Taller de git

Impartido por Positive Zero - Abril, 2018

+0

Instructores

Alejandro A. Vilchis **Fernando J. Pinedo**

Desarrolladores de haztuhorario.com. Han colaborado en múltiples proyectos con Git. Tienen experiencia en la industria manufacturera, creando software especializado capaz de resolver problemáticas y generar ahorros económicos para la empresa.

+0

Temario

Módulo 1 Introducción

Módulo 2 *Git basics* (las bases)

Módulo 3 *Git branching* (ramificando)

Módulo 4 *Git contributing* (contribuyendo)

Módulo 5 Herramientas

+0

INTRO A GIT

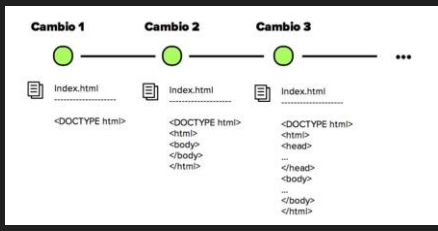
Sistema de Control de Versiones más utilizado en la actualidad

¿Qué es el control de versiones?

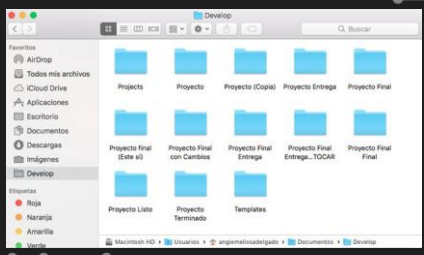
Se llama control de versiones a la **gestión de los diversos cambios** que se realizan sobre los elementos de algún producto o una configuración del mismo.

Una **versión**, revisión o edición de un producto, es el estado en el que se encuentra el mismo en un momento dado de su desarrollo o modificación

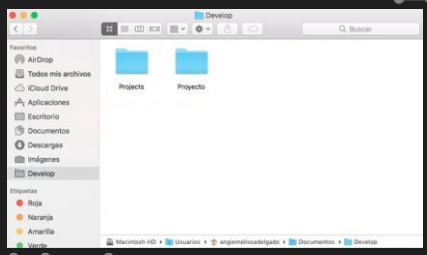
+0



+0



+0



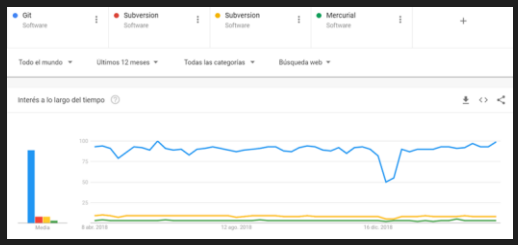
+0

VSC - Sistema de Control de Versiones

Software que facilita la administración de las distintas versiones de cada producto desarrollado.

+0

¿Por qué Git?



+0

¿Por qué Git?

- **Rendimiento**
  - Las operaciones de Git son rápidas, ya que gran parte está escrito en **lenguaje C** y por ser **distribuído**
  - Se pueden hacer en local **fuera de línea**.
  - Git no es engañado por el nombre de los archivos, se fija sólo en el **contenido**

+0

¿Por qué Git?

- **Trabajo colaborativo y flexibilidad**

Distintos **desarrolladores** pueden trabajar sobre un **mismo proyecto** y sus cambios se verán reflejados en el producto final, sin importar en qué parte del mundo se encuentren ni cuál sea su **flujo de trabajo**.

+0

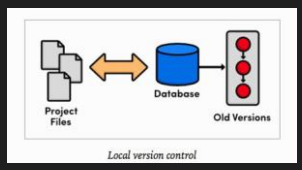
¿Por qué Git?

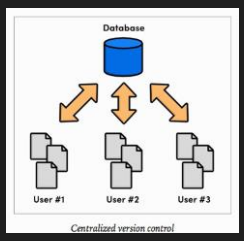
- **Escalabilidad**

Git es realmente **escalable** por lo que no varía si el proyecto en el que estés trabajando va en crecimiento. Puedes comenzar rápidamente **pequeños proyectos y** escalar hasta proyectos **muy grandes**.

+0

¿Por qué Git?





+0

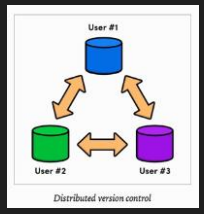
¿Por qué Git?

+0

¿Por qué Git?

- **DVCS**

Git es un ejemplo de un **Sistema Distribuido de Control de Versiones**. En lugar de tener un sólo lugar con el historial completo de las versiones de un proyecto (como lo hace **CVS** y **SVN**), en Git cada desarrollador tiene una copia del historial completo.

