GIT. Contenidos Generales:

Introducción a GIT: Instalación. Configuración. Estados.

Repositorio Local: Creando un repositorio. Add y Commit.

Deshacer cambios. Historial de cambios.

Repositorio remoto con GitHub.

Creación de un repositorio remoto.

Sincronizar GIT con GitHub.

Conflictos.

Pull y Push.

Introducción a GIT

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.

Para conocer uno de los sistemas de control de versiones existentes que en la actualidad se ha popularizado hasta convertirse casi en un standard, 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 esta lectura entenderás que 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: marcar 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.

Para aclarar, 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 el trabajo 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.

Existen 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 y se utiliza cuando se necesita conectar 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.

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.

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 nos ha ayudado a ser más eficientes en nuestro 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 en ventanas gráficas, 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 página oficial de descargas está en gitscm.com, en donde encontrarás para descarga lo que se llama la "Git Bash" es decir, la interfaz de Git por línea de comandos. Tienes que descargar aquella versión adecuada para tu sistema (o si estás en Linux instalarla desde los

repositorios dependiendo de tu distribución). Aparte de Windows, Linux o Mac, también hay versión para Solaris. La instalación es muy sencilla, lo que resulta un poco más complejo al principio es el uso de Git.

Conceptos e Instalación

Conceptos rápidos sobre Git y GitHub

Antes de estudiar a fondo Git y comenzar a practicar, hay que cubrir algunas cuestiones que existen cuando se comienza a trabajar con un sistema de control de versiones. Se trata de algunas claves rápidas sobre las características de Git, el flujo de trabajo y algunas diferencias con GitHub.

En Git, cada desarrollador dispone de un repositorio completo instalado en su máquina; es algo inherente a los sistemas de control de versiones distribuidos, como vimos en el artículo sobre Qué son Git y GitHub. Es condición indispensable. Todos los cambios de nuestros archivos a lo largo del desarrollo los vamos a tener en local. Opcionalmente, esos cambios los enviaremos a repositorios remotos, como puede ser GitHub o cualquier otro.

Esto quiere decir que mi máquina tendrá todo lo que necesito para trabajar. Tendremos una copia del repositorio completo, cada uno de los archivos de todo el proyecto. Con el repositorio Git en local, luego decidiré a qué otros servidores o máquinas mando mis cambios.

GitHub es un hosting de repositorios Git, por tanto, el uso que le daremos es como repositorio remoto. Pero debe quedar claro que primero debo tener los cambios en el repositorio local y luego podré "empujar" esos cambios al repositorio remoto.

Por tanto, ya se ve la primera diferencia entre Git y GitHub: Git es la tecnología para hacer el control de versiones y GitHub simplemente es un hosting de repositorios Git, con una interfaz web que nos ofrece algunas utilidades basadas en el propio control de versiones Git.

En GitHub puedo tener repositorios diversos y si quiero trabajar con alguno de ellos, primero debo tenerlo en local. No necesitamos tener todos los repositorios que has publicado en GitHub en local, solo aquellos con los que vayamos a trabajar. En el momento que los tenemos en local podremos hacer cambios, almacenarlos en nuestro repositorio local y cuando lo juzguemos oportuno, enviarlo (empujar, push) a tantos servidores o repositorios remotos como queramos.

Nota: también al referirnos a GitHub hablamos como un repositorio remoto en general. Es decir, realmente lo que describimos de GitHub en líneas generales sirve para entender otros servicios de hosting de repositorios Git como Bitbucket.

Para concluir y tener más claro el flujo de trabajo con un control de versiones Git, imaginar un equipo de trabajo con varios componentes. Todos y cada uno de los desarrolladores deberán tener una copia completa de todo el repositorio de software que se está desarrollando. Luego el equipo decidirá a qué servidor con un repositorio remoto quiere enviar los cambios.

Cada desarrollador podrá enviar los cambios que tiene en local hacia el repositorio remoto y ese repositorio remoto lo usarán todos los componentes del equipo para sincronizarse y tener la versión más nueva del código en cada momento que lo deseen.

Este esquema podemos considerarlo como la base del trabajo con Git, aunque luego en la práctica hay que resolver algunos temas importantes.

Instalar Git

Tener Git instalado en local es condición indispensable para trabajar con el sistema de control de versiones. El proceso para instalar Git es bien sencillo porque no difiere de la instalación de cualquier otro software que hayas hecho.

