

```

\documentclass[11pt]{article}
\usepackage{amsmath}
\usepackage{bm}
\usepackage{tikz}

%Write an article
\title{An Article}
\author{Me}
\date{Today}

\begin{document}
\maketitle

\section{First Topic}
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nulla facilisis cursus justo, quis sodales orci tempus vitae. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos. Aliquam vitae ex nec ligula blandit pharetra nec pretium arcu. Nunc metus quam, iaculis vel mi nec, placerat ultricies nunc. Quisque magna sem, sodales eget tempus posuere, finibus ut quam. Integer feugiat nibh lectus, eget vestibulum est pellentesque vel. Vivamus commodo lorem metus, ac faucibus magna commodo sit amet. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Nullam varius tempor odio, id finibus lectus. Suspendisse eget diam diam. Duis sed ex in ex blandit varius vitae id lorem. Duis suscipit leo non accumsan sagittis. Etiam quis auctor ligula.

\[\iint_{\Sigma}\nabla\!\times\!\mathbf{F}\cdot d\mathbf{bm}_{\Sigma}=\oint_{\partial\Sigma}\mathbf{F}\cdot d\mathbf{bm}_r.\]

\section{Second Topic}

```

LaTeX y Git aplicado a la investigación científica

Módulo 1 – Git básico

del 15 de marzo al 17 de mayo de 2022

AULA VIRTUAL



Manual: *"LaTeX y Git aplicado a la investigación científica"*.

Curso Virtual

Módulo 1. *Git básico*

34 páginas.

Marzo 2022. Elaborado por Pablo Hinojosa.

Asociación Darwin Eventur

Git Básico

Introducción.....	2
Software Libre	2
Sistemas de control de Versiones.....	3
Tipos de Sistemas de Control de Versiones	4
git.....	4
Abrir una cuenta en GitHub	5
Instalar git.....	6
Clientes GUI para Linux, Windows y Mac	18
Empezando a usar git.....	18
¿Cómo funciona git?.....	21
Manteniendo nuestro repositorio al día	22
Sincronizando repositorios	24
El archivo .gitignore	28
Comportamiento por defecto de push.....	29
Obtener Ayuda	30
Viendo el historial.....	31
Borrado de archivos.....	32
Rehacer un commit.....	32
Deshacer cambios en un archivo	32
Resolviendo conflictos	33
Retrocediendo al pasado.....	34
Viendo (y recuperando) archivos antiguos	34

Introducción

Git es el sistema de control de versiones más utilizado hoy día, tanto para proyectos privados en empresas de toda escala, como para grandes proyectos abiertos, como el kernel del sistema operativo GNU-Linux.

Además, es un sistema que ha tenido especial éxito en el ámbito académico e investigador, más allá de entornos puramente informáticos, y especialmente en el contexto de la [ciencia abierta](#).

En esta primera mitad del curso vamos a ver el uso de este sistema de control de versiones.

Software Libre

En los últimos años, el software libre se ha ido extendiendo hasta abarcar la mayor parte de sistemas de Internet, supercomputadores o servicios de red, llegando finalmente a ordenadores personales, tablets, teléfonos e incluso electrodomésticos.

Llamamos Software Libre al que permite al usuario ejercer una serie de libertades, a saber:

- la libertad de usar el programa, con cualquier propósito.
- la libertad de estudiar cómo funciona el programa y modificarlo, adaptándolo a tus necesidades.
- la libertad de distribuir copias del programa, con lo cual puedes ayudar a tu prójimo.
- la libertad de mejorar el programa y hacer públicas esas mejoras a los demás, de modo que toda la comunidad se beneficie.

Para que un programa sea libre hacen falta dos requisitos fundamentales:

- Una licencia que permita el uso, modificación y distribución del programa
- Acceso al código fuente del programa

Además, en la comunidad de software libre se dan algunas peculiaridades, como la internacionalización o el trabajo colaborativo de gran cantidad de desarrolladores, que han promovido la creación de forjas, repositorios y sistemas que permitan compartir ese código de forma abierta y, además administrar y gestionar todo ese trabajo de una forma eficiente.

git es uno de estos sistemas, y es software libre.

Sistemas de control de Versiones

En cualquier proyecto de desarrollo es necesario gestionar los cambios, modificaciones, añadidos etc. que se han hecho a lo largo de la historia de dicho proyecto.

Si se trata de un trabajo pequeño, de corta duración y es llevado a cabo por un solo programador, quizás un archivo histórico de copias de seguridad pueda ser suficiente, pero esto se vuelve claramente insuficiente al crecer la complejidad del proyecto.

Alguno de los problemas más habituales a los que se enfrenta cualquier persona que participe en un desarrollo informático son:

- Dos o más programadores modifican el mismo archivo de código y hay que gestionar ese conflicto.
- Es necesario mantener (al menos) dos versiones del proyecto, una en producción y otra en desarrollo.
- Algún cambio ha sido desechado y es necesario regresar a una fase anterior del proyecto.
- Se inicia un "fork" o proyecto derivado a partir del estado actual del proyecto.

Llevar a cabo la administración de estos aspectos (y de muchos otros) manualmente es materialmente imposible, y es para ello para lo que se inventaron los Sistemas de Control de Versiones.

Debido al crecimiento en extensión y complejidad del software, los Sistemas de Control de Versiones están tomando cada vez más importancia. En especial, los proyectos de software libre, que tienden a aunar los esfuerzos de un gran número de programadores trabajando simultáneamente en múltiples versiones del código, han estado a la vanguardia del uso de estas herramientas.

Estos sistemas nacieron para gestionar código fuente, pero pueden ser usados para cualquier tipo de documentación (aunque no pueden aprovechar todo su potencial en archivos que no sean de texto plano). Por ejemplo, este curso ha sido desarrollado bajo git y su código puede encontrarse en su repositorio oficial en github.com/oslugr/curso-git

Tipos de Sistemas de Control de Versiones

Existen muchos Sistemas de Control de Versiones como [CVS](#), [Subversion](#), [Bazaar](#), [Mercurial](#) o, por supuesto, git, pero todos pueden clasificarse en dos tipos fundamentales según su modo de trabajo.

Sistemas centralizados

Los sistemas de control de versiones centralizados fueron los primeros en aparecer y son los de funcionamiento más simple e intuitivo.

En ellos, existe un repositorio central donde se archiva el código y toda la información asociada (marchas de tiempo, autores de los cambios, etc.). Los distintos desarrolladores trabajan con copias de ese código que actualizan a partir del servidor central, a donde también envían sus cambios.

Es decir, la versión "oficial" de referencia es la que hay en ese repositorio, y todos los programadores sincronizan sus versiones con esa.

Los programas más conocidos de este tipo son CVS y Subversion.

Sistemas distribuidos

Los sistemas distribuidos son más complejos, pero a cambio ofrecen una mayor flexibilidad.

En estos sistemas no existe un servidor central, sino que cada programador tiene su propio repositorio que puede sincronizar con el de cada uno de los demás.

Hay que hacer notar que, aunque no es necesario en este tipo de arquitectura, en la práctica se suele definir un repositorio de referencia para albergar la versión "oficial" del software.

Los sistemas distribuidos más conocidos son Bazaar y, por supuesto, git.

git

git nació para ser el Sistema de Control de Versiones del kernel de Linux (de hecho, fue originalmente programado por el mismo *Linus Torvals*) y es por ello que está preparado para proyectos grandes, con muchos desarrolladores y un gran número de ramas.

Se trata, como ya hemos dicho, de un Sistema de Control de Versiones distribuido, por lo que cada programador cuenta con su propio repositorio. Esto hace que, salvo cuando llega el momento de sincronizar con otro repositorio, todo el trabajo se haga en local, con ventajas como la velocidad o que se pueda trabajar sin acceso a la red.

git es software libre, y su código está disponible en su [repositorio de GitHub](#).

El sitio oficial de git es git-scm.com

git es también multiplataforma, y existen versiones para Linux, Mac, Windows y Solaris.

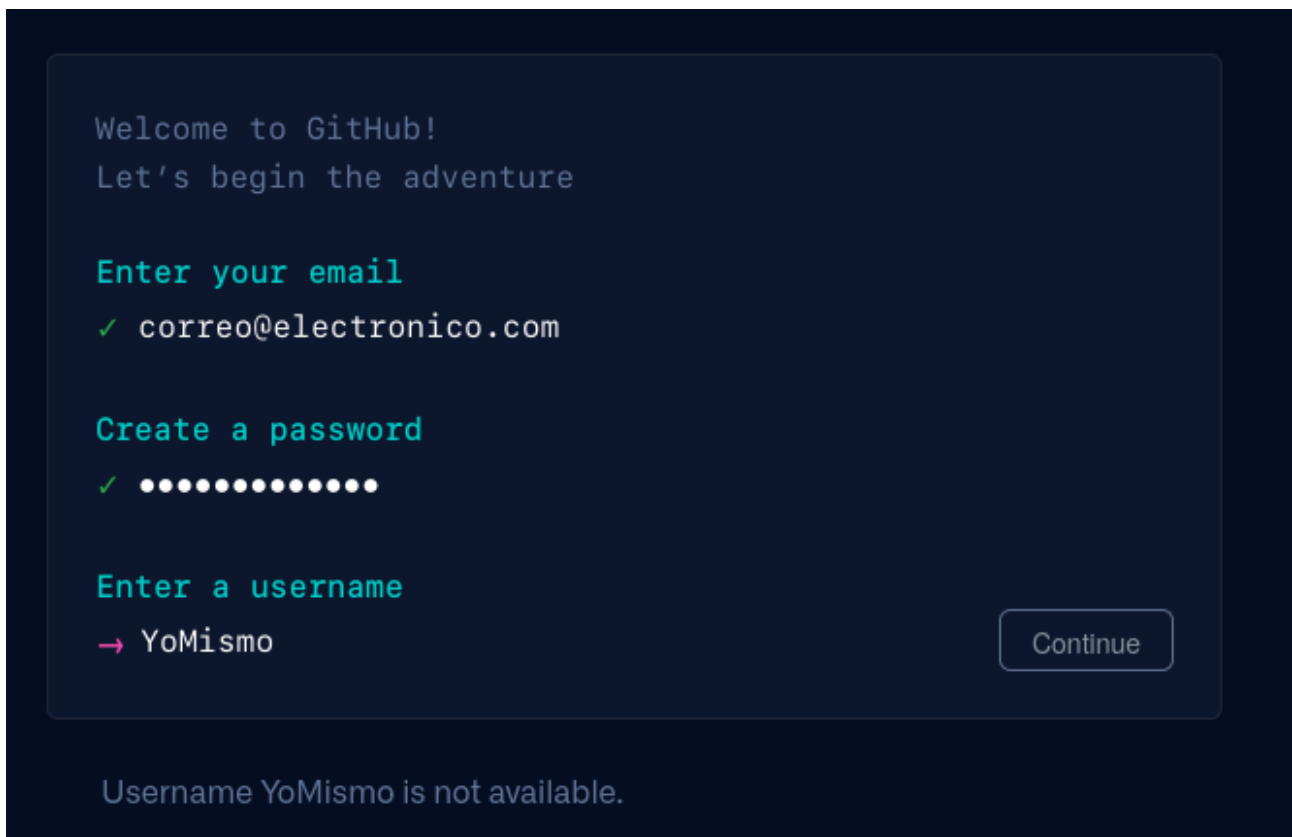
En este curso se usará Linux para los ejemplos y referencias, y se recomienda encarecidamente su uso. Otros sistemas operativos no están tan preparador para algunas tareas como administrar sesiones remotas etc.

