

Git y Github

AUTORES: DETKE, Ramiro Fernando, FINOCHIETTO, Jorge Manuel

Índice

Introducción	2
Sistemas de Control de Versiones	3
Git	4
Instalación	5
Windows	5
Áreas y Estados	12
Comandos Básicos	15
Github	16
¿Cómo empezar?	16
Proyectos	16
¿Cómo crear un Tablero de Proyecto?	17
Incidencias	18
¿Cómo crear una Incidencia?	19
Metadatos	20
Hitos	20
Flujo de Trabajo	22
Ramas	22
Opciones comunes	24
Creación de ramas	24
Creación de ramas remotas	25
Eliminación de ramas	26
Gitflow	26
Ramas en desarrollo y maestras	28
Ramas de función	29
Ramas de publicación	30
Ramas de corrección	32
GitHub Flow	35
Referencias	40

Introducción

Objetivos

- Introducir el concepto de sistema de control de versiones, su funcionamiento general y su relevancia en el proceso de desarrollo de software
- Comprender el proceso de instalación y de uso de la herramienta Git para el control de versiones de un proyecto de software
- Crear y gestionar un proyecto de software a través de un repositorio en Github, generando un flujo de trabajo con ramificaciones

Sistemas de Control de Versiones

En el proceso de desarrollo de software es un requisito casi indispensable mantener un registro de los cambios que se realizan sobre el código fuente a lo largo del tiempo. Es debido a esto que cobran importancia los sistemas de control de versiones. Estos sistemas son herramientas que permiten realizar un seguimiento de los cambios y también permiten proteger el código de errores humanos accidentales. Además, un sistema de control de versiones facilita el trabajo en equipo a la hora de desarrollar software, ya que mientras un integrante trabaja en alguna funcionalidad específica, otro podría estar trabajando en alguna corrección de errores o bien en otra funcionalidad, para luego integrar las soluciones y realizar una sincronización del trabajo de cada uno.

El uso de un sistema de control de versiones tiene tres ventajas principales:

1. Gracias al historial de cambios se puede saber el autor, la fecha y notas escritas sobre los cambios realizados. También permite volver a versiones anteriores para ayudar a analizar causas raíces de errores y es crucial cuando hay que solucionar problemas de versiones anteriores.
2. Creación y fusión de ramas. Al tener varios integrantes del equipo trabajando al mismo tiempo, cada uno en una tarea diferente, pueden beneficiarse de tener flujos de trabajo independientes. Posteriormente se pueden fusionar estos flujos de trabajos o ramas a una principal. Los sistemas de control de versiones tienen mecanismos para identificar que los cambios entre ramas no entren en conflicto para asegurar la funcionalidad y la integración.
3. Trazabilidad de los cambios que se hacen en el software. Poder conectar el sistema de control de versiones con un software de gestión de proyectos y seguimiento de errores ayuda con el análisis de la causa raíz de los problemas y con la recopilación de información.

El concepto de versión (también llamado revisión o edición) de un proyecto (código fuente) hace referencia al estado en el que se encuentra el mismo en un momento dado de su desarrollo o modificación. Los sistemas de control de versiones utilizan repositorios para almacenar el proyecto actualizado junto a sus cambios históricos. Los sistemas de control de versiones centralizados almacenan todo el código en un único repositorio, es decir que un único servidor contiene todos los archivos versionados. Esto representa un único punto de falla dado que si el servidor no está disponible por un tiempo nadie podrá colaborar o guardar cambios en archivos en los que hayan estado trabajando.

Los sistemas de control de versiones distribuidos permiten en cambio continuar el trabajo aún cuando el repositorio de referencia no está disponible. En estos sistemas los clientes no solo descargan la última copia del código, sino que se replica completamente el repositorio con los cambios históricos (versiones). De esta manera, si un servidor deja de funcionar y estos sistemas estaban colaborando a través de él, cualquiera de los repositorios disponibles en los clientes puede ser copiado al servidor con el fin de restaurarlo.

Git

Git es un proyecto de código abierto maduro y con un activo mantenimiento desarrollado originalmente por Linus Torvalds. Este sistema de control de versiones distribuido que funciona bajo cualquier plataforma (Windows, MacOS, Linux, etc) y está integrado en una amplia variedad de entornos de desarrollo (IDEs). Este sistema presenta una arquitectura distribuida, es decir que, cada desarrollador posee una copia del trabajo en un repositorio local donde puede albergar el historial completo de todos los cambios y, mediante determinados comandos, realiza sincronizaciones al repositorio remoto.

Git fue diseñado teniendo en cuenta las siguientes características

- **Rendimiento:** Los algoritmos implementados en Git aprovechan el profundo conocimiento sobre los atributos comunes de los auténticos árboles de archivos de código fuente, cómo suelen modificarse con el paso del tiempo y cuáles son los patrones de acceso. El formato de objeto de los archivos del repositorio de Git emplea una combinación de codificación delta (que almacena las diferencias de contenido) y compresión, y guarda explícitamente el contenido de los directorios y los objetos de metadatos de las versiones.
- **Seguridad:** la principal prioridad es conservar la integridad del código fuente gestionado. El contenido de los archivos y las verdaderas relaciones entre estos y los directorios, las versiones, las etiquetas y las confirmaciones, están protegidos con un algoritmo de hash criptográficamente seguro llamado "SHA1". De este modo, se salvaguarda el código y el historial de cambios frente a las modificaciones accidentales y maliciosas, y se garantiza que el historial sea totalmente trazable.
- **Flexibilidad:** es flexible en varios aspectos, en la capacidad para varios tipos de flujos de trabajo de desarrollo no lineal, en su eficiencia en proyectos tanto grandes como pequeños y en su compatibilidad con numerosos sistemas y protocolos. Se ha ideado para posibilitar la ramificación y el etiquetado como procesos de primera importancia y las operaciones que afectan a las ramas y las etiquetas (como la fusión o la reversión) también se almacenan en el historial de cambios.

Instalación

Windows

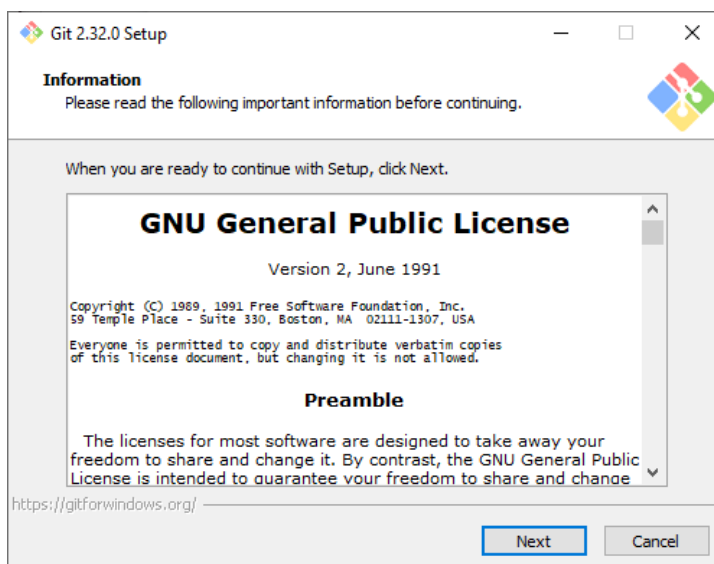
Para instalar Git hay que seguir los siguientes pasos:

1. Descargar la versión adecuada, según el sistema operativo, desde este [enlace](#).

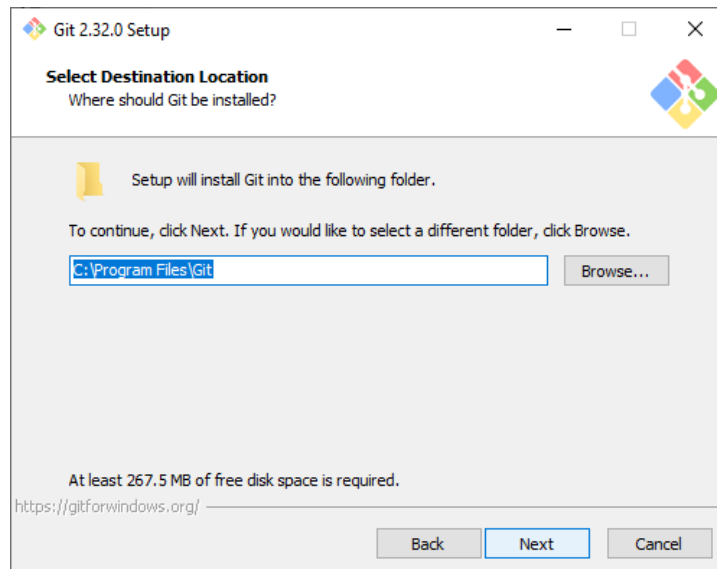
Downloading Git



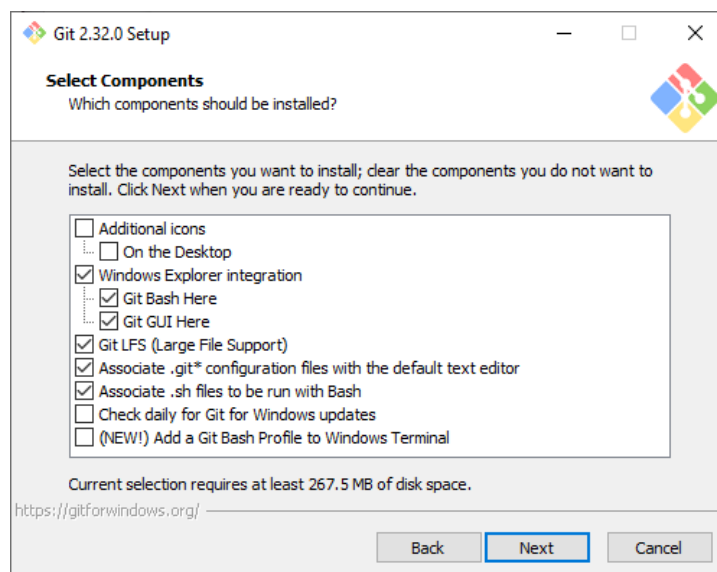
2. Ejecutar el instalador (capturas del instalador versión *Git-2.32.0-64-bit*), leer y aceptar los términos y condiciones de las Licencias .



