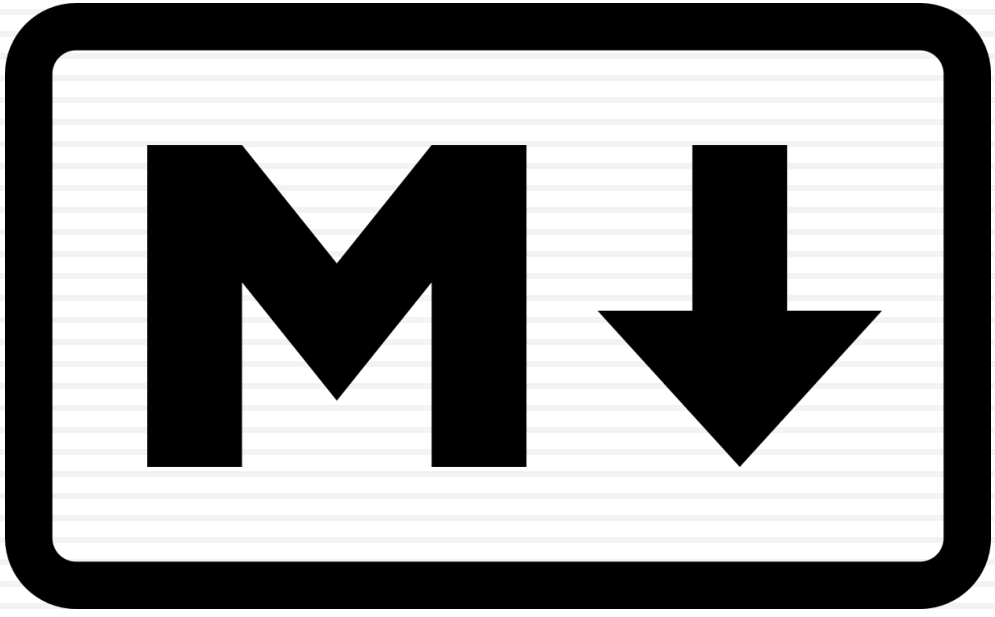
UT3 - DOCUMENTACIÓN Y CONTROL DE VERSIONES

Despliegue de Aplicaciones Web

2° DAW

Markdown



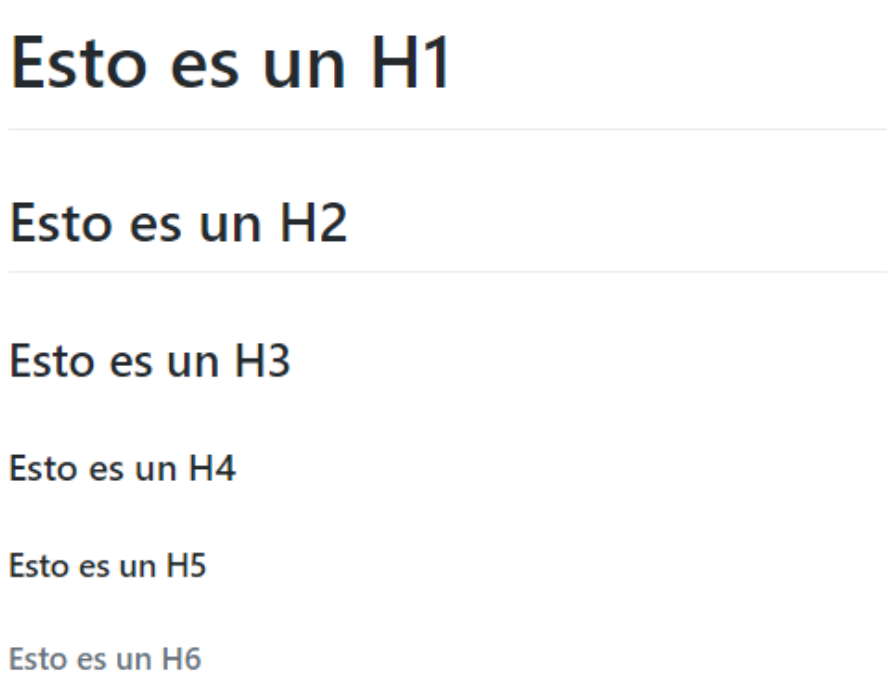
Markdown

- Markdown nació como herramienta de conversión de texto plano a HTML.
- Fue creada en 2004 por John Gruber
- Aunque en realidad Markdown también se considera un lenguaje que tiene la finalidad de permitir crear contenido de una manera sencilla de escribir, y que en todo momento mantenga un diseño legible.
- Es un lenguaje de marcado ligero parecido al que se emplea en muchas wikis y basado originalmente en convenciones existentes en el marcado de los correos electrónicos.
- Emplea texto plano, procurando que sea legible pero consiguiendo que se convierta en XHTML correctamente formateado.
- Comienza a ser muy popular entre **programadores**.

Sintaxis Markdown: cabeceras

- Los encabezamientos HTML se producen colocando un número determinado de almohadillas # antes del texto correspondiente al nivel de encabezamiento deseado (HTML ofrece hasta seis niveles). Los encabezamientos son los siguientes:

```
# Esto es un H1
## Esto es un H2
### Esto es un H3
#### Esto es un H4
##### Esto es un H5
###### Esto es un H6
```



Sintaxis Markdown: enlaces

- Podemos crear enlaces de distintos modos:

```
[Web de Apache](https://httpd.apache.org/ "ejemplo de enlace")
[Enlace 1, para que funcione hay que dar 1 intro][1] , [Enlace 2][2]

[1]: <https://httpd.apache.org/>
[2]: <https://httpd.apache.org/>
```

Web de Apache

Enlace 1, para que funcione hay que dar 1 intro , Enlace 2

Sintaxis Markdown: párrafos

- Para crear párrafos se deja una línea en blanco:

```
Este es el primer párrafo.

Este es el segundo párrafo.
```

Este es el primer párrafo.