Te tienes que descargar la versión de tu sistema operativo en la página oficial (o si usas Linux lo bajarás de los repositorios de software que usas habitualmente en tu distribución).

gitscm.com

Lo instalas como cualquier otro software. Si estás en Windows tendrás un asistente al que harás "siguiente, siguiente" hasta acabar el proceso. Puedes ver este vídeo que aclara algunos puntos sobre la instalación.

Para usuarios Windows, ¿Git Bash?
El único sitio donde puedes tener dudas es en el paso que te dice si quieres instalarlo como comando en la línea de comandos de tu consola o si simplemente quieres el "git bash".

Si lo instalas en la propia consola del Windows, la única ventaja es que lo tendrás disponible desde la ventana de línea de comandos de Windows y podrás hacer desde allí los comandos de Git. Si lo instalas solo en Git Bash no habrá más problemas, solo que cuando quieras usar Git tendrás que abrir la consola específica de Git que ellos llaman "git bash".

La manera más sencilla de saber si Git está instalado en tu sistema es a través del comando:

git version

Esto te mostrará en la pantalla el número de la versión de Git que tengas instalada. Si en Windows instalaste Git en consola (la ventana de línea de comandos), observarás que en cualquier consola de comandos de Windows tienes disponible Git. Si lo instalaste únicamente en el Git Bash, no tendrás ese comando y Git no está disponible desde la consola de Windows.

Git Bash es la línea de comandos de Git para Windows, que además te permite lanzar comandos de Linux básicos, "ls -l" para listar archivos, "mkdir" para crear directorios, etc. Es la opción más común para usar Git en Windows.

Es indiferente si instalas Git en la línea de comandos del Windows o si lo instalas solamente en el Git Bash. Simplemente escoge la que te sea más cómoda.

Primera configuración de Git, primeros comandos que debes lanzar
Antes que nada, inmediatamente después de instalar Git, lo primero que deberías hacer es lanzar un par de comandos de configuración.

```
git config global user.name "Tu nombre aquí"
git config global user.email "tu_email_aquí@example.com"
```

Con estos comandos indicas tu nombre de usuario (usas tu nombre y apellidos generalmente) y el email. Esta configuración sirve para que cuando hagas commits en el repositorio local, éstos se almacenen con la referencia a ti mismo, meramente informativa. Gracias a ello, más adelante cuando obtengas información de los cambios realizados en el los archivos del "repo" local, te aparecerá como responsable de esos cambios a este usuario y correo que has indicado.

Nota: recomendamos ver el video del Profesor Alejandro Zapata relacionado a este tema en el material multimedia de la Unidad.

Consola de GIT

Abrir una consola para trabajar con Git
Una vez instalado Git, vamos a trabajar con la consola de comandos para aprender de verdad a usar este sistema. Existen diversas aplicaciones de interfaz gráfica para trabajar con Git, pero no son realmente las mejores opciones para aprender a usar este sistema. Es por ello que te recomendamos aprender por línea de comandos, tal como te explicamos en estos artículos.

Si estamos en Windows, podremos abrir el "Git Bash" (ver artículo mencionado anteriormente) y si estamos en Linux / Mac, simplemente tendremos que abrir un terminal del sistema operativo.

Una vez dentro de tu consola de comandos, vamos a crear un directorio donde vamos a dejar todos nuestros repositorios. Aunque realmente puedes tenerlos dispersos en las carpetas que desees, eso es indiferente.

Nota: Si no sabes qué es un repositorio, podemos aclarar que es simplemente como una estantería de tu biblioteca donde guardas un software. Es un lugar, una carpeta, donde almacenas todos los archivos de un proyecto y cuando trabajas con Git tienes la posibilidad de tener no solo el estado actual de tus ficheros, sino el estado por el que han pasado en todas sus versiones a lo largo de la vida de ese software.
Formas de comenzar a trabajar con Git
Para trabajar con Git o Github (sabemos las diferencias porque ya lo explicamos en el artículo de conceptos iniciales) tenemos dos formas de trabajar:

1) Trabajar en local, en un repositorio que me cree en mi máquina.
2) Clonar un repositorio de Github (u otro hosting de repositorios) para traernos a local el repositorio completo y empezar a trabajar con ese proyecto.

Vamos a elegir de momento la opción 1) que nos permitirá comenzar desde cero y con la que podremos apreciar mejor cuáles son las operaciones básicas con Git. En este sentido, cualquier operación que realizas con Git tiene que comenzar mediante el trabajo en local, por lo que tienes que comenzar por crear el repositorio en tu propia máquina. Incluso si tus objetivos son simplemente subir ese repositorio a Github para que otras personas lo puedan acceder a través del hosting remoto de repositorios, tienes que comenzar trabajando en local.

