

1. Organización de la práctica

Este documento contiene la información necesaria para realizar la tercera práctica de la asignatura. Esta práctica tiene una duración de dos semanas, en las que se explicarán los conceptos básicos relacionados con la gestión del código y documentación de un proyecto software. A continuación se detallan los objetivos y la evaluación de la práctica.

Práctica 3: control de versiones con *Git* y *GitHub*

- Objetivos:

1. Conocer los fundamentos de los sistemas de control de versiones.
2. Crear y manejar los repositorios colaborativos de código fuente.
3. Aprender los elementos básicos del lenguaje *Markdown*.

- Preparación: Los estudiantes deben haber instalado *Git* previamente¹ (ver Sección 2.4). También se recomienda que tengan creada una cuenta de usuario en *GitHub*² antes del comienzo de la clase.
- Seguimiento y evaluación: esta práctica se explica como un taller de iniciación a *Git*, de forma que los estudiantes deben ir realizando los ejemplos propuestos. La práctica no tiene evaluación, y la entrega en Moodle consiste únicamente en indicar la dirección del repositorio *GitHub* creado durante el taller. Dicha entrega servirá para comprobar que el estudiante ha completado la actividad de forma síncrona durante el horario de clase.

¹<https://git-scm.com/downloads>

²<https://github.com/>

2. Control de versiones con *Git* y *GitHub*

2.1. Introducción a los sistemas de control de versiones

Los sistemas de control de versiones (VCS, *Version Control System*) permiten el control y gestión de los cambios que experimenta un producto durante su desarrollo. Los VCS se han convertido en una herramienta esencial para los programadores, facilitando el manejo de versiones del software al registrar toda la actividad (como creación, modificación y borrado) de los artefactos que lo componen. Además, los VCS cuentan con mecanismos para recuperar copias y trabajar de forma colaborativa, siendo por tanto una pieza fundamental en el desarrollo profesional de software.

Por lo general, un VCS consta de un repositorio donde se almacenan los archivos que se encuentran bajo el control de versiones. Uno o más clientes se conectan al repositorio para leer o escribir datos en él. Se distinguen dos tipos de VCS en función de su arquitectura [1], cuyos esquemas pueden verse también en la Figura 1:

1. VCS centralizado. Existe un único repositorio central que almacena los archivos y registra los cambios realizados por los clientes. Es un modelo en el que todos los clientes tienen la visión completa del sistema. Su principal inconveniente es que el repositorio es un punto crítico ante posibles fallos. Uno de los VCS centralizados más conocidos es *Subversion* [2].
2. VCS distribuido. Existe un repositorio remoto del que cada cliente tiene una copia completa, conocida como *mirror repository*. Ante un fallo, el repositorio puede restaurarse a partir del de un cliente, pero mantener la integridad de los datos es más complejo. *Git* es uno de los VCS distribuidos más popular.

A continuación, se definen algunos términos generales necesarios para la comprensión de los VCS:

- **Consignar** (*commit*). Acción de publicar uno o más cambios en el repositorio mediante una transacción atómica. Toda la operación es una unidad que se realiza al completo; si se produce un fallo, se revoca en su totalidad.
- **Revisión** (*revision*). Estado que se crea cada vez que se realiza un *commit*. A cada revisión se le asigna un identificador único.

