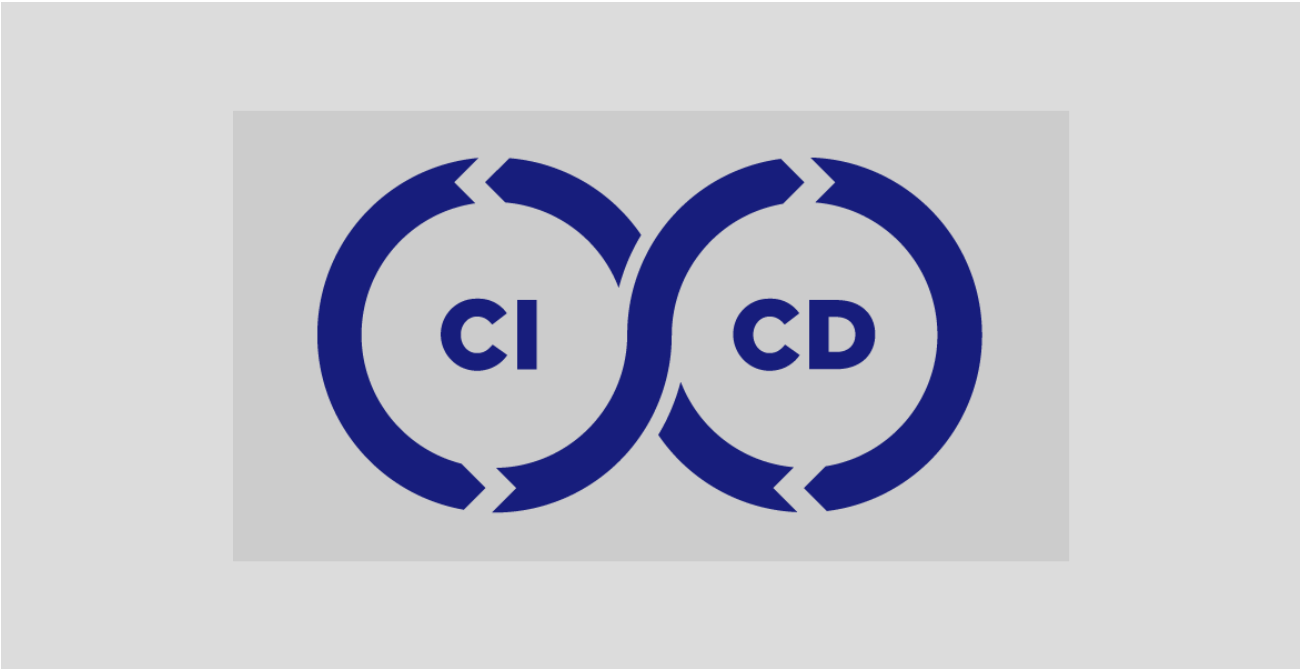
1 Integración continua y entrega continua

2 Control de versiones

Integración continua y Entrega continua (CI/CD)



En este tema presentamos una introducción a las prácticas de Entrega Continua (Continuous Delivery, abreviado CD) y de Integración Continua (Continuous Integration - abreviado CI), elementos importantes en el repertorio de prácticas de desarrollo de software moderno y DevOps.

¿Cuándo debo considerar CI/CD?

Cada vez son más las organizaciones que se benefician al adoptar la integración continua y la entrega continua, conocidas como CI/CD, como un elemento clave para darle agilidad, velocidad y mayor calidad al software que transforma las empresas.

> ***Adoptamos CI/CD para entregar software más rápido, más frecuentemente y con menos errores.***

Algunas de las principales ventajas de adoptar las prácticas de CI/CD incluyen las siguientes:

- Entregar o liberar código de forma inmediata, sin esperar a acumular muchas funcionalidades, tickets o desarrollos independientes.
- Menor esfuerzo y más confianza al momento de desplegar código a la etapa de producción.

- Se facilita el trabajo en equipo y la comunicación entre los desarrolladores de software.
- Mayor visibilidad del código que se está desarrollando.
- Se pueden identificar errores antes de publicar el código.

En algunos casos es posible que los miembros más antiguos del equipo de trabajo se resistan a la adopción de CI/CD en el flujo de trabajo, pues la necesidad de reorganizar algunos roles, de utilizar nuevas herramientas y de poner al descubierto debilidades técnicas llevan a que se perciba el cambio como una amenaza.

Por esta razón es fundamental definir desde el inicio del proyecto la forma en que la metodología CI/CD puede beneficiar al equipo, teniendo claridad respecto a cuales serán las acciones, responsabilidades y funciones de cada un de los integrantes del mismo.

A través de la metodología CI/CD el equipo tiene completo control del proceso de desarrollo y producción, probando e integrando cambios para verificar que todo funcione correctamente. De esta forma es posible identificar fácilmente cualquier error que impida su funcionamiento correcto.

¿Cuánto esfuerzo requiere desplegar código?

El nivel de esfuerzo requiere integrar y desplegar código pone en evidencia el nivel de madurez de los procesos técnicos de integración continua y entrega continua (CI/CD).

Existen muchas limitaciones en los procesos existentes. Hacer una sola integración de código al final del proceso de desarrollo, acumular código para desplegar cada semana o cada quince días o en general una tasa alta de fallas en los despliegues son un indicador importante de una oportunidad de mejora.