Este es el segundo párrafo.

- Para crear un salto de línea dentro de un párrafo, simplemente se dejan dos espacios al final de la última palabra de esa línea:

```
Este es el primer párrafo.
Este es el segundo párrafo.
```

Este es el primer párrafo.
Este es el segundo párrafo.

Sintaxis Markdown: formato

- El formato básico del texto, es decir negritas y cursiva, se pueden realizar de varias maneras:

```
**Esto es negrita**
__Esto también es negrita__
*Esto es cursiva*
_Esto también es cursiva_
***Esto es negrita y cursiva***
___Esto también es negrita y cursiva___
```

Esto es negrita
Esto también es negrita
Esto es cursiva
Esto también es cursiva
Esto es negrita y cursiva
Esto también es negrita y cursiva

Sintaxis Markdown: citas

- Para crear bloques de cita, se emplea el carácter mayor que > antes del bloque de texto:

```
Esto es una línea normal

>Esto es parte de un bloque de cita.
>Esto es parte del mismo bloque de cita.
```

Esto es una línea normal

> Esto es parte de un bloque de cita.
> Esto es parte del mismo bloque de cita.

```
> Esto es parte de un bloque de cita.
Esto continúa el bloque incluso aunque no hay símbolo 'mayor que'.

La línea en blanco finaliza el bloque.
```

> Esto es parte de un bloque de cita.
> Esto continúa el bloque incluso aunque no hay símbolo 'mayor que'.

La línea en blanco finaliza el bloque.

Sintaxis Markdown: listas

- Markdown permite crear dos tipos de listas, ordenadas y desordenadas, es decir numeradas o listas de puntos:

```
Lista numerada (ordenada)

1. Este es el primer elemento
2. Este es el segundo elemento
3. Este es el tercer elemento
```

Lista numerada (ordenada)

1. Este es el primer elemento
2. Este es el segundo elemento
3. Este es el tercer elemento

```
Lista de puntos (desordenada)

* Un elemento de la lista
* Otro elemento de la lista
* El tercer elemento de la lista
```

Lista de puntos (desordenada)

- Un elemento de la lista
- Otro elemento de la lista
- El tercer elemento de la lista

Sintaxis Markdown: listas


```
Se pueden mezclar distintos tipos de listas y anidar unas dentro de otras.

1. Esto es una lista ordenada
2. Segundo elemento de la lista ordenada
    1. Esta es una lista ordenada anidada dentro de otra
        * Lista desordenada anidada a tercer nivel
        * Segundo elemento de esta lista
    2. Este es el segundo elemento de la lista ordenada anidada
```

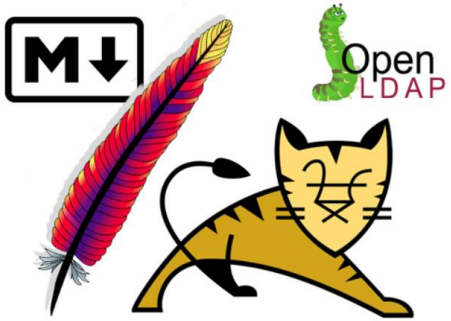
Se pueden mezclar distintos tipos de listas y anidar unas dentro de otras.

1. Esto es una lista ordenada
2. Segundo elemento de la lista ordenada
    i. Esta es una lista ordenada anidada dentro de otra
        - Lista desordenada anidada a tercer nivel
        - Segundo elemento de esta lista
    ii. Este es el segundo elemento de la lista ordenada anidada

Sintaxis Markdown: imágenes

- La manera de enlazar imágenes es básicamente la misma de crear enlaces, con un única diferencia, se añade el carácter exclamación **!** al principio de la pareja de corchetes que definen el nombre del enlace.

```
![DAW con título](imagenes/daw.png "título")
```




```
![Imagen 1][1]  ![Imagen 2][2]

[1]: imagenes/DAW.png
[2]: imagenes/markdown.png
```

Sintaxis Markdown: tablas

- Crear tablas es muy sencillo, simplemente debemos indicar cuáles son los elementos de la cabecera y separar los campos con el símbolo |

```
Cabecera A | Cabecera B
-- | --
Campo A0 | Campo B0
Campo A1 | Campo B1
```

Cabecera A	Cabecera B
Campo A0	Campo B0
Campo A1	Campo B1

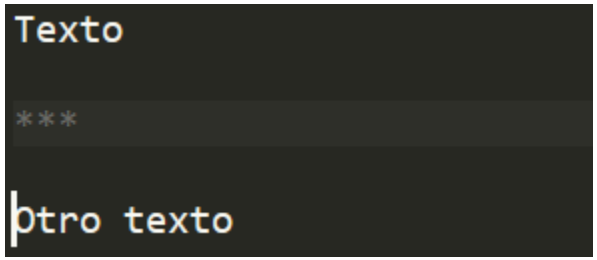
- Se puede especificar la alineación de cada columna mediante la adición de dos puntos a las líneas de separación. Dos puntos a la izquierda de la línea de separación hará que la columna esté alineada a la izquierda, dos puntos a la derecha de la línea hará que la columna esté alineada a la derecha, dos puntos en ambos lados significa que la columna se alinea al centro

```
| Izquierda | Centrado | Derecha |
| :-------- | :------: | ------: |
| Item 1    | 15       | 150€    |
| Item 2    | 3250     | 23,65€  |
```

Izquierda	Centrado	Derecha
Item 1	15	150€
Item 2	3250	23,65€

Sintaxis Markdown: líneas horizontales

- Para crear líneas horizontales se debe crear una línea rodeada de líneas en blanco y compuesta por 3 o más símbolos, que pueden ser guiones, asteriscos o guiones bajos. Pueden crearse espacios entre estos caracteres si así se desea por estética.

