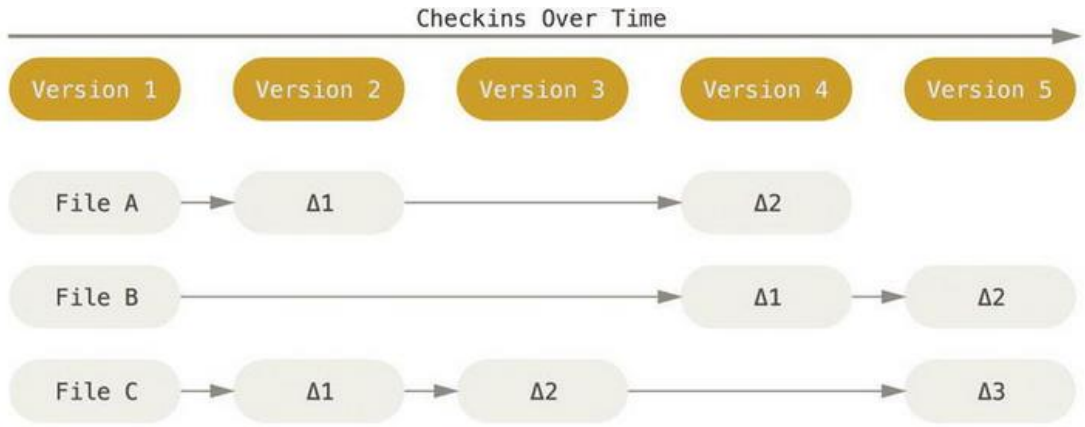


Ilustración 3. Funcionamiento de un sistema de control de versiones clásico

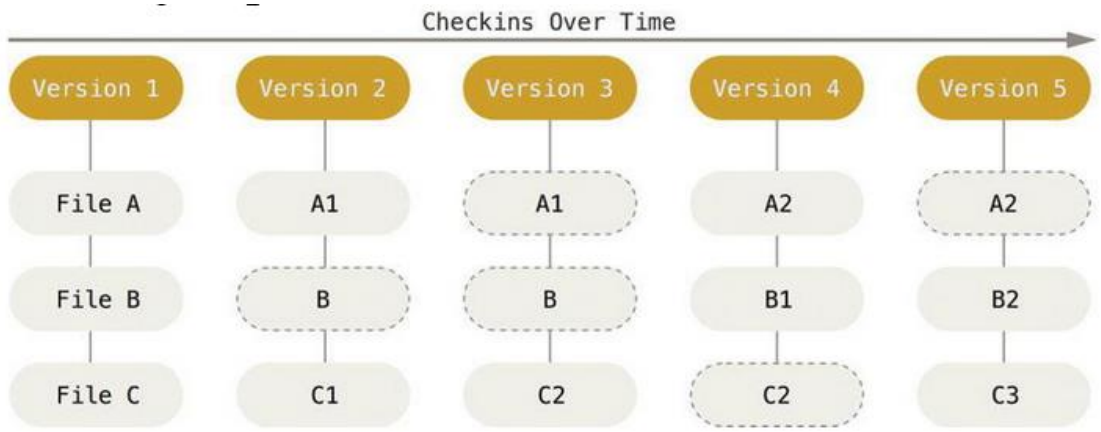


Ilustración 4. Funcionamiento de Git

Los ficheros en Git pueden tener el estado '*tracked*' o '*untracked*':

- **Tracked**: aquellos ficheros cuya existencia es conocida por Git
- **Untracked**: aquellos ficheros cuya existencia no es conocida por Git

Los ficheros con el estado '*tracked*' pueden a su vez tener uno de los siguientes tres estados: committed, modificado (*modified*) y staged.

- **Modificado**: el fichero ha sido modificado con respecto al original pero los cambios no han sido guardados.
- **Staged**: el fichero ha sido marcado para ser almacenado en la base de datos en el próximo commit.
- **Commited**: el fichero se encuentra registrado en la base de datos local.

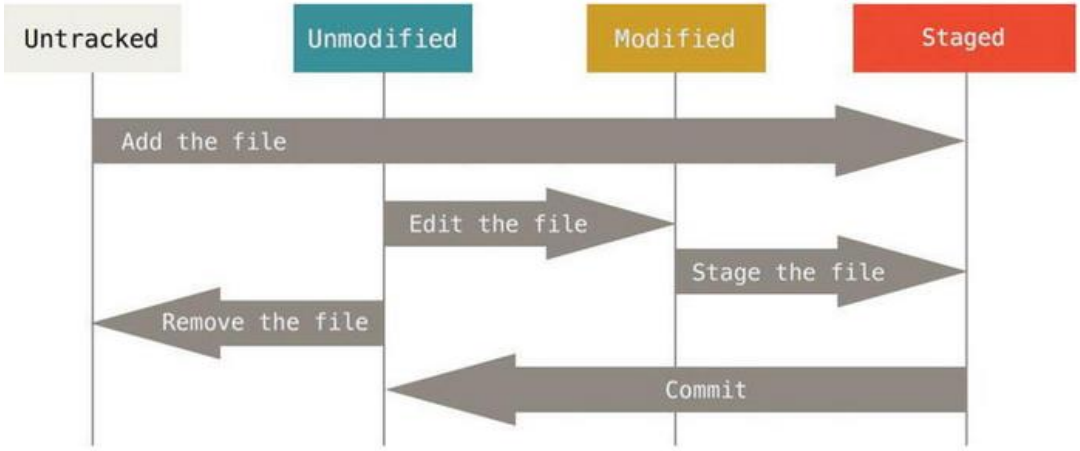


Ilustración 5. Posibles estados de un fichero en Git

Dentro del proyecto tendremos el directorio oculto .git que contiene la base de datos con todas las versiones anteriores de nuestro proyecto

GitHub

GitHub es una página web que permite almacenar online tu código y compartirlo con el resto del mundo. Es gratuito siempre y cuando tu proyecto sea de código abierto, puedes utilizarlo con código con licencia privativa siempre y cuando pagues. Es uno de los repositorios online de Git más famosos.