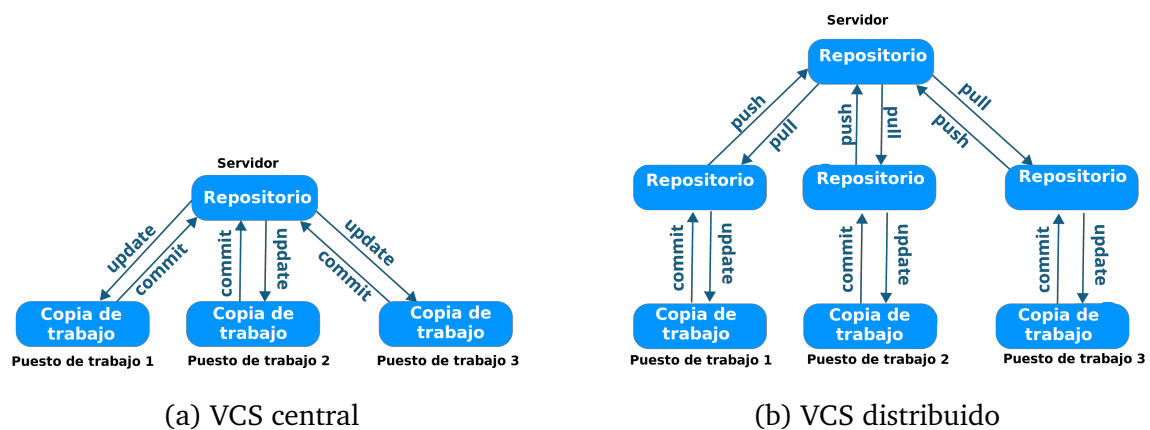


Figura 1: Dos arquitecturas distintas para un sistema de control de versiones. Fuente: <https://www.opentix.es/sistema-de-control-de-versiones/>.

- **Conflicto** (*conflict*). Situación que ocurre cuando varios clientes realizan cambios en sus copias locales que se solapan al intentar publicarlos en el repositorio.
- **Rama** (*branch*). Línea de desarrollo que se crea a partir de una revisión concreta y que evoluciona de forma independiente.

Dado que *Git* es el VCS distribuido que se utilizará en prácticas, el resto de secciones se centrarán en explicar en detalle sus particularidades.

2.2. Conceptos básicos

Git es un VCS distribuido creado en 2005 que se caracteriza por su rapidez y eficiencia para gestionar proyectos, así como por su potente sistema de ramificación [1]. Su funcionamiento se basa en el concepto de *instantánea* (*snapshot*). Cada vez que se realice un *commit*, el sistema evalúa el estado de cada archivo y guarda esa información. Para hacerlo de forma eficiente, los archivos no modificados no vuelven a guardarse, sino que se guarda un enlace al archivo anterior idéntico.

En la Figura 2 se ilustra esta idea. Como se observa, existe una versión inicial para los ficheros A, B, y C. En la segunda versión, los ficheros A y C se modifican, por lo que se convierten en A1 y C1, mientras que el fichero B no ha sido modificado, y se almacena un enlace al fichero anterior. En la versión 3, se modifica únicamente el fichero C, que pasa a ser C2, y los ficheros A1 y B son enlaces a la última versión de los mismos.

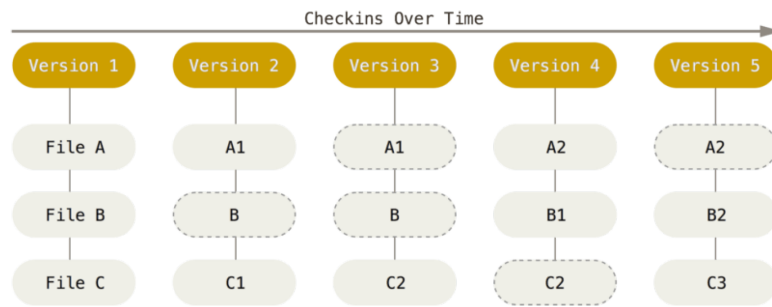


Figura 2: Funcionamiento de los *snapshots* en *Git*. Fuente: [1].

En *Git*, un archivo se encuentra en uno de los tres estados siguientes:

- Confirmado (*committed*). Los datos están actualizados en la copia central del repositorio.
- Modificado (*modified*). El archivo ha sufrido cambios que aún no han sido confirmados.
- Preparado (*staged*). El archivo modificado ha sido marcado (o está preparado) para su próxima confirmación.