En cualquier caso, el uso del propio git en todos ellos es similar por lo que, salvo las particularidades y limitaciones que pueda tener cada uno, no hay ningún problema a la hora de seguir este curso desde otro sistema operativo.

Abrir una cuenta en GitHub

Más adelante en este curso se hablará de GitHub y se darán detalles sobre su uso, pero, por ahora, será suficiente para nosotros con saber abrir una cuenta (que usaremos para enviar nuestros ejercicios).

Desde la propia página principal de la web de [GitHub](https://github.com) y como cualquier otro registro, se nos solicitan sólo tres datos: nombre o nick, e-mail y contraseña:



Wellcome to GitHub!
Let's begin the adventure

Enter your email
✓ correo@electronico.com

Create a password
✓ ●●●●●●●●●●

Enter a username
→ YoMismo

Continue

Username YoMismo is not available.

Formulario de GitHub

Además, como puede verse en la imagen, se nos avisará si el nombre que tratamos de usar ya ha sido elegido por alguien antes.

Instalar git

En Linux

Instalar git en Linux es tan simple como usar tu gestor de paquetes favorito. Por ejemplo (recuerda que normalmente necesitarás privilegios de *root* para instalar cualquier programa):

En Arch Linux

```
# pacman -S git
```

En sistemas Debian, Ubuntu, Mint...

```
# apt-get install git
```

En Gentoo

```
# emerge --ask --verbose dev-vcs/git
```

En sistemas Red Hat, Fedora:

```
# yum install git
```


En Windows

Para instalar git en Windows debemos descargar el programa instalador en su web oficial en <http://git-scm.com/downloads>. La última versión disponible es la 2.35.1.

Una vez descargado, sólo tenemos que ejecutarlo y se abrirá una ventana que nos irá solicitando paso a paso los datos necesarios para la instalación.

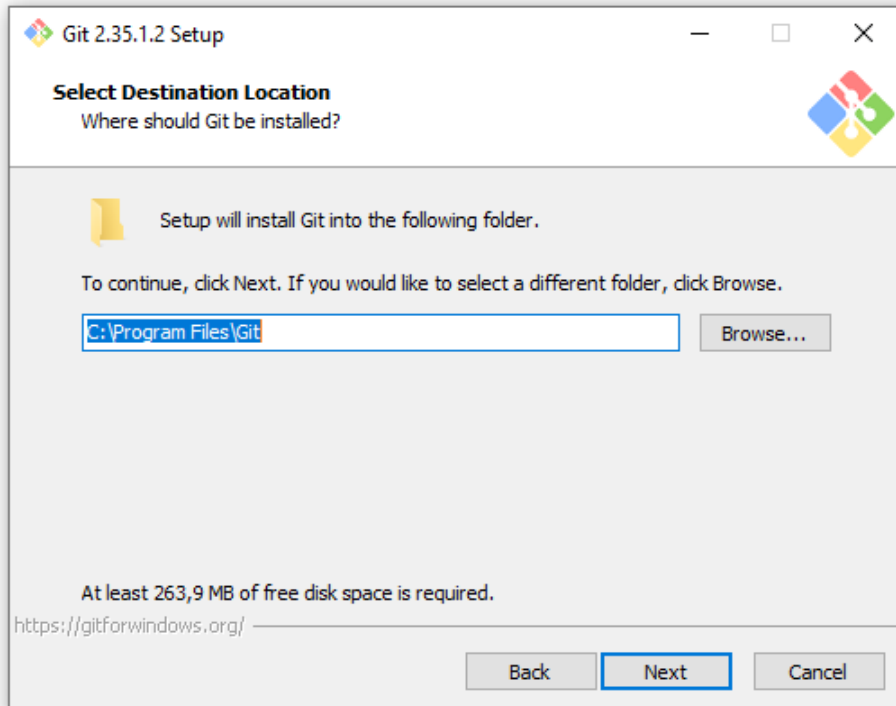
El proceso es un poco tedioso, porque la instalación de git tiene muchas opciones de configuración, y la mayoría se deciden en una pantalla individual, lo que hace que haya que pasar unas quince pantallas.

Afortunadamente, la mayoría podemos dejarlas con su valor por defecto.

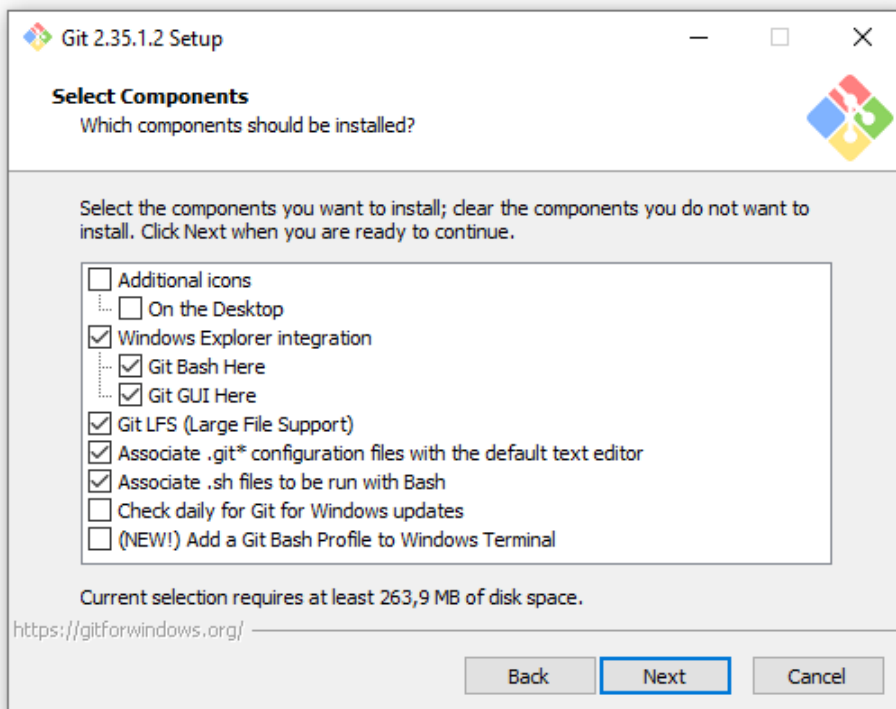
La primera página de la instalación nos muestra la licencia (*es una licencia libre que permite copiar, modificar y distribuir el programa*).



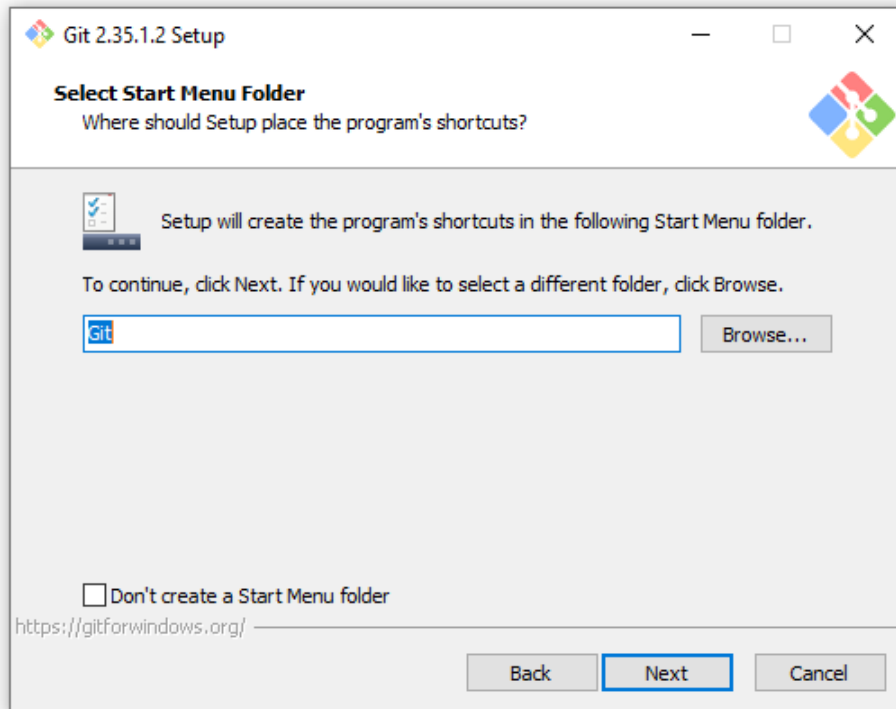
La siguiente es una ventana que nos permite elegir el lugar de instalación. Si no tenemos especial interés en que sea otro, el que viene por defecto está bien.



En la siguiente, podemos elegir una serie de cosas como el que aparezcan iconos de git en Inicio Rápido y el Escritorio o tener dos nuevas órdenes en el menú contextual (*el que aparece al hacer clic derecho con el ratón en una ventana*) para iniciar una ventana de git en esa carpeta.



En la siguiente ventana se nos permite cambiar el nombre del grupo de programas que aparecerá en el menú Inicio

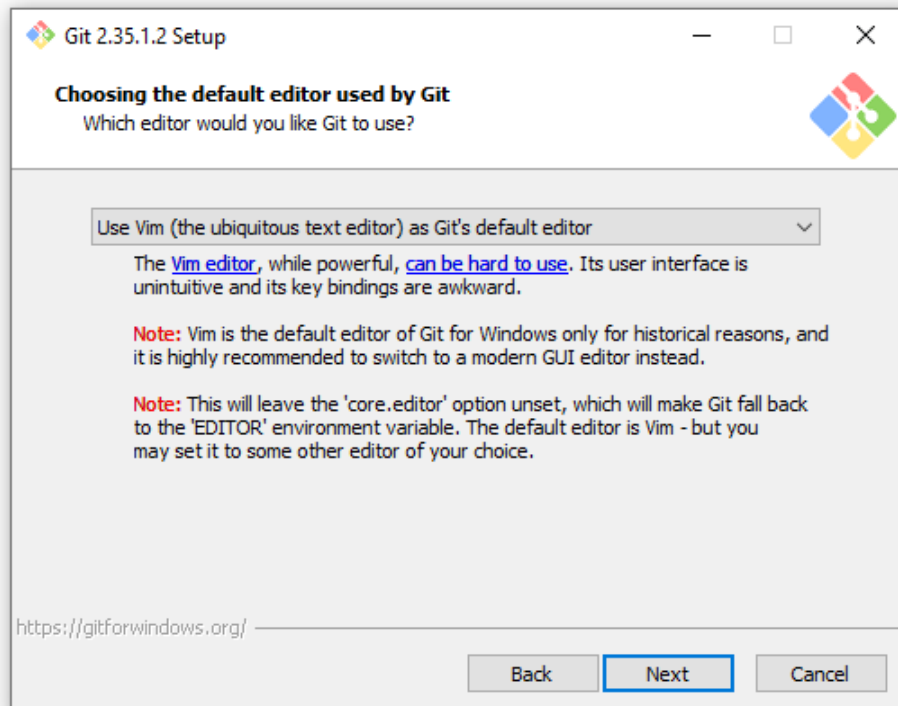


La siguiente página es importante, ya que nos permite decidir el editor de texto que usaremos para redactar los mensajes de commit. (más adelante veremos qué es eso).

