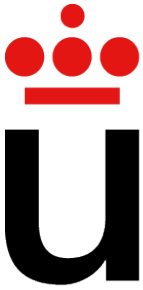

Universidad
Rey Juan Carlos

GRADO EN INGENIERÍA EN TECNOLOGÍAS DE LA TELECOMUNICACIÓN

Curso Académico 2020/2021

Trabajo Fin de Grado

CREACIÓN AUTOMÁTICA DE PULL-REQUESTS A PARTIR DE RESULTADOS DE PYLINT

Autor : Raúl Cano Montero

Tutor : Dr. Gregorio Robles

Trabajo Fin de Grado

Creación Automática de Pull-Requests a partir de Resultados de Pylint

Autor : Raúl Cano Montero

Tutor : Dr. Gregorio Robles

La defensa del presente Proyecto Fin de Carrera se realizó el día de julio de 2021, siendo calificada por el siguiente tribunal:

Presidente:

Secretario:

Vocal:

y habiendo obtenido la siguiente calificación:

Calificación:

Fuenlabrada, a de julio de 2021

Dedicado a

mi familia / mi abuelo / mi abuela

II

Agradecimientos

En primer lugar quiero agradecérselo a mis padres, Andrés y Belén, y mi hermano, David. Sin su apoyo y esfuerzo jamás habría podido estudiar una carrera ni llegar hasta donde he llegado.

A mi pareja, Macarena, por aguantar y estar ahí durante estos años.

A mis compañeros, Adri, José, Razvan y Tomás, por hacer más fácil este camino y, sobre todo, a Lydia, por tirar de mí en los peores momentos. Sin ella esto habría sido imposible.

Y, por último, a Gregorio, mi tutor, por su ayuda y paciencia al hacer este trabajo.

Gracias.

Resumen

El objetivo de este Trabajo de Fin de Grado es analizar proyectos realizados en lenguaje Python para evaluar el nivel de cumplimiento con la guía de estilo PEP8 y corregirlos para aumentar éste.

El proyecto se basa en el modelo de arquitectura cliente-servidor y se ha desarrollado como una aplicación web accesible desde Internet que permite a cualquier usuario evaluar el nivel de cumplimiento de sus proyectos con PEP8 mediante el análisis de repositorios alojados en GitHub. Algunos de los errores observados son corregibles por la aplicación y los cambios se suben al repositorio de GitHub mediante el uso de Pull-Requests.

La aplicación ha sido desarrollada en lenguaje Python y se han usado tecnologías como Django, HTML, CSS, JavaScript o PostgreSQL.

Summary

This Final Degree Project aims to analyze projects developed in Python in order to evaluate their compliace level with the style guide for Python known as PEP8 and correct their code to increase their compliance level.

This project is based on the client-server model and has been developed as a web application accesible from the Internet that allows any user to evaluate the compliance level of their projects with PEP8 by analyzing repositories hosted on GitHub. Some of the detected errors are correctables by the application and the changes are committed and uploaded to the GitHub repository using Pull-Requests.

The application has been developed in Python language and technologies such as Django, HTML, CSS, JavaScript or PostgreSQL have been used.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

Este documento contiene la memoria de mi trabajo de fin de grado de ingeniería en tecnologías de la telecomunicación. El proyecto consiste en la creación automática de Pull-Requests que aumenten el nivel de cumplimiento con la guía de estilo PEP8 de repositorios escritos Python que se encuentren alojados en GitHub. Para ello se desarrollará una aplicación que analice dicho código y posteriormente pueda corregirlo para aumentar dicho nivel de cumplimiento.

La aplicación será de tipo web, desarrollada con Django y haciendo uso de tecnologías como Python, HTML, CSS o JavaScript, y será accesible desde Internet para que cualquier usuario pueda acceder a la aplicación y realizar el análisis y la corrección sobre los repositorios que quiera, por lo que se realizará un despliegue de la aplicación en Heroku.

El análisis se ejecutará mediante la herramienta Pylint realizando un escaneo de todos los ficheros con extensión *.py* y obteniendo la lista de errores indicados por Pylint. Algunos de estos errores serán corregibles por la aplicación, de manera que se pueda aumentar el nivel de cumplimiento indicado anteriormente.

El código analizado por la aplicación estará almacenado en plataformas de almacenamiento de repositorios, como GitHub o GitLab, y las modificaciones realizadas serán subidas automáticamente al repositorio correspondiente mediante el uso de Pull-Requests.

Además, la aplicación mostrará información sobre el número total de errores encontrados y corregidos en los repositorios que han sido analizados.