Ejercicios finales primera sesión

Ejercicio 1. Si no tienes cuenta en GitHub https://github.com/ ábrete una.

Ejercicio 2. ¿Qué tipo de programa es CVS? Haz un resumen del mismo. Realiza una comparación con Git

Ejercicio 3. Encuentra otros SCV que no sean ni Git ni CVS. Haz un breve resumen de cada uno de ellos.

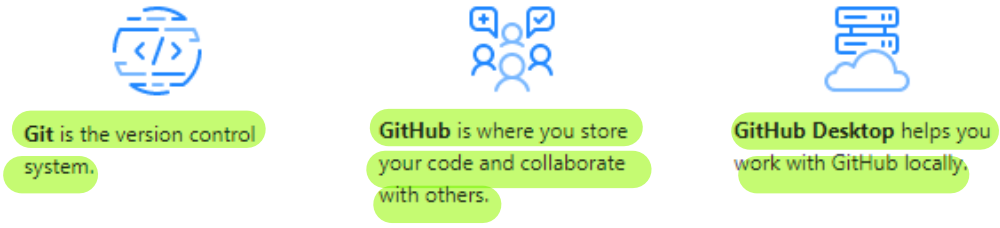
Ejercicio 4. ¿Qué es GitLab? Haz una comparación entre GitLab y GitHub

Ejercicio 5. ¿Qué significa el término 'DevOps'? ¿Cómo crees que se relaciona el término con los SCV?

GitHub Desktop

Herramienta gráfica para el control de versiones de tus códigos integrada con una cuenta de GitHub. No es tan pontente como Git Bash pues hay funcionalidades que no se encuentra implementadas en la versión gráfica.

Descargar de https://desktop.github.com/



Principales comando Git

Antes de comenzar a utilizar GitHub Desktop necesitamos unos conocimientos mínimos sobre los comandos disponibles en Git.

Para poder utilizar Git con un proyecto tuyo necesitarás crear un repositorio. Podemos definir, de forma simplificada, un repositorio como un conjunto de ficheros cuyos cambios se gestionaran mediante Git.

Lo principal para comprender como funciona Git es comprender que Git funciona con dos bases de datos. Una que tienes en local, en tu PC, y otra que hay en remoto.

Cuando modifiques un fichero de tu repositorio el código de tus ficheros en local será distinto al contenido que tengas en la BD local. Para sincronizar el contenido de tus ficheros con el de tu BD local realizamos el comando ‘commit’.

Ahora bien, el código será sin estar sincronizado con la BD remota, y por lo tanto el resto de compañeros, o tú desde otro PC, no tendrás acceso a los últimos cambios realizados. Para que la BD remota se sincronice con al BD en local necesitaras realizar el comando ‘push’.

Entonces… ¿siempre que haga commit debo hacer push? No tiene por qué ser así. Se recomienda utilizar un commit cada vez que implementas una nueva funcionalidad, arreglas fallos… mientras que push se realiza cuando quieras que tus cambios en local se vean reflejados en remoto, i.e.: quieres que tus compañeros de trabajo vean los cambios que has realizado.

Cuando quieras descargarte un repositorio en un nuevo PC deberás realizar el comando ‘clone’.

Si ya tienes el repositorio en un PC la BD local está desactualizada con respecto a la remota deberás realizar el comando ‘pull’

Uso de GitHub Desktop

Ejercicio 6. Conecta tu cuenta de GitHub con el programa GitHub Desktop. Para ello primero haz login en la web desde el explorador y luego en GitHub Desktop, file>options>Accounts>Sign in

Ejercicio 7. Utilizando GitHub desktop, crea un nuevo repositorio en tu cuenta llamado RandomWorldGenerator. Haz que la carpeta del repositorio esté en el escritorio. Haz que el repositorio se suba a tu cuenta en GitHub y comprueba que efectivamente el repositorio está online.

Ejercicio 8. Utilizando Netbeans, crea un proyecto Java para programar el proyecto RandomWorldGenerator, un potente generador procedural de mapas para un juego estilo Minecraft… Evidentemente solo debes crear el proyecto con ese nombre, no hace falta que lo programes. De momento. Eso sí, haz al menos un 'Hola Mundo' que podamos comprobar que el código se está subiendo.

Ejercicio 9. Comprueba que en local NetBeans ha generado los ficheros del proyecto dentro de la carpeta. ¿Están subidos a la web? ¿Por qué?

Commit

Para que los cambios se vean reflejados en la base de datos local (para posteriormente pasar a la web) debemos realizar el comando 'commit'.

Ejercicio 10. Abre GitHub Desktop ¿Qué te está mostrando?

Ejercicio 11. Haz un commit. ¿Se ha subido el fichero a la web? ¿Por qué?

Push

Para sincronizar la base de datos local de nuestro repositorio con la base de datos del servidor deberemos realizar el comando 'push'.

Ejercicio 12. Realiza un push. Comprueba que el código se ha subido a la web.

Ejercicio 13. Cuando llegues a cada bájate el repositorio en tu PC ¿qué comando debes realizar, clone o pull? ¿Por qué?

Ejercicio 14. Una vez hayas terminado de realizar cambios sobre el código ¿qué comandos se deben realizar?

Ejercicios finales segunda sesión

Ejercicio 15. Define con tus propias palabras los términos: Git, GitHub, repositorio, 'clone', 'commit', 'push' y 'pull'.

Ejercicio 16. A partir de lo visto en tu clase, escribe un breve texto en el que intentes convencer a un amigo que está empezando a programar para que use Git. Explícale cuales son las ventajas de usar Git.