Todos los archivos confirmados están alojados en el repositorio central, una base de datos que puede ser local o remota, y cuya metainformación se almacena en un archivo con extensión *.git*. El directorio de trabajo (*working directory*) es la copia local del repositorio central, a la que se accede por el sistema de archivos del ordenador. Finalmente, el área de preparación (a veces llamada también *index*) es donde se registran los cambios que van a confirmarse. La relación entre las tres áreas se muestra en la Figura 3.

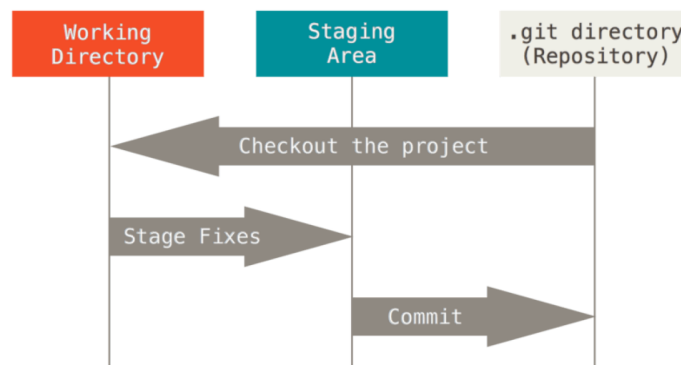


Figura 3: Flujo de trabajo en *Git*. Fuente: [1].

El flujo básico en *Git* consta de los siguientes pasos:

1. Actualizar la copia local del repositorio (*checkout the project*).
2. Realizar los cambios en los archivos que corresponda, y prepararlos para su confirmación (*stage*).
3. Registrar los cambios en el repositorio local (*commit*).

La unidad de información fundamental es el *commit*, que se compone de los siguientes elementos (ver Figura 4, elemento en la izquierda):

- Un identificador único, creado automáticamente por una función *hash* en base a su contenido.
- Referencia al “árbol” de archivos en su estado actual (visión del *snapshot*). La raíz de dicho árbol se encuentra en la parte central de la figura, de la que salen relaciones a los *blobs* con las modificaciones para cada archivo.
- Referencia al *commit* anterior (*commit* padre) si lo hay.
- Mensaje descriptivo, fecha, y autor.

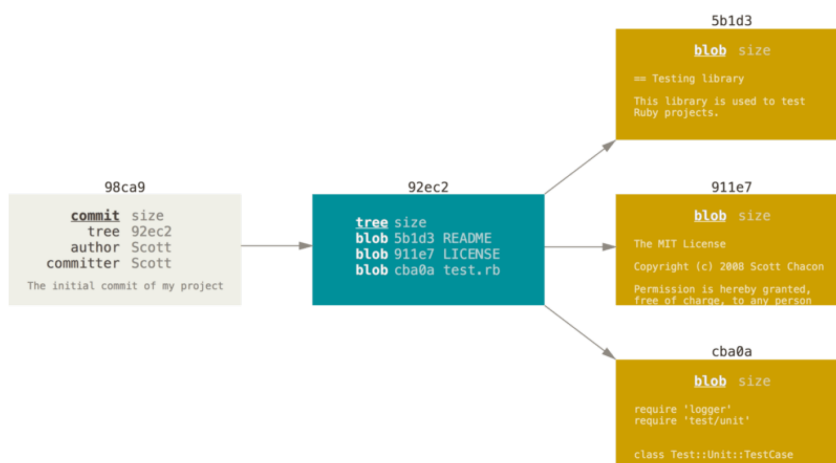


Figura 4: Estructura de un *commit* y su “árbol”. Fuente: [1].

2.3. Gestión de ramas

El sistema de ramificación es una funcionalidad habitual de los VCS modernos, y especialmente relevante en *Git*. La idea es mantener líneas independientes de desarrollo donde se alojan cambios que no queremos que afecten a la rama principal, conocida como rama *master*. Por ejemplo, el código en producción (entregado al cliente) reside en la rama *master*, mientras que el desarrollo de nuevas funcionalidades se realiza en una rama aparte. Una vez la nueva funcionalidad haya sido probada, puede integrarse en la rama *master*, tal y como se representa en la Figura 5.

