UNIVERSIDAD
REY JUAN CARLOS

LINELENGTH:
HERRAMIENTA Y ANÁLISIS DE CUMPLIMIENTO DE LA GUÍA DE ESTILO DE PYTHON EN RELACIÓN CON LA LONGITUD MÁXIMA DE LÍNEA

Curso Académico 2017/2018

Trabajo Fin de Grado

Autor : Kevin Oliva Muñoz

Tutor : Dr. Gregorio Robles

Trabajo Fin de Grado

LINELENGTH: Herramienta y Análisis de Cumplimiento de la Guía de Estilo de Python en Relación con la Longitud Máxima de Línea

Autor : Kevin Oliva Muñoz

Tutor : Dr. Gregorio Robles Martínez

La defensa del presente Trabajo Fin de Grado se realizó el día de de 2017, siendo calificada por el siguiente tribunal:

Presidente:

Secretario:

Vocal:

y habiendo obtenido la siguiente calificación:

Calificación:

Fuenlabrada, a de de 2017

Si tú sabes lo que vales, ve y consigue lo que mereces

pero tendrás que soportar los golpes

y no puedes estar diciendo que no estás donde querías llegar por culpa de él o de ella,

eso lo hacen los cobardes y tú no lo eres.

Tú eres capaz de todo.

Rocky Balboa

II

Agradecimientos

Quiero dar las gracias a mis padres, Gustavo y Yoli. Por darme siempre todo lo que han tenido, por apoyarme en todas mis decisiones, y por creer siempre en mi, incluso cuando yo no lo hacía. En definitiva, por hacerme la persona que hoy soy. Estoy completamente seguro que sin ellos, nada de esto habría sido posible. Os quiero.

Quiero dar las gracias a mi hermana, Sheila. Es mi hermana pequeña, pero en este caso, y aunque ella no lo sabe, siempre ha sido mi ejemplo a seguir, en todos los aspectos. Conseguirás todo lo que te propongas. Te quiero.

No puedo olvidarme de Adrián e Iván, con los que he compartido toda mi vida universitaria y muchas alegrías y tristezas. Nunca olvidaré las horas y horas que hemos compartido en la biblioteca o en el laboratorio, sufriendo juntos, pero sabiendo que seríamos capaces de afrontar cualquier problema que se pusiera por delante.

Gracias a toda mi familia, y en especial a mis abuelos. Gracias a todos, porque sé que os alegráis más que yo por todo lo bueno que me sucede.

Resumen

Los lenguajes de programación suelen tener una guía de estilo que indica, entre otras cuestiones, cómo ha de formatearse el código. Este proyecto pretende realizar un estudio sobre el uso de una de las reglas que contiene la guía de estilo de Python PEP8[1]. 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, ya que –según muchos desarrolladores– seguirla al pie de la letra muchas veces tiene como resultado un peor código. Esto es debido a que, por ejemplo, hay desarrolladores que acortan el nombre de las variables hasta hacerlas ininteligibles. 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.

En el estudio realizado en este Trabajo Fin de Grado intentamos detectar las líneas de código que han sufrido esta mala práctica por parte de los desarrolladores. Para ello, intentamos identificar –a partir de un análisis de repositorios de versiones con código de Python– cuándo se ha cambiado el nombre de alguna variable, acortándolo para cuadrar la longitud de la línea a 79 caracteres. Los resultados obtenidos muestran que el número de ocurrencias en los que se cambia el nombre de una variable para cumplir la regla de longitud máxima de línea es muy bajo. Un análisis manual posterior, además de ofrecer una mayor perspectiva, indica que la incidencia de esta regla –a partir de los datos a los que hemos podido acceder– en la calidad del código es realmente muy pequeña.

Para realizar este estudio se ha implementado el lado del servidor con Node.js (utilizando Express) y el lenguaje JavaScript.

[1] `https://www.python.org/dev/peps/pep-0008/`

Summary

Programming languages usually have a style guide that indicates, among other things, how to format the code. This project aims to conduct a study on the use of one of the rules contained in the Python style guide named PEP8[2].

The rule in question is as follows: "**Maximum line length:** Limit all lines to a maximum of 79 characters". This rule has produced a small debate in the Python community, since –according to many developers– following it often results in worse code. This is because, for example, there are developers who shorten the name of variables to make them unintelligible. If this occurs we face a problem, since the rule of maximum length style is producing an effect contrary to what it intended.

In the study carried out in this Undergraduate Thesis we try to detect the lines of code that have suffered this malpractice. To do this, we try to identify -from an analysis of version repositories with Python code– if a developer has changed the name of a variable, shortening it to meet that the maximum length of the line is 79 characters or less. The results obtained show that the number of occurrences in which a variable name is changed to meet the maximum line length rule is very low. A later manual analysis, besides providing a greater perspective, indicates that the incidence of this rule –from the data that we have been able to access– in the quality of the code is very small.

To perform this study, the server side has been implemented with Node.js (using Express) and the JavaScript language.

[2] `https://www.python.org/dev/peps/pep-0008/`

Índice general

1. Introducción **1**

1.1. Motivación 1

1.2. La Guía de Estilo PEP8 3

1.3. Estructura de la memoria 4

2. Objetivos **7**

2.1. Objetivo principal 7

2.2. Objetivos secundarios 8

3. Estado del arte **11**

3.1. App Cliente-Servidor 11

3.2. Python 12

3.3. `pycodestyle` (antes `pep8`) 13

3.4. Otros programas del estilo de `pycodestyle` 14

3.5. JavaScript 15

3.6. Node JS 16

3.7. Express 17

3.8. MongoDB 18

3.9. JSON 19

3.10. Git 20

3.11. GitFlow 21

4. Diseño e implementación **25**

4.1. Análisis de los repositorios 25

4.2. Análisis del grupo de líneas seleccionadas para nuestro estudio 29

4.3. Ejecución de `git log` 29
4.4. Análisis del pasado de una línea 32
4.5. Clasificación de líneas conflictivas 35

5. Resultados **37**
5.1. Resultados del análisis general 37
5.2. Resultados del análisis del grupo conflictivo 40
5.3. Análisis (manual) de líneas particulares 42

6. Conclusiones **49**
6.1. Consecución de objetivos 49
6.2. Aplicación de lo aprendido 51
6.3. Lecciones aprendidas 52
6.4. Trabajos futuros 52
6.5. Valoración personal 53

A. Instalaciones necesarias **55**
A.1. NodeJS 55
A.2. NPM 56
A.3. Creación de un nuevo proyecto Express 57

Bibliografía **59**

Índice de figuras

3.1. Esquema básico de la estructura cliente-servidor 12
3.2. Python 12
3.3. JavaScript 15
3.4. Node.js 16
3.5. Express.js 17
3.6. MongoDB 18
3.7. Estructura de aplicación MEAN: Mongo + Express + Angular + Node 19
3.8. Ejemplo del formato JSON 20
3.9. Esquema de como organizan la información los demás VCS 21
3.10. Esquema de como `git` organiza la información 21
3.11. Diagrama GitFlow 23
3.12. Inicialización del método `GitFlow` en un proyecto. 24
3.13. Imágenes de ejemplo de uso de Source Tree. 24

4.1. Diagrama de flujo del funcionamiento del estudio 26
4.2. Salida por defecto de la instrucción `git log` 30
4.3. Salida de la instrucción `git log` 31

5.1. Diagrama de barras de los resultados generales. 38

A.1. Descarga de NodeJS 55
A.2. Formato del `package.json` después de lanzar el comando npm init 56
A.3. Formato de nuestro `package.json`. 57
A.4. Estructura básica de un proyecto 58

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, que nunca, o casi nunca, habías pensado en tu vida de estudiante: en tu trabajo pasas más tiempo leyendo código de otra(s) persona(s) 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 me he dado cuenta de la importancia de escribir código de calidad, y no sólo "algo que funcione", mi tutor, Gregorio me habló de un vídeo con una charla en una *PyCon*, el congreso anual más grande sobre el lenguaje Python, en el que Raymond Hettinger, uno de los desarrolladores principales de Python, expone buenas prácticas para escribir un código legible y de calidad[1]. En el vídeo se indica que se ha de seguir unas normas de estilo (que vienen recogidas en la guía de estilo de Python, conocida como PEP8[2], ya que eso facilita la tarea de comprensión del código. Sin embargo, en el vídeo se plantea lo que para mí es una interesante cuestión: ¿Podemos dejar un código peor de lo que estaba al realizar un cambio para cumplir con una regla de la guía de estilo de Python?

[1] `https://www.youtube.com/watch?v=wf-BqAjZb8M`

[2] PEP es el acrónimo de Python Enhancement Proposal, un documento que se propone y debate en la comunidad Python antes de ser definitivamente aceptado como "oficial", que es cuando se le asigna finalmente un número.

Dicha cuestión es la que vamos a estudiar en este TFG. Nos centraremos en un caso muy concreto, a la vez que muy debatido en la comunidad de Python. La regla en cuestión es la siguiente:

"Máxima longitud de las líneas: Limita todas las líneas a un máximo de 79 caracteres"

El caso que nos preocupa es el siguiente: un desarrollador se encuentra ante una línea de código mayor a los 79 caracteres, por lo que decide cambiarla, acortando para ello el nombre de alguna variable. Al hacerlo, el nombre de la variable carece de sentido, por lo que la legibilidad del código se resiente.

Para intentar ilustrar esta cuestión, vamos a poner un pequeño ejemplo. Imaginemos que el desarrollador ha escrito la siguiente línea de código en lenguaje **Python**:

```
'subtitles': self.extract-subtitles(video_id, video_subtitles_id)
```

Si nos encontramos analizando código y leemos esta línea, en principio, se entiende bastante bien, sin necesidad de contar con más contexto. Podemos suponer fácilmente que lo que estamos obteniendo en la variable "subtitles" son los subtítulos de un vídeo, los cuales extraemos llamando a la función "extract-subtitles" pasándole como parámetros el id del vídeo y el id de los subtitulos.

Con el contexto que nos daría el resto de código de este fichero, es posible que no haga falta ir a la documentación de la función "extract-subtitles", porque sabemos todo lo que nos hace falta con esta línea y podremos seguir leyendo el código que estamos analizando sin perder más tiempo.

Esta línea en sí, no ocupa más que 69 caracteres. Pero, supongamos que por motivos de *indentación*[3], esta línea superara los 79 caracteres. El desarrollador ha intentado refactorizar su código pero no consigue quitar las *indentaciones* y como le ha saltado el *warning* de que no cumple la regla de estilo únicamente por la longitud de la línea, decide para no perder más tiempo, acortar el nombre de las variables, y forzando un caso extremo, también el de la función, de tal forma que la línea quedara con menos de 80 caracteres y cumplir con la regla. De forma que la línea resultante podría quedar de la siguiente forma:

[3]Este término *indentación* significa mover un bloque de texto hacia la derecha insertando espacios o tabuladores, para así separarlo del margen izquierdo y mejor distinguirlo del texto adyacente. Es utilizado para mejorar la legibilidad del código fuente por parte de los programadores.

```
'subtitles': self.e_s(v_id, v_s_id)
```

Como se puede ver, la línea ahora es más difícil de explicar de por sí. Si estamos analizando código bien puede pasar que seríamos incapaces de saber cómo estamos obteniendo la variable "subtitles", necesitaríamos irnos a la función para intentar entender qué hace, y qué parámetros le están llegando. La línea ahora cumple las reglas de estilo, pero es ilegible. Hemos ido a peor.