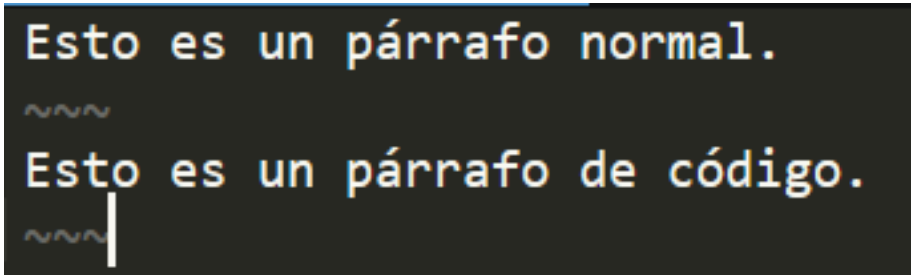


Texto

Otro texto

Sintaxis Markdown: código

- Se pueden crear bloques de código para albergar extractos de código fuente de un lenguaje de programación o para reproducir literalmente cualquier tipo de texto sin que sea interpretado por markdown. Para ello hay que encerrarlo entre dos líneas formadas por tres o más caracteres tilde ~



Esto es un párrafo normal.

```
Esto es un párrafo de código.
```

Sintaxis Markdown: código

- Pero es mucho más interesante resaltar la sintaxis de un determinado lenguaje de programación. Para ello hay que especificar el lenguaje que está utilizando:

````
Esto es un párrafo normal.
```java
public class Hoja01_Java_02_1 {

    public static void main(String[] args)
    {
        int x = 100;
        x++;
        int y = 20;
        y--;
        x = x + y;
        System.out.println("El valor de la variable x es " + x);
    }
}
```


```php
<?php
    $var1=5;
    $var2=10;
    $var3=12;
    $var_resul=($var1+$var2+$var3)/3;
    print("El resultado es $var_resul");
?>
```
````

Esto es un párrafo normal.

```java
public class Hoja01_Java_02_1 {

    public static void main(String[] args)
    {
        int x = 100;
        x++;
        int y = 20;
        y--;
        x = x + y;
        System.out.println("El valor de la variable x es " + x);
    }
}
```

```php
<?php
    $var1=5;
    $var2=10;
    $var3=12;
    $var_resul=($var1+$var2+$var3)/3;
    print("El resultado es $var_resul");
?>
```

Hoja03_Markdown_01

Control de versiones

Control de versiones

- **Definición:** Gestión de los cambios en documentos, programas, páginas web o cualquier otra colección de información.
- Sistema de Control de Versiones. En castellano se suelen usar las siglas SCV y en inglés VCS (Version Control System):
  - Son aplicaciones para el control de versiones.
  - *Independientes* (con su propio interfaz) o *integradas* (las gestionamos a través de otro programa).
  - *Locales* (en el equipo en el que trabajamos) o *distribuidas* (en remoto, ya sea en red local o en la nube).

RAZONES PARA USAR SCV EN DESARROLLO SOFTWARE:

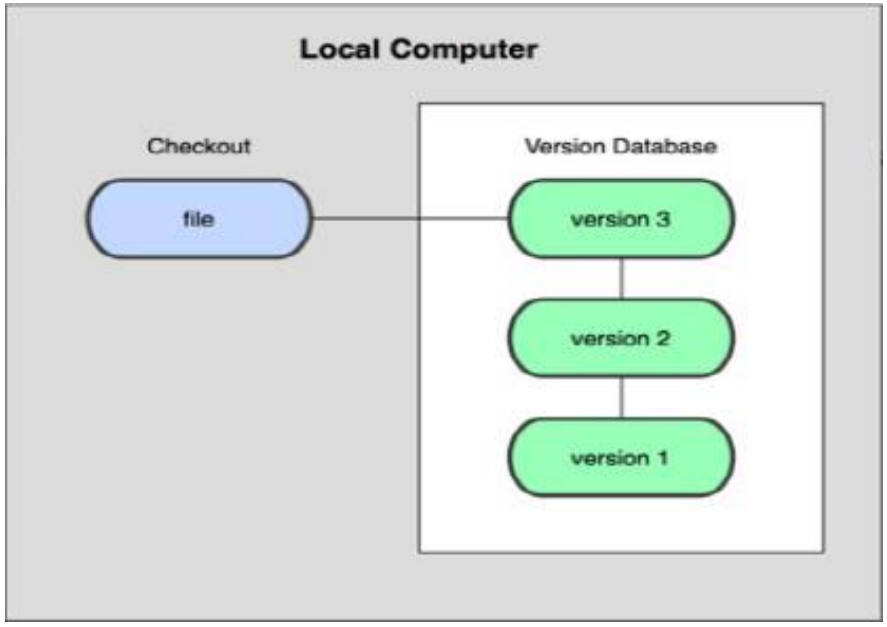
- Cuando el desarrollo lo realiza una sola persona:
  - Posibilidad de deshacer cambios grandes que no han ido como se esperaba.
  - Posibilidad de hacer varias versiones con distintas funcionalidades pero una base común.
  - Posibilidad de hacer pruebas con el código sin riesgo.
- Cuando el desarrollo lo realiza un equipo:
  - Fusionar las versiones de varios en una sola.
  - Gestionar (trazar) los cambios realizados por cada uno.