Uso de Git desde la terminal

GitHub Desktop está genial para empezar pero si queremos tener acceso a la funcionalidad completa de git deberemos aprender a utilizarlo desde la consola de comandos. Además aprender a utilizar git desde la consola nos hace independientes de la interfaz gráfica que se esté utilizando.

Git Bash

Consola de comandos integrada en Git que nos permite realizar las operaciones necesarias en nuestro repositorio:

- Ofrece todas las funcionalidades de Git mientras que las herramientas gráficas ofrecen sólo un subconjunto de las mismas.
- Podemos automatizar el proceso mediante scripts
- La solución creada funcionará en cualquier sistema operativo

A parte de permitirnos manejar repositorio Git Bash ofrece comandos Unix muy útiles en el manejo y monitorización automáticos de ficheros y procesos.

Para sistemas Windows puedes decargala de https://gitforwindows.org/

Ejercicio 17. Comprueba que Git Bash está instalada en el PC de clase.

Para sistemas Linux nos bastará con descargarnos la app de Git

Ejercicio 18. Instala Git en la MV del tema anterior.

Para utilizar Git desde la terminal llamaremos al comando git seguido del comando que queramos realizar, i.e.: clone, commit, push, pull …

Los comandos, al igual que los comando Unix estudiados en el tema anterior, podrán recibir parámetros adicionales y ser modificados mediante opciones.

clone

Para comenzar a trabajar con git y la terminal lo primero que debemos aprender es a descargarnos nuestros repositorios al PC mediante la terminal.

Lo primero que debemos tener es la URL de nuestro repositorio, para ello accede desde el explorador a tu perfil de GitHub, entra en tu repositorio y en el botón code deberás tenerla.

Una vez tengas la url, dentro de la carpeta en la que te quieras descargar el repositorio haz click derecho y pulsa en 'Git Bash here'

Una vez dentro de la terminal teclea:

```
$ git clone X
```

Donde X será la URL de tu proyecto.

Ejercicio 19. Comprueba que el código del repositorio se ha descargado en tu PC tras hacer el comando clone. Abre el proyecto con

Ejercicio 20. Prueba algunos de los comandos vistos en el tema anterior en la terminal. ¿Funcionan? ¿por qué si estamos en Windows?

Ejercicio 21. Repite el ejercicio 19, ahora en la MV con Linux ¿cambia la forma de realizar el clone por estar en Linux?

Si no estás pudiendo hacer el clone este enlace te servirá de ayuda:

https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/creating-a-personal-access-token

Ejercicio 22. Una vez tengas el repositorios cargando tanto en Windows como en Linux ábrelo con Netbeans. Modifica el código en Windows, por ejemplo añade un nuevo 'sout'

commit

Para realizar el comando 'commit' primero debemos asegurarnos que si hemos creado algún nuevo fichero, este fichero pase a estar en el estado 'tracked', por lo tanto antes de realizar un 'commit' se suele hacer:

```
$ git add .
```

Y ahora sí, realizamos el commit:

```
$ git commit
```

Si el comentario del commit es corto es más comodo especificarlo en la misma linea de comandos, para ello:

```
$ git commit –m "Descripción de mi commit"
```

Ejercicio 23. En Windows, realiza el commit del cambio realizado.

push

Ya tenemos los cambios en la BD local, pero… ¿se han subido a la BD remota?

Ejercicio 24. Visita la web de tu repositorio, navega hasta el fichero .java en el que hayas realizado el cambio ¿se ve reflejado? ¿por qué?

Para realizar un 'push' utilizamos, vaya sorpresa' el comando push:

```
$ git push origin main
```

Quédate con que ese es el comando, cuando veamos el concepto de rama sabrás porque estamos poniendo 'origin' y 'main', de momento limítate a recordar que después del push, y de momento, siempre pondremos 'origin' y 'main'

Ejercicio 24. Haz un push al servidor y comprueba desde la web que los cambios se han realizado.

pull

Para sincronizar una BD local con al BD en remoto necesitaremos hacer un pull.

Para ello:

```
$ git pull origin main
```

Ejercicio 26. ¿El código del proyecto en Linux está actualizado? ¿por qué? ¿qué comando debemos realizar?

Ejercicio 27. Realiza un pull de proyecto desde Linux. Comprueba que el código pasa a estar actualizado.

Ejercicios finales tercera sesión

Ejercicio 28. Mira el video https://www.youtube.com/watch?v=PT4otSBaieM y repite todo lo que va haciendo el profe. Comprueba que no surgen errores. Si tienes errores comunícaselo al profesor.

Ejercicio 29. ¿Con qué comando podemos volver a versiones anteriores de Git? ¿Cuál es su sintaxis? Intenta volver a la primera versión de tu proyecto

Ejercicio 30. En Git, ¿qué es un rama?, ¿cómo crees que pueden beneficiarte a la hora de organizar el código de tu proyecto?

.gitignore

Habrá ocasiones en las que ciertos ficheros no querrás subirlos al servidor, que sólo querrás tenerlos en local, como por ejemplo los fichero multimedia de mucho peso, ejecutables, ficheros de la base de datos, bibliotecas de terceros...

Dicho ficheros se especifican en el fichero '.gitignore'.

Para ignorar un fichero pondremos la ruta relativa de dicho fichero.

Para ignorar todos los ficheros de una determinada extensión: *.zip, ignoraría todos los ficheros zip de nuestro proyecto

También podemos ignorar carpetas enteras –no olvides poner / al final del nombre la carpeta.

Listado completo de funcionalidades del fichero .gitignore

Ignorar un único fichero

```
privado.txt
```

Ignorar múltiples ficheros con una misma extensión

```
*.exe
```

Ignorar múltiples ficheros que comiencen por la misma cadena

```
privado_*
```

Ignorar carpetas enteras

```
privado/
```

Ignorar ficheros concretos dentro de carpetas

```
bd/passwords.txt
```

Mantener un fichero concreto que contradiga una regla

```
*.log
!public.log
```

Ficheros con un conjunto de caracteres en su nombre

```
[Pp]rivado.txt
```

En https://gist.github.com/octocat/9257657 tienes un ejemplo de ficheros genéricos de .gitignore

Volviendo a versiones anteriores

Otra de las funcionalidades de los sistemas de control de versiones es volver a versiones anteriores de nuestro código.