La Integración Continua (CI) es una práctica que incrementa la eficacia y la eficiencia de los resultados del equipo de desarrolladores de software. Se trata de combinar, de forma periódica y en un repositorio central, los cambios realizados en el código de un proyecto, para luego ejecutar pruebas, detectar y reparar errores lo antes posible.

Esta metodología permite mejorar la calidad del código, entregar avances al cliente más frecuentemente y trabajar ágilmente con nuevos participantes del equipo de desarrollo, incluso si estos no conocen en profundidad el proyecto completo.

> *“Detectar problemas de código es mucho más fácil con CI, evitando que estos errores lleguen a los clientes”.*

Brecha de conocimiento

Muchas organizaciones que buscan implementar estas metodologías fracasan en el proceso por no tener personas dentro del equipo con los conocimientos necesarios para el diseño de la solución CI/CD adecuada.

Integración Continua (Continuous Integration CI)

Con la integración continua los desarrolladores integran frecuentemente su código a la rama principal de un repositorio común. En lugar de desarrollar partes del código de manera aislada e integrarlas al final del ciclo de producción, un desarrollador contribuirá con cambios a una parte del repositorio varias veces al día.

La idea aquí es reducir los costos de integración, haciendo que los desarrolladores realicen integraciones más rápidamente y con mayor frecuencia. En la práctica, un desarrollador a menudo descubrirá conflictos entre el código nuevo y el existente en el momento de la integración. Si esta se realiza temprano y con frecuencia, la expectativa es que la resolución de conflictos será más fácil y menos costosa.

Entrega continua (Continuous Delivery CD)

La entrega continua es en realidad una extensión de la Integración Continua, en la cual el proceso de entrega de software se automatiza para permitir implementaciones fáciles y confiables en la producción, en cualquier momento.

Un proceso de Entrega Continua (CD) bien implementado requiere una base de código que se pueda desplegar en cualquier momento. Con la Entrega Continua, **los lanzamientos de nuevos cambios ocurren de manera frecuente y rutinaria**. Los equipos continúan con las tareas diarias de desarrollo con la confianza de que pueden mandar a producción un lanzamiento de calidad, en cualquier momento que deseen, sin una complicada implementación y sin tener que seguir pasos de manuales ejecutados por un especialista.

Se considera que la entrega continua es atractiva principalmente porque automatiza todos los pasos que van desde la integración del código en el repositorio base, hasta la liberación de cambios totalmente probados y funcionalmente adecuados.

El proceso consiste en una compleja automatización de los procesos de compilación (si es necesaria), pruebas, actualización de servidores de producción y ajuste de código usado en nuevas máquinas. En todo momento, se mantiene la autonomía del negocio de decidir qué modificaciones se publican y cuándo deben hacerse.

Despliegue Continuo (Continuous Deployment)

Esta es una modalidad más avanzada de Entrega Continua en la medida en que los despliegues a producción no pasan por una validación humana sino que están totalmente automatizadas.

La única forma de detener nuevos desarrollos en producción es a través de una prueba que falla e identifica los errores. La implementación es una manera de acelerar la retroalimentación, haciendo más eficiente el trabajo del equipo.

Gracias a este tipo de despliegue, los desarrolladores pueden llegar a ver su trabajo funcionando poco tiempo después de haberlo realizado, pero requiere el mayor esfuerzo en buenas prácticas y procesos automatizados.

¿Cuáles son los beneficios de cada práctica?

Cada una de las prácticas descritas requiere ciertas condiciones al tiempo que ofrece beneficios en su adopción. Exponemos estos a continuación:

Requisitos para Integración Continua

- El equipo debe crear pruebas automatizadas para detectar inconsistencias en los nuevos desarrollos.
- Necesitará un servidor de integración continuo que permita el monitoreo y la ejecución de las pruebas de forma automática para cada nueva confirmación que se logre.
- Los desarrolladores deben fusionar los cambios de forma rápida y continua, por lo menos una vez por día.

Beneficios de Integración Continua

- No existen tantos errores y se evita que estos se envíen a producción. Las pruebas automáticas captan las regresiones muy rápido.
- Desarrollar las versiones es más fácil. Los problemas que ocurren en la integración se solucionan mucho más rápido.
- Las pruebas ya no son un gran costo para el equipo, pues el CI permite realizar muchas pruebas en cuestión de segundos.
- El control de calidad es mucho mejor.

ENTREGA CONTINUA

Requisitos:

- Base sólida de Integración Continua. Su entorno de pruebas debe cubrir una gran cantidad de código base.
- Las implementaciones que se requieren deben ser automatizadas. Aunque se activa de forma manual, una vez inicia no necesita de presencia humana.

Beneficios:

- Mayor velocidad y menor complejidad en el proceso de desarrollo.
- Lanzamientos mucho más seguidos, proporcionando retroalimentación inmediata.

DESPLIEGUE CONTINUO

Requisitos:

- Mientras mejor sea la calidad de las pruebas, mejor será la calidad de cada uno de los lanzamientos.
- Se debe mantener actualizado el proceso de documentación, coordinando la comunicación con otros departamentos como “soporte” o “marketing”.