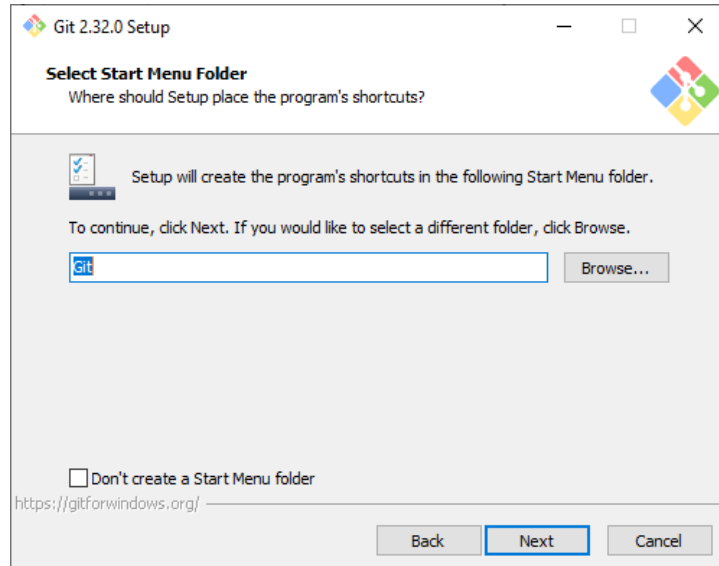
3. Elegir el destino de la instalación



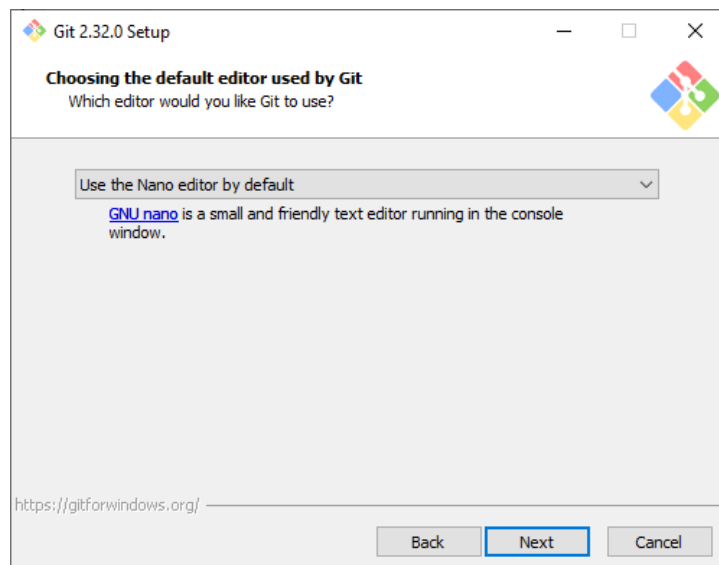
4. Seleccionar los componentes que se van a instalar.



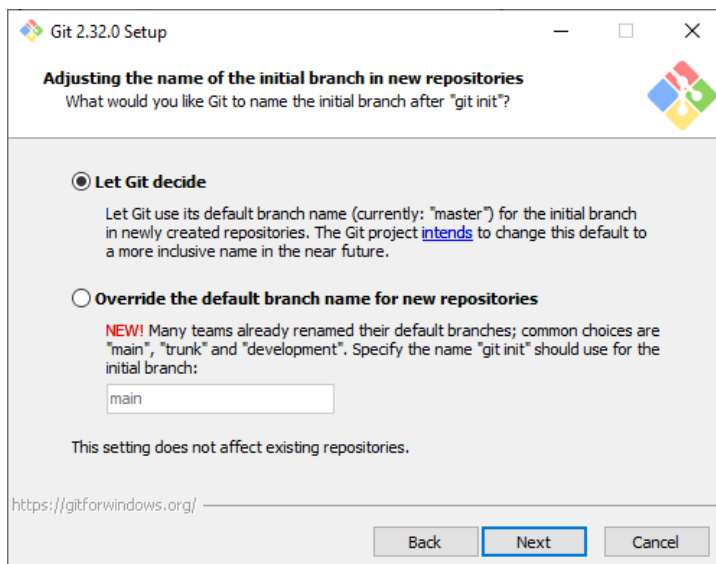
5. Asignar un nombre a la entrada que aparecerá en el menú de inicio de Windows.



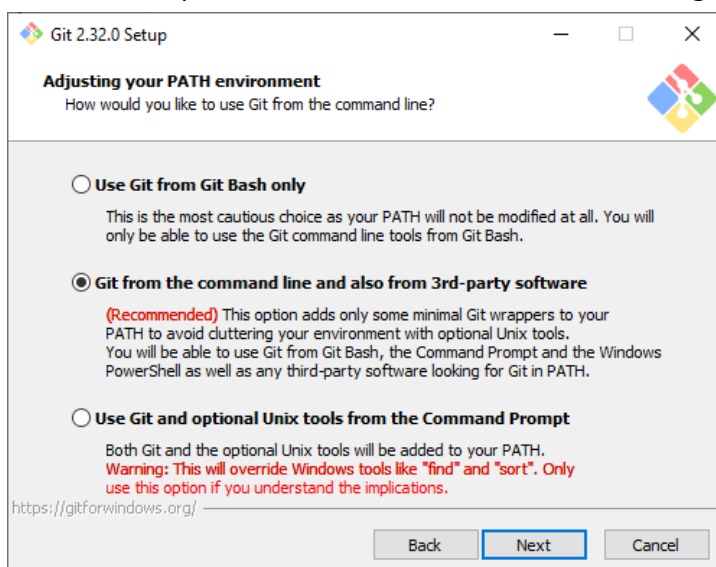
6. Seleccionar el editor que se utilizará para los comentarios y configuraciones de Git.



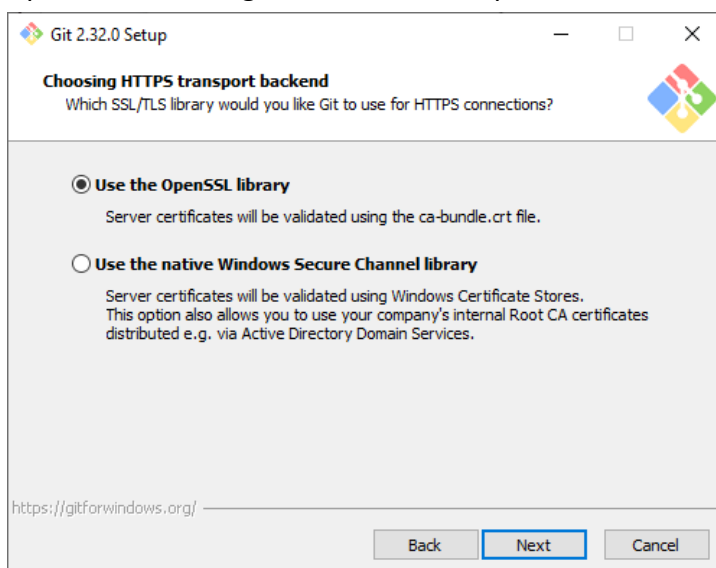
7. Elegir el nombre que se le asignará a la rama principal durante la creación de un nuevo repositorio.



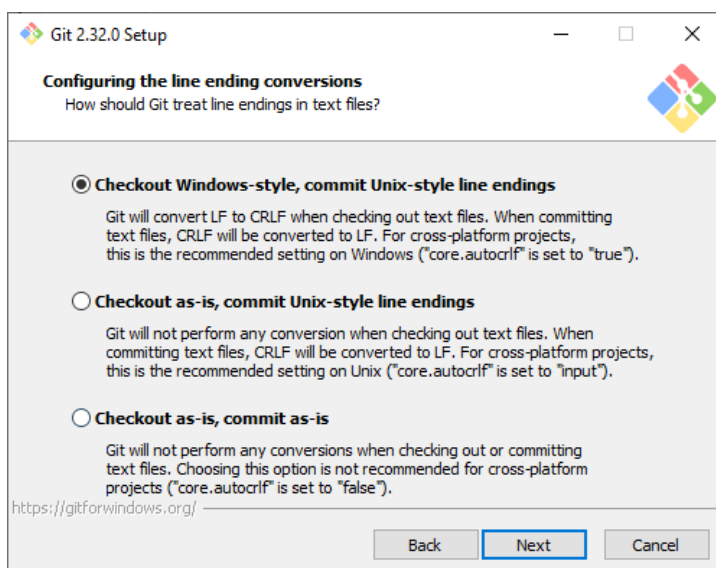
8. Configurar desde donde podrán ser accesibles los comandos de git.



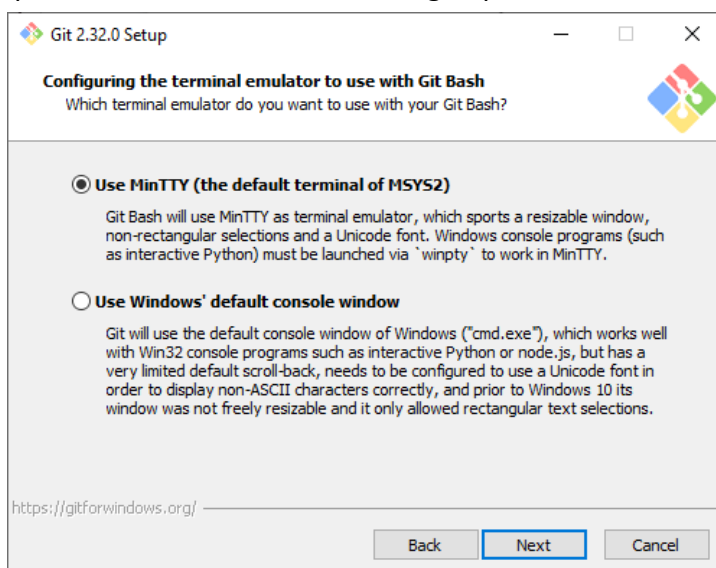
9. Seleccionar qué librería de seguridad se utilizará para las comunicaciones.



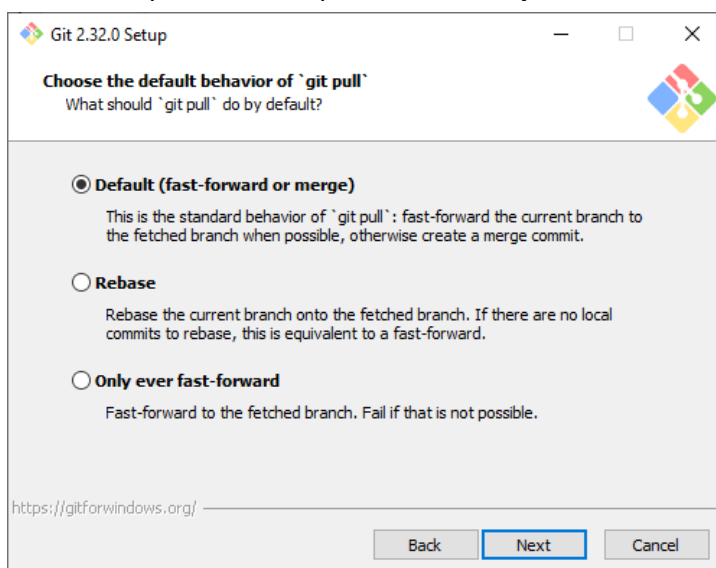
10. Configurar los finales de línea en los archivos de texto.



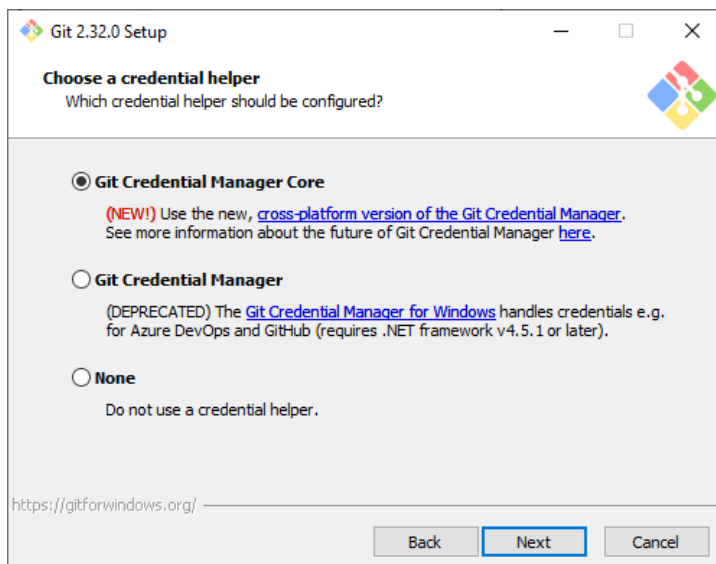
11. Configurar qué emulador de terminal se elegirá para usar con Git Bash.



