

# Documentación y control de versiones.

## Caso práctico

En la empresa **BK programación** están centrándose especialmente en el desarrollo de aplicaciones web, debido a que es el entorno más solicitado por los clientes; a medida que el número de proyectos que se incorporan aumenta surge la necesidad de automatizar y gestionar, mediante algún tipo de herramientas software, determinados aspectos relacionados con el desarrollo de aplicaciones web, como es el caso de la documentación de las aplicaciones y el establecimiento de un sistema de control de versiones.



La documentación de un proyecto de software es tan importante como su código. Una buena documentación nos facilita, en gran medida, el mantenimiento futuro de la aplicación. Si además estamos trabajando en equipo es muy útil saber lo que hacen las partes que desarrollan otras personas, sobre todo si tenemos que utilizarlas en nuestra parte.

**Juan** ha empezado a utilizar diversas herramientas que permiten generar documentación de forma automática a partir del código fuente. Javadoc es la herramienta estándar en Java. Para PHP una de las herramientas más utilizadas es *phpDocumentor*.

Un sistema de control de versiones se encarga de controlar los diversos cambios que se realizan sobre los elementos de algún producto o una configuración del mismo. Los sistemas de control de versiones facilitan la administración de las distintas versiones de cada producto desarrollado, así como las posibles especializaciones realizadas (por ejemplo, para algún cliente específico).

El control de versiones se realiza, principalmente, en la industria informática para controlar las distintas versiones del código fuente. Sin embargo, los mismos conceptos son aplicables a otros ámbitos como documentos, imágenes, sitios web, etcétera.

**María** se ha encargado de documentarse acerca de los programas de software libre existentes para sistemas de control de versiones con la finalidad de implantar alguno de ellos en la empresa, entre los que ha destacado *Git* y *Subversion*.



[Ministerio de Educación y Formación Profesional](#) (Dominio público)

## **Materiales formativos de FP Online propiedad del Ministerio de Educación y Formación Profesional.**

[Aviso Legal](#)

# 1.- Documentación de aplicaciones web.

## Caso práctico

Todos los proyectos software deben disponer de una documentación consistente, de forma que en cualquier etapa de su ciclo de vida, empezando en su fase de análisis y diseño, pasando por su fase de codificación o implementación en un lenguaje de programación determinado e, incluso, en la fase de explotación, debe haber una documentación robusta con la finalidad de suministrar información relevante acerca del código, funcionalidades de la aplicación o, incluso, limitaciones de la misma, para que la aplicación pueda ser explotada en su total amplitud y para que su mantenimiento resulte cómodo.



En **BK programación**, Juan está estudiando diversas herramientas que ayuden a automatizar el proceso de documentación de las aplicaciones, entre las que destaca *phpDocumentor* y *JavaDoc*.

En primer lugar sería necesario responder a la siguiente cuestión, ¿qué conviene documentar en una aplicación? En principio tres aspectos fundamentales de la aplicación:

- 1.- **La interfaz:** Qué hace (no como lo hace) una función o un método de una clase, qué parámetros hay que pasar y qué devuelve. Esta información es tremendamente útil para las personas que utilizan funciones o clases diseñadas por otros.
- 2.- **La implementación:** Indicar cómo está implementada cada función, cómo se lleva a cabo cada paso, por qué se utiliza determinada variable, qué algoritmo se utiliza, qué hacen los métodos privados de una clase. Toda esta información resulta interesante a quienes tengan que depurar o actualizar bloques de código de la aplicación.
- 3.- **La toma de decisiones:** Por qué se ha implementado de determinada forma y no de otra la aplicación, por ejemplo, para analizar el rendimiento de la aplicación y optimización de recursos. Esto resulta interesante a nivel de implementación para los desarrolladores y a nivel funcional para los responsables del desarrollo.

Normalmente la información sobre la implementación no necesita salir del código pero, por el contrario, la información de la interfaz conviene pasarla a un documento independiente del código fuente (manual de uso). La persona que necesite utilizar una determinada librería de clases o funciones tendrá toda la información necesaria: qué hace cada elemento y cómo se utiliza. No necesita acceder al código fuente.

El problema con este tipo de documentación es que cada vez que se modifica algo en el código (actualizaciones, corrección de errores, etc...) hay que reflejarlo también en el manual de uso, lo que implica doble trabajo. Lo ideal, por tanto, sería poder automatizar de alguna forma este proceso

Existen algunas herramientas que permiten generar documentación de forma automática a partir del código fuente. **Javadoc** es la herramienta estándar en Java. Para PHP una de las herramientas más utilizadas es **phpDocumentor**.

Los entornos de programación modernos, por ejemplo, son capaces de obtener la información de los comentarios, de forma que la muestran en el "autocompletado" de código, que se convierte en una herramienta estupenda, y aun imprescindible en lenguajes como PHP, que no necesita que se declare el tipo del argumento de una función, por poner un caso. Entornos como *NetBeans* o *Eclipse*, aprovechan los comentarios de nuestro código fuente para mostrar información muy útil, sobre todo para terceras personas.

Hay que tener en cuenta que todas estas herramientas que venimos viendo, NetBeans, Eclipse, phpDocumentor, esperan el mismo tipo de comentarios, basado en el estándar establecido por Javadoc, de modo que haremos el trabajo una sola vez y podremos aprovecharnos del mismo en varios entornos y con varias herramientas. Más aún, toda persona que se acerque a nuestro proyecto podrá aprovechar la documentación, incluso más que nosotros.

## Citas para pensar

"La programación es una carrera entre ingenieros de software luchando para construir programas cada vez más grandes, mejores y a prueba de idiotas, y el universo intentando producir cada vez más grandes y mejores idiotas. Por ahora, gana el universo. "

*Rich Cook*

## 2.- PhpDocumentor.

### Caso práctico

Una de las herramientas que la empresa **BK programación** ha decidido someter a estudio, con la finalidad de ayudar a documentar el software desarrollado en PHP, y así decidir su implantación, es *phpDocumentor*. Para ello Juan ha empezado a documentarse sobre dicha herramienta.



[betacontinua](#) (CC BY-NC-SA)

Existen algunas herramientas que permiten generar documentación de forma automática a partir del código fuente, **Javadoc** es la herramienta estándar para Java, para PHP una de las herramientas más utilizadas es **phpDocumentor**.

Como ya comentamos antes, la documentación de un proyecto de software es tan importante como su código. Una buena documentación nos facilita, en gran medida, el mantenimiento futuro de la aplicación. Si además estamos trabajando en equipo es muy útil saber lo que hacen las partes que desarrollan otras personas, sobre todo si tenemos que utilizarlas en nuestra parte.

Para ayudarnos existe la aplicación **phpDocumentor**, que nos permite generar automáticamente una buena documentación de nuestro código, de una forma parecida a cómo lo hace **JavaDoc**. Mediante comentarios y unas etiquetas especiales podemos definir de forma sencilla qué hace cada clase, cada método y cada función de nuestro código. Para saber más sobre esta aplicación se puede acceder a su página web, desde donde se puede descargar la aplicación (es *software* libre) y acceder a la documentación de ésta, de todas maneras intentaremos ampliar aquí los conocimientos sobre esta herramienta.

[phpDocumentor](#)

**phpDocumentor** permite generar la documentación de varias formas y en varios formatos.

- ✓ Desde línea de comandos (php CLI - Command Line Interpreter).
- ✓ Desde interfaz web (incluida en la distribución) (No disponible en las versiones más recientes).
- ✓ Desde código. Como phpDocumentor está desarrollado en PHP, podemos incluir su funcionalidad dentro de scripts propios.
- ✓ Integrado con los IDEs de desarrollo en PHP más potentes como Apache Netbeans o Visual Code.

En todo caso, es necesario especificar los siguientes parámetros:

- 1.- El directorio en el que se encuentra nuestro código. PhpDocumentor se encargará luego de recorrer los subdirectorios de forma automática.

- 2.- Opcionalmente los paquetes (**@package**) que deseamos documentar, lista de ficheros incluidos y/o excluidos y otras opciones interesantes para personalizar la documentación.
- 3.- El directorio en el que se generará la documentación.
- 4.- Si la documentación va a ser pública (sólo interfaz) o interna (en este caso aparecerán los bloques **private** y los comentarios **@internal**).
- 5.- El formato de salida de la documentación.

## Formatos de salida

- 1.- **HTML** a través de un buen número de plantillas predefinidas (podemos definir nuestra propia plantilla de salida).
- 2.- **PDF**.
- 3.- **XML** (DocBook). Muy interesante puesto que a partir de este dialecto podemos transformar (**XSLT**) a cualquier otro utilizando nuestras propias reglas y **hojas de estilo**.

Una alternativa a phpDocumentor es Doxygen que puede también documentar código **PHP**, la principal diferencia es que **Doxygen** es un programa, mientras phpDocumentor es una colección de código en PHP. Es decir, genera la documentación con el mismo PHP usado para ejecutar el propio código PHP, es por ello que se necesita tener también PHP instalado, sin embargo no se necesita instalar un servidor web.

Una buena documentación facilita el mantenimiento de la aplicación. Existen herramientas que generan documentación de forma automática a partir del código fuente de las aplicaciones; entre las más utilizadas están **Javadoc**, que es la herramienta estándar en Java, y para PHP una de las herramientas más utilizadas es **phpDocumentor**.

## 2.1.- Instalación de phpDocumentor.

Para proceder a la instalación de **phpDocumentor** vamos a partir de una máquina en la que tenemos instalado la distribución **Ubuntu 20.04 o superior**.

Como requisito previo deberemos instalar los paquetes de XAMPP probaremos si **php** y **apache** están funcionando correctamente; lo podemos establecer con las siguientes pruebas:

1. Para probar si Apache sirve peticiones, abrimos un navegador e introducimos la siguiente URL **http://localhost** y debería aparecer una página de bienvenida a XAMPP Apache.
2. Probar que funciona PHP, lo podemos hacer del siguiente modo, ejecutamos desde línea de comandos: **php -r "phpinfo();"** y deberíamos ver como resultado la configuración de la versión instalada de PHP.
3. Probar que Apache ejecuta código PHP, para ello en la carpeta donde Apache busca las páginas web, es decir en donde se encuentra la página index.html, en nuestro caso la carpeta **/opt/lampp/htdocs**, creamos un archivo al que vamos a llamar **"phpinfo.php"** con el siguiente contenido :

```
<?php
phpinfo();
?>
```

posteriormente tecleamos en el navegador la siguiente URL **http://localhost/phpinfo.php** y deberíamos encontrar información similar a la de la imagen, en donde vemos parte de las características de Apache y PHP de nuestro equipo.

Una vez que probamos que las pruebas anteriores han confirmado el correcto funcionamiento de Apache y PHP, comenzamos la instalación de **phpDocumentor**.

