



TITULACIÓN EN MAYÚSCULAS

Curso Académico 2020/2021

Trabajo Fin de Carrera/Grado/Máster

TÍTULO DEL TRABAJO EN MAYÚSCULAS

Autor : Nombre del Alumno

Tutor : Dr. Gregorio Robles



# Proyecto Fin de Carrera

FIXME: Título

**Autor :** FIXME

**Tutor :** Dr. Gregorio Robles Martínez

La defensa del presente Proyecto Fin de Carrera se realizó el día                      de  
de 20XX, siendo calificada por el siguiente tribunal:

**Presidente:**

**Secretario:**

**Vocal:**

y habiendo obtenido la siguiente calificación:

**Calificación:**

Fuenlabrada, a                      de                      de 20XX



*Dedicado a  
mi familia / mi abuelo / mi abuela*



# Agradecimientos

Aquí vienen los agradecimientos... Aunque está bien acordarse de la pareja, no hay que olvidarse de dar las gracias a tu madre, que aunque a veces no lo parezca disfrutará tanto de tus logros como tú... Además, la pareja quizás no sea para siempre, pero tu madre sí.





# Resumen

(DARLE UNA VUELTA CUANDO AL MEMORIA ESTÉ COMPLETA) Este proyecto pretende realizar un estudio sobre la mala interpretación, o el mal uso, de una de las reglas de estilo que contiene el pep8<sup>1</sup>. La regla en cuestión es la siguiente: “**Máxima longitud de las líneas:** Limita todas las líneas a un máximo de 79 caracteres”. Esta regla ha producido un pequeño debate en la comunidad Python, consistente en saber si se está dejando un peor código, un código más ilegible, debido a una mala práctica de los desarrolladores a la hora de “forzar” su código a que cumpla dicha regla del pep8. En el estudio realizado en este proyecto intentamos detectar las líneas de código que han sufrido esta mala práctica por parte de los desarrolladores, la cual consiste en cambiar el nombre de variables, dándoles un nuevo nombre más corto, para cuadrar la longitud de la línea a 79 caracteres. Esto produce que el nuevo nombre de la variable carezca de sentido alguno, lo que hace que el código sea totalmente ilegible.

Si esto ocurre nos encontramos ante un problema, ya que la regla de estilo de máxima longitud está produciendo un efecto contrario al que pretendía.

Para realizar este estudio se ha implementado el lado del servidor con Node.js (utilizando Express) y lenguaje javascript. Con la idea de dejar preparado un servidor por si en un futuro se quiere envolver esta idea en una aplicación web, sólo tener que realizar la parte del cliente en javascript y conectarlo de forma fácil al servidor con Express.

---

<sup>1</sup><http://https://pythonwiki.wikispaces.com/pep8>



# Summary

Here comes a translation of the “Resumen” into English. Please, double check it for correct grammar and spelling. As it is the translation of the “Resumen”, which is supposed to be written at the end, this as well should be filled out just before submitting.



# Índice general

<b>1. Introducción</b>	<b>1</b>
1.1. Motivación . . . . .	1
<b>2. Objetivos</b>	<b>3</b>
2.1. Objetivo principal . . . . .	3
2.2. Objetivos secundarios . . . . .	4
2.3. Estructura de la memoria . . . . .	5
<b>3. Estado del arte</b>	<b>7</b>
3.1. App Cliente-Servidor . . . . .	7
3.2. Node JS . . . . .	8
3.3. Express . . . . .	10
3.4. MongoDB . . . . .	11
3.5. JSON . . . . .	11
3.6. GIT . . . . .	12
3.7. GITFLOW . . . . .	14
<b>4. Diseño e implementación</b>	<b>17</b>
4.1. Instalaciones necesarias . . . . .	17
4.1.1. NodeJS . . . . .	17
4.1.2. NPM . . . . .	18
4.2. Nuevo proyecto . . . . .	19
4.3. Análisis del estudio realizado . . . . .	20
4.3.1. Análisis de los repositorios . . . . .	22
4.3.2. Analizar el grupo de líneas seleccionadas para nuestro estudio . . . . .	24

4.3.3. Ejecución del comando GIT LOG . . . . .	24
4.3.4. Análisis del pasado de una línea . . . . .	28
4.3.5. Clasificación de líneas conflictivas . . . . .	29
<b>5. Resultados</b>	<b>31</b>
<b>6. Conclusiones</b>	<b>33</b>
6.1. Consecución de objetivos . . . . .	33
6.2. Aplicación de lo aprendido . . . . .	33
6.3. Lecciones aprendidas . . . . .	33
6.4. Trabajos futuros . . . . .	34
6.5. Valoración personal . . . . .	34
<b>A. Manual de usuario</b>	<b>35</b>
<b>Bibliografía</b>	<b>37</b>

# Índice de figuras

3.1. Esquema básico de la estructura cliente-servidor . . . . .	7
3.2. Node.js . . . . .	8
3.3. Estructura de aplicación MEAN: Mongo + Express + Angular + Node . . . . .	9
3.4. Express.js . . . . .	10
3.5. MongoDB . . . . .	11
3.6. Ejemplo del formato JSON . . . . .	12
3.7. Esquema de como organizan la información los demás VCS . . . . .	13
3.8. Esquema de como git organiza la información . . . . .	13
3.9. Diagrama GITFLOW . . . . .	15
3.10. Vemos como podemos iniciar el método GITFLOW en nuestro proyecto de manera sencilla. . . . .	16
3.11. Imágenes de ejemplo de uso de Source Tree . . . . .	16
4.1. Descarga de NodeJS . . . . .	17
4.2. Formato del package.json después de lanzar el comando npm init . . . . .	18
4.3. Formato nuestro package.json . . . . .	19
4.4. Estructura básica de un proyecto . . . . .	20
4.5. Diagrama de flujo del funcionamiento del estudio . . . . .	21
4.6. Salida por defecto del comando git log . . . . .	25
4.7. Salida del comando git log con opciones . . . . .	26





# Capítulo 1

## Introducción

### 1.1. Motivación

Cuando terminas de estudiar y por fin empiezas a dedicarte profesionalmente al desarrollo de software, rápidamente te das cuenta de una cosa, la cual nunca, o casi nunca, habías pensado en tu vida de estudiante. En tu trabajo pasas más tiempo leyendo código de otra persona que tu propio código. Esto hace que repentinamente, empiecen a cobrar sentido las frases que tanto te repetían tus profesores sobre estructurar bien tu código o elegir nombres “inteligentes” para las variables o funciones.

Coincidiendo con el inicio de mi etapa profesional, cuando verdaderamente aprendo la importancia de escribir un código de calidad, y no sólo “algo que funcione”, mi tutor, Gregorio me habla sobre un vídeo<sup>1</sup> en el que Raymond Hettinger expone buenas prácticas para escribir un código inteligente y de calidad. En este vídeo se plantea, lo que para mi, es una interesante cuestión, la cual, podemos resumir en la siguiente pregunta: ¿Podemos dejar un peor código al realizar un cambio para cumplir una regla del pep8?.

Dicha cuestión es la que vamos a estudiar en este TFG, sin embargo, nos centraremos en un caso muy concreto. La regla en cuestión es la siguiente: “Máxima longitud de las líneas: Limita todas las líneas a un máximo de 79 caracteres” y el caso que nos preocupa es que el desarrollador se encuentre ante una línea de código mayor a 80 caracteres y decida cambiarla haciendo más corto el nombre de alguna variable para cuadrar el tamaño a 79 caracteres, de forma que el nombre de la variable, después del cambio, carezca de ningún sentido.