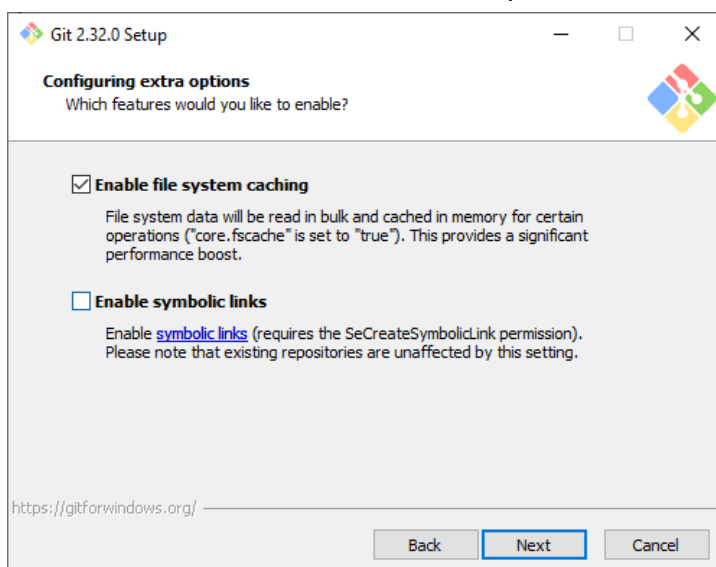
12. Elegir cuál será el comportamiento por defecto al ejecutar el comando "git pull".



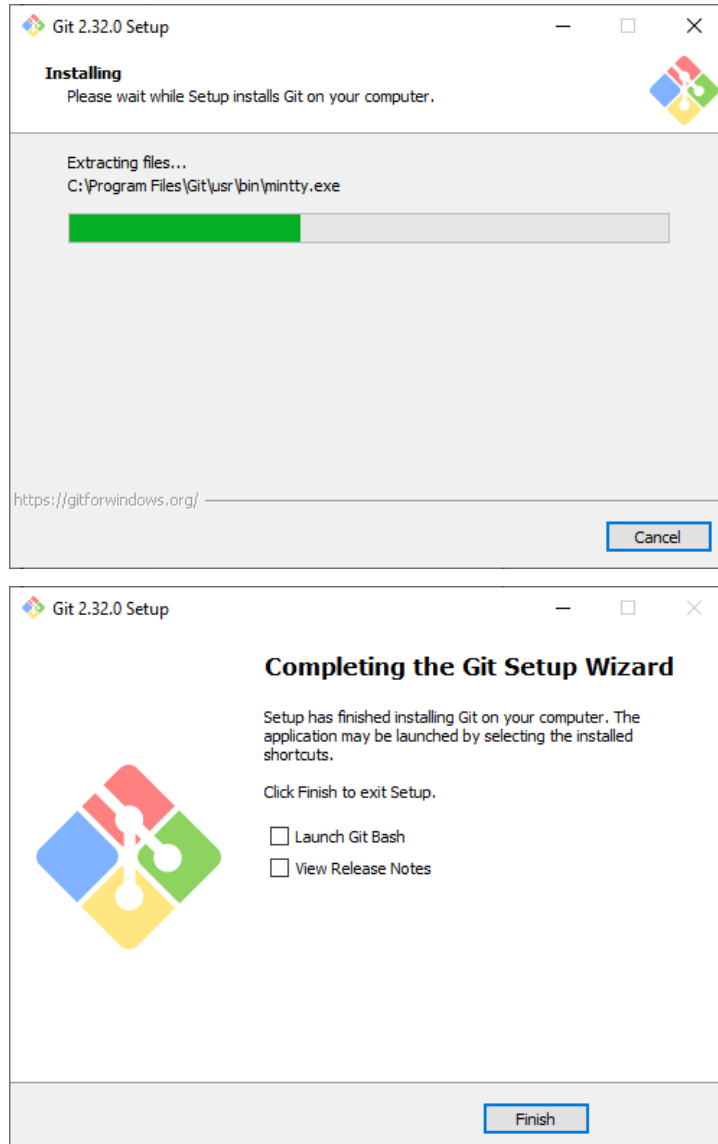
13. Configurar el gestor de credenciales que se utilizará para mantener información de usuarios asociados.



14. Configurar si se desea activar el sistema de caché y/o enlaces simbólicos.



15. Esperar a que finalice la instalación



Áreas y Estados

Para trabajar con git es fundamental entender los estados por los que pueden pasar los archivos durante todo el flujo de desarrollo.

En un proyecto de Git hay 4 secciones fundamentales:

- Directorio de trabajo (*Working Directory*): es una copia de una versión del proyecto, Son archivos sacados de la base de datos comprimida y se colocan en el disco para poder ser usados o modificados.

- Área de preparación (*Staging Area*): Es un archivo que se encuentra dentro del directorio de Git y que contiene información acerca de lo que va a ir en la próxima confirmación.
- Directorio de Git (*Local Repository*): Es el lugar donde se almacenan los metadatos y la base de datos de objetos del proyecto. Es lo que se copia cuando se clona un repositorio desde otra fuente.
- Repositorio Remoto (*Remote Repository*): Es el repositorio que se encuentra en un servidor remoto y con el que eventualmente se sincronizan los trabajos entre los diferentes integrantes del equipo.

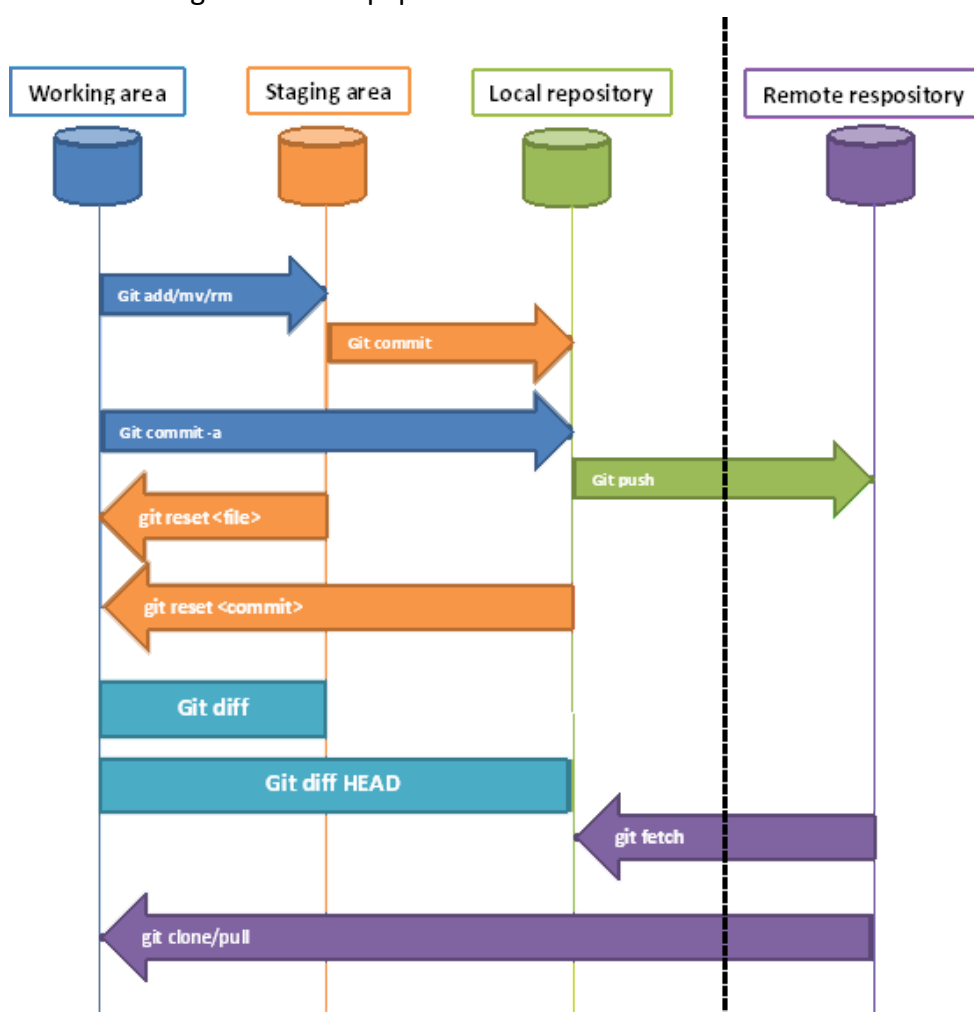


Diagrama del flujo de trabajo en Git. Adaptado de:

https://www.reddit.com/r/git/comments/99ul9f/git_workflow_diagram_showcasing_the_role_of

Comandos Básicos

Git se puede ver como un set de herramientas muy completo, pero para un manejo básico de repositorios en Git es necesario conocer, por lo menos, los siguientes comandos:

- **git init** es el comando para inicializar un directorio como repositorio Git, se ejecuta dentro del directorio del proyecto y, como resultado, crea un subdirectorio **.git** que contiene todos los archivos para poder realizar el seguimiento de los cambios, etiquetas, etc.
- **git add <file>** luego de la creación, modificación o eliminación de un archivo, los cambios quedan únicamente en el área de trabajo, por lo tanto es necesario pasarlos al área de preparación mediante el uso del comando **git add**, para que sea incluido dentro de la siguiente confirmación (*commit*).
- **git status** es un comando que permite conocer en qué estado se encuentran los archivos
- **git commit**, con este comando se confirman todos los cambios registrados en el área de preparación, o lo que es lo mismo, se pasan los cambios al repositorio local.
- **git push** es el comando que se utiliza para enviar todas las confirmaciones registradas en el repositorio local a un repositorio remoto.
- **git pull** funciona al inverso de **git push**, trayendo todos los cambios al repositorio local, pero también dejándolos disponibles directamente para su modificación o revisión en el área de trabajo. Es importante mencionar que se utiliza cuando ya se tiene un repositorio local vinculado a uno remoto, al igual que con el comando **git push**.
- **git clone**, en el caso de necesitar “bajar” un repositorio remoto de algún proyecto ya existente se puede ejecutar este comando. Genera un directorio (con el nombre del repositorio o uno especificado explícitamente) que contiene todo lo propio al proyecto, además del subdirectorio **.git** necesario para poder gestionar los cambios y todo lo pertinente al repositorio Git.

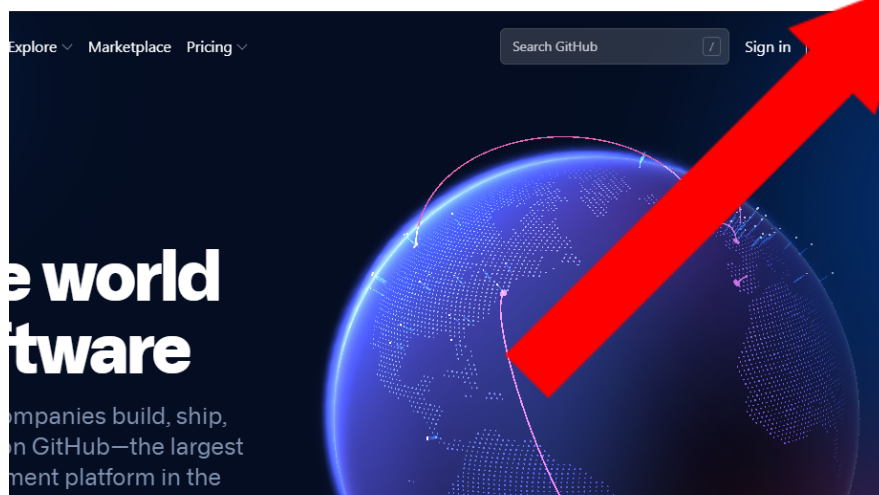
Github

GitHub es una plataforma de colaboración formal e informal de desarrollo de software (conocida también como plataforma de *social coding*). En esta plataforma se pueden publicar repositorios remotos que funcionan bajo el sistema de control de versiones Git. La plataforma configura los proyectos nuevos como de código abierto, por lo que cualquier persona puede verlos, pero esto es configurable.