Clasificación de los SCV

- Podemos clasificar los sistemas de control de versiones según la arquitectura para almacenar la información en **locales, centralizados** o **distribuidos.**

Locales

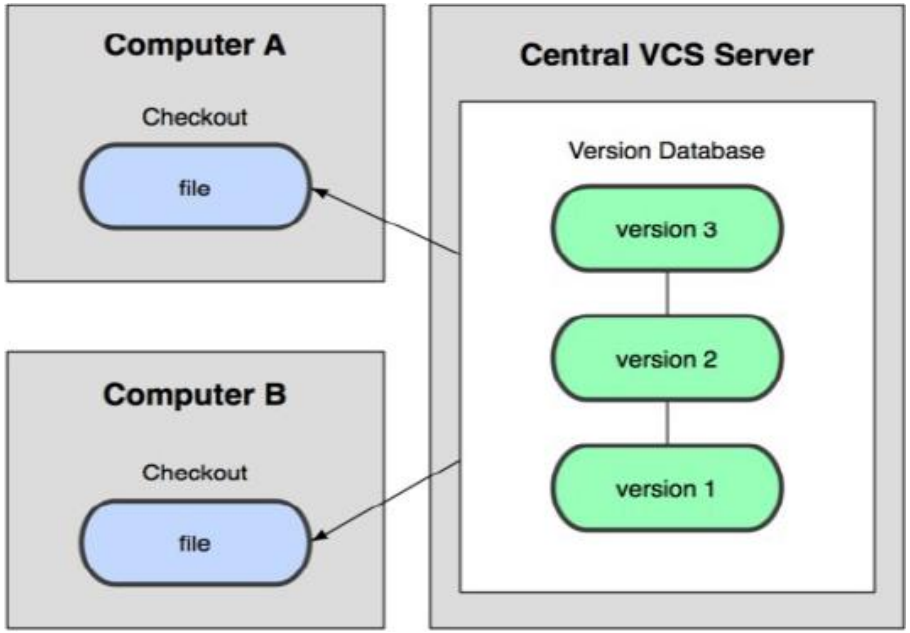
- La información se guarda en un ordenador o repositorio local con lo que no sirve para trabajar en forma colaborativa.
- Ejemplo: RCS (Revision Control System).



Clasificación de los SCV

Centralizados o CVCS (Centralized Version Control System)

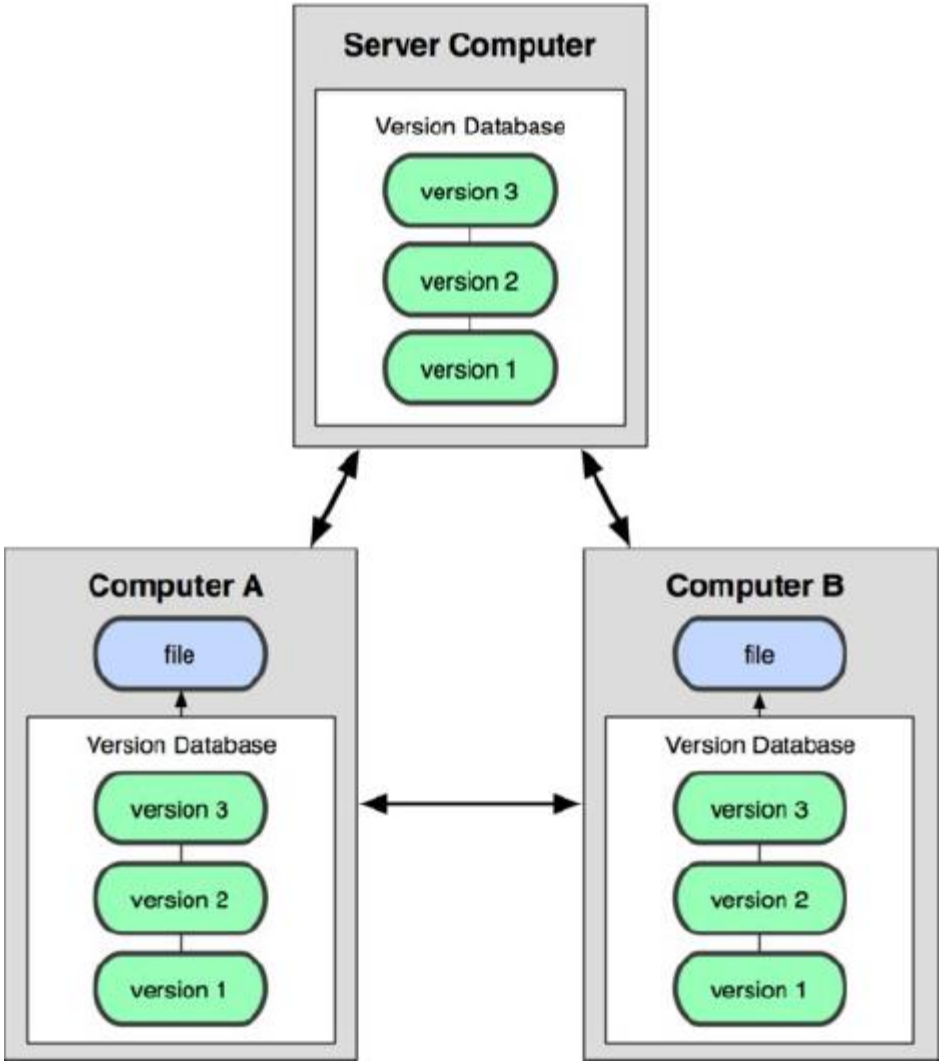
- La información se guarda en un servidor dentro de un repositorio centralizado.
- Existe un usuario o usuarios responsables con capacidad de realizar tareas administrativas a cambio de reducir flexibilidad, necesitan la aprobación del responsable para realizar acciones, como crear una rama nueva.
- Ejemplos: *Subversion* y CVS.