Este hecho me parece digno de estudio, por varios motivos. Uno de ellos, indudablemente, está relacionado con el debate que puede crear en la comunidad Python, sobre si un código que cumple la guía de estilo PEP8 puede ser de peor calidad que uno que no lo cumpla.

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 de la guía de estilo 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 retomamos código propio de hace tiempo. Son cosas simples, pero es importante que desde el principio de nuestra formación se adopten estas buenas costumbres.

1.2. La Guía de Estilo PEP8

Desde que empezamos a escribir código y conforme progresamos, adquirimos ciertas pautas que definen la manera en que expresamos nuestras ideas en el lenguaje que manejemos, ya sea JavaScript, Python, C u otro. Esto define nuestro estilo, y tiene en cuenta la manera en la que *indentamos*, definimos nombres de variables y funciones, o el proceso de realizar o no comentarios de código, entre otros.

¿Por qué es tan importante seguir un estilo cuando estamos programando? El principal y más importante motivo es porque el código debe ser mantenido, ya sea por nosotros mismos o por otras personas. Y a lo largo de nuestra vida como desarrolladores, pasamos muchas más

horas leyendo código que escribiéndolo.

Para esto nacieron las guías de estilo de código. El principal objetivo de una guía de estilo es que nuestro código sea más fácil de leer, compartir y analizar. Facilita la consistencia entre el código fuente de distintos usuarios y hace que nuestro código sea mucho más mantenible.

Las guías de estilo son algo que está generalizado en los distintos lenguajes de programación, por ejemplo para PHP tenemos PSR-2[4], y en JavaScript una de las más notables es Crockford[5]. No hay una sola guía de estilo para cada lenguaje, podemos encontrar varias de distintas fuentes (es muy común que los proyectos de software libre tengan una propia). Algo muy común es consultar las guías que ha creado Google. Por ejemplo, para Java podemos encontrar la siguiente[6].

Esta memoria se centra en la guía de estilo para Python PEP8. La guía de estilo PEP8 fue realizada por Guido van Rossum, Barry Warsaw y Nick Coghlan. Está dedicada a la recopilación de los estándares seguidos por los desarrolladores de Python a la hora de escribir código Python para la librería estándar. Esta guía está conformada por varias convenciones, en las cuales no entraremos en profundidad, destacaremos una de ellas, alrededor de la que se centra este proyecto:“Limitar los tamaños de línea a 79 caracteres como máximo”.

Para aplicar las reglas de calidad que marca PEP8, tenemos a nuestra disposición varias herramientas automáticas para pasarle a nuestro código. En el capítulo 3 se habla más en profundidad sobre una de ellas, esta herramienta es `pycodestyle`, antes conocida como `pep8`.

1.3. Estructura de la memoria

En este apartado del capítulo se explica muy brevemente la estructura de la memoria, dando a conocer los objetivos de cada capítulo para facilitar así la lectura de la misma:

[4]http://www.php-fig.org/psr/psr-2/

[5]http://javascript.crockford.com/code.html

[6]https://google.github.io/styleguide/javaguide.html

1. **Introducción:** En este capítulo se explica el contexto de este proyecto, así como las razones por las que se elige el tema a tratar.

2. **Objetivos:** En este capítulo 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 capítulo 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 del trabajo y de la memoria.

Capítulo 2

Objetivos

2.1. Objetivo principal

Este proyecto tiene como objetivo principal:

> "Analizar repositorios de GitHub con código Python para determinar si los desarrolladores han acortado nombres de variables para cumplir con el requisito de tener una longitud máxima de línea de 79 caracteres, tal y como viene indicado en la guía de estilo de Python PEP8".

El proyecto consistirá en un programa software que analizará código fuente de un programa en el lenguaje de programación Python y detectará aquellas líneas que han sido acortadas cambiando el nombre de una variable por un nombre más corto, y por tanto, menos entendible, para conseguir pasar la regla de la guía de estilo PEP8 de la máxima longitud de línea.

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 se encuentra extendido entre los desarrolladores de Python.

También se pretende alcanzar otro objetivo más a largo alcance, 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 entregar y no vas a volver a ver en tu vida. Este ciclo es lo que suele pasar con el código que creas 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 generalmente se creen malos hábitos por parte de los alumnos a la hora de programar.

Este proyecto de fin de grado intenta llegar a este objetivo creando un "debate" alrededor del caso concreto que se estudia. ¿Es mejor el código que no cumple la guía de estilo PEP8 antes de realizar el cambio, pero que tiene un buen nombre de variable?, ¿o es mejor el código después del cambio que cumple la guía de estilo 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 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 la guía de estilo 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 del objetivo explicado anteriormente, a la hora de realizar este proyecto, se han tenido en cuenta los siguientes objetivos secundarios:

- **Trabajar de una forma orientada al mundo profesional:** 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.

- **Aplicar tecnologías e ideas no vistas durante la vida universitaria:** 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 conozco en profundidad, ya que o no se han enseñado, o se ha pasado por ellas de "refilón" en mi vida universitaria. De esta forma lo que busco es ampliar mis conocimientos y aprovechar esto para conocer 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 un servidor JavaScript, Node.js con Express. Mi elección se debe a que en la universidad se ha enseñado a montar servidores en lenguajes como Java, y sobre todo, con Python y Django. 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 cuál 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.

Capítulo 3

Estado del arte

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

3.1. App Cliente-Servidor

La arquitectura *cliente-servidor* es una de las más extendidas en la actualidad [1]. 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** (véase figura 3.1). Un cliente realiza peticiones a otro programa, y es el servidor quien atiende su petición y le da respuesta:

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, y que potencialmente sea capaz de manejar de forma distinta las peticiones dependiendo de quién las realice.

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 un 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. 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.

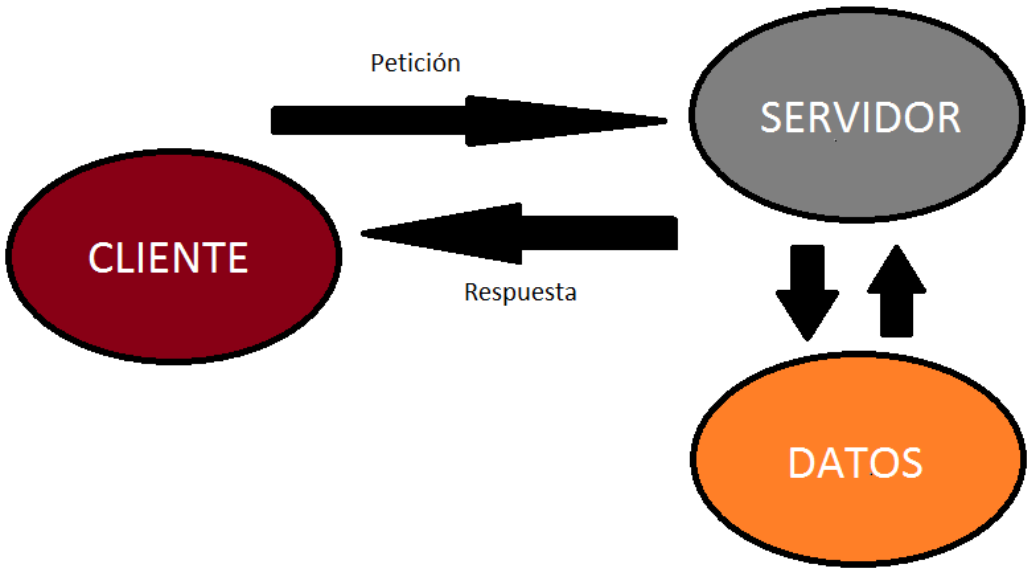


Figura 3.1: Esquema básico de la estructura cliente-servidor

3.2. Python

Figura 3.2: Python

Python es un lenguaje de programación de muy alto nivel, multiplataforma y multiparadigma, el cual, fue creado a finales de los años 80 por el holandés Guido van Rossum, Python fue creado como sucesor del lenguaje de programación ABC [2]. El logo de Python se puede ver en la figura 3.2.

¿Qué quiere decir que es un lenguaje multiplataforma? Quiere decir que es un lenguaje interpretado, y por tanto no necesita compilación. Esto tiene como ventaja, por ejemplo, que da igual el sistema operativo donde se ejecute, o que aporta mayor rapidez al desarrollarlo.

¿Qué quiere decir que es multiparadigma? Esto quiere decir que no obliga al desarrollador a adaptarse a un estilo de programación concreto, Python soporta programación funcional, programación imperativa y programación orientada a objetos.

Python se caracteriza por tener una sintaxis limpia y ordenada, haciendo mucho hincapié en la

legibilidad del código, a pesar de ello, tiene funcionalidades muy avanzadas, propias de lenguajes como C o C++.
Python es un lenguaje que ha adquirido gran popularidad en los últimos años, algunas de las razones por las que esto ha sucedido son las siguientes:

- Sencillez y velocidad a la hora de crear desarrollos completos, normalmente un programa en Python tendrá muchas menos líneas que sus equivalente en otros lenguajes (java, C, etc).

- Ofrece una gran cantidad de librerías, con funciones incorporadas, funcionalidades o tipos de datos, que favorecen la realización de muchas tareas sin necesidad de empezar completamente de cero.

- Se puede obtener de forma gratuita y utilizarse con fines comerciales.

3.3. `pycodestyle` (antes `pep8`)

Como ya hemos dicho anteriormente cuando hablábamos de PEP8, existen herramientas con las que podemos comprobar de forma automática si nuestro código cumple las reglas que dicta el PEP8. Aquí vamos a hablar un poco más en profundidad de una de ellas: "`pycodestyle`", antes conocida como "PEP8". Se cambió de nombre para evitar confusiones con la propia guía de estilo.

Este herramienta nos permite pasar un análisis una vez nuestro código está terminado, para ver si cumple calidad. Podemos encontrar información sobre esta herramienta en GitHub[1]. Básicamente, la forma de utilizarla y lo que nos aporta es lo siguiente:

Podemos instalar, actualizar o desinstalar la herramienta de forma fácil con los siguientes comandos:

```
$ pip install pycodestyle
$ pip install --upgrade pycodestyle
$ pip uninstall pycodestyle
```

[1] `https://github.com/PyCQA/pycodestyle`

Una vez instalado podemos revisar nuestro código de forma sencilla, por ejemplo, con el siguiente comando, analizamos el código fuente del fichero “optparse.py”:

```
$ pycodestyle --first optparse.py
```

La salida que `pycodestyle` nos proporciona sería la siguiente:

```
optparse.py:69:11: E401 multiple imports on one line
optparse.py:77:1: E302 expected 2 blank lines, found 1
optparse.py:88:5: E301 expected 1 blank line, found 0
optparse.py:222:34: W602 deprecated form of raising exception
optparse.py:347:31: E211 whitespace before &#39;(&#39;
optparse.py:357:17: E201 whitespace after &#39;{&#39;
optparse.py:425:80: E501 line too long (82 characters)
optparse.py:472:29: E221 multiple spaces before operator
optparse.py:544:21: W601 .has_key() is deprecated, use &#39;in&#39;
```

Como vemos, `pycodestyle` nos muestra de manera clara la línea y el sitio justo donde estamos cometiendo el error de calidad y por el cual no cumplimos la norma correspondiente de la guía PEP8.

