Indice o index

una zona de espera antes de que los archivos sean realmente añadidos al repositorio por un commit
Una vez creados los archivos se hace `$git add ejemplo.html` para agregar el archivo `ejemplo.html al índex y que sea tenido en cuenta al hacer el próximo commit`
$`git status -> verifica los archivos que están en el indice esperando al próximo commit`

Rama

Una rama corresponde en realidad a una versión paralela del curso de desarrollo.
Una rama puede servir para desarrollar nuevas funcionalidades o corregir posibles errores sin integrar estos cambios en la versión principal del software.
Las ramas permiten segmentar diferentes versiones en curso de desarrollo.

En cada repositorio Git, se encuentra una rama por defecto: la rama *master*. Esta rama *master* corresponde generalmente a la rama principal del proyecto.

En otras palabras, una rama se utiliza para mantener una versión especial del proyecto para un propósito específico y fijado de antemano.
Pueden servir para:

- desarrollar una nueva funcionalidad sin contaminar la rama superior con modificaciones no validadas,

- guardar una rama correspondiente a una versión estable definida. Por ejemplo, un desarrollador crea una versión 1.0 de un software y no desea incorporar en ella las últimas modificaciones.

`$git branch color_buttons -> crea una rema`
`$git checkout color_buttons ->` Este comando permite cambiar de rama. Un cambio de rama significa que el directorio de trabajo se actualizará a partir de los datos de la rama especificada en el comando.

`$git branch -> muestra la rama que estamos actualmente anteponiendo un asterisco al nombre`

TAGS

Los tags son referencias estáticas, es decir que un desarrollador añadirá un tag para servir de atajo con el fin de apuntar a un commit particular. La principal diferencia de los tags con respecto a las ramas es que estos no cambian. Si un desarrollador no modifica el tag de forma explícita, este señalará siempre al mismo commit sin que las modificaciones introducidas en el histórico tengan el menor efecto sobre este tag.

Clonar

Clonar un repositorio equivale a copiar el contenido de un repositorio a un nuevo repositorio. El nuevo repositorio contendrá un directorio de trabajo que permitirá recuperar y guardar las modificaciones.

El hecho de clonar un repositorio vuelve a copiar todo el contenido de un repositorio en un nuevo repositorio. Las ramas, los commits y todos los demás objetos almacenados por Git se duplicarán en un nuevo repositorio.

Clonar un repositorio no solo corresponde a copiar el directorio de trabajo, sino a copiar todos los elementos de un repositorio menos los archivos configurados en *.gitignore*

commit

Los commits deben representar un cambio único que satisfará una necesidad concreta (parche de un bug, módulo de una nueva funcionalidad, etc.).
Cada commit debe contener un mensaje de commit que explique el origen de los cambios.
`$git commit –m “app ejemplo y añadir en localstorage"`
`$git log –>` Este comando permite listar los commits del repositorio.
Por otra parte

Repositorio

Proyecto git

Repositorio remoto

Un remote es un repositorio remoto para que un exista un vínculo con el repositorio local.

directorio de trabajo

El directorio que nos encontramos corresponde con una rama, y dentro de

esa rama podemos estar trabajando en el ultimo commit o en uno anterior

índice
Archivo temporal donde se almacenan los archivos que se van a actualizar el el próximo commit
$git add archivo.txt -> comando para agregar archivos al indice

Historico

HEAD
Esta referencia apunta al commit más reciente de la rama actual. Es decir, que para cada nuevo commit y cada cambio de rama esta referencia se actualiza para apuntar al commit más reciente. Esta referencia siempre está presente al usar Git.

El software Tower (https://www.git-tower.com/mac), que permite gestionar los repositorios Git

git clon url -> clona repositorio

Comandos de utilidad

`$git log ->`
`$git log -> Muestra info de los commints de la rama en que estamos`
`$git log -n 2 -> Muestra los últimos dos commints`
`$git log --pretty=fuller -1 ->` ver en forma completa el commit más reciente

`$git log -1 --stat ->` visualizar las estadísticas del número de líneas añadidas y eliminadas en las modificaciones del commit.
`$git log --oneline -5 ->` para mostrar los cinco commits más recientes del proyecto solo en cinco líneas.
`$git log -3 --author "Pepe" ->` consultar los tres últimos commits realizados por Pepe
`$git log -2 -- README.rst ->` verificar los commits que llevan modificaciones del archivo *README.rst*

`$git diff ->` ver las diferencias entre el directorio de trabajo y el índice
`$git diff --cached ->` Este comando permite comprobar todo lo que se guardará justo antes de usar el comando `git commit`.