Clasificación de los SCV

Distribuidos o DVCS (Distributed Version Control System)

- Cada usuario tiene su propio repositorio.
- Los distintos repositorios pueden intercambiar y mezclar revisiones entre ellos.
- Es frecuente el uso de un repositorio, que está normalmente disponible, que sirve de punto de sincronización de los distintos repositorios locales.
- Ejemplos: *Git* y Mercurial



Git

- Git es un sistema de control de versiones distribuido y gratuito
- Fue diseñado por **Linus Torvalds** pensando en la eficiencia y confiabilidad del mantenimiento de versiones de aplicaciones cuando éstas tienen un gran número de archivos
- Basado en línea de órdenes… pero con clientes para quién no les guste ☺

GitHub

- Herramienta **web** para gestionar nuestros repositorios.
- Gratis si los repositorios son abiertos.
- De pago si queremos tener repositorios privados y múltiples colaboradores...
- Wiki para cada proyecto y página web para cada proyecto
- Gráfico para ver cómo los desarrolladores trabajan en sus repositorios y bifurcaciones del proyecto
- Funcionalidades como si se tratase de una red social, como por ejemplo: seguidores
- Herramienta para **trabajo colaborativo** entre programadores.

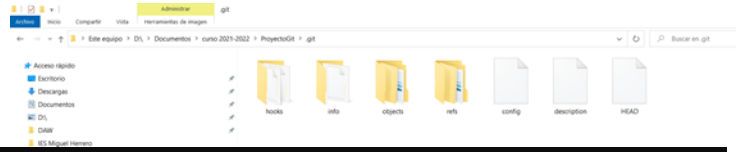
Otras alternativas: GitLab

- ¿Si queremos algo sistema similar en nuestro propio servidor, orientado a organizaciones que dependen del almacenamiento de sus proyectos?
- ¿Y si queremos que nuestros proyectos sean privados inicialmente, sin tener que pagar por la cuenta mínima del servicio?
- **GitLab** es un proyecto de **código libre** que se puede **instalar en nuestro servidor** y que nos permite tener repositorios privados sin coste alguno
- Además podremos utilizar sus servidores al igual que en GitHub permitiéndonos tener repositorios **ilimitados privados sin coste.**

Comandos básicos

- **Configuración**
  - Usado para establecer una configuración específica de usuario, como sería el caso del email, nombre de usuario y tipo de formato, etc…

```
D:\Documentos\curso 2021-2022\ProyectoGit>git config
usage: git config [<options>]
```



```
D:\Documentos\curso 2021-2022\ProyectoGit>git config --global user.email nsalayag01@educantabria.es
```

```
D:\Documentos\curso 2021-2022\ProyectoGit>git config --global user.name "Nieves Salaya"
```

- **Crear un repositorio nuevo**
  - Si queremos iniciar el seguimiento en Git de un proyecto existente, vamos al directorio del proyecto y escribimos:
  - **git init**
  - Esto crea un nuevo subdirectorio llamado .git que contiene todos los archivos necesarios del repositorio —un esqueleto de un repositorio Git. IMPORTANTE: Todavía no hay nada en el proyecto que esté bajo seguimiento.

```
D:\Documentos\curso 2021-2022\ProyectoGit>git init
Reinitialized existing Git repository in D:/Documentos/curso 2021-2022/ProyectoGit/.git/
```

Comandos básicos

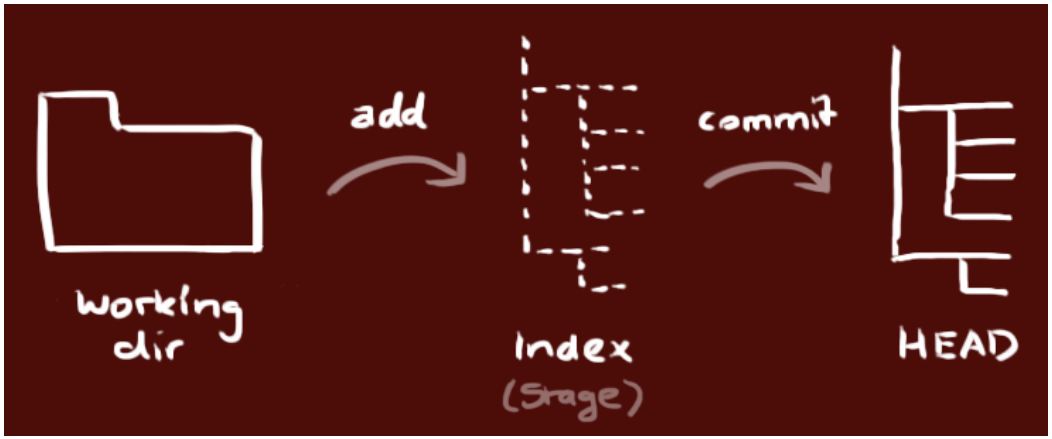
- **Clonar un repositorio**
  - **git clone /path/to/repository**
  - **git clone username@host:/path/to/repository** (en caso de usar un repositorio remoto)
  - Se obtienen los ficheros de trabajo (código) y toda la información de control de versiones (historia)
  - Para obtener una copia de un repositorio Git existente (por ejemplo, un proyecto en el que nos gustaría contribuir)
  - Al contrario que con git init, con git clone no es necesario crear un directorio para el proyecto. Al clonar se creará un directorio con el nombre del proyecto dentro del que te encuentres al llamar a la orden.

```
D:\Documentos\curso 2021-2022\ProyectoGit>git  clone https://github.com/nieveslourdes/Nieves_Markdown.git
Cloning into 'Nieves_Markdown'...
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (3/3), done.

D:\Documentos\curso 2021-2022\ProyectoGit>
```