---

<sup>1</sup><https://www.youtube.com/watch?v=wf-BqAjZb8M>

Este hecho me parece digno de estudio, por varios motivos, uno de ellos, indudablemente es el debate que puede crear en la comunidad Python sobre si un código que cumple el pep8 puede ser de peor calidad que uno que no lo cumpla, debido a esta cuestión.

Pero siendo sincero, el motivo por el cual decidí realizar el proyecto sobre este tema fue que, con mi experiencia como estudiante, no pude evitar imaginar el siguiente pensamiento por parte de cualquiera de nosotros: “He terminado mi práctica de python, voy a pasar a mi código un filtro que me he descargado del pep8 para entregar un buen código, lo más importante es que pase este filtro, por tanto, si me salta algún error haré cualquier cambio rápido y sin pensar mucho sólo para forzar a que el filtro pase correctamente”.

Creo firmemente que como estudiantes no somos capaces de asimilar la importancia de escribir un código de calidad, y de las horas y horas que ahorraremos a nuestros compañeros de trabajo, o a nosotros mismo cuando retomemos nuestro propio código de hace tiempo. Son cosas simples, pero es importante que desde el principio de nuestra formación se cojan estas buenas costumbres.

# Capítulo 2

## Objetivos

### 2.1. Objetivo principal

En consecuencia, este proyecto se plantea la siguiente tarea, dado una serie de repositorios de GitHub, analizar todo el código python que haya en cada uno de ellos. El proyecto analiza este código línea por línea y detecta aquellas que han sufrido el cambio que se explica en el apartado anterior, es decir, detecta las líneas que para forzar a pasar la regla del pep8 de la máxima longitud han sido acortadas cambiando el nombre de una variable por un nombre más corto, y por tanto, menos entendible.

Por una parte se pretende hacer un estudio bastante amplio, con un volumen de repositorios de GitHub grande, para obtener unos resultados lo suficientemente amplios como para poder llegar a una conclusión. Se busca detectar si este mal hábito está muy extendido en los desarrolladores de python.

Por otra parte, también se pretende alcanzar otro objetivo más profundo, el cual consiste en intentar interiorizar una cuestión, sobre todo a nivel de estudiantes. Dicha cuestión es la importancia de saber realmente lo que es un buen código, sobre todo, no caer en el típico error cuando eres estudiante de pensar que un buen código es algo que funciona.

Según está orientado en la actualidad el aprendizaje de la programación a nivel de universidad, es muy difícil darse cuenta que el código que produces no es algo que vas a escribir una vez, vas a probar que funciona, vas a estregar y no vas a volver a ver en tu vida. Este ciclo es lo que suele pasar con el código que generas como estudiante. Esto unido a ciertas circunstancias como agobios y prisas por los plazos de entrega, o que en algunas ocasiones sólo se valore el

resultado del programa hace que se creen malos hábitos o malas costumbres a la hora de generar código de calidad por parte de los alumnos.

El proyecto intenta llegar a este objetivo creando un “debate” alrededor del caso concreto que se estudia. ¿Es mejor el código que no cumple el pep8 antes de realizar el cambio, pero que tiene un buen nombre de variable?, ¿o es mejor el código después del cambio que cumple el pep8 pero que ha dejado la variable con un nombre sin sentido?.

Quizás la respuesta correcta sea decir que se debe cambiar la línea de código para que cumpla la regla de máxima longitud de línea pero realizando otro cambio, y si no es posible cambiar esa sola línea, darle una vuelta al código para refactorizarlo y hacer que todas las líneas cumplan de esta forma las reglas de calidad.

Lo que se intenta decir con esto es que cuando un estudiante ya ha adquirido ciertos conocimientos técnicos sobre la programación y sobre la generación de código de calidad, debe darse cuenta que tenemos en nuestra mano una serie de reglas de calidad como el pep8 o similares, las cuales se deben utilizar, pero no de cualquier forma, es decir, hay que pararse a pensar un minuto y ser coherente e inteligente a la hora de aplicarlas en nuestro desarrollo, y no dejar un código peor que el que teníamos antes por intentar forzar a que pasen estas reglas de cualquier manera.

## 2.2. Objetivos secundarios

Aparte de objetivo explicado anteriormente, a la hora de realizar este proyecto, se han tenido en cuenta los siguientes objetivos secundarios:

- **Metodología de trabajo:** Se ha intentado llevar una metodología de trabajo lo más parecida posible a lo que nos encontraremos en un futuro entorno profesional. Para ello se ha intentado mantener un uso constante de Git y GitFlow.
- **Aplicación de tecnologías e ideas de trabajo no vistas en la universidad:** El proyecto de fin de carrera es el broche, la guinda a nuestra vida universitaria. Debido a esto, he intentado basar este proyecto sobre tecnologías e ideas de trabajo que no controlo en profundidad, ya que no se han enseñado o se ha pasado de refilón sobre ellas en mi vida universitaria. De esta forma lo que busco es ampliar mis conocimientos sobre este mundo

y aprovechar esto para conseguir nuevas herramientas que me pueden valer para mi futura vida laboral. Esta cuestión puede dividirse, sobre todo, en dos casos:

1. En cuanto a la aplicación de tecnologías he usado para hacer la parte del servidor javascript, utilizando para ello node.js con express (entraremos más en detalle en siguientes capítulos), mi elección se debe a que en la universidad se ha enseñado a montar servidores con java, y sobre todo con python y django. De esta forma quería dominar otros casos para tener más donde comparar. Para la base de datos he utilizado MongoDB, en lugar de SQL por el mismo motivo, en la universidad hemos utilizado bases de datos relacionales, y quería utilizar una no relacional, para entender mejor los pros y contras de cada una de ellas, y tener más conocimientos sobre cual elegir en mis futuros desarrollos.
2. Enfocar una idea de trabajo más orientada a la investigación. Normalmente en nuestra vida como estudiante partimos siempre de un enunciado redactado al empezar a desarrollar, en el proyecto tenemos la oportunidad de investigar sobre una idea para realizar luego nuestro desarrollo.

## 2.3. Estructura de la memoria

Creemos conveniente explicar muy brevemente la estructura de la memoria, dando a conocer los objetivos de cada capítulo para facilitar así la lectura de la misma:

1. **Introducción:** En esta punto de intenta explicar el contexto que envuelve este proyecto, así como las razones por las que se elige el tema a tratar.
2. **Objetivos:** En esta sección se detalla cada uno de los objetivos que se han planteado desde el inicio del proyecto.
3. **Estado del arte:** Aquí presentamos las tecnologías con las que se ha implementado el proyecto, además de algún concepto para entender mejor la estructura del mismo.
4. **Diseño e implementación:** Este punto profundiza e intenta explicar en detalle el desarrollo del proyecto.

5. **Resultados:** Aquí se presenta , a modo de resumen, como ha quedado finalmente el proyecto.
6. **Conclusiones:** Es el capítulo final, en él se intenta evaluar de forma general el proyecto, haciendo hincapié en los conceptos aprendidos en su elaboración.

Finalmente se expondrá la bibliografía que se ha consultado para la elaboración de la memoria.

# Capítulo 3

## Estado del arte

En esta sección vamos a describir brevemente las tecnologías aplicadas en este proyecto.