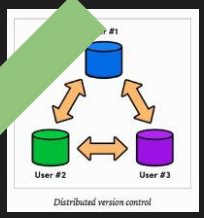


+0

¿Por qué Git?

- **DVCS**

Git es un ejemplo de un **Sistema Distribuido de Control de Versiones**. En lugar de tener un sólo lugar con el historial completo de las versiones de un proyecto (como lo hace **CVS** en Git cada desarrollador tiene del proyecto completo.



+0

¿Por qué Git?

- **Seguridad**
  - Todos los archivos, directorios y versiones de un proyecto en Git son protegidos con un **algoritmo criptográfico de seguridad** llamado **SHA1**, protegiendo todo el código de modificaciones accidentales y maliciosas.
  - Asegura el completo seguimiento de todo el historial.

`24b9da6552252987aa493b52f8696cd6d3b00373`

+0

¿Por qué Git?

- **Software libre y open source**
  - Tiene una gran **comunidad de apoyo** y cuenta con mucha **documentación**
  - No tienes que pagar para usar Git

+0

¿Por qué Git?

- **Es un estándar**
- Muchos IDEs tienen integración con Git
- Las empresas más grandes de desarrollo de software consideran git como un requerimiento básico para ser contratado

Microsoft Google +0

+0

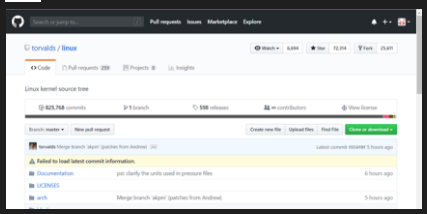
Orígenes de Git

- Creado por Linus Torvalds en 2005 para ser utilizado en el desarrollo del kernel de Linux.
- Sucesor de BitKeeper porque no cumplía con los requerimientos de Linus.

BitKeeper

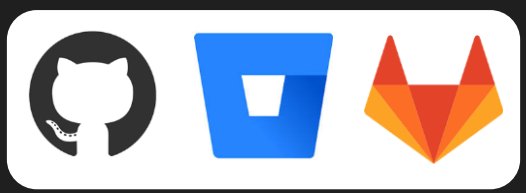
+0

Linux Kernel



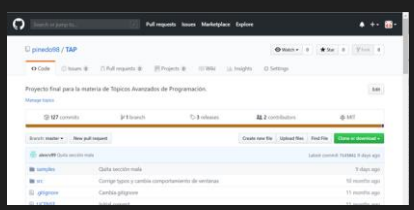
https://github.com/torvalds/linux +0

Hosting de Git



+0

"TAP"



https://github.com/pinedo98/tap +0

Terminología básica

Repositorio: es un contenedor de información digital que puede distribuirse a través de Internet. Pueden ser públicos o privados

Working directory: El directorio de trabajo es una copia de una versión del proyecto. Estos archivos se sacan de la base de datos comprimida en el directorio de Git, y se colocan en disco para que los puedas usar o modificar

Commit: Significa confirmar los cambios que deseas que se publiquen en el repositorio de Git

+0

GIT BASICS

Comandos básicos, inicializar repositorio, hacer commits

Instalación de materiales

Git Bash

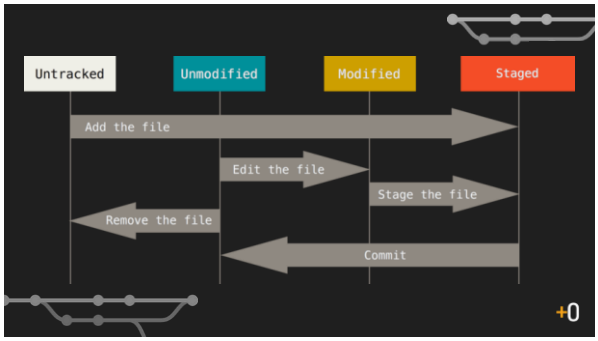
Sublime Text

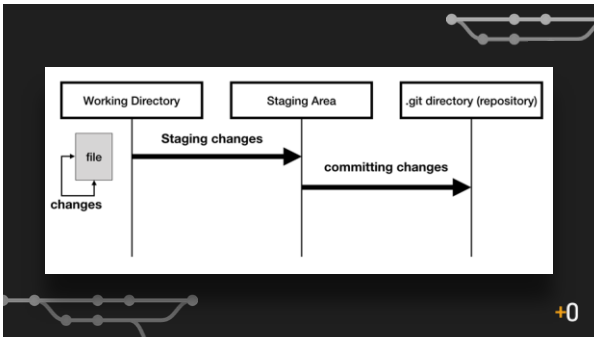
+0

Tipos de cambios en Git

- Agregar un nuevo archivo
- Modificar el contenido de un archivo
- Eliminar un archivo
- Mover o renombrar un archivo

+0





Índice

También llamado *Staging Area*, es el lugar en donde uno o más cambios permanecen **antes** de que se haga un commit.