Crear una carpeta para tu proyecto y colocar archivos
Entonces, estando en una carpeta de tu ordenador donde hemos dicho que vamos a crear todos nuestros repositorios, puedes crear una carpeta específica para tu primer proyecto con Git. La carpeta de tu proyecto la puedes crear con el explorador de archivos o si quieres por línea de comandos, es indiferente.

Una vez creada tu carpeta puedes crear archivos dentro. Como estamos hablando de desarrollo de software, los archivos que meterás dentro de tu carpeta contendrán el código de tu aplicación, página web, etc.

Inicializar el repositorio Git
Para crear tu repositorio Git, donde monitorizamos los archivos de un proyecto, tienes que realizar una operación previa. Es la inicialización del repositorio Git que queremos crear

en la carpeta de tu proyecto y es una operación que deberás realizar una única vez para este proyecto.

Nota: Cada proyecto lo tendrás en una carpeta distinta y será un repositorio independiente. Esta operación de inicialización de tu repositorio, por tanto, la tendrás que realizar una única vez para cada proyecto que quieras controlar sus versiones por medio de Git.
Así pues, antes que nada (antes de enviar cualquier archivo al repositorio), creo el repositorio Git con el comando "git init".

git init

Una vez has inicializado el repositorio, podrás observar que se ha creado una carpeta en tu proyecto llamada ".git" . Si ves esa carpeta en tu proyecto es que el repositorio se ha inicializado correctamente y que ya tienes convertida esa carpeta en un repositorio de software Git.

Guardar los archivos en el repositorio (commit)
Una vez que crees tus primeros archivos, puedes comenzar a trabajar con Git, enviando esos ficheros al repositorio. Aunque los archivos estén en la carpeta de tu proyecto y hayas iniciado el repositorio Git previamente, el sistema de control de versiones no los está monitorizando, por lo que a nivel de tu repositorio Git todavía es como si no estuvieran.

A esa acción de guardar los archivos en el repositorio se llama, en la jerga de Git, hacer un "commit". En este caso el commit lo estás haciendo en local, porque los archivos los estás enviando a tu repositorio local que tienes en tu máquina.

Un commit en Git se hace mediante dos pasos.

1) Tengo que añadir el fichero a una zona intermedia temporal que se llama "Zona de Index" o "Staging Area" que es una zona que utiliza Git donde se van guardando los ficheros que posteriormente luego se van a hacer un commit.

Cualquier archivo que quieras mandar a la zona de index lo haces con el comando "add".

git add nombre-del-fichero

Una vez añadido podrías ver que realmente tienes ese fichero en el "staging area", mediante el comando "status".

git status

Verás que, entre las cosas que te aparecen como salida de ese comando, te dice que tienes tu fichero añadido como "new file". Además te dice que estos cambios están preparados para hacer commit.

2) Tengo que enviar los archivos de la zona de Index al repositorio, lo que se denomina el commit propiamente dicho. Lo haces con el siguiente comando:

git commit -m "mensaje para el commit"

Con esto ya tenemos nuestro primer archivo dentro del repositorio. A partir de aquí podremos mantener y controlar los cambios que se hagan sobre este archivo (u otros que hayamos incluido por medio del commit).

Si haces de nuevo un comando "git status" podrás comprobar que no hay nada para enviar al repositorio. Eso quiere decir que la zona de Index está limpia y que todos los cambios están enviados a Git.

Etiquetas

Git tiene la posibilidad de marcar estados importantes en la vida de un repositorio, algo que se suele usar habitualmente para el manejo de las releases de un proyecto. A través del comando "git tag" podemos crear etiquetas, en una operación que se conoce comúnmente con el nombre de "tagging". Es una operativa que tiene muchas variantes y utilidades, nosotros veremos las más habituales que estamos seguros te agradará conocer.

Además de mantener informados a los usuarios del código de los proyectos y otros desarrolladores de las versiones de una aplicación, el etiquetado es una herramienta fundamental para que otros sistemas sepan cuándo un proyecto ha cambiado y se permitan desencadenar procesos a ejecutar cada vez que esto ocurre. De hecho es importante porque algunos gestores de paquetes te obligan a usarlo para poder publicar packages en ellos. Así pues, vamos a relatar las bases para trabajar con el sistema de tagging de modo que puedas usarlo en tu día a día en el trabajo con Git.

git tag

Numeración de las versiones
Git es un sistema de control de versiones. Por tanto permite mantener todos los estados por los que ha pasado cualquier de sus archivos. Cuando hablamos de Git tag no nos referimos a la versión de un archivo en particular, sino de todo el proyecto de manera global. Sirve para etiquetar con un tag el estado del repositorio completo, algo que se suele hacer cada vez que se libera una nueva versión del software.

