# Resumen Tema 6: Documentación y control de versiones.

"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.

## 1.- Documentación de aplicaciones web.

¿Qué conviene documentar en una aplicación? Tres aspectos fundamentales de la aplicación:

* **La interfaz:** qué hace (no como lo hace) una función o un método de una clase, qué parámetros hay que pasar y qué devuelve.
* **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.
* **La toma de decisiones:** por qué se ha implementado de determinada forma y no de otra la aplicación.

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 indenpendiente 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.

Debido a actualizaciones, correciones, etc se necesita automatizar el proceso.

Herramientas que permiten generar documentación de forma automática a partir del código fuente:

- **Javadoc** herramienta estándar para Java.

- **phpDocumentor** herramienta para PHP.

Los entornos de programación modernos como **NetBeans** o **Eclipse** aprovechan los comentarios de nuestro código fuente para mostrar información muy útil.

Estas herramientas, 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.

## 2.- PhpDocumentor.

Herramienta más utilizada para php.

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.

Y de gran utilidad si trabajamos en equipo.

phpDocumentor, genera automáticamente documentación de nuestro código, mediante comentarios y unas etiquetas especiales podemos definir de forma sencilla qué hace cada clase, cada método y cada función de nuestro código.

phpDocumentor (software libre) permite generar la documentación de varias formas y en varios formatos.

* Desde línea de comandos (CLI).
* Desde interfaz web.
* Desde código, scripts.
* Integrado con los IDEs de desarrollo en PHP (NetBeans, VSC, Eclipse).

Parámetros específicos:

1. El directorio en el que se encuentra nuestro código.
2. Opcionalmente los paquetes (@pakage) 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.

Formato de salida:

* **HTML** a través de un buen número de plantillas predefinidas.
* **PDF**
* **XML** (DocBook), a partir de este podemos transformar (XSLT) a cualquier otro utilizando nuestras propias reglas y hojas de estilo.

Alternativa **Doxygen** la diferencia principal es que es un programa, mientras que phpDocumentor es una colección de código en PHP es por eso que se necesita tener PHP instalado.

## 2.1.- Instalación de phpDocumentor

* Instalación de Ubuntu 20.04 o superior
* Instalación de Xampp (el interprete de php esta en: **/opt/lampp/bin/php**.)
* Instalar php, php-xml
* Hacer una prueba: <http://localhost>
* CLI: php -r “phpinfo();”
* En /opt/lampp/htdocs : crear phpinfo.php
* Descargar el paquete mediante: wget <https://phpdoc.org/phpDocumentor.phar>
* Luego:
  + Chmod +x phpDocumentor.phar
  + Sudo mv phpDocumentor.phar /usr/local/bin/phpdoc
* Manera global: phpdoc run –d . –t docs/api
* Este comando genera una estructura de documentación vacía en la carpeta docs/api.

## 2.2.-Funcionamiento de phpDocumentor

La documentación se distribuye en bloques “DocBlock”. Estos bloques siempre se colocan justo antes del elemento al que documentan y su formato es:

|  |
| --- |
| /\*\*  \* texto  \* mas texto  \*/  function suma(){  } |

Los elementos que pueden ser documentados son:

* Function
* Costant
* Class
* Interface
* Trait
* Class constant
* Property
* Method

Se puede incluir documentación global a nivel de fichero y clase mediante **@package.**

Dentro de cada **DockBlock** se pueden incluir marcas o etiquetas:

* @author
* @copyright
* @deprecated
* @example: permite especificar la ruta hasta un fichero con código PHP.
* @ignore: evita que phpDocumentor documente un determinado elemento.
* @internal: documentación interna, NO es pública.
* @link: enlace <http://> a un determinado recurso.
* @see:crea enlaces internos.
* @since: indica que el elemento está disponible desde una determinada versión del paquete o distribución.
* @version

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 devuelo por una función.
* @throws: indica si el método o la función puede lanzar algún tipo de excepción.