Podemos descargar el paquete mediante:

```
wget https://phpdoc.org/phpDocumentor.phar
```



Elaboración propia (CC0)

Posteriormente se cambia el permiso del fichero para que pueda ser ejecutado y se copia en la carpeta `/usr/local/bin/`.

```
$ chmod +x phpDocumentor.phar
$ sudo mv phpDocumentor.phar /usr/local/bin/phpdoc
```

Ahora ya puedes ejecutar la utilidad de manera global con el comando:

```
$ phpdoc -d . -t docs/api
```

Este comando genera una estructura de documentación vacía en la carpeta docs/api.

## Autoevaluación

Indica si las siguientes afirmaciones son verdaderas o falsas:

- ☐ phpDocumentor es una colección de código en PHP.

- ☐ Para que funcione phpDocumentor es necesario tener instalado PHP.

- ☐ phpDocumentor únicamente va a funcionar en los servidores web.

- ☐ Apache no es necesario para trabajar con phpDocumentor.

Mostrar retroalimentación

## Solución

1. Correcto
2. Correcto
3. Incorrecto
4. Incorrecto



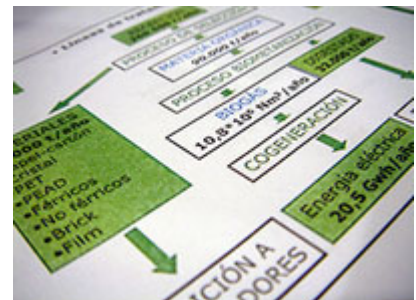
## 2.1.- Funcionamiento de phpDocumentor.

En **phpDocumentor** la documentación se distribuye en bloques "*DocBlock*". Estos bloques siempre se colocan justo antes del elemento al que documentan y su formato es:

```
/**
 * Descripción breve (una línea)
 *
 * Descripción extensa. Todas las líneas que
 * sean necesarias
 * Todas las líneas comienzan con *
 * (línea en blanco=
 * Conjunto de etiquetas para proporcionar información adicional
 */
function suma()
{
    ...
}
```

Los elementos que pueden ser documentados son:

Function  
Constant  
Class  
Interface  
Trait  
Class constant  
Property  
Method



[JaulaDeArdilla](#) (CC BY-NC-ND)

También se puede incluir documentación global a nivel de fichero y clase mediante la marca **@package**.

Dentro de cada **DocBlock** se pueden incluir marcas o etiquetas que serán interpretadas por phpDocumentor con un significado especial, dichas marcas pueden ser las siguientes:

- ✓ **@author**: Autor del código.
- ✓ **@copyright**: Información sobre derechos.
- ✓ **@deprecated**: Para indicar que el elemento no debería utilizarse, ya que en futuras versiones podría no estar disponible.
- ✓ **@example**: Permite especificar la ruta hasta un fichero con código PHP. phpDocumentor se encarga de mostrar el código resaltado (**syntax-highlighted**).
- ✓ **@ignore**: Evita que phpDocumentor documente un determinado elemento.
- ✓ **@internal**: Para incluir información que no debería aparecer en la documentación pública, pero sí puede estar disponible como documentación interna para desarrolladores.
- ✓ **@link**: Para incluir un enlace (<http://...>) a un determinado recurso.
- ✓ **@see**: Se utiliza para crear enlaces internos (enlaces a la documentación de un elemento).

- ✓ **@since**: Permite indicar que el elemento está disponible desde una determinada versión del paquete o distribución.
- ✓ **@version**: Versión actual del elemento.

Existen otras marcas que solamente se pueden utilizar en bloques de determinados elementos:

- ✓ **@global**: Para especificar el uso de variables globales dentro de una función.
- ✓ **@param**: Para documentar parámetros que recibe una función.
- ✓ **@return**: Valor devuelto por una función.
- ✓ **@throws**: Indica si el método o la función puede lanzar algún tipo de excepción.

## Para saber más

Para conocer el repertorio completo de etiquetas o marcas soportadas por PHPDocumentor puedes consultar el siguiente enlace:

[Referencia de etiquetas PHPDocumentor](#)

Vamos a ver un ejemplo de código PHP con marcas para la generación automática de documentación con phpDocumentor.

Vamos a crear un proyecto PHP llamado banco almacenado en el área de despliegue del paquete XAMPP (/opt/lampp/htdocs). Creamos la siguiente estructura de directorios:

/opt/lampp/htdocs/banco

```
|_ public
    |_ index.php
    |_ src
        |_ CuentaBanco.php
```

Los scripts PHP tienen el siguiente código:

Fichero **index.php**:

```
<?php

/*
 * @author daw-profesor daw-profesor@daw.es
 * @copyright 2023 Equipo Daw Distancia
 * @link https://elbanco.com Documentación de condiciones de uso del banco.
 */

require '../src/CuentaBanco.php';

$cuenta1 = new CuentaBanco(150);
```

```

$cuenta2 = new CuentaBanco(300);

$cuenta1->ingreso(10);

try {
    $cuenta2->retirada(50);
} catch (Exception $e) {
    echo 'Message: ' . $e->getMessage();
}

echo "La cuenta ", $cuenta1->getId(), " tiene un saldo de ", $cuenta1->getSaldo(), "</br>";
echo "La cuenta ", $cuenta2->getId(), " tiene un saldo de ", $cuenta2->getSaldo();

```

### Fichero CuentaBanco.php:

```

<?php

/**
 * Clase para representar un cuenta de banco
 *
 * Esta clase representa una cuenta de banco con propiedades como el identificador,
 * y saldo. También incluye métodos para realizar ingresos y retiradas de efectivo
 *
 * @category Banco
 * @package CuentaBanco
 * @subpackage Clases
 * @since 2.0.0
 * @version 2.1.0
 */
class CuentaBanco {

    /**
     * El identificador de la cuenta de banco
     *
     * @var string
     */
    private $id;

    /**
     * El valor del saldo de la cuenta
     *
     * @var float
     */
    private $saldo;

    /**
     * Constructor de la clase CuentaBanco
     *
     * @param float $saldo El saldo de la cuenta del banco
     */
    public function __construct($saldo) {
        $this->id = uniqid();
        $this->saldo = $saldo;
    }
}

```

```
/**
 * Obtiene el identificador de la cuenta de banco
 *
 * @return string El identificador de la cuenta de banco
 */
public function getId() {
    return $this->id;
}

/**
 * Obtiene el saldo de la cuenta de banco
 *
 * @return float El nuevo saldo de la cuenta de banco
 */
public function getSaldo() {
    return $this->saldo;
}

/**
 * Establece el identificador de la cuenta de banco
 *
 * @param string $id El identificador de la cuenta de banco
 */
public function setId($id) {
    $this->id = $numero;
}

/**
 * Establece el saldo de la cuenta de banco
 *
 * @param float $saldo El saldo de la cuenta de banco
 */
public function setSaldo($saldo) {
    $this->saldo = $saldo;
}

/**
 * Realiza un ingreso en la cuenta de banco
 *
 * @param float $cantidad La cantidad a ingresar en la cuenta de banco
 * @return saldo El saldo de la cuenta de banco
 */
public function ingreso($cantidad) {
    if ($cantidad > 0) {
        $this->saldo += $cantidad;
    }
    return $this->saldo;
}

/**
 * Realiza una retirada de la cuenta de banco
 *
 * @param float $cantidad La cantidad a retirar de la cuenta de banco
 * @return saldo El saldo de la cuenta de banco o lanza una excepción si no se ha podido hacer
 * @throws Exception Excepción por intentar retirar dinero de una cuenta sin suficientes fondos
 */
public function retirada($cantidad) {
```

```
        if ($cantidad <= $this->saldo) {  
            $this->saldo -= $cantidad;  
            return $this->saldo;  
        } else {  
            throw new Exception('La retirada no puede realizarse por falta de fondos');  
        }  
    }  
}
```

Una vez construido el proyecto podremos generar la documentación de manera automática ejecutando el siguiente comando desde la carpeta del proyecto:

```
$ phpdoc -d . -t doc
```

La documentación se generará y almacenará en la carpeta doc. Para acceder a ella desde un navegador puedo utilizar la URL: <http://localhost/banco/doc>

## 2.3.- Configuración de phpDocumentor.

En esta sección vamos a describir los parámetros u opciones con los que podemos invocar el comando de generación de documentación con phpDocumentor.

Las tres opciones esenciales para generar la documentación son:

- **-d.** Especifica el directorio o directorios del proyecto que quieres documentar.
- **-f.** Especifica el fichero o ficheros que quieres documentar.
- **-t.** Especifica la localización donde quieres que se almacenen los archivos generados. Este parámetro es opcional. Si no se especifica, los archivos se almacenarán en la carpeta **output**.

Un ejemplo de uso puede ser el siguiente:

```
# phpdoc -d path/to/my/project -f path/to/an/additional/file -t path/to/my/output/folder
```

Otras opciones de configuración son:

- ✓ **--ignore.** Se pueden excluir ciertos ficheros del proceso de generación de documentación.
- ✓ **--hidden, --ignore-symlinks.** Por defecto phpDocumentor ignora los archivos ocultos y los enlaces simbólicos. Si se quieren incluir se puede escribir la opción **--hidden=off** y **--no-ignore-symlinks**.
- ✓ **--template.** Se pueden usar varias plantillas HTML para generar la documentación.
- ✓ **--title.** Esta opción cambia el título en la pestaña del navegador.
- ✓ **--encoding.** Se asume que los caracteres del proyecto están codificados con UTF-8 pero se puede cambiar si no fuera el caso.
- ✓ **--visibility.** Se pueden limitar los elementos que aparecen en la documentación dependiendo del modificador de acceso (**public**, **protected** y **private**) con el que hayan sido definidos.
- ✓ **--ignore-tags.** Se pueden omitir ciertas etiquetas durante la generación de la documentación.
- ✓ **--config.** Sirve para indicar el fichero de configuración utilizado en el proceso de generación de documentación.

Por otro lado, phpDocumentor permite utilizar un fichero de configuración con todos los datos relevantes que deben tenerse en cuenta para generar la documentación.

A continuación, se muestra un ejemplo de dicho fichero:

```
<?xml version="1.0" encoding="UTF-8" ?>
<phpdocumentor
  configVersion="3"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="https://www.phpdoc.org"
  xsi:noNamespaceSchemaLocation="https://raw.githubusercontent.com/phpDocumentor/phpDocum
>
  <paths>
    <output>build/api</output>
    <cache>build/cache</cache>
```

```
</paths>
<version number="3.0.0">
  <api>
    <source dsn=".">
      <path>src</path>
    </source>
  </api>
</version>
</phpdocumentor>
```

## Para saber más

Esta web sirve como manual de referencia, guía de usuario, tutoriales prácticos, etc. sobre phpDocumentor.