`$git diff HEAD ->` ver las diferencias entre la última versión del

repositorio (a la que apunta HEAD) y la versión actual en el directorio de trabajo

`$git reset --hard HEAD ->` elimina todas las modificaciones del directorio de trabajo y del índice. Es irreversible; por tanto, debe utilizarse con mucha prudencia. Vuelve al estado del ultimo commit, HEAD

`$git reset HEAD ->` Sin el argumento `--hard`, no modificará el directorio de trabajo. Este comando solo pondrá el índice en el mismo estado que la versión más reciente del repositorio. ¡Las modificaciones presentes en el índice se eliminarán de manera irreversible!

```
$git branch -> mostrar ramas existentes
     * master -> solo existe la rama master y el asterisco
indica que es la rama actual
```

`$git branch -v ->` mostrar la lista de las ramas con el hash y el mensaje del último commit de cada rama

`$git check-ref-format --branch nombre_rama ->` verificar si un nombre de rama es válido

```
$git branch -v -> observamos las ramas existentes
$git branch ejemplo -> creamos la rama ejemplo
```

`$git status ->` verificar los cambios pendientes de la rama actual

`$git stash ->` cambiar de rama rápidamente sin tener tiempo de terminar el código en curso para hacer un commit. permite conservar cambios de lado para volver a un estado idéntico entre el directorio de trabajo y la rama HEAD. El desarrollador puede entonces volver a importar sus modificaciones fácilmente sin haber tenido que crear un commit sin terminar.

`$git checkout ejemplo ->` permite ubicarse en la rama ejemplo
`$git checkout -b ejemplo2 ->` permite crear una nueva rama y ubicarse en ella directamente

```
$git checkout master -> se vuelve a la rama master
$git merge ejemplo -> la rama master incorpora todos los
commit de la rama ejemplo
```

`$git branch -d ejemplo ->` elimina la rama ejemplo

`$git remote ->` tener la lista de los repositorios remotos vinculados con el repositorio local

`$git push origin master` -> enviar al remote llamado *origin* las modificaciones añadidas en la rama *master*

`$git pull` -> actualizar la rama actual con los commits contenidos en el repositorio remoto

`$git branch --all` -> obtener la lista de todas las ramas (locales y remotas seguidas)

Listar los commits con git log

`$git log` -> nos permitirá ver mucha información sobre los commits. Este comando es muy potente y tiene muchas opciones. Por defecto se muestran todos los commits realizados durante toda la vida del repositorio.

```
$git log -n -> Limitar el número de commits mostrados

$git log -n 2 -> Muestra los últimos dos commints
$git log -2 -> Muestra los últimos dos commints
$git log --max-count=2 -> Muestra los últimos dos commints
```

`$git log -1 --stat` -> visualizar las estadísticas del número de líneas añadidas y eliminadas en las modificaciones del commit.

se puede ver cada commit en una sola línea mediante el argumento `--oneline`.
`$git log --oneline -5` -> para mostrar los cinco commits más recientes del proyecto solo en cinco líneas.

`$git log --oneline -5 --abbrev=40` -> igual al anterior pero muestra todo el hashs completo de cada commit

```
$git log --before="2015-02-09"
$git log --after="2015-02-09"
$git log --after="2015-02-09" --before="2015-02-13"
$git log --since=1.weeks
$git log --since=1.days
$git log --since=1.months
$git log --since=1.years
```

utilizando los argumentos `--author` o `--committer` se puede consultar los commits de uno o varios contribuyentes
`$git log -3 --author "Tim Graham"` -> consultar los tres últimos commits realizados por Tim Graham

`$git log -3 --author "Edward Henderson\|Tim Graham"` -> ver los tres commits más recientes realizados por Tim Graham y Edward Henderson

La diferencia entre los argumentos `--author` y `--committer` se explica, por ejemplo, cuando los cambios no son enviados al repositorio central (este tipo de repositorio se abordó en detalle en el capítulo Compartir un repositorio), pero cuando se envían por patch. Un patch corresponde a un `diff` (visualización de las modificaciones introducidas por un desarrollo), puede ser enviado por mail o cualquier otro medio posible. En este caso, el autor («author») del commit es la persona que ha efectuado el desarrollo y la envía al administrador del proyecto. En cuanto al commiter, corresponde a la persona que va a recibir, validar e integrar el parche en el repositorio.

`$git log --graph --oneline -8 ->` mostrar por consola un gráfico en las distintas ramas donde se efectuaron los commits.

Especificar un formato de salida

```
$git log --graph --pretty=format:'%Cred%h%Creset :
%Cgreen%d%Creset %s %C(yellow)(%cr) %C(bold blue)<%an>%Creset'
```

después del argumento `--pretty=format`, se encuentra una cadena que contiene un patrón. Este patrón está compuesto por palabras clave correspondientes a los elementos que hay que mostrar o los colores definidos para algunos tipos de elementos.