Beneficios:

- Todo se activa de forma automática para cada cambio.
- Cada nueva versión tiene menos riesgos, pues a medida de que ocurren cambios se identifican los problemas que surgen y se pueden solucionar más fácilmente.
- El flujo es continuo y de alta calidad. Este es el beneficio más importante para los clientes.

¿Qué necesito para adoptar CI/CD?

Implementar el proceso CI/CD podría ser un proyecto difícil y en ocasiones hasta desalentador. Más que un proyecto tecnológico es de cambio organizacional. Existen muchas herramientas que facilitan el desempeño del proceso e incluso existen complementos vía online que se pueden adquirir.

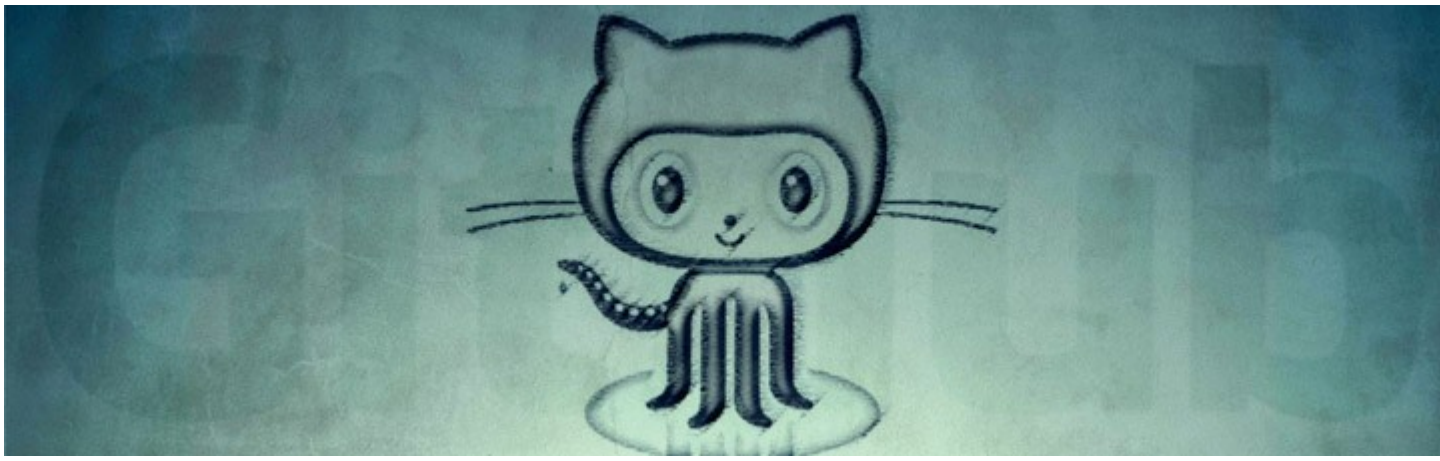
Control de versiones

Una de las herramientas imprescindibles que forman parte de la integración y entrega continua son los sistemas de control de versiones por tanto vamos a introducirnos en algunos de sus conceptos.

Introducción a los sistemas de control de versiones, su importancia y sus características a lo largo de la historia. Descripción general de lo que son Git y Github.

Los sistemas de control de versiones son programas que tienen como objetivo controlar los cambios en el desarrollo de cualquier tipo de software, permitiendo conocer el estado actual de un proyecto, los cambios que se le han realizado a cualquiera de sus piezas, las personas que intervinieron en ellos, etc.

Introducimos uno de los sistemas de control de versiones existentes en la actualidad que se ha popularizado tremendamente, gracias al sitio Github. Se trata de Git, el sistema de control de versiones más conocido y usado actualmente, que es el motor de Github. Al terminar su lectura entenderás qué es Git y qué es Github, dos cosas distintas que a veces resultan confusas de entender por las personas que están dando sus primeros pasos en el mundo del desarrollo.



Necesidad de un control de versiones

El control de versiones es una de las tareas fundamentales para la administración de un proyecto de desarrollo de software en general. Surge de la necesidad de mantener y llevar control del código que vamos programando, conservando sus distintos estados. Es absolutamente necesario para el trabajo en equipo, pero resulta útil incluso a desarrolladores independientes.

Aunque trabajemos solos, sabemos más o menos cómo surge la necesidad de gestionar los cambio entre distintas versiones de un mismo código. Prueba de ello es que todos los programadores, más tarde o más temprano, se han visto en la necesidad de tener dos o más copias de un mismo archivo, para no perder su estado anterior cuando vamos a introducir diversos cambios. Para ir solucionando nuestro día a día habremos copiado un fichero, agregándole la fecha o un sufijo como "antiguo". Aunque quizás esta acción nos sirva para salir del paso, no es lo más cómodo ni mucho menos lo más práctico.

En cuanto a equipos de trabajo se refiere, todavía se hace más necesario disponer de un control de versiones. Seguro que la mayoría hemos experimentado las limitaciones y problemas en el flujo de trabajo cuando no se dispone de una herramienta como Git: machacar los cambios en archivos hechos por otros componentes del equipo, incapacidad de comparar de manera rápida dos códigos, para saber los cambios que se introdujeron al pasar de uno a otro, etc.