[Documentación phpDocumentor.](#)

## 3.- JavaDoc.

### Caso práctico

De igual forma que en **BK programación** se ha decidido someter a estudio la aplicación *phpDocumentor* para así poder decidir su implantación, lo mismo ha ocurrido con *JavaDoc*. De esta manera se está investigando el funcionamiento, configuración, limitaciones, etc. de *JavaDoc*, que nos permitirá desarrollar documentación de forma automatizada de los programas Java que en la empresa se desarrollen.



[Javier Benek](#) (CC BY-NC-SA)

Se pueden encontrar en la red diversas reglas para la generación de documentación de los programas en Java, cada una de ellas con unas características específicas, aunque todas persiguen el mismo objetivo que es documentar los programas Java para que sean más legibles.

**Javadoc** es una utilidad de Sun Microsystems empleado para generar APIs (Aplication Programing Interface) en formato HTML de un archivo de código fuente Java. **Javadoc** es el estándar de la industria para documentar clases de Java, la mayoría de los IDEs los generan automáticamente. Esto facilita la tarea de los desarrolladores, ya que, con sólo seguir una serie de reglas a la hora de generar los comentarios en su código, podrán obtener una buena documentación simplemente usando esta herramienta.

La finalidad de **Javadoc** es intentar evitar que la documentación se quede rápidamente obsoleta cuando el programa continúa su desarrollo y no se dispone del tiempo suficiente para mantener la documentación al día. Para ello, se pide a los programadores de Java que escriban la documentación básica (clases, métodos, etc.) en el propio código fuente (en comentarios en el propio código), con la esperanza de que esos comentarios sí se mantengan actualizados cuando se cambie el código. La herramienta Javadoc extrae dichos comentarios y genera con ellos un juego de documentación en formato HTML.

Básicamente **Javadoc** es un programa, que recoge los comentarios que se colocan en el código con marcas especiales y construye un archivo HTML con clases, métodos y la documentación que corresponde. Este HTML tiene el formato de toda la documentación estándar de Java provista por Sun.

La documentación a ser utilizada por Javadoc se escribe en comentarios que comienzan con `/**` y que terminan con `*/`, comenzando cada línea del comentario por `*` a su vez, dentro de estos comentarios se puede escribir código HTML y operadores para que interprete **Javadoc** (generalmente precedidos por `@`).



**Javadoc** localiza las etiquetas incrustadas en los comentarios de un código Java. Estas etiquetas permiten generar una API completa a partir del código fuente con los comentarios. Las etiquetas comienzan con el símbolo @ y son sensibles a mayúsculas-minúsculas. Una etiqueta se sitúa siempre al principio de una línea, o sólo precedida por espacio(s) y asterisco(s) para que la herramienta Javadoc la interprete como tal. Si no se hace así las interpretará como texto normal.

Hay dos tipos de etiquetas:

- ✓ **Etiquetas de bloque:** sólo se pueden utilizar en la sección de etiquetas que sigue a la descripción principal. Son de la forma: **@etiqueta**
- ✓ **Etiquetas inline:** se pueden utilizar tanto en la descripción principal como en la sección de etiquetas. Son de la forma: **{@tag}**, es decir, se escriben entre los símbolos de llaves.

- ✓ **Javadoc** es una utilidad de Sun Microsystems para generar APIs (Application programming Interface) en formato HTML de un archivo de código fuente Java.
- ✓ La herramienta Javadoc extrae los comentarios del código fuente de los programas Java y genera con ellos un juego de documentación en formato html.
- ✓ La documentación a ser utilizada por Javadoc se escribe en comentarios que comienzan con **/\*\*** y que terminan con **\*/**
- ✓ Las etiquetas se ubican dentro de los comentarios, comienzan con el símbolo @ y son sensibles a mayúsculas-minúsculas.

## 3.1.- Instalación de Javadoc.

---

Para proceder a la instalación de **Javadoc** vamos a partir de una máquina en la que tenemos instalado la distribución **Ubuntu 20.04 o superior**, igual que hemos hecho en el caso del phpDocumentor.

Previamente a la instalación de **Javadoc**, tendremos en cuenta que estamos realizando la programación Java desde una herramienta IDE como puede ser **Eclipse** o **NetBeans**; sin duda son los dos entornos de desarrollo integrados que más han crecido en los últimos tiempos. Su comunidad de desarrolladores sirve como pilar para su crecimiento y evolución constante. El avance de las nuevas tecnologías, nuevos lenguajes y metodologías en el desarrollo del software hacen que lo nuevo quede viejo en poco tiempo. Esto presiona a los programadores a trabajar de manera más intensa agregando nuevas funcionalidades y perfeccionando sus productos en una competencia por ser el mejor IDE.

Para realizar la instalación de **Eclipse** únicamente ejecutamos desde un terminal:

```
$ sudo apt install default-jre  
$ sudo snap install --classic eclipse
```

En Debian, previamente, habremos tenido que instalar Snap:

```
$ sudo apt install -y snapd
```

En el caso de querer realizar la instalación de **NetBeans**, accedemos a la página de NetBeans para realizar la descarga, previamente debemos seleccionar idioma del IDE y plataforma, en este caso español y Linux(x86/x64) respectivamente:

[NetBeans](#)

Una vez descargado el paquete procedemos del siguiente modo:

- ✓ Como requisito, deberemos tener instalado el Java JDK, lo podemos hacer con:

```
$ sudo apt install default-jdk
```

- ✓ Descargamos NetBeans en nuestro sistema:

```
$ sudo wget https://downloads.apache.org/netbeans/netbeans/17/netbeans-17-bin.zip
```

- ✓ Descomprimos el archivo descargado:

```
$ unzip netbeans-17-bin.zip
```

- ✓ Movemos la carpeta al directorio `/opt`:

```
$ sudo mv netbeans/ /opt/
```

- ✓ Tenemos que modificar el archivo `~/.bashrc` y añadirle la ruta:

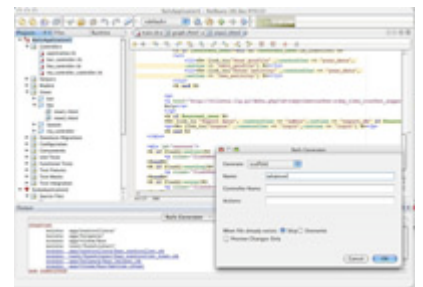
```
export PATH="$PATH:/opt/netbeans/bin/"
```

- ✓ Ejecutamos:

```
$ source ~/.bashrc
```

- ✓ Por último, podemos crear un acceso directo en el escritorio, con lo que crearemos el archivo `/usr/share/applications/netbeans.desktop` y le introducimos las siguientes líneas:

```
[Desktop Entry]
Name=Netbeans IDE
Comment=Netbeans IDE
Type=Application
Encoding=UTF-8
Exec=/opt/netbeans/bin/netbeans
Icon=/opt/netbeans/nb/netbeans.png
Categories=GNOME;Application;Development;
Terminal=false
StartupNotify=true
```



[nicksieger](#) (CC BY-SA)

Tanto Eclipse como NetBeans disponen entre sus opciones la de generar javadoc y mediante diversas ventanas que ofrecen se pueden seleccionar las opciones para **javadoc**. Pero no sólo eso, sino también ofrecen el completado de código javadoc.

Por ejemplo, en el caso de Eclipse, si disponemos del siguiente código (típico ejemplo "*hola mundo*"):

```
public class holamundo {
    /**
     * @param args
     */
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        System.out.println("Hola mundo!");
    }
}
```

```
}  
}
```

ahora si accedemos a la opción de menú "Project" observamos una opción en la que podemos seleccionar "Generate Javadoc..." donde nos permite seleccionar el proyecto del que generar la documentación y también la ruta de la carpeta donde generarla; una vez establecidos dichos parámetros se genera la documentación, que podremos consultar accediendo desde un navegador a los documentos html generados; existe un "*index.html*" desde el que podremos iniciar la navegación e ir accediendo a la documentación generada.

La mayor parte de los entornos de desarrollo incluyen un botón para llamar a **javadoc** así como opciones de configuración; no obstante, siempre se puede ir al directorio donde se instaló el JDK y ejecutar javadoc directamente sobre el código fuente Java. # **javadoc ejemplo.java**

## 3.2.- Funcionamiento de Javadoc.

---

Los comentarios **JavaDoc** están destinados a describir, principalmente, clases y métodos. Como están pensados para que otro programador los lea y utilice la clase (o método) correspondiente, se decidió fijar, al menos parcialmente, un formato común, de forma que los comentarios escritos por un programador resultaran legibles por otro. Para ello los comentarios JavaDoc deben incluir unos indicadores especiales, que comienzan siempre por '@' y se suelen colocar al comienzo de línea. Veamos cómo se introducen los comentarios para Javadoc en un ejemplo parecido al realizado con PHP:

Fichero: **Main.java**

```
package banco;

/**
 *
 * Clase principal que muestra el funcionamiento de la aplicación
 *
 * Esta clase crea dos cuentas bancarias y realiza operaciones de ingreso y
 * retirada
 *
 * @author daw-profesor
 *
 * @version 1.0.0
 *
 * @since 1.0.0
 */
public class Main {

    /**
     *
     * Método principal de la aplicación
     *
     * @param args Argumentos de la línea de comandos
     */
    public static void main(String[] args) {

        // Se crean dos cuentas bancarias con un saldo inicial
        CuentaBanco cuenta1 = new CuentaBanco(150);
        CuentaBanco cuenta2 = new CuentaBanco(300);

        // Se realiza un ingreso en la cuenta1 y una retirada en la cuenta2
        cuenta1.ingreso(10);
        try {
            cuenta2.retirada(50);
        } catch (Exception e) {
            e.printStackTrace();
        }

        // Se muestran los saldos de ambas cuentas
        System.out.println("La cuenta " + cuenta1.getId() + " tiene un saldo de " + cuenta1.getSaldo());
        System.out.println("La cuenta " + cuenta2.getId() + " tiene un saldo de " + cuenta2.getSaldo());
    }
}
```

```
}  
}
```