- `%n`: salto de línea.
- `%h`: hash abreviado del commit.
- `%H`: hash del commit.
- `%an`: nombre del autor del commit.
- `%ae`: e-mail del autor del commit.
- `%cr`: la fecha del commit.
- `%d`: nombre de refs, es decir, que esta palabra clave mostrará una lista de nombres que apuntan al commit. En el ejemplo anterior, la cadena (`HEAD`, `origin/master`, `origin/HEAD`, `master`) representa los objetos que apuntan al commit. Esta cadena tendrá más sentido después de haber leído el capítulo siguiente sobre las ramas y los tags.
- `%s`: título del commit.
- `%P`: hash del commit padre.

Luego, para cambiar de color, se puede utilizar las palabras clave:

- `%Cred`: color rojo.
- `%Cgreen`: color verde.
- `%Cblue`: color azul.
- `%Creset`: activa el color por defecto.

Existe una serie de formatos predefinidos integrados en Git. Estos formatos se utilizan con el argumento `--pretty`. Estos formatos se explican en la documentación oficial (http://git-scm.com/docs/pretty-formats). A continuación, una lista no exhaustiva de estos formatos clasificados de la salida más corta a la salida más completa:

- `oneline`: este formato permite visualizar cada commit en una sola línea mostrando solo el hash del commit y su mensaje.
- `short`: este formato es un compendio del formato estándar (sin fecha).
- `medium`: este formato es el formato por defecto.
- `full`: este formato muestra la misma información que el formato medium sustituyendo la fecha indicada por la persona que hizo el commit.
- `fuller`: este formato muestra el máximo de información, incluidas las fechas en que el autor y dueño del commit realizaron sus commits.

`$git log --pretty=fuller -1` -> utilizar el formato `fuller` para mostrar el commit más reciente

Tener en cuenta los merges

Los merges son las fusiones de varias versiones diferentes del proyecto, Suelen indicar una integración de nuevas funcionalidades o parches. Estos comandos son muy útiles cuando el desarrollador busca información relativa a un merge.

```
$git log --merges -2
$git log --no-merge
```

Listar los commits que afectan a un archivo

`$git log -2 -- README.rst` -> verificar los commits que llevan modificaciones del archivo *README.rst (*los dobles guiones sirven para separar el comando de los nombres de archivo*)*

```
$git log -2 -- README.rst Gruntfile.js -> Mismo que el
anterior pero de dos archivos
```

Mostrar las diferencias de contenido

muy utilizado se produce cuando el desarrollador quiere ver los cambios presentes en su directorio de trabajo en relación con su índice

`$git diff ->` ver las diferencias entre el directorio de trabajo y el índice

```
diff --git a/README.rst b/README.rst
index 6e9bc5f..9f4dcc9 100644
--- a/README.rst
+++ b/README.rst
@@ -41,4 +41,4 @@ To run Django's test suite:
 * Follow the instructions in the "Unit tests" section of
   docs/internals/contributing/writing-code/unit-tests.txt,
published online at https://docs.djangoproject.com/en/dev/
internals
/contributing/writing-code/unit-tests/#running-the-unit-tests
-Yo pruebo el comando git diff
+Yo pruebo el comando diff de git
```

La primera línea indica que se ha efectuado la diferencia entre las versiones del directorio de trabajo y el índice. La segunda línea identifica los dos archivos cuyas modificaciones se quiere obtener. La vista que importa realmente comienza en la línea 5, donde se visualiza el contenido del archivo. Las líneas que no llevan como prefijo el signo - ni el signo + son líneas que no han sido modificadas. Estas se utilizan principalmente para conocer las líneas que definen el contexto del código modificado. Las líneas con el prefijo + son las líneas que se han añadido en el directorio de trabajo. Las líneas con el prefijo - son las líneas que fueron eliminadas del directorio de trabajo.

Diferencias entre el índice y HEAD

> Otra situación donde `git diff` se utiliza mucho se produce antes de que el desarrollador haga commit de las modificaciones contenidas en el índice. Para validarlas y no hacer commit de errores, comparará las diferencias entre el índice y HEAD (recuerde, HEAD es el commit más reciente de la rama actual).

`$git diff --cached ->` Este comando permite comprobar todo lo que se guardará justo antes de usar el comando `git commit`.

Diferencias entre el directorio de trabajo y HEAD

`$git diff HEAD ->` ver las diferencias entre la última versión del repositorio (a la que apunta HEAD) y la versión actual en el directorio de trabajo

Diferencias introducidas por uno o varios commits

hay que comparar dos versiones: la versión del commit de la cual se quiere obtener los cambios y la versión de su antecesor

`$git log --pretty=%P -1 ->` identificar el commit padre de un commit

`$git diff HEAD^1..HEAD ->` ver los cambios introducidos por el último commit
El atajo `HEAD^1` permite solicitar a Git el commit padre de HEAD

Diferencias de palabras

Sin embargo, un código no tiene la misma definición de lo que es una palabra que un texto redactado. En efecto, en un texto redactado o literario las palabras están separadas por espacios. En un código, una sola línea puede presentar varios elementos que no se encuentran separados por un espacio. Se debe indicar a Git los separadores que deseamos para separar las palabras en `git diff`. Luego se debe utilizar una expresión regular para indicar a Git lo que es una palabra en el código empleando el argumento `--color-words`:

`$git diff --color-words=. ->` indica a Git que una palabra debe ser considerada como un carácter. En efecto, un punto se interpreta para definir cualquier carácter en una expresión regular.