Además, en todo proyecto surge la necesidad de trabajar en distintas ramas al mismo tiempo, introduciendo cambios a los programas, tanto en la rama de desarrollo como la que tenemos en producción. Teóricamente, las nuevas funcionalidades de tu aplicación las programarás dentro de la rama de desarrollo, pero constantemente tienes que estar resolviendo *bugs*, tanto en la rama de producción como en la de desarrollo.

Nota: Un sistema en producción se refiere a que está disponible para los usuarios finales. O sea, es una versión que está ya puesto en marcha y por lo tanto debería funcionar correctamente. Un sitio web, cuando lo estás creando, está en su etapa de desarrollo y cuando lo liberas en el dominio definitivo y lo pones a disposición de tu cliente, y/o los usuarios de Internet en general, se dice que está en producción.

Para facilitarnos la vida existen sistemas como Git, Subversion, CVS, etc. que sirven para controlar las versiones de un software y que deberían ser una obligatoriedad en cualquier desarrollo. Nos ayudan en muchos ámbitos fundamentales, como podrían ser:

- Comparar el código de un archivo, de modo que podamos ver las diferencias entre versiones

- Restaurar versiones antiguas
- Fusionar cambios entre distintas versiones
- Trabajar con distintas ramas de un proyecto, por ejemplo la de producción y desarrollo

En definitiva, con estos sistemas podemos crear y mantener repositorios de software que conservan todos los estados por el que va pasando la aplicación a lo largo del desarrollo del proyecto. Almacenan también las personas que enviaron los cambios, las ramas de desarrollo que fueron actualizadas o fusionadas, etc. Todo este mundo de utilidades para llevar el control del software resulta complejo en un principio, pero veremos que, a pesar de la complejidad, con Git podremos manejar los procesos de una manera bastante simple.

Alternativas y variantes de sistemas de control de versiones

Comenzaron a aparecer los sistemas de control del versionado del software allá por los años setenta, aunque al principio eran bastante elementales. Para hacerse una idea, en los primeros sistemas existía una restricción por la que sólo una persona podía estar a la vez tocando el mismo código. Es posible imaginarse que cosas semejantes provocaban retraso en los equipos de trabajo, por ello, a lo largo de los años fueron surgiendo nuevos sistemas de control de versiones, siempre evolucionando con el objetivo de resolver las necesidades de los equipos de desarrollo.

Tenemos principalmente dos tipos de variantes:

Sistemas centralizados: En estos sistemas hay un servidor que mantiene el repositorio y en el que cada programador mantiene en local únicamente aquellos archivos con los que está trabajando en un momento dado. Yo necesito conectarme con el servidor donde está el código para poder trabajar y enviar cambios en el software que se está programando. Ese sistema centralizado es el único lugar donde está todo el código del proyecto de manera completa. Subversion o CVS son sistemas de control de versiones centralizados.

Sistemas distribuidos: En este tipo de sistemas cada uno de los integrantes del equipo mantiene una copia local del repositorio completo. Al disponer de un repositorio local, puedo hacer *commit* (enviar cambios al sistema de control de versiones) en local, sin necesidad de estar conectado a Internet o cualquier otra red. En cualquier momento y en cualquier sitio donde esté puedo hacer un commit. Es cierto que es local de momento, luego podrás compartirlo con otras personas, pero el hecho de tener un repositorio completo me facilita ser autónomo y poder trabajar en cualquier situación. Git es un sistema de control de versiones distribuido.

Llegados a este punto, uno quieren saber dónde se encuentra Github y qué tiene que ver con Git, por lo que intentaremos aclarar algo en este sentido. Git es un sistema de control de versiones distribuido. Con Git hacemos repositorios de software. GitHub es un servicio para hacer *hosting* de repositorios de software que se administra con Git. Digamos que en GitHub mantienes una copia de tus repositorios en la nube, que además puedes hacer disponible para otros desarrolladores. De todos modos, sigue leyendo que más tarde en este mismo artículo te explicaremos todo esto con más detalle.

Tanto sistemas distribuidos como centralizados tienen ventajas e inconvenientes comparativas entre los unos y los otros. Para no hacer muy larga la introducción, cabe mencionar que los sistemas un poco más antiguos como CVS o también Subversion, por sus características son un poco más lentos y pesados para la máquina que hace de servidor central. En los sistemas distribuidos no es necesario que exista un servidor central donde enviar los cambios, pero en caso de existir se requiere menor capacidad de procesamiento y gestión, ya que muchas de las operaciones para la gestión de

versiones se hacen en local.

Es cierto que los sistemas de control de versiones distribuidos están más optimizados, principalmente debido al ser sistemas concebidos hace menos tiempo, pero no todo son ventajas. Los sistemas centralizados permiten definir un número de versión en cada una de las etapas de un proyecto, mientras que en los distribuidos cada repositorio local podría tener diferentes números de versión. También en los centralizados existe un mayor control del desarrollo por parte del equipo.

De todos modos, en términos comparativos nos podemos quedar con la mayor ventaja de los sistemas distribuidos frente a los sistemas centralizados: La posibilidad de trabajar en cualquier momento y lugar, gracias a que siempre se mandan cambios al sistema de versionado en local, permite la autonomía en el desarrollo de cada uno de los componentes del equipo y la posibilidad de continuar trabajando aunque el servidor central de versiones del software se haya caído.