Crear proyecto. Y almacenarlo en **/opt/lampp/htdocs**

* **Instalar xampp**
* **Sudo /opt/lampp/lampp start**
* **Sudo ./manager-linux.run**

Generar la documentación de manera automática: phpdoc run –d . –t doc

La documentación se generará y almacenará en la carpeta doc. <http://localhost/banco/doc>

## 2.3.-Configuración de 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 fcheros que quieres documentar.

**-t.** Especfica 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.

Otras opciones de configuración son:

**--ignore.** Se pueden excluir ciertos ficheros del proceso de generación de documentación.

**--hidden, --gnore-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.

phpDocumentor tiene un fichero de configuración:

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8" ?>  <phpdocumentor  configVersion="3"  xsi="http://www.w3.org/2001/XMLSchema-instance"  xmlns="https://www.phpdoc.org"  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> |

## 3.-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.

**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.

Se utiliza /\*\* al comienzo y al final \*/, dentro de estos comentarios se puede escribir código HTML y etiquetas @.

Estas etiquetas permiten generar una API completa a partir del código fuente con los comentarios.

Son sensibles a mayúsculas-minúsculas, se sitúa siempre al principio de una línea o sólo precedida por espacio(s) y arterisco(s) para que la herramienta Javadoc la interprete como tal. Si no se hace así las interpretará como texto normal.

Dos tipos:

* Etiquetas de bloque: sólo se pueden utilizar en la sección de etiquetas que sigue a la descripción principal. @etiqueta
* Etiquetas inline: se pueden utilizar tanto en la descripción principal como en la sección de etiquetas. {@tag}

## 3.1.- Instalación de Javadoc.

* Instalación de Ubuntu 20.04 o superior
* Instalación de Eclipse o NetBeans
* Eclipse
  + Apt install default-jre
  + Snap install --classic eclipse
* NetBeans
  + Apt install default-jdk
* wget <https://downloads.apache.org/netbeans/netbeans/17/netbeans-17-bin.zip>
* unzip netbeans-17-bin.zip
* mv netbeans/ /opt/
* export PATH="$PATH:/opt/netbeans/bin/"
* $ source ~/.bashrc
* /usr/share/applications/netbeans.desktop (acceso directo)

|  |
| --- |
| [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 |

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.

Menú > Project > Generate Javadoc > Seleccionar proyecto y ruta donde se guardará. Documentación generada.

JDK instalado > 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.

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.

**@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 de opciones para generar javadoc y completado de código.
* Menú > Project > Generate Javadoc (seleccionar proyecto y ruta de generación de la documentación)
* Consultar el Html generado.
* También se puede ir al proyecto y ejecutar directamente javadoc: **javadoc -d doc src/banco/\***

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

La documentación del API de Java ha sido creada de este modo. Esto hace 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.

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**. 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.

Los sistemas de control de versiones, que nos ayudan a guardar las distintas versiones de los fuentes.

Con un **VCS** 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.

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.

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

* **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:
  + 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:** 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:** es un conjunto ordenado de revisiones. La revisión más reciente se denomina principal (main). 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.
* **Conflicto**: ocurre cuando varias personas han hecho cambios contradictorios en un mismo documento/s. 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 un **VCS** se consigue mantener un repositorio con la información actualizada. La forma habitual de trabajar consiste en mantener una capa en local y modificarla. Después actualizarla en el repositorio.

Ventaja, no es necesario el acceso continuo al repositorio.

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

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 usuario.
  + Actualización de ficheros en local (**update**): los ficheros son modificados en local y luego se sincronizan con los ficheros existentes en el repositorio.
  + Resolución de conflictos: se informa del conflicto y 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 scv comprueba que las versiones que se suben estén actualizadas.

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.

Los scv 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.

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

Sistema de control de versiones locales, contenían una simple base de datos en la que se llevaba registro de todos los cambios realizados sobre los archivos.