Descartar cambios entre un fichero y la última versión de la base de datos en local:

```
$ git restore <fichero>
```

Para revertir los cambios en todos los ficheros:

```
$ git restore .
```

Básicamente lo que estamos haciendo con 'restore' es descartar todos los cambios hechos desde el último commit en local.

Para poder volver a la versión de un commit concreto debemos encontrar el hash de la versión a la que queremos revertir. Para ello:

```
$ git log > log.txt
```

Para cada commit tendremos:

```
commit 3f8536a1de6c724d60fa925d07a80289d472a637
Author: Borja Molina Zea <x@gmail.com>
Date:   Tue Dec 29 20:32:59 2020 +0100
```

Una vez que tengamos el hash de la versión haremos:

```
$ git reset --hard <hash>
```

Ramas (branches)

Por defecto el proyecto comienza con una única rama que suele recibir el nombre de 'main' (antiguamente 'master')

Pero... ¿Qué es una rama? Una rama es una línea de desarrollo de código dentro de nuestro proyecto. Imagina que tenemos un videojuego, mientras que una parte del grupo está desarrollando la inteligencia artificial otro puede estar desarrollando las físicas del juego, en principio tendríamos tres ramas, la principal, la rama donde se está implementando la ia y la rama dónde se está implementando las físicas. Cuando se terminase de implementar la ia dicha rama se fusionaría con la rama principal quedando la ia implementada en el proyecto final y visible para el resto de grupo de desarrollo.

En la práctica cuando trabajamos solos podemos trabajar sobre la rama principal pero si queremos publicar nuestra código es buena idea que programemos en una rama y que dejemos la principal para la versión estable y descargable de nuestro proyecto y que solo unamos la rama de desarrollo con la principal una vez que sepamos que todos los cambios que hemos realizado están libres de fallos.

La idea sería que una vez que hayamos terminado de implementar aquello por lo que la rama fue creada que unamos la nueva rama con la rama 'main'.

¿Cuántas ramas se recomienda tener? Muchos grupos trabajan con tres ramas fijas (main, release y develop) más una rama adicional por cada nueva característica que se está programando.

Imagina que trabajas para una empresa que está creando un juego. El juego está en early access y ya está casi terminado tan solo hace falta pulir la inteligencia artificial de los enemigos y las físicas del juego. Tendríamos tres ramas (main, release y develop) más dos ramas adicionales (una para la inteligencia artificial y otra para las físicas).

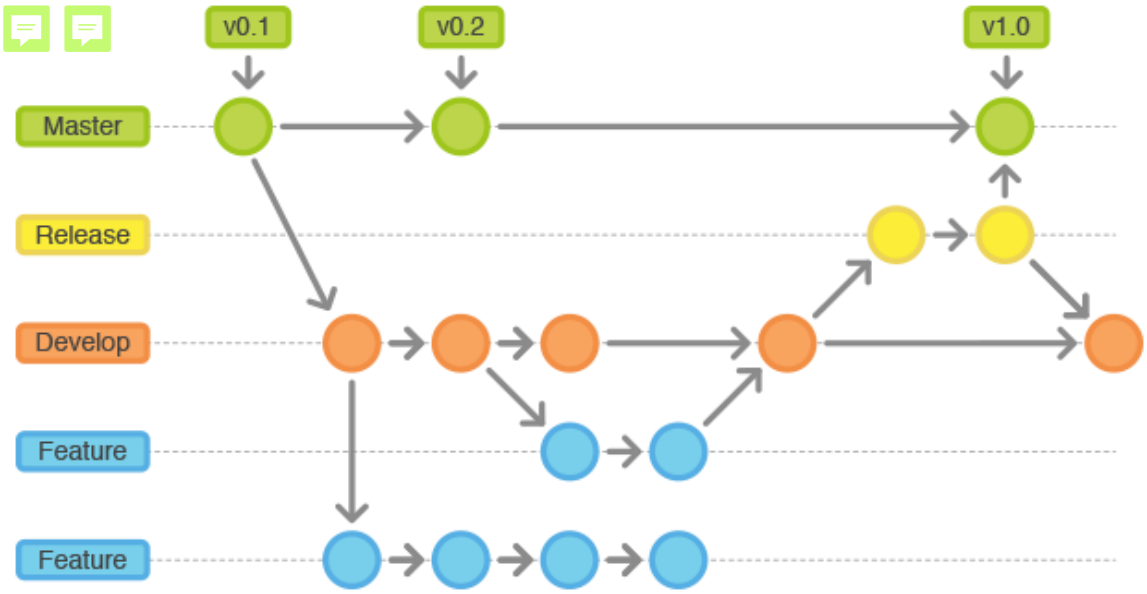


Ilustración 6. Proyecto con cinco ramas. Las ramas main, release y develop más dos ramas adicionales para dos funcionalidades concretas que estás siendo programadas.

Lo anteriormente expuesto sería un flujo 'ideal' de trabajo. En muchas ocasiones te encontraras una rama main y una rama de desarrollo en la que trabajaras directamente.

Ejercicios finales cuarta sesión

Ejercicio 31. Explica con tus propias palabras que tipos de ficheros ignora el fichero https://gist.github.com/octocat/9257657 ¿Por qué crees que se ignoran dichos ficheros?

Ejercicio 32. Crea un fichero .gitignore, o si ya está creado modifícalo, con el contenido de https://gist.github.com/octocat/9257657 dentro de tu proyecto

Ejercicio 33. Añade una carpeta con el nombre 'privado' al proyecto y dentro un fichero con el nombre 'log.txt'. Haz que todo lo que hay dentro de dicha carpeta no se sincronice con el servidor.