Podemos ver por ejemplo como en la siguiente línea:

```
optparse.py:425:80: E501 line too long (82 characters)
```

se nos muestra el mensaje del error que un programador verá cuando una línea supera el máximo tamaño de línea permitido. Esta herramienta detecta el error, pero es el desarrollador el que debe tomar tomar le decisión de cómo solucionarlo.

3.4. Otros programas del estilo de `pycodestyle`

A parte del actual `pycodestyle`, existen otras herramientas parecidas que también nos sirven para analizar la calidad de nuestro código Python. Entre ellas cabe destacar `pylint` y `pyflakes`.

- **`pyflakes:`** Es un programa sencillo que comprueba los errores de los archivos fuente de Python. Funciona analizando el archivo de origen, no importándolo, por lo que es seguro utilizar en módulos con efectos secundarios. También es mucho más rápido.

- **`pylint:`** Es otro comprobador de código fuente para Python. Sigue el estilo recomendado por PEP 8. Es similar a `pychecker` y `pyflakes`, pero incluye las siguientes características: comprobación de la longitud de cada línea, comprobar si los nombres de las variables están bien formados de acuerdo con la norma de codificación del proyecto, comprobación de si las interfaces declaradas se implementan realmente.

3.5. JavaScript

Figura 3.3: JavaScript

JavaScript es un lenguaje orientado a objetos y basado en prototipos [3]. Una de sus características más importantes es que se define como un lenguaje asíncrono, por lo que es muy útil para reaccionar a eventos por parte del usuario. Es un lenguaje interpretado, por lo que no es necesario compilar los programas para ejecutarlos. En otras palabras, los programas escritos con JavaScript se pueden probar directamente en cualquier navegador sin necesidad de procesos intermedios. En la figura 3.3 se puede ver el logo de JavaScript.

Se utiliza principalmente en su forma del lado del cliente (*client-side*), implementado como parte de un navegador web permitiendo mejoras en la interfaz de usuario y páginas web dinámicas, una página web dinámica es aquella que incorpora efectos como texto que aparece y desaparece, animaciones, acciones que se activan al pulsar botones y ventanas con mensajes de aviso al usuario.

Sin embargo, existe una forma de JavaScript del lado del servidor(*Server-side JavaScript* o SSJS), como veremos más adelante en esta memoria cuando profundicemos en Node.js.

3.6. Node JS

Figura 3.4: Node.js

Node.js [4, 5] 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. El logo de Node.js se muestra en la figura 3.4.

Node.js ejecuta JavaScript utilizando el motor V8, desarrollado por Google. Este motor permite a Node.js 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 total, 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 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.js 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.js. Node.js también afirma que nunca se quedará en punto muerto, porque no se permiten bloqueos.

3.7. Express

Figura 3.5: Express.js

Express [6] es sin duda el framework más conocido de Node.js, es una extensión del poderoso connect[2] y esta inspirado en sinatra[3], además es robusto, rápido, flexible, simple, etc. [7]. 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. El logo de Express se puede ver en la figura 3.5.

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

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

[2] `https://github.com/senchalabs/connect#readme`
[3] `http://www.sinatrarb.com/documentation.html`

3.8. MongoDB

Figura 3.6: MongoDB

MongoDB [8] 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 [9]. El logo de MongoDB se muestra en la figura 3.6. 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.

La arquitectura completa de las tecnologías utilizadas en este proyecto se puede ver en la figura 3.7.

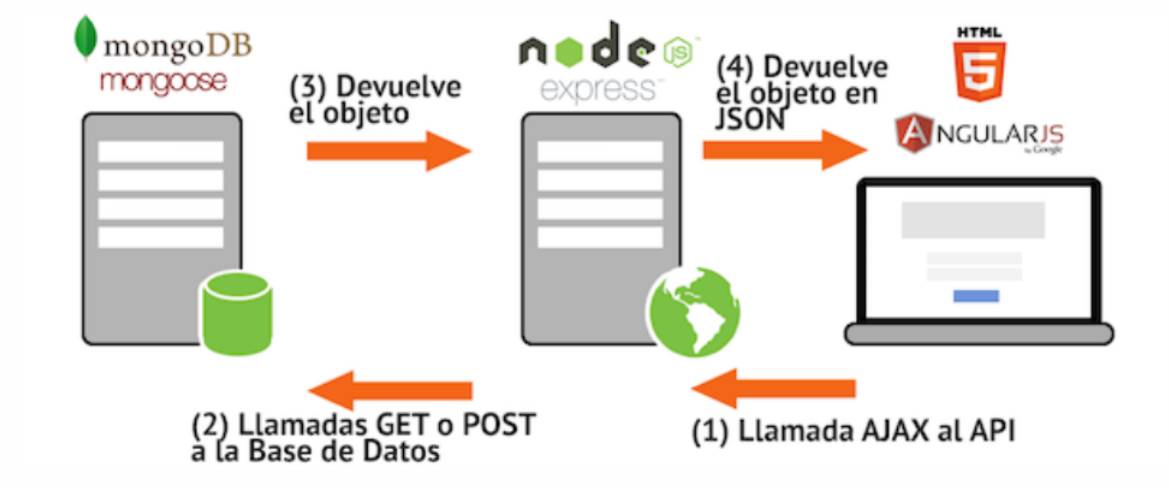


Figura 3.7: Estructura de aplicación MEAN: Mongo + Express + Angular + Node

3.9. JSON

JSON (JavaScript Object Notation) [10] es un formato de texto ligero para el intercambio de datos [11]. 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. Se puede ver un ejemplo de una estructura de datos en JSON en la figura 3.8.

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, que 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 3.8: Ejemplo del formato JSON

3.10. 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` [12] 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 [13]. 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, como se puede ver en la figura 3.9.

Como se puede ver en la figura 3.10, `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

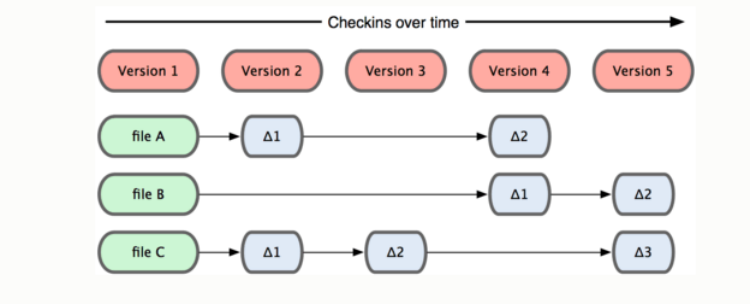


Figura 3.9: Esquema de como organizan la información los demás VCS

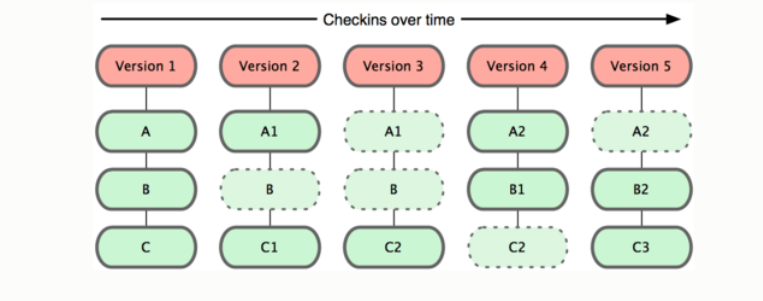


Figura 3.10: Esquema de como `git` organiza la información

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.

3.11. GitFlow

GitFlow [14] 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 [15], como se puede ver en la figura 3.11:

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 que 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.

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), y no ha contado con varios desarrolladores de forma simultánea. 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

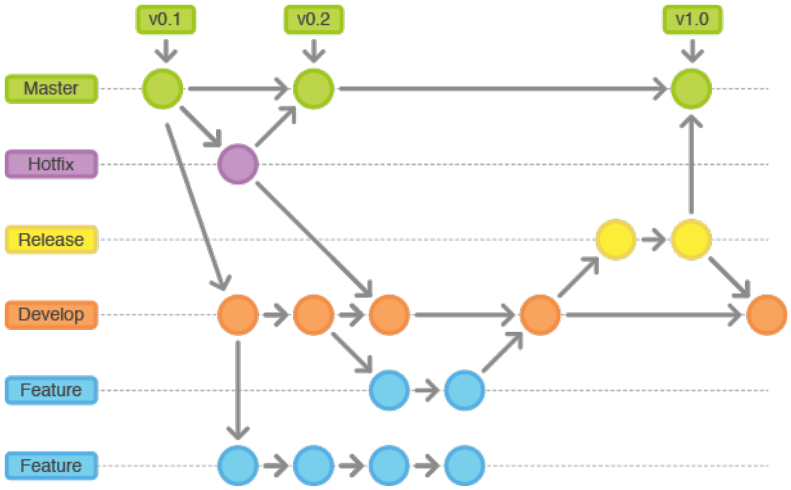


Figura 3.11: Diagrama GitFlow

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* (véanse la figura 3.12 y la figura 3.13), que 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.

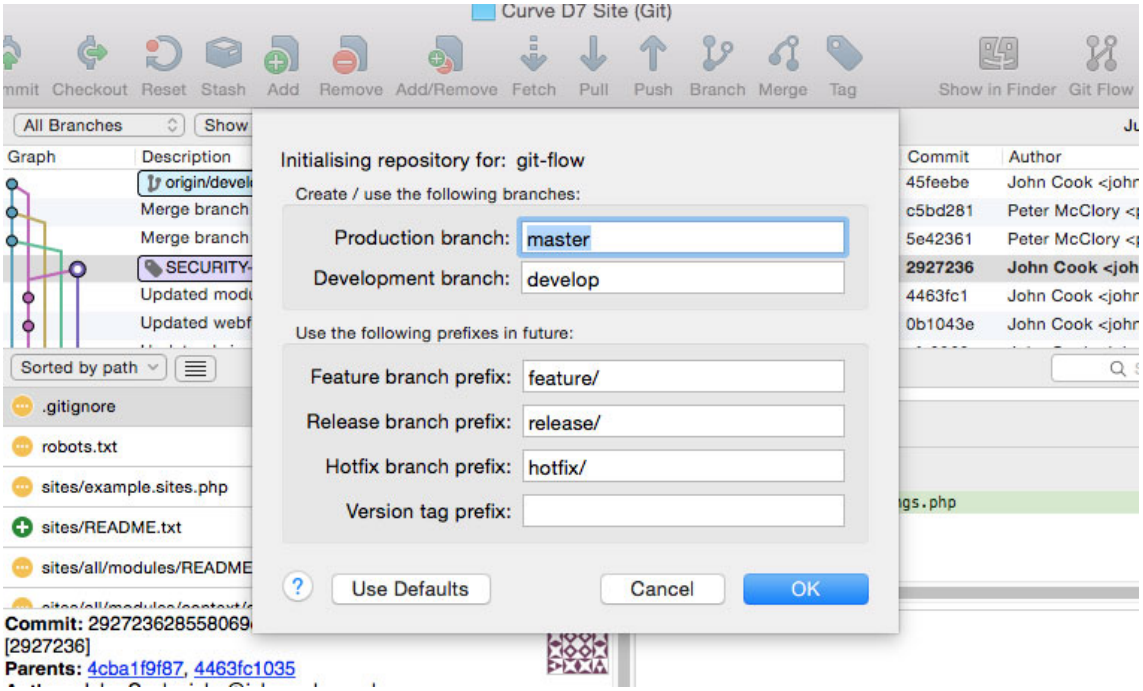
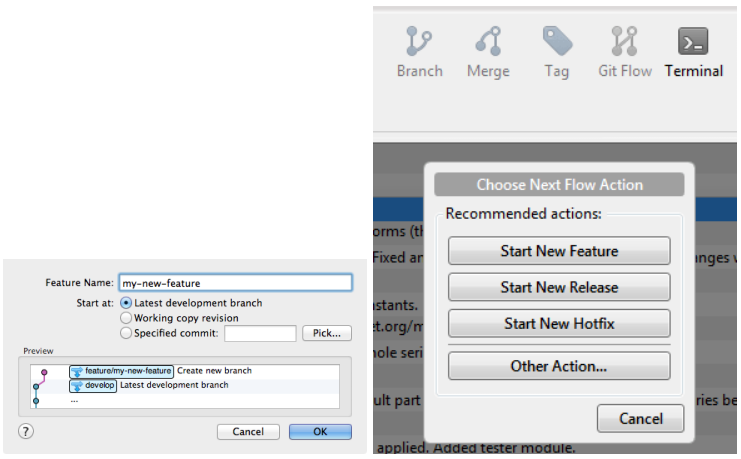


Figura 3.12: Inicialización del método `GitFlow` en un proyecto.