Identificar al autor de una línea de código

es frecuente preguntarse cuándo, quién y por qué una línea de código se ha añadido o modificado en un archivo

`$git blame README.rst ->` todas las líneas del archivo con la siguiente información antes de cada una de ellas:

- El hash del commit que ha introducido o modificado la línea.
- El archivo en el que se ha introducido o modificado la línea. En este ejemplo, el archivo *README.rst* se llamaba probablemente *README* en el commit.
- El dueño del commit que ha introducido o modificado la línea.
- La fecha del commit.

El comando `git blame` tiene varios argumentos prácticos que permiten, en particular, no tener en cuenta las líneas blancas o tener en cuenta las copias de archivos.

Buscar los commits con el modo pick axe

la opción pick axe permitirá saber en qué commits el contenido especificado se ha añadido o eliminado.

El modo pick axe puede ser utilizado con `git log`, y también con `git diff` o con el comando `git format-patch`. Para utilizar esta funcionalidad, se debe añadir el argumento `-S` seguido de la palabra o de la cadena que se ha de buscar:

`$git log -S "cadena a buscar"`
`$git log -S "Web framework that"` -> buscar los commits del proyecto que agreguen o eliminen la cadena «Web framework that»
`$git log -G "Web framework.{0, 20}that"` -> buscar los commits que añadan o eliminen las líneas que corresponden a la expresión regular «Web framework.{0, 20}that». La opción `-G`, que permite buscar los commits a partir de sus modificaciones en función de una expresión regular

Eliminar los cambios en el directorio de trabajo y/o el indice

Es frecuente que, durante una prueba o un nuevo desarrollo, el desarrollador desee volver al estado de la última versión del repositorio (apuntado por HEAD).

`$git reset --hard HEAD` -> elimina todas las modificaciones del directorio de trabajo y del índice. Es irreversible; por tanto, debe utilizarse con mucha prudencia.

`$git reset HEAD` -> Sin el argumento `--hard`, no modificará el directorio de trabajo. Este comando solo pondrá el índice en el mismo estado que la versión más reciente del repositorio. ¡Las modificaciones presentes en el índice se eliminarán de manera irreversible!

Volver a un estado anterior

`$git reset hash_del_commit` -> volver al estado anterior utilizando el hash del commit en el que el desarrollador quiere situarse

volver al proyecto tal como estaba a finales del año 2014

```
$git log --before="2014-12-31" -1 -> devulve el hash del commit 013c2d8
$git reset 013c2d8
```

En efecto, el directorio de trabajo no ha sido modificado, contiene los cambios incluidos en el último commit donde se encontraba el desarrollador. Se puede modificar también el directorio de trabajo mediante el argumento `--hard`.

Modificar el último commit

Para poder modificar el último commit, es necesario en primer lugar haber cumplido un imperativo: el commit no debe haber sido enviado (push) a un repositorio remoto

crea una carpeta nueva, aquí crea un repositorio, y añade un commit con modificaciones en un archivo

```
$ mkdir commitamend
$ cd commitamend
$ git init
$ vi archivo.txt
$ git add archivo.txt
$ git commit -m "Commit archivo"

$ vi archivo.txt
$ git commit -m "Doc acerca de commit amend" --amend ->
```

sustituir el último commit por uno nuevo

```
$git los -2 -> verifica que el ultimo commit ha sido remplazado
```

Los tags (etiquetas)

Un tag es un alias (un nombre) definido por un desarrollador, cuya función es apuntar a un commit. Permite identificar fácilmente un commit. Los tags se utilizan para nombrar en determinados momentos el estado del repositorio.

Numeración de las versiones

El sistema SemVer (*Semantic Versioning*) es un sistema cuya filosofía es dar un número de versión que tenga sentido en el ciclo de vida del software. Este sistema es utilizado en proyectos de gran envergadura como Python o Django y está perfectamente adaptado a los proyectos

de pequeño tamaño.

El número de versión se construye a partir de tres números separados por puntos: **x.y.z**.

Significado de los tres números:

- La **x** marcará una versión mayor. Una versión mayor es una versión que entraña cambios importantes en el funcionamiento de la aplicación o que induce una incompatibilidad con una versión anterior.
- La **y** marcará una versión menor. Una versión menor es una versión que añade funcionalidades manteniendo la compatibilidad con el sistema anterior.
- La **z** señalará un parche. Un parche corresponde a correcciones de errores («bugs») sin añadir funcionalidad.