Sobre Git

Como ya hemos dicho, Git es un sistema de control de versiones distribuido. Git fue impulsado por Linus Torvalds y el equipo de desarrollo del Kernel de Linux. Ellos estaban usando otro sistema de control de versiones de código abierto, que ya por aquel entonces era distribuido. Todo iba bien hasta que los gestores de aquel sistema de control de versiones lo convirtieron en un software propietario. Lógicamente, no era compatible estar construyendo un sistema de código abierto, tan representativo como el núcleo de Linux, y estar pagando por usar un sistema de control de versiones propietario. Por ello, el mismo equipo de desarrollo del Kernel de Linux se tomó la tarea de construir desde cero un sistema de versionado de software, también distribuido, que aportase lo mejor de los sistemas existentes hasta el momento.

Así nació Git, un sistema de control de versiones de código abierto, relativamente nuevo que nos ofrece las mejores características en la actualidad, pero sin perder la sencillez y que a partir de entonces no ha parado de crecer y de ser usado por más desarrolladores en el mundo. A los programadores les ha ayudado a ser más eficientes en su trabajo, ya que ha universalizado las herramientas de control de versiones del software que hasta entonces no estaban tan popularizadas y tan al alcance del común de los desarrolladores.

Git es multiplataforma, por lo que puedes usarlo y crear repositorios locales en todos los sistemas operativos más comunes, Windows, Linux o Mac. Existen multitud de GUIs (Graphical User Interface o Interfaz de Usuario Gráfica) para trabajar con Git a golpe de ratón, no obstante para el aprendizaje se recomienda usarlo con línea de comandos, de modo que puedas dominar el sistema desde su base, en lugar de estar aprendiendo a usar un programa determinado.

Para usar Git debes instalarlo en tu sistema. Hay unas instrucciones distintas dependiendo de tu sistema operativo, pero en realidad es muy sencillo. La instalación en linux la podemos hacer desde los propios repositorios de la distribución.Podremos ver cuáles son los primeros pasos con Git y veremos que para realizar unas cuantas funciones básicas el manejo resulta bastante asequible para cualquier persona. Otra cosa es dominar Git, para lo cual te hará falta más tiempo y esfuerzo.

Sobre Github

Como se ha dicho, Github github.com es un servicio para alojamiento de repositorios de software gestionados por el sistema de control de versiones Git. Por tanto, Git es algo más general que nos sirve para controlar el estado de un desarrollo a lo largo del tiempo, mientras que Github es algo más particular: un sitio web que usa Git para ofrecer a la comunidad de desarrolladores repositorios

de software. En definitiva, Github es un sitio web pensado para hacer posible el compartir el código de una manera más fácil y al mismo tiempo darle popularidad a la herramienta de control de versiones en sí, que es Git.

Cabe destacar que Github es un proyecto comercial, a diferencia de la herramienta Git que es un proyecto de código abierto. No es el único sitio en Internet que mantiene ese modelo de negocio, pues existen otros sitios populares como Bitbucket que tienen la misma fórmula. No obstante, aunque Github tenga inversores que inyectan capital y esté movido por la rentabilidad económica, en el fondo es una iniciativa que siempre ha perseguido (y conseguido) el objetivo de hacer más popular el software libre. En ese sentido, en Github es gratuito alojar proyectos *Open Source*, lo que ha posibilitado que el número de proyectos no pare de crecer, y en estos momentos haya varios millones de repositorios y usuarios trabajando con la herramienta.

Pero ojo, para no llevarnos a engaño, al ser Git un sistema de control de versiones distribuido, no necesito Github u otro sitio de alojamiento del código para usar Git. Simplemente con tener Git instalado en mi ordenador, tengo un sistema de control de versiones completo, perfectamente funcional, para hacer todas las operaciones que necesito para el control de versiones. Claro que usar Github nos permite muchas facilidades, sobre todo a la hora de compartir código fuente, incluso con personas de cualquier parte del mundo a las que ni conoces.

Esa facilidad para compartir código del repositorio alojado en la nube con Gitgub y la misma sencillez que nos ofrece el sistema de control de versiones Git para trabajar, ha permitido que muchos proyectos Open Source se hayan pasado a Github como repositorio y a partir de ahí hayan comenzado a recibir muchas más contribuciones en su código.

Quizás te estés preguntando ¿cómo obtienen dinero en Github si alojar proyectos en sus repositorios es gratis? Realmente solo es gratuito alojar proyectos públicos, de código abierto. El servicio también permite alojar proyectos privados y para ello hay que pagar por una cuenta comercial o un plan de hosting que no es gratuito. Existen planes iniciales, para alojar hasta cinco proyectos privados a partir de 7 dólares por mes, lo que resulta bastante barato. Ese mismo plan es gratuito para los estudiantes universitarios.

Github además se ha convertido en una herramienta para los reclutadores de empleados, que revisan nuestros repositorios de Github para saber en qué proyectos contribuimos y qué aportaciones hemos realizado. Por ello, hoy resulta importante para los programadores no solo estar en Github sino además mantener un perfil activo.

Resumen de comandos Git

Comandos principales para trabajar con Git de forma individual.

Al final del tema se incluyen enlaces a apartados del libro **Pro Git**. Es un libro totalmente recomendable, deberías bajártelo y guardarlo como material de aprendizaje y de referencia. Está disponible de forma gratuita en múltiples versiones (PDF, eBook, HTML y mobi).