Las versiones de los proyectos las define el desarrollador, pero no se recomienda crearlas de manera arbitraria. En realidad la recomendación sería darle un valor semántico. Esto no tiene nada que ver con Git, pero lo indicamos aquí porque es algo que consideramos interesante que sepas cuando empiezas a gestionar tus versiones en proyectos.

Generalmente los cambios se pueden dividir en tres niveles de "importancia": Mayor, menor y pequeño ajuste. Si tu versión de proyecto estaba en la 0.0.1 y haces un cambio que no altera la funcionalidad ni la interfaz de trabajo entonces lo adecuado es versionar tu aplicación como 0.0.2. Si el proyecto ya tiene alguna ampliación en funcionalidad, pero sigue manteniendo completa compatibilidad con la versión anterior, entonces tendremos que aumentar el número de enmedio, por ejemplo pasar de la 1.0.0 a la 1.1.0. Ahora bien, si los cambios introducidos en el proyecto son tales que impliquen una alteración sobre cómo se usará esa aplicación, haciendo que no sea completamente retrocompatible con versiones anteriores, entonces habría que aumentar en 1 la versión en su número más relevante, por ejemplo pasar de la 1.1.5 a la 2.0.0.

Realmente importa poco ahora el tema de la semántica, porque queremos hablar de Git. Sin embargo lo encuentras muy bien explicado en este documento Versionamiento Semántico 2.0.0-rc.2, por lo que, si te interesa el tema, te recomendamos leerlo.

Crear un tag con Git
Se supone que cuando comienzas con un repositorio no tienes ninguna numeración de versión y ningún tag, por lo que empezaremos viendo cómo se crean.

Supongamos que empezamos por el número de versión 0.0.1. Entonces para crear la correspondiente etiqueta lanzarás el subcomando de Git "git tag":

git tag v0.0.1 -m "Primera versión"
Como ves, es una manera de etiquetar estados del repositorio, en este caso para definir números de versión. Los acompañas con un mensaje, igual que se envían mensajes en el commit.

Nota: Este es el mecanismo que se conoce como "Etiquetas ligeras", existen otros tipos de etiquetado que es posible hacer mediante Git.
Generalmente, después de hacer cambios en tu repositorio y subirlos al sistema (después de hacer el commit), podrás generar otro número de versión etiquetando el estado actual.

git tag v0.0.2 -m "Segunda versión, cambios menores"
Ver los estados de versión en el repositorio con Git tag
Después de haber creado tu primer tag, podrás lanzar el comando "git tag", a secas, sin más parámetros, que te informará sobre las versiones que has etiquetado hasta el momento.

git tag
Si tienes un repositorio donde has etiquetado ya tres números de versiones, podría arrojar una salida como la que ves en la siguiente imagen.

Otro comando interesante en el tema de versionado es "show" que te permite ver cómo estaba el repositorio en cada estado que has etiquetado anteriormente, es decir, en cada versión.

git show v0.0.2
Recibirás como salida un mensaje parecido a este:

tag v0.0.2
Tagger: AlejandroZapata <ale.zapata@bue.edu.ar>
Date: Fri Oct 06 16:23:00 2020 -0200

"

commit 8ef366190b73d56e267c5324aa8074db3c3f0ed9
Tagger: AlejandroZapata <ale.zapata@bue.edu.ar>
Date: Fri Oct 06 16:23:00 2020 -0200

...
Enviar tags a GitHub
Si quieres que tus tags creados en local se puedan enviar al repositorio en GitHub, puedes lanzar el push con la opción --tags. Esto es una obligatoriedad, porque si no lo colocas, el comando push no va a enviar los nuevos tags creados.

git push --tags
En concreto la opción --tags, tal cual la hemos usado, envía todas las nuevas tag creadas, aunque podrías también enviar una en concreto mediante la especificación de la que quieres enviar, tal como se puede ver en el siguiente comando.

git push origin v0.0.4
En este caso debemos especificar qué repositorio remoto es el destino de las tags que acabamos de crear ("origin" en este caso), pues si no se especifica el comando entenderá

que el nombre de nuestro tag es el nombre del repositorio remoto que estamos queriendo usar para enviar los cambios locales, lo que nos dará un error.