(a) (b)

Figura 3.13: Imágenes de ejemplo de uso de Source Tree.

Capítulo 4

Diseño e implementación

En el diagrama de flujo de la figura 4.1 se muestra el funcionamiento de nuestro estudio sobre el problema que estamos tratando en este proyecto. En las siguientes secciones se entrará en detalle sobre cada uno de los pasos.

4.1. Análisis de los repositorios

Para la realización de este proyecto contaremos con una batería de repositorios almacenados previamente, los cuales serán la muestra de este estudio.

En primer lugar se analizan dichos repositorios. Si se detecta un fichero python dentro del repositorio que se está analizando, se procesa línea por línea, y en función de la longitud que posean se dividen en tres grupos:

- 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.
- Grupo 3: "Zona roja", la línea posee 80 o más caracteres de longitud. Este grupo de líneas no cumple con las reglas de estilo PEP8.

El Grupo 2: "Zona naranja", es el grupo determinante para el estudio. Las líneas pertenecientes a este grupo tienen la posibilidad de haber pertenecido al Grupo 3: "Zona roja" en algún momento de su "pasado", y por tanto, tienen la posibilidad de estar, en la actualidad, en la "Zona naranja" debido a un cambio motivado por la modificación de una variable.

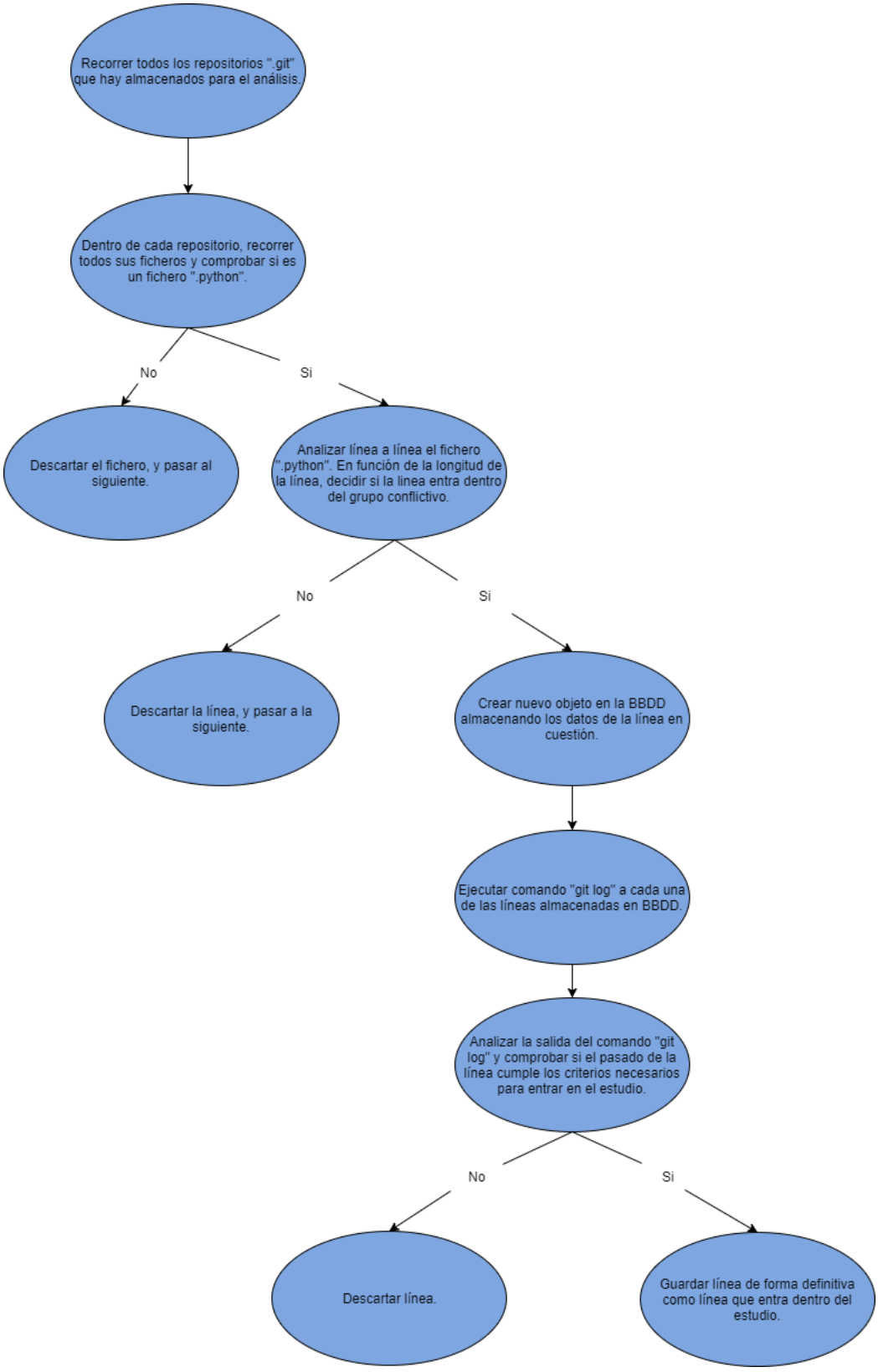


Figura 4.1: Diagrama de flujo del funcionamiento del estudio

Como se ha explicado antes, las líneas procesadas se dividen en 3 grupos, en el momento en que se detecta una línea perteneciente al grupo "Zona naranja", se crea un nuevo objeto en la BBDD, en este caso, dentro del esquema Linea. El objeto guardado tiene el siguiente formato:

```
var lineaSchema = mongoose.Schema({

linea: String,
fichero: String,
numLinea: String,
longitudLinea: String,
pasadoRojo: Boolean
historia: []

});
```

Donde cada parámetro contiene:

- **línea:** Contenido de la línea.
- **fichero:** Ruta del fichero dentro del repositorio git.
- **numLínea:** Número de la línea dentro del fichero.
- **longitudLinea:** Longitud de la línea.
- **pasadoRojo:** Parámetro *boolean*. Este parámetro indica si la línea en cuestión ha pertenecido en algún momento de su pasado al grupo "Zona roja". En la creación de este objeto en este punto del análisis siempre se setea este parámetro a *False*, debido a que esta cuestión todavía no se ha comprobado. Se hará más adelante.
- **historia:** Parámetro *array*. Este parámetro almacena los diferentes estados que ha tenido la línea a lo largo de su historia. Al igual que ocurre con el parámetro anterior, en la creación del objeto en este punto, este array se inicializa y guarda vacío, debido también a que en este punto no se ha realizado todavía esta comprobación. Más adelante se explica como se procesa y se actualiza este parámetro.

Cuando se finaliza este primer proceso, además del objeto perteneciente al esquema Línea, que acabamos de contar, se dispone de un recuento total de las líneas analizadas, así como un resumen de toda la muestra utilizada indicando por cada repositorio, su nombre y líneas analizadas de cada tipo. De esta forma se tiene disponible, de manera organizada, toda la información de los datos analizados. De manera que resulte más fácil llegar a una conclusión en el futuro. Para ello se crea un nuevo objeto en la BBDD, dentro del esquema Repositorios, con el siguiente formato:

```
var repositorioSchema = mongoose.Schema({

nombreRepo: Array,
lineasVerdes: String,
lineasRojas: String,
lineasNaranjas: String

});
```

Donde cada parámetro contiene:

- **nombreRepo:** Parámetro *array*. Contiene una serie de objetos, los cuales poseen los parámetros: nombre del repositorio, líneas verdes analizadas en dicho repositorio, líneas naranjas analizadas en dicho repositorio, líneas rojas analizadas en dicho repositorio.
- **lineasVerdes:** Número de líneas pertenecientes a este grupo encontradas en el total del análisis.
- **lineasRojas:** Número de líneas pertenecientes a este grupo encontradas en el total del análisis.
- **lineasNaranjas:** Número de líneas pertenecientes a este grupo encontradas en el total del análisis.

4.2. Análisis del grupo de líneas seleccionadas para nuestro estudio

Cabe destacar que en este punto ya se han descartado las líneas que consideramos comentarios, en el proceso de clasificación de las líneas en los 3 grupos mencionados. De forma que en la BBDD se cuenta únicamente con las líneas que consideramos conflictivas, es decir, las pertenecientes al Grupo 2: "Zona naranja", y que se deben seguir analizando.

Posteriormente, se procesan una a una todas las líneas almacenadas dentro de la BBDD y se ejecuta la instrucción `git log` para cada una de ellas, para así poder estudiar su pasado.

4.3. Ejecución de `git log`

En primer lugar, vamos a abrir un pequeño paréntesis para poder poner algo de contexto sobre la instrucción `git log` y la información que se necesita en este aquí. Lo que se busca en este proyecto en concreto es saber todo el historial de cambios que ha sufrido una determinada línea dentro de un fichero.

Para llevar a cabo esto, en un principio se estudió la posibilidad de utilizar el comando `git blame`, el cual aporta información sobre la historia de una línea. `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, la información que proporciona `blame` se nos queda un poco escasa para el propósito de este proyecto, por lo que se decide seguir investigando sobre la funcionalidad que ofrecen las distintas instrucciones de git, y revisar las distintas librerías JavaScript que nos permitan implementar esto mediante código en nuestro servidor, hasta tener conocimientos suficientes para tomar la siguiente decisión: explotar la instrucción `git log`.