Comandos básicos

- Inicializar git en un directorio:

```
$ cd /ruta/a/mi/directorio
$ git config --global user.name <nombre-usuario>
$ git config --global user.email <email>
$ git init
```

```
$ git add .
$ git commit -m "Versión inicial"
```

- Publicar por primera vez el repositorio local en el remoto (en GitHub):

  ```
  $ git remote add origin https://github.com/<usuario>/<nombre-repo>.git
  $ git push -u origin master
  ```

  El nombre del repositorio remoto será `origin` (nombre estándar del repositorio remoto en el caso en el que sólo haya uno). Subimos al repositorio la rama `master` (la rama por defecto que se crea al inicializar el repositorio local).

- Comprobar el estado del repositorio local:

  ```
  $ git status
  ```

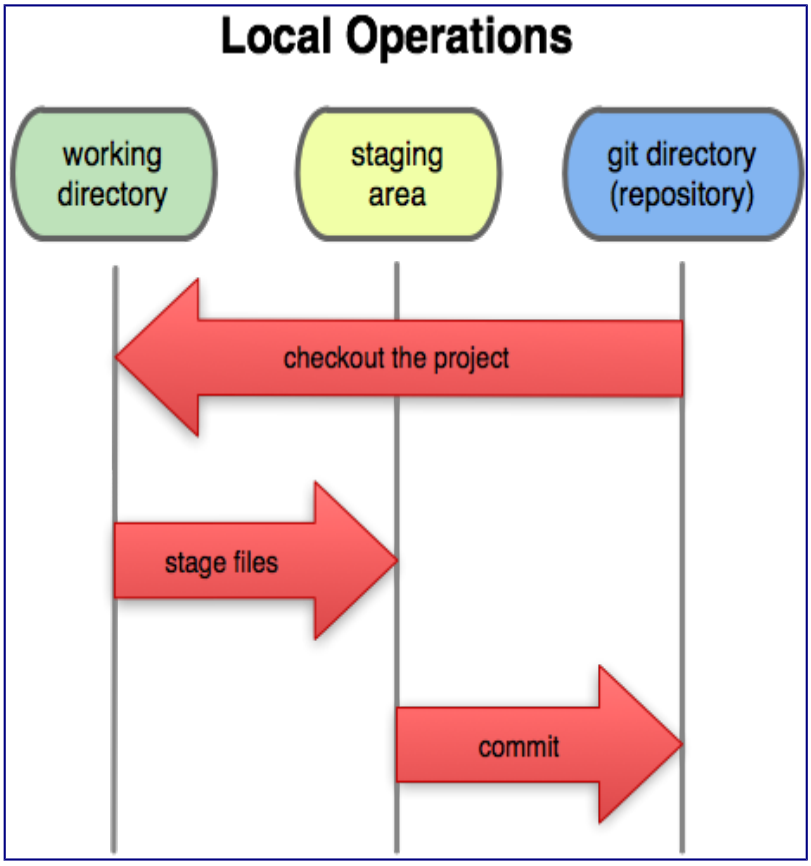
- Comprobar las diferencias entre los ficheros modificados en el directorio de trabajo y el último commit:

  ```
  $ git diff
  ```

- Añadir un fichero al *stage* (añadirlo para el próximo commit):

  ```
  $ git add <fichero o directorio>
  ```

  El área de *stage* también se llama el *index*. Es muy importante entender su funcionamiento para trabajar con Git. El siguiente dibujo muestra su funcionamiento:



- Hacer un commit de los ficheros en el *stage*:

```
$ git commit -m "Mensaje"
```

- Eliminar un fichero del *stage* (si lo hemos añadido, pero al final decidimos no añadirlo en el siguiente commit):

  ```
  $ git reset HEAD <fichero>
  ```

- Se puede combinar en un único comando el `add` y el `commit` en ficheros ya añadidos al control de versiones:

  ```
  $ git commit -a -m "Mensaje"
  ```

  Se puede abreviar como

  ```
  $ git commit -am "Mensaje"`
  ```

- Eliminar todos los cambios realizados en el directorio, volviendo al último commit:

  ```
  $ git reset --hard HEAD
  $ git clean -fd (si se ha añadido algún fichero)
  ```

- Publicar los cambios en el repositorio remoto:

  ```
  $ git push
  ```

- Consultar los mensajes de los commits (toda la historia de la rama actual). La opción `--oneline` muestra sólo la primera línea del mensaje, la opción `--graph` muestra el grafo de dependencias y la opción –`all` muestra el grafo completo, no sólo aquel en el que estamos (HEAD).

  ```
  $ git log [--oneline] [--graph] [--all]
  ```