### 3.1. App Cliente-Servidor

La arquitectura cliente-servidor es una de las más extendidas en la actualidad. Dicha estructura es un modelo de aplicación distribuida en el que las tareas se reparten entre los proveedores de recursos o servicios, llamados **servidores**, y los demandantes, llamados **clientes**. Un cliente realiza peticiones a otro programa, el servidor, quien le da respuesta:

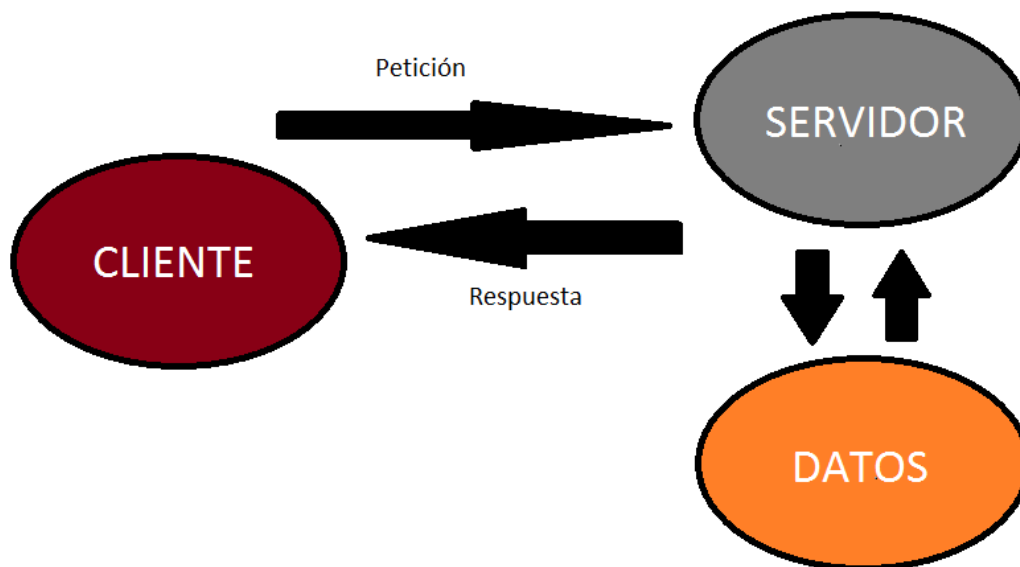


Figura 3.1: Esquema básico de la estructura cliente-servidor

Esta arquitectura presenta una clara separación de las responsabilidades lo que facilita y clarifica el diseño del sistema. Es una arquitectura claramente centralizada, ya que toda la información reside en el servidor, y el cliente es el que realiza peticiones para obtener dicha información. Esto provoca que el servidor posea una lógica más compleja, es capaz de manejar de forma distinta las peticiones dependiendo de quien las realice. Esto también se hace por motivos de seguridad, ya que la parte del cliente es mucho más sensible a ataques.

En nuestro proyecto no hemos realizado como tal una app de cliente-servidor, ya que ha ido más enfocado a la investigación y a la obtención de resultado. Sin embargo, el código realizado para obtener dichos resultados se ha dejado organizado en la parte del servidor de una aplicación web, para obtenerlo con distintas peticiones. Por si en un futuro se quiere mejorar este proyecto, y se decide hacer una aplicación web con el contenido investigado aquí, solo sería necesario realizar la parte del cliente.

## 3.2. Node JS



Figura 3.2: Node.js

Node.js es una librería y entorno de ejecución de E/S dirigida por eventos y por lo tanto, asíncrona que se ejecuta sobre el intérprete de JavaScript creado por Google V8.

Node ejecuta javascript utilizando el motor V8, desarrollado por Google. Este motor permite a Node proporcionar un entorno de ejecución del lado del servidor que compila y ejecuta javascript a velocidades increíbles. El aumento de velocidad es importante debido a que V8 compila Javascript en código de máquina nativo, en lugar de interpretarlo o ejecutarlo como bytecode. Node es de código abierto, y se ejecuta en Mac OS X, Windows y Linux.



Aunque Javascript tradicionalmente ha sido relegado a realizar tareas menores en el navegador, es actualmente un lenguaje de programación totalmente, tan capaz como cualquier otro lenguaje tradicional como C++, Ruby o Java. Además Javascript tiene la ventaja de poseer un excelente modelo de eventos, ideal para la programación asíncrona.

¿Cuál es la razón por la que decidimos adentrarnos en elaborar un servidor en Javascript con node.js?

¿Cuál es el problema con los programas de servidor actuales? A medida que crece su base de clientes, si quieres que tu aplicación soporte más usuarios, necesitarás agregar más y más servidores. Esto hace que el cuello de botella en toda la arquitectura de aplicación Web era el número máximo de conexiones concurrentes que podía manejar un servidor.

Node resuelve este problema cambiando la forma en que se realiza una conexión con el servidor. En lugar de generar un nuevo hilo de OS para cada conexión (y de asignarle la memoria acompañante), cada conexión dispara una ejecución de evento dentro del proceso del motor de Node. Node también afirma que nunca se quedará en punto muerto, porque no se permiten bloqueos.

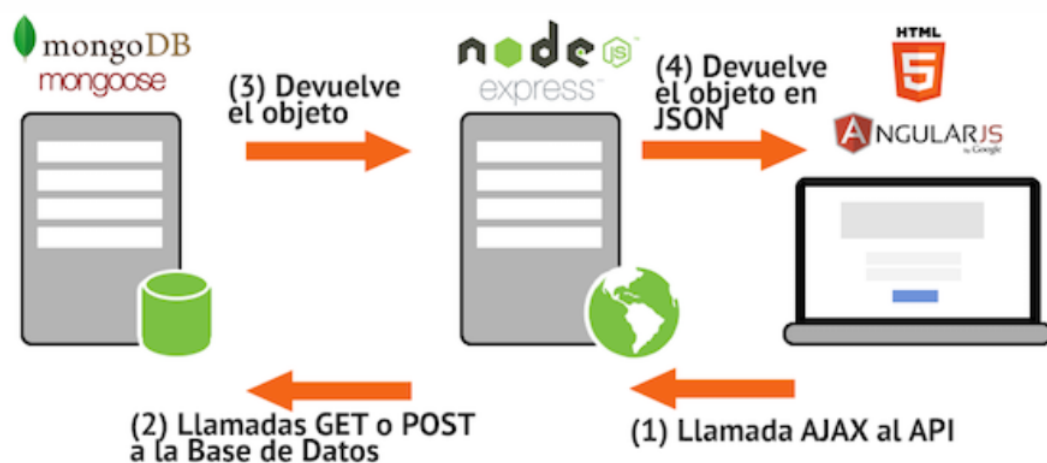


Figura 3.3: Estructura de aplicación MEAN: Mongo + Express + Angular + Node

### 3.3. Express



Figura 3.4: Express.js

Express es sin duda el framework más conocido de node.js, es una extensión del poderoso connect<sup>1</sup> y esta inspirado en sinatra<sup>2</sup>, además es robusto, rápido, flexible, simple, etc.

Proporciona una delgada capa de características de aplicación web básicas, que no ocultan las características de Node.js que tanto ama y conoce.

El verdadero éxito de Express se encuentra en lo sencillo que es de usar. Tienes la capacidad de crear de forma rápida y sencilla una API sólida. Además, express abarca un sin número de aspectos que muchos desconocen pero son necesarios.

De entre las tantas cosas que tiene este framework podemos destacar:

- Session Handler.
- 11 middleware poderosos así como de terceros.
- cookieParser, bodyParser ...
- vhost
- router

---

<sup>1</sup><https://github.com/senchalabs/connect#readme>

<sup>2</sup><http://www.sinatrarb.com/documentation.html>

### 3.4. MongoDB



Figura 3.5: MongoDB

MongoDB forma parte de la nueva familia de sistemas de base de datos NoSQL, es la base de datos NoSQL líder y permite a las empresas ser más ágiles y escalables.