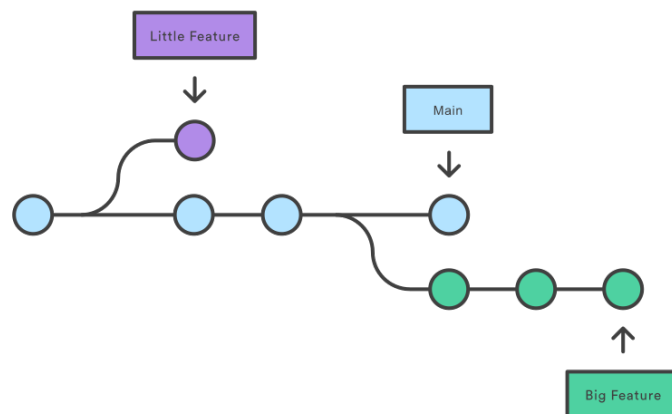


Figura 5: Ramas en un VCS. Fuente: [3].

A diferencia de otros sistemas VCS, las ramas en *Git* no implica copiar archivos entre directorios, ya que es un modelo más abstracto. Cada rama es en realidad una referencia a una secuencia de *commit*. *Git* permite moverse entre ramas, de manera que los *commit* que se realicen se enlazarán a la rama activa. Para saber cuál es la rama activa, *Git* mantiene un “puntero” denominado *HEAD*.

La operación de unificar dos ramas se denomina *merge*. El proceso consiste en identificar un punto de confirmación común entre las dos ramas. En ese punto, se crea una “confirmación de fusión” (*merge commit*). Este *commit* tendrá dos “padres”, y al confirmarlo *Git* intentará mezclar ambas ramas automáticamente. Existen dos estrategias para realizar la fusión (ver Figura 6):

- Fusión con avance rápido (*fast forward merge*). Ocurre cuando podemos recorrer el historial de revisiones de forma lineal entre ambas ramas. Por ejemplo, hemos creado una rama *Some Feature* a partir de la *master*, pero la rama *master* no ha sufrido cambios después (Figura 6a). En este caso, *Git* añade los *commit* de la rama

Some Feature a la rama *master*. La creación del punto de fusión es opcional. Esta estrategia es adecuada para cambios pequeños o corrección de errores.

- Fusión de tres vías (*3-way merge*). Si la rama *master* ha sufrido modificaciones, hacen falta tres confirmaciones (en los dos extremos y en el predecesor común). Se creará explícitamente un punto de fusión a continuación de los dos extremos. Esta estrategia es más habitual cuando se integran funcionalidades de mayor complejidad o que implican mayor tiempo de desarrollo.