Cuando haces un commit, lo que se *commitea* es lo que se encuentra actualmente en el índice.

Práctica de Git Basics

Comandos básicos para la consola

```
$ cd [directorio]   # Cambiar de directorio
$ ls                # Listar contenido de directorio
$ pwd               # Mostrar directorio actual
```

+0

Configuración inicial

```
$ git config  user.name "Alejandro Vilchis"
$ git config  user.email "alexrv4299@gmail.com"
```

```
$ git config –global user.name "Alejandro Vilchis"
$ git config –global user.email "alexrv4299@gmail.com"
```

+0

Crear un repositorio nuevo

Para inicializar un repositorio de Git, se utiliza el comando:

```
$ git init
```

Esto creará el directorio oculto *.git* en la raíz.

¡Ha comenzado la historia de tu proyecto!

+0

Ver el estado del repositorio

Puedes verificar el estado de tu repositorio con el comando:

```
$ git status
```

Mostrará:

- Los cambios hechos en el *working directory*
- Tu *branch* actual
- Los cambios agregados al *index*

+0

Agregar un cambio al *index*

Para preparar un cambio que se va a *commitear,* usa:

```
$ git add [archivo]          # Agregar el cambio hecho al archivo
$ git add .                  # Agregar todos los cambios
```

+0

Hacer un commit

Para realizar un commit en tu repositorio, ejecuta:

```
$ git commit
```

Esto abrirá un editor de texto (Vim por default) para que pongas el **mensaje de commit**. Al guardar el mensaje, se registrará el commit en el repositorio.

+0

Mensajes de commit

Es un texto libre que se debe agregar a cada *commit* en un proyecto, una descripción de los cambios hechos.

Ejemplo:

Mejora márgenes de la tabla en la vista móvil

Cambia el margen de la tabla del horario generado de 10px a 5px. Esto ayuda a que se pueda ver más información en menores resoluciones y...

+0

Recomendaciones

- Ser descriptivo
- Explicar el *por qué* en vez de *cómo*
- Usar imperativo para el asunto
- Comenzar con mayúscula
- Evitar rebasar los 50 caracteres en el asunto

+0

Hacer un commit con mensaje

Se puede especificar directamente el mensaje al hacer el commit:

```
$ git commit -m "Mi mensaje de commit"
```

Se sacrifica el cuerpo en el mensaje por un mensaje corto y descriptivo.

+0

Ver la historia de commits

Para visualizar la historia completa de commits:

```
$ git log
```

Mostrará:

- El id del commit
- El autor
- Fecha
- Asunto del mensaje

+0

Variantes de git log

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

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

```
$ git log --abbrev-commit
```

```
$ git log --reverse
```

+0

Cómo comparar las etapas

A veces tenemos la necesidad de ver qué cambios hemos hecho desde el commit anterior.

Nos mostrará qué líneas hemos borrado y qué líneas hemos agregado en cada archivo.

```
$ git diff
```

```
$ git diff --staged
```

+0

Salida de git diff

```
λ git diff
diff --git a/index.html b/index.html
index 2fbdcb4..9654c1f 100644
--- a/index.html
+++ b/index.html
@@ -17,7 +17,7 @@
                <li>Ver series</li>
            </ul>

-           <p>Hola, me llamo Alejanrdo</p>
+           <p>Hola, me llamo Alejandro</p>
```

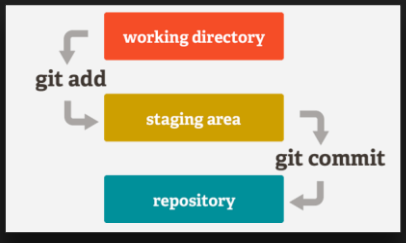
+0

Comparar dos commits

Con git diff, podemos incluso ver la diferencia entre dos commits pasados:

```
$ git diff 74ebc66833d3f39f4cfe698b4dce44b2ab6d196a
8a75499fed323743e7327089d6b522f3c6f488e2
```

+0

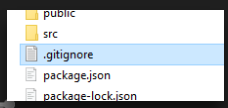


+0

El archivo .gitignore

Es un archivo especial que le dice a Git qué archivos (o tipos de archivos) **NO** incluir en el repositorio.