La ejecución de la instrucción `git log` sin ningún parámetro, devuelve por defecto una lista de todos los `commits` realizados en ese repositorio en orden cronológico inverso. Es decir, muestra primero los `commits` más recientes. En la figura 4.2 podemos observar el formato en que `git log` nos da información.

```
$ 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 4.2: Salida por defecto de la instrucción `git log`

La salida del comando es cada *commit* con su suma de comprobación SHA-1, el nombre y correo del autor, la fecha y el mensaje que el autor ha escrito para ese *commit*.

La instrucción `git log` posee múltiples opciones para ajustar la búsqueda que realmente se quiere, las opciones que se deciden utilizar en este proyecto son las siguientes:

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

Con la utilización de estas opciones se obtiene una lista de todos los cambios que ha sufrido la línea indicada en el parámetro desde su creación en el fichero hasta la actualidad.

Podemos ver en el siguiente ejemplo, la salida del comando que estamos tratando en nuestro código, se muestra en la figura 4.3.

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

Analizando dicha salida, se observa 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)
```

```
$ 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 4.3: Salida de la instrucción `git log` con las siguientes opciones:
git log -L nºLínea,nºLínea:nombreFichero

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 implementar esta idea en código dentro del proyecto, hemos utilizado la librería de JavaScript `simple-git`[1].

[1] `https://www.npmjs.com/package/simple-git`

Una vez explicado esto, podemos cerrar el paréntesis y seguir con el análisis del proceso. A modo de recordatorio diremos que este punto se han almacenado las líneas del grupo "Zona naranja" con el siguiente formato:

```
var lineaSchema = mongoose.Schema({

linea: String,
fichero: String,
numLinea: String,
longitudLinea: String,
pasadoRojo: Boolean
historia: []

});
```

Se ejecuta la instrucción `git log` con las opciones correspondientes explicadas anteriormente, para cada una de ellas. La salida proporcionada por la instrucción se analiza para comprobar si la línea en cuestión posee algún momento de su historia con una longitud mayor a 80 caracteres.

Si esto ocurre se modifica su registro en la base de datos de tal manera que su parámetro "pasadoRojo" pasa a tener un valor igual a `True` y en su parámetro "historia" se guardan todos los estados que ha tenido dicha línea en su pasado y que cumplen dicha condición.

4.4. Análisis del pasado de una línea

Realmente la cuestión buscada en este punto del proceso que explico en esta memoria, 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).

Hay que entender que muchas veces, se puede dar esta iteración y aun así no ser el caso que incumbe a este estudio, por tanto, se ha de analizar el pasado de la línea, y para implemetar esta idea en código, la única forma que tenemos es 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 el nombre de una variable.

Para intentar clarificar este último párrafo mostramos esta cuestión en un ejemplo. La clave que se intenta explicar es que la línea puede tener infinitos cambios en su pasado, de hecho puede ser que pase de la zona roja a la zona naranja, vuelva otra vez a la zona roja, pase a la verde, etc. Lo que se intenta comprobar es que la línea actual es lo suficientemente parecida a la última versión de la línea en la que tenía más de 80 caracteres, para comprobar, como ya se explica en el párrafo anterior, si el cambio se produce por el motivo estudiado. A continuación tenemos el siguiente ejemplo sacado de nuestro propio estudio:

```
settings = {'FEED_FORMAT': format, 'FEED_EXPORT_INDENT': None}
settings = {'FEED_FORMAT': format, 'FEED_EXPORT_INDENT_WIDTH': None}
```

La primera línea de código, es la que hay en la actualidad en el repositorio, y la segunda es la última versión que tenía más de 80 caracteres en su pasado. Es decir, esta línea entraría claramente en nuestro análisis, ya que podemos comprobar que la línea en la actualidad es prácticamente igual a la versión de la línea que tenía más de 80 caracteres, de hecho, podemos intuir que la línea no ha sufrido más cambios después, y que la iteración de zona roja a zona naranja es el último cambio sufrido por dicha línea.

Sin embargo, se puede dar el caso en que la línea ha sufrido ese cambio, pero más adelante sufre otros, de forma que la Línea actual queda bastante cambiada, pero sigue teniendo las características de estar en la zona naranja, la comparativa entre la última versión de la línea en la zona roja y la línea actual queda entonces así:

```
[settings, data] = {'FEED_FORMAT': None}
settings = {'FEED_FORMAT': format, 'FEED_EXPORT_INDENT_WIDTH': None}
```

Esta línea no entra en los filtros del estudio, ya que como podemos ver la línea en la actualidad no es lo suficientemte parecida a su última versión en la zona roja. Esta línea ha pasado de estar en la zona roja a la zona naranja a lo largo de su historia, sin embargo, no entraría dentro de los parámetros de nuestro proceso.

Volviendo al análisis del proceso, en este punto todavía no se puede saber si la línea ha sufrido ese cambio por el motivo del estudio que explicamos en esta memoria. Lo que se realiza aquí es un filtrado para comprobar que la línea actual es suficientemente parecida a su pasado como para que esta cuestión pueda llegar a darse. Para ello se utiliza la distancia de Levenshtein[2].

Analizando la salida que nos proporciona la distancia de Levenshtein, se fija un umbral de decisión.

Se obtiene toda la historia de la línea en cuestión en la que su longitud fue mayor a 80 caracteres, accediendo para ello al parámetro en el que se ha guardado dicha información. En este punto se compara la línea actual con cada uno de los estados en su pasado.

En función de la respuesta que proporcione Levenshtein y en base al umbral de decisión que anteriormente se ha fijado se comprueba la siguiente cuestión:

Si la respuesta está dentro del umbral, se mantiene almacenado "el estado de la línea en el pasado" que se acaba de comprobar, en caso contrario, se desecha, actualizando el registro correspondiente en BBDD.

Cuando se termina esta comprobación para cada una de las líneas, cabe destacar el estado en el que se encuentra el proceso de análisis en este punto.

Se hace una limpieza en BBDD de los registros de líneas que no cumplen todos los parámetros de los filtros expuestos anteriormente. De esta manera, se mantienen únicamente las líneas que todavía se consideran conflictivas. En el parámetro "historia:[]" de cada una de ellas se mantienen guardados todos los estados del pasado de dicha línea, los cuales son mayores a 80 caracteres y se parecen lo suficiente a la línea actual como para poder darse el caso que se maneja en este estudio.

[2] `https://es.wikipedia.org/wiki/Distancia_de_Levenshtein`

4.5. Clasificación de líneas conflictivas

En este punto, se dispone en la BBDD únicamente de las líneas que se han considerado conflictivas, y que por tanto, han superado todos los filtros del proceso de estudio. En este punto del análisis se realiza una clasificación basada en el motivo por el cual la línea ha acortado su longitud.

Dentro de los posibles casos, hemos decidido clasificar las líneas en 4 grupos –recordemos 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 motivo del cambio de longitud de una sola palabra (caso a estudiar).
2. La línea se diferencia por el único motivo de la eliminación de 4 espacios iniciales (tabulaciones).
3. La línea se diferencia por la eliminación de 4 espacios iniciales (tabulaciones), y además por el cambio de longitud de una sola palabra. (posible caso a estudiar).
4. Resto de líneas.

Para plasmar esta clasificación en el código de nuestro proyecto se han elaborado 3 procesos distintos. Cada par de líneas a estudiar (línea antigua con más de 80 y línea actual entre 70 y 79) será analizado por cada uno de los procesos. Según los resultados obtenidos se decide si la línea encaja en algunos de los 3 grupos de los que se dispone, o en caso contrario, será descartada en el último grupo **Resto de líneas**.

- **La línea se diferencia por el único motivo del cambio de longitud de una sola palabra:** Se realiza una partición, tanto de la línea antigua como de la línea actual, por espacios. De manera que dispondremos de dos arryas en los cuales tendremos guardadas cada una de las distintas palabras de cada una de las líneas. A continuación se cruzan ambos arrays realizando una comparación de cada una de sus posiciones, con esta comparación se busca encontrar el caso en el que todas las posiciones de ambos arrays sean idénticas excepto una. Si esto ocurre nos encontramos ante un caso en el que las dos líneas

son idénticas exceptuando una palabra. Esta palabra puede ser una posible variable, por tanto, si esto ocurre se decide añadir el par de líneas a este grupo.

- **La línea se diferencia por el único motivo de la eliminación de los 4 caracteres iniciales:** En primer lugar se eliminan los 4 primeros espacios de la línea antigua, y se guarda en una línea auxiliar el resto de la línea. Una vez hecho esto, se realiza la siguientes comprobaciones: 1 - se compara la línea actual con la línea auxiliar y se comprueba si son exactamente iguales. 2 - se comprueba que los 4 primeros caracteres que anteriormente hemos eliminado de la línea antigua, son efectivamente, 4 espacios en blanco (tabulaciones). Si ambas comprobaciones se cumplen, el par de líneas será introducido en este grupo.

- **La línea se diferencia por la eliminación de 4 espacios iniciales (tabulaciones), y además por el cambio de longitud de una sola palabra:** Al igual que en el primer grupo, se realiza una partición, tanto de la línea antigua como de la línea actual, por espacios. De manera que dispondremos de dos arryas en los cuales tendremos guardadas cada una de las distintas palabras de cada una de las líneas. Una vez disponemos de estos dos arrays, se realizan 3 comprobaciones. 1 - se comprueba mediante ambos arrays que las dos líneas contienen el mismo número de palabras. 2 - si se cumple la comprobación uno, se comparan las dos líneas originales, para comprobar que en un principio no son iguales, de esta manera se llega a la conclusión de que lo único que diferencia a ambas líneas son espacios en blanco. 3 - si se cumplen las comprobaciones uno y dos, pasaremos a comprobar mediante los dos arrays obtenidos que las palabras de ambas líneas coinciden exactamente exceptuando una sola palabra. Si las tres comprobaciones se cumplen, dicho par de líneas será introducido en este grupo.

Una vez realizada esta clasificación de todas las líneas conflictivas, nos encontramos en disposición de mostrar, explotar y analizar los resultados de nuestro estudio.

Capítulo 5

Resultados

5.1. Resultados del análisis general

Una vez que hemos creado el software, podemos analizar un gran volumen de líneas de código en lenguaje Python.

Cabe destacar que para la realización de este proyecto hemos intentado recoger datos del "mundo real", es decir, de proyectos reales de software libre, que nos permitan sacar conclusiones reales. Por ello, se analizan repositorios de GitHub [16].

Para dar a dicha muestra las características oportunas para este proyecto, se ha pensado en la variedad. Esto quiere decir que para obtener datos de un mayor número de desarrolladores, se ha decidido seleccionar el mayor número de repositorios posibles, en lugar de escoger un único repositorio de gran tamaño. Se ha tomado esta decisión pensando que en un mismo proyecto se suele trabajar bajo las mismas pautas y reglas de calidad marcadas de antemano. Por tanto, aunque dicho proyecto cuente con varios desarrolladores, todos ellos generarán código bajo las mismas normas y nuestro estudio quedaría reducido a ese único proyecto.

Así, hemos analizado en total el siguiente número de líneas, tal y como se muestra en la figura 5.1:

- **Número total de líneas analizadas:** 369.911

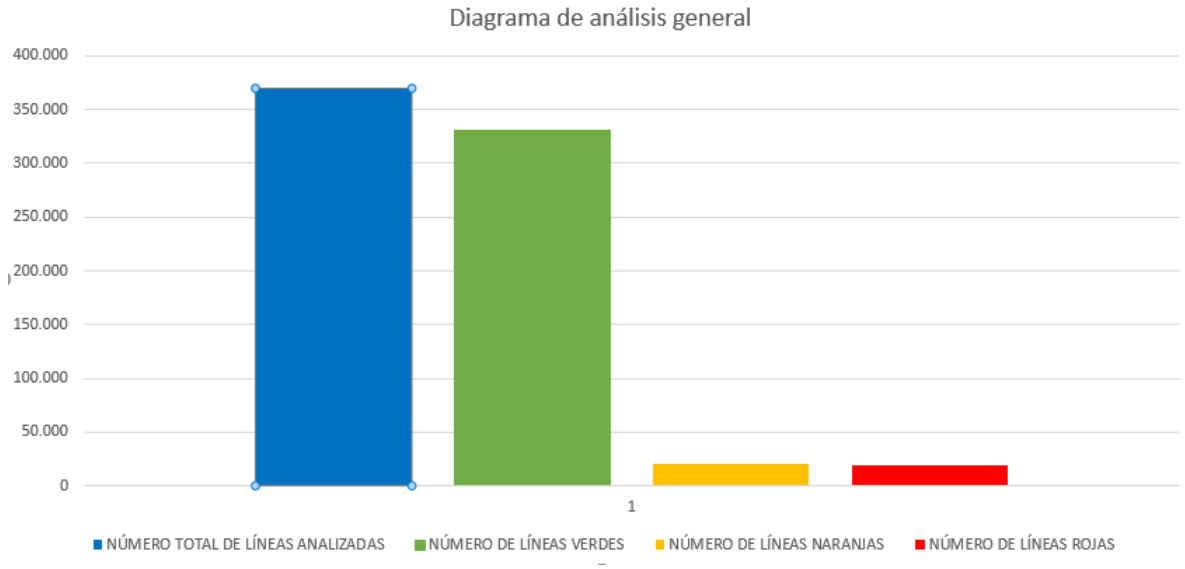


Figura 5.1: Diagrama de barras de los resultados generales.

De estas líneas, y con la nomenclatura explicada en el capítulo 4, hemos encontrado:

- **Número de líneas verdes:** 330.900
- **Número de líneas naranjas:** 20.107
- **Número de líneas rojas:** 18.904

El primer punto a destacar de estos resultados son los datos obtenidos para cada grupo de líneas, se ha analizado código subido a repositorios de Github que contienen código Python. Esto hace pensar que dicho código sigue la guía de estilo PEP8, de manera que no se encontrarán apenas "líneas rojas". Sin embargo, se puede observar que el número de este grupo de líneas es similar al del grupo "líneas naranjas".

Para esclarecer este dato se ha realizado un análisis más en profundidad sobre el mismo. Se puede observar que el número de líneas (18.904) puede parecer grande en un principio, sin embargo, cabe destacar que esto tan sólo supone el 5,11 % del total analizado, un porcentaje bastante bajo. Para indagar más sobre este dato se decide en este punto añadir una comprobación en el código del proyecto. Esta nueva comprobación consiste en averiguar cuantas de esas líneas son comentarios (tanto comentarios *in-line*, como comentarios de varias líneas). De esta manera se sabrá si este dato está falseado y nos encontramos ante líneas que no son líneas *reales* de código fuente.

En dicho análisis obtenemos un resultado de 1.000 líneas, es decir, un 5,28 %. Lo que quiere decir que sí hemos encontrado líneas de código fuente que están en nuestra *zona roja*.

Poniendo el foco en los grupos de líneas menores a 80 caracteres, se puede comprobar que la inmensa mayoría están en las llamadas "línea verdes" un 89,45 %. Esto quiere decir que el grupo conflictivo que nos incumbe en este proyecto, las "línea naranjas", queda en un 5,43 %.

Una vez procesados los datos totales del estudio, centraremos nuestro interés en los distintos repositorios que hemos utilizado. Tenemos el siguiente listado:

Cuadro 5.1: Proyectos analizados y resultados obtenidos.

Proyecto	Verdes	Naranjas	Rojas	Rojas Comentarios	Totales
youtube-dl	112.898	6.332	8.507	5 (0,06 %)	127.737
SerpentAI	3.396	130	282	0 (0,00 %)	3.808
requests	29.164	1.475	3.235	79 (2,44 %)	33.874
scrapy	31.174	2.400	1.709	99 (5,79 %)	35.283
pythondotorg	10.609	380	601	3 (0,49 %)	11.590
models	56.246	2.949	1.058	683 (0,64 %)	60.253
lbry	24.832	1.700	1.885	59 (3,12 %)	28.417
keras	49.456	3.428	1.525	60 (3,93 %)	54.409
flask	13.030	1.307	99	11 (11,11 %)	14.436
awesome-python	70	3	3	1 (33,30 %)	76
awesome-machine-learning	25	3	0	0 (0,00 %)	28

Se puede observar que hemos tomado como muestra repositorios de muy diversas características. Cabe destacar:

- Como es de esperar, en todos los repositorios, la gran mayoría de líneas son verdes.
- Sólo hemos encontrado un repositorio que no tenga líneas rojas, aunque solo tiene 28 líneas en total.
- Hay 5 repositorios en los que hay más líneas naranjas que rojas, 5 repositorios en los que hay más líneas rojas que naranjas, y un repositorio en el que hay las mismas líneas rojas que naranjas.

5.2. Resultados del análisis del grupo conflictivo

En esta sección analizamos con detalle el grupo que hemos considerado *conflictivo*. Después de pasar nuestro primer filtro (véase la sección 4.2), las primeras líneas que podemos considerar *conflictivas* según nuestros parámetros, son las siguientes:

- **Número de líneas conflictivas:** 396

Esto nos da:

- 0.10 % del total de líneas analizadas (369.911)
- 1.96 % del total de líneas en zona naranja (20.107)

Se puede observar que los porcentajes que se manejan en este punto son bajísimos. Realizando sobre estas 396 líneas la clasificación que se detallada en el capítulo 4 (véase la sección 4.5) se obtienen como resultado[1]:

- **La línea se diferencia por el único motivo de la eliminación de 4 espacios iniciales (tabulaciones):** 1

  Ejemplo:
  Línea actual:

```
'url': 'http://91porn.com/view_video.php?viewkey=7e42283b4f5ab36da134',
```

  Línea en el pasado:

```
    'url':'http://91porn.com/view_video.php?viewkey=7e42283b4f5ab36da134',
```

- **La línea se diferencia por el único motivo del cambio de longitud de una sola palabra:** 125

  Ejemplo:
  Línea actual:

```
'subtitles':self.extract_subtitles(video_id, subtitles_id, hl),
```

[1]En los ejemplos que se muestran *verbatim* en esta memoria, se ha omitido la indentación de las líneas para que quepan en el ancho de página.

Línea en el pasado:

```
'subtitles':self.extract_subtitles(video_id, video_subtitles_id, hl),
```

- **La línea se diferencia por la eliminación de 4 espacios iniciales (tabulaciones), y además por el cambio de longitud de una sola palabra:** 3

  Ejemplo:
  Línea actual:

  ```
  'https://tv.nrk.no/program/Episodes/{series}/{season}'.format(
  ```

  Línea en el pasado:

  ```
  self.url_result('https://tv.nrk.no/program/Episodes/{series}/{season}'.format(
  ```

- **Resto de líneas:** 267

  Ejemplo:
  Línea actual:

  ```
  'url':'http://www.chilloutzone.net/video/eine-sekunde-bevor.html',
  ```

  Línea en el pasado:

  ```
  u'url':u'http://www.chilloutzone.net/video/enemene-meck-alle-katzen-weg.html',
  ```

En esta clasificación disponemos de dos grupos que entran dentro del interés de nuestro estudio, los cuales son: "La línea se diferencia por la eliminación de 4 espacios iniciales (tabulaciones), y además por el cambio de longitud de una sola palabra" y "La línea se diferencia por el único motivo del cambio de longitud de una sola palabra". Unificando estos dos grupos y estableciendo los porcentajes en función de las 396 líneas conflictivas, se obtiene que:

- **Grupo conflictivo:** 32,32 %
- **Grupo NO conflictivo:** 67,68 %

Como podemos observar en todos los datos mostrados hasta ahora, el porcentaje de líneas que entran en nuestro estudio es mucho menor que el porcentaje que se descarta.

Comparando este último filtrado con los datos iniciales, tenemos que **128 líneas de 369.911** pueden haber sufrido el cambio que incumbe a este proyecto. Esto supone un **0.034 %**. Este dato es un porcentaje ínfimo, que da a entender que el debate sobre esta cuestión carece de sentido, o al menos, no debería tener la importancia que se le ha dado.

5.3. Análisis (manual) de líneas particulares

Por último, cabe destacar que hemos obtenido un resultado total de 128 líneas, sin embargo, no podemos afirmar que dichas líneas hayan sufrido el cambio explicado en esta memoria a propósito, esa afirmación es totalmente subjetiva. En este punto, queda pendiente el ejercicio de que un humano revise esas líneas y decida si la línea ha pasado de tener una longitud mayor a 80 caracteres a tener una longitud menor a 80 caracteres, reduciendo para ello la longitud de una sola variable, y por tanto, el motivo para ello ha sido el problema explicado en este proyecto.

Para ilustrar esta cuestión, se recoge una pequeña muestra de nuestros resultados, y se procede a anlizarlos manualmente:

Par de líneas:

Línea actual:

```
_VALID_URL = r'https?://(?:www\.)?alpha\.com/videos/(?P<id>[^/]+)'
```

Línea en el pasado:

```
_VALID_URL = r'https?://(?:www\.)?alpha\.com/videos/(?P<display_id>[^/]+)'
```

Análisis:

En primer lugar intento detectar la parte que se ha modificado y analizar el porqué. En este caso la parte modificada no parece ser una variable, las razones que me llevan a llegar a esta conclusión son las siguientes. El nombre de la variable inicial ("_VALID_URL") y su contenido me hace pensar claramente que estamos ante una url que se está almacenando en una variable. No dispongo de más contexto del resto del código que tiene esta línea (las líneas colindantes), pero basándome en mi experiencia, aunque a veces las url con las que trabajamos se conformen dinámicamente dentro del código, no me parece que la parte modificada ("display_id" a "id") sea una variable en sí. Por tanto, creo que en este caso no se trata de un cambio en el nombre de la variable con el fin de cumplir con la guía de estilo PEP8.

Par de líneas:

Línea actual:

```
return self.playlist_result(entries, video_id, title, description)
```

Línea en el pasado:

```
return self.playlist_result(entries, video_id, playlist_title, description)
```

Análisis:

En esta línea observo que la parte que se ha modificado es un parámetro que se le está pasando a una función. Esto me hace pensar que muy probablemente me encuentro ante una variable. Observando la modificación en sí ("playlist_title" a simplemente "title"), puedo afirmar que el nombre de la variable ha perdido sentido. No parece que el hecho de denominar a una variable con el único nombre de *título* tenga demasiado sentido. En esta línea, por tanto, me atrevo a decir que sí se trata de un cambio para cumplir con la guía de estilo PEP8, y además, el código ha quedado peor que el original. Esto quiere decir que esta línea sufre el problema explicado en esta memoria.

Par de líneas:

Línea actual:

```
'skip': 'redirect to http://swrmediathek.de/index.htm?hinweis=swrlink',
```

Línea en el pasado:

```
'_skip': 'redirect to http://swrmediathek.de/index.htm?hinweis=swrlink',
```

Análisis:

Observo en esta línea que me encuentro ante una asignación de su valor a una variable inicial, dicha variable inicial es la palabra modificada. Esto quiere decir que cumple la primera condición, sin embargo, observo la modificación ("_skip" a "skip") y no parece que el nombre de la variable haya perdido ningún sentido. Puedo afirmar por tanto, que en este caso no estamos ante una línea que ha sufrido la modificación estudiada en este proyecto.

Par de líneas:

Línea actual:

```
except (TimeoutError, HTTPException, SocketError, ProtocolError) as e:
```

Línea en el pasado:

```
except (TimeoutError, HTTPException, SocketError, ConnectionError) as e:
```

Análisis:

Observo en esta línea que la palabra modificada es un parámetro que esta recibiendo una función, y por tanto, es muy probable que nos encontremos de nuevo ante una variable. De nuevo me voy a analizar la modificación sufrida por esta: "ConnectionError" a "ProtocolError"). A simple vista puedo ver que la variable ha cambiado su nombre porque ha cambiado su sentido, y el motivo en este caso no ha sido intentar cumplir con la guía de estilo PEP8. Tampoco esta línea ha sufrido el cambio que se explica en esta memoria.

Par de líneas:

Línea actual:

```
elif status['startup_status']['code'] == LOADING_WALLET_CODE:
```

Línea en el pasado:

```
elif status['result']['startup_status']['code'] == LOADING_WALLET_CODE:
```

Análisis:

En esta línea observo algo que también puede ocurrir en el proceso de análisis. Debido a la forma en la que se clasifican las líneas debido a su modificación (véase la sección 4.5), la parte de línea que detecta que ha cambiado es "status['result']['startup_status']['code']" por "status['startup_status']['code']". Se observa que al no tener espacios, la herramienta lo ha detectado como una palabra y posible variable. De esta forma podemos afirmar de nuevo que no estamos ante una línea que haya sufrido el cambio que nos incumbe en este proyecto.

Par de líneas:

Línea actual:

```
`(batch, new_rows, new_cols, filters)` if data_format='channels_last'.
```

Línea en el pasado:

```
`(batch, new_rows, new_cols, nb_filter)` if data_format='channels_last'.
```

Análisis:

Observo en esta línea que de nuevo me puedo encontrar ante una variable, debido al formato de dicha línea. Sin embargo, analizando el cambio en sí ("nb_filter" a "filters"), no veo que el

nombre de la variable haya perdido sentido. Por tanto, no puedo afirmar que este cambio no se haya producido para cumplir la guía de estilo PEP8, sin embargo, sí puedo afirmar que no estoy ante una línea que ha sufrido el cambio del que se habla en esta memoria, ya que el código resultante no me parece peor que el original.

Par de líneas:

Línea actual:

```
def _download_name(self, name, timeout=None, download_directory=None,
```

Línea en el pasado:[2]:

```
def _download_name(self, name, timeout=conf.settings.download_timeout,
                                                    download_directory=None,
```

Análisis:

Observando la parte modificada, puedo ver que no me encuentro ante el nombre de una variable. El cambio es pasar de "timeout=conf.settings.download_timeout" a ser "timeout=None". La herramienta ha detectado la variable y su asignación como una sola palabra al carecer de espacios. Observando el cambio no me parece en absoluto que este se haya dado para cumplir la guía de estilo PEP8.

Par de líneas:

Línea actual:

```
'lynda returned error: %s' % video['Message'], expected=True)
```

Línea en el pasado:

```
'lynda returned error: %s' % video_json['Message'], expected=True)
```

Análisis:

Observo esta línea y llego a la conclusión de que la parte afectada no es una variable, sin embargo, puede ser una función o método. El cambio es "video_json['Message']" a "video['Message']". Creo que este cambio ha hecho que el nombre del método en cuestión pierda

[2]La línea original está en una única línea. Se muestra así para que quepa en el ancho de página de esta memoria.

sentido, ya que se le ha quitado información sobre el formato al nombre, y por tanto el código resultante me parece peor que el original. Creo que esta línea si que puede entrar dentro del grupo que está afectado por el cambio que se explica en este proyecto.

Par de líneas:
Línea actual:

```
with mock.patch.object(attr, '__setattr__') as mock_setattr, \
```

Línea en el pasado:

```
with mock.patch.object(SettingsAttribute, '__setattr__') as mock_setattr, \
```

Análisis:

Veo en esta línea que la palabra afectada es un parámetro que se le pasa a una función, por tanto, al igual que en ocasiones anteriores puedo afirmar que estamos ante una variable. Observando el cambio ("SettingsAttribute" a "attr"), parece bastante claro que el nombre de la variable ha perdido todo el sentido que poseía. En este caso, de nuevo el código resultante ha quedado peor que el código original. De manera que esta línea también ha sufrido el cambio estudiado en esta memoria.

Par de líneas:
Línea actual:

```
log.info("Responding with %s infos", str(len(blob_infos)))
```

Línea en el pasado:

```
logging.info("Responding with %s infos", str(len(blob_infos)))
```

Análisis:

Observo en esta línea que la parte modificada ("logging.info" a "log.info") no es una variable. Además puedo afirmar viendo el cambio, que no se ha producido por el motivo de cumplir la guía de estilo PEP8. Por tanto, se puede decir que esta línea no ha sufrido el cambio mencionado en esta memoria.

Conclusión del análisis manual: Hemos seleccionado una pequeña muestra de 10 líneas obtenida de nuestros resultados. Hemos obtenido 3 SÍ y 7 NO. Estos resultados son subjetivos y sólo aportan evidencia parcial y no completa. Además nos falta mucho contexto sobre el código analizado, ya que estamos tratando sobre líneas totalmente aisladas. Aun así, permiten analizar en detalle el fondo de la cuestión y ampliar la perspectiva del análisis, tanto para ver la problemática a la que nos enfrentamos como los resultados de la misma. De nuevo, hemos obtenido generalmente unos resultados negativos respecto al caso buscado.

Capítulo 6

Conclusiones

6.1. Consecución de objetivos

En el Capítulo 2 (en las Secciones 2.1 y 2.2) de esta memoria se exponen una serie de objetivos, tanto principales como secundarios, que, después de desarrollar los Capítulos 3, 4 y 5, podemos evaluar si se han cumplido o no.

En primer lugar, se planteaba como gran objetivo principal ser capaces de realizar un amplio estudio sobre código Python para poder llegar a una conclusión sobre si los desarrolladores estaban realizando en líneas generales una determinada práctica.

Para ello necesitábamos que nuestro estudio fuese lo suficientemente amplio como para que los parámetros utilizados fueran reales para poder dar una conclusión a nivel global, también necesitábamos que nuestro código tuviese los filtros suficientes, y la capacidad de acotar lo suficiente nuestro problema como para dar justo el resultado que necesitamos.

Creo que este objetivo se ha cumplido completamente, ya que si bien es cierto que el estudio podría ser más amplio, o que en los resultados se podría haber acotado incluso más, los datos obtenidos después de analizar los puntos 4 y 5 de la memoria son suficientes como para llegar a una conclusión sobre el estudio propuesto. Dicha conclusión es que los desarrolladores de código Python NO están realizando la siguiente práctica: acortar el nombre de una variable para forzar que el código cumpla la regla de máxima longitud de línea de la guía de estilo PEP8, produciendo de esta forma un peor código. Ya que los resultados son claramente negativos.

En este punto también se plantea otro objetivo, quizás más secundario, pero sobre todo más abstracto. Planteamos la idea, sobre todo a nivel de estudiante, de plantearse que es realmente un código de calidad, y la correcta utilización de algunas reglas o guías de estilos como PEP8 o similares.

No se sí se puede decir que esto sea un objetivo en sí, y si lo es, que se haya cumplido como tal. Creo que este proyecto en sí no se ha centrado en expandir esta idea, quizás si algún alumno lee esta memoria sí que pueda obtenerla. Sin embargo puedo decir que este objetivo se ha cumplido con creces en mi persona. Antes de realizar este proyecto no me había planteado tanto la diferencia entre un código que funciona y un código que es bueno.

En cuanto a los objetivos secundarios que comentamos también en el Capítulo 2 (en la Sección 2.2):

- **Trabajar de una forma orientada al mundo profesional:** Este objetivo se ha cumplido con el uso de GitHub, centrándonos para el desarrollo de código en GitFlow. También lo he notado en que es un trabajo en el que estás mucho más solo que en el resto de tu vida universitaria, esto te permite enfrentarte sólo a los problemas, organizarte y dividir tu proyecto de la forma que creas necesaria.

- **Aplicar tecnologías e ideas no vistas durante la vida universitaria:** Creo que este objetivo también se ha cumplido. En este proyecto he buscado en todo momento utilizar tecnologías que no he visto en mi vida universitaria o aplicarlas de distinta forma. Por eso en el caso de hacer un desarrollo en la parte del servidor en una posible futura aplicación cliente-servidor he utilizado Node.js y JavaScript como lenguaje. En la universidad hemos visto JavaScript, pero únicamente en la parte del cliente, que es lo más común. En la universidad sólo habíamos trabajado en servidor con java y Python con Django. Para el tema de las bases de datos solo hemos visto SQL, trabajando sólo con bases de datos relaciones, por ese motivo decidí utilizar Mongo, para trabajar con una base de datos no relacional y poder captar las diferencias entre una y otra a la hora de utilizar realmente las dos.

  Otra cosa que me gustaría destacar aquí es la elección de la estructura final del proyecto,

ojeando proyectos anteriores como hacemos todos antes de realizar el nuestro he visto que casi todos se plantean como una aplicación cliente-servidor con Python en el servidor, JavaScript en el *front-end*, y SQL como base de datos. He querido enfocar este proyecto como algo un poco distinto, y en mi caso, aunque es cierto que he querido crear una estructura cliente-servidor, para aprender más sobre el tema y utilizar *frameworks* nuevos como Express, el desarrollo se ha centrado más en un estudio en el que queremos obtener unos resultados. El proyecto no ha tenido parte *front-end* como tal.

6.2. Aplicación de lo aprendido

Durante la realización de mi grado en la universidad Rey Juan Carlos me he formado en diferentes áreas y campos, muchos de ellos me han facilitado la realización de este proyecto. Algunas de ellas han sido:

1. La programación: La base de este proyecto es la programación, y antes de entrar en la universidad, prácticamente yo no sabía lo que era eso (cabe destacar que ahora que he terminado, me dedico y quiero dedicarme toda la vida a ello). Algunas asignaturas como Fundamentos de la programación, Programación de Sistemas de Telecomunicación o Sistemas Operativos me han enseñado lo que es la programación a bajo nivel.

2. Una vez que sabemos (o algo parecido) lo que es el paradigma de la programación a bajo nivel, hablando a más alto nivel, he aprendido todo lo relacionado con la programación en la parte del servidor en asignaturas como Servicios y Aplicaciones Telemáticas o Ingeniería de Sistemas de Telecomunicaciones.

3. El aprendizaje de `GitHub` ha sido para mi muy importante para este proyecto. Es otra de las cosas que puedo decir que no sabía antes de empezar la carrera, ni `GitHub` ni cualquier control de versiones, esto es algo que me enseñó la asignatura Ingeniería de Sistemas de la Información. Además, puedo decir que sé a ciencia cierta que será algo que utilice constantemente en mi vida laboral.

4. Por último, y para nada menos importante, está el concepto general de ser capaz de afrontar cualquier problema sin miedo y ser capaz de resolverlo, en este caso concreto el enfrentarme a nuevas tecnologías que nunca he utilizado y no tener miedo a utilizarlas. Esto

es algo que no se aprende en ninguna asignatura en concreto pero que sin duda es algo con lo que salimos de nuestra vida estudiantil.

6.3. Lecciones aprendidas

Como ya he comentado en alguna parte de la memoria, uno de mis objetivos era buscar tecnologías y formas de trabajar que no hubiera visto en la universidad, esto al principio fue bastante costoso, sin embargo creo que en el futuro puede tener bastantes beneficios, algunas de las cosas para las que esto me ha servido han sido:

1. Montar un servidor con NodeJS implementado en JavaScript utilizando un framework desconocido para mi como Express.

2. Utilización de una base de datos desconocida para mi como es MongoDB, y su conexión con el servidor.

3. Enfocar un proyecto de grandes dimensiones desde el principio, y planificarlo de forma correcta uno mismo, tanto temporal como estructuralmente.

4. El uso de LaTeX. Una tecnología con la que he escrito esta memoria, y que aunque al principio parece muy farragosa, enseguida ves la facilidad que te aporta para redactar documentos de este tipo.

6.4. Trabajos futuros

Todo software implementado tiene margen de mejora, en este proyecto tenemos varios puntos que merece la pena destacar.

1. En primer lugar, y como ya hemos comentado también en algún punto, una mejora muy clara es montar la parte *front-end* y hacer de este estudio una aplicación en la que quizás se puedan mostrar los resultados de forma más atractiva al usuario, y quizás añadir alguna funcionalidad o añadir algún otro estudio para dar más contenido a la aplicación.

2. Mejorar la rapidez de la aplicación para poder analizar un mayor volumen de código. La base del análisis esta programada con varios hilos en paralelo para cada repositorio,

esto hace que si el repositorio es demasiado grande y la capacidad del ordenador no es demasiado buena, se creen demasiados hilos y pueda dar problemas de rendimiento.

3. Mejorar el filtrado para quedarnos con las líneas determinadas. En este proyecto creo que hemos conseguido afinar bastante, pero aun así, siempre se cuela la llamada basura, y en los resultados finales hemos tenido que dejar un apartado de "Resto de líneas".

4. Mejorar el análisis de este mismo problema pensando otras opciones. Por ejemplo, hemos tenido en cuenta que el desarrollador cambia para ello una sola variable, o una sola variable y tabulaciones, pero ¿y si acorta dos variables en una misma línea? Ese caso no lo tenemos contemplado.

6.5. Valoración personal

Creo que en este punto es hora de echar la vista atrás y reflexionar. Echar la vista atrás al principio de este proyecto y ver como ha quedado finalmente, de lo cual estoy, sinceramente orgulloso. Los resultados obtenidos como proyecto en sí me parecen más bien normales, de lo que sí me siento orgulloso es de como lo he tenido que compaginar con un trabajo a jornada completa desde su inicio, y aunque he tenido que hacer parones no deseados en su elaboración y no he tardado lo que me gustaría, lo he conseguido. También me gusta que es un proyecto algo distinto, en mi opinión, a muchos de los últimos que he visto. Creo que también es hora de echar la vista más atrás y reflexionar sobre la vida universitaria, con este proyecto termino una etapa de mi vida, una etapa muy importante en la que he adquirido conocimientos, habilidades, he perdido miedos, me he hecho una persona adulta y todo ello me servirá para el resto de mi vida, tanto profesional como personal. Cierro esta etapa con mucha alegría y muy orgulloso, y empiezo una nueva con mucha ilusión.

Apéndice A

Instalaciones necesarias

Todo lo instalado se ha hecho sobre Windows 10.

A.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.

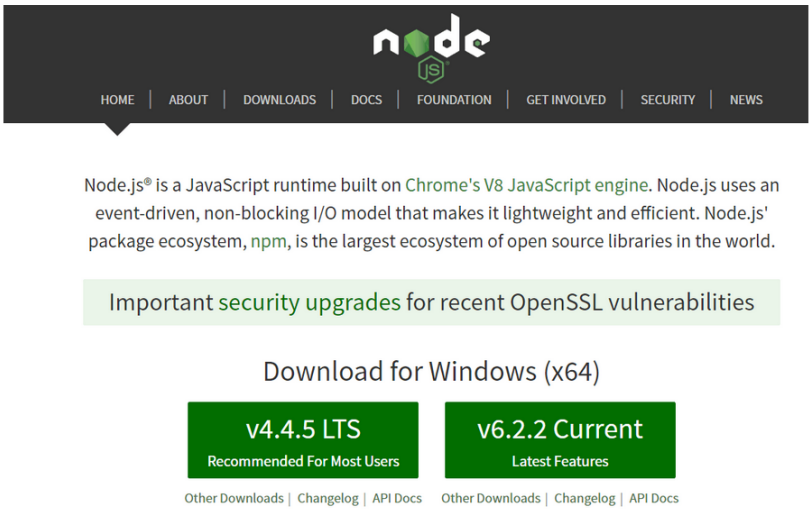


Figura A.1: Descarga de NodeJS

A.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
```