Nota: Aparentemente, la opción --tag hace el mismo efecto que --tags. Las dos envían los tags que tengamos en local al repositorio remoto. Por eso puedes probar usar ambas, aunque en la documentación de Git usan siempre --tags.
git push --tag
Enviar tags y hacer push de los commits al mismo tiempo
Solo un pequeño detalle relativo al comando push cuando lo usamos para enviar tags. Cuando en el comando push usamos la opción --tags en principio no se mandan los cambios que tengamos en el repostiorio. Es decir, aunque hayamos hecho cambios en la rama master y se hayan realizado los correspondientes commits en local, si lanzamos "git push --tags", únicamente los nuevos tags se van a enviar a remoto. No se enviarán los commits que se hayan podido realizar en cualquier rama.

Si queremos hacer un push de una rama en concreto, por ejemplo la rama master, y enviar los tags al mismo tiempo, entonces podríamos lanzar el siguiente comando.

git push origin master --tags

Crear Repositorio y subir archivos

Crear un repositorio en Github

Desde Github creas un repositorio con el botón "+" de arriba a la derecha. Obviamente tienes que haberte registrado en Github para comenzar. El registro es gratuito.

Hacer el repositorio público o privado lo decide quien ha pedido realizar la tarea. Para hacer repositorios en privado tenés que tener una cuenta de pago en Github.

Subir el proyecto a Github con Push

Para ello, se utiliza el propio Git, el sistema de control de versiones. La operación que tienes que realizar se llama "push".

Desde la línea de comandos, aunque hay programas de interfaz gráfica que también te hacen estos pasos quizás más fácilmente.

Una vez tienes Git instalado, tienes que ir, en el terminal, a la carpeta de tu proyecto, entonces allí generas tu repositorio en local con la orden "init" (si es q no lo has hecho ya).

git init
Luego, desde la carpeta de haces el comando "add" para agregar todos los archivos al "staging area".

git add .
Luego lanzas el comando para el commit, que se lleva los archivos al repositorio para control de cambios. Es el siguiente:

git commit -m 'mi primer commit'
En vez de ‘mi primer commit’ pon algo que sea menos genérico y más relevante, relacionado con tu proyecto y el estado en el que estás ;)

Luego tienes que hacer el "push" desde tu repositorio local a remoto con los comandos que aparecen en la página de Github que hay justo después de haber creado el repositorio (allí donde te pedí que permanecieras con el navegador). Vuelve a tu navegador y los verás, abajo del todo, en la alternativa de subir un repositorio existente en local.

Es algo como esto:

git remote add origin https://github.com/aqui-tu-repo.git

Y luego haces el propio push también con git:

git push -u origin master

Estos dos comandos aparecen justo en la página que llegas al crear un repositorio. Es bueno copiar y pegar de allí, porque aparecerá la URL de tu repositorio en GitHub y así no corres el riesgo de equivocarte al escribir.

Git Ignore

Git tiene una herramienta imprescindible casi en cualquier proyecto, el archivo "gitignore", que sirve para decirle a Git qué archivos o directorios completos debe ignorar y no subir al repositorio de código.

Su implementación es muy sencilla, por lo que no hay motivo para no usarlo en cualquier proyecto y para cualquier nivel de conocimientos de Git que tenga el desarrollador. Únicamente se necesita crear un archivo especificando qué elementos se deben ignorar y, a partir de entonces, realizar el resto del proceso para trabajo con Git de manera habitual.

En el gitignore se especificarán todas las rutas y archivos que no se requieren y con ello, el proceso de control de versiones simplemente ignorará esos archivos. Es algo tan habitual que no debíamos de dejarlo pasar en el Manual de Git.

Por qué usar gitignore
Piensa que no todos los archivos y carpetas son necesarios de gestionar a partir del sistema de control de versiones. Hay código que no necesitas enviar a Git, ya sea porque sea privado para un desarrollador en concreto y no lo necesiten (o lo deban) conocer el resto de las personas. Pueden ser también archivos binarios con datos que no necesitas mantener en el control de versiones, como diagramas, instaladores de software, etc.

El ejemplo más claro que se puede dar surge cuando se trabaja con sistemas de gestión de dependencias, como npm, Bower, Composer. Al instalar las dependencias se descargan muchos archivos con documentos, tests, demos, etc. Todo eso no es necesario que se mantenga en el sistema de gestión de versiones, porque no forma parte del código de nuestro proyecto en concreto, sino que es código de terceros. Si Git ignora todos esos archivos, el peso total del proyecto será mucho menor y eso redundará en un mejor mantenimiento y distribución del código.