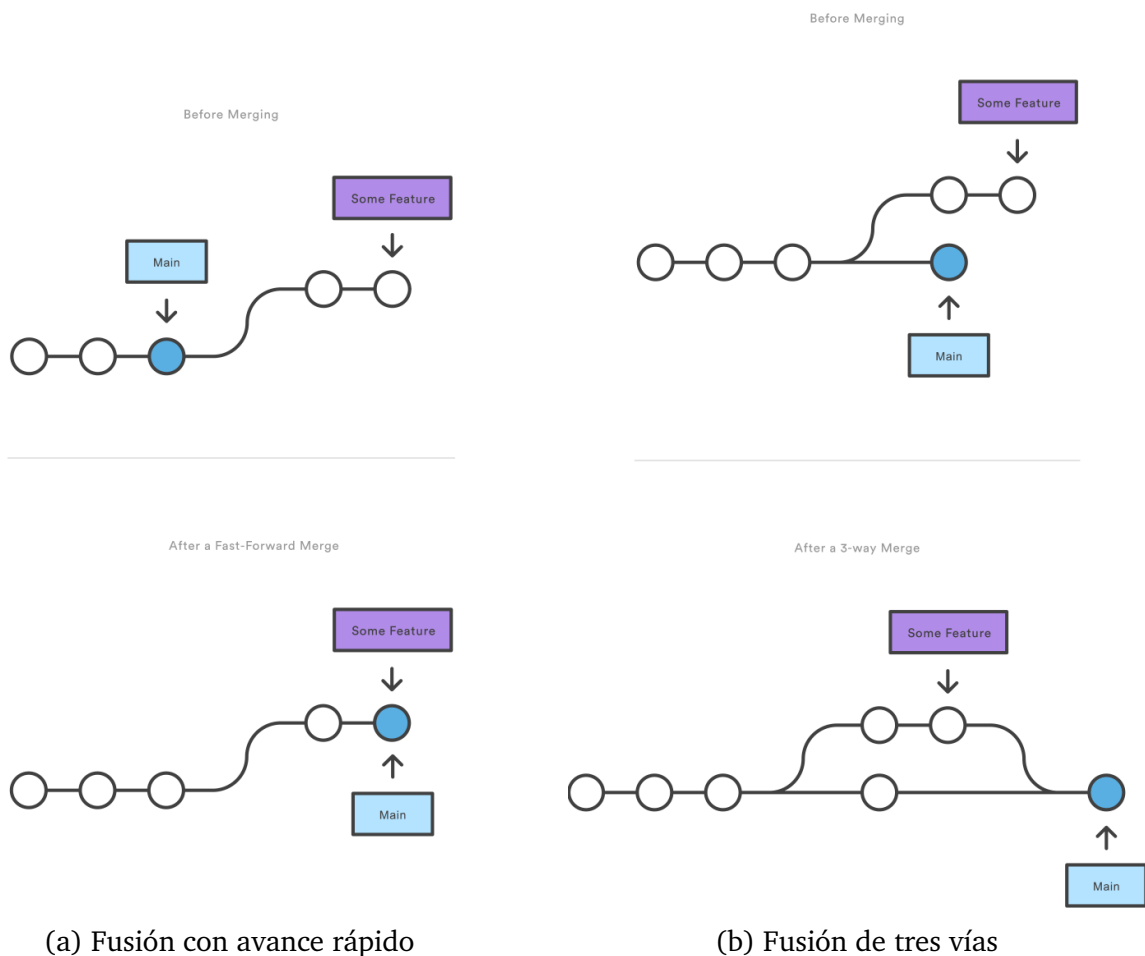
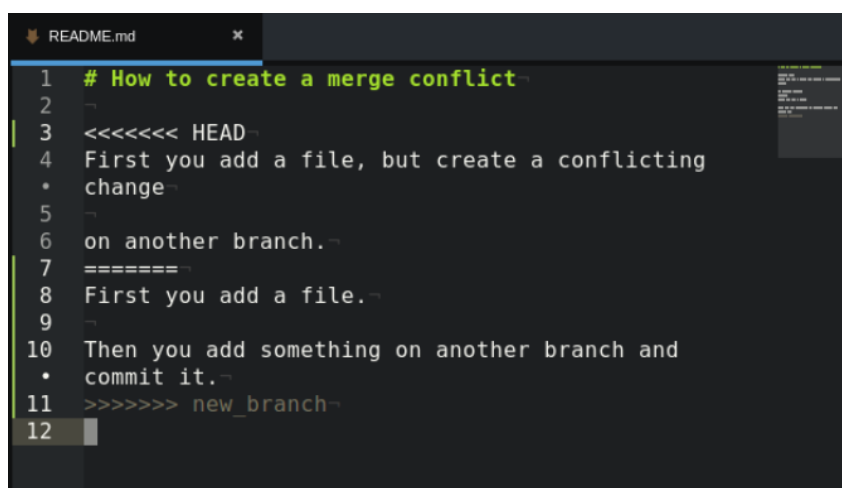


Figura 6: Estrategias de fusión en *Git*. Fuente: [3].