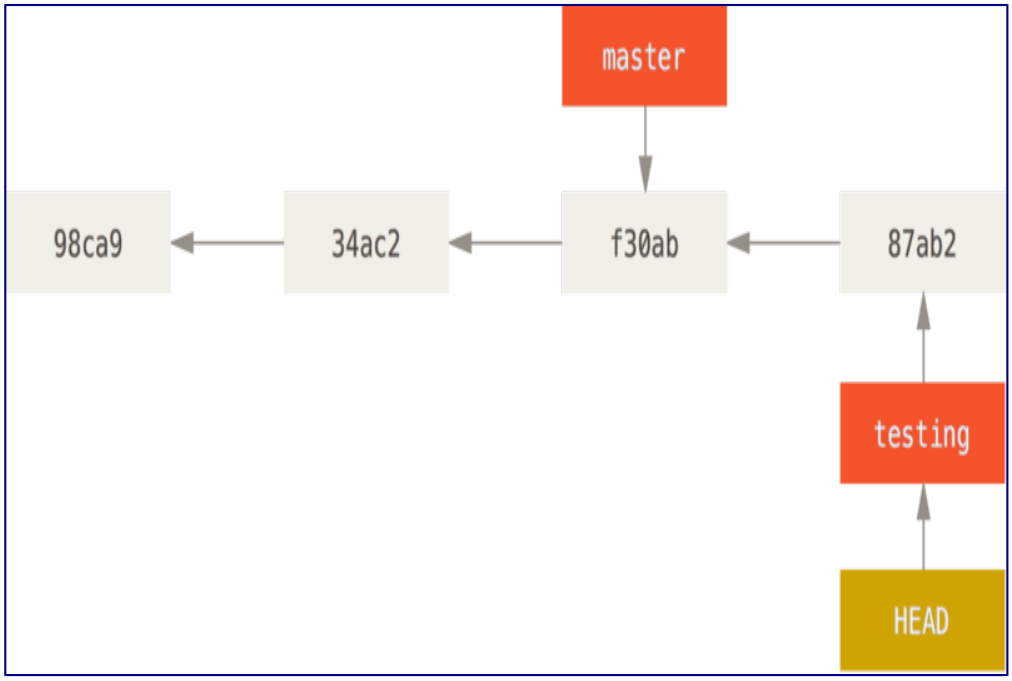
- Comprobar las diferencias entre dos commits:

  ```
  $ git diff <hash-previo> <hash-nuevo>
  ```

  Devuelve las cambios que se han introducido desde el commit identificado por y hasta el .

Ramas

Es muy importante entender que las ramas en Git son como etiquetas móviles. La rama en la que estamos se actualiza de posición cada vez que hacemos un nuevo commit. Git mantiene en la referencia HEAD la rama actual.



- Crear una rama nueva:

```
$ git checkout -b nueva-rama
M   hola.txt (si hay cambios en el espacio de trabajo se llevan a la
nueva rama)
Switched to a new branch 'nueva-rama'
```

- Listar las ramas de un repositorio:

```
$ git branch
master
* nueva-rama
$ git commit -a -m "Confirmamos los cambios en la nueva rama"
```

- Moverse a otra rama:

```
$ git checkout master
Switched to branch 'master'
```

- Mostrar un fichero de una rama (o commit) dado:

```
$ git show <commit o rama>:<nombre-fichero>
```

- Comparar dos ramas:

```
$ git diff master nueva-rama
```

  El comando `git diff master nueva-rama` devuelve las diferencias entre las ramas `master` y `nueva-rama`: las modificaciones que resultarían de mezclar la rama `nueva-rama` en la rama `master`.

- ***Merge* de ramas**: Mezclar la rama `nueva-rama` en la rama `master` (añade a la `master` los commits adicionales de la rama `nueva-rama`):

```
$ git checkout master
```

```
$ git merge [--no-ff] nueva-rama -m "Mensaje de commit"
```

La opción `--no-ff` no hace un fast forward y mantiene separados los commits de la rama en el log de commits. Es útil para revisar la historia del repositorio.

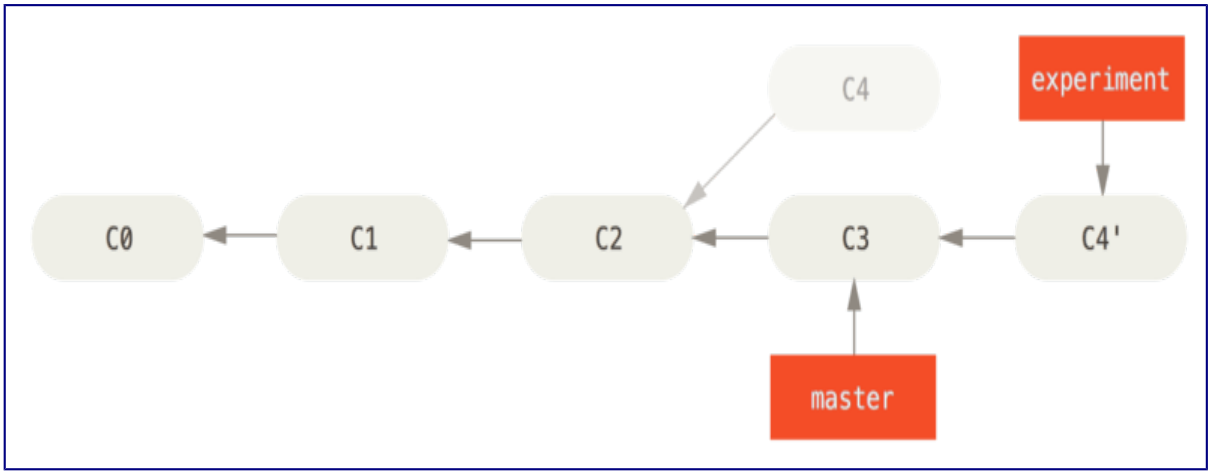
- Si en la rama que se mezcla y en la actual hay cambios que afectan a las mismas líneas de un fichero, git detecta un conflicto y combina esas líneas conservando las dos versiones y añadiendo la información de la procedencia. Debemos resolver el conflicto: editarlos a mano y volver a hacer add y commit.

```
$ git merge nueva-rama
CONFLICT (content): Merge conflict in hola.txt
Automatic merge failed; fix conflicts and then commit the result.
# editar a mano el fichero con conflictos
$ git commit -a -m "Arreglado el conflicto en el merge"
$ git merge nueva-rama
```