Por defecto, git usa el clásico editor vim, pero no es un editor fácil de usar si no tenemos experiencia previa, así que es mejor elegir uno que sepamos usar en el menú desplegable que hay en esta ventana.

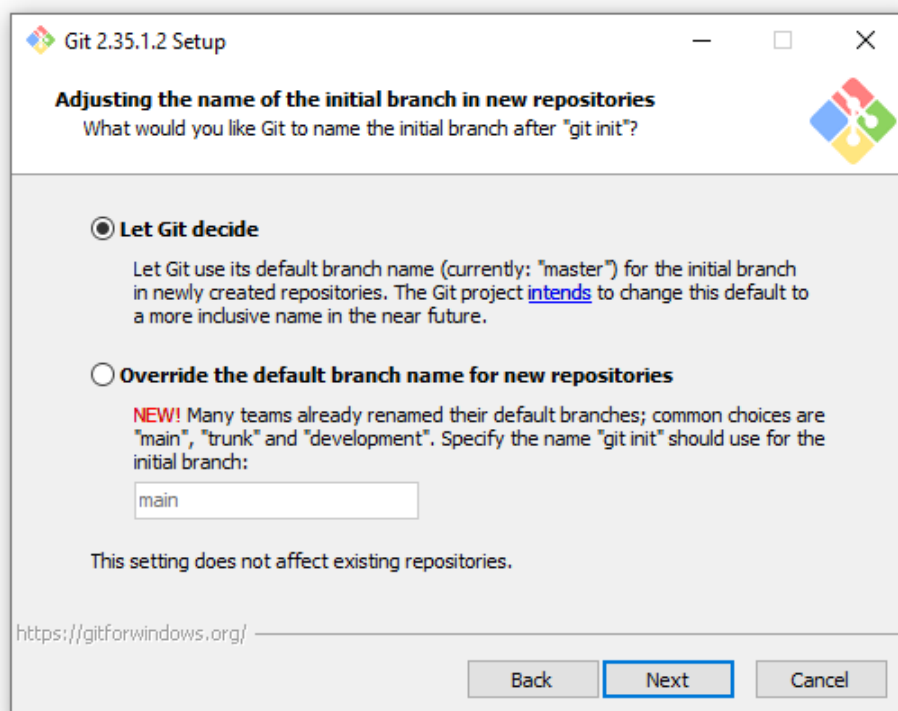
Debe ser un editor de TEXTO PLANO, lo que quiere decir texto sin formato de ningún tipo, como negritas, cursivas, tipos o tamaños de letra.

El editor de texto que Windows tiene por defecto es el bloc de notas (Notepad), y es una buena opción.

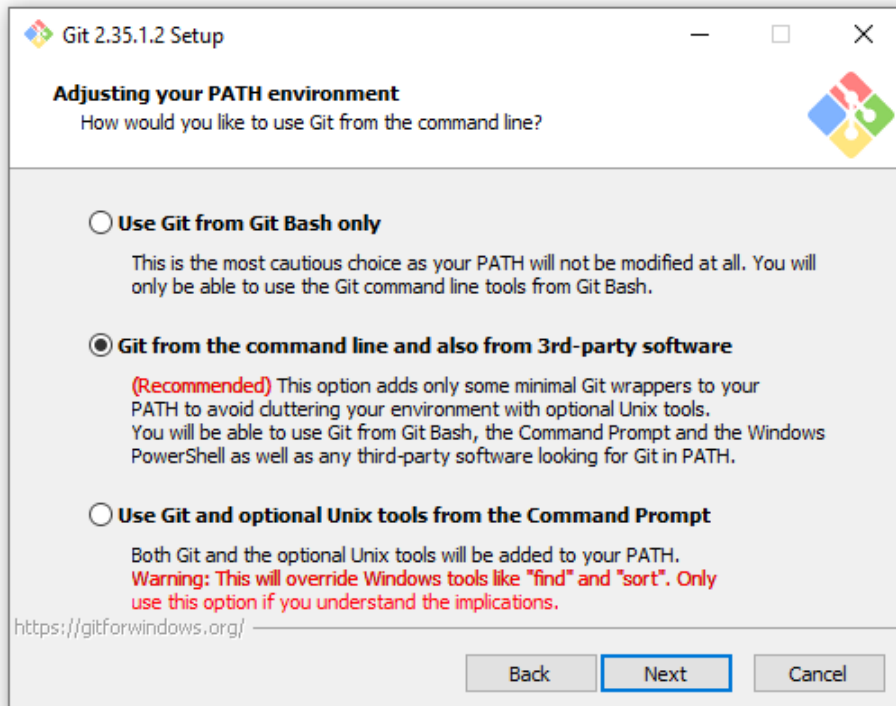


La siguiente pantalla nos permite decidir el nombre por defecto de la rama principal de nuestro proyecto.

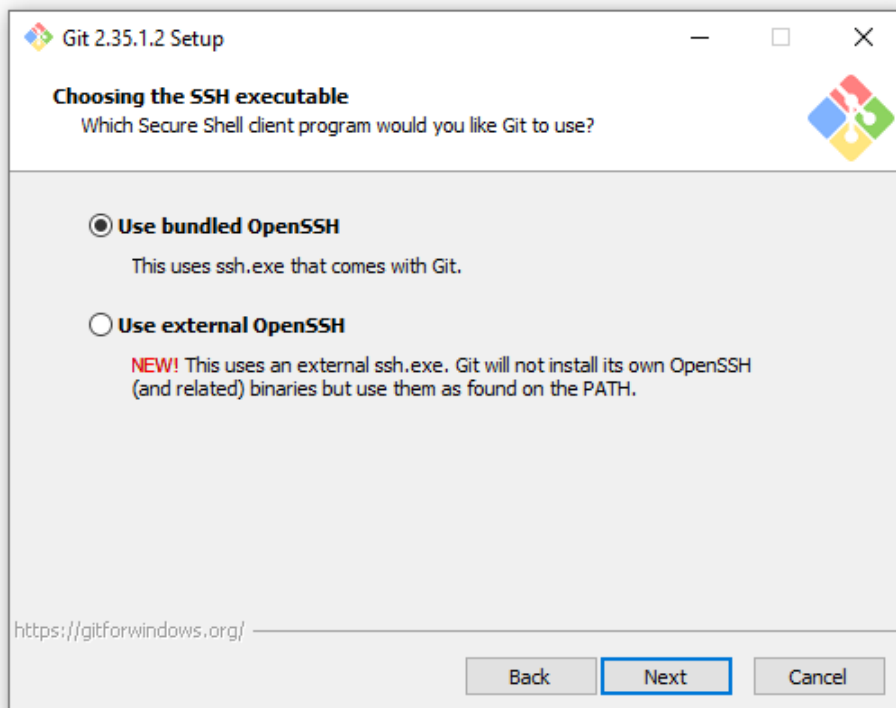
Pronto veremos qué significa eso, pero por ahora nos basta saber que, tradicionalmente, ese nombre era “máster”, pero que ahora se prefiere usar “main”. Es mejor dejar esta opción como está.

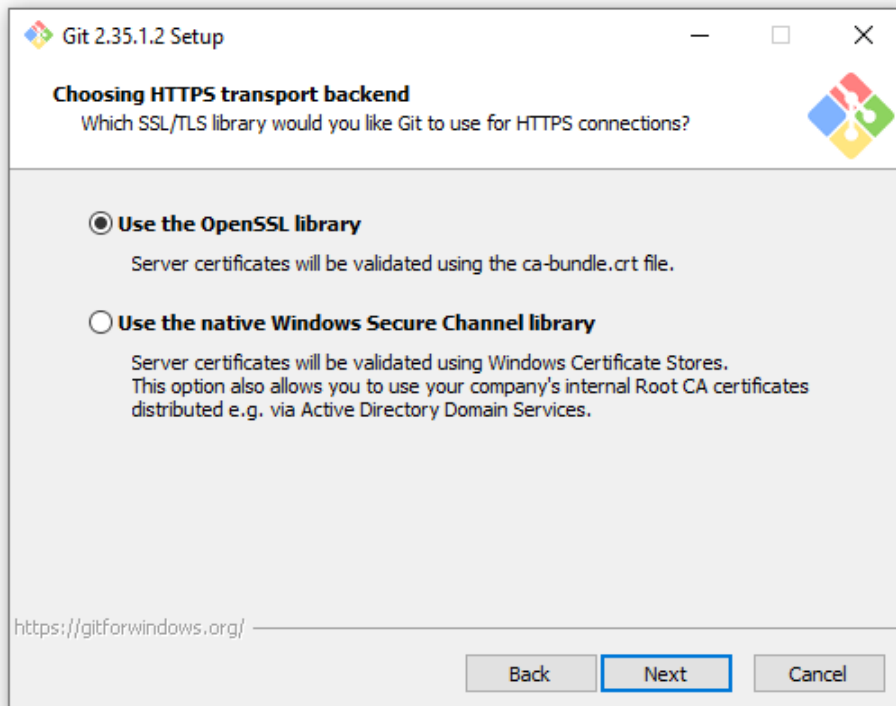


Después de esto, podemos decidir cómo se configurará el PATH (la variable de entorno que almacena la ruta necesaria para ejecutar git) para permitir el uso del prompt (en qué entorno se podrá ejecutar git) La opción por defecto es perfecta.