Flujo de trabajo en git

- Tu repositorio local esta compuesto por tres "árboles" administrados por git.
  - El primero es tu **Directorio de trabajo** que contiene los archivos, los modificas, los borras, creas nuevos, etc.
    - Git sabe que tiene que controlar ese directorio, pero no lo hace hasta que se lo digas expresamente
  - El segundo es el **Index** que actúa como una zona intermedia
    - Se preparan los archivos que le indiques poniéndolos en una especie de lista virtual a la que llamamos el "Index". En Index ponemos los archivos que hemos ido modificando, pero las cosas que están en el "Index" aun no han sido archivadas por git.
  - El último es el **HEAD** que apunta al último commit realizado



Manteniendo nuestro repositorio al día (add)

- Cuando tenemos el repositorio iniciado (o clonado) empezamos a trabajar creando ficheros, editándolos, modificándolos, etc.
- Para que git sepa que tiene que empezar a tener en cuenta un archivo (a esto se le llama preparar un archivo), usamos la orden git add de este modo:
  - **git add** NOMBRE_DEL_ARCHIVO
- Esto, como vimos antes, añadirá el archivo indicado al Index. No lo archivará realmente en el sistema de control de versiones ni hará nada. Solo le informa de que debe tener en cuenta ese archivo para futuras instrucciones
- Una función interesante es **git status**
  - Da un resumen de cómo están las cosas ahora mismo respecto a la versión del repositorio (concretamente, respecto al HEAD): qué archivos se han modificado, qué hay en el Index, etc.
  - Cada vez que no tengas muy claro qué has cambiado y qué no, consulta git status.

```
D:\Documentos\curso 2021-2022\ProyectoGit\Nieves_Markdown>git status
On branch main
Your branch is up to date with 'origin/main'.
```

Manteniendo nuestro repositorio al día (add)


```
D:\Documentos\curso 2021-2022\ProyectoGit\Nieves_Markdown>git  add readme.md
```

```
D:\Documentos\curso 2021-2022\ProyectoGit\Nieves_Markdown>git status
On branch main
Your branch is up to date with 'origin/main'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   README.md

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        imagenes/

no changes added to commit (use "git add" and/or "git commit -a")
```

Manteniendo nuestro repositorio al día (commit)

- Cuando se han hecho los cambios necesarios y se ha puesto en el Index todo lo que se quiera poner bajo el control de versiones, llega el momento de "hacer commit".
- Esto significa mandar al HEAD los cambios que tenemos en el Index
- **Para hacer commit** a estos cambios:
  - **git commit -m "Mensaje del commit"**
- También se podrán confirmar todos los cambios que haya en el directorio de trabajo aunque no hayan sido preparados (es decir, aunque no se haya hecho add)
  - **git commit -a**
  - Esto incluye tanto los ficheros modificados como los eliminados, con lo que sería equivalente a hacer un git add -A seguido de un git commit.

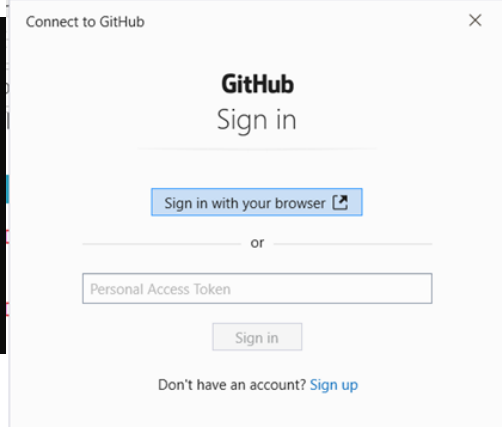
```
D:\Documentos\curso 2021-2022\ProyectoGit\Nieves_Markdown>git commit -m "Primer commit de Nieves Salaya"
[main 1236bf1] Primer commit de Nieves Salaya
 1 file changed, 90 insertions(+), 1 deletion(-)
 rewrite README.md (100%)
```

Envío de cambios (push)

- Ahora el archivo esta incluido en el **HEAD**, pero aún no en el repositorio remoto, sólo tenemos una copia el local.
- **Para enviar estos cambios al repositorio remoto:**
  - **git push origin main**
  - *master* es la rama del repositorio donde vamos a hacer los cambios. Habrá que reemplazar *master* por la rama a la que queramos enviar los cambios.
  - *origin* es el repositorio remoto

```
D:\Documentos\curso 2021-2022\ProyectoGit\Nieves_Markdown>git remote -v
origin  https://github.com/nieveslourdes/Nieves_Markdown.git (fetch)
origin  https://github.com/nieveslourdes/Nieves_Markdown.git (push)
```

```
D:\Documentos\curso 2021-2022\ProyectoGit\Nieves_Markdown>git push
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 8 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 1.16 KiB | 1.16 MiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/nieveslourdes/Nieves_Markdown.git
   8d2582c..1236bf1  main -> main
```



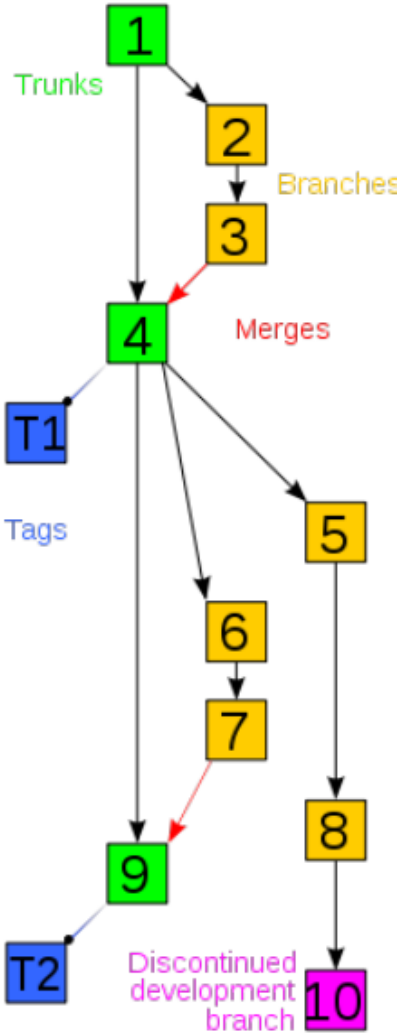
Envío de cambios (pull)