- Debe ubicarse en la raíz del repositorio.
- Cada línea del .gitignore especifica un patrón.



+0

Algunos patrones

- Especificando un nombre de **archivo**. Si queremos excluir readme.txt, sólo escribimos `readme.txt` en el archivo .gitignore.
- Especificando una **extensión**. Si queremos excluir todos los archivos .txt, escribimos `*.txt`.
- Especificando un **folder** completo. Si queremos excluir un folder llamado "test", escribimos `test/` en el archivo.

+0

Ejemplo de archivo .gitignore

.gitignore

```
1  # Ignore file named `README.txt`
2  README.txt
3
4  # Ignore folder named `output`
5  output/
6
7  # Ignore all .java files
8  *.java
```

+0

Tags

Una *tag* es una referencia a un punto específico de la historia con un nombre (usualmente números y puntos). Crea una con:

```
$ git tag [nombre]
```

```
$ git tag -a [nombre]
```

Una tag *annotated* guarda más información que una normal: autor, correo, fecha.

+0

Asignar tags a commits viejos

Por defecto, una tag se asocia al commit más reciente. Si se quiere asociar una tag a un commit viejo, debe agregarse el id:

```
$ git tag -a v1.2 15027957951b64cf874c3557a0f3547bd83b3ff6
```

Al igual que los commits, las tags requieren de un mensaje.

+0

Ver las tags creadas

```
$ git tag
v0.11.2
v0.12.0
v0.12.1
v0.12.2
```

+0

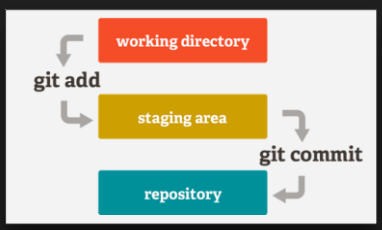
Repaso

- Git: **Sistema de Control de Versiones**
- ¿Por qué Git?
- Comandos básicos de git
  - git **init**
  - git **config**
  - git **status**
  - git **add**
  - git **commit**
  - git **log**
  - git **diff**
- .gitignore (carpetas y archivos)

+0

Repaso

Tres áreas:

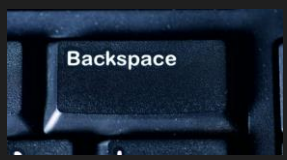


+0

Deshacer acciones

Existen dos tipos de cambios locales que se pueden deshacer:

- **Antes** de hacer **commit**
  - *Unstaged changes*
  - *Staged changes* (git add)
- **Después** de hacer **commit**



+0

Unstaged changes

Existen tres maneras de deshacer los cambios no *stageados*

- Descartar todos los cambios y guardarlos para después utilizarlos

```
$ git stash
```

- Descartar un cambio de un archivo de manera permanente

```
$ git checkout -- [nombre de archivo]
```

- Descartar todos los archivos de manera permanente

```
$ git reset --hard
```

+0

$ git stash

Guardar los cambios de manera temporal y regresarlos cuando sean necesarios

- Rehace lo stasheado anteriormente

```
$ git stash pop
```

- Enlista todos los cambios stasheados

```
$ git stash list
```

+0

Staged changes

Maneras de deshacer los cambios *stageados* (después de git add)

- Descartar todos los cambios y guardarlos para después utilizarlos

```
$ git stash
```

- *Unstagear* todos los cambios y retenerlos

```
$ git reset
```

- Regresar los cambios de un archivo al commit HEAD

```
$ git reset HEAD [nombre de archivo]
```

+0

Staged changes

Descartar todos los cambios de manera permanente

```
$ git reset --hard
```

+0

Commited changes

Existen tres maneras de deshacer los cambios

- Descartar los cambios de un commit (invertir líneas agregadas y eliminadas)

```
$ git revert [commit-id]
```

- Descartar los cambios de un archivo y retenerlos en estado staged

```
$ git checkout [commit-id] [nombre de archivo]
```

- Descartar los cambios de un archivo y retenerlos en estado unstaged

```
$ git reset [commit-id] [nombre de archivo]
```

+0

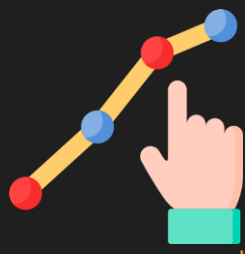
Viajar en el tiempo

Para regresar a un *commit* anterior se puede utilizar:

```
$ git checkout [commit-id]
```