Los **rcs**, funciona básicamente guardando conjuntos de parches (es decir, la 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.

Los sistemas de control de versiones centralizados (**CVCS**), como **CVS**, **Subversión** y Perforce, tienen un único servidor que contiene todos los archivos versionados y varios clientes que descargan los archivos de ese lugar central.

**Ventajas**:

* 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.

**Desventaja**:

* Un servidor centralizado con todas la problemática que acarrea.

Sistema de control de versiones distribuido (**DVCS**): **git**, **mercurial**, bazaar, darcs, los clientes no sólo descargan la última instantánea de los archivos sino que replican completamente el repositorio (copia 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.

**Git** es un sistema rápido de control de versiones, esta **escrito en C** y se ha hecho popular sobre todo a raíz de ser 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 el historial 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 versión antigua del archivo del servidor remoto y hacerlo de manera local.

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.

## 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.

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**).

Tres secciones principales de un proyecto 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.

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**).

## 4.6.-Instalación y configuración de Git.

Instalación en Windows:

* Instalador desde la web oficial.
* Ejecutar e instalar el instalador.
* Configuración, seguir el asistente y opciones por defecto.
* Finalizar la instalación. Git --version

Instalación en Ubuntu:

* apt update
* Apt install git
* Git -–version

Configuración de git:

* Lado cliente
* Lado servidor

Comandos:

* Git config –help

Primeros pasos:

* Git config --global user.name “alumno”
* Git config --global user.mail “alumno@example.com”

Sin la opción global si se quiere modificar la configuración.

El editor de texto por defecto es Vi o Vim, pero si se quiere modificar:

* git config -–global core.editor emacs

Diferencias por defecto es:

* git config --global merge.tool vimdiff

Comprobar la configuración:

* git config --list

Ver página de manual:

* Git help <comando>
* Git <comando> -–help
* Man git-<comando>

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

Agregar archivos al área de preparación:

* Git init
* Git add <archivo>

Para agregar todos los cambios:

* Git add .

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

* Git commit –m “mesaje”

Realizar un commit directamente con los archivos modificados del área de trabajo sin tener que añadirlos:

* Git commit –a –m “mensaje”

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, lista detallada de los commits en la rama actual:

* Git log

Mostrar el historial de manera simplificada con un commit por línea:

* git log --oneline
* git log --graph --decorate --oneline
* git config --global alias.<nombre\_del\_alias> '<comando\_git>'
* nano ~/.gitconfig

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 <actual> <nuevo>

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:

* 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>

Resetear 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.

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>

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 compresión 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.

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>
* Git remote add origin [git@192.168.56.56](http://git@192.168.56.56):/var/lib/git/repo.git

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 los commits de la rama local especificada al repositorio remoto:

* git push <nombre\_remoto> <nombre\_de\_la\_rama>
* git push origin master

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>

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.

Máquina Ubuntu 20.04 o superior.

La manera más directa:

* Sudo apt update
* Sudo apt install git

Versión instalada:

* Git -–version

Para crear un servidor Git deberemos seguir los siguientes pasos:

* Sudo adduser git

Creamos un directorio para guardar los repositorios Git:

* Sudo mkdir /var/lib/git
* Sudo chown git:git /var/lib/git

SSH ejecutandose:

* Sudo systemctl status ssh
* Sudo apt install openssh-server
* Sudo systemctl start ssh

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 --strict-paths /var/lib/git  [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**.

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

* sudo systemctl daemon-reload
* sudo systemctl start git

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.

Instalación del entorno web de Git:

* sudo apt 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 lineas 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.

* Si no sale habilitar : sudo a2enmod cgi

## 4.10.-Seguridad documentación en Git.

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.

¿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 Mejorado: Puedes fácilmente revocar el acceso cambiando las claves o eliminándolas de tu servidor o servicio Git.