¿Cómo empezar?

Lo primero para comenzar en GitHub es **crear una cuenta**, para ello hay que:

- Acceder al siguiente [enlace](#).
- Hacer click en el botón *Sign up*.



- Completar el formulario.
- Validar la cuenta mediante correo electrónico.

Proyectos

GitHub permite a los equipos de desarrollo coordinar, trackear y actualizar las tareas de un proyecto desde cualquier lugar, a fin de mantener los proyectos transparentes no sólo para con los integrantes del equipo de desarrollo sino también clientes, usuarios, etc. Todo desde un mismo lugar y en una misma plataforma.

Los proyectos en GitHub permiten organizar incidencias y notas en categorías mediante tarjetas en columnas. Estas tarjetas se pueden arrastrar entre las columnas según los estados en los que se encuentren las tareas que representan.

Los paneles de proyectos son personalizables y, por tanto, adaptables a las necesidades de cada proyecto. Dentro de los paneles se pueden reordenar columnas y cartas según los criterios que mejor se adapten al proyecto o la organización.

Dentro de las columnas o incluso de las tarjetas se pueden crear notas o comentarios que ayuden a entender las incidencias asociadas o que aporten información relevante al proyecto.

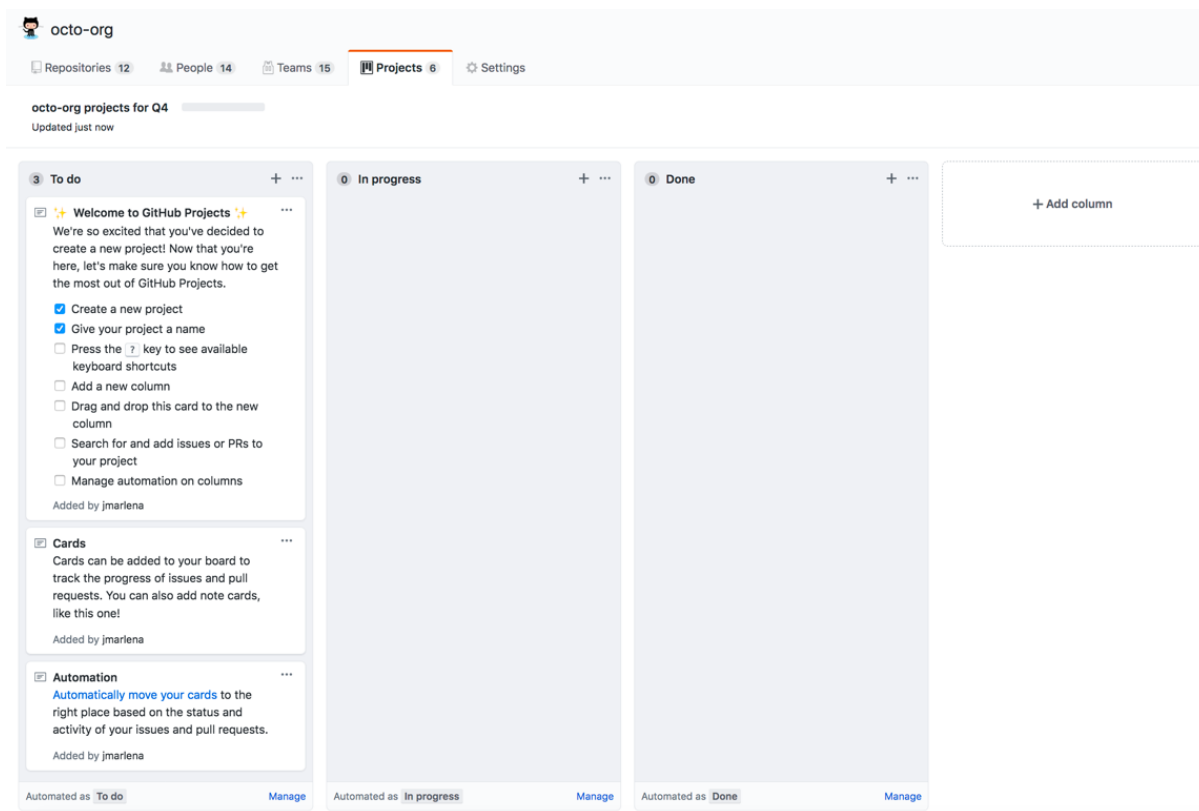
En GitHub existen tres tipos de tableros de proyectos:

- Pertenecientes al usuario: pueden tener incidencias de cualquier repositorio personal.
- De proyectos a nivel de organización: pueden contener incidencias de cualquier repositorio que pertenece a una organización.
- Tableros por repositorio: están enfocados en las incidencias de un único repositorio. Pueden incluir referencias o notas a incidencias de otros repositorios.

Se pueden vincular hasta 25 repositorios a cada tablero de proyecto. Vincular repositorios a proyectos facilita agregar informes de problemas al tablero a través del botón + o desde la barra lateral de la pestaña *Issues*.

¿Cómo crear un Tablero de Proyecto?

Para crear un tablero de proyecto, hacer clic en el siguiente enlace: [Instructivo Crear Tablero de Proyecto](#)



Tablero Kanban, recuperado de:

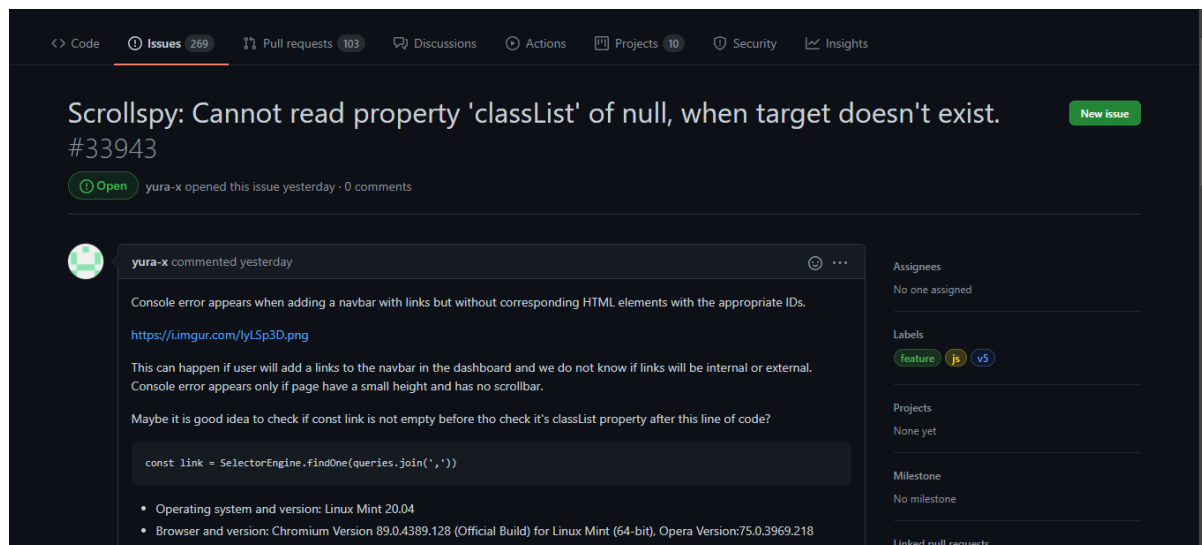
<https://docs.github.com/es/issues/organizing-your-work-with-project-boards/managing-project-boards/about-project-boards>



Investiga, ¿Qué es un tablero KANBAN? ¿Para qué sirve?

Incidencias

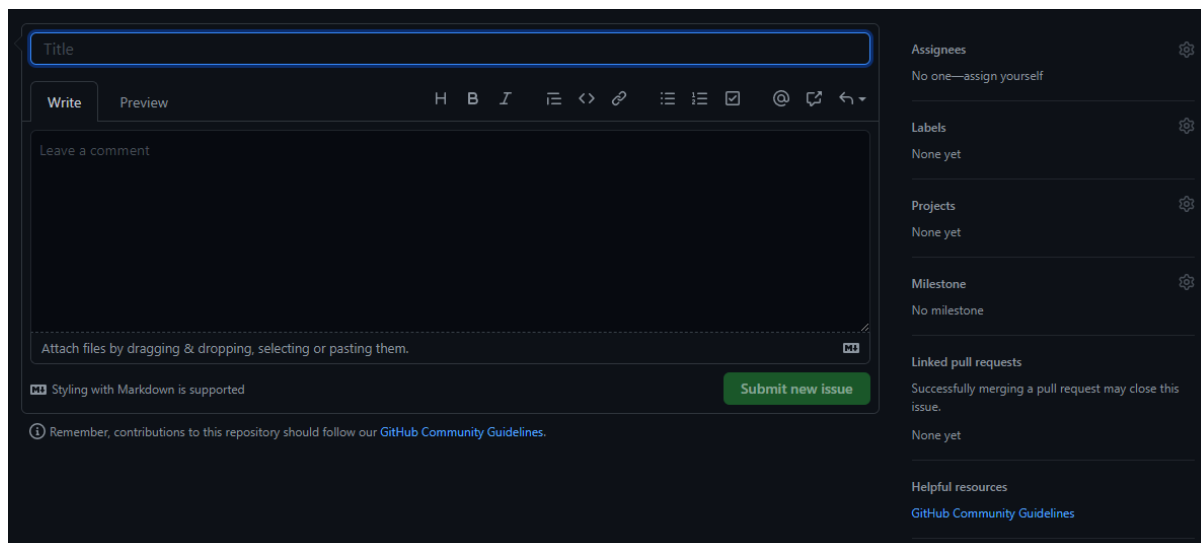
Una incidencia o asunto (*issue*) es una unidad de trabajo designada para realizar alguna mejora en un sistema informático. Puede ser el arreglo de un fallo, una característica pedida, una tarea, una solicitud de documentación en específico y todo tipo de ideas o sugerencias al equipo de desarrollo.



Ejemplo de *Issue*. Recuperado de: <https://github.com/twbs/bootstrap/issues/33943>

¿Cómo crear una Incidencia?

Para abrir una nueva incidencia hay que hacer click en el botón **New issue** y luego completar un formulario como el de la imagen a continuación:



Obligatoriamente una incidencia debe tener un título y una descripción, es muy importante adjuntar toda la información que pueda ser relevante para atender o resolver esta incidencia. La información adjuntada puede ser un documento, una imagen, vídeo, audio, etc.

Además, si la persona que crea la incidencia es miembro del equipo con permisos sobre el repositorio, puede opcionalmente asignar etiquetas, hitos, proyecto al que pertenece o responsables encargados de atender la incidencia.

Metadatos

Assignees (opcional)

Permite asignar 1 hasta 10 personas a la tarea.

Milestone (opcional)

Son categorías que se utilizan para tener un filtro más adecuado de la información. Cada milestone puede tener una fecha programada indicando el tiempo que es necesario para cumplir cierta tarea.