Y para volver atrás utilizar:

```
$ git checkout -
```



+0

Deshacer cambios

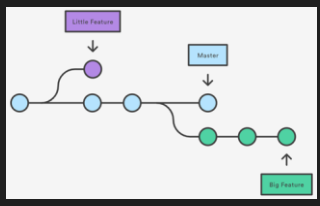
working area	staged area	repositorio
stash, checkout, reset --hard	stash, reset, reset HEAD, reset --hard	revert, checkout, reset

+0

GIT BRANCHING

Creando líneas de trabajo independientes

¿Qué es una rama/branch?

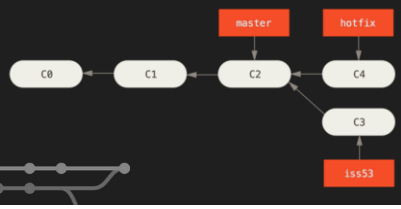


Las ramas son **versiones** del proyecto que pueden trabajar de manera **independiente**, permitiendo experimentar, hacer pruebas y desarrollar nuevos cambios sin afectarse entre ellas mismas.

+0

¿Qué es una rama/branch?

Una rama en Git es simplemente un **apuntador móvil** apuntando a una de las **confirmaciones** (commits). La rama por defecto de Git es la rama **master**.



+0

Accesibilidad de las ramas

Una rama Git es realmente un simple archivo que contiene los 40 caracteres de una suma de control SHA-1, (representando la confirmación de cambios a la que apunta), no cuesta nada el crear y destruir ramas en Git.

+0

Comandos básicos de git branching

Crear una nueva *branch*:

```
$ git branch [nombre rama]
```

Saltar o cambiar a una *branch*:

```
$ git checkout [nombre rama]
```

+0

Comandos básicos de git branching

En un sólo comando crear y saltar a una *branch*

```
$ git checkout -b [nombre rama]
```

Comando para borrar una *branch*

```
$ git branch -d [nombre rama]
```

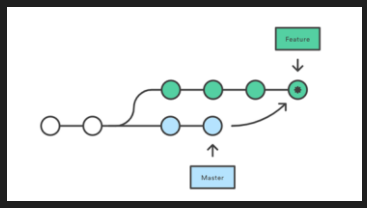
Comando para mostrar todas las *branches* disponibles

```
$ git branch <-a>
```

+0

Fusión de *branches*: *git merge*

Une **dos** líneas de desarrollo independientes (*branches*) en **una sola *branch***.



+0

Comando para fusionar *branches*

```
$ git merge [nombre branch]
```

El nombre branch se refiere a la rama que será fusionada con la branch actual.

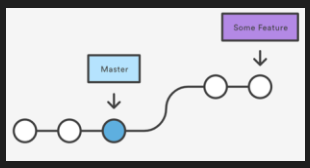
Ejemplo:

```
$ git checkout master
$ git merge funcionalidad-carrusel
$ git branch -d funcionalidad-carrusel
```

+0

Fast Forward Merge

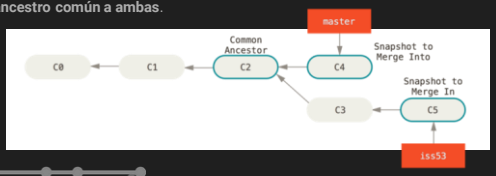
"**Avance rápido**", ocurre cuando el apuntador de Git sólo tiene que moverse hacia delante, pues las branches que se desean fusionar están **sobre la misma línea** y no hay avances divergentes a fusionar.



+0

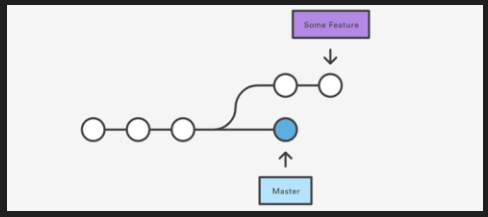
Fusión a tres bandas / *Three way merge*

Cuando las ramas divergen, Git hace trabajo extra: realizará una fusión a tres bandas, utilizando las dos *commits* por el extremo de cada una de las ramas y por el **ancestro común a ambas**.