En lugar de guardar los datos en tablas como se hace en las base de datos relacionales, MongoDB guarda estructuras de datos en documentos similares a JSON con un esquema dinámico (MongoDB utiliza una especificación llamada BSON). Esto hace que sea una base de datos ágil ya que permite a los esquemas cambiar rápidamente cuando las aplicaciones evolucionan, proporcionando siempre la funcionalidad que los desarrolladores esperan de las bases de datos tradicionales, tales como índices secundarios, un lenguaje completo de búsquedas y consistencia estricta.

### 3.5. JSON

JSON (JavaScript Object Notation) es un formato de texto ligero para el intercambio de datos. JSON es un subconjunto de la notación literal de objetos de JavaScript aunque hoy, debido a su amplia adopción como alternativa a XML, se considera un formato de lenguaje independiente.

JSON nació como una alternativa a XML, el fácil uso en javascript ha generado un gran número de seguidores de esta alternativa.

Una de las supuestas ventajas de JSON sobre XML como formato de intercambio de datos es que es mucho más sencillo escribir un analizador sintáctico (parser) de JSON, (), lo cual ha sido fundamental para que JSON haya sido aceptado por parte de la comunidad de desarrolladores AJAX, debido a la ubicuidad de JavaScript en casi cualquier navegador web.

Otra ventaja a tener en cuenta del uso de JSON es que puede ser leído por cualquier lenguaje de programación. Por lo tanto, puede ser usado para el intercambio de información entre distintas tecnologías

```
{ "users": [
  {
    "firstName": "Ray",
    "lastName": "Villalobos",
    "joined": {
      "month": "January",
      "day": 12,
      "year": 2012
    }
  },
  {
    "firstName": "John",
    "lastName": "Jones",
    "joined": {
      "month": "April",
      "day": 28,
      "year": 2010
    }
  }
]}
```

Figura 3.6: Ejemplo del formato JSON

## 3.6. GIT

Los sistemas de control de versiones (VCS) son programas que tienen como objetivo controlar los cambios en el desarrollo de cualquier tipo de software, permitiendo conocer el estado actual de un proyecto, los cambios que se le han realizado a cualquiera de sus piezas, las personas que intervinieron en ellos, etc.

Git es un software de control de versiones diseñado por Linus Torvalds, pensando en la eficiencia y la confiabilidad del mantenimiento de versiones de aplicaciones cuando éstas tienen un gran número de archivos de código fuente. La principal diferencia entre Git y cualquier otro VCS 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.

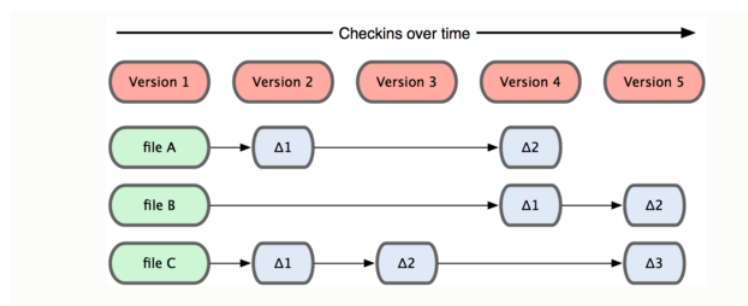


Figura 3.7: Esquema de como organizan la información los demás VCS

Git no modela ni almacena sus datos de este modo. En cambio, Git 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.

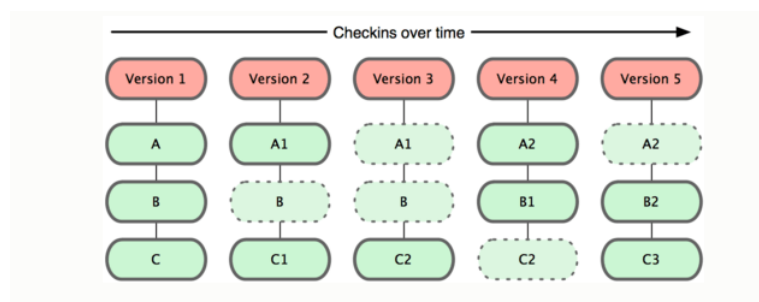


Figura 3.8: Esquema de como git organiza la información

## 3.7. GITFLOW

GitFlow es un modelo de flujo de trabajo para Git que da muchísima importancia a las ramas y que de hecho las crea de varios tipos, de forma temática, tal que cada tipo de rama es creada con un objetivo en concreto.

1. **Master:** La rama master es la única rama existente que nos proporciona Git al crear un repositorio nuevo. Esta rama tiene como objetivo ser el contenido del servidor de producción. Es decir, el HEAD de esta rama ha de apuntar en todo momento a la última versión de nuestro proyecto.

No se va a desarrollar desde esta rama en ningún momento.

2. **Develop:** Esta rama funciona paralelamente a la master. Si la anterior contenía las versiones desplegadas en producción, esta (que también estará sincronizada con origin/develop) contendrá el último estado de nuestro proyecto. Es decir, esta rama contiene todo el desarrollo del proyecto hasta el último commit realizado.

Cuando esta rama adquiera estabilidad y los desarrolladores quieran lanzar una nueva versión, bastará con hacer un merge a la rama master. Esto será lo que cree una nueva versión de nuestro proyecto.

3. **Features:** Que varias personas trabajen sobre la misma rama es bastante caótico ya que se aumenta el número de conflictos que se dan. A pesar de que los repositorios distribuidos faciliten esta tarea al guardar los commits solo localmente, tiene mucho más sentido usar la potencia de las ramas de Git.

Cada vez que necesitemos programar una nueva característica en nuestro proyecto crearemos una nueva rama para la tarea.

Ya que develop contiene la última foto de nuestro proyecto, crearemos la nueva rama a partir de aquí. Una vez finalizada la tarea, solo tendremos que integrar la rama creada dentro de develop. Una vez integrada la rama en develop, podremos eliminarla y actualizar origin.

4. **Release:** Cuando hemos decidido que el código desarrollado hasta ahora pertenece a una versión de nuestro proyecto, y tenemos actualizado la rama develop con dicho código, crearemos una rama release.

5. **HotFix:** Cuando detectamos algún error en producción(rama Master), creamos una rama HotFix, en la cual se arregla rápidamente el error(sin hacer ningún desarrollo más). Cuando tenemos el arreglo preparado integramos esta rama con la rama Master y la Develop para que todos los desarrolladores tengan el error solventado.



Figura 3.9: Diagrama GITFLOW

En el caso de este proyecto, es difícil usar todo el esquema explicado aquí de GITFLOW, debido a que es un proyecto pequeño, es decir, sin distintos entornos de desarrollo(desarrollo, preproducción, producción, etc). Además sólo lo ha desarrollado una persona a la vez. Sin embargo, he visto muy útil la utilización de la rama *feature*, para tener siempre la última versión estable guardada en develop, y aislar el desarrollo de nuevas funcionalidades en éstas ramas. Una vez terminada y probada la nueva funcionalidad, integraba la rama feature con la develop y borraba la feature. Este ha sido el uso que he dado en mi caso a GITFLOW.

Para llevar esto a cabo, he encontrado *Source Tree* es una interfaz visual para utilizar Git. Es especialmente útil cuando queremos utilizar GITFLOW, ya que es un poco lioso de entender con tantas ramas, sobre todo cuando se mantiene todo de forma manual. Con Source Tree podremos ver todo el esquema de ramas que forme nuestro proyecto de forma visual y crear las distintas ramas de forma mucho más fácil.