## Fichero: CuentaBanco.java

```
package banco;  
  
import java.util.UUID;  
  
/**  
 *  
 * Clase para representar una cuenta de banco  
 *  
 * Esta clase representa una cuenta de banco con propiedades como el  
 * identificador  
 *  
 * y saldo. También incluye métodos para realizar ingresos y retiradas de  
 * efectivo.  
 *  
 *  
 * @since 2.0.0  
 *  
 * @version 2.1.0  
 */  
public class CuentaBanco {  
  
    /**  
     * El identificador de la cuenta de banco.  
     *  
     * @var String  
     */  
    private String id;  
  
    /**  
     *  
     * El valor del saldo de la cuenta.  
     *  
     * @var float  
     */  
    private float saldo;  
  
    /**  
     *  
     * Constructor de la clase CuentaBanco.  
     *  
     * @param saldo El saldo de la cuenta de banco.  
     */  
    public CuentaBanco(float saldo) {  
        this.id = UUID.randomUUID().toString();  
        this.saldo = saldo;  
    }  
  
    /**  
     *  
     * Obtiene el identificador de la cuenta de banco.  
     *  
     */
```

```
* @return El identificador de la cuenta de banco.
*/
public String getId() {
    return this.id;
}

/**
 *
 * Obtiene el saldo de la cuenta de banco.
 *
 * @return El saldo de la cuenta de banco.
 */
public float getSaldo() {
    return this.saldo;
}

/**
 *
 * Establece el identificador de la cuenta de banco.
 *
 * @param id El identificador de la cuenta de banco.
 */
public void setId(String id) {
    this.id = id;
}

/**
 *
 * Establece el saldo de la cuenta de banco.
 *
 * @param saldo El saldo de la cuenta de banco.
 */
public void setSaldo(float saldo) {
    this.saldo = saldo;
}

/**
 *
 * Realiza un ingreso en la cuenta de banco.
 *
 * @param cantidad La cantidad a ingresar en la cuenta de banco.
 * @return El saldo de la cuenta de banco.
 */
public float ingreso(float cantidad) {
    if (cantidad > 0) {
        this.saldo += cantidad;
    }
    return this.saldo;
}

/**
 *
 * Realiza una retirada de la cuenta de banco.
 *
 * @param cantidad La cantidad a retirar de la cuenta de banco.
 * @return El saldo de la cuenta de banco.
 * @throws Exception Excepción por intentar retirar dinero de una cuenta sin
 * suficientes fondos.
 */
```

```

*/
public float retirada(float cantidad) throws Exception {
    if (cantidad <= this.saldo) {
        this.saldo -= cantidad;
        return this.saldo;
    } else {
        throw new Exception("La retirada no puede realizarse por falta de fondos");
    }
}
}

```

Como se ve, y esto es usual en **JavaDoc**, la descripción de la clase o del método no va precedida de ningún indicador. Se usan indicadores para el número de versión (**@version**), el autor (**@author**) y otros. Es importante observar que los indicadores no son obligatorios; por ejemplo, en un método sin parámetros no se incluye obviamente el indicador **@param**. También puede darse que un comentario incluya un indicador más de una vez, por ejemplo varios indicadores **@param** porque el método tiene varios parámetros. Resumiendo, los indicadores más usuales:

- ✓ **@author nombreDelAutor descripción.**  
Indica quién escribió el código al que se refiere el comentario. Si son varias personas se escriben los nombres separados por comas o se repite el indicador, según se prefiera. Es normal incluir este indicador en el comentario de la clase y no repetirlo para cada método, a no ser que algún método haya sido escrito por otra persona.
- ✓ **@version númeroVersión descripción.**  
Si se quiere indicar la versión. Normalmente se usa para clases, pero en ocasiones también para métodos.
- ✓ **@param nombreParámetro descripción.**  
Para describir un parámetro de un método.
- ✓ **@return descripción.**  
Describe el valor de salida de un método.
- ✓ **@see nombre descripción.**  
Cuando el trozo de código comentado se encuentra relacionada con otra clase o método, cuyo nombre se indica en nombre.
- ✓ **@throws nombreClaseExcepción descripción.**  
Cuando un método puede lanzar una excepción ("romperse" si se da alguna circunstancia) se indica así.
- ✓ **@deprecated descripción.**  
Indica que el método (es más raro encontrarlos para una clase) ya no se usa y se ha sustituido por otro.

Tanto Eclipse como NetBeans disponen entre sus opciones la de generar javadoc y mediante diversas ventanas que ofrecen se pueden seleccionar las opciones para javadoc. Pero no sólo eso, sino también ofrecen el completado de código javadoc.

Por ejemplo, en Eclipse, si accedemos a la opción de menú "Project" observamos una opción en la que podemos seleccionar "Generate Javadoc..." donde nos permite seleccionar el proyecto del que generar la documentación y también la ruta de la carpeta donde generarla; una vez establecidos dichos parámetros se genera la documentación, que podremos consultar accediendo desde un navegador a los documentos html generados; existe un



"index.html" desde el que podremos iniciar la navegación e ir accediendo a la documentación generada.

banco

CuentaBanco

+

file:///home/boss/eclipse-workspace/banco/doc/banco/CuentaBanco.html

PACKAGE CLASS USE TREE INDEX HELP

SUMMARY: NESTED | FIELD | CONSTR | METHODDETAIL: FIELD | CONSTR | METHOD

SEARCH:

Package banco

**Class CuentaBanco**

java.lang.Object<sup>®</sup>  
banco.CuentaBanco

public class CuentaBanco

extends Object<sup>®</sup>

Clase para representar una cuenta de banco Esta clase representa una cuenta de banco con propiedades como el identificador y saldo. También incluye métodos para realizar ingresos y retiradas de efectivo.

Since:  
2.0.0

Version:  
2.1.0

Constructor Summary

Constructors

Constructor	Description
CuentaBanco(float saldo)	Constructor de la clase CuentaBanco.

Method Summary

All MethodsInstance MethodsConcrete Methods

Modifier and Type	Method	Description
String <sup>®</sup>	getId()	Obtiene el identificador de la cuenta de banco.
float	getSaldo()	Obtiene el saldo de la cuenta de banco.
float	ingreso(float cantidad)	Realiza un ingreso en la cuenta de banco.
float	retirada(float cantidad)	Realiza una retirada de la cuenta de banco.
void	setId(String <sup>®</sup> id)	Establece el identificador de la cuenta de banco.
void	setSaldo(float saldo)	Establece el saldo de la cuenta de banco.

La mayor parte de los entornos de desarrollo incluyen un botón para llamar a javadoc así como opciones de configuración; no obstante, siempre se puede ir al directorio donde se instaló el proyecto y ejecutar javadoc directamente sobre el código fuente Java.

```
$ javadoc -d doc src/banco/*
```

# Para saber más

Puedes consultar el listado completo de etiquetas del comando javadoc en el enlace de su documentación.

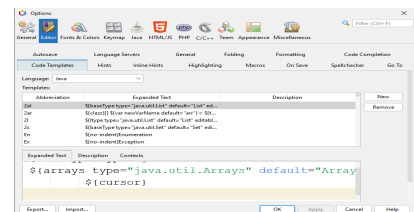
[Oracle javadoc Documentation](#)

## 3.3.- Creación y uso de plantillas de código

Si se usan ciertas convenciones en el código fuente Java (como comenzar un comentario con `/**` y terminarlo con `*/`), **javadoc** puede fácilmente generar páginas HTML con el contenido de esos comentarios, que pueden visualizarse en cualquier navegador. La documentación del API de Java ha sido creada de este modo. Esto hace que el trabajo de documentar el código de nuevas clases Java sea trivial.

Las plantillas:

- ✓ Son sugerencias de código asociadas a palabras clave.
- ✓ En Apache Netbeans se hallan definidas en Tools -> Options -> Editor -> Code Templates.
- ✓ Es aconsejable examinar todas, ya que pueden ahorrar mucho trabajo.
- ✓ Muchas de ellas utilizan nombres similares a las construcciones Java que encapsulan (`try`, `for`, `while`, `if`,...).
- ✓ Podemos definir y crear nuestras propias plantillas.
- ✓ Además existen plantillas Javadoc predefinidas.



Elaboración propia ([CC0](#))

Una plantilla se compone de:

- ✓ un nombre,
- ✓ una descripción,
- ✓ un contexto en función del lenguaje (en java, si estamos en el código, en el javadoc,...) y
- ✓ un pattern, que es el código de la plantilla. Dentro del código de la plantilla podemos usar texto fijo o una serie de variables predefinidas, por ejemplo:
  - `${cursor}`: posición en la que se establecerá el cursor de texto tras desplegar el código de la plantilla.
  - `${enclosing_type}`: tipo de la clase en la que nos encontramos.
  - `${enclosing_method}`: nombre del método en el que nos encontramos.
  - `${year}`: año en curso.
  - `${time}`: hora en curso.

Estas plantillas se mostrarán como sugerencias en el código tras comenzar a escribir su nombre y pulsar CTRL+ espacio. Lo más interesante es que nosotros podemos crearnos nuestras propias plantillas, además de modificar las existentes. Para ello no tenemos más que añadir una nueva desde la opción de "Templates", asignarle un nombre, descripción y elegir el código que queremos que se muestre al seleccionar la misma.

## 4.- Sistemas de control de versiones.

### Caso práctico

En **BK programación**, debido al incremento de trabajo que están teniendo últimamente, han pensado en ampliar la plantilla. Su intención es ofrecer un contrato de trabajo a **Carlos**, pues sus conocimientos de diseño web serán de utilidad para trabajar como programador en la empresa.



**Ada** se está dando cuenta que, a medida que la empresa crece, el número de proyectos aumenta al igual que lo va a hacer la plantilla de personal. Por eso cree fundamental instalar un sistema de control de versiones para facilitar la integración del código fuente y demás documentos, de cada uno de los programadores, a los respectivos proyectos en desarrollo; de este modo la integración conjunta del trabajo individual de cada uno de los empleados será más sencilla, controlada y existirán nuevas posibilidades en cuanto a la disposición del código y documentación de los proyectos.

Cuando realizamos un proyecto software es bastante habitual que vayamos haciendo pruebas, modificando nuestros fuentes continuamente, añadiendo funcionalidades, etc. Muchas veces, antes de abordar un cambio importante que requiera tocar mucho código, nos puede interesar guardarnos una versión de los fuentes que tenemos en ese momento, de forma que guardamos una versión que sabemos que funciona y abordamos, por separado, los cambios.

Si no usamos ningún tipo de herramienta que nos ayude a hacer esto, lo más utilizado es directamente hacer una copia de los fuentes en un directorio separado. Luego empezamos a tocar. Pero esta no es la mejor forma. Hay herramientas, los sistemas de control de versiones, que nos ayudan a guardar las distintas versiones de los fuentes.

Con un sistema de control de versiones hay un directorio, controlado por esta herramienta, donde se van guardando los fuentes de nuestro proyecto con todas sus versiones. Usando esta herramienta, nosotros sacamos una copia de los fuentes en un directorio de trabajo, ahí hacemos todos los cambios que queramos y, cuando funcionen, le decimos al sistema de control de versiones que nos guarde la nueva versión. El sistema de control de versiones suele pedirnos que metamos un comentario cada vez que queremos guardar fuentes nuevos o modificados.

