Curso GitHub

Sección 1 aprendizaje:

* Qué son (y para qué sirven) los sistemas de control de versiones y cómo pueden ayudar a nuestro flujo de desarrollo
  + Nos ayudan a mantener un historial de cambios;
  + Nos ayudan a tener control sobre cada cambio en el código;
  + Nos ayudan a que un cambio de una persona no influya en el cambio realizado por otra;
  + Etc.
* Qué es Git y cómo instalarlo
* Que con el comando git init pudimos crear un repositorio Git;
* Cómo analizar el estado de nuestro repositorio usando el comando git status.

También aprendí a como se usa bien el comando “ cd ” que no me estaba saliendo bien su uso pero se logro

Sección 2 Aprendizaje:

Solo para aclarar un poco, ya que entenderemos mejor cómo funciona Git durante el curso, aquí hay algunas definiciones interesantes:

* HEAD: Estado actual de nuestro código, es decir, donde nos colocó Git
* Working tree: Lugar donde los archivos realmente están siendo almacenados
* index: Lugar donde Git almacena lo que será *commiteado*, es decir, la ubicación entre el *working tree* y el repositorio de Git en sí.

Como vimos en el último video, podemos ver el historial de cambios de nuestro proyecto de manera muy sencilla, usando el comando git log.

Aunque es fácil, este comando es muy poderoso. Ejecuta git log --help y ve algunas de las posibles opciones. Para obtener algunos ejemplos más fáciles de entender, puedes buscar sobre git log o echar un vistazo a este link: <https://devhints.io/git-log>.

Commitear

Deberíamos generar un *commit* siempre que nuestra base de código esté en un estado que nos gustaría recordar. Nunca deberíamos tener *commits* de código que no funcionen, pero tampoco es interesante dejar el *commit* solo al final de una función.

Esta puede ser una discusión interminable y cada empresa o equipo puede seguir una estrategia diferente. ¡Estudien al respecto, comprendan qué tiene más sentido para ustedes y su equipo y sean felices! :-D

* Que un commit es la forma de guardar un estado o versión de nuestro código;
* Cómo agregar archivos para *commitear* con git add;
* Cómo *commitear* archivos usando el comando git commit;
* Cómo verificar el historial de *commits*, a través de git log y algunas de sus opciones:
  + git log --oneline
  + git log -p
  + git log --pretty="parámetros de formato"
* Cómo hacer que Git no monitoree archivos, a través de **.gitignore**
* Que no debemos realizar *commits*, es decir, guardar un estado, de nuestra aplicación que no esté funcionando.

Sección 3 Aprendizaje:

git **init** --bare

¡Opción correcta! Con este comando creamos un repositorio que no tendrá la *working tree*, o sea, no contendrá una copia de nuestros archivos. Como el repositorio servirá solo como servidor, para que otros miembros del equipo sincronicen sus trabajos, disminuimos espacio de almacenamiento de esta forma.

git remote add nombre-repositorio camino/del/repositorio

¡Opción correcta! De esta forma tendremos un link de nuestro repositorio local con el repositorio remoto, que llamamos de nombre-repositorio, que está almacenado en camino/del/repositorio.

git push [repositorio] master

¡Opción correcta! De esta forma enviamos las modificaciones de nuestra *Branch* master (hablaremos más sobre branches dentro de poco) hacia el repositorio remoto. Basta con substituir [repositorio] por el nombre que dimos al repositorio al agregarlo. Ahora, para traer los datos que están en el repositorio remoto, podemos utilizar git pull [repositorio] master.

* Lo que son los repositorios remotos
* Cómo crear un repositorio de Git sin una copia de los archivos (con --bare) para usar como servidor;
* Cómo agregar links a repositorios remotos, con el comando git remote add;
* Cómo descargar un repositorio por primera vez clonándolo con el comando git clone;
* Cómo enviar nuestros cambios a un repositorio remoto, con git push;
* Cómo actualizar nuestro repositorio con los datos en el repositorio remoto, usando git pull;
* Qué es **GitHub** y para qué sirve;
* Cómo crear un repositorio en **GitHub**;
* Cómo agregar un repositorio de **GitHub** como repositorio remoto.

Sección 4 aprendizajes:

Las *branches* ("ramas") se utilizan para desarrollar funcionalidades aisladas entre sí. La *branch* master es la *branch* "predeterminada" cuando creas un repositorio.

Es interesante separar el desarrollo de funcionalidades en diferentes *branches*, para que los cambios en el código de una no influyan en el funcionamiento de otra.

¿Cómo podemos hacer el merge de la Branch titulo hacia la Branch master?

Principio del formulario

git checkout master

y

git **merge** titulo

¡Opción correcta! De esta forma colocaremos el HEAD en la *Branch* master, o sea, haremos que nuestro código esté en el estado que lo dejamos con el últimocommit en master. Después, uniremos el trabajo de la *Branch* titulo con la *Branch* actual (master).

Final del formulario

En este escenario, ¿cuál es la diferencia entre los comandos rebase y merge?

El merge junta los trabajos y genera un merge commit. El rebase aplica los *commits* de otra Branch en la *Branch* actual.

¡Opción correcta! Con esto evitamos los *commits* de *merge*. Hay una larga discusión sobre lo que es “mejor”: rebase o merge; Estudie, busque, y genere sus propias conclusiones. Acá tienen un artículo (de miles de otros) que hablan sobre el asunto: https://medium.com/datadriveninvestor/git-rebase-vs-merge-cc5199edd77c.

Vimos lo simple que es resolver los conflictos identificados por Git al intentar hacer merge.