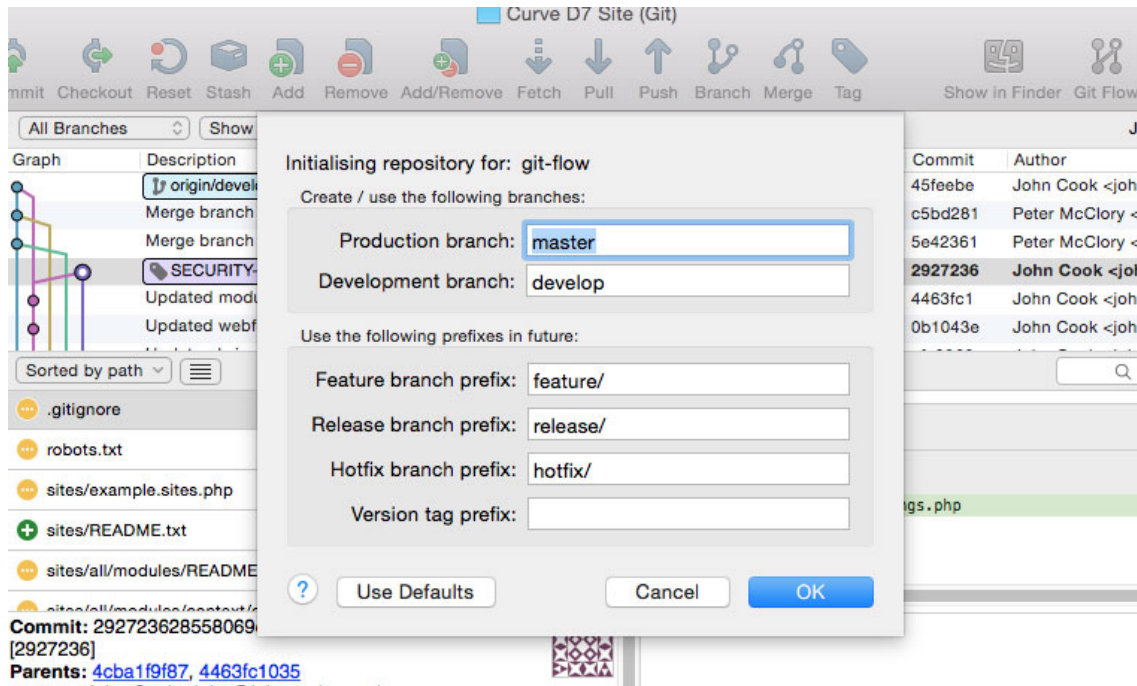


Figura 3.10: Vemos como podemos iniciar el método GITFLOW en nuestro proyecto de manera sencilla.

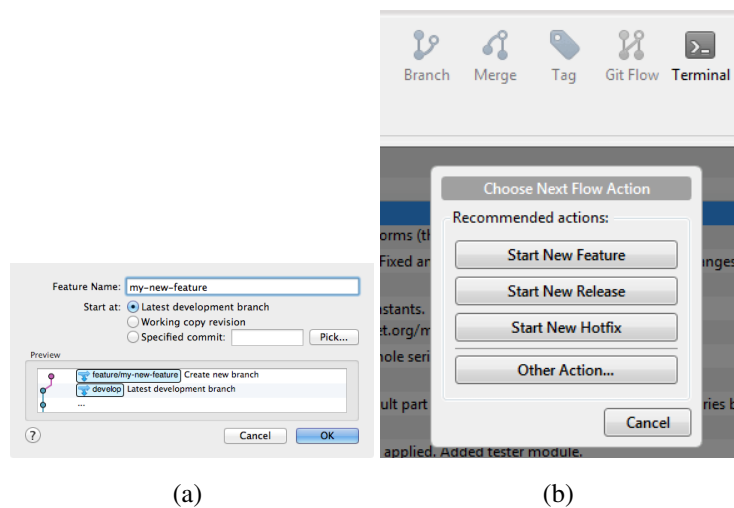


Figura 3.11: Imágenes de ejemplo de uso de Source Tree



# Capítulo 4

## Diseño e implementación

### 4.1. Instalaciones necesarias

Todo lo instalado se ha hecho sobre Windows 10.

#### 4.1.1. NodeJS

Lo primero que necesitamos instalar en este caso es Node en nuestro ordenador. Al igual que muchas aplicaciones, Node tiene su propio MSI(Microsoft Installer), el cual podemos descargar desde la página principal de Node e instalar fácilmente en nuestro ordenador siguiendo todos los pasos.



Figura 4.1: Descarga de NodeJS

### 4.1.2. NPM

Node Package Manager es un gestor de paquetes que nos ayuda a gestionar las dependencias de nuestro proyecto.

Entre otras cosas nos permite instalar librerías o programas de terceros, eliminarlas o mantenerlas actualizadas.

Generalmente se instala conjuntamente con Node.js de forma automática.

NPM se apoya en un fichero llamado `package.json` para guardar el estado de las librerías. Lanzamos el siguiente comando para realizar la creación del `package.json` e iniciar su configuración.

```
npm init
```

#### ■ NPM - package.json

```
{
  "name": "myapp",
  "version": "1.0.0",
  "description": "Esta es la descripción",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Javier Miguel",
  "license": "ISC"
}
```

Figura 4.2: Formato del `package.json` después de lanzar el comando `npm init`

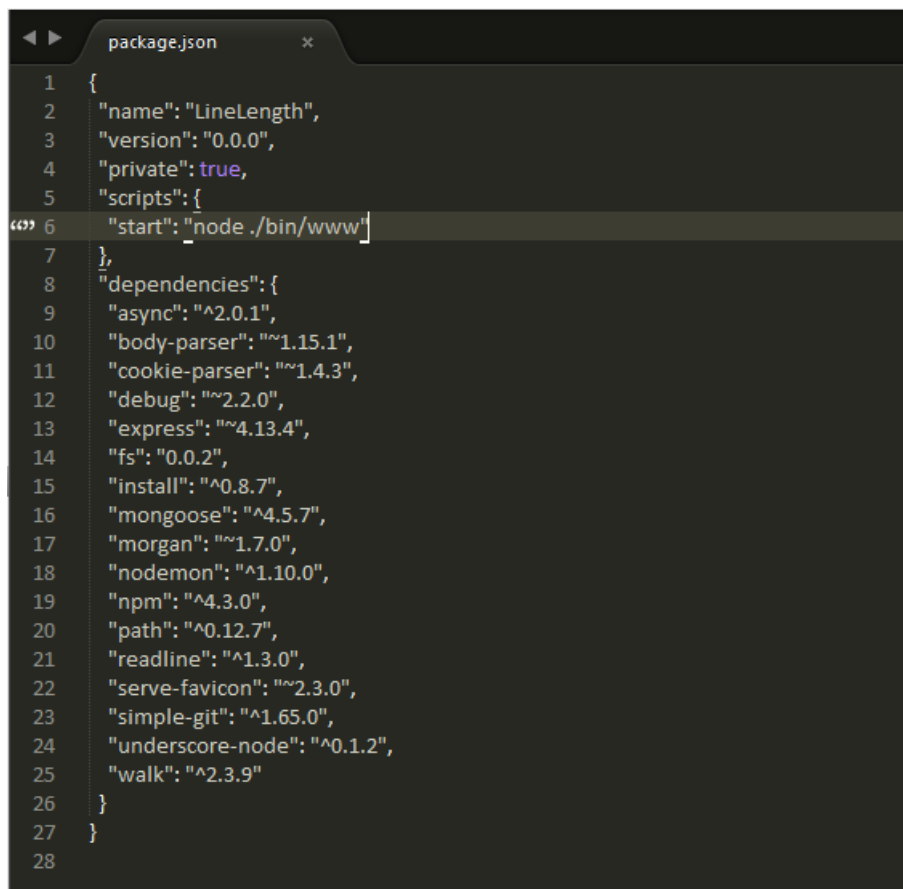
Una vez tenemos el `package.json`, cada vez que queramos incluir una librería nueva a nuestro proyecto, debemos lanzar el siguiente comando:

Por ejemplo, en este caso, si queremos incluir la librería de Express, explicada en el apartado 3, lanzaríamos el siguiente comando.

```
npm install express --save
```

A medida que vayamos necesitando incluir más librerías a nuestro proyecto, iremos ejecutando este comando con cada una de ellas.