Label (opcional)

Especifica el tipo de issue. Pudiendo ser: bug, documentation, duplicate, invalid, etc. Es posible crear nuevas etiquetas.

"Etiquetar los issues permite buscarlos rápidamente más tarde".

Project (opcional)

Permite configurar el proyecto al que pertenece la incidencia.



Ahora ya sabemos qué es un incidente (issue). Sin embargo, puede que existan recomendaciones al momento de escribir un issue ¿no?. ¿Habrá mejores formas de escribir un issue? ¿Cuáles podrían ser esas recomendaciones?

Para más información, consulta el siguiente enlace: <https://docs.github.com/en/issues>

Hitos

Los hitos (*Milestones*) son grupos de incidencias y que ayudan a seguir el progreso de estas. Desde la página de detalle de un hito se pueden observar los siguientes datos:

- Una descripción del hito proporcionada por el usuario, que puede incluir información como una descripción general del proyecto, equipos relevantes y fechas de vencimiento proyectadas.

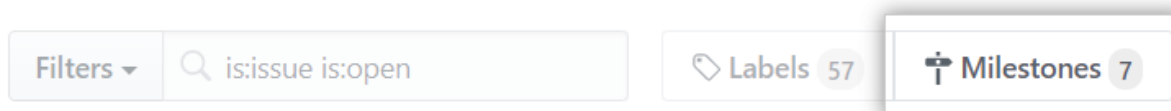
- La fecha de vencimiento del hito
- El porcentaje de finalización del hito
- La cantidad de incidencias abiertas y cerradas asociadas con el hito.
- Una lista de incidencias abiertas y cerradas asociadas con el hito.



Accede al repositorio de [Angular](#), analiza y observa los milestone y responde: En un proyecto de software: ¿Qué entiendes por hito (milestone)? ¿Cuándo deberíamos crear uno?

Para crear un hito se tienen que seguir los pasos a continuación:

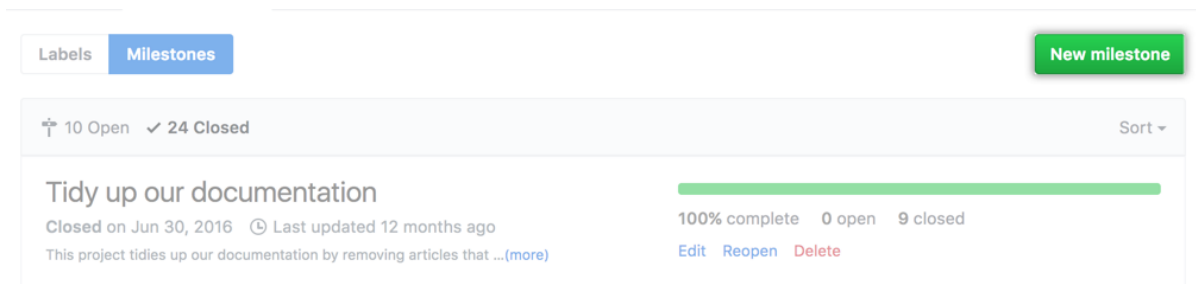
1. Desde la pestaña de incidencias (*Issues*), hacer click en el botón **Milestones**.



Recuperado de:

<https://docs.github.com/es/issues/using-labels-and-milestones-to-track-work/creating-and-editing-milestones-for-issues-and-pull-requests>

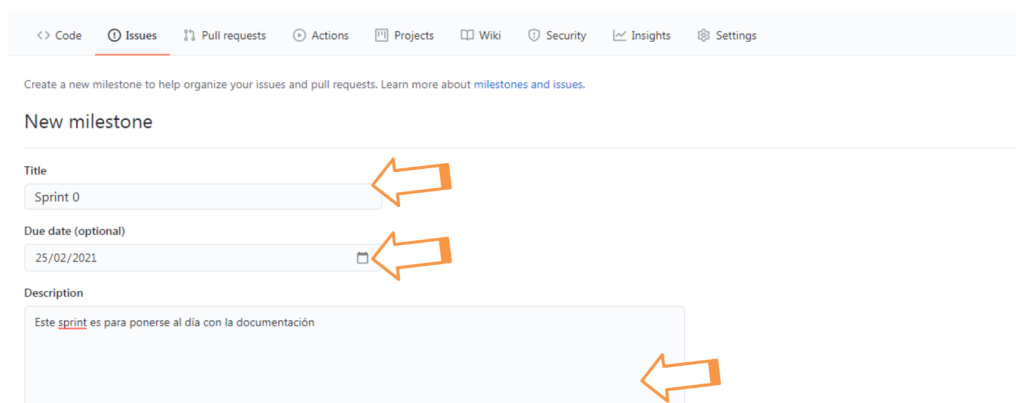
2. Hacer click en el botón **New milestone**.



Recuperado de:

<https://docs.github.com/es/issues/using-labels-and-milestones-to-track-work/creating-and-editing-milestones-for-issues-and-pull-requests>

3. Completar los campos de título, descripción, fecha de vencimiento, etc.



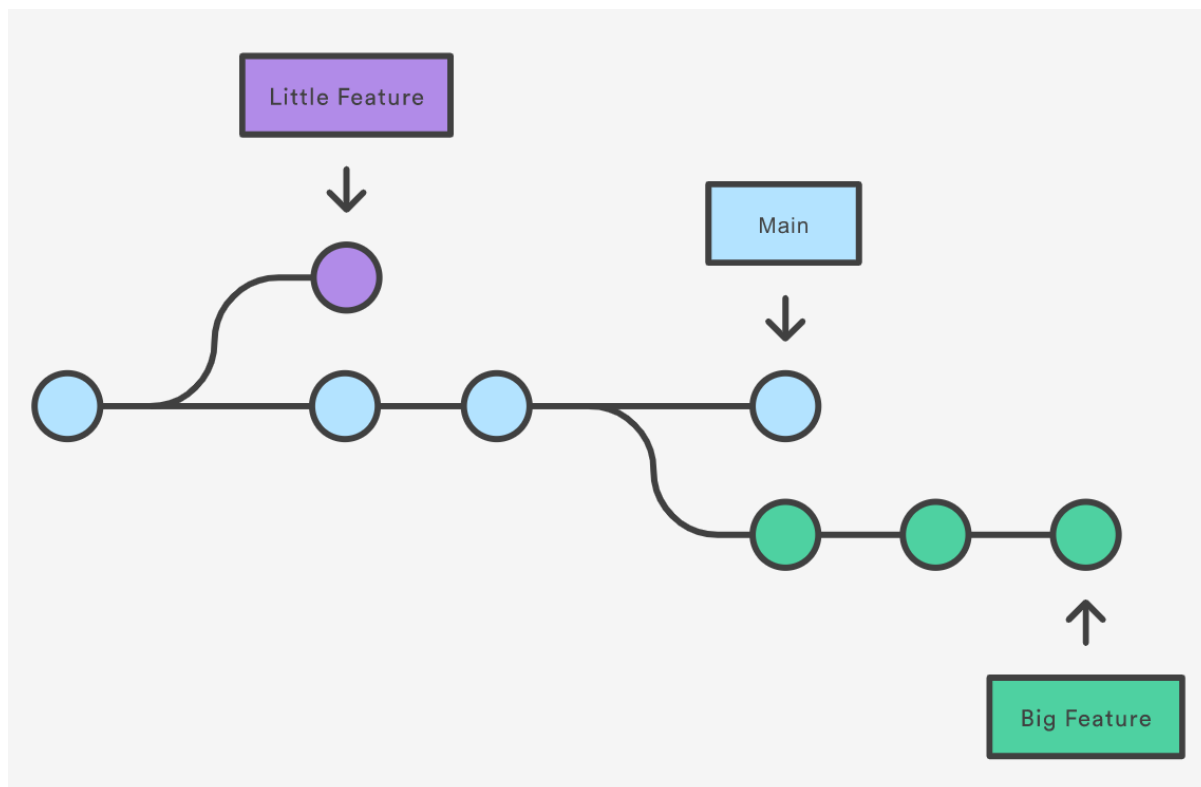
4. Hacer click en **Create milestone** para guardar los cambios.

Flujo de Trabajo

Ramas

La siguiente sección está extraída del material, con imágenes inclusive, disponible en:
<https://www.atlassian.com/es/git/tutorials/using-branches>

La creación de ramas es una función disponible en la mayoría de los sistemas de control de versiones modernos. La creación de ramas en otros sistemas de control de versiones puede tanto llevar mucho tiempo como ocupar mucho espacio de almacenamiento. En Git, las ramas son parte del proceso de desarrollo diario. Las ramas de Git son un puntero eficaz para las instantáneas de tus cambios. Cuando quieres añadir una nueva función o solucionar un error, independientemente de su tamaño, generas una nueva rama para alojar estos cambios. Esto hace que resulte más complicado que el código inestable se fusione con el código base principal, y te da la oportunidad de limpiar tu historial futuro antes de fusionarlo con la rama principal. Con git, la gestión de ramas se hace a través del comando **git branch**.



Representación de un repositorio con dos líneas de desarrollo aisladas, una para una función pequeña y otra para una función más extensa.

La implementación que subyace a las ramas de Git es mucho más sencilla que la de otros modelos de sistemas de control de versiones. En lugar de copiar archivos entre directorios, Git almacena una rama como referencia a una confirmación. En este sentido, una rama representa el extremo de una serie de confirmaciones, es decir, no es un contenedor de confirmaciones. El historial de una rama se extrapola de las relaciones de confirmación. Durante la lectura, recuerda que las ramas de Git no son como las ramas de SVN. Las ramas de SVN solo se usan para capturar el esfuerzo de desarrollo a gran escala ocasional, mientras que las ramas de Git son una parte integral del flujo de trabajo diario. El siguiente contenido amplía la información sobre la arquitectura interna de creación de ramas de Git.

Funcionamiento

Una rama representa una línea independiente de desarrollo. Las ramas sirven como una abstracción de los procesos de cambio, preparación y confirmación. Pueden concebirse como una forma de solicitar un nuevo directorio de trabajo, un nuevo entorno de ensayo o un nuevo historial de proyecto. Las nuevas confirmaciones se registran en el historial de la rama actual, lo que crea una bifurcación en el historial del proyecto.

El comando **git branch** te permite crear, enumerar, cambiar el nombre y eliminar ramas. No te permite cambiar entre ramas o volver a unir un historial bifurcado. Por este motivo, **git branch** está estrechamente relacionado con los comandos **git checkout** y **git merge**.

Opciones comunes

El comando principal, sin parámetros, permite listar todas las ramas del repositorio

```
git branch
```

Para crear una rama nueva se usa el comando como se muestra a continuación:

```
git branch <branch>
```

Para eliminar una rama, es necesario agregar la opción **-d**

```
git branch -d <branch>
```

Si se desea cambiar el nombre de una rama, se puede utilizar la opción **-m**, es importante mencionar que cambia el nombre de la rama sobre la cual se está trabajando.