1.1. Estructura de la memoria

La memoria está estructurada de la siguiente manera:

- En el capítulo 1 se hace una introducción al proyecto.
- En el capítulo 2 se muestran los objetivos generales y específicos del proyecto y la planificación temporal del mismo.
- En el capítulo 3 se presenta el estado del arte, introduciendo las diferentes tecnologías usadas en el proyecto.
- En el capítulo 4 se describe de manera detallada el diseño del proyecto, profundizando en la arquitectura general y el diseño e implementación del servidor.
- En el capítulo 5 se detallan las diferentes pruebas realizadas para probar el funcionamiento correcto de la aplicación durante diferentes etapas del desarrollo.
- En el capítulo 6 se explican los resultados obtenidos tras ejecutar la aplicación con proyectos reales.
- En el capítulo 7 se presentan las conclusiones finales acerca del proyecto.

Capítulo 2

Objetivos

2.1. Objetivo general

El objetivo de este Trabajo de Fin de Grado es realizar el análisis, mediante la herramienta Pylint, la corrección del código y la creación de Pull-Requests sobre proyectos desarrollados en Python y almacenados en GitHub, para que éstos cumplan con las normas recogidas en la guía de estilo de Python denominada PEP8

2.2. Objetivos específicos

Para poder cumplir con el objetivo general se han tenido en cuenta los siguientes objetivos específicos:

- **Trabajar con proyectos reales.** Utilizar la aplicación para analizar y corregir proyectos reales.
- **Aplicación web.** Desarrollar como una aplicación web para hacer mejor y más sencilla la experiencia de usuario.
- **Accesibilidad desde Internet.** Hacer que la aplicación esté en funcionamiento continuamente y sea accesible desde cualquier ubicación.
- **Compatibilidad con el servidor GitLab de la ETSIT.** Hacer que la aplicación sea compatible con el GitLab de la ETSIT para permitir que los alumnos de la escuela puedan

analizar y corregir sus proyectos.

- **Análisis de aceptación de las Pull-Requests.** Evaluar la aceptación de los cambios realizados en los proyectos mediante el análisis del estado de las Pull-Requests realizadas.

2.3. Planificación temporal

La planificación que se ha seguido para la elaboración de este trabajo es la siguiente:

- Reunión con el tutor para escoger el tema sobre el que se va a realizar el trabajo.
- Estudio de la API de GitHub.
- Desarrollo de funciones para conectar y realizar peticiones a la API de GitHub.
- Documentación sobre los errores que puede detectar Pylint.
- Selección aleatoria de repositorios de GitHub y análisis con Pylint para obtener los errores más frecuentes y realizar la selección de los que van a ser corregibles por la aplicación.
- Desarrollo de las primeras funciones para corregir los errores detectados por Pylint junto a sus tests unitarios.
- Realización de pruebas con ficheros con código Python y, posteriormente, con repositorios alojados en GitHub.
- Inicio del proyecto Django diseñando las vistas, el modelo de datos e integrando las funciones ya desarrolladas para conectar con la API de GitHub y corregir los errores.
- Estudio de la API de GitLab y desarrollo de las funciones para conectar y realizar peticiones a su API.
- Realización de pruebas de análisis de repositorios de GitHub y GitLab con la aplicación levantada en Django.
- Desarrollo de los templates HTML y CSS.
- Despliegue de la aplicación en Heroku.

- Realización de pruebas con repositorios de test para comprobar el correcto funcionamiento de la aplicación tras el despliegue en Heroku.
- Ejecución de análisis y correcciones sobre proyectos reales.
- Elaboración de la memoria.

Capítulo 3

Estado del arte

En este capítulo se describen brevemente las principales tecnologías utilizadas en el desarrollo del proyecto.

3.1. Python

Python [**?**] es un lenguaje de programación Open Source escrito por Guido van Rossum entre finales de los años 80 y principios de los 90. El objetivo de la creación de Python era tener un lenguaje de programación con una sintaxis sencilla, como el lenguaje ABC en el que se inspira, añadiendo la posibilidad de realizar llamadas al sistema con compatibilidad con distintos sistemas operativos.

Python es un lenguaje de programación interpretado, interactivo y orientado a objetos. Al ser un lenguaje interpretado se reduce el tiempo entre la escritura del código y la ejecución del mismo, ya que no hay que compilarlo cada vez que se realice una modificación del código. Sin embargo, ésto implica que sea necesaria la instalación de un intérprete en la máquina que va a ejecutar el código y aumenta el tiempo de la ejecución respecto a los lenguajes compilados.

Python incorpora módulos, excepciones, tipado dinámico, tipos de datos dinámicos de muy alto nivel y clases. La librería estándar de Python le permite cubrir distintas