Ejercicio 34. Ignora todos los ficheros .com excepto a.com

Ejercicios 35. Haz que los cambios realizados en el fichero .gitignore tengan efecto sobre el repositorio. ¿Se ha subido la carpeta 'private'? ¿Por qué?

Ejercicio 36. Añade los ficheros a.com y b.com en local. Sincroniza la BD remota ¿se han subido los dos ficheros? ¿Por qué?

Ejercicio 37. Modifica un fichero del proyecto y guarda los cambios. Cierra el fichero y haz un restore del repositorio. ¿Qué ha pasado?

Ejercicio 38. Hemos visto cómo volver a versiones concretas en local pero... ¿cómo podríamos volver a dichas versiones en remoto?

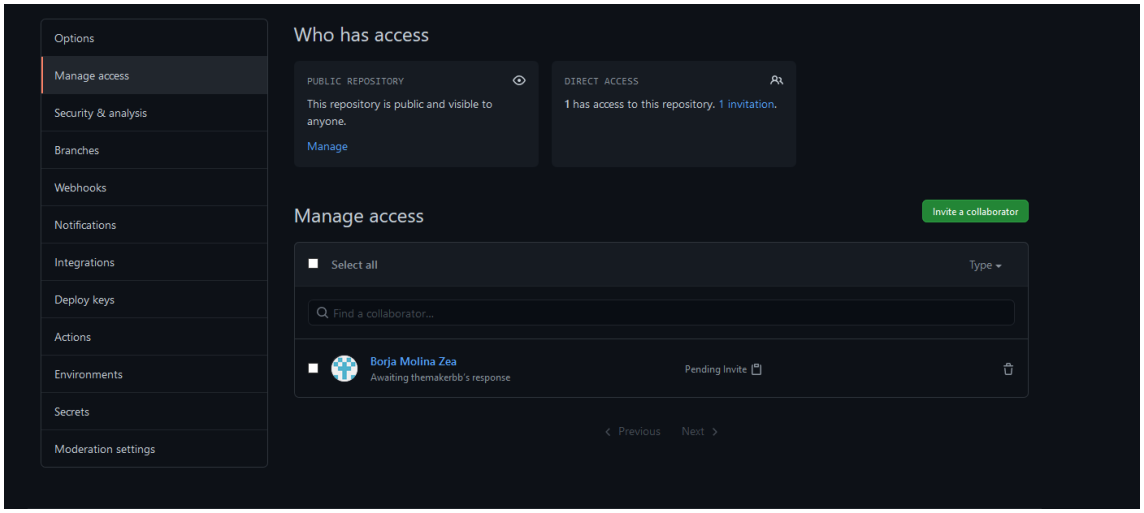
Trabajar en equipo

Todo lo que hemos visto hasta ahora está muy bien, ya que nos permite:

a) Tener una copia de seguridad de nuestro código
b) Compartir nuestro código con la comunidad
c) Trabajar cómodamente con nuestro código desde diferentes ordenadores

Pero... seguimos sin saber trabajar en equipo.

El primer paso será dar acceso a nuestro proyecto a otra persona. Lo haremos desde la web de GitHub:



Ejercicio 39. (por parejas) El alumno A deberá crear un nuevo proyecto, descargarlo en su pc y dar acceso a su proyecto al alumno B. El alumno B deberá aceptar la invitación y descargarse el repositorio en una carpeta en su PC.

Cuando haya varias personas trabajando sobre el mismo código pueden surgir conflictos si la versión del código de una persona es distinta a la de otra, para minimizar dichos conflictos debes recordar la siguiente regla de oro:

Haz pull frecuentemente. Y siempre, SIEMPRE, haz pull antes de hacer push.

Aun siguiendo esa regla habrá ocasiones en las que haya conflictos, pero lo que está seguro es que si no sigues dicha regla es muy MUY probables los haya y lo más seguro que sobrescribas trabajo de tus compañeros (tranquilo ya que se podrá recuperar pero estás añadiendo trabajo a tus compañeros porque arreglarlo no será automático)

Para quedarnos con el flujo de trabajo vamos a realizar dos ejercicios, uno en el que haremos pull antes del push y ello nos evitará conflictos y otro en el que aunque hagamos pull antes del push habrá conflictos.

En grupos de trabajo grandes, donde hay distintos subgrupos encargados de distintas partes de la aplicación o la web, se utilizan ramas como te he comentado anteriormente, en este ejercicio para simplificar (y porque al ser sólo dos persona las ramas nos estorban más que otra cosa) trabajaremos directamente sobre la rama 'main' (aunque recuerda que una vez tengas una versión estable de tu proyecto es mejor trabajar en una rama de desarrollo).

Ejercicio 40. (por parejas) Emula, junto a un compañero, el siguiente flujo de trabajo.

Alumno A: empieza a trabajar en el código. Nada más ponerse hace pull para ver si su compañero ha hecho algo mientras él estaba descansando. Cambia el código de su web y hace commit cada vez que considera oportuno. Antes de irse a casa quiere hacer un push porque considera que el código que ha hecho cumple una tarea y no tiene errores. Para no pisar el posible trabajo de su compañero (que puede haberse puesto a trabajar desde su útlimo pull) hace de nuevo pull. Su compañero B no ha entrado en acción, por lo tanto podrá hacer push sin ningún conflicto.

Alumno B: una vez el alumno A ha terminado su jornada de trabajo el alumno B la comienza. Empieza el día haciendo un pull para ver si hay código nuevo y ve que efectivamente lo hay, al hacer pull podrá haber o no conflicto con su último commit, como la última vez que terminó de trabajar hizo push no hay conflicto ninguno y su código se actualiza sin problema a la nueva versión. Sigue trabajando todo el día, haciendo 'commit' conforme va cerrando funcionalidades. Cuando termina su jornada de trabajo considera que es útil hacer 'push', pues ha terminado de implementar funcionalidades y ha probado que no tengan errores, pero antes de hacer push hace pull para asegurarse que su compañero no haya trabajado durante este tiempo. Como el alumno A no ha hecho nada puede hacer push sin problema alguno.