Así, podemos imaginar un extracto de la lista de cambios (*changelog*) de un software de conversión de vídeo:

- **1.8.5**: corrección de diversos errores en la conversión de vídeo HD.
- **1.9**: adición del soporte del formato Matroska.
- **2**: modificación del algoritmo de compresión del archivo de salida.

Las ramas («Branch»)

Una rama corresponde en realidad a una versión paralela del curso de desarrollo. Una rama puede servir para desarrollar nuevas funcionalidades o corregir posibles errores sin integrar estos cambios en la versión principal del software. Las ramas permiten segmentar diferentes versiones en curso de desarrollo.

Por defecto, cuando trabajamos con un repositorio de Git, se crea una rama *master*. Es en esta rama donde hay que realizar todas las operaciones.

Lista de las ramas existentes

```
$git branch
     * master -> solo existe la rama master y el asterisco
indica que es la rama actual
```

`$git branch -v ->` mostrar la lista de las ramas con el hash y el mensaje del último commit de cada rama

Creación de una rama

Antes de crear una rama, hay que buscar en primer lugar de qué rama debe derivar. Esta rama de origen será también la rama que recibirá los cambios de la nueva rama.

Por ejemplo, si un desarrollador necesita crear una nueva funcionalidad, se creará una rama especial para esta funcionalidad. Esta nueva rama debe derivar de la rama *dev*, es decir, de la rama destinada a recibir todas las nuevas funcionalidades que aún no están publicadas.

`$git check-ref-format --branch nombre_rama ->` verificar si un nombre de rama es válido

```
$git branch -v -> observamos las ramas
*  dev             2d874c2  Estadisticas  Comandos  ->  Tabla  de
estadísticas
  master      63a5e8f API Rest -> Fix -> Token expiración

$git branch cuadro_de_mando_pedidos
$git branch -v -> observamos las ramas

*  dev             2d874c2  Estadisticas  Comandos  ->  Tabla  de
estadísticas
  master      63a5e8f API Rest -> Fix -> Token expiración
   cuadro_de_mando_pedidos  2d874c2  Estadisticas  Comandos  ->
Tabla de
estadísticas
```

`Se ha creado la rama cuadro_de_mando_pedidos que depende de "Estadisticas Comandos -> Tabla de estadísticas", la rama dev, esto es así porque al crear la rama cuadro_de_mando_pedidos estábamos parados en la rama dev.` Esto significa que cada nuevo commit actualizará el puntero *dev* al nuevo commit, o dicho de otro modo, que cada nuevo commit se efectuará en la rama *dev*. Para que los nuevos commits se realicen en la rama *cuadro_de_mando_pedidos*, primero hay que situarse encima.

Posicionamiento en una rama

Las ramas representan el sistema más fácil de usar para conservar y hacer evolucionar varias versiones del mismo proyecto. Con mucha frecuencia, el desarrollador debe cambiar de rama por varias razones:

- Corregir un error prioritario en su tarea actual,
- Continuar trabajando en una nueva funcionalidad (por lo tanto, en otra rama),

- Fusionar dos ramas para actualizar la versión de desarrollo con una nueva funcionalidad, por ejemplo.

Cambiar de rama (o ubicarse en una rama) significa varias cosas:

- Se modifica el directorio de trabajo.
- El índice se mantiene intacto.
- La referencia HEAD se actualiza y corresponde al commit más reciente de la rama en la cual el desarrollador se ha ubicado. El próximo commit tendrá como commit padre el nuevo HEAD y se considera un commit de la rama en la cual el desarrollador se ha ubicado.

Antes de cambiar de rama, debemos asegurarnos de que no hay ningún cambio en curso en el directorio de trabajo y en el índice.
`$git status ->` verificar los cambios pendientes de la rama actual

`$git stash ->` cambiar de rama rápidamente sin tener tiempo de terminar el código en curso para hacer un commit. permite conservar cambios de lado para volver a un estado idéntico entre el directorio de trabajo y la rama HEAD. El desarrollador puede entonces volver a importar sus modificaciones fácilmente sin haber tenido que crear un commit sin terminar.

`$git checkout nombre_rama ->` permite ubicarse en otra rama

```
$git checkout cuadro_de_mando_pedidos
  dev
  master
* cuadro_de_mando_pedidos

Ahora el asterisco esta en la nueva rama
```

`$git checkout -b conexión_social ->` permite crear una nueva rama y ubicarse en ella directamente

Fusionar dos ramas

recuperación de las diferencias de una rama a otra

Una fusión viene a a integrar las modificaciones de una rama en otra. Por ejemplo, cuando un desarrollador utiliza `git pull` para recuperar los commits de la rama seguida de forma remota, después de haber descargado los commits en el servidor remoto, Git realiza una fusión entre la rama actual y la rama seguida de forma remota.

el desarrollador desea añadir en la rama *master* las modificaciones introducidas por la rama *añadir_tarifa*. Para esto, se ubicará en primer lugar en la rama *master* y luego utilizará el comando `git merge` para fusionar los cambios en la rama *añadir_tarifa* en la rama *master*

```
$git checkout master
$git merge añadir_tarifa
```

Git buscará entonces el último commit que las dos ramas tienen en común (el commit se denomina «ancestro común»), creará un nuevo commit y efectuará las modificaciones de la rama *añadir_tarifa*. Este commit es especial, ya que hereda de dos padres: es el commit de merge. Estos pasos son totalmente transparentes para el desarrollador.