El comando `git status` después de un merge nos indica qué ficheros no se han mezclado y hay que editar manualmente.

- ***Rebase* de una rama**. Si la rama master ha avanzado después de separar una rama alternativa y queremos incorporar esos cambios en la rama alternativa podemos hacer un `git rebase`:

```
$ git checkout -b experiment
# hacemos cambios
$ git commit -m "Cambios en experiment"
$ git checkout master
# hacemos cambios
$ git commit -a -m "Cambios en master"
$ git checkout experiment
$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: Corregido bug1
Applying: Corregido bug2
```



El comando cambia la historia de la rama: primero la mueve al final de la rama master (*rewind head*) y a partir de ahí aplica los cambios propios de la rama.

IMPORTANTE: No se debe hacer un *rebase* de commits que existan en otros repositorios locales de compañeros. Al volver a aplicar los commits sobre los commits rebobinados, se

cambia su número de hash (identificador) y se convierten en commits distintos.

Una vez que hemos hecho el *rebase* ya podemos añadir mover la rama `master` y tener una historia lineal:

```
$ git checkout master
$ git merge nueva-rama
# Borramos la rama una vez mezclada
$ git branch -d nueva-rama
```

- Igual que en el *merge*, al hacer un rebase pueden aparecer conflictos al hacer el *rebase*, basta con modificar los ficheros con conflictos, añadirlos y continuar el *rebase*:

```
$ git rebase master
CONFLICT (content): Merge conflict in <some-file>
# hacemos git status para comprobar donde están los conflictos
$ git status
# Unmerged paths:
# (use "git reset HEAD <some-file>..." to unstage)
# (use "git add/rm <some-file>..." as appropriate to mark resolution)
#
# Editamos los ficheros para corregir los conflictos
$ git add <some-file>
$ git rebase --continue
```

IMPORTANTE: Es posible integrar los cambios de una rama haciendo un *merge* o haciendo un *rebase*. Ambas estrategias son correctas y cada una tiene sus pros y contras. Nosotros vamos a usar ambas para aprender su funcionamiento.

- Log en forma de grafo:

```
$ git log --graph --oneline
```

- Borrar una rama:

```
$ git branch -d nueva-rama
Deleted branch nueva-rama (was c241d7b)
```

Sólo podemos borrar de la forma anterior ramas en las que no estamos y que se han mezclado con alguna otra. El comando anterior no permite borrar ramas activas que tienen commits sin mezclar con otras.

- Borrar una rama descartando sus commits:

```
$ git branch -D rama
```

- Subir una rama al repositorio remoto:

```
$ git push -u origin <rama>
```

Para no tener que escribir la contraseña del repositorio remoto cada vez puedes utilizar el siguiente comando que la guarda en una caché:

```
$ git config --global credential.helper cache.
```

- Descargar una rama del repositorio remoto (origin, por ejemplo, el repositorio remoto por defecto)

```
$ git fetch
$ git checkout -b <rama> origin/<rama>
```

- Consultar ramas locales y conexiones repositorio remoto (origin, por ejemplo)

```
$ git remote show origin
```

- Subir todas las ramas y etiquetas:

```
$ git push -u -all origin
```

  Al poner la opción -u hacemos tracking del repositorio remoto y las referencias quedan almacenadas en el fichero de configuración .git/config. A partir de ahora sólo es necesario hacer `git push` para subir los cambios en cualquiera de las ramas presentes.

- Borrar una rama en repositorio remoto:

```
$ git push origin --delete <branchName>
```

Modificar la historia

- Modificar el mensaje del último commit. Se abrirá un editor en el que modificar el mensaje. También se puede escribir el mensaje a mano:

```
$ git commit --amend [--m "Nuevo mensaje"]
```

- Deshacer el último commit (sólo la acción del commit, dejando los cambios en el *stage*):

```
$ git reset --soft HEAD^
```

- Descartar el último merge y volver a la situación anterior al hacer el merge:

```
$ git reset --merge ORIG_HEAD
```

- Movernos atrás a un commit pasado, mirar los ficheros, crear una nueva rama allí (o no) y volver al commit actual:

```
$ git checkout <hash> (o tag, por ejemplo v2.0)
You are in 'detached HEAD' state.
# Ahora estás en un detached HEAD
$ git branch
* (HEAD detached at 594b606)
master
$ git checkout -b v2.0.1
Switched to a new branch 'v2.0.1'
$ git branch
master
* v2.0.1
$ git checkout master
```

- Movernos atrás a un commit pasado, descartando todos los commits realizados después (**peligroso**)

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