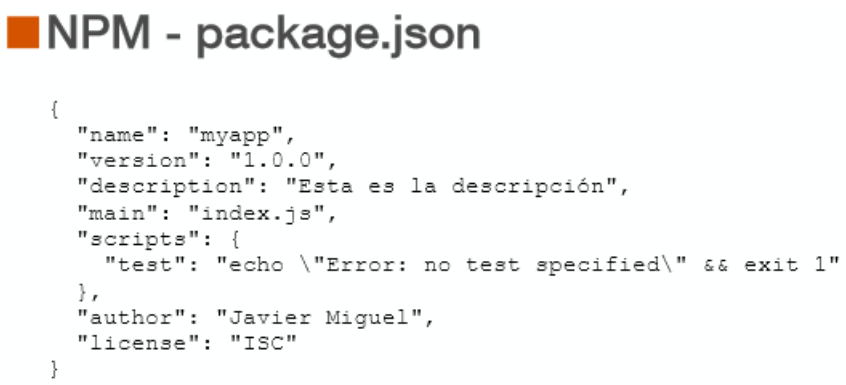


Figura A.2: Formato del `package.json` después de lanzar el comando npm init

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:

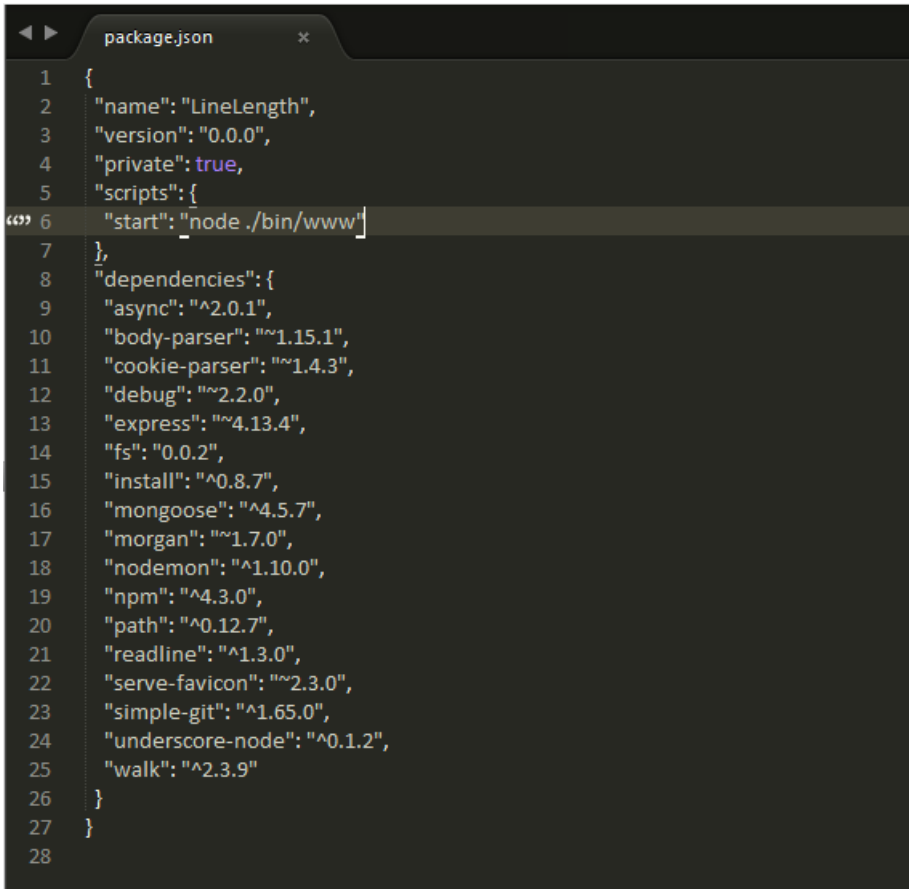


Figura A.3: Formato de nuestro `package.json`.

A.3. Creación de un nuevo proyecto Express

Una vez tenemos el equipo listo, pasamos a crear un nuevo proyecto. En el punto 3, hemos explicado en que consiste Express (véase la sección 3.7), el framework de Node.js (véase la sección 3.6) con el que vamos a realizar nuestro proyecto.

Dentro de Express vamos a utilizar la librería de Express Generator [17], que 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:

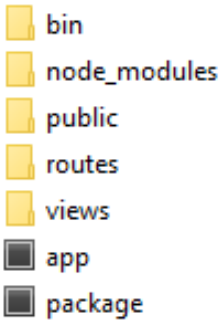


Figura A.4: Estructura básica de un proyecto

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.

Por último añadir que se puede encontrar el código de este proyecto en el siguiente repositorio público de GitHub:

`https://github.com/kivenoliva/LineLength`

Bibliografía

[1] M David Hanson. The client/server architecture. *Server Management*, page 3, 2000.

[2] Guido Van Rossum et al. Python programming language. In *USENIX Annual Technical Conference*, volume 41, page 36, 2007.

[3] David Flanagan. *JavaScript: the definitive guide*. .º'Reilly Media, Inc.", 2006.

[4] Node.js. `http://www.netconsulting.es/blog/nodejs/`. Accessed: 2017-02-20.

[5] JUAN ÍSCAR MARTÍNEZ. *NODE. JS Do's and Don'ts*. PhD thesis, 2015.

[6] Express. `https://geekytheory.com/introduccion-a-express-js`. Accessed: 2017-02-25.

[7] Ethan Brown. *Web Development with Node and Express: Leveraging the JavaScript Stack*. .º'Reilly Media, Inc.", 2014.

[8] MongoDB. `https://www.mongodb.com/es`. Accessed: 2017-02-27.

[9] Kyle Banker. *MongoDB in action*. Manning Publications Co., 2011.

[10] JSON. `https://geekytheory.com/json-i-que-es-y-para-que-sirve-json/`. Accessed: 2017-03-15.

[11] Nurzhan Nurseitov, Michael Paulson, Randall Reynolds, and Clemente Izurieta. Comparison of json and xml data interchange formats: a case study. *Caine*, 2009:157–162, 2009.

[12] Git. https://git-scm.com/book/es/v1/Empezando-Fundamentos-de-Git. Accessed: 2017-04-1.

[13] Jon Loeliger and Matthew McCullough. *Version Control with Git: Powerful tools and techniques for collaborative software development*. .º'Reilly Media, Inc.", 2012.

[14] Gitflow. https://sysvar.net/es/entendiendo-git-flow/. Accessed: 2017-04-1.

[15] Abhishek Dwaraki, Srini Seetharaman, Sriram Natarajan, and Tilman Wolf. Gitflow: Flow revision management for software-defined networks. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, page 6. ACM, 2015.

[16] Laura Dabbish, Colleen Stuart, Jason Tsay, and Jim Herbsleb. Social coding in github: transparency and collaboration in an open software repository. In *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work*, pages 1277–1286. ACM, 2012.

[17] ExpressGenerator. http://malnuer.es/js/crear-una-app-nodejs-con-express-ge
Accessed: 2017-02-25.

[18] PEP8. https://alexanderae.com/pep8-guia-de-estilo-para-python.html#fn-1. Accessed: 2016-02-18.