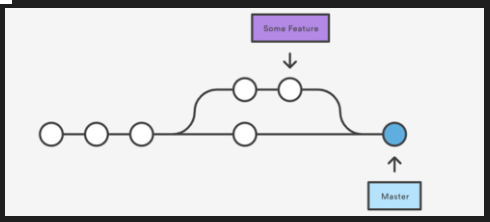
+0

Three way merge



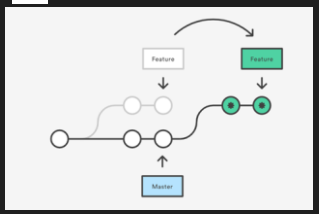
+0

Three way merge



+0

Git rebase (reorganización)



Rebase es otra forma de integrar cambios de una rama/*branch* en otra. En español se conoce como **reorganización** y es el proceso de mover los **commits** de una *branch* al último **commit** de la otra branch

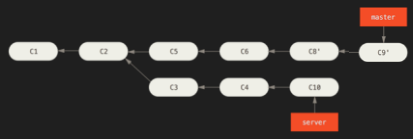
+0

Rebase

El siguiente comando tomará los *commits* de la ***branch*** de trabajo **actual** y los moverá al HEAD de la branch-a-mover

$ git rebase [branch-a-mover]

Ejemplo:



+0

Rebase

$ git checkout server

$ git rebase master

+0

Rebase

$ git checkout master

$ git merge server

+0

Rebase

$ git checkout master

$ git merge server

RECOMENDACIÓN IMPORTANTE: Nunca reorganices confirmaciones de cambio (commits) que hayas enviado (push) a un repositorio público.

+0

Branching workflow / Flujos de trabajo

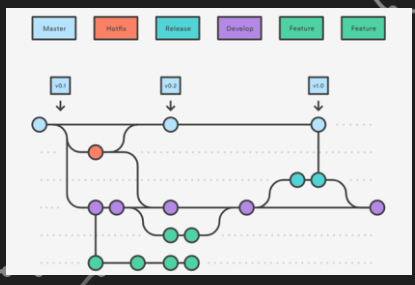
Ramas de largo recorrido

+0

Branching workflow / Flujos de trabajo

Ramas puntuales
Una rama puntual es aquella rama de corta duración que abres para un tema o para una funcionalidad determinada.

+0



+0

Repositorios remotos

Son versiones de tu proyecto que están hospedadas en Internet en cualquier otra red.

Comando para mostrar remotos configurados:

```
$ git remote -v
```

+0

Agregar remoto

Comando para agregar remoto a repositorio:

```
$ git remote add [nombre de remoto]  [URL del remoto]
```

Ejemplo:

```
$ git remote add origin  https://github.com/user/repo.git
```

+0

Enviar a remotos: *push*

Cuando tienes un proyecto que quieres compartir, debes enviarlo a un servidor. El comando para hacerlo es simple:

```
$ git push [nombre de remoto]  [nombre-branch]
```

Para obtener más información de un remoto puedes utilizar el comando:

```
$ git remote show [nombre de remoto]
```

+0

Git y SSH Keys

Para poder enviar cambios a un repositorio remoto debes de establecer un canal de comunicación segura.

El protocolo SSH te permite hacerlo sin tener que escribir tu cuenta y contraseña cada vez que te comuniques con el servidor.

+0

Pasos para generar nueva clave SSH

Utilizar siguiente comando para generar clave

```
$ ssh-keygen -t ed25519 -C "email@example.com"
```

Existen otros **tipos** de claves SSH que se pueden generar como las **RSA**, sin embargo, **ED25519** es la que tiene **mayor seguridad** y **mejor rendimiento** en la actualidad.

+0

Pasos para agregar clave SSH a GitHub

1. Copiar la clave en portapapeles
   a. En Mac

      $ pbcopy < ~/.ssh/id_ed25519.pub

   a. En Linux

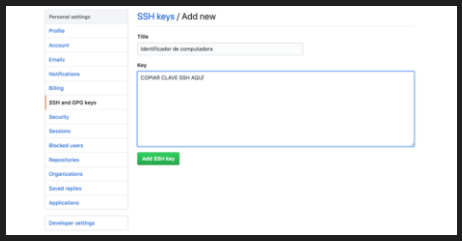
      $ xclip -sel clip < ~/.ssh/id_ed25519.pub

   a. En Windows

      $ cat ~/.ssh/id_ed25519.pub | clip

+0

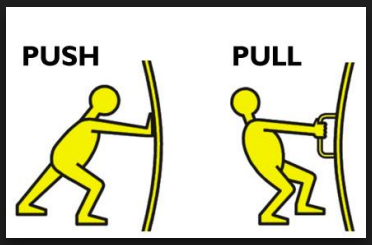
3. Crear nueva SSH Key en GitHub y pegar clave del portapapeles



+0

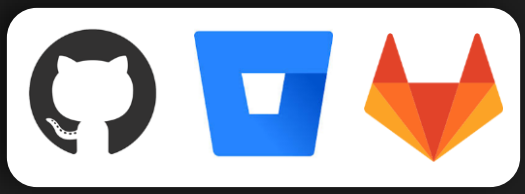
GIT CONTRIBUTING

¿Cómo trabajar en un sólo proyecto de manera sincronizada?