Alumno A: vuelve a comenzar su jornada de trabajo y hace pull. Su código se actualiza y ve lo último que ha añadido su compañero.

Lo anteriormente descrito es un flujo de trabajo muy tranquilo. Como no trabajan a la vez, y siempre y cuando sigan un flujo de trabajo como el descrito, nunca habrá conflicto entre los códigos. De hecho cuando no se está trabajando a la vez bastaría con hacer pull al empezar a trabajar y push al terminar de trabajar (pero por favor recuerda que en la vida real tienes que hacer pull siempre justo antes de push)

Veamos ahora, con otro ejercicio, la aparición de un conflicto y como solucionarlo.

Ejercicio 41. (por parejas) Emula, junto a un compañero, el siguiente flujo de trabajo

Alumno A: empieza su jornada de trabajo. Hace pull. Implementa una funcionalidad (para clase con añadir una línea es suficiente). Pero no hace commit ya que al ser una funcionalidad muy importante se queda haciendo pruebas para comprobar que todo va bien antes de hacer push.

Alumno B: empieza su jornada de trabajo mientras que A está depurando su función. Hace pull. Implementa varias funcionalidades (en vuestro caso líneas) y para cada funcionalidad hace un commit. Comprueba que todo funciona correctamente y por lo tanto decide hacer un push, pero recuerda, antes de ello hace pull (como A no ha enviado nada aún no tendrá ningún conflicto). Al no haber conflicto B hace push.

Alumno A: termina de comprabar que lo que ha implementado funciona bien y decide hacer push, pero antes hace pull para ver si B ha hecho algo y...

```
CONFLICT (content): Merge conflict in README.md
Automatic merge failed; fix conflicts and then commit the result.
```

Tu fichero deberá ser algo similar a lo siguiente (allí dónde está el conflicto):

```
<<<<<<< HEAD
Usuario a añade una linea
=======
Alumno b añade linea

Alumno b vuelve añadir otra linea
>>>>>>> 6ea58a50776b098eb45cdd7d52a4afc2113e7e08
```

Los conflictos pueden ser resueltos de modo automático mediante los comandos merge o mergetool. Pero en nuestro caso lo arreglaremos a mano, como lo que añadió B no entra realmente en conflicto con los nuestro –ya que realmente lo que han hecho A y B es añadir nuevo código que no es dependiente el uno del otro- simplemente dejaremos tanto nuestro código como el de B.

```
Usuario a añade una linea

Alumno b añade linea

Alumno b vuelve añadir otra linea
```

Una vez resuelto el conflicto hacemos push

Cuando el alumno B haga pull verá el código con el conflicto ya resuelto.

Hemos resuelto el conflicto más común y sencillo de todos, que tanto A como B hayan añadido código en un mismo fichero, sin que dichos códigos interfieran entre sí. Cuando esto sucede es seguro resolver el conflicto sin tener que informar a la otra parte ya que solo tendremos que aceptar los cambios realizados tanto por nosotros como por ellos.

Sin embargo no siempre será tan fácil, habrá ocasiones que el conflicto deberá ser resuelto comunicándoselo a la otra/s persona/s implicada/s y determinando entre todos cual es la solución por la que se opta.

Ejercicios finales quinta sesión

Ejercicio 42. Realiza un resumen de cómo se debe utilizar Git cuando estás trabajando en un proyecto junto a más personas.

Ejercicio 43. ¿Qué es un fork? Encuentra ejemplos de forks famosos en la vida real.

Ejercicio 44. ¿Qué es una pull request?¿Puedo hacer una pull request en mi propio código? ¿Pueden otras personas hacer una pull request en mi código?

Fork

Una de las ventajas de GitHub es que puedes participar activamente en el desarrollo de software libre proponiendo cambios en el código. De forma análoga también puede suceder que alguien se interese por tu código y lo modifique o incluso implemente nuevas funcionalidades informádote de ello y puediendo aceptarlas y que queden implementadas en tu proyecto.

Para participar activamente en el proceso de desarrollo de un proyecto deberemos hacer un ‘fork’ del proyecto. Un ‘fork’ sería una copia del proyecto en la que los cambios que realizamos en el código no pueden ser enviados al mismo, nos permite trabajar con el código de proyectos ya existentes sin afectar al código del proyecto.

Para hacer un fork tenemos que ingresar en la web de GitHub, ir a la página del proyecto y pulsar en el botón de fork:

Una vez que hagas el fork el proyecto debería aparecer en tu cuenta. Es importante que entiendas que esto no quiere decir que estés colaborando con el proyecto –no aún al menos- simplemente has hecho una copia del proyecto y los cambios que hagan se verán reflejados en dicha copia pero no en el proyecto original.

Crar un repositorio en local para trabajar con el fork será tan sencillo como clonar el fork en la carpeta en la que quieras descargar el proyecto.

Ejercicio 45. Haz un fork del proyecto de un compañero y clónalo en tu PC de forma que se descarguen todos los ficheros del proyecto.