Otro claro ejemplo de uso de gitignore son los archivos que crean los sistemas operativos automáticamente, archivos que muchas veces están ocultos y no los vemos, pero que existen. Si no evitas que Git los procese, estarán en tu proyecto como cualquier otro archivo de código y generalmente es algo que no quieres que ocurra.

Implementar el gitignore
Como hemos dicho, si algo caracteriza a gitignore es que es muy fácil de usar. Simplemente tienes que crear un archivo que se llama ".gitignore" en la carpeta raíz de tu proyecto. Como puedes observar, es un archivo oculto, ya que comienza por un punto ".".

Nota: Los archivos cuyo nombre comienza en punto "." son ocultos solamente en Linux y Mac. En Windows los podrás ver perfectamente con el explorador de archivos.
Dentro del archivo .gitignore colocarás texto plano, con todas las carpetas que quieres que Git simplemente ignore, así como los archivos.

La notación es muy simple. Por ejemplo, si indicamos la línea

bower_components/

Estamos evitando que se procese en el control de versiones todo el contenido de la carpeta "bower_components".

Si colocamos la siguiente línea:

*.DS_Store

Estaremos evitando que el sistema de control de versiones procese todos los archivos acabados de .DS_Store, que son ficheros de esos que crea el sistema operativo del Mac (OS X) automáticamente.

Hay muchos tipos de patrones aplicables a la hora de especificar grupos de ficheros, con comodines diversos, que puedes usar para poder indicar, de manera muy específica, lo que quieres que Git no procese al realizar el control de versiones. Puedes encontrar más información en la documentación de gitignore, pero generalmente no lo necesitarás porque lo más cómodo es crear el código de este archivo por medio de unas plantillas que ahora te explicaremos.

Cómo generar el código de tu .gitignore de manera automática
Dado que la mayoría de las veces los archivos que necesitas ignorar son siempre los mismos, atendiendo a tu sistema operativo, lenguajes y tecnologías que uses para desarrollar, es muy sencillo crear un archivo .gitginore por medio de una especie de plantillas.

Existe una herramienta online que yo uso siempre que se llama gitignore.io. Básicamente permite escribir en un campo de búsqueda los nombres de todas las herramientas, sistemas, frameworks, lenguajes, etc. que puedas estar usando. Seleccionas todos los valores y luego generas el archivo de manera automática.

Por ejemplo una alternativa sería escribir las siguientes palabras clave: OSX, Windows, Node, Polymer, SublimeText.

Una vez generado el código de tu gitignore, ya solo te queda copiarlo y pegarlo en tu archivo .gitignore, en la raíz de tu proyecto.

Eliminar archivos del repositorio si te has olvidado el .gitignore
Nos ha pasado a todos más de una vez que se nos olvida generar el correspondiente .gitignore después de haber hecho un commit. Observarás que, por mucho que estés diciendo que ahora sí quieres ignorar ciertas carpetas o rutas, éstas continúan en tu repositorio. Básicamente, esto es así porque estaban allí antes de informar que se debían ignorar. En los siguientes commits serán ignoradas todas las modificaciones de las carpetas en cuestión, pero lo que había antes perdurará en el repositorio.

Por ejemplo, imagina que estás en un proyecto NodeJS. Olvidas de hacer el gitignore de la carpeta "node_modules". Entonces haces un commit y metes un montón de dependencias a tu repositorio git, que no debían estar. Si ves ahora en un sistema de interfaz gráfica tu repositorio (o en Github) podrás observar que los archivos de "node_modules" están ahí.

Luego creas tu .gitignore con el código para node, que puede ser muy grande pero donde querrás al menos ignorar los gestores de dependencias que puedas estar usando, como por ejemplo:

Código gitignore para evitar procesar los directorios de las dependencias

node_modules
jspm_packages

Nota: Las líneas que comienzan por "#" en un .gitignore son simplemente comentarios.
Ahora haces nuevos commits pero los archivos no se borran. ¿Qué hacer entonces?

Básicamente lo solucionas con un comando de Git llamado "rm" que básicamente funciona igual que el comando del mismo nombre que usas para borrar archivos en una consola del estilo de Mac o Linux.

git rm -r --cached node_modules
Luego tendrás que hacer un commit para que esos cambios se apliquen al sistema de control de versiones.

git commit -m 'Eliminada carpeta node_modules del repo'

A partir de ahora esa carpeta no se verá en tu repositorio y gracias a tu .gitignore tampoco se tendrá en cuenta en las siguientes operaciones que realices mediante Git.