+0

Servicios de hosting de Git



+0

Git NO es Github

+0

GitHub

- Más popular de todos
- *Issue tracking*
- Propiedad de Microsoft (desde 2018)
- Enfocado a proyectos *open source*
- Repositorios privados gratis con 3 colaboradores

+0

GitLab

- Exportación de proyectos
- Herramientas de *deployment* integradas
- CI/CD
- Repositorios privados gratis con colaboradores ilimitados
- IDE Web

+0

Bitbucket

- Mejor integración con otras herramientas de Altassian
- Soporta Git y Mercurial
- Repositorios privados gratis con 5 colaboradores
- CI/CD integrado

+0

https://github.com/alexrv99/git-colab

+0

Generalidades

1. Cada miembro del equipo realiza commits en su repositorio local.
2. Cuando los cambios estén listos, el programador *pushea* sus commits al repositorio remoto.
3. Para poder hacer *push*, debiste haber hecho *pull* previamente.

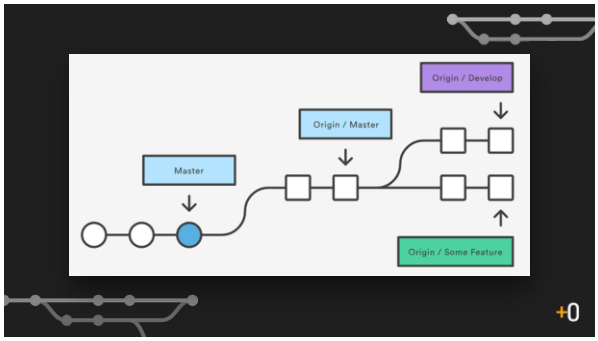
+0

Actualizar referencias remotas

Para sincronizar tu repositorio local con el remoto, simplemente:

```
$ git fetch
$ git fetch [remote]    # Todas las branches
$ git fetch [remote] [branch]
```

+0



Cómo actualizar la *branch* local

Ya que **git fetch** no integra los cambios a nuestras *branch* locales, necesitamos hacer un *merge* a nuestro repositorio local.

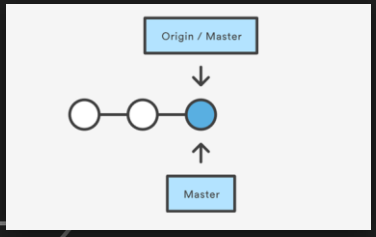
```
$ git merge [remote]/[branch]
```

```
$ git merge origin/master
```

origin/master es una *branch* local que es una copia de la *branch* remota.

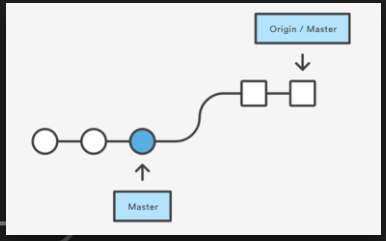
+0

Antes de git fetch



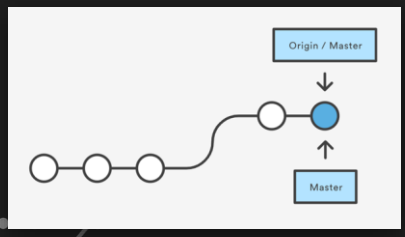
+0

Después de git fetch



+0

Después de git merge



+0

Adquirir cambios directamente

Para sincronizar tu repositorio local con el remoto:

```
$ git pull
```

Esto:

- Actualizará las referencias remotas
- Integrará los cambios a tu rama local

+0

Variantes de git pull

```
$ git pull                          # Solo la branch actual
$ git pull [remote]                 # Remote específico
$ git pull [remote] [branch]        # Branch específica
$ git pull --all                    # Todas las branches
```