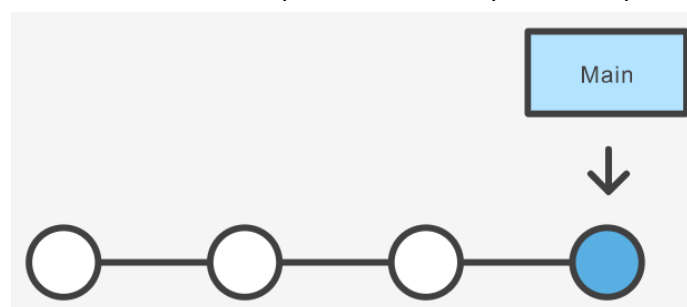
```
git branch -m <branch>
```

Finalmente, para listar todas las ramas en el repositorio remoto se puede utilizar el comando:

```
git branch -a
```

Creación de ramas

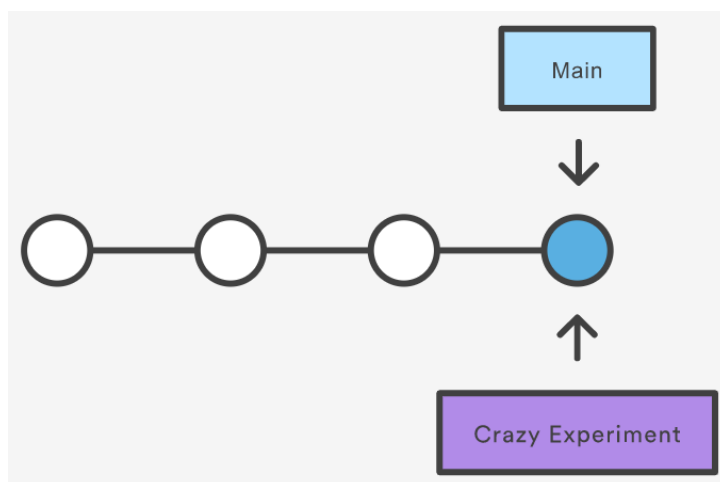
Es importante comprender que las ramas son solo punteros a las confirmaciones. Cuando creas una rama, todo lo que Git tiene que hacer es crear un nuevo puntero, no modifica el repositorio de ninguna otra forma. Si empiezas con un repositorio que tiene este aspecto:



Y, a continuación, creas una rama con el siguiente comando:

```
git branch crazy-experiment
```

El historial del repositorio no se modificará. Todo lo que necesitas es un nuevo puntero de la confirmación actual:



Ten en cuenta que este comando solo crea la nueva rama. Para empezar a añadir confirmaciones, necesitas seleccionarla con el comando **git checkout** y, a continuación, utilizar los comandos estándar **git add** y **git commit**.

Creación de ramas remotas

Por ahora, todos los ejemplos han ilustrado operaciones de ramas locales. El comando **git branch** también funciona con ramas remotas. Para trabajar en ramas remotas, primero hay que configurar un repositorio remoto y añadirlo a la configuración del repositorio local.

```
git remote add <remote-repo> <remote-repo-URL>

git push <remote-repo> crazy-experiment~
```

Este comando enviará una copia de la rama local crazy-experiment al repositorio remoto **<remote-repo>**

Eliminación de ramas

Una vez que hayas terminado de trabajar en una rama y la hayas fusionado con el código base principal, puedes eliminar la rama sin perder ninguna historia:

```
git branch -d crazy-experiment
```

No obstante, si la rama no se ha fusionado, el comando anterior mostrará un mensaje de error: *"error: The branch 'crazy-experiment' is not fully merged. If you are sure you want to delete it, run 'git branch -D crazy-experiment'."*

Esto te protege ante la pérdida de acceso a una línea de desarrollo completa. Si realmente quieres eliminar la rama (por ejemplo, si se trata de un experimento fallido), puedes usar el indicador -D (en mayúscula):

```
git branch -D crazy-experiment
```

Este comando elimina la rama independientemente de su estado y sin avisos previos, así que úsalo con cuidado.

Los comandos anteriores eliminarán una copia local de la rama. La rama seguirá existiendo en el repositorio remoto. Para eliminar una rama remota, ejecuta estos comandos.

```
git push origin --delete crazy-experiment
```

o bien con el comando:

```
git push origin :crazy-experiment
```

Enviarán una señal de eliminación al repositorio de origen remoto que desencadena la eliminación de la rama remota crazy-experiment.

Gitflow

Para esta sección se propone utilizar como material la información disponible en el siguiente enlace (disponible en español):

<https://www.atlassian.com/es/git/tutorials/comparing-workflows/gitflow-workflow>

El flujo de trabajo Gitflow, propuesto por Vincent Driessen en [nvie](#), define un modelo de creación de ramas estricto diseñado con la publicación del proyecto como fundamento. Proporciona un marco sólido para gestionar proyectos más grandes.

Gitflow es ideal para los proyectos que tienen un ciclo de publicación programado, así como para la práctica recomendada de DevOps de entrega continua. Este flujo de trabajo no añade ningún concepto o comando nuevo, aparte de los que se necesitan para el flujo de trabajo de ramas de función. Lo que hace es asignar funciones muy específicas a las distintas ramas y define cómo y cuándo deben estas interactuar. Además de las ramas feature, utiliza ramas individuales para preparar, mantener y registrar publicaciones. Por supuesto, también te aprovechas de todas las ventajas que aporta el flujo de trabajo de ramas de función: solicitudes de incorporación de cambios, experimentos aislados y una colaboración más eficaz.

Existe un conjunto de herramientas de línea de comandos para gestionar este tipo de flujo de trabajo. El proceso de instalación de git-flow es sencillo. Los paquetes de git-flow están disponibles en varios sistemas operativos. En los sistemas OSX, puedes ejecutar:

```
brew install git-flow
```

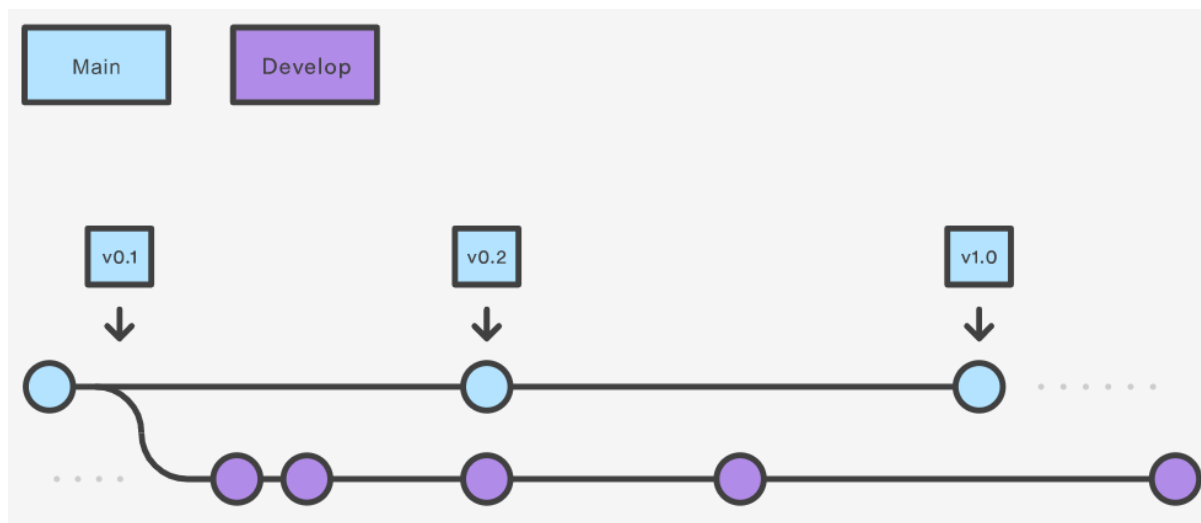
En Windows, tendrás que descargar e instalar git-flow. Después de instalar git-flow, puedes utilizarlo en tu proyecto ejecutando:

```
git flow init
```

Git-flow es un contenedor para Git. El comando **git flow init** es una prolongación del comando predeterminado **git init** y no cambia nada de tu repositorio aparte de crear ramas para ti.

Ramas en desarrollo y maestras

En vez de una única rama master, este flujo de trabajo utiliza dos ramas para registrar el historial del proyecto. La rama master almacena el historial de publicación oficial y la rama develop sirve como rama de integración para las funciones. Asimismo, conviene etiquetar todas las confirmaciones de la rama master con un número de versión.



El primer paso es complementar la master predeterminada con una rama develop. Una forma sencilla de hacerlo es que un desarrollador cree una rama develop vacía localmente y la envíe al servidor:

```
git branch develop
git push -u origin develop
```

Esta rama contendrá el historial completo del proyecto, mientras que la **master** contendrá una versión abreviada. A continuación, otros desarrolladores deberían clonar el repositorio central y crear una rama de seguimiento para **develop**.

A la hora de utilizar la biblioteca de extensiones de git-flow, ejecutar git flow init en un repositorio existente creará la rama develop:

```
git flow init
```

```

Initialized empty Git repository in ~/project/.git/
No branches exist yet. Base branches must be created now.
Branch name for production releases: [main]
Branch name for "next release" development: [develop]
How to name your supporting branch prefixes?
Feature branches? [feature/]
Release branches? [release/]
Hotfix branches? [hotfix/]
Support branches? [support/]
Version tag prefix? []

```

```
git branch
```

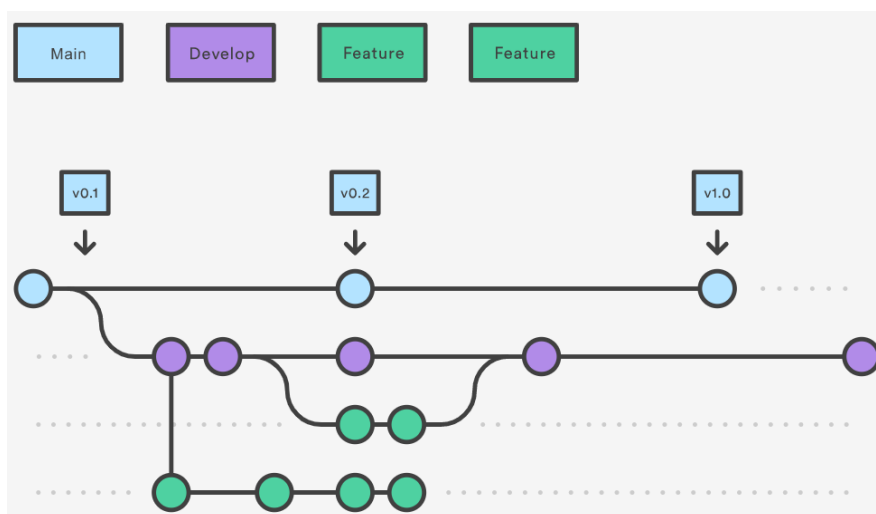
```

* develop
main

```

Ramas de función

Todas las funciones nuevas deben residir en su propia rama, que se pueden enviar al repositorio central para copia de seguridad/colaboración. Sin embargo, en vez de ramificarse de la **master**, las ramas **feature** utilizan la **develop** como rama primaria. Cuando una función está terminada, se vuelve a fusionar en la de desarrollo. Las funciones no deben interactuar nunca directamente con la master.



Ten en cuenta que las ramas **feature** combinadas con la rama **develop** conforman, a todos efectos, el flujo de trabajo de ramas de función. Sin embargo, el flujo de trabajo Gitflow no termina aquí.

Las ramas **feature** suelen crearse a partir de la última rama **develop**.

Para crear una rama de función sin las extensiones git-flow se pueden ejecutar los comandos:

```
git checkout develop
git checkout -b feature_branch
```

O bien utilizando la extensión, con el comando:

```
git flow feature start feature_branch
```

Luego de esto se continúa trabajando normalmente con Git.