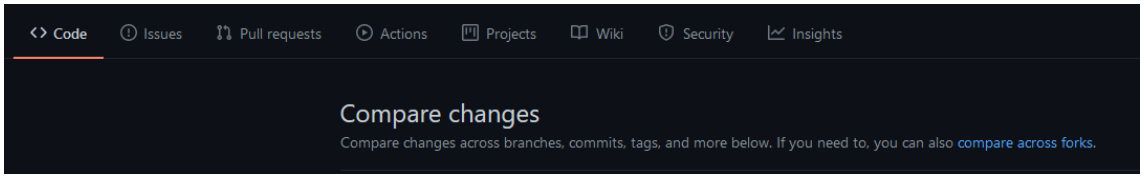
Pull request

Imagina que haces un fork, que implementas una nueva funcionalidad y que crees que dicha funcionalidad podría servir al resto de la gente. Cuando consideres que los cambios realizados al proyecto o las nuevas funcionalidades añadidas merezcan la pena ser compartidas puedes informar de dichos cambios al dueño del proyecto mediante un 'pull request'. El dueño del código verá los cambios realizados por tí y determinará si quiere añadirlos. Si los acepta quedarán registrados como una aportación de tu perfil al proyecto. Si algún día llegas a colaborar en un proyecto de software libre conocido será una buenísima notica para ti, ya que las empresas suelen tenerlo en cuenta a la hora de contratar.

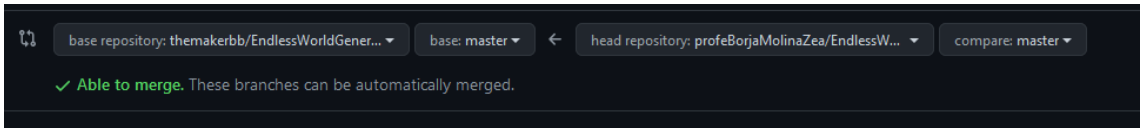
Para conseguir lo anteriormente expuesto hacemos uso de las 'pull request'.

En grupos de trabajo grandes también se suele trabajar con 'pull request' pero nosotros las estudiaremos como una forma de nosotros proponer cambios a un proyecto que no es nuestro.

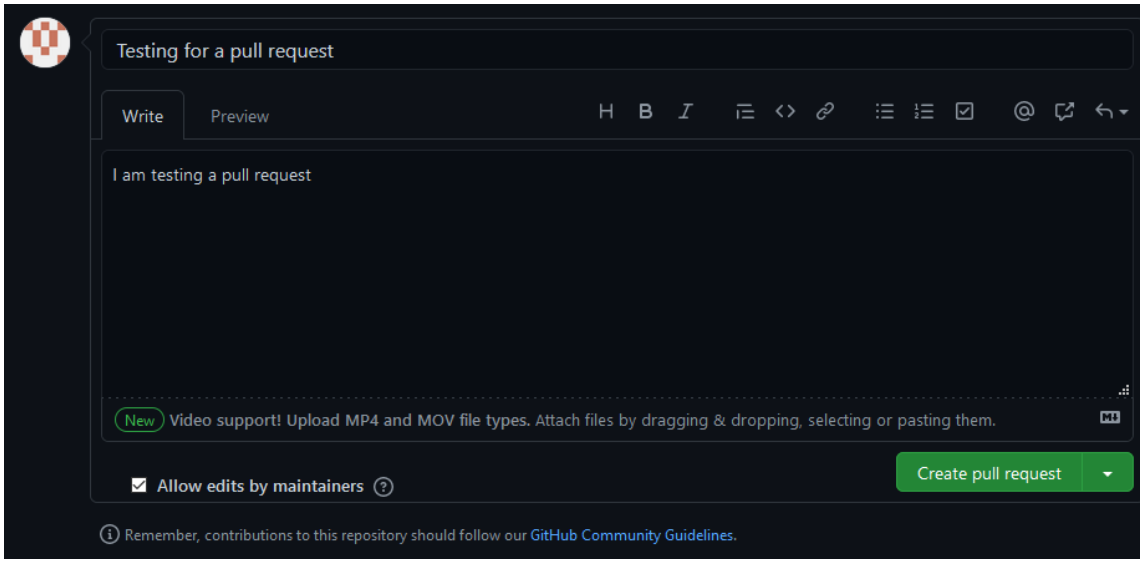
Para realizar un pull request modifica el código del fork. Visita la página del proyecto y pulsa en la pestaña de pull request



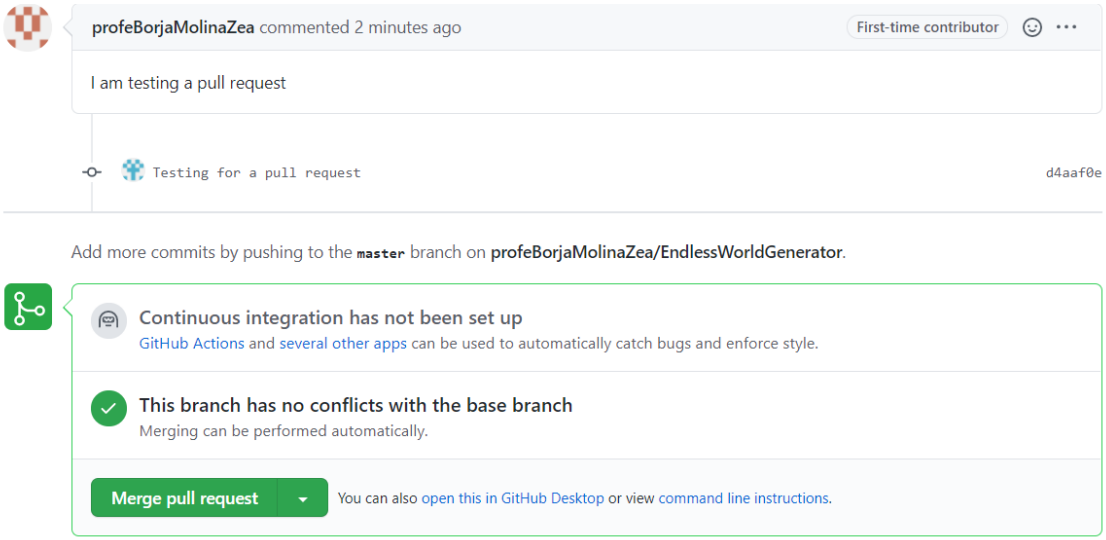
Pulsa en 'compare across forks'. Especifica la rama del proyecto original y el fork y rama del mismo de la cual harás el pull request:



Pulsa en 'create pull request' y especifica un título y cuerpo para tu pull request.