Durante el proceso de fusión de 3 vías, *Git* puede encontrarse con un archivo que ha sido modificado en ambas ramas, por lo que la unión no podrá realizarse automáticamente. La detección de este conflicto requiere la intervención del usuario para resolverlo manualmente. *Git* marca la zona de conflicto en el archivo correspondiente, tal y como se muestra en la Figura 7. La marca “<<<<<< HEAD” en la línea 3 indica el comienzo

del cambio considerando el archivo que se encuentra en la rama activa donde se iba a hacer la fusión (*HEAD*). La marca “=====” en la línea 7 indica el final del cambio en la rama actual, y el comienzo del cambio en la rama bifurcada. Por último, la marca “>>>>>>> new_branch” en la línea 11 representa la línea donde acaba el cambio en la rama bifurcada. El usuario deberá modificar el archivo para mantener uno de los cambios y confirmarlo mediante un nuevo *commit*.



```
1 # How to create a merge conflict
2
3 <<<<<<< HEAD
4 First you add a file, but create a conflicting
5 • change
6 on another branch.
7 =====
8 First you add a file.
9
10 Then you add something on another branch and
11 • commit it.
12 >>>>>>> new_branch
13
```

Figura 7: Visualización de un conflicto en *Git*. Fuente: <https://jonathanmh.com/how-to-create-a-git-merge-conflict/>.

2.4. Instalación, configuración, y comandos útiles

Podemos instalar *Git* en nuestro sistema operativo accediendo a su página web oficial³. Disponemos de un gran número de recursos para iniciarnos en su uso, y aprender aspectos avanzados:

- Documentación oficial (incluye recursos externos como libros, tutoriales, etc.): <https://git-scm.com/doc>
- Tutoriales en vídeo de *GitHub*: <https://www.youtube.com/githubguides>
- Tutoriales de *Atlassian*: <https://www.atlassian.com/es/git/tutorials>

Durante el taller se explicará el uso de *Git* a través de la línea de comandos. En función del sistema operativo en el que trabaje el estudiante, existen diferentes clientes gráficos

³<https://git-scm.com/>

que facilitan la preparación y control de los *commit*⁴. Además, algunos IDEs (como Eclipse, que se utilizará más adelante), disponen de *plug-ins* para utilizar *Git* sin salirnos del entorno de programación.

A continuación, las Tablas 1 y 2 muestran los comandos de configuración de *Git* y comandos útiles en *Git*, respectivamente. Dichos comandos se verán con más detalle durante el tutorial de *Git*. El listado completo de comandos puede consultarse en la página web oficial de *git*⁵. Como referencia rápida, también pueden consultarse las *cheatsheets* de *GitHub*⁶ y *Atlassian*⁷.

Tabla 1: Comandos de configuración de *Git*.

Comando	Descripción
<code>git config --global user.name <name></code>	Establecer nombre de usuario
<code>git config --global user.email <email></code>	Establecer dirección de correo
<code>git config --system core.editor <editor></code>	Establecer el editor de texto
<code>git config --global color.ui true</code>	Habilitar el uso de color
<code>git config --list</code>	Ver la configuración

Tabla 2: Comandos útiles de *Git*.

Comando	Descripción
<code>git init <nombre></code>	Crea un repositorio
<code>git clone <url></code>	Descargar un repositorio
<code>git status</code>	Lista de archivos nuevos o modificados
<code>git diff</code>	Muestra diferencias entre archivos o ramas
<code>git add <archivo></code>	Añadir un archivo al área de preparación
<code>git commit -m <mensaje></code>	Crear un <i>commit</i>
<code>git log</code>	Ver historial de la rama actual
<code>git branch <nombre></code>	Crea una nueva rama
<code>git checkout <rama></code>	Cambia a la rama indicada
<code>git merge <rama></code>	Combinar la rama actual y la indicada
<code>git fetch <nombre></code>	Descargar cambios del repositorio remoto
<code>git pull</code>	Descarga e integra los cambios del repositorio remoto
<code>git push <rama></code>	Sincroniza los <i>commit</i> en el repositorio remoto