- Para actualizar el repositorio local al commit más nuevo:
  - **git pull**
  - Hay que realizarlo en el directorio de trabajo para bajar y fusionar los cambios remotos

```
D:\Documentos\curso 2021-2022\ProyectoGit\Nieves_Markdown>git pull
Already up to date.
```

Cierre de versión (creación de tag)

- Puede ser conveniente cerrar versiones de un proyecto. Se sigue desarrollando aunque la primera versión ya está en producción.
- A veces hay que volver atrás a resolver un bug de una versión ya acabada y seguir desarrollando el resto de versiones.
- El cierre de versión se denomina crear un Tag de la versión desarrollada. Esto implica **llevar una copia de la versión a cerrar a la rama de gestión de versiones**.
- **No** es costoso hacer Tags de un proyecto, así que se pueden hacer cada vez que se llegue a un objetivo del proyecto (hito).
- OJO: no modificar nunca un Tag, porque, aunque Subversion lo permite, se estaría perdiendo la referencia a la versión que en su momento se decidió congelar.
- Se debe utilizar el Master o bien un Branch para la evolución de la siguiente versión.



Etiquetas (tag)

- Se recomienda crear etiquetas para cada nueva versión publicada de un software.
- Este concepto no es nuevo, ya que estaba disponible en SVN.
- Puedes crear una nueva etiqueta llamada 1.0.0 ejecutando
  - **git tag v0.1 1b2e1d63ff**
  - 1b2e1d63ff se refiere a los 10 caracteres del commit id al cual quieres referirte con tu etiqueta.
- Puedes obtener el commit id con:
  - **git log**

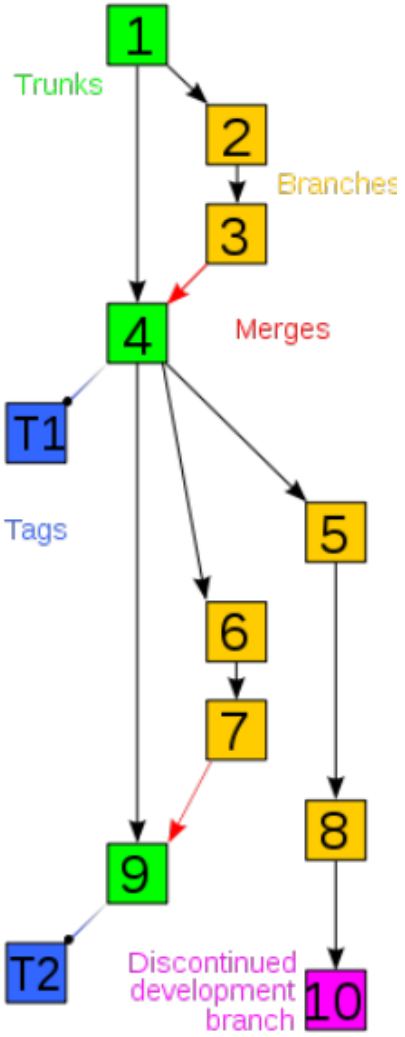
```
D:\Documentos\curso 2021-2022\ProyectoGit\Nieves_Markdown>git tag v0.1
```

```
(base) PS D:\Documentos\curso 2021-2022\ProyectoGit\Nieves_Markdown> git log
commit 7701bf009fa20735a3897c4c8ac8b707937848b8 (HEAD -> main, tag: v0.2)
Merge: 604d7e0 f608e3a
```

Hoja03_Markdown_02

Ramificación de código (branch)

- Existen situaciones en las que el ciclo de vida de un proyecto implica una evolución paralela de su código. Para ello se crean distintas ramas que al final se pueden fusionar todas ellas.
- Motivos para bifurcar:
  - Seguir evolucionando un software mientras se corrigen bugs de una versión puesta en producción. (Branch evolutivo y otro correctivo)
  - Muchas modificaciones que dejan al repositorio inestable, se crea un Branch para finalizar dichas modificaciones
  - Dos evoluciones de distinta naturaleza y no sea conveniente desarrollarlas de forma conjunta
- La ramificación no tiene mucho coste, pero no conviene abusar
- Mejor sólo cuando no haya alternativa.
- El problema es que crece de manera exponencial y es difícil la gestión de todas las versiones del proyecto.
- Los branches se consideran **ramas de vida limitada que deben terminar con un cierre de version (Tag)** o una fusión de cambios a la rama de la que se bifurco.



Ramas

- Las ramas (branch) son utilizadas para desarrollar funcionalidades aisladas unas de otras.
- La rama *main* es la rama "por defecto" cuando se crea un repositorio.
- Ya sabemos que se pueden crear nuevas ramas durante el desarrollo y fusionarlas a la rama principal cuando terminemos.
- Para crear una nueva rama hacemos lo siguiente:
  - **git branch rama-Nieves**
- Para crear una nueva rama llamada “nuevaRama" y cambiarte a ella:
  - **git checkout -b rama-Nieves**
- Para volver a la rama principal
  - **git checkout main**
- Para borrar la rama
  - **git branch -d rama-Nieves**
- Una rama nueva no estará disponible para los demás a menos que se suba (push) la rama al repositorio remoto
  - **git push origin rama-Nieves**