Ahora, genera un conflicto y, en lugar de usar merge, usa rebase para actualizar el master:

* Vé a la *branch* titulo
* *Commitea* algo
* Vé a la *branch* master, *commitea* un cambio en la misma línea
* Ejecuta git rebase titulo

Mira la salida de Git y usa la información que te da; después de corregir el conflicto, continúa con el rebase.

En esta aula aprendimos:

* Que una *branch* (o rama) es una línea de *commits* separada, y que se puede usar para desarrollar funcionalidades independientes;
* Que con *branches* separadas, podemos evitar que el código de una funcionalidad interfiera con otra;
* Cómo traer el trabajo realizado en una *branch* a otra *branch*, como la master, usando el comando git merge;
* Que git merge genera un nuevo *commit*, informando que hubo una mezcla entre dos *branches*;
* Cómo traer *commits* de una *branch* a otra con git rebase
* Que git rebase no genera un *commit* de merge, lo que simplifica nuestro *log*;
* Cómo presenta Git los conflictos;
* Cómo resolver conflictos y conservar sólo los cambios deseados con Git.

Seccion 5 aprendizajes:

¿Cuáles comandos, respectivamente, deshacen modificaciones antes de agregarlas (1), después de agregarlas, pero antes de *commitearlas* (2), y después de realizar el *commit* (3)?

1 - git restore

2 - git restore --staged

3 - git revert

¡Opción correcta! Con el git restore deshacemos una modificación que aún no fue agregada al index o stage, o sea, antes de hacer git add. Después de agregar con git add, para deshacer una modificación, necesitamos sacarlo de este estado, con git restore --staged. Ahora, si ya hicimos el commit, el comando git revert puede salvarnos.

Vimos cómo podemos utilizar git stash para almacenar temporalmente algunas de nuestras modificaciones.

¿En qué momento el stash es útil?

Cuando necesitamos parar en el medio del desarrollo de algo, para poder hacer otra cosa.

¡Opción correcta! Cuando necesitamos pausar el desarrollo de alguna funcionalidad, o corrección, antes de finalizar, y tal vez no sea interesante realizar un commit, pues nuestro código puede no estar funcionando aún. En este caso es interesante guardar el trabajo para poder volver a él después.

Resumidamente, ¿para qué sirve el comando git checkout?

Para dejar nuestro código en un estado determinado.

¡Opción correcta! La descripción del comando git checkout --help, en una traducción libre es: “Actualizar los archivos en *working tree* para que queden en la versión especificada. […]”.

Básicamente, podemos dejar nuestro código en el estado de último *commit* de una *branch*, de un *commit* específico, o incluso *tags* (que veremos más adelante).

En esta aula aprendimos:

* Que Git puede ayudarnos a deshacer cambios que no vamos a utilizar;
* Que, para deshacer un cambio antes de agregarlo para commit(con git add), podemos usar el comando git restore <archivos>;
* Que, para deshacer un cambio después de agregarlo paracommit, primero debemos ejecutar git restore --staged <archivos> y luego podemos deshacerlos con git restore <files>;
* Que, para revertir los cambios realizados en un commit, el comando git revert puede ser la solución;
* Que el comando git revert genera un nuevo commit informando que los cambios fueron deshechos;
* Que, para guardar un trabajo y reanudarlo más tarde, podemos usar git stash;
* Que, para ver qué cambios hay en el stash, podemos usar el comando git stash list;
* Que, con el comando git stash apply <número>, podemos aplicar un cambio específico al stash;
* Que el comando git stash drop <número> elimina un elemento determinado del stash;
* Que el comando git stash pop aplica y elimina el último cambio que se agregó alstash;
* Que git checkout sirve para dejar la copia del código de nuestra aplicación en el estado que queremos:
  + git checkout <branch> deja el código en el estado de una branchcon el nombre <branch>;
  + git checkout <hash> deja el código en el estado de *commit* con el hash <hash>.

Seccion 6 aprendizaje:

Con el comando git diff, vimos que es posible visualizar las modificaciones realizadas en determinado código. Podemos ver las diferencias entre *commits*, *branches*, etc.

¿Cómo exhibe git diff las modificaciones en el código?

+ línea agregada

- línea eliminada

- línea modificada (versión antigua)

+ línea modificada (nueva versión)

¡Opción correcta! El símbolo de sustracción (-) antes de la línea indica que ella no está más presente en el archivo. Ahora, el símbolo de adición (+) muestra que es una línea nueva. Las modificaciones son representadas por una eliminación y adición de línea.

¿Qué resultado genera el envío de una *tag* para Github?

Genera una *Release*, o sea, conseguiremos bajar un archivo compactado con nuestro código en este punto.

¡Opción correcta! Github nos da la posibilidad de bajar un archivo compactado que contiene el código en el estado en que el *tag* fue generado.

En esta clase aprendemos:

* Que es posible ver qué cambios se hicieron en cada archivo, con el comando git diff;
* Que, al escribir sólo git diff, vemos los cambios en nuestros archivos que no se agregaron para commit (con git add);
* Que es posible comparar cambios entre dos ramas con git diff <branch1> .. <branch2>
* Que es posible comparar los cambios realizados entre un commit y otro, usando el comando git diff <commit1> .. <commit2>;
* Que Git nos permite guardar “marcos” de nuestra aplicación, por ejemplo, generando versiones, a través de git tag;
* Que el comando git tag -a se usa para generar una nueva *tag*;
* Las ***Releases*** de GitHub, que son generadas para cada *tag* de Git que creamos en nuestro repositorio.