⁴<https://git-scm.com/downloads/guis>

⁵https://git-scm.com/docs/git#_git_commands

⁶https://training.github.com/downloads/es_ES/github-git-cheat-sheet/

⁷<https://www.atlassian.com/es/git/tutorials/atlassian-git-cheatsheet>

2.5. Repositorios colaborativos en *GitHub*

GitHub es una aplicación web que da soporte a la creación de repositorios utilizando *Git*, permitiendo a varios usuarios trabajar de manera colaborativa en proyectos. Además, proporciona funcionalidades para registrar errores (*issue tracker*), alojar documentación (web, wikis, etc.), e interaccionar con usuarios de nuestros productos. Cada estudiante deberá registrarse en la plataforma para realizar la segunda parte del taller y alojar el código y documentación que se generen durante el resto de las prácticas.

En el taller se explica cómo crear un repositorio remoto y cómo configurarlo para dar acceso a colaboradores. Una vez creado, podemos clonarlo en nuestro equipo para trabajar de forma local, o utilizar su interfaz para realizar algunas acciones como subir archivos y crear documentación.

2.6. Lenguaje *Markdown*

Markdown es un lenguaje de texto ligero surgido en 2004 como método de conversión entre texto plano y HTML. Tiene una sintaxis sencilla que permite dar formato a contenido web, incluyendo la creación de listas, tablas, etc. Este lenguaje está soportado por una gran cantidad de aplicaciones, entre ellas *GitHub*, convirtiéndose en una forma rápida de generar documentación. Los archivos escritos en *Markdown* por lo general tienen una extensión *.md* o *.markdown*. Como ejemplo, al crear un repositorio en *GitHub*, se nos permite crear un archivo *README.md* que mostrará la información básica de nuestro proyecto.

En la Tabla 3 se muestran algunos de los comandos más básicos para dar formato al texto. Para conocer más detalles del lenguaje, puede consultarse la guía en español de *Markdown*⁸, o ejemplos de sintaxis desde la web de *GitHub*⁹.

⁸<https://markdown.es/>

⁹<https://guides.github.com/features/mastering-markdown/>

Tabla 3: Sintaxis básica del lenguaje *Markdown*.

Formato	Sintaxis
Encabezados	# Primer nivel ## Segundo nivel ### Tercer nivel
Negrita	**Texto negrita**
Cursiva	<i>*Texto cursiva*</i>
Lista numerada	1. Primer elemento 2. Segundo elemento
Lista no numerada	* Primer elemento * Segundo elemento * Sub-elemento
Cita en bloque	> Citas en bloque ¹⁰
Línea de código	'Línea de código'
Bloque de código	'''Bloque de código'''
Enlace	[Texto alternativo](URL enlace)
Imagen	![Texto alternativo](URL o ruta imagen)
Imagen con vínculo	[![Texto alternativo](URL o ruta imagen)](URL enlace)
Línea horizontal	--- ¹¹
Salto de línea	 ¹²

¹⁰Debe empezar y terminar con una línea en blanco.

¹¹Necesita salto de línea antes y después.

¹²Necesita un salto de línea antes.

Referencias

- [1] Scott Chacon and Ben Straub. *Pro git*. Springer Nature, 2014. Disponible en: <https://git-scm.com/book/es/v2>.
- [2] Ben Collins-Sussman, Brian W Fitzpatrick, and C Michael Pilato. *Version control with subversion*. O'Reilly Media, Inc., 2004. Disponible en: <https://svnbook.red-bean.com/>.
- [3] Tutorial git. Disponible en: <https://www.atlassian.com/es/git/tutorials>.