Descargar vs Clonar

Al inicio de uso de un sitio como GitHub, si no tenemos ni idea de usar Git, también podemos obtener el código de un repositorio descargando un simple Zip.

Sin embargo, descargar un repositorio así, aunque muy sencillo no te permite algunas de las utilidades interesantes de clonarlo, como:

-No crea un repositorio Git en local con los cambios que el repositorio remoto ha tenido a lo largo del tiempo. Es decir, te descargas el código, pero nada más.
-No podrás luego enviar cambios al repositorio remoto, una vez los hayas realizado en local.

En resumen, no podrás usar en general las ventajas de Git en el código descargado. Así que es mejor
clonar, ya que aprender a realizar este paso es también muy sencillo.

Clonar el repositorio Git
Entonces veamos cómo debes clonar el repositorio, de modo que sí puedas beneficiarte de Git con el código descargado.

Luego abrirás una ventana de terminal, para situarte sobre la carpeta de tu proyecto que quieras clonar. Yo te recomendaría crear ya directamente una carpeta con el nombre del proyecto que estás clonando, o cualquier otro nombre que te parezca mejor para este repositorio. Te sitúas dentro de esa carpeta (cd, cd...) y desde ella lanzamos el comando para hacer el clon, que sería algo como esto:

git clone https://github.com/Escuela/java-avanzado.git . El último punto le indica que el clon lo vas a colocar en la carpeta donde estás situado, en tu ventana de terminal. La salida de ese comando sería más o menos como tienes en la siguiente imagen:

Instalar las dependencias
Habitualmente los desarrollos de Git tienen ignoradas las dependencias mediante el archivo .gitignore, por ello es importante que las instalemos de nuevo en el repositorio clon, en local.

Para cada tipo de proyecto, con cada lenguaje, existirán comandos para instalar las dependencias. Este comando lo tendrías que conocer tú mismo, si es de PHP dependerá de Composer, si es de NodeJS (o Javascript) dependerá de npm.

Por ejemplo, así reinstalamos las dependencias en un proyecto Angular, que usa npm como gestor de dependencias.

npm install

Esto depende del gestor de dependencias, o gestores de dependencias del proyecto, si es que se están usando.

Subir cambios al repositorio remoto realizados por nosotros
Una vez modificado el código en local podrás querer subir los cambios al repositorio remoto. Es algo muy sencillo, ya que al clonar el repositorio en local está asociado el

origen remoto desde donde lo trajiste. Sin embargo, tenemos que llevar en consideración algunos puntos:

Dependiendo de la empresa o del equipo de trabajo pueden haber algunas normas para enviar cambios, como por ejemplo trabajar siempre con ramas y enviar los cambios a una rama en concreto, que luego se puede fusionar con un pull request.

Si el repositorio que clonaste no es tuyo y no tienes permisos por no pertenecer a una organización que lo posea, obviamente, no podrás enviar cambios.

Nos vamos a quedar aquí en lo sencillo, pensando que: 1) El repositorio es tuyo y 2) que nadie se molesta por enviar cambios directamente a la rama master. Entonces, para enviar cambios a GitHub, o cualquier otro hosting de repositorios, haríamos:

git push
Con eso se envían los cambios en el instante y los podrás ver reflejados en el repo remoto.

Cómo enviar cambios a GitHub si el repositorio no es tuyo (vía fork)
Si el repositorio que vas a modificar no es tuyo y pretendes continuar el desarrollo, agregando cambios que querrás enviar a GitHub, tendrías que clonar el repositorio de otra manera.

Primero tienes que crear un "fork". Esto es, un repositorio que es copia de otro que ya está publicado en GitHub. Lo bueno de un fork es que el repositorio será exactamente igual, pero estará en tu cuenta de GitHub, con tu usuario, por lo que tendrás permisos para hacer lo que tú quieras con él.

Hay un botón para hacer el fork en la parte de arriba de la página del repositorio.

Una vez hecho el fork, el repositorio ya te pertenece. Entonces podrás clonar (el fork, no el repositorio original) y realizar los cambios que quieras, que podrás subir perfectamente a ti propio repositorio (tu fork).

En la parte de arriba puedes ver que el repositorio es un fork de otro repositorio existente en GitHub.