Limpiar su repositorio

tras la fusión de una rama con la rama *master*, la rama fusionada ya no tiene utilidad (a menos que otros cambios no estén previstos en esta) y es necesario eliminarla.

Solo hay por lo general dos casos que requieren la eliminación de una rama:

- La rama se fusionó en *master* y ninguna otra modificación está prevista en esta rama.
- Los cambios en esta rama ya no están al día y nunca serán integrados.

`$git branch -d nombre_rama ->` elimina realmente el puntero de la rama

¿Qué es un repositorio remoto?

Un repositorio remoto es un repositorio que servirá para centralizar un repositorio. Permite centralizar el trabajo de cada desarrollador.

Crear un repositorio remoto

`$git init new_browser ->` crear un nuevo **repositorio** llamado *new_browser*
`$git init --bare new_browser ->` crear un nuevo **repositorio remoto** llamado *new_browser*

```
$mkdir test_echange
```

```
$cd test_echange
$git init --bare remoto
```

Cuando la interfaz de línea de comandos se sitúa en el repositorio *remoto*, el comando `git log` mostrará la siguiente salida:

```
fatal: bad default revision 'HEAD'
```

Esta salida indica que el repositorio está vacío y que Git no encuentra referencia HEAD, lo que es normal porque HEAD no tiene ningún commit al que referenciar.

- Clonado en dos lugares diferentes del repositorio remoto:

```
git clone remote local1
git clone remote local2
```

Para cada uno de estos comandos, Git indica que el repositorio clonado está vacío con la siguiente salida:

```
Cloning into 'local1'...
warning: You appear to have cloned an empty repository.
done.
```

- Commit de una modificación de un repositorio local (aquí *local1*):

```
vi README
git add README
git commit –m "README : initial"
```

- Enviar el commit al repositorio remoto usando el comando siguiente:

```
git push
```

- Listar los commits en el repositorio remoto con `git log`:

```
commit cc6bb2a
Author: Samuel DAUZON <git@dauzon.com>
Date:   Sat Aug 22 02:28:21 2015 +0200

    README : initial
```

Esta salida indica que el commit presente en el repositorio *local1* se ha enviado al repositorio *remoto*.

Existen varios métodos por defecto para el envío de los cambios:

- **matching**: este método se activa por defecto en las instalaciones de Git 1.x. Envía todas las ramas locales que corresponden a las ramas remotas

- **simple**: este método se activa por defecto en las instalaciones de Git 2.X. Envían solo la rama actual si su nombre se corresponde con el de la rama remota seguida.
- **nothing**: este método no enviará nada.
- **current**: este método enviará la rama actual a la rama remota seguida del mismo nombre.
- **upstream**: este método enviará la rama actual a su rama remota seguida, sea cual fuere el nombre de las ramas.

Estos métodos deben configurarse usando la sintaxis siguiente:

```
git config --global push.default nombre_método
```

Cuando el desarrollador debe enviar una rama desconocida de un repositorio remoto, en concreto debe enviarlo con la siguiente sintaxis:

```
git push --set-upstream origin nombre_rama
```

Los protocolos de intercambio

Existen cuatro protocolos para intercambiar información entre dos repositorios:

- Local: empleando el sistema de archivos.
- SSH: utilizando un acceso SSH al equipo que contiene el repositorio remoto.
- HTTP: utilizando el protocolo HTTP a través de un servidor HTTP.
- Git: empleando el protocolo Git diseñado para un buen rendimiento.

Protocolos	Ventajas	Inconvenientes
Local	• Sencillez de implementación. • Mismas restricciones de acceso que las del sistema de archivos.	• Creación de un acceso vía Internet complicada.
SSH	• Protocolo popular. • Protocolo securizado mediante autenticación, restricciones y cifrado.	• Requiere algunos conocimientos de administración del sistema para la creación y gestión de derechos.
HTTP	• El acceso es sencillo a través de este protocolo universal. • Implementación simple a través de un servidor HTTP.	• Protocolo lento. • Sin autenticación ni restricciones. • Tráfico de red no cifrado (excepto en HTTPS).
Git	• Protocolo muy rápido	• Sin autenticación ni restricciones. • Requiere conocimientos de administración del sistema. • Tráfico de red no cifrado.