Para finalizar una rama de función, cuando el desarrollo de esta haya culminado, hay que fusionar la rama con la rama **develop**. y para esto tenemos dos opciones, sin la extensión git-flow:

```
git checkout develop
git merge feature_branch
```

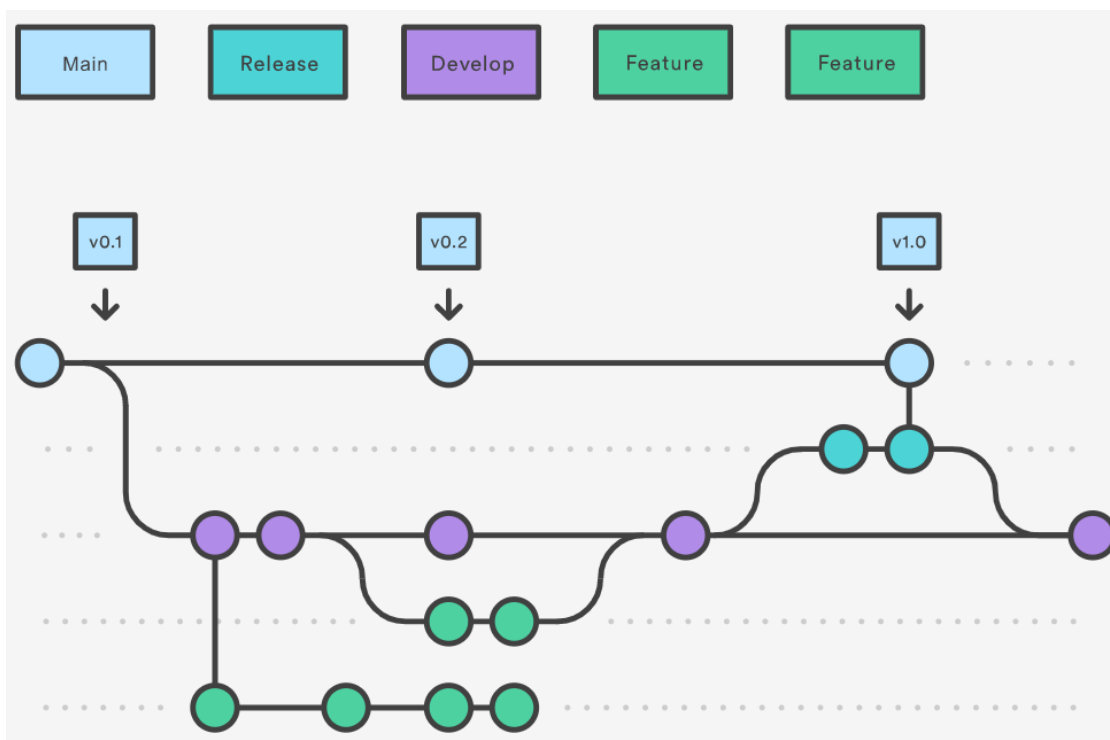
O bien con la extensión:

```
git flow feature finish feature_branch
```

Ramas de publicación

Cuando **develop** haya adquirido suficientes funciones para una publicación (o se acerque una fecha de publicación predeterminada), debes bifurcar una rama **release** a partir de una **develop**. Al crear esta rama, se inicia el siguiente ciclo de publicación, por lo que no pueden añadirse nuevas funciones una vez pasado este punto (en esta rama solo deben producirse las soluciones de errores, la generación de documentación y otras tareas orientadas a la publicación). Cuando está lista para el lanzamiento, la rama **release** se fusiona en la **master** y se etiqueta con un número de versión. Además, debería volver a fusionarse en **develop**, que podría haber progresado desde que se iniciara la publicación.

Utilizar una rama específica para preparar publicaciones hace posible que un equipo perfeccione la publicación actual mientras otro equipo sigue trabajando en las funciones para la siguiente publicación. Asimismo, crea fases de desarrollo bien definidas (por ejemplo, es fácil decir: "Esta semana nos estamos preparando para la versión 4.0" y verlo escrito en la estructura del repositorio).



Crear ramas **release** es otra operación de ramificación sencilla. Al igual que las ramas **feature**, las ramas **release** se basan en la rama **develop**. Se puede crear una nueva rama **release** utilizando los siguientes métodos. Sin las extensiones de git-flow:

```
git checkout develop
git checkout -b release/0.1.0
```

O bien con las extensiones:

```
git flow release start 0.1.0

Switched to a new branch 'release/0.1.0'
```

En cuanto la publicación esté lista para su lanzamiento, se fusionará en la **master** y la **develop**; y luego se eliminará la rama **release**. Es importante volver a fusionarla en **develop** porque podrían haberse añadido actualizaciones críticas a la rama **release**, y las funciones nuevas tienen que poder acceder a ellas. Si tu organización enfatiza la revisión de código, este sería el lugar ideal para una solicitud de incorporación de cambios.

Para finalizar una rama **release**, utiliza los siguientes métodos. Sin las extensiones de git-flow:

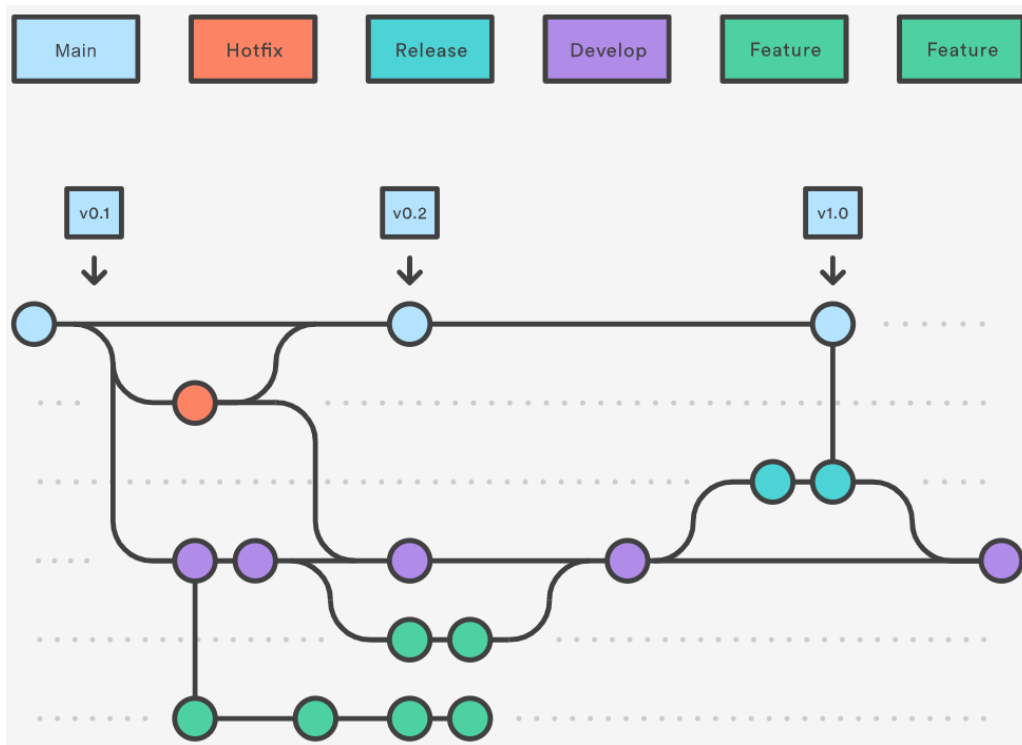
```
git checkout main
git merge release/0.1.0
```

O con la extensión:

```
git flow release finish '0.1.0'
```

Ramas de corrección

Las ramas de mantenimiento o de "corrección" (hotfix) sirven para reparar rápidamente las publicaciones de producción. Las ramas hotfix son muy similares a las ramas release y feature, salvo por el hecho de que se basan en la master, no en la develop. Esta es la única rama que debería bifurcarse directamente a partir de la master. Cuando se haya terminado de aplicar la corrección, debería fusionarse en la master y la develop (o la rama release actual), y la master debería etiquetarse con un número de versión actualizado.



Tener una línea de desarrollo específica para la corrección de errores permite que tu equipo aborde las incidencias sin interrumpir el resto del flujo de trabajo ni esperar al siguiente ciclo de publicación. Puedes concebir las ramas de mantenimiento como ramas release ad hoc que trabajan directamente con la master. Se puede crear una nueva rama hotfix utilizando los siguientes métodos. Sin las extensiones de git-flow:

```
git checkout main
git checkout -b hotfix_branch
```

O bien utilizando las extensiones:

```
git flow hotfix start hotfix_branch
```

Al igual que al finalizar una rama **release**, una rama **hotfix** se fusiona tanto en la **master** como en la **develop**.

Sin las extensiones:

```
git checkout main
git merge hotfix_branch
```



```
git checkout develop  
  
git merge hotfix_branch  
  
git branch -D hotfix_branch
```

O con las extensiones:

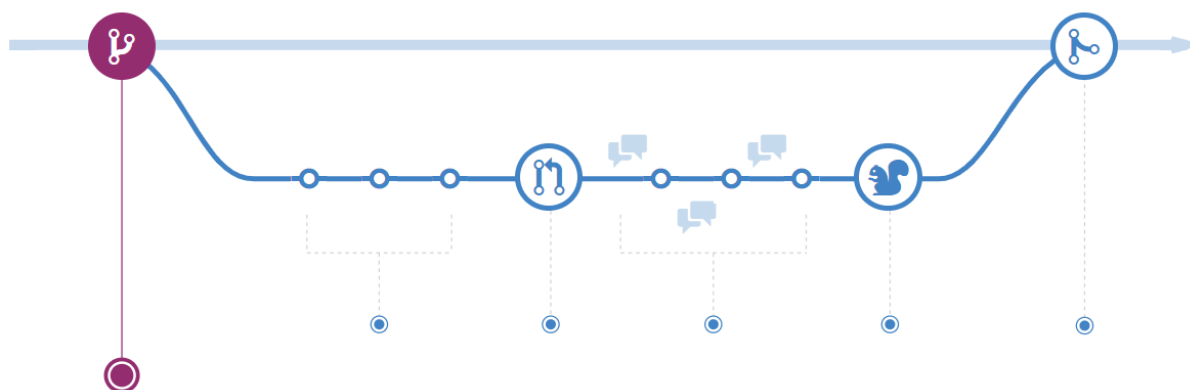
```
git flow hotfix finish hotfix_branch
```

GitHub Flow

El siguiente material es una traducción del disponible en <https://guides.github.com/introduction/flow/>

El flujo de GitHub es un flujo de trabajo ligero basado en ramas que admite equipos y proyectos en los que se realizan despliegues con regularidad. Esta guía explica cómo y por qué funciona el flujo de GitHub.

1. Crear una rama



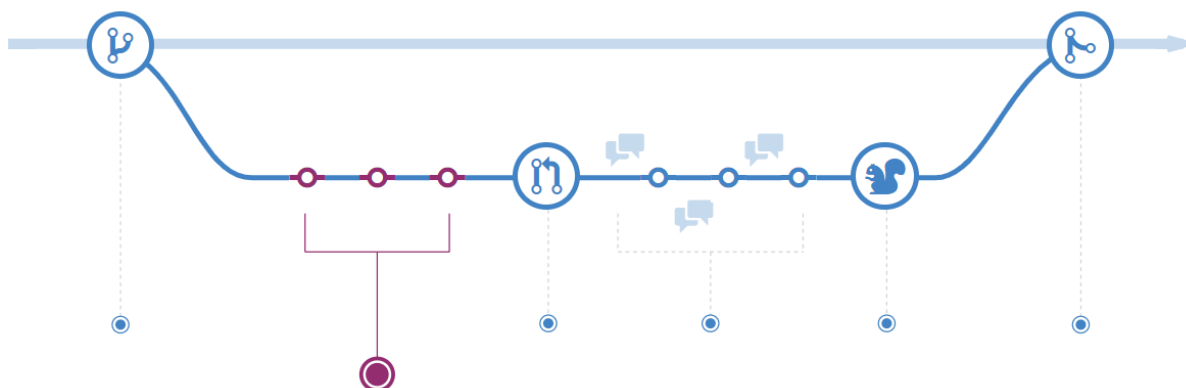
Cuando esté trabajando en un proyecto, tendrá un montón de características o ideas diferentes en progreso en un momento dado, algunas de las cuales están listas para funcionar y otras no. La ramificación existe para ayudar a administrar este flujo de trabajo.