Eliminar Archivos

Se trata de un problema sencillo pero recurrente: borrar archivos de un repositorio GIT, que no deberían estar ahí por el motivo que sea.

Como sabes, existe un archivo llamado .gitignore con el que configuramos los archivos o carpetas que se deben pasar por alto al agregar contenido al repositorio. El típico contenido para meter en el .gitignore es archivos binarios, dependencias, variables de entorno, etc.

Gestionando correctamente el .gitignore conseguimos el efecto deseado, que solo entre en el repositorio aquel material que debería estar ahí. ¿Pero qué se puede hacer cuando hemos agregado al repositorio material que no queríamos trackear? Es decir, queremos que tales archivos continúen en el proyecto, pero no queremos que estén en el repositorio Git por cualquier motivo.

Cuando nos damos cuenta que estamos dando seguimiento a archivos que no deberíamos podemos encontrarnos en dos situaciones que vamos a resolver en este artículo:

Que el archivo se haya agregado simplemente al staging area, pero que no se haya hecho commit todavía a estos archivos

Que el archivo, o carpeta, esté ya en el repositorio, porque se haya hecho un commit anteriormente.

Uno o varios, antes de darnos cuenta que el archivo no debería estar en el repositorio.

Eliminar el seguimiento de archivos que sólo están en staging area
El "Staging Area", es el área intermedia de Git donde residen los archivos a los que se está haciendo seguimiento, antes de confirmarlos con un commit.

Cuando los archivos están en el staging area la operativa de eliminar los archivos del seguimiento con Git es bastante sencilla. Basta con lanzar un comando:

git reset HEAD nombre_de_archivo
Si quieres eliminar del staging area todos los ficheros del directorio donde nos encontramos.

git reset HEAD .
Recuerda que luego puedes lanzar un comando "git status" con el que puedes comprobar que los archivos ya no están en seguimiento. "Untracked files: …".

Eliminar del repositorio archivos que ya se han confirmado (se ha hecho commit)
El problema mayor te puede llegar cuando quieras quitarte de enmedio un archivo que hayas confirmado anteriormente. O una carpeta con un conjunto de archivos que no deberían estar en el repositorio.

Una vez confirmados los cambios, es decir, una vez has hecho commit a esos archivos por primera vez, los archivos ya forman parte del repositorio y sacarlos de allí requiere algún paso adicional. Pero es posible gracias a la instrucción "rm". Veamos la operativa resumida en una serie de comandos.

Borrar los archivos del repositorio
El primer paso sería eliminar esos archivos del repositorio. Para eso está el comando "rm". Sin embargo, ese comando tal cual, sin los parámetros adecuados, borrará el archivo también de nuestro directorio de trabajo, lo que es posible que no desees.

Si quieres que el archivo permanezca en tu ordenador pero simplemente que se borre del repostorio tienes que hacerlo así.

git rm --cached nombre_archivo
Si lo que deseas es borrar un directorio y todos sus contenidos, tendrás que hacer algo así:

git rm -r --cached nombre_directorio
El parámetro "--cached" es el que nos permite mantener los archivos en nuestro directorio de trabajo.

Asegurarse que estamos ignorando los archivos con .gitignore
Luego se trataría de asegurarse que nuestros archivos se encuentran correctamente ignorados, para que no los volvamos a meter en el repositorio cuando confirmemos cambios más adelante con commit.

Esto lo tienes que hacer en el archivo .gitignore y ya lo hemos explicado anteriormente, por lo que te recomiendo leer el artículo sobre .gitignore.

Hacer el commit para confirmar los cambios
Una vez hemos quitado del repositorio los archivos que no queremos, y ahora sí están ignorados, tenemos que confirmar los cambios. Esta operación hará que ese archivo o esa carpeta desaparezca del estado actual de la rama de nuestro repositorio.

git commit -m 'Eliminados archivos no deseados'

Enviar esos cambios al repositorio remoto
Como último paso, en el caso que tengas un repositorio remoto donde envías tu código (GitHub o similar), tendrás que trasladar los datos allí.

Esto lo haces con "push", como de costumbre.

git push
O en el caso que tengas que especificar el repositorio remoto y la rama, sería algo como:

git push origin master

Histórico de Git
Ten en cuenta que, aunque en un momento dado elimines archivos de tu repositorio, éstos seguirán en el histórico del repositorio y por tanto se podrán ver en commits anteriores, si es que alguien los busca. Es porque, cuando se confirmaron los cambios por primera vez, ya se indexaron por Git y quedarán registrados en los commits antiguos.