Cuando tengamos todas las librerías añadidas a nuestro proyecto, nuestro `package.json` tendrá un formato parecido a este:

A screenshot of a code editor window titled 'package.json'. The editor shows a JSON object with the following structure: { "name": "LineLength", "version": "0.0.0", "private": true, "scripts": { "start": "node ./bin/www" }, "dependencies": { "async": "^2.0.1", "body-parser": "~1.15.1", "cookie-parser": "~1.4.3", "debug": "~2.2.0", "express": "~4.13.4", "fs": "0.0.2", "install": "^0.8.7", "mongoose": "^4.5.7", "morgan": "~1.7.0", "nodemon": "^1.10.0", "npm": "^4.3.0", "path": "^0.12.7", "readline": "^1.3.0", "serve-favicon": "~2.3.0", "simple-git": "^1.65.0", "underscore-node": "^0.1.2", "walk": "^2.3.9" } }. The code is syntax-highlighted, with strings in blue, keywords in purple, and punctuation in white. Line numbers 1 through 28 are visible on the left side of the editor.

```
1 {
2   "name": "LineLength",
3   "version": "0.0.0",
4   "private": true,
5   "scripts": {
6     "start": "node ./bin/www"
7   },
8   "dependencies": {
9     "async": "^2.0.1",
10    "body-parser": "~1.15.1",
11    "cookie-parser": "~1.4.3",
12    "debug": "~2.2.0",
13    "express": "~4.13.4",
14    "fs": "0.0.2",
15    "install": "^0.8.7",
16    "mongoose": "^4.5.7",
17    "morgan": "~1.7.0",
18    "nodemon": "^1.10.0",
19    "npm": "^4.3.0",
20    "path": "^0.12.7",
21    "readline": "^1.3.0",
22    "serve-favicon": "~2.3.0",
23    "simple-git": "^1.65.0",
24    "underscore-node": "^0.1.2",
25    "walk": "^2.3.9"
26  }
27 }
28
```

Figura 4.3: Formato nuestro `package.json`

## 4.2. Nuevo proyecto

Una vez tenemos el equipo listo, pasamos a crear un nuevo proyecto. En el punto 3, hemos explicado en que consiste Express, el framework de node.js con el que vamos a realizar nuestro proyecto.

Dentro de Express vamos a utilizar la librería de Express Generator, la cual nos crea una estructura base para una aplicación.

Para ello ejecutamos en el directorio en el que queramos crear nuestro proyecto los siguientes comandos:

Primero instalamos express-generator de forma global, añadiéndole el -g

```
npm install express-generator -g
```

Seguidamente lanzamos:

```
express <nombreApp> [--ejs]
```

```
cd <nombreApp>
```

```
npm install
```

Con esto se nos creará un proyecto con la siguiente forma:

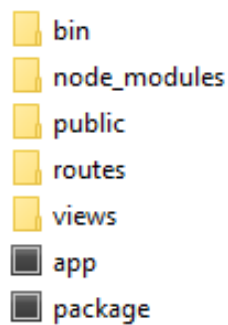


Figura 4.4: Estructura básica de un proyecto

Dentro de la carpeta routes meteremos el código javascript que ejecutaremos en la parte del servidor, y que tendrá toda la lógica de nuestro proceso.

### 4.3. Análisis del estudio realizado

En el siguiente diagrama de flujo explicamos el funcionamiento de nuestro estudio sobre el problema que estamos tratando en este proyecto:

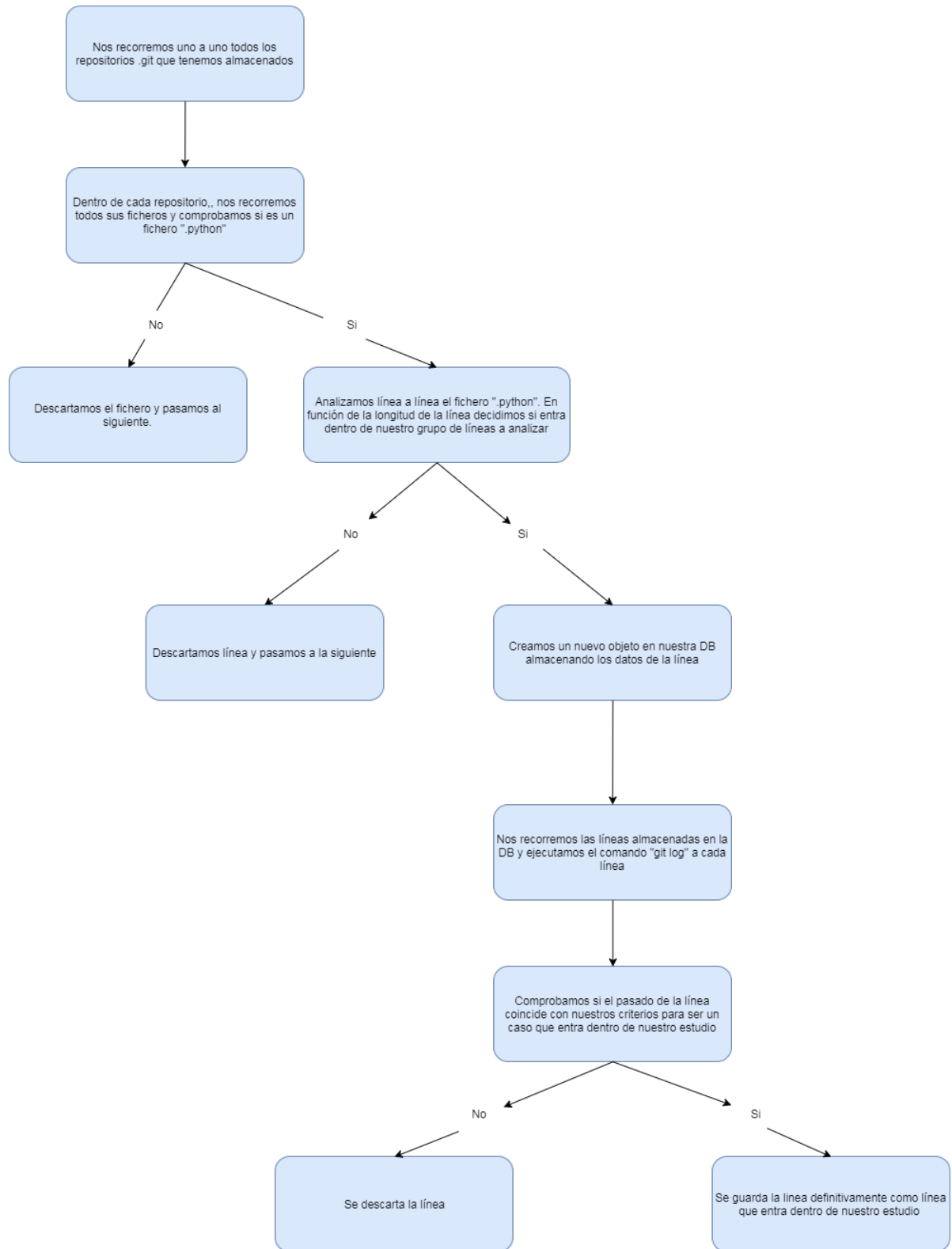


Figura 4.5: Diagrama de flujo del funcionamiento del estudio

### 4.3.1. Análisis de los repositorios

Lo primero que hacemos es analizar la serie de repositorios que tenemos preparados para realizar el estudio. Cuando nos encontremos con un fichero Python y analicemos sus líneas, las dividiremos en tres grupos, dependiendo de la longitud que poseen en la actualidad:

- Grupo 1: “Zona verde“, la línea posee menos de 70 caracteres de longitud.
- Grupo 2: “Zona naranja“, la línea posee entre 70 y 79 caracteres de longitud. Este grupo de líneas es el que nos va a interesar, ya que tienen la posibilidad de haber estado en la ?Zona roja? (veremos en el siguiente grupo) en algún momento de su historia, y en la actualidad estar en la zona naranja, por un cambio de una variable..
- Grupo 3: “Zona roja“, la línea posee más de 80 caracteres de longitud. Este grupo de líneas directamente no cumpliría con las reglas de estilo del pep8.

Cuando nos encontramos una línea que pertenece a la zona naranja, lo que haremos es crear un nuevo objeto en nuestra base de datos en nuestro esquema Linea, el objeto que guardaremos tiene el siguiente aspecto:

```
var lineaSchema = mongoose.Schema({  
  
  linea: String,  
  fichero: String,  
  numLinea: String,  
  longitudLinea: String,  
  pasadoRojo: Boolean  
  historia: []  
  
});
```

Donde:

- **línea:** será la línea en sí, su contenido.
- **fichero:** será la ruta donde encontraremos el fichero dentro del repositorio que estemos analizando.
- **numLínea:** será el número que tiene esa línea dentro del fichero.
- **longitudLinea:** será la longitud de la línea.
- **pasadoRojo:** En este punto, siempre setearemos a false este parámetro, ya que todavía no hemos comprobado el pasado de la línea. Lo haremos más adelante.
- **historia:** En este punto, siempre setearemos un array vacío en este parámetro. Al igual que el anterior, realizaremos esta comprobación más adelante. Cuando lo hagamos, lo que guardaremos aquí será las distintas versiones de la línea en su pasado, en las que su longitud era mayor de 80 caracteres. Más adelante veremos con más detalle las diferentes comprobaciones que hacemos para actualizar este parámetro.

Además de esto, llevaremos la cuenta del total de líneas que nos hemos ido encontrando de cada grupo, en el análisis conjunto de todo todos los repositorios, para al final mostrar los datos analizados organizados, y poder sacar una buena conclusión sobre el estudio. De esta forma, al terminar de analizar todos los repositorios crearemos un nuevo objeto en nuestra DB, con el siguiente formato:

```
var repositorioSchema = mongoose.Schema({  
  
  nombreRepo: Array,  
  lineasVerdes: String,  
  lineasRojas: String,  
  lineasNaranjas: String  
  
});
```

Donde:

- **nombreRepo:** Será un array que posea el nombre de todos los repositorios que hemos analizado.
- **lineasVerdes:** Será el número total de líneas pertenecientes a este grupo que hemos encontrado en nuestro análisis.
- **lineasRojas:** Será el número total de líneas pertenecientes a este grupo que hemos encontrado en nuestro análisis
- **lineasNaranjas:** Será el número total de líneas pertenecientes a este grupo que hemos encontrado en nuestro análisis.

#### 4.3.2. Analizar el grupo de líneas seleccionadas para nuestro estudio

Lo primero que hacemos es recorrer todas las líneas que tenemos guardadas en la DB. Como hemos visto anteriormente, en este punto tendremos sólo las líneas que pertenecen al grupo de la “zona naranja“, por tanto, nos recorreremos todas las líneas, para analizarlas.

La primera comprobación que haremos es mirar si la línea es un comentario, por tanto, no estamos ante una línea de código real. Si nos encontramos ante un comentario descartaremos la línea. En el caso contrario, lo que haremos es ejecutar el comando “git log“ a dicha línea para estudiar su pasado.

#### 4.3.3. Ejecución del comando GIT LOG

En primer lugar, vamos a poner un poco en contexto:



En este punto, nosotros lo que buscamos es saber todo el historial de cambios que ha sufrido una línea en concreta dentro de un fichero. Para así poder analizar todas las líneas que tenemos guardadas previamente como posibles líneas conflictivas.

Para llevar a cabo esto, y repasando la funcionalidad que git nos ofrece, pensamos en utilizar el comando “git blame“, blame se suele utilizar en git para depurar errores, sobre todo. Este comando nos permite filtrar por unas determinadas líneas dentro de un fichero, podemos ver el commit que modificó por última vez cada una de las líneas. Sin embargo, esto se nos quedaba un poco escaso para nuestro propósito final, por lo que, terminando de revisar otras funcionalidades de git, y también revisando las distintas librerías javascripts que nos permitieran implementar esto mediante código en nuestro servidor, tomamos la decisión de explotar el comando git log.

En segundo lugar, vamos a ver en que consiste el comando git log, y viendo justo las opciones que nosotros hemos utilizado. Si ejecutamos el comando git log, sin ningún parámetro, por defecto lo que nos devuelve es una lista de todos los commits realizados en ese repositorio en orden cronológico inverso. Es decir, muestra primero los commits más recientes. Como podemos ver en la siguiente imagen, el formato en que git log nos devuelve su respuesta es el siguiente:

```
$ git log
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test code

commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 10:31:28 2008 -0700

    first commit
```

Figura 4.6: Salida por defecto del comando git log

Lo que nos muestra cómo podemos ver es cada commit con su suma de comprobación SHA-1, el nombre y correo del autor, la fecha el mensaje que el autor ha escrito para ese commit.

El comando `git log` posee múltiples opciones para ajustar la búsqueda que realmente buscamos, la opción que nosotros utilizamos fue la siguiente:

```
git log -L numeroLinea,numeroLinea:nombreFichero
```

De esta forma obtendremos la lista de todos los cambios que ha sufrido la línea que indiquemos desde su creación en el fichero hasta la actualidad.