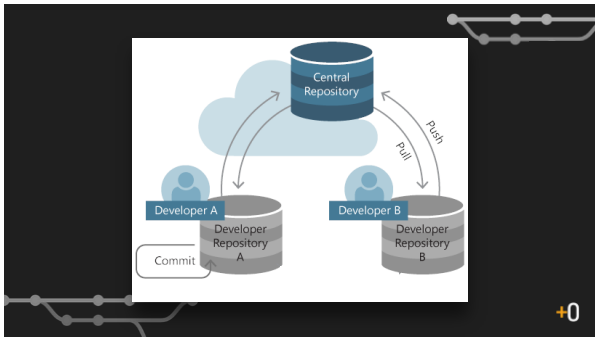
+0

```
$ git pull
```

=

```
$ git fetch
$ git merge origin/master
```

+0

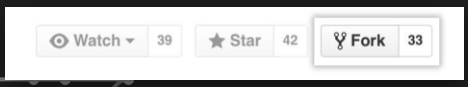


Fork

Se traduce como "bifurcación" y es copiar un repositorio a tu cuenta de Github.

Con un fork puedes hacer cambios a proyectos de otras personas para proponer cambios o simplemente para experimentar y crear algo nuevo.

Es equivalente a clonar un repositorio y luego publicarlo en tu cuenta.

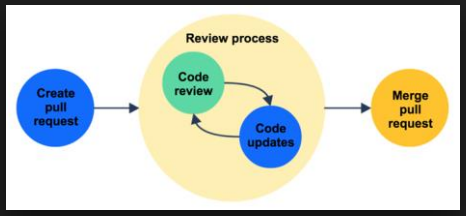


Pull request

Es una solicitud de integrar una *branch* a un proyecto. Uno o más *reviewers* deben aprobar los cambios.

Se utilizan en empresas con equipos grandes, en donde cada línea debe ser revisada y probada antes hacer *merge* en master.

Este concepto es implementado en servicios de Hosting para Git.



Push directo

Se pueden asignar permisos de escritura a una persona agregándolo como **colaborador** del proyecto.

No se requiere de una pull request para poder añadir commits al repositorio remoto.

```
$ git push [remote] [branch]
```

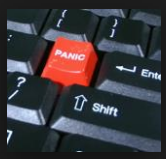
```
$ git push origin master
```

+0

Conflictos

Es una **contradicción** que sucede cuando dos personas modifican la misma región de un archivo.

Es habitual tener conflictos al ejecutar **git pull** o al hacer **git merge.**




```
D:\Documents\taller-git (master -> origin)
λ git merge otraBranch
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

+0

Cómo resolver un conflicto

Cuando ocurre un conflicto, Git agrega varias líneas para que el programador sepa qué causó un conflicto. Los conflictos deben arreglarse manualmente y se debe hacer un *commit*.

```
<<<<<<< HEAD
    <p>El mejor instructor es Alex Vilchis</p>
=======
    <p>El mejor instructor es Fernando Pinedo</p>
>>>>>>> otraBranch
```

Separador

Tu branch

Branch *mergeada*

+0

Cómo resolver un conflicto

1

```
<<<<<<< HEAD
    <p>El mejor instructor es Alex Vilchis</p>
=======
    <p>El mejor instructor es Fernando Pinedo</p>
>>>>>>> otraBranch
```

2

```
<p>El mejor instructor es Alex Vilchis</p>
```

3

```
D:\Documents\taller-git (master -> origin)
λ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Unmerged paths:
  (use "git add <file>..." to mark resolution)

        both modified:   index.html
```

4

```
D:\Documents\taller-git (master -> origin)
λ git add .

D:\Documents\taller-git (master -> origin)
λ git commit
[master 25992d1] Merge branch 'otraBranch'
```

+0

Cómo saber quién modificó un archivo

```
$ git blame [archivo]
```

Se mostrarán los siguientes datos de cada línea:

- Autor
- Fecha
- Commit de origen
- Número de línea

+0

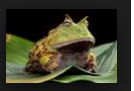
Instrucciones del reto

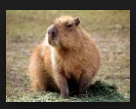
Júntate en equipo (3 personas) y crea un sitio web de un animal. Deben de crear un repositorio público en GitHub y agregar como colaboradores a sus compañeros. No se requiere un estilo específico ni una estructura compleja.

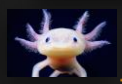
https://randomanimalgenerator.com

Páginas del sitio

- Información general
  - Características
  - Alimentación
- Hábitat
- Galería







+0

Requerimientos del reto

- Una *branch* para la página Hábitat
- Una *branch* para la página Galería
- Mínimo 3 commits por branch
- Poner archivo README.md
- *Mergear* todas las branches en master al final
- Excluir archivos con extensión .bak del repositorio
- Agregar hoja de estilos CSS (opcional)

+0