`$git clone ~/Repositorios/Servidor/CMS.git` -> clonar un repositorio a través del sistema de archivos
`$git clone ssh://git.entreprise@servidor_ssh:CMS.git` -> clonar un repositorio a través de ssh
`$git clone http://git-conflict.com/repositorios/CMS.git` -> clonar un repositorio a través de http
`$git clone git://git-conflict.com/repositorios/CMS.git` -> clonar un repositorio a través de git

Funcionamiento interno y ramas remotas

El uso de ramas locales y remotas es sencillo gracias a los comandos `git pull` y `git push`. Estos comandos ocultan el sistema interno de Git que permite gestionar estas ramas sin dificultad.

Un repositorio local puede vincularse con varios repositorios remotos.

`$git remote` -> tener la lista de los repositorios remotos vinculados con el repositorio local

Enviar sus modificaciones

Hay que saber que, si varios desarrolladores trabajan en el mismo servidor remoto, si otro desarrollador envió modificaciones (mediante uno o varios commits) en el servidor y nadie los ha recuperado, entonces nadie podrá enviar sus modificaciones antes de haber recuperado aquellas presentes en el servidor.

Aquí, el remote *origin* es en realidad un acceso directo a un repositorio remoto. Si el repositorio ha sido clonado a partir de la URL *http://git-conflict.com/repositorios/CMS.git*, entonces *origin* apuntará siempre a esta URL.

`$git push origin master` -> enviar al remote llamado *origin* las modificaciones añadidas en la rama *master*

Recibir las modificaciones

`$git pull` -> actualizar la rama actual con los commits contenidos en el repositorio remoto

> En realidad, el comando `git pull` es un acceso directo de dos comandos Git ejecutados sucesivamente:

`$git fetch` -> descargará los commits contenidos en el repositorio remoto para la rama específica. Los commits se integrarán en la rama remota seguida.
`$git merge FETCH_HEAD` -> merge de las modificaciones contenidas en la rama remota seguida para integrarlas en la rama local

`$git branch --all` -> obtener la lista de todas las ramas (locales y remotas seguidas)

Un sistema de gestión de ramas Git-Flow

Vincent Driessen propuso un sistema eficaz de gestión de ramas en su blog profesional (http://nvie.com/posts/a-successful-git-branching-model). Este sistema de gestión de ramas está destinado a utilizarse desde equipos de pequeño tamaño hasta equipos de gran envergadura. El objetivo de este sistema es separar eficazmente las ramas y las diferentes versiones del proyecto. Este método de trabajo está muy difundido en los proyectos que utilizan Git

Con el método Git-Flow, todos los merges deben efectuarse desactivando el avance rápido con el argumento `--no-ff` del comando `git merge` para que cada merge produzca un commit. Esto permite tener un histórico más claro y no contaminado por avances rápidos.

1. Las ramas eternas

Las ramas *master* y *develop* son ramas que nunca serán suprimidas durante toda la vida útil del proyecto. Son las únicas ramas que existen al inicio del proyecto y al final del proyecto.

a. La rama de producción (master)

Esta es la rama que contendrá todas las versiones publicadas por los usuarios. Es decir, que es la rama que recibirá todas las nuevas funcionalidades y todas las correcciones de errores («bugs»). Cada commit de esta rama está representado por una nueva versión definida por un tag. Todos los commits de esta rama son commits producidos por las fusiones de otras ramas (a excepción del commit de origen).

Estas son las ramas que pueden crearse a partir de *master*:

- la rama *develop* (que solo se creará una vez al inicio del proyecto),
- las ramas de parches (*hotfix*).

Estas son las ramas donde *master* podrá recibir las modificaciones a través de un merge:

- las ramas de nuevas versiones (*release*),
- las ramas de parches (*hotfix*).

b. La rama de desarrollo (develop)

La rama de desarrollo corresponde a la rama que recibirá todas las nuevas funcionalidades que aún no están integradas en la versión principal. Esta rama corresponde a las futuras versiones que se publicarán y que no se consideran estables todavía. Esta rama se crea desde el comienzo del proyecto a partir de la rama *master*.

Estas son las ramas que pueden crearse a partir de *develop*:

- las ramas de funcionalidades (*feature*),
- las ramas de versiones (*release*).

Estas son las ramas a partir de las cuales *develop* podrá recibir las modificaciones:

- las ramas de parches (*hotfix*),
- las ramas de versiones (*release*),
- las ramas de funcionalidades (*feature*).

2. Las ramas efímeras

Las ramas efímeras son ramas que tienen una vida limitada. Se crean con un objetivo muy concreto y, una vez conseguido, estas ramas se eliminan.

a. Las ramas de versiones (release)

Estas ramas son aquellas de versiones Beta del proyecto. Tomemos el ejemplo de un software cuya versión 2.0 está prevista próximamente. La empresa editora decide un mes antes de la publicación de esta versión crear una rama *release* para preparar esta salida. Antes de crear esta rama, la empresa debe asegurarse de que todas las nuevas funcionalidades esperadas para la versión 2.0 se han integrado en la rama *develop*. Una vez efectuada esta comprobación, la rama *release-2.0* puede crearse a partir de la rama *develop*. Esta rama tendrá dos propósitos principales:

- Servirá para crear la próxima versión estable. Es decir, que la rama *release-2.0* se pondrá a prueba al máximo y que los parches específicos de las nuevas funcionalidades se incluirán en esta rama.