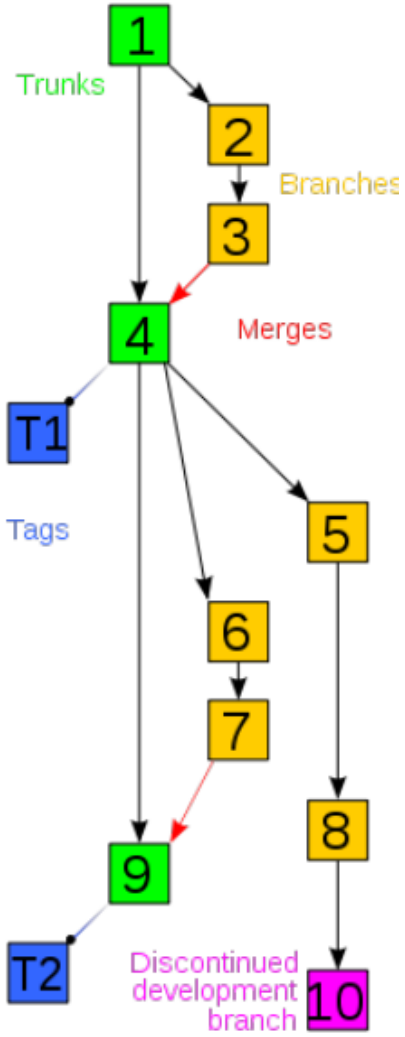
```
D:\Documentos\curso 2021-2022\ProyectoGit\Nieves_Markdown>git branch rama-Nieves
```

```
D:\Documentos\curso 2021-2022\ProyectoGit\Nieves_Markdown>git checkout rama-Nieves
Switched to branch 'rama-Nieves'
M       nieves.md
```

```
D:\Documentos\curso 2021-2022\ProyectoGit\Nieves_Markdown>git branch -d rama-Nieves
Deleted branch rama-Nieves (was f608e3a).
```

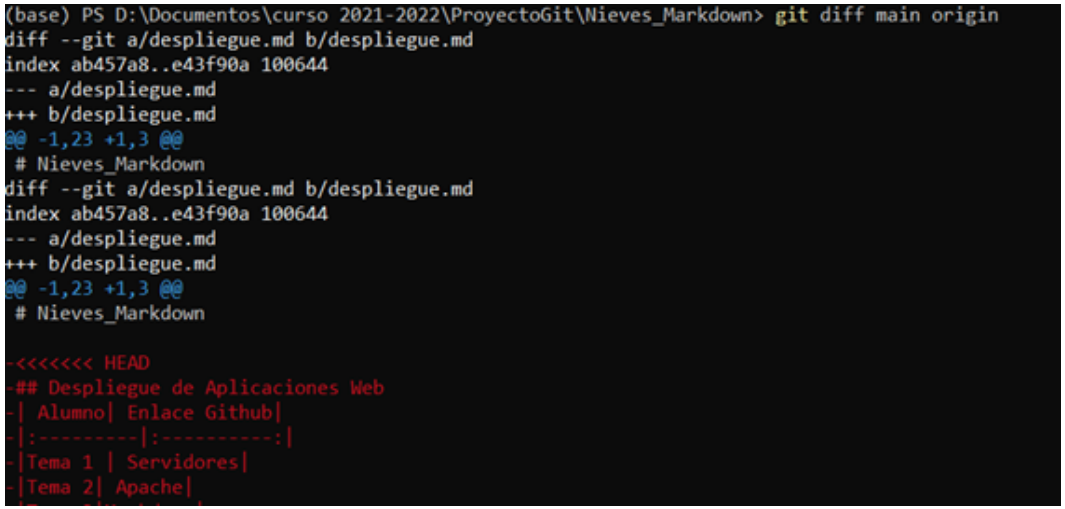
Fusión de cambios (merge)

- Se usa después de una ramificación para aplicarlo a algún desarrollo en paralelo.
- Se utiliza el comando **Merge**, que aplica los cambios producidos entre dos revisiones en una rama a otra rama.
- En caso de rama evolutiva y rama correctiva, se fusionan los cambios realizados en la correctiva con los surgidos simultáneamente en la evolutiva.
- Se realizará cuando se finalice un desarrollo paralelo.
- Mejor con Merge que intentar hacerlo a mano.



Actualizar y fusionar

- Para fusionar otra rama a la rama activa (por ejemplo master):
  - **git merge <nombreRama>**
- En ambos casos git intentará fusionar automáticamente los cambios. Desafortunadamente, no siempre será posible y se podrán producir **conflictos**.
- En última instancia habrá que evaluar manualmente estos conflictos editando los archivos mostrados por git.
- Después de modificarlos, habrá que marcarlos como fusionados con
  - **git add <filename>**
- Antes de fusionar los cambios podemos revisarlos usando:
  - **git diff <source_branch> <target_branch>**



Remplazar cambios locales

- En caso de hacer algo algo mal (lo que seguramente nunca suceda ☺) se puede reemplazar los cambios locales usando el comando
  - **git checkout -- <filename>**
- Este comando reemplaza los cambios en el directorio de trabajo con el último contenido de HEAD.
- Los cambios que ya han sido agregados al Index, así como también los nuevos archivos, se mantendrán sin cambio.
- Si lo que queremos es borrar los commits y dejar uno anterior podemos usar el comando:
  - git reset --hard COMMIT_ID

```
(base) PS D:\Documentos\curso 2021-2022\ProyectoGit\Nieves_Markdown> git log --oneline
27384d5 (tag: v0.1) añadido nieves.md
825bdfe añadido fichero .gitignore
1236bf1 Primer commit de Nieves Salaya
8d2582c Initial commit
(base) PS D:\Documentos\curso 2021-2022\ProyectoGit\Nieves_Markdown> git reset --hard 8d2582c
```