También, con esta herramienta, podemos obtener fácilmente cualquiera de las versiones de nuestros fuentes, ver los comentarios que pusimos en su momento e, incluso, comparar distintas versiones de un mismo fuente para ver qué líneas hemos modificado.

Aunque los sistemas de control de versiones se hacen imprescindibles en proyectos de cierta envergadura y con varios desarrolladores, de forma que puedan mantener un sitio común con las versiones de los fuentes a través de un sistema de control de versiones, también puede

ser útil para un único desarrollador en su casa, de forma que siempre tendrá todas las versiones de su programa controladas.

Los sistemas de control de versiones son programas que permiten gestionar un repositorio de archivos y sus distintas versiones; utilizan una arquitectura cliente-servidor en donde el servidor guarda la(s) versión(es) actual(es) del proyecto y su historia. Sirven para mantener distintas versiones de un fichero, normalmente código fuente, documentación o ficheros de configuración.

## Citas para pensar

"Hay que unirse, no para estar juntos, sino para hacer algo juntos. "  
*Juan Donoso Cortés (1808-1853); Ensayista español.*

## 4.1.- Conceptos básicos de sistemas de control de versiones.

Existen una serie de conceptos necesarios para comprender el funcionamiento de los sistemas de control de versiones, entre los cuales destacamos los siguientes:

- ✓ **Revisión:** Es una visión estática en el tiempo del estado de un grupo de archivos y directorios. Posee una etiqueta que la identifica. Suele tener asociado metadatos como pueden ser:
  - Identidad de quién hizo las modificaciones.
  - Fecha y hora en la cual se almacenaron los cambios.
  - Razón para los cambios.
  - De qué revisión y/o rama se deriva la revisión.
  - Palabras o términos clave asociados a la revisión.
- ✓ **Copia de trabajo:** También llamado "*Árbol de trabajo*", es el conjunto de directorios y archivos controlados por el sistema de control de versiones, y que se encuentran en edición activa. Está asociado a una rama de trabajo concreta.
- ✓ **Rama de trabajo(o desarrollo):** En el más sencillo de los casos, una rama es un conjunto ordenado de revisiones. La revisión más reciente se denomina principal (main) o cabeza. Las ramas se pueden separar y juntar según como sea necesario, formando un grafo de revisión.
- ✓ **Repositorio:** Lugar en donde se almacenan las revisiones. Físicamente puede ser un archivo, colección de archivos, base de datos, etc.; y puede estar almacenado en local o en remoto (servidor).
- ✓ **Conflicto:** Ocurre cuando varias personas han hecho cambios contradictorios en un mismo documento (o grupo de documentos); los sistemas de control de versiones solamente alertan de la existencia del conflicto. El proceso de solucionar un conflicto se denomina **resolución**.
- ✓ **Cambio:** Modificación en un archivo bajo control de revisiones. Cuando se unen los cambios en un archivo (o varios), generando una revisión unificada, se dice que se ha hecho una **combinación** o integración.
- ✓ **Parche:** Lista de cambios generada al comparar revisiones, y que puede usarse para reproducir automáticamente las modificaciones hechas en el código.

Con el empleo de los sistemas de control de versiones se consigue mantener un repositorio con la información actualizada. La forma habitual de trabajar consiste en mantener una copia en local y modificarla. Después actualizarla en el repositorio. Como ventaja tenemos que no es necesario el acceso continuo al repositorio.

Algunos sistemas de control de versiones permiten trabajar directamente contra el repositorio; en este caso tenemos como ventaja un aumento de la transparencia, a pesar de que como desventaja existe el bloqueo de ficheros.



[spiritquest](#) (CC BY)

## Reflexiona

Supongamos que estamos 3 programadores aportando código al mismo proyecto dentro de la empresa **BK programación**; para la integración del código se emplea un sistema de control de versiones; ¿en qué caso se pueden producir conflictos?, ¿cómo soluciona el sistema de control de versiones el conflicto?

Mostrar retroalimentación

El conflicto se va a producir cuando más de un programador intente integrar cambios en el mismo código. La herramienta de control de versiones no soluciona el conflicto, sólo informa de su existencia.

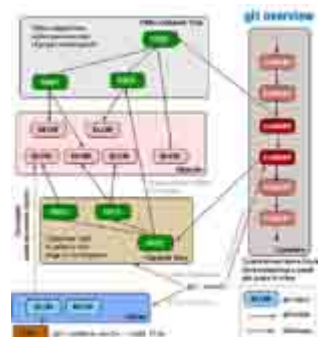
## 4.2.- Procedimiento de uso habitual de un sistema de control de versiones.

El funcionamiento general de un sistema de control de versiones sigue el siguiente ciclo de operaciones:

- ✓ Descarga de ficheros inicial (Checkout):
  - El primer paso es bajarse los ficheros del repositorio.
  - El "checkout" sólo se hace la primera vez que se usan esos ficheros.
- ✓ Ciclo de trabajo habitual:
  - Modificación de los ficheros, para aplicar los cambios oportunos como resultado de la aportación de cada una de las personas encargadas de manipular el código de los ficheros.
  - Actualización de ficheros en local (Update): Los ficheros son modificados en local y, posteriormente, se sincronizan con los ficheros existentes en el repositorio.
  - Resolución de conflictos (si los hay): Como resultado de la operación anterior es cuando el sistema de control de versiones detectará si existen conflictos, en cuyo caso informa de ello y siendo los usuarios implicados en la manipulación del código afectado por el conflicto, los encargados de solucionarlo.
  - Actualización de ficheros en repositorio (Commit): Consiste en la modificación de los ficheros en el repositorio; el sistema de control de versiones comprueba que las versiones que se suben estén actualizadas.

Como conclusión, vemos que los sistemas de control de versiones permiten las siguientes funciones:

- ✓ Varios clientes pueden sacar copias del proyecto al mismo tiempo.
- ✓ Realizar cambios a los ficheros manteniendo un histórico de los cambios:
  - Deshacer los cambios hechos en un momento dado.
  - Recuperar versiones pasadas.
  - Ver históricos de cambios y comentarios.
- ✓ Los clientes pueden también comparar diferentes versiones de archivos.
- ✓ Unir cambios realizados por diferentes usuarios sobre los mismos ficheros.
- ✓ Sacar una "foto" histórica del proyecto tal como se encontraba en un momento determinado.
- ✓ Actualizar una copia local con la última versión que se encuentra en el servidor. Esto elimina la necesidad de repetir las descargas del proyecto completo.
- ✓ Mantener distintas ramas de un proyecto.



Fennq(dbanotes) (CC BY-NC-SA)

Debido a la amplitud de operaciones que el sistema de control de versiones permite aplicar sobre los proyectos que administra, muchos de los sistemas de control de versiones ofrecen establecer algún método de autorización, es decir, la posibilidad por la cual a ciertas personas se les permite, o no, realizar cambios en áreas específicas del repositorio.

Algunos proyectos utilizan un sistema basado en el honor: cuando a una persona se le permite la posibilidad de realizar cambios, aunque sea a una pequeña área del repositorio, lo que reciben es una contraseña que le permite realizar cambios en cualquier otro sitio del repositorio y sólo se les pide que mantenga sus cambios en su área. Hay que recordar que no existe ningún peligro aquí; de todas formas, en un proyecto activo, todos los cambios son revisados. Si alguien hace un cambio donde no debía, alguien más se dará cuenta y dirá

algo. Es muy sencillo, si un cambio debe ser rectificado todo está bajo el control de versiones de todas formas, así que sólo hay que volver atrás.

Entre las funciones que permite realizar un sistema de control de versiones cabe destacar:

- ✓ Sacar copias del proyecto al mismo tiempo.
- ✓ Realizar cambios a los ficheros manteniendo un histórico de los cambios.
- ✓ Los clientes pueden también comparar diferentes versiones de archivos.
- ✓ Unir cambios realizados por diferentes usuarios sobre los mismos ficheros.
- ✓ Sacar una "foto" histórica del proyecto tal como se encontraba en un momento determinado.
- ✓ Actualizar una copia local con la última versión que se encuentra en el servidor.
- ✓ Mantener distintas ramas de un proyecto.



## 4.3.- Sistemas de control de versiones centralizados y distribuidos.

El método de control de versiones usado por mucha gente es copiar los archivos a otro directorio controlando la fecha y hora en que lo hicieron. Este enfoque es muy común porque es muy simple, pero también tremendamente propenso a errores. Es fácil olvidar en qué directorio nos encontramos, y guardar accidentalmente en el archivo equivocado o sobrescribir archivos que no queríamos. Para hacer frente a este problema, los programadores desarrollaron hace tiempo **sistemas de control de versiones locales** que contenían una simple base de datos en la que se llevaba registro de todos los cambios realizados sobre los archivos.

Una de las herramientas de control de versiones más popular fue un sistema llamado rcs. Esta herramienta funciona básicamente guardando conjuntos de parches (es decir, las diferencias entre archivos) de una versión a otra en un formato especial en disco; puede entonces recrear cómo era un archivo en cualquier momento sumando los distintos parches.

El siguiente gran problema que se encuentra la gente es que necesitan colaborar con desarrolladores en otros sistemas. Para solventar este problema, se desarrollaron los **sistemas de control de versiones centralizados** (Centralized Version Control Systems o CVCSs en inglés). Estos sistemas, como CVS, Subversion y Perforce, tienen un único servidor que contiene todos los archivos versionados, y varios clientes que descargan los archivos de ese lugar central. Durante muchos años, éste ha sido el estándar para el control de versiones.



[boirax](#) (CC BY)

Esta configuración ofrece muchas ventajas, especialmente frente a VCSs locales. Por ejemplo, todo el mundo sabe hasta cierto punto en qué está trabajando el resto de gente en el proyecto.

Los administradores tienen control detallado de qué puede hacer cada uno; y es mucho más fácil administrar un CVCS que tener que lidiar con bases de datos locales en cada cliente.

Sin embargo, esta configuración también tiene serias desventajas. La más obvia es el punto único de fallo que representa el servidor centralizado. Si ese servidor se cae durante una hora, entonces durante esa hora nadie puede colaborar o guardar cambios versionados de aquello en que están trabajando. Si el disco duro en el que se encuentra la base de datos central se corrompe, y no se han llevado copias de seguridad adecuadamente, se pierde absolutamente todo, toda la historia del proyecto salvo aquellas instantáneas que la gente pueda tener en sus máquinas locales.

Es aquí donde entran los **sistemas de control de versiones distribuidos** (Distributed Version Control Systems o DVCSs en inglés). En un DVCS (como Git, Mercurial, Bazaar o Darcs), los clientes no sólo descargan la última instantánea de los archivos sino que replican completamente el repositorio. Así, si un servidor muere, y estos sistemas estaban colaborando a través de él, cualquiera de los repositorios de los clientes puede copiarse en el servidor para restaurarlo. Cada vez que se descarga una instantánea, en realidad se hace una copia de seguridad completa de todos los datos.