- Esta rama, permitirá a los desarrolladores comenzar a trabajar sobre las características de la versión 3.0, mientras que la versión 2.0 está en fase de prueba. Ninguna funcionalidad propia de la versión 3.0 deberá encontrarse en la rama *release-2.0*.

Esta rama no sirve para recibir nuevas funcionalidades. Debe crearse a partir de *develop*, que debe contener todas las nuevas funcionalidades de la versión 2.0.

Esta rama se fusionará en las ramas *develop* y *master*. El commit en la rama *master* dará lugar a un commit en el cual se aplicará el tag v2.0.

Esta rama se borrará después de haber sido fusionada

en *master* y *develop*.

b. Las ramas de parches (hotfix)

Estas ramas son las que van a incluir los parches destinados a producción. La creación de una rama de parches se produce cuando se descubre un bug y el desarrollador comienza a repararlo. El commit o los commits de esta rama son solo los commits destinados a la resolución del bug (y la actualización de los números de versión si es necesario).

Una rama de parche se crea a partir de la rama *master* y, una vez hecho el commit del elemento corrector, esta rama se fusiona en *master* y *develop* y luego se elimina. Para respetar la norma de Git-Flow, se debe fijar con antelación los nombres de las ramas de parches por *hotfix*-.

c. Las ramas de funcionalidades (feature)

Cada una de las ramas de funcionalidad tiene el objetivo de incorporar una nueva funcionalidad a la rama *develop*. Estas ramas son locales, es decir, que no se comparten en el repositorio central. La creación de dicha rama siempre tendrá lugar a partir de la rama *develop* y, cuando la nueva funcionalidad se crea y se guarda con un commit, esta rama debe incorporarse a *develop*.

Cuando la funcionalidad se ha integrado en *develop*, la rama de la funcionalidad debe suprimirse.

Dejar de lado los cambios con git stash

`git stash` guardará todas las modificaciones a un lado. Estas no estarán incluidas en el directorio de trabajo ni el índice, no estarán tampoco presentes en el histórico, ya que no han pasado por el proceso de commit.

`$git stash --include-untracked ->` dejará de lado las modificaciones que realizó con el siguiente comando

El comando `git stash` puede utilizarse sin el argumento ––

`include-untracked`. Esto tendrá por efecto a dejar los archivos no seguidos en el directorio de trabajo.

Si las modificaciones se colocan en el índice, también estarán incluidas en el stash, salvo si el argumento `keep-index` se utiliza.

`$git stash list ->` listar los stashs de un repositorio

`$git stash apply stash@{0} ->` recuperar las modificaciones contenidas en un stash,

`$git stash drop stash@{0} ->` eliminar el stash una vez recuperado

`$git stash clear ->` eliminar todos los stash del repositorio

Repositorios integrados con submódulos

Enviar un repositorio local a Bitbucket

`$ git remote add origin https://jjolivares@bitbucket.org/jolivares/listacumple.git ->` permite introducir en la configuración del repositorio local el repositorio remoto al que se debe hacer referencia.
`$ git push -u origin --all ->` permite enviar los elementos del repositorio local al repositorio remoto. Al ejecutar el segundo comando, se solicita la contraseña
`$git push -u origin --tags ->` permite enviar los tags del repositorio

`$git pull -> actualiza repositorio local desde el remoto`

`$git diff ->` Revisa los cambios entre el directorio de trabajo y lo que se encuentra en la última versión del repositorio

vi `ejemplo.html`
`git add ejemplo.html`
`git commit -m "App de ejemplo“.`
`git log -1 ->` Para comprobar que el commit se ha añadido al repositorio

`$git checkout cumple.html ->` permite solicitar a Git restaurar el estado del archivo *cumple.html* tal como está en HEAD.

útil en caso de que se modifique un archivo y sin hacer commit se quiere volver a la versión de antes de empezar a editarlo

`$git add -A ->` permite añadir todos los archivos agregados/ modificados en el índice

`$git add --all --dry-run > lista_de_archivos.txt ->` creará un archivo *lista_de_archivos.txt* que contendrá toda la lista de archivos que se añadirían con un `git add --all`.

El uso de commits atómicos: los commits deben representar una modificación independiente de las otras. Con el comando `git add -A`, no se debe hacer commit de un conjunto de modificaciones donde varios commits serían más adecuados.

`$git commit -a -m "mensaje de commit" ->` todos los archivos modificados ya seguidos se añadirán directamente en el próximo commit. Aunque no estén agregados al índex

`$git push -> Enviar los commits al repositorio remoto`

`$git clone https://github.com/cmido/tfm.git TFM_UCM`