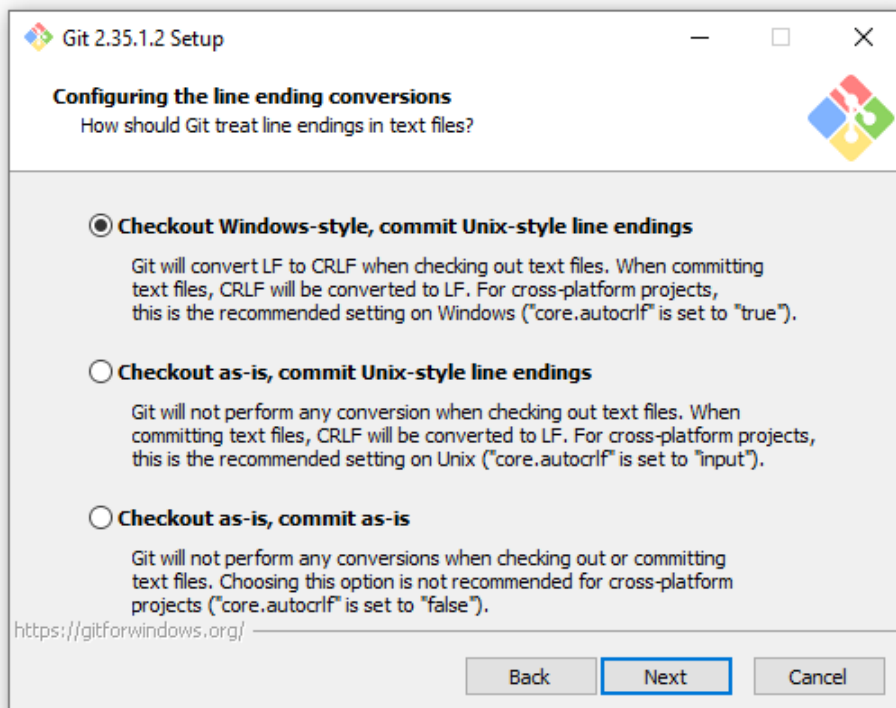


Las siguientes dos páginas permiten cambiar el software que se usará para asegurar las conexiones remotas. Lo más adecuado es dejar ambas como están.

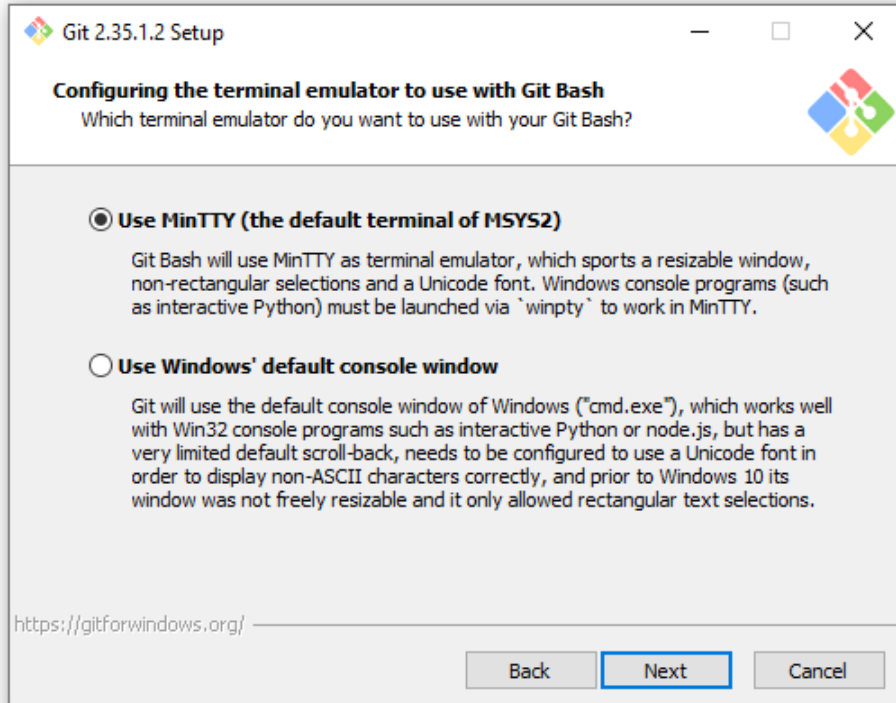




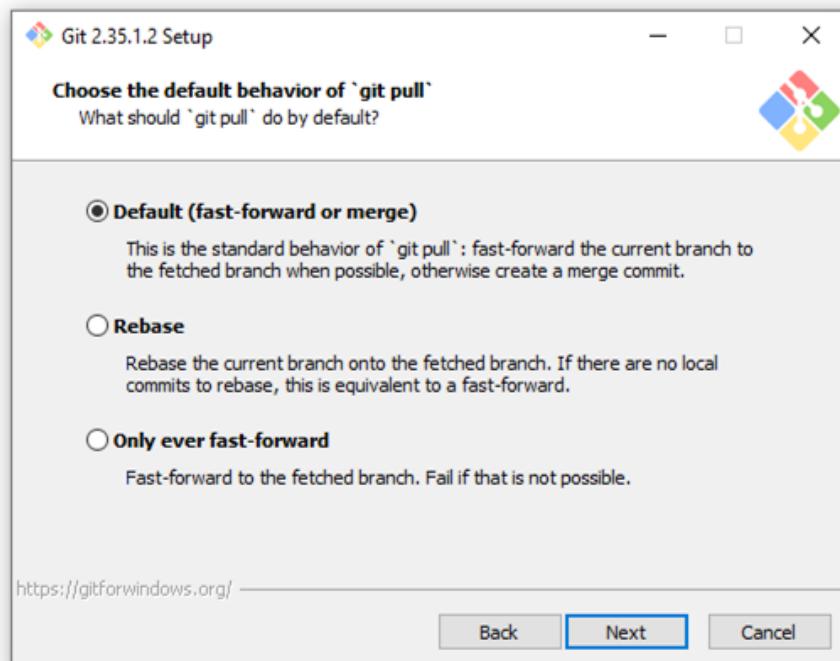
La siguiente opción configura el manejo de retornos de carro. Los sistemas operativos de Windows y los que están basados en UNIX usan caracteres distintos para indicar los finales de línea, lo que puede dar muchos quebraderos de cabeza. La opción por defecto evita la mayoría de los problemas.



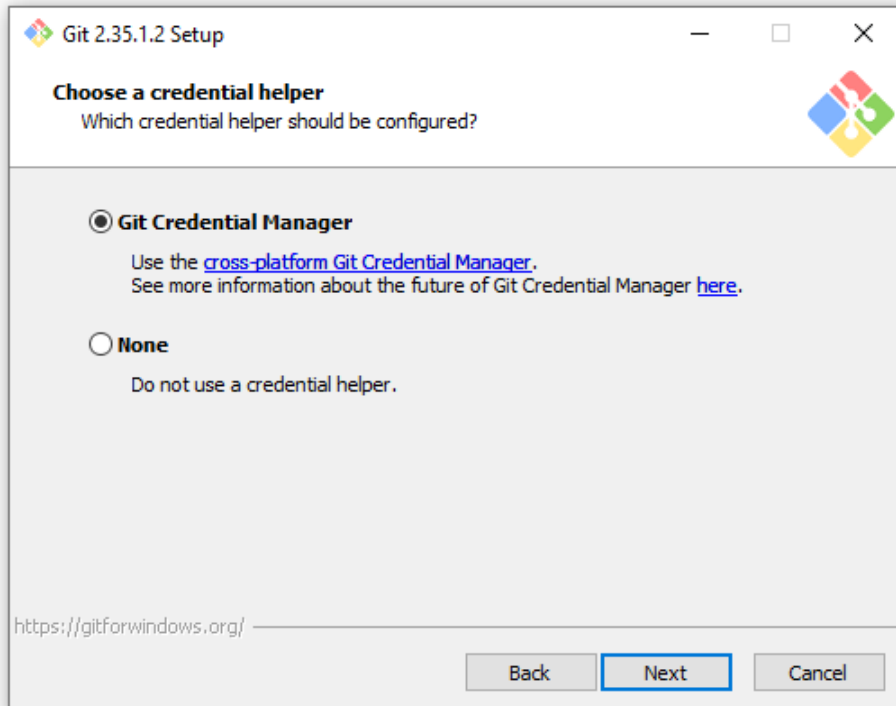
En la siguiente pantalla podemos elegir el emulador de terminal que usará git Bash. Se puede elegir la propia terminal que incorpora la instalación de git o usar la que tiene Windows por defecto. Pero la consola de Windows tiene un diseño francamente horrible, así que no hay motivo para cambiar la opción por defecto.



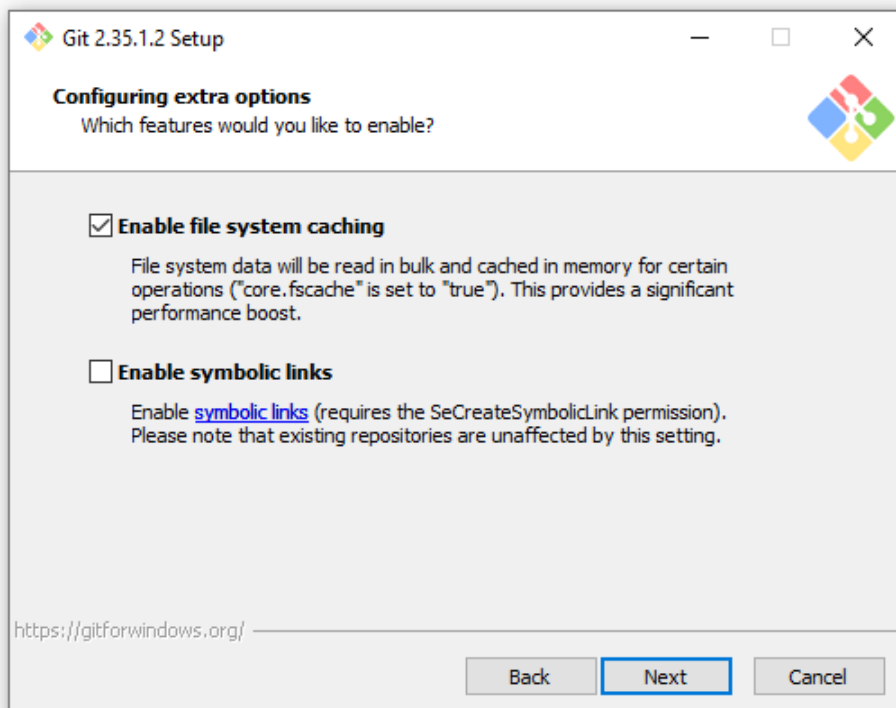
La siguiente opción configura como se usará el comando de git *“git pull”* por defecto. La opción preseleccionada es la más cómoda (al menos, cuando estamos empezando) y, además, es posible cambiar esto posteriormente, así que por ahora lo dejaremos como está.