Podemos ver en el siguiente ejemplo, a salida del comando que estamos tratando en nuestro código:

```
git log -L 155,155:git-web--browse.sh
```

```
$ git log --pretty=short -u -L 155,155:git-web--browse.sh
commit 81f42f11496b9117273939c98d270af273c8a463
Author: Giuseppe Bilotta <giuseppe.bilotta@gmail.com>

    web--browse: support opera, seamonkey and elinks

diff --git a/git-web--browse.sh b/git-web--browse.sh
--- a/git-web--browse.sh
+++ b/git-web--browse.sh
@@ -143,1 +143,1 @@
-firefox|iceweasel)
+firefox|iceweasel|seamonkey|iceape)

commit a180055a47c6793eaaba6289f623cff32644215b
Author: Giuseppe Bilotta <giuseppe.bilotta@gmail.com>

    web--browse: coding style

diff --git a/git-web--browse.sh b/git-web--browse.sh
--- a/git-web--browse.sh
+++ b/git-web--browse.sh
@@ -142,1 +142,1 @@
-    firefox|iceweasel)
+firefox|iceweasel)

commit 5884f1fe96b33d9666a78e660042b1e3e5f9f4d9
Author: Christian Couder <chriscool@tuxfamily.org>

    Rename 'git-help--browse.sh' to 'git-web--browse.sh'.

diff --git a/git-web--browse.sh b/git-web--browse.sh
--- /dev/null
+++ b/git-web--browse.sh
@@ -0,0 +127,1 @@
+    firefox|iceweasel)
```

Figura 4.7: Salida del comando `git log` con opciones

Analizando esta salida podemos ver que la línea 155 del fichero `git-web-browse.sh` ha sufrido los siguientes cambios.

1. Cuando la línea se incluyó dentro del fichero por primera vez en un commit fue como:

```
firefox|iceweasel)
```

2. El primer cambio que sufrió fue borrar las tabulaciones del principio:

```
firefox|iceweasel)
```

3. Y por último se le añadió texto:

```
firefox|iceweasel|seamonkey|iceape)
```

Para poder llevar todo esto a cabo en nuestro código hemos utilizado la librería de javascript “simple-git”<sup>1</sup>.

Siguiendo ahora con el análisis de nuestro proceso, recordamos que tenemos guardadas en nuestra BD las líneas que hemos detectado como posibles conflictivas, y que queremos analizar, guardadas con el siguiente formato:

```
var lineaSchema = mongoose.Schema({  
  
  linea: String,  
  fichero: String,  
  numLinea: String,  
  longitudLinea: String,  
  pasadoRojo: Boolean  
  historia: []  
  
});
```

---

<sup>1</sup><https://www.npmjs.com/package/simple-git>

Lo que hacemos en este punto es ejecutar el comando git log (con las opciones que acabamos de explicar), y en el caso de que nuestra línea haya tenido en algún momento de su historia una longitud mayor a 80, modificaremos su registro en la base de datos cambiando el valor del parámetro “pasadoRojo” a true y también nos guardaremos todo el pasado de la línea para analizarlo.

#### 4.3.4. Análisis del pasado de una línea

Como hemos ido explicando durante esta memoria, lo que nosotros estamos buscando exactamente en este punto, es la iteración en la que la línea ha pasado de estar en la zona roja (mayor de 80 caracteres) a estar en la zona naranja (entre 70 y 79 caracteres).

Esta iteración se puede dar muchas veces y aun así no ser el caso que incumbe a nuestro estudio, por tanto, debemos analizar el pasado de la línea para comprobar que en ese cambio, la línea actual y la línea en el pasado son suficientemente parecidas como para que el cambio pueda deberse a nuestro motivo de estudio, es decir, que la línea actual haya pasado de tener más de 80 caracteres, a tener entre 70 y 79, y que el cambio haya sido acortar una el nombre de una variable.

Como decimos, en este punto todavía no podemos saber si la línea ha sufrido ese cambio por nuestro motivo, sin embargo, lo que si que podemos filtrar es que la línea es suficientemente parecida a su pasado como para que pueda darse, y para ello vamos a utilizar la distancia de levenshtein (meter un índice abajo donde lo explique, buscarlo y añadirlo a bibliografía).

Teníamos guardadas todo el pasado de la línea, lo que haremos es comparar cada estado de la línea en su pasado, con el estado actual, y basándonos en la respuesta que nos devuelva la distancia de levenshtein tomar una decisión. Si la respuesta está dentro de nuestro umbral, nos quedaremos con ese pasado de la línea?, en caso contrario, lo desecharemos.

Al terminar este proceso tendremos un array con todos los estados de la línea en su pasado que han sido mayores de 80 y que se parecen lo suficiente a la línea actual como para ser nuestro caso. (En este array normalmente sólo tendremos un registro, que será el paso anterior a cómo era la línea justo antes de cómo es en la actualidad. En caso de que más estados de la línea en su pasado entren en estos criterios y se cuelen aquí, solo interesaría el último caso en que la línea ha estado en la zona roja antes de pasar a la naranja, por tanto, más adelante los filtraremos).

En este punto volvemos a actualizar el objeto que tenemos para dicha línea guardado en nuestra DB, en este caso modificaremos el parámetro “historia:[]”. Dándole como valor el array que comentamos en el párrafo anterior.

Una vez hecho todo esto, limpiaremos nuestra DB, de forma que borraremos todos los registros de las líneas que no hayan cumplido alguna de estas comprobaciones.

#### **4.3.5. Clasificación de líneas conflictivas**

En este punto, tenemos en nuestra DB únicamente las líneas que consideramos conflictivas para nuestro caso de estudio, después de pasar por todas nuestras comprobaciones. Ahora lo que necesitamos es clasificar estas líneas según creamos cual es la razón por la que esa línea se ha acortado y ha pasado de nuestra zona roja a nuestra zona naranja.

Dentro de los posibles casos, hemos decidido clasificar las líneas en 4 grupos, (recordamos que siempre comparando la iteración en la que la línea pasa de tener más de 80 caracteres a tener entre 70 y 79):

1. La línea se diferencia por el único cambio de una palabra que ha pasado a ser más corta (caso que buscamos).
2. La línea se diferencia por que se han eliminado 4 espacios (tabulaciones).
3. La línea se diferencia porque se han eliminado 4 espacios (tabulaciones), y además ha cambiado una sola palabra, la cual se ha acortado (posible caso buscado).
4. Resto de palabras.

Hemos elegido estos 4 casos porque son los más comunes, por ejemplo, el caso de las tabulaciones, y los casos que afectan directamente a nuestro estudio, como es el 1 principalmente, y como puede ser el 3. De esta forma ya tenemos los resultados obtenidos por nuestro estudio.

## **Capítulo 5**

### **Resultados**





# Capítulo 6

## Conclusiones

### 6.1. Consecución de objetivos

Esta sección es la sección espejo de las dos primeras del capítulo de objetivos, donde se planteaba el objetivo general y se elaboraban los específicos.

Es aquí donde hay que debatir qué se ha conseguido y qué no. Cuando algo no se ha conseguido, se ha de justificar, en términos de qué problemas se han encontrado y qué medidas se han tomado para mitigar esos problemas.

### 6.2. Aplicación de lo aprendido

Aquí viene lo que has aprendido durante el Grado/Máster y que has aplicado en el TFG/TFM. Una buena idea es poner las asignaturas más relacionadas y comentar en un párrafo los conocimientos y habilidades puestos en práctica.

1. a

2. b

### 6.3. Lecciones aprendidas

Aquí viene lo que has aprendido en el Trabajo Fin de Grado/Máster.

1. a

2. b

## **6.4. Trabajos futuros**

Ningún software se termina, así que aquí vienen ideas y funcionalidades que estaría bien tener implementadas en el futuro.

Es un apartado que sirve para dar ideas de cara a futuros TFGs/TFM.

## **6.5. Valoración personal**

Finalmente (y de manera opcional), hay gente que se anima a dar su punto de vista sobre el proyecto, lo que ha aprendido, lo que le gustaría haber aprendido, las tecnologías utilizadas y demás.

## **Apéndice A**

### **Manual de usuario**



# Bibliografía

- [1] E. Bonabeau, M. Dorigo, and G. Theraulaz. *Swarm Intelligence: From Natural to Artificial Systems*. Oxford University Press, Inc., 1999.