Es más, muchos de estos sistemas se las arreglan bastante bien teniendo varios repositorios con los que trabajar, por lo que se puede colaborar con distintos grupos de gente de maneras distintas simultáneamente dentro del mismo proyecto. Esto permite establecer varios tipos de flujos de trabajo que no son posibles en sistemas centralizados, como pueden ser los modelos jerárquicos.



## 4.4.- Git como sistema de control de versiones.

---

El núcleo de Linux es un proyecto de software de código abierto con un alcance bastante grande. Durante la mayor parte del mantenimiento del núcleo de Linux (1991-2002), los cambios en el software se pasaron en forma de parches y archivos. En 2002, el proyecto del núcleo de Linux empezó a usar un DVCS propietario llamado BitKeeper.

En 2005, la relación entre la comunidad que desarrollaba el núcleo de Linux y la compañía que desarrollaba BitKeeper se vino abajo, y la herramienta dejó de ser ofrecida gratuitamente. Esto impulsó a la comunidad de desarrollo de Linux (y en particular a Linus Torvalds, el creador de Linux) a desarrollar su propia herramienta basada en algunas de las lecciones que aprendieron durante el uso de BitKeeper. Algunos de los objetivos del nuevo sistema fueron los siguientes:

- ✓ Velocidad.
- ✓ Diseño sencillo.
- ✓ Fuerte apoyo al desarrollo no lineal (miles de ramas paralelas).
- ✓ Completamente distribuido.
- ✓ Capaz de manejar grandes proyectos como el núcleo de Linux de manera eficiente (velocidad y tamaño de los datos).

Git es un sistema rápido de control de versiones, está escrito en C y se ha hecho popular sobre todo a raíz de ser el elegido para el kernel de linux.

Desde su nacimiento en 2005, **Git** ha evolucionado y madurado para ser fácil de usar y. aún así. conservar estas cualidades iniciales. Es tremendamente rápido, muy eficiente con grandes proyectos, y tiene un increíble sistema de ramificación (branching) para desarrollo no lineal.

La principal diferencia entre **Git** y cualquier otro VCS (Subversion y compañía incluidos) es cómo Git modela sus datos. Conceptualmente, la mayoría de los demás sistemas almacenan la información como una lista de cambios en los archivos. Estos sistemas (CVS, Subversion, Perforce, Bazaar, etc.) modelan la información que almacenan como un conjunto de archivos y las modificaciones hechas sobre cada uno de ellos a lo largo del tiempo.

**Git** no modela ni almacena sus datos de este modo, modela sus datos más como un conjunto de instantáneas de un mini sistema de archivos. Cada vez que confirmas un cambio, o guardas el estado de tu proyecto en Git, él básicamente hace una "*foto*" del aspecto de todos tus archivos en ese momento, y guarda una referencia a esa instantánea. Para ser eficiente, si los archivos no se han modificado, Git no almacena el archivo de nuevo, sólo un enlace al archivo anterior idéntico que ya tiene almacenado.

Casi cualquier operación es local, la mayoría de las operaciones en **Git** sólo necesitan archivos y recursos locales para operar; por ejemplo, para navegar por la historia del proyecto, no se necesita salir al servidor para obtener la historia y mostrarla, simplemente se lee directamente de la base de datos local. Esto significa que se ve la historia del proyecto casi al instante. Si es necesario ver los cambios introducidos entre la versión actual de un archivo y ese archivo hace un mes, **Git** puede buscar el archivo hace un mes y hacer un cálculo de diferencias localmente, en lugar de tener que pedirle a un servidor remoto que lo haga, u obtener una



[Treviño \(CC BY-NC-SA\)](#)

versión antigua del archivo del servidor remoto y hacerlo de manera local.

Aparte de todo lo anterior, **Git** posee integridad debido a que todo es verificado mediante una suma de comprobación antes de ser almacenado, y es identificado a partir de ese momento mediante dicha suma. Esto significa que es imposible cambiar los contenidos de cualquier archivo o directorio sin que Git lo detecte. Como consecuencia de ello es imposible perder información durante su transmisión o sufrir corrupción de archivos sin que **Git** sea capaz de detectarlo.

En la siguiente presentación se resume el concepto de las herramienta de control de versiones, entre ellas GIT.

## Sistemas de Control de Versiones

- 1.- ¿QUÉ SON?
- 2.- TIPOS DE SISTEMAS DE CONTROL DE VERSIONES.
- 3.- CONCEPTOS RELACIONADOS
- 4.- ORIGEN Y DESARROLLO DE GIT
- 5.- IMPORTANCIA DE GIT

## Sistemas de Control de Versiones

### 1.- ¿QUÉ SON?:

- ✓ 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.
- ✓ En informática los **Sistemas de Control de Versiones** se encargan de controlar las distintas versiones del código fuente sobre un proyecto de desarrollo de software determinado.

## Sistemas de Control de Versiones

### 2.- TIPOS DE SISTEMAS DE CONTROL DE VERSIONES (I):

- ✓ Centralizados:
  - ➡ Único repositorio.
  - ➡ Almacena todo el código del proyecto.
  - ➡ Lo administra un único usuario o grupo de ellos.
  - ➡ Ejemplos: **CVS**, **Subversion**.

# Sistemas de Control de Versiones

## 2.- TIPOS DE SISTEMAS DE CONTROL DE VERSIONES (II):

- ✓ Distribuidos:
  - Cada usuario tiene su repositorio.
  - Los distintos repositorios pueden intercambiar y mezclar revisiones.
  - Ejemplos: **Git**, **Mercurial**.

# Sistemas de Control de Versiones

## 3.- CONCEPTOS RELACIONADOS:

- ✓ **Repositorio**: Lugar en el que se almacenan los datos actualizados e históricos del proyecto.
- ✓ **Revisión**: Es una visión estática en el tiempo del estado de un grupo de archivos y directorios.
- ✓ **Rama (branch)**: Un módulo o proyecto puede ser bifurcado en un determinado momento (ramificado), y a partir de ahí las dos copias del proyecto ó módulo pueden sufrir distintas transformaciones de modo independiente.
- ✓ **Integración, unión o merge**: Une 2 conjuntos de cambios sobre uno o varios ficheros en una revisión unificada.

# Sistemas de Control de Versiones

## 4.- ORIGEN Y DESARROLLO DE GIT:

- ✓ Durante el desarrollo del núcleo de Linux se empezó utilizando un Sistema de Control de Versiones Distribuido llamado Bitkeeper.
- ✓ En 2005 Bitkeeper dejó de ser gratuito y Linus Torvalds, junto con su equipo decidieron desarrollar su propia herramienta de control de versiones, de donde surge GIT.
- ✓ Git es un sistema rápido de control de versiones, está escrito en C y se ha hecho popular sobre todo a raíz de ser el elegido para el kernel de Linux.
- ✓ Git es rápido, muy eficiente con grandes proyectos, y tiene un increíble sistema de ramificación.
- ✓ Posee integridad debido a que todo es verificado mediante una suma de comprobación antes de ser almacenado, y es identificado a partir de ese momento mediante dicha suma.

# Sistemas de Control de Versiones

## 5.- IMPORTANCIA DE GIT:

- ✓ La autoridad y poder que ha adquirido Git se observa simplemente con ver diversos proyectos que lo utilizan, entre ellos:
  - ➡ Android, Debian, Fedora, Eclipse, CakePHP, GNOME, OpenSUSE, PostgreSQL, Ruby on Rails, Samba, VLC
- ✓ A parte de lo anterior **GitHub**:
  - ➡ Es un servicio de hospedaje web para proyectos que utilizan el sistema de control de versiones Git. GitHub ofrece tanto planes comerciales como planes gratuitos para proyectos de código abierto.

## 4.5.- Funcionamiento de Git.

Git tiene tres estados principales en los que se pueden encontrar los archivos: confirmado (committed), modificado (modified) y preparado (staged).

- ✓ **Confirmado** significa que los datos están almacenados de manera segura en la última foto fija o commit del proyecto en el repositorio.
- ✓ **Modificado** estado en el que se ha modificado el archivo pero todavía no se ha añadido a ninguna foto fija o commit del repositorio.
- ✓ **Preparado** significa que el fichero modificado se ha trasladado al área de preparación para que participe en la siguiente foto fija o commit del repositorio.

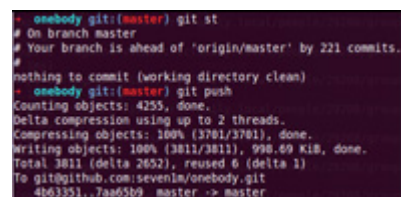
Por otro lado, si un fichero es nuevo en el área de trabajo y no ha sido todavía parte del proceso de creación de una foto fija o commit se dice que es **no rastreado** (untracked). A partir de ahora usaremos el término **commit** para referirnos a una instantánea o foto fija del estado de todos los ficheros del proyecto.

Esto nos lleva a las tres secciones principales de un proyecto de **Git**:

- ✓ El **área de trabajo** (working directory): Se trata del área donde residen los ficheros del proyecto, normalmente se encuentran los ficheros en su último estado de desarrollo, aunque es posible cargar los ficheros tal y como se encontraban en un commit creado en el pasado.
- ✓ El **área de preparación** (staging area): es un sencillo archivo, generalmente contenido en tu directorio de Git, que almacena información acerca de lo que va a ir en el próximo commit. A veces se denomina el índice, pero se está convirtiendo en estándar el referirse a ello como el área de preparación.
- ✓ El **directorio de Git** (Git directory): Almacena los metadatos y la base de datos de objetos que conforman la secuencia de commits creados durante la vida del proyecto. Es la parte más importante de Git, y es lo que se copia cuando se clona un repositorio desde otro ordenador.

El flujo de trabajo básico en Git consiste en:

1. Modificar una serie de archivos en el directorio de trabajo.
2. Preparar los archivos, añadiendo instantáneas de ellos al área de preparación.
3. Confirmar los cambios, lo que toma los archivos tal y como están en el área de preparación, y almacena ese commit de manera permanente en el directorio de Git.



```

onebody git:(master) git st
# On branch master
# Your branch is ahead of 'origin/master' by 221 commits.
#
nothing to commit (working directory clean)
onebody git:(master) git push
Counting objects: 4255, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (3701/3701), done.
Writing objects: 100% (3811/3811), 998.69 KiB, done.
Total 3811 (delta 2652), reused 6 (delta 1)
To git@github.com:sevenm/onebody.git
4b63351..7aa65b9 master -> master
  
```

[Tim Morgan](#) (CC BY-NC-SA)

Si una versión concreta de un archivo está en el directorio de Git, se considera **confirmada** (committed). Si ha sufrido cambios desde que se obtuvo del repositorio, pero ha sido añadida al área de preparación, está **preparada** (staged). Y si ha sufrido cambios desde que se obtuvo del repositorio, pero no se ha preparado, está **modificada** (modified).