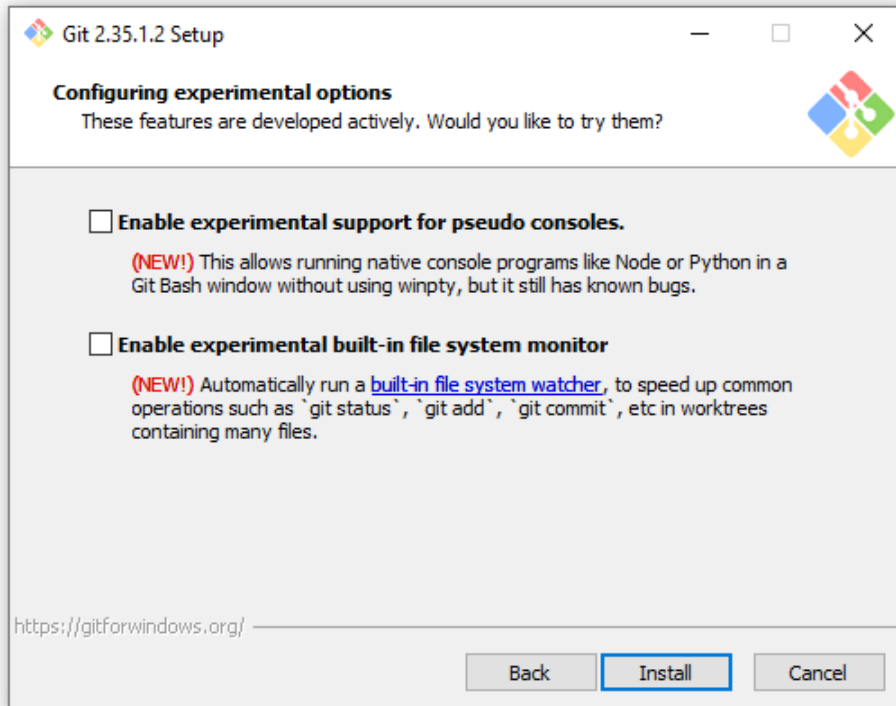
La siguiente pantalla nos permite decidir si git gestionará el manejo de las credenciales. La dejaremos como está.



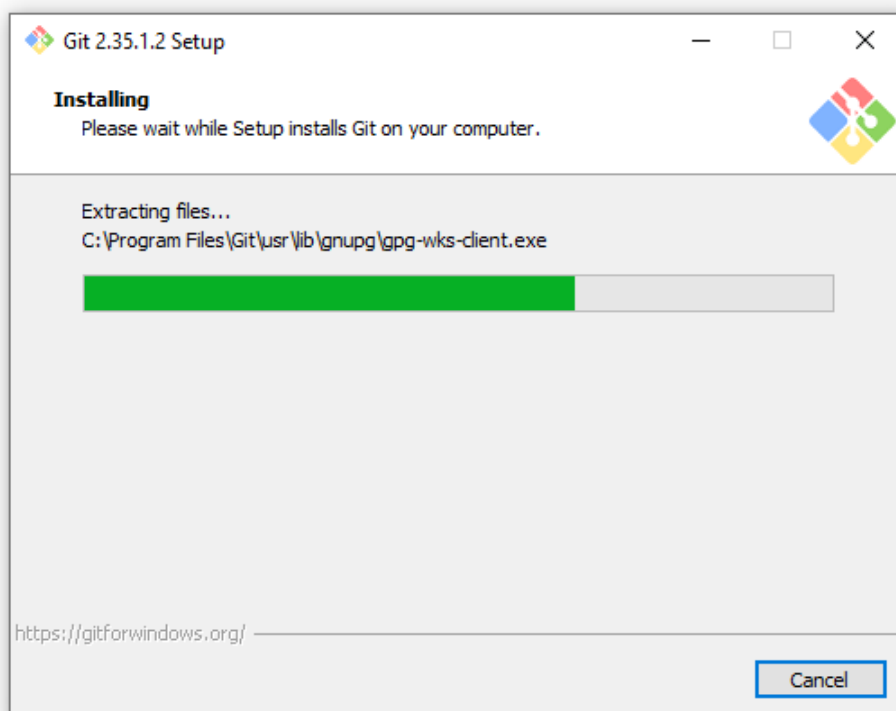
En la penúltima pantalla tenemos opción a configurar algunos aspectos de la gestión de ficheros por parte de git. Los dejaremos con su valor por defecto.



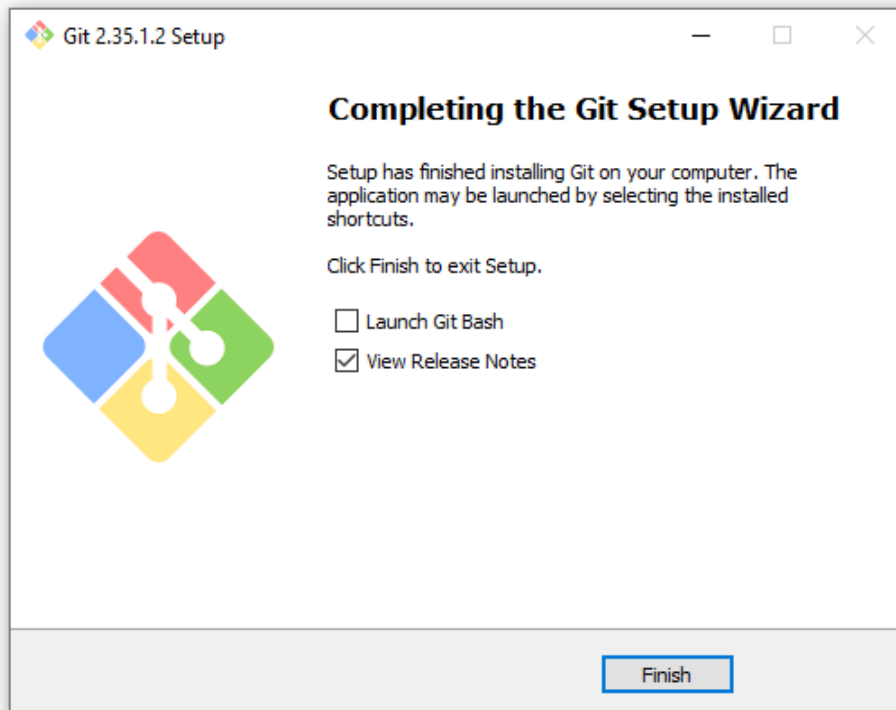
Y, por fin, la última página nos presenta un par de opciones experimentales que no vamos a usar por ahora.



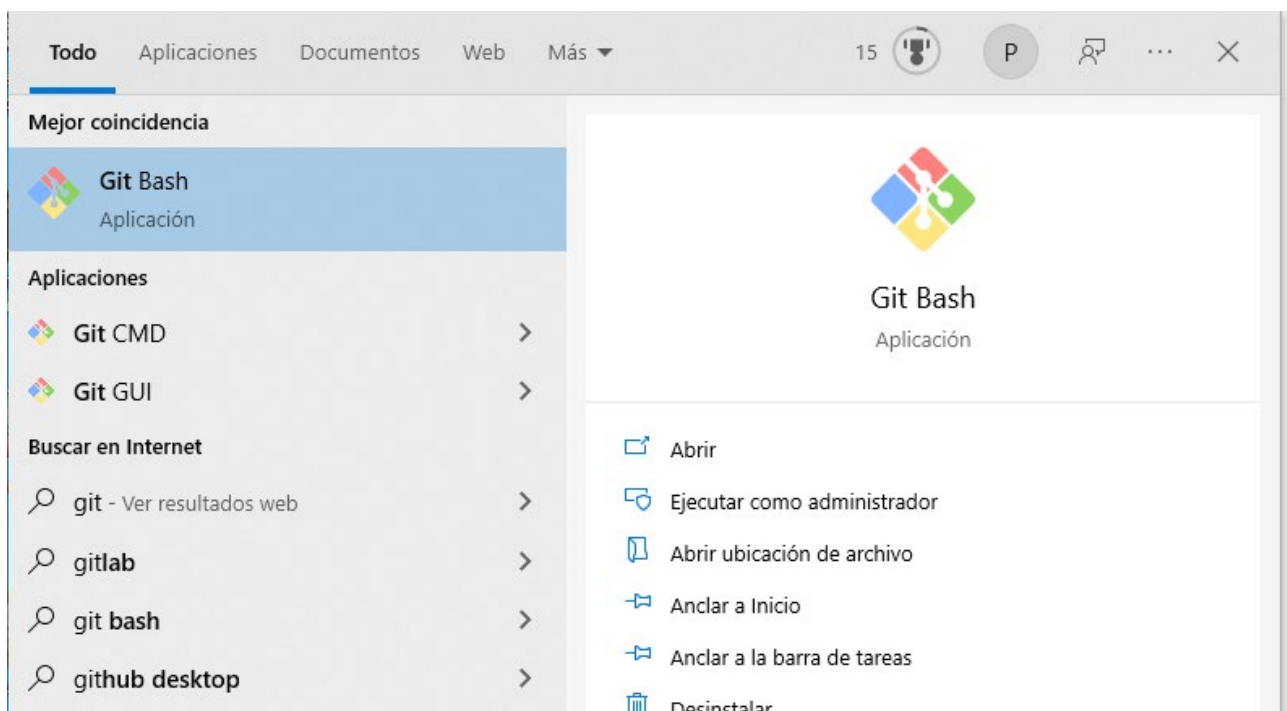
Git ocupa poco espacio, y la instalación no consume demasiado tiempo, aunque depende de la potencia de nuestro sistema.



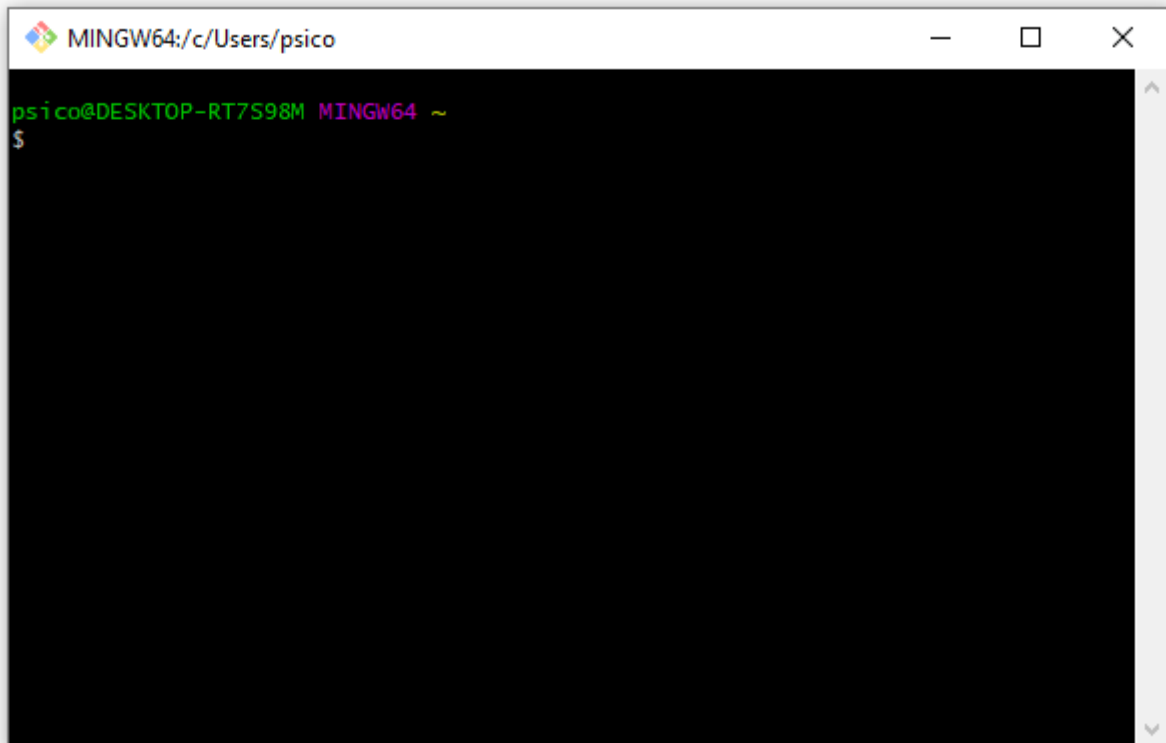
Y, por fin, hemos finalizado nuestra instalación.