El dueño del proyecto verá que tiene una nueva pull request y podrá aceptarla o rechazarla:



Si la acepta tus cambios se verán reflejados en el proyecto oficial y se podrá ver desde tu cuenta que realizaste una pull request y que fue aceptada:

De esta manera puedes participar en el desarrollo de software libre o echar una mano a tus amigos con sus proyectos sin necesidad de que te den permisos de acceso al proyecto.

Ejercicio 46. Haz una pull request del proyecto de tu compañero del que hiciste fork.

Ejercicio 47. Acepta la pull request que te haya hecho tu compañero.

Creando una página personal en GitHub

La principal función de GitHub es la de albergar el código fuente de programas informáticos pero también permite crear página personales –GitHub pages- donde crear tu perfil como informático (realmente puedes utilizar la web para lo que quieras, pero es GitHub, la gente la utiliza como escaparate de su perfil como informático).

Para crear una página web personal en GitHub comienza creando un nuevo repositorio y ponle como nombre <nombreUsuario>.github.io . Es importante que pongas exactamente ese nombre al proyecto, es la forma que tiene GitHub de entender que se trata del proyecto referente a tu página web personal y que por lo tanto cuando la gente lo visite lo que tiene que mostrarse es la web y no el código.

Una vez que hayas creado el proyecto crea un carpeta en tu pc y clona el proyecto. Inserta un fichero llamado index.html y súbelo a remoto. Podrás visitar tu página accediendo a la URL https://**<nombreUsuario>**.github.io

Por ejemplo la página del profe: https://profeborjamolinazea.github.io/ (no tengas en cuenta la falta de información... tan solo era para hacer la prueba)

Seis buenas prácticas utilizando Git

Esta sección es un resumen de https://medium.com/better-programming/six-rules-for-good-git-hygiene-5006cf9e9e2

1. **Haz pull siempre antes de hacer push**. Si no lo haces puedes sobrescribir lo programado por muchas personas y que muchos compañeros de trabajo tengan conflictos cuando hagan el próximo pull. Es mejor que los conflictos los tenga una sola persona a que los tenga todo el grupo de trabajo. Para ello, recuerda: haz pull antes del push. Si al hacer pull hay conflictos, resuelvelos, y sólo entonces has push.
2. Haz pull frecuentemente. De esta manera podrás ir incorporando los cambios que hagan tus compañeros en el proyecto en tu versión del código.
3. Haz push pocas veces. Haz push solo cuando hayas terminado de implementar por completo una nueva funcionalidad y te hayas asegurado que no tiene bugs.
4. Haz commit frecuentemente. Cada vez que termines de implementar una pequeña funcionalidad haz commit, esto te permitirá tener un buen mapa de referencia de tu trabajo realizado en el proyecto y mantener bien diferenciados las distintas funcionalidades que has ido programado. Toma como brújula que el comentario de cada uno de comentarios debería ser corto y definir una única nueva funcionalidad añadida.
5. Une tu rama de desarrollo –merge- frecuentemente. Esto te permitrá estar al tanto de lo programado por todo el grupo. Normalmente programarás en una

rama que será creada a partir de la rama de desarrollo –que a su vez es creada de la rama main-

6. Haz pocas pull request. Hazlas solo cuando estés seguro que las funcionalidades que queires añadir al proyecto están completamente terminadas y sin errores. Sin embargo no esperes a hacer pull request con muchas varias funcionalidades añadidas, intenta que cada pull request encapsule una única funcionalidad, pero hazla cuando estés seguro que la funcionalidad de encuentra perfectamente implementada.

Conclusiones finales

Hemos visto tan solo la capa más superficial de Git y GitHub. Intentar explicar Git en un documento de menos de 30 páginas es imposible. Espero, eso sí, que te hayan quedado claras las bases de Git. Si el flujo de trabajo en grupo no te ha quedado claro no te preocupes, cada grupo de trabajo establece su propio flujo, habrá grupos grandes con varias ramas dentro de un mismo proyecto, grupos más pequeños con solo dos ramas –una de desarrollo y la rama main- (e incluso me he llegado a encontrar empresas donde todos los cambios se realizaban directamente sobre la rama 'main', metodología que como hemos visto no es muy acertada).

Lo que si me gustaría que te quedase claro es que la regla de oro cuando trabajes en grupo es siempre hacer pull antes de commit, a partir de ahí cada grupo establecerá su propio flujo de trabajo. Habrá grupos que trabajen con muchas ramas y con pull request incluso dentro del grupo y los habrá con flujos de trabajo mucho menos complejos.

Los conocimientos principales que deberías haber adquierido al teminar este tema son:

- Qué es un sistema de control de versiones.
- Qué es una rama –branch- y cómo unirlas.
- Qué es un fork.
- Cómo volver a una versión anterior de tu proyecto.
- Cómo trabajar con tus propios proyectos utilizando Git para tener una copia de seguridad en la nube y para trabajar cómodamente con tu código desde distinto ordenadores.
- Aprender lo básico sobre cómo trabajar en grupo utilizando Git.
- Realizar 'pull request' a un proyecto
- Crear una página web personal en tu perfil de GitHub

Aunque el objetivo principal de un sistema de control de versiones es el de permitir que varias personas trabajen de forma eficaz sobre un mismo código te recomiendo que para todos las prácticas del curos, aunque sean individuales, utilices Git. Irás aprendiendo a manejarlo para cuando tengas que hacerlo en un grupo de trabajo y te permitirá tener una copia de seguridad de tu proyecto. Además si no quieres compartirlo siempre puedes usar Git sin GitHub.

Ejercicios finales sexta sesión

Ejercicio 48. Crea tu página personal en GitHub.