Cuando crea una rama en su proyecto, está creando un entorno en el que puede probar nuevas ideas. Los cambios que realiza en una rama no afectan a la rama principal, por lo que puede experimentar y realizar cambios, con la seguridad de saber que su rama no se fusionará hasta que esté lista para ser revisada por alguien con quien colabora.

La ramificación es un concepto central en Git, y todo el flujo de GitHub se basa en él. Solo hay una regla: cualquier cosa en la rama principal siempre se puede desplegar.

Debido a esto, es extremadamente importante que su nueva rama se cree a partir de main cuando se trabaja en una función o una solución. El nombre de su rama debe ser descriptivo (por ejemplo, **refactor-authentication**, **user-content-cache-key**, **make-retina-avatars**), para que otros puedan ver en qué se está trabajando.

2. Agregar confirmaciones



Una vez que se haya creado su rama, es hora de comenzar a hacer cambios. Siempre que agrega, edita o elimina un archivo, está realizando una confirmación (*commit*) y agregándola a su rama. Este proceso de agregar confirmaciones realiza un seguimiento de su progreso a medida que trabaja en una rama de funciones.

Las confirmaciones también crean un historial transparente de su trabajo que otros pueden seguir para comprender lo que ha hecho y por qué. Cada confirmación tiene un mensaje asociado, que es una descripción que explica por qué se realizó un cambio en particular. Además, cada confirmación se considera una unidad de cambio separada. Esto le permite revertir los cambios si se encuentra un error o si decide ir en una dirección diferente.

Los mensajes de confirmación son importantes, especialmente porque Git rastrea sus cambios y luego los muestra como confirmaciones una vez que se envían al servidor. Al escribir mensajes de compromiso claros, puede facilitar que otras personas lo sigan y brinden comentarios.

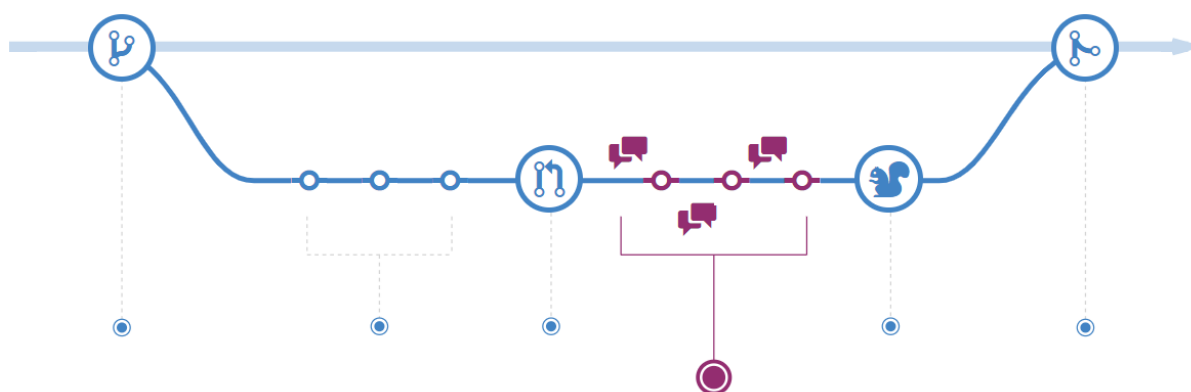
3. Abrir una solicitud de incorporación



Las solicitudes de incorporación (*pull requests*) inician la discusión sobre sus confirmaciones. Debido a que están estrechamente integrados con el repositorio de Git subyacente, cualquiera puede ver exactamente qué cambios se fusionarán si aceptan su solicitud.

Puede abrir una solicitud de incorporación en cualquier momento durante el proceso de desarrollo: cuando tiene poco o ningún código pero desea compartir algunas capturas de pantalla o ideas generales, cuando está atascado y necesita ayuda o consejo, o cuando está listo para alguien para revisar su trabajo. Al usar el sistema @menciones de GitHub en su mensaje de *Pull Request*, puede solicitar comentarios de personas o equipos específicos, ya sea que estén al final del pasillo o diez zonas horarias de distancia.

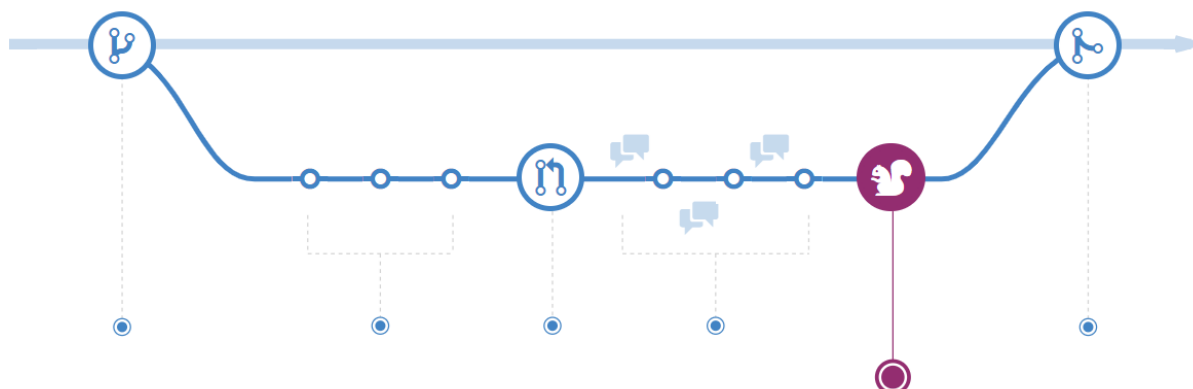
4. Discutir y revisar el código



Una vez que se ha abierto una solicitud de incorporación, la persona o el equipo que revisa sus cambios puede tener preguntas o comentarios. Quizás el estilo de codificación no coincide con las pautas del proyecto, al cambio le faltan pruebas unitarias o tal vez todo se ve bien y los accesorios están en orden. Las solicitudes de incorporación están diseñadas para fomentar y capturar este tipo de discusiones.

También puede continuar fusionando su rama luego de la discusión y los comentarios sobre sus confirmaciones. Si alguien comenta que se olvidó de hacer algo o si hay un error en el código, puede solucionarlo en su rama y fusionar el cambio. GitHub mostrará sus nuevas confirmaciones y cualquier comentario adicional que pueda recibir en la vista *Pull Request* unificada.

5. Desplegar

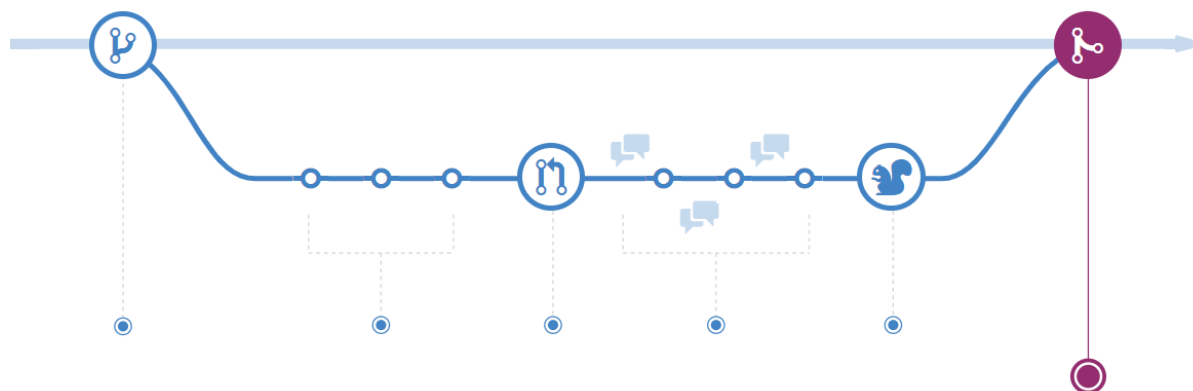


Con GitHub, se puede desplegar desde una rama para la prueba final en producción antes de fusionarse con **main**.

Una vez que se haya revisado su solicitud de incorporación y la rama pase las pruebas, puede desplegar sus cambios para verificarlos en producción. Si su rama causa problemas, puede revertirla implementando la rama principal existente en producción.

Los diferentes equipos pueden tener diferentes estrategias de despliegue. Para algunos, puede ser mejor desplegar en un entorno de prueba especialmente provisto. Para otros, el despliegue directamente en producción puede ser la mejor opción en función de los otros elementos de su flujo de trabajo.

6. Fusionar



Figuras: Flujo de trabajo propuesto por GitHub, Recuperado de: <https://guides.github.com/introduction/flow/>

Ahora que sus cambios se han verificado en producción, es hora de fusionar su código en la rama **main**.

Una vez fusionadas, las solicitudes de incorporación conservan un registro de los cambios históricos en su código. Debido a que se pueden buscar, permiten que cualquiera retroceda en el tiempo para comprender por qué y cómo se tomó una decisión.

Referencias

1. Atlassian (2021). ¿Qué es el control de versiones?. Recuperado de:
<https://www.atlassian.com/es/git/tutorials/what-is-version-control>
2. Atlassian (2021). ¿Qué es Git? Recuperado de:
<https://www.atlassian.com/es/git/tutorials/what-is-git>
3. Git SCM (2021). Inicio - Sobre el Control de Versiones - Instalación de Git. Recuperado de:
<https://git-scm.com/book/es/v2/Inicio---Sobre-el-Control-de-Versiones-Instalaci%C3%B3n-de-Git>
4. Git SCM (2021). Inicio - Sobre el Control de Versiones - Fundamentos de Git. Recuperado de:
<https://git-scm.com/book/es/v2/Inicio---Sobre-el-Control-de-Versiones-Fundamentos-de-Git>
5. Free Code Camp (2021). 10 Comandos de Git que todo Desarrollador debería Saber. Recuperado de:
<https://www.freecodecamp.org/espanol/news/10-comandos-de-git-que-todo-desarrollador-deberia-saber/>
6. GitHub (2021). Crear y editar hitos para incidencias. Recuperado de:
<https://docs.github.com/es/issues/using-labels-and-milestones-to-track-work/creating-and-editing-milestones-for-issues-and-pull-requests>
7. GitHub (2021). Sobre paneles de proyectos. Recuperado de:
<https://docs.github.com/es/issues/organizing-your-work-with-project-boards/managing-project-boards/about-project-boards>
8. Atlassian (2021). Ramas de Git. Recuperado de:
<https://www.atlassian.com/es/git/tutorials/using-branches>
9. Atlassian (2021). Flujo de trabajo Gitflow. Recuperado de:
<https://www.atlassian.com/es/git/tutorials/comparing-workflows/gitflow-workflow>
10. GitHub (2021). Comprendiendo el flujo de trabajo de GitHub. Recuperado de:
<https://guides.github.com/introduction/flow/>