A partir de este momento, podemos ir al menú inicio y ejecutar la aplicación "Git Bash", como se ve en la siguiente imagen:



Esto abrirá una consola, que a partir de ahora será la herramienta con la que podremos interactuar con 'git' tal como se ve en este curso.



(Al terminar todos estos pasos, y como se ven en la imagen, también se instalará una versión gráfica "git GUI", pero en este curso se seguirá la interfaz de línea de comandos)

En Mac

Hay dos maneras de instalar git en Mac, la más fácil es utilizar el instalador gráfico:

[git for OS X](#)

Cientes GUI para Linux, Windows y Mac

En este curso se seguirá la interfaz de *línea de comandos* (o *línea de órdenes*), pero existen varias aplicaciones para diversos sistemas operativos que permiten interactuar gráficamente (*Interfaz GUI*) con git de forma más o menos completa.

[GUI Mac](#)

[GUI Windows](#)

[GUI for Linux, Windows y Mac](#)

Empezando a usar git

git es un programa en línea de comandos, y se te supone un conocimiento básico del manejo de esta (cosas como moverse por el árbol de directorios y poco más). No es necesario saber nada complejo, sólo los rudimentos básicos.

Configurar

Lo primero que hay que hacer antes de empezar a usar git es configurar un par de parámetros básicos que nos identifican como usuario, que son nuestro correo electrónico y nuestro nombre.

git usará estos datos para identificar nuestros aportes o modificaciones a la hora de mostrarlos en logs, etc.

Configurar estos parámetros es muy fácil. Desde la línea de comandos escribimos las siguientes órdenes:

```
git config --global user.name "Nombre que quieras mostrar"
```

y

```
git config --global user.email "correo@electroni.com"
```

¿qué acabamos de hacer? Veámoslo, paso a paso:

Todos los comandos de git empiezan con la palabra git.

En este caso, el comando en sí mismo es config, que sirve para configurar varias opciones de git, en el primer caso user.name y en el segundo user.email.

Habrás notado que hay un parámetro --global en cada uno de los comandos. Este sirve para decirle a git que esos datos se aplican a todos los repositorios que abras.

Si quieres que algún repositorio concreto use unos datos distintos, puedes llamar al mismo comando, desde el directorio de ese repositorio, pero usando el parámetro `--local` en lugar de `--global`.

Las opciones que configures como `--global` se almacenarán en un archivo de tu carpeta `home`, llamado `.gitconfig`. Las opciones `--local` lo harán en un archivo `config` dentro del directorio `.git` de tu proyecto.

Hay más opciones que se pueden configurar, puedes verlas (y ver los valores que tienen) con el comando:

```
git config --list
```

Si te has equivocado al escribir alguno de estos datos o quieres cambiarlo, sólo tienes que volver a ejecutar el comando correspondiente de nuevo, y sobrescribirá los datos anteriores.

Una opción de configuración muy cómoda es `git config --global color.ui true`, que hace que el interfaz de git use (si es posible) colores para resaltar distintos aspectos en el texto de sus mensajes.

Iniciando un repositorio

Un repositorio de git no es más que un directorio de nuestro ordenador que está bajo el control de git. En la práctica, esto significa que en el directorio raíz de nuestro proyecto hay otro directivo oculto llamado `".git"` donde se guardan, por ejemplo, los archivos para el control de historiales y los cambios.

Para iniciar un repositorio sólo hay que situarse en el directorio de nuestro proyecto (el que contiene o va a contener los archivos que queremos controlar) y ejecutar la siguiente orden:

```
git init
```

Si todo va bien, este comando responderá algo parecido a `"Initialized empty git repository in /ruta/a/mi/proyecto/.git/"`, que significa que ya tienes creado tu primer repositorio. Vacío, pero por algo hay que empezar.

Clonando un repositorio

Un repositorio también puede iniciarse copiando (*clonando*) otro ya existente.

`git clone REPOSITORIO`

por ejemplo:

`git clone https://github.com/oslugr/repo-ejemplo.git`

git usa su propio protocolo "git" para el acceso remoto (también se puede clonar un repositorio local, simplemente indicando el path), pero también soporta otros protocolos como ssh, http, https...

Al contrario que con git init, con git clone no es necesario crear un directorio para el proyecto. Al clonar se creará un directorio con el nombre del proyecto dentro del que te encuentres al llamar a la orden.

Clonar un repositorio significa copiarlo completamente. No sólo los archivos, sino todo su historial, cambios realizados, etc. Es decir que en tu repositorio local tendrás exactamente lo mismo que había en el repositorio remoto de donde lo has clonado.

Si has clonado el repositorio del ejemplo anterior (y si no, hazlo ahora), podemos ver un par de cosas interesantes. ¿Recuerdas la orden `git config --list`? Entra en el directorio del repositorio (para ello tendrás que hacer algo como `cd repo-ejemplo/`) y lista las opciones de configuración.

Verás, entre otras muchas, las `user.name` y `user.email` que ya conoces. Pero hay otra que es importante, y es `remote.origin.url`, que debe contener la dirección original del repositorio del que has clonado el tuyo.

Ahora mismo no nos sirve de mucho, pero, cuando más adelante trabajemos en red con otros repositorios, nos va a venir bien recordarlo.

IMPORTANTE En adelante, a menos que se diga lo contrario, todos los comandos y órdenes que se indique se deberán ejecutar en el directorio de nuestro proyecto (o uno de sus subdirectorios, lógicamente). git reconoce el proyecto con el que está trabajando en función del lugar donde te encuentres al ejecutar los comandos

¿Cómo funciona git?

Antes de continuar, vamos a detenernos un momento para entender el funcionamiento de git.

Cuando trabajas con git lo haces, evidentemente, en un directorio donde tienes tus archivos, los modificas, los borras, creas nuevos, etc.

Ese directorio es lo que llamamos "Directorio de trabajo", puede contener otros directorios y, de hecho es el que contiene el directorio .git del que hablábamos al principio.

git sabe que tiene que controlar ese directorio, pero no lo hace hasta que se lo digas expresamente.

Más adelante veremos con algo más de detalle la orden git add, pero ya te adelanto que lo que hace es preparar los archivos que le indiques poniéndolos en una especie de lista virtual a la que llamamos el "Index". En Index ponerlos los archivos que hemos ido modificando, pero las cosas que están en el "Index" aún no han sido archivadas por git.

Ojo, que algo esté en el index no significa que se borre de tu directorio de trabajo ni nada parecido, el Index es sólo una lista de cosas que tendrás que actualizar en el repositorio porque han cambiado.

Por último, la instrucción git commit, que también veremos en breve, es la que realmente envía las cosas que hay en el Index al repositorio. Solo que, en lugar de "repositorio" lo vamos a llamar "HEAD", porque el lugar exacto al que va puede significar cosas distintas en según qué casos, como ya veremos cuando hablemos de ramas y esas cosas.

Lo sé, es todo un poco lioso ahora mismo, pero ya se irá aclarando conforme aprendamos más cosas.

Tú sólo mantén esta secuencia en la cabeza: Directorio de trabajo -> Index -> HEAD

Manteniendo nuestro repositorio al día

Tienes tu repositorio iniciado (o clonado) con una serie de archivos con los que empiezas a trabajar, creándolos, editándolos, modificándolos, etc.

Para que git sepa que tiene que empezar a tener en cuenta un archivo (a esto se le llama *preparar* un archivo), usamos la orden `git add` de este modo:

```
git add NOMBRE_DEL_ARCHIVO
```

Esto, como vimos antes, añadirá el archivo indicado con `NOMBRE_DEL_ARCHIVO` al Index. No lo archivará realmente en el sistema de control de versiones ni hará nada. Sólo le informa de que debe tener en cuenta ese archivo para futuras instrucciones (que es, básicamente, en lo que consiste el Index).

Si intentas añadir al Index un archivo que no existe te dará un error.

También puedes usar *comodines*, con cosas como:

```
git add miarchivo.*
```

(que reconocería, por ejemplo "miarchivo.txt", "miarchivo.cosas" y "miarchivo.png")

o

```
git add miarchivo$.txt
```

(que identificaría cosas como "miarchivo1.txt", "miarchivo2.txt" y "miarchivoZ.txt")

Y, en general, todos los comodines que permita usar tu sistema operativo.

Si, en lugar de un archivo, indicas un directorio, se agregarán al Index todos los archivos de ese directorio.

De este modo, la forma más fácil de agregar todos los archivos al Index es mediante la orden:

```
git add .
```

que añadirá el directorio en el que te encuentras y todo su contenido (incluyendo subdirectorios etc).

Un detalle importante es que, si mandas algo al Index con `git add` y luego lo modificas, no tendrás en Index la última versión, sino lo que hayas hecho hasta el momento del hacer el `add`.

Esto es muy útil (a veces tienes que hacer cambios que aún no quieres "archivar") pero puede llevarte a alguna confusión.

Ahora vamos a ver una orden que será tu gran amiga:

`git status`

`git status` te da un resumen de cómo están las cosas ahora mismo respecto a la versión del repositorio (concretamente, respecto al HEAD). Qué archivos has modificado, que hay en el Index, etc (también te cuenta cosas como en qué rama estás, pero eso lo veremos más adelante). Cada vez que no tengas muy claro que has cambiado y qué no, consulta `git status`.

En principio, si no has modificado nada, el mensaje básico que te da `git status` es este:

```
# On branch master
nothing to commit (working directory clean)
```

Pero, y esa es una cosa que vas a ver a menudo en git, si hay algo que hacer te informa de las posibles acciones que puedes llevar a cabo dependiendo de las circunstancias actuales diciendo como, por ejemplo, *use "git add"*.

Cuando ya has hecho los cambios que consideres necesarios y has puesto en el Index todo lo que quieras poner bajo el control de versiones, llega el momento de "hacer commit" (también se le llama "*confirmar*"). Esto significa mandar al HEAD los cambios que tenemos en el Index, y se hace de este modo:

`git commit NOMBRE_DEL_ARCHIVO`

Como te estarás imaginando, aquí también puedes usar comodines del mismo modo que vimos en `git add`. Además, si haces simplemente

`git commit`

Esto mandará todos los cambios que tengas en el Index.

Al hacer un commit se abre automáticamente el editor de texto que tengas por defecto en el sistema, para que puedas añadir un comentario a los cambios efectuados. Si no añades este comentario, recibirás un error y el commit no se enviará.

Puedes cambiar el editor por otro de tu gusto con `git config --global core.editor EDITOR'`, por ejemplo: `git config --global core.editor vim'`

Si no quieres que se abra el editor puedes añadir el comentario en el mismo commit del siguiente modo:

`git commit -m "Comentario al commit donde describo los cambios"`

Recuerda lo que dijimos antes: si modificas un archivo después de haber hecho `git add`, esos cambios no estarán incluidos en tu commit (si quieres incluir la última versión, no tienes más que volver a hacer `git add` antes del commit).

Ahora nos puede surgir un problema:

Si sólo podemos confirmar con commit de un archivo que hayamos preparado con add, y sólo podemos hacer add de un archivo que existe en nuestro directorio de trabajo ¿Cómo le decimos a git que elimine un archivo del repositorio? Para ello tenemos la orden:

```
git add -u
```

que agregará al Index la información de los archivos que deben ser borrados.

Muy similar a la anterior, git add -A sirve para hacer git add -u (preparar los archivos eliminados) y git add . (preparar todos los archivos modificados) en una sola orden.

Un efecto parecido se puede conseguir con

```
git commit -a
```

Esta orden sirve para confirmar todos los cambios que haya en el directorio de trabajo, *aunque no hayan sido preparados* (es decir, aunque no hayas hecho add). Esto incluye tanto los ficheros modificados como los eliminados, con lo que sería equivalente a hacer un git add -A seguido de un git commit.

Esta opción ahorra escribir órdenes, pero también te da más oportunidades de meter la pata. En general se recomienda usar por separado adds y commits, convenientemente salteados de git status para comprobar que todo va bien.

Sincronizando repositorios

Como sistema de control de versiones distribuido, una de las principales utilidades de git es poder mantener distintos repositorios sincronizados (es decir, que contengan la misma información), exportando e importando cambios.

Para importar (o exportar) cambios de un repositorio remoto se necesita, lógicamente, tener acceso de lectura a ese repositorio (En sentido estricto, ya hemos importado el estado de un repositorio cuando lo clonamos al hacer git clone).

Para sincronizar con uno o más repositorios remotos, debemos saber qué repositorios remotos son esos. Para ello tenemos remote, que se usa así:

```
git remote
```

Y, seguramente, te retornará algo parecido a

```
origin
```

Lo que nos dice que el repositorio es el que le indicamos como "origen" al hacer el clone y, la verdad, no es mucha información.

Para obtener algo más útil, prueba a hacerlo con el parámetro -v de este modo:

```
git remote -v
```

lo que te retornará algo parecido a esto:

```
origin https://github.com/oslugr/repo-ejemplo.git (fetch)
origin https://github.com/oslugr/repo-ejemplo.git (push)
```

Esto te dice que hay un repositorio llamado "origin" que se usará tanto para recibir (fetch) como para enviar (push) los cambios. "origin" es el nombre del repositorio remoto por defecto, pero puedes tener muchos más y sincronizar con todos ellos.

Para añadir otro repositorio remoto se hace con la misma instrucción remote de este modo:

```
git remote add ALIAS_DEL_REPOSITORIO DIRECCION_DEL_REPOSITORIO
```

Donde ALIAS_DEL_REPOSITORIO es un nombre corto para usar en las instrucciones de git (el equivalente al "origin" que hemos visto) y DIRECCION_DEL_REPOSITORIO la dirección donde se encuentra. Por ejemplo:

```
git remote add personal git://github.com/psicobyte/repo-ejemplo.git
```

Esto añade un repositorio remoto llamado "personal" con la dirección que se indica.

Si ahora hacemos un git remote -v, veremos algo como:

```
mio git://github.com/psicobyte/repo-ejemplo.git (fetch)
mio git://github.com/psicobyte/repo-ejemplo.git (push)
origin https://github.com/oslugr/repo-ejemplo.git (fetch)
origin https://github.com/oslugr/repo-ejemplo.git (push)
```

Para eliminar un repositorio tienes:

```
git remote rm NOMBRE
```

Y para cambiarle el nombre:

```
git remote rename NOMBRE_ANTERIOR NOMBRE_ACTUAL
```

Nota que git no comprueba si realmente existen los repositorios que agregas o si tienes permisos de lectura o escritura en ellos, de forma que el hecho de que estén ahí no significa que vayas a poder usarlos realmente.

Recibiendo cambios

Ha llegado el momento de importar cambios desde un repositorio remoto. Para ello tenemos git pull que se usa así:

```
git pull REPOSITORIO_REMOTO RAMA
```

el REPOSITORIO_REMOTO es uno de los nombres de repositorio que hemos visto antes (si no pones ninguno, se supone "origin"). Sobre las ramas se hablará un poco más adelante, pero baste decir que, si no ponemos ninguna, se supone que es la rama "master")

de este modo, la forma más usual de llamar esta orden es, simplemente:

```
git pull
```

(que significaría lo mismo que git pull origin master)

Esta instrucción trae del repositorio remoto indicado (o de "origin" si no indicas nada, como hemos visto), todos los cambios que haya respecto al tuyo (lógicamente, no se molesta en traer los que son iguales).

Si el repositorio del que tratas de importar no existe o no tienes permiso de lectura, te dará un mensaje de error advirtiéndote de ello.

Si hay archivos que tú has modificado pero el otro repositorio no, te quedarás con los tuyos. Cuando se trate de archivos que tú no has cambiado pero que sí son distintos en el remoto, actualizarás los tuyos a este último. Pero, si importas archivos que se han modificado en ambos repositorios ¿qué pasa con las diferencias? ¿Sobrescribirá tus archivos? ¿Perderás los del otro repositorio?

Ahí es donde entra la solución de problemas, y lo veremos dentro de poco.

En realidad, git pull es la unión de dos herramientas distintas, que son git fetch, que trae los cambios remotos creando una nueva rama, y git merge, que une esos cambios con los tuyos. En ocasiones te convendrá más usarlas por separado, pero como aún no hemos visto el manejo de las ramas, dejaremos esto por ahora.

Enviando cambios

Si con pull importamos cambios desde otro repositorio, la instrucción push es la que nos permite enviar cambios a un repositorio remoto.

Se usa de un modo bastante parecido:

```
git push REPOSITORIO_REMOTO RAMA
```

Igual que hemos visto con git pull, los valores por defecto son "origin" para el repositorio y "master" para la rama, con lo que se puede poner simplemente:

```
git push
```

Lo que enviará nuestros cambios al servidor remoto.

Salvo que algo haya cambiado allí.

Si la versión que hay en el servidor es posterior a la última que sincronizamos (es decir, alguien más ha cambiado algo), git mostrará un error y no nos dejará hacer el push. Antes debemos hacer un pull.

Sólo cuando hayamos hecho el pull (y resuelto los conflictos, si es que hubiera alguno), nos dejará hacer el push y enviar nuestra versión.

Al hacer tu push, git te retornará información de los cambios realizados, número de archivos, etc.

Contraseñas

Naturalmente, como ya hemos comentado, no puedes hacer push a un repositorio en el que no tengas permiso de escritura. Para eso puede ser que sea un repositorio abierto a todo el que conozca la dirección, pero eso sería muy raro (e inseguro). Lo usual es que cuentes con un usuario y contraseña que te permitan acceder (normalmente por ssh) al servidor.

En otros repositorios (más raros), también necesitarás usuario y contraseña para acceder a la lectura y, por tanto, para hacer pull.

En ambos casos, git te solicitará el nombre de usuario y la contraseña cada vez que hagas push. No tiene por qué ser muy a menudo, pero puede ser un engorro.

En muchos sitios puedes ahorrarte ese trabajo usando pares de claves ssh. Básicamente consiste en que tu ordenador y el del repositorio se reconozcan entre ellos y no tengas que andar identificándote.

Las instrucciones para hacer esto en GitHub están en [esta página de ayuda](#)

El archivo .gitignore

Cuando hacemos `git add .` o algo parecido, preparamos todos los archivos que hayan sido modificados. Esto es, sin duda, mucho más cómodo que ir añadiendo los archivos uno a uno. Pero muy a menudo hay montones de archivos en tu directorio de trabajo que no quieres que se añadan nunca. Archivos de contraseñas, temporales, borradores, binarios compilados, archivos de configuración local...

Por ejemplo, muchos editores de texto mantienen una copia temporal de los archivos que estás editando, con el mismo nombre, pero terminado en el signo "~". Si haces `git add .`, estos archivos se acabarán añadiendo a tu repositorio, cosa que no tiene demasiada utilidad.

Para evitar este problema y facilitarte el trabajo, git nos permite crear un archivo (varios, en realidad, como veremos enseguida) donde describir qué archivos quieres ignorar.

El archivo en cuestión debe llamarse `".gitignore"` (empezando por un punto) y ubicarse en el directorio raíz de tu proyecto.

En este archivo podemos incluir los nombres de archivos que queramos ignorar. Por ejemplo, imaginemos que nuestro `.gitignore` tiene este (poco útil) contenido:

```
# Los archivos que se llamen "passwords.txt" serán ignorados
passwords.txt
```

Las líneas de `.gitignore` que comienzan con el signo "#" son comentarios (útiles para quién lo lea), y git las ignora.

Gracias a esto, git ignorará cualquier archivo que se llame `passwords.txt`, y no los incluirá en tus adds.

Esto es demasiado simple y no nos va a ser muy útil pero, afortunadamente, `.gitignore` permite comodines y otras herramientas útiles.

Por ejemplo:

```
# Ignoramos todos los archivos que terminen en "~"
*~
```

```
# Ignoramos todos los archivos que terminen en ".temp"
*.temp
```

```
# Ignoramos todos los archivos que se llamen "passwords.txt", "passwords.c", "passwords.csv"...
passwords.*
```

El archivo `.gitignore` permite hacer cosas mucho más complejas, aunque para la mayoría de los casos con algo como lo visto arriba es suficiente.

Pese a que cada repositorio puede tener su propio *.gitignore*, puede ser útil tener además un archivo general para todos los repositorios.

git busca por defecto este archivo general en el directorio *".config/git/ignore"* de tu directorio "Home", pero esto puede cambiarse con la siguiente orden:

```
git config --global core.excludesfile RUTA_AL_ARCHIVO_IGNORE
```

Por ejemplo, para usar un archivo llamado "ignorar" en mi directorio personal, pondría algo así:

```
git config --global core.excludesfile ~/ignorar
```

El símbolo "~" en un path significa "El directorio Home del usuario"

Puedes encontrar muchos ejemplos de archivos *.gitignore* en este [repositorio de GitHub](#)

Comportamiento por defecto de push

Las versiones anteriores de git tenían un comportamiento por defecto a la hora de hacer push llamado 'matching'.

Este consiste en que, al hacer push, se sincronizan todas las ramas del proyecto con sendas ramas en el servidor con el mismo nombre (ya hablaremos en detalle de las ramas más adelante). Si en el servidor no existe una rama con el nombre de alguna local, se crea automáticamente.

La versión 2 de git cambiará ese comportamiento, que pasará a ser simple, lo que significa que se sube sólo la rama que tienes activa en este momento a la rama de la que has hecho el pull, pero te dará un error si el nombre de esa rama es distinto.

Mientras tanto, actualmente, git te avisa (a cada push) de que se va a hacer este cambio y te avisa de que puedes configurar este comportamiento por defecto con un mensaje como este:

```
warning: push.default is unset; its implicit value is changing in
git 2.0 from 'matching' to 'simple'. To squelch this message
and maintain the current behavior after the default changes, use:
```

```
git config --global push.default matching
```

To squelch this message and adopt the new behavior now, use:

```
git config --global push.default simple
```

When push.default is set to 'matching', git will push local branches to the remote branches that already exist with the same name.

In git 2.0, git will default to the more conservative 'simple' behavior, which only pushes the current branch to the corresponding remote branch that 'git pull' uses to update the current branch.

See 'git help config' and search for 'push.default' for further information. (the 'simple' mode was introduced in git 1.7.11. Use the similar mode 'current' instead of 'simple' if you sometimes use older versions of git)

Para elegir el comportamiento que prefieres sólo tienes que usar, como ya hemos visto para otras configuraciones, el comando git config de este modo:

```
git config --global push.default OPCION
```

Por ejemplo:

```
git config --global push.default matching
```

Usaría la opción matching en todos tus repositorios, pero:

```
git config --local push.default simple
```

Usaría la opción simple sólo en el repositorio en el que te encuentras.

Otras opciones posibles son:

- current: Sube los cambios de la rama activa a una rama remota del mismo nombre. Si no existe esa rama remota, se crea.
- nothing: Esta opción sólo tiene sentido para test, debugs y esas cosas. Al hacer push no se subirá nada a repositorio remoto.
- upstream: Al igual que simple, sube la rama que tienes activa a la rama de la que has hecho el pull pero, en este caso, *no* te dará error si el nombre de esa rama es distinto.

Obtener Ayuda

Lo primero que necesitamos a la hora de enfrentarnos a las dificultades es conocer nuestras herramientas.

git dispone de una ayuda detallada que nos resultará muy útil. Para invocarla sólo hay que hacer

```
git help
```

también se puede obtener ayuda de un comando concreto con git help COMANDO, por ejemplo:

```
git help commit
```


Viendo el historial

Has hecho una serie de modificaciones seguidas de commits con sus comentarios ¿Cómo puedes ver todo eso? Para ello tienes la instrucción

`git log`

La orden `git log` te mostrará un listado de todos los cambios efectuados, con sus respectivos comentarios, empezando desde el más reciente hasta el más antiguo.

El formato de cada commit en la respuesta es como este:

```
commit c2ac7c356156177a50df5b4870c72ce01a88ae63 Author: psicobyte <psicobyte@gmail.com>
Date: Sun Mar 30 11:54:15 2014 +0200 Cambios menores en las explicaciones de git add y git status
```

La primera línea es un *hash* único que identifica al commit (y que más adelante nos será muy útil), seguida del autor, la fecha en que se hizo y el comentario que acompañó al commit.

Por defecto, `git log` nos mostrará todas las entradas del log. para ver sólo un número determinado sólo tienes que añadirle el parámetro `-NUMERO`, donde `NUMERO` es el número de entradas que quieres ver, por ejemplo:

`git log -4`

mostrará las últimas cuatro entradas en el log.

Otra opción posible, si quieres ver una versión más resumida y compacta de los datos, es `--oneline`, que te muestra una versión compacta.

Si, por el contrario, quieres más detalles, la opción `-p` te mostrará, para cada commit, todos los cambios que se realizaron en los archivos (en formato diff).

Otra ayuda visual es `--graph`, que dibuja (con caracteres ASCII) un árbol indicando las ramas del proyecto (ya veremos eso un poco más adelante).

`git log` tiene un montón de opciones más (para filtrar por autor o fecha, mostrar estadísticas...) que, además, se pueden usar en combinación. Por ejemplo, la instrucción

`git log --graph --oneline`

Mostrará los commits en versión compacta y dibujando las ramas (cuando las haya), dando una salida parecida a esta:

```
* 6ad05c1 Sólo una cosilla * 0678363 Resuelve conflicto como ejemplo |\ | * bf454ef Prueba para crear
conflictos. Así mismo. * | 8785174 Añade título nueva sección |/ * afee5ab Acaba un-solo-usuario y listo
para conflictos * fd04eff Corregido (más o menos) el markdown)
```

Para más detalles, recuerda que `git help log` es tu amigo.

Borrado de archivos

En git se pueden borrar archivos con la orden `git rm`.

```
git rm NOMBRE_DEL_FICHERO
```

Funciona como la propia orden del sistema operativo, con la salvedad de que *también* borra el archivo del Index, si estuviera allí. Esto lo hace muy útil en ocasiones.

Si necesitas borrar el archivo del Index pero sin borrarlo de Directorio de trabajo (porque, por ejemplo, te has arrepentido y no quieres incluirlo en el próximo commit), se puede hacer con la opción `--cached` del siguiente modo:

```
git rm --cached NOMBRE_DEL_FICHERO
```

Otra opción para hacer esto mismo es con `git reset HEAD` que se usa del siguiente modo:

```
git reset HEAD NOMBRE_DEL_ARCHIVO
```

Rehacer un commit

Puedes rehacer el último commit usando la opción `--amend` de este modo:

```
git commit --amend
```

Si no has modificado nada en tus archivos, esto simplemente te permitirá reescribir el comentario del commit pero, si por ejemplo habías olvidado añadir algo al Index, puedes hacerlo antes del `git commit --amend` y se aplicará en el commit.

Deshacer cambios en un archivo

Has cambiado un archivo en tu directorio de trabajo, pero te arrepientes y quieres recuperar la versión del HEAD (la del último commit). Nada más fácil que:

```
git checkout -- NOMBRE_DEL_ARCHIVO
```

Resolviendo conflictos

Normalmente los conflictos suceden cuando dos usuarios han modificado la misma línea, o bien cuando han modificado un fichero binario; por eso los ficheros binarios **no** deben estar en un repositorio. Te aparecerá un conflicto de esta forma cuando vayas a hacer push

```
To git@github.com:oslugr/curso-git.git
! [rejected]      master -> master (non-fast-forward)
error: failed to push some refs to 'git@github.com:oslugr/curso-git.git'
consejo: Updates were rejected because the tip of your current branch is behind
consejo: its remote counterpart. Merge the remote changes (e.g. 'git pull')
consejo: before pushing again.
consejo: See the 'Note about fast-forwards' in 'git push --help' for details.
```

El error indica que la *punta* de tu rama está detrás de la rama remota (es decir, que hay modificaciones posteriores a tu última sincronización). Rechaza por lo tanto el push, pero vamos a hacer pull para ver qué es lo que ha fallado

```
remote: Counting objects: 7, done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 4 (delta 3), reused 0 (delta 0)
Unpacking objects: 100% (4/4), done.
De github.com:oslugr/curso-git
afee5ab..bf454ef master -> origin/master
Auto-merging texto/mas-usos.md
CONFLICTO(contenido): conflicto de fusión en texto/mas-usos.md
Automatic merge failed; fix conflicts and then commit the result.
```

El conflicto de fusión te indica que no hay forma de combinar los dos ficheros porque hay cambios en la misma línea. Así que hay que arreglar los conflictos. En general, si se trata de este tipo de conflictos, no es complicado.

Al mirar el fichero aparecerá algo así

```
<<<<<<< HEAD
## Resolviendo conflictos en `git`
=====
## Vamos a ver cómo se resuelven los problemas en git
>>>>>>> bf454eff1b2ea242ea0570389bc75c1ade6b7fa0
```

Lo que uno tiene está entre la primera línea y los signos de igual; lo que hay en la rama remota está a continuación y hasta la cadena que indica el número del commit en el cual está el conflicto. Resolver el conflicto es simplemente borrar las marcas de conflicto (los <<<<< y los >>>>> y los =====) y elegir el código con el que nos quedamos; en este caso, como se ve, es el que aparece efectivamente en este capítulo.

Una vez hecho eso, se puede ya hacer push directamente sin ningún problema.

Retrocediendo al pasado

Para recuperar el estado de tu directorio de trabajo tal como estaba en algún momento del pasado, primero necesitas saber qué momento es ese. Eso se consigue con `git log` que, como vimos, nos devuelve (entre otras cosas) un hash que identifica al commit.

Con ese hash ya podemos hacer un `git reset` del siguiente modo:

```
git reset --hard HASH_DEL_COMMIT_A_RECUPERAR
```

Eso, en cierto modo, volverá atrás en el tiempo, deshará los commits posteriores al indicado y traerá a tu directorio de trabajo los archivos tal y como estaban entonces. Todos los cambios posteriores desaparecerán, así que mucho cuidado.

Como consejo, recuerda hacer `push` antes de jugar con `git reset --hard`. De este modo, si quieres recuperar todo el trabajo posterior, no tienes más que hacer `pull` y los recuperarás de nuevo.

También te puedes salvaguardar usando otra rama para hacer el `git reset --hard` sobre ella, pero el uso de ramas es algo que veremos un poco más adelante.

Viendo (y recuperando) archivos antiguos

Puedes ver los cambios que hiciste en un commit si haces

```
git show HASH_DE_UN_COMMIT
```

Esto puede ser muy útil, pero aún hay más. Si haces

```
git show HASH_DE_UN_COMMIT:ruta/a/un/archivo
```

te mostrará el estado de ese archivo en aquel commit.

Esto nos va a servir para hacer un pequeño truco:

```
git show HASH_DE_UN_COMMIT:ruta/a/un/archivo > archivo_copia
```

La orden anterior nos permite redireccionar la salida de `git show` a un archivo llamado `archivo_copia`, con lo que obtendremos una copia del archivo tal y como estaba en el commit indicado.