## Debes conocer

<https://www.youtube.com/embed/fxolktqi3Rs>





## 4.6.- Instalación y configuración de Git.

---

Para instalar `git` en Windows, puedes seguir estos pasos:

1. Descarga el instalador: Ve al sitio web oficial de Git [Git for Windows](#) y descarga el [instalador](#).
2. Ejecuta el instalador: Una vez descargado, ejecuta el archivo `.exe` para comenzar la instalación.
3. Configuración: Sigue los pasos en el asistente de instalación. Puedes dejar la mayoría de las opciones en sus valores predeterminados, pero presta atención a la pantalla que pregunta sobre el editor de texto predeterminado y la forma en que Git manejará los finales de línea.
4. Finalizar la instalación: Completa la instalación y luego verifica que se haya instalado correctamente abriendo una terminal (CMD o Git Bash) y escribiendo `git --version`.

Puedes encontrar una descripción más detallada del proceso de instalación en este [enlace](#).

Para instalar `git` en Ubuntu se debe ejecutar el comando:

```
$ sudo apt update
$ sudo apt install git
$ git --version
```

Las opciones de configuración reconocidas por **Git** pueden distribuirse en dos grandes categorías: las del lado cliente y las del lado servidor. La mayoría de las opciones que permiten configurar las preferencias personales de trabajo están en el lado cliente. Aunque hay multitud de ellas, aquí vamos a ver solamente unas pocas, nos centraremos en las más comúnmente utilizadas y en las que afectan significativamente a la forma personal de trabajar. Para consultar una lista completa con todas las opciones contempladas en la versión instalada de Git, se puede emplear el siguiente comando:

```
$ git config -help
```

Git trae una herramienta llamada `git config` que permite obtener y establecer variables de configuración que controlan el aspecto y funcionamiento de `git`.

Lo primero que se debe hacer cuando se instala `git` es establecer el nombre de usuario y dirección de correo electrónico. Esto es importante porque las confirmaciones de cambios (commits) en Git usan esta información, y es introducida de manera inmutable en los commits que el usuario va a enviar:

```
$ git config --global user.name "alumno"
$ git config --global user.email "alumno@example.com"
```

Solamente se necesita hacer esto una vez si se especifica la opción `--global`, ya que Git siempre usará esta información para todo lo que se haga en ese sistema. En el caso de querer sobrescribir esta información con otro nombre o dirección de correo para proyectos

específicos, puedes ejecutar el mismo comando sin la opción `--global` cuando estemos en el proyecto concreto.

Una vez que la identidad está configurada, podemos elegir el editor de texto por defecto que se utilizará cuando Git necesite que introduzcamos un mensaje. Si no se indica nada, Git usa el editor por defecto del sistema que, generalmente, es `vi` o `vim`. En el caso de querer usar otro editor de texto, como `emacs`, podemos hacer lo siguiente:

```
$ git config --global core.editor emacs
```

Otra opción útil que puede ser interesante configurar es la herramienta de diferencias por defecto, usada para resolver conflictos de unión (merge). Supongamos que quisiéramos usar `vimdiff`:

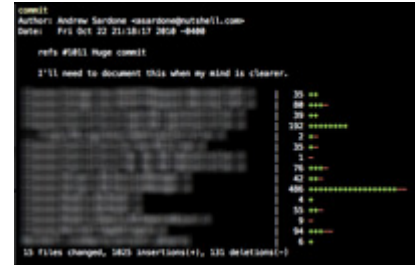
```
$ git config --global merge.tool vimdiff
```

Git acepta *kdifff3*, *tkdiff*, *meld*, *xxdiff*, *emerge*, *vimdiff*, *gvimdiff*, *ecmerge* y *opendiff* entre otras como herramientas válidas. En cualquier momento podremos comprobar la configuración que tenemos mediante el comando:

```
$ git config --list
```

Cuando necesitemos ayuda utilizando Git tenemos tres modos de conseguir ver su página del manual (manpage) para cualquier comando:

```
$ git help <comando>
$ git <comando> --help
$ man git-<comando>
```



[andrew sardone](#) (CC BY-NC-SA)

## 4.7.- Uso básico de Git (Áreas del repositorio).

---

A continuación, vamos a describir los comandos git más utilizados para llevar a cabo distintas operaciones relacionadas con las distintas áreas de un repositorio Git.

### Manejo básico de commits y movimiento de ficheros.

Agregar archivos al área de preparación (staging).

```
git add <archivo>
```

Agregar un archivo específico al área de staging. Para agregar todos los cambios (incluyendo archivos nuevos, modificados, y eliminados):

```
git add .
```

Realizar un commit con todos los archivos en el área de staging, usando un mensaje descriptivo.

```
git commit -m "Mensaje del commit"
```

Realizar un commit directamente con los archivos modificados del área de trabajo sin tener que añadirlos manualmente al área de preparación puedes utilizar el comando:

```
git commit -a -m "Mensaje del commit"
```

Mostrar el estado actual de los ficheros que han sido modificados o que se encuentran en el área de preparación.

```
git status
```

### Ver el historial de commits

Mostrar una lista detallada de los commits en la rama actual, incluyendo el autor, la fecha y el mensaje de commit.

```
git log
```

Mostrar el historial de manera simplificada con un commit por línea o de manera gráfica mostrando las ramas de desarrollo que han sido creadas. Los comandos son:

```
git log --oneline
git log --graph --decorate --oneline
```

## Visualizar diferencias

Mostrar las diferencias en los archivos que han sido modificados pero no añadidos al área de staging.

```
git diff
```

Mostrar las diferencias entre los archivos en el área de staging y el último commit.

```
git diff --staged
```

## Renombrar y borrar archivos

Cambiar el nombre de un archivo y lo añade al área de staging.

```
git mv <nombre_actual> <nuevo_nombre>
```

Eliminar un archivo del directorio de trabajo y lo marca para eliminación en el próximo commit.

```
git rm <archivo>
```

## Restaurar (deshacer) cambios en archivos.

Deshacer cualquier modificación no guardada en el área de trabajo, restaurando el archivo a su último estado commiteado.

```
git restore <archivo>
```

Restaurar temporalmente el estado de tu proyecto a cómo se encontraba en un commit específico. Esto es especialmente útil para la depuración o el análisis del historial.

```
git checkout <hash_del_commit>
```

Mover el archivo especificado fuera del área de staging, pero no altera el contenido del archivo en el directorio de trabajo.

```
git restore --staged <archivo>
```

Crear un nuevo commit que deshace los cambios introducidos por el commit especificado.

```
git revert <hash_del_commit>
```

Quitar todos los archivos del área de staging, pero mantiene los cambios en el directorio de trabajo.

```
git reset
```

Quitar un archivo específico del área de staging, dejándolo en el directorio de trabajo.

```
git reset HEAD <archivo>
```

Resetea tanto el directorio de trabajo como el área de staging al estado de un commit específico, descartando todos los cambios posteriores.

```
git reset --hard <hash_del_commit>
```

## 4.7.1.- Uso de ramas de desarrollo.

---

A continuación, se presentan los comandos de git que permiten la utilización y gestión de ramas de desarrollo dentro de un mismo repositorio.

### Creación y manejo de ramas.

Crea una nueva rama local basada en el commit actual.

```
git branch <nombre_de_la_rama>
```

Cambiar de la rama actual a la rama especificada.

```
git checkout <nombre_de_la_rama>
```

Combina la creación y el cambio de rama en un solo paso.

```
git checkout -b <nombre_de_la_rama>
```

### Visualización y revisión de ramas.

Mostrar una lista de todas las ramas locales.

```
git branch
```

Fusionar la historia de la rama especificada con la rama en la que te encuentras actualmente.

```
git merge <nombre_de_la_rama>
```

Mover toda la serie de commits de la rama actual para que aparezca como si hubiera comenzado desde el último commit de la rama principal. Esto es útil para mantener un historial lineal, lo que facilita la comprensión de la historia del proyecto.

```
git rebase <nombre_de_la_rama_principal>
```

### Renombrado y borrado de ramas

Borrar la rama local especificada.

```
git branch -d <nombre_de_la_rama>
```

Forzar el borrado de la rama, incluso si tiene cambios que no están fusionados.

```
git branch -D <nombre_de_la_rama>
```

Cambiar el nombre de la rama actual a un nuevo nombre.

```
git branch -m <nuevo_nombre_de_la_rama>
```

## 4.7.2.- Uso de repositorios remotos

---

A menudo, cuando se trabaja con Git los desarrolladores trabajan con copias locales de un repositorio remoto al que van a parar sus contribuciones en el desarrollo del proyecto. Este modo de funcionamiento es la clave para entender los procesos de trabajo colaborativo que soporta el uso de Git junto con un espacio de almacenaje de repositorio tal como GitHub, GitLab, Bitbucket u otro servidor de Git en la nube.

A continuación se describen los comandos de trabajo con repositorios remotos más frecuentes.

### Clonar un repositorio remoto.

Crear una copia local del repositorio remoto especificado. La <url> puede ser la dirección del repositorio en un servidor Git. El resultado es un nuevo directorio en tu máquina local que contiene todos los archivos del repositorio, así como las ramas y commits del historial.

```
git clone <url>
```

Esta opción te permite clonar solo una rama específica en lugar de todas las ramas del repositorio.

```
git clone --branch <nombre_de_la_rama> <url>
```

### Configuración y manejo de repositorios remotos.

Agregar un nuevo repositorio remoto con el que te puedes sincronizar subiendo o descargando commits, <nombre\_remoto> es un nombre corto, como origin, para referirse al repositorio, y <url> es la URL del repositorio.

```
git remote add <nombre_remoto> <url>
```

Mostrar una lista de los repositorios remotos configurados con sus URLs.

```
git remote -v
```

Actualizar la URL asociada a un nombre remoto específico.

```
git remote set-url <nombre_remoto> <nueva_url>
```

### Subir y bajar cambios.



Subir los commits de la rama local especificada al repositorio remoto.

```
git push <nombre_remoto> <nombre_de_la_rama>
```

Descargar todos los cambios del repositorio remoto, pero no los integra en tu directorio de trabajo. Útil para ver lo que otros están haciendo sin fusionar esos cambios.

```
git fetch <nombre_remoto>
```

Integrar cambios del repositorio remoto. Es un atajo para ejecutar git fetch seguido de git merge y fusiona los cambios de la rama remota a tu rama actual.

```
git pull <nombre_remoto> <nombre_de_la_rama>
```

## **Inspección y comparación de ramas.**

Mostrar las diferencias entre tu rama local y una rama en el repositorio remoto.

```
git diff <nombre_de_la_rama_local> <nombre_remoto>/<nombre_de_la_rama_remota>
```

## **Limpieza y sincronización**

Eliminar una rama del repositorio remoto.

```
git push <nombre_remoto> --delete <nombre_de_la_rama>
```

Limpiar las referencias locales a ramas remotas que ya no existen en el repositorio remoto.

```
git remote prune <nombre_remoto>
```

## 4.8.- Instalación de servidor Git.

---

Procederemos a instalar Git en una máquina Ubuntu 20.04 o superior.

La manera más directa de instalar Git es desde los repositorios, para ello:

```
$ sudo apt update
$ sudo apt install git
```

Una vez instalado, podemos comprobar la versión instalada:

```
$ git --version
git version 2.34.1
```

Si prefieres instalar Git en Windows puedes descargar el SW de la siguiente [URL](#) y ejecutar el instalador.

Una vez instalado el software de Git podremos hacer uso de todos los comandos para gestionar nuestros repositorios y sus versiones en nuestra máquina local. Además, podemos configurar nuestra máquina para que sirva de servidor de repositorios para que se puedan clonar y actualizar desde otras máquinas.

Para crear un servidor Git deberemos seguir los siguientes pasos:

1. Creamos una cuenta de usuario para Git con el siguiente comando:

```
$ sudo adduser git
```

2. Creamos un directorio para guardar los repositorios Git. Por ejemplo podemos crear el directorio `/var/lib/git`. Establece los permisos de propietario adecuados.

```
$ sudo mkdir /var/lib/git
$ sudo chown git:git /var/lib/git
```

3. Asegúrate que el servicio SSH está corriendo. Puedes comprobar el estado del servicio con el comando:

```
$ sudo systemctl status ssh
```

Si el servicio no está instalado puedes instalarlo con el comando:

```
$ sudo apt-get install openssh-server
```

Si se encuentra parado puedes arrancar el servicio con el comando:


```
$ sudo systemctl start ssh
```

4. Puedes gestionar el servicio de Git mediante el siguiente script de control que debes almacenar en el archivo `/etc/systemd/system/git.service`.

```
[Unit]
Description=Git server

[Service]
User=git
Group=git
ExecStart=/usr/bin/git daemon --verbose --export-all --base-path=/var/lib/git --reuseaddr --stirring

[Install]
WantedBy=multi-user.target
```



Este script describe un nuevo servicio llamado git que ejecuta el demonio Git con el usuario `git` y exporta todos los repositorios almacenados en el directorio `/var/lib/git`. Este servicio escucha en el puerto 9418.

5. Recarga la configuración de la aplicación de gestión de servicios systemd e inicia el servicio `git`.

```
$ sudo systemctl daemon-reload
$ sudo systemctl start git
```

6. Verifica que el servicio funciona correctamente con los comandos:

```
$ sudo systemctl status git
$ sudo netstat -nautp | grep git
```

Para comprobar que podemos descargar repositorios desde el servidor Git vamos a crear un nuevo repositorio en la carpeta `/var/git` y después vamos a clonar dicho repositorio desde tu máquina Windows. Nos colocamos en la carpeta `/var/lib/git` y creamos un repositorio vacío con el usuario `git`.

```
$ sudo -u git git init --bare /var/lib/git/repo.git
```

Ahora ya puedes instalar Git en tu máquina windows y clonar el repositorio vacío almacenado en el servidor con el comando:

```
$ git clone git@<IP_servidor>:/var/lib/git/repo.git
```

Utiliza la IP de la máquina donde hayas instalado el servidor Git. En la primera clonación se realizará el intercambio de la clave SSH del servidor y después se descargará el repositorio.

## 4.9.- Uso de GitWeb en el servidor

Debido a la importancia que actualmente poseen las interfaces web, pasaremos a instalar y configurar el entorno web de Git, éste integra un aspecto más intuitivo y cómodo para el usuario.

Partimos de que en nuestra máquina Debian ya está instalado el servidor web **Apache** como parte de la instalación de **xampp**, de manera que pasamos a instalar el entorno web de Git mediante el comando:



[The Apache Software Foundation \(Apache 2.0\)](#)

```
$ sudo apt-get install gitweb libapache2-mod-perl2
```

Lo siguiente que debemos realizar es crear el archivo de configuración de **gitweb** en el directorio de configuración de Apache localizado en `/etc/apache2/conf`.

```
$ sudo mkdir /etc/apache2/conf  
$ nano /etc/apache2/conf/gitweb.conf
```

Debemos escribir las siguientes líneas en este archivo:

```
Alias /gitweb "/usr/share/gitweb"  
<Directory "/usr/share/gitweb">  
    DirectoryIndex gitweb.cgi  
    Options ExecCGI  
    Require all granted  
    <Files gitweb.cgi>  
        SetHandler cgi-script  
    </Files>  
    SetEnv GITWEB_CONFIG /etc/gitweb.conf  
</Directory>
```

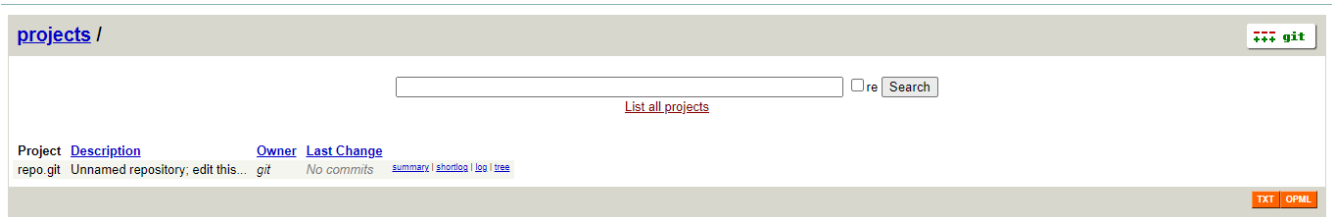
Añadimos una línea al fichero de configuración de Apache `/etc/apache2/apache2.conf` para cargar el fichero de configuración recién creado.

```
# gitweb configuration  
Include conf/gitweb.conf
```

Si los repositorios se quisieran almacenar en otro directorio distinto a `/var/lib/git`, editamos el fichero de configuración de gitweb en `/etc/gitweb.conf` estableciendo el valor de `$projectroot` a la ruta donde residen los repositorios.

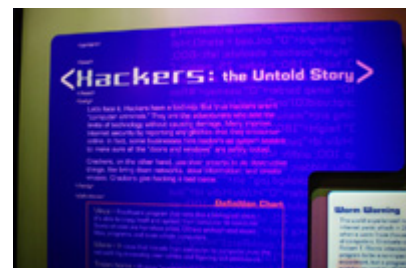
Por último, recargamos el servidor apache desde el panel de control de Xampp.

Si ahora nos conectamos con un navegador a la URL `http://<IP_servidor>/gitweb` podremos acceder a los proyectos que están almacenados en el servidor.



## 4.10.- Seguridad documentación en Git.

Git se encuadra en la categoría de los sistemas de gestión de código fuente distribuido. Con `git`, cada directorio de trabajo local es un repositorio completo no dependiente, de acceso a un servidor o a la red. El acceso a cada repositorio se realiza a través del protocolo de `git`, montado sobre `SSH`, o usando `HTTPS`, aunque no es necesario ningún servidor web para poder publicar el repositorio en la red.



[Morydd](#) (CC BY-NC-ND)

En el flujo de trabajo que hemos dibujado hasta el momento, los desarrolladores no subían directamente cambios al repositorio público, sino que era el responsable del mismo quien aceptaba los cambios y los incorporaba después de revisarlos. Sin embargo, Git también soporta que los desarrolladores puedan subir sus modificaciones directamente a un repositorio centralizado al más puro estilo CVS o Subversion (eliminando el papel de responsable del repositorio público).

El acceso a Git usando `SSH` es una forma segura y conveniente de comunicarse con repositorios remotos sin necesidad de ingresar tus credenciales cada vez que realices operaciones como `push`, `pull` o `clone`. Utilizar `SSH` te proporciona una conexión cifrada entre tu cliente y el servidor Git, asegurando que tus credenciales y tu código permanezcan seguros durante la transmisión. Aquí te explico cómo funciona y cómo puedes configurarlo:

¿Cómo Funciona el Acceso a `git` con `SSH`?

1. **Claves SSH:** Primero necesitas un par de claves `SSH`, una privada y una pública. La clave privada la guardas de forma segura en tu computadora y nunca la compartes. La clave pública se puede compartir de manera segura con otros servicios.
2. **Agregar la Clave Pública al Servicio Git:** Subes tu clave pública a tu perfil de usuario en el servicio Git remoto que estés utilizando (como GitHub, GitLab, Bitbucket, etc.). Esto le permite al servicio identificar y autorizar de forma segura las conexiones entrantes que pretenden ser tuyas.
3. **Configurar Git para Usar SSH:** Configuras tu cliente `git` para usar `SSH` para autenticar con el servidor remoto. Esto implica usar URLs `SSH` para tus remotos en lugar de `HTTPS`.

Configuración Paso a Paso

Aquí está cómo puedes configurarlo, asumiendo que ya tienes Git y SSH instalados:

1. Generar un Par de Claves `SSH`

Si aún no tienes un par de claves `SSH`, puedes generar uno siguiendo estos pasos:

```
ssh-keygen -t rsa -b 4096 -C "tu-email@example.com"
```

Este comando creará una nueva clave `SSH`, usando tu dirección de correo electrónico como etiqueta.

2. Agregar la Clave Pública a tu servidor `git`

Copia el contenido de tu clave pública (por ejemplo, `~/.ssh/git_rsa.pub`) al servidor `git` con el comando:

```
$ ssh-copy-id git@git.dawdistancia.net
```

Ahora ya puedes usar el acceso a través de git clonando los repositorios remotos a través de una URL ligada al acceso por **SSH**, en contraposición al acceso vía **HTTPS**. Por ejemplo:

```
$ git clone git@git.dawdistancia.net:/var/lib/git/banco.git
```

Como puedes comprobar el sistema ya no pide la clave del usuario **git** para realizar dicha petición.

### Ventajas del Uso de **SSH** con **Git**

- ✔ Seguridad Mejorada: La comunicación a través de **SSH** es segura y cifrada, lo que hace que tus datos sean difíciles de interceptar y manipular.
- ✔ Comodidad: Una vez configurado, no necesitas ingresar tus credenciales de usuario cada vez que interactúas con el repositorio.
- ✔ Control de Acceso Mejorada: Puedes fácilmente revocar el acceso cambiando las claves o eliminándolas de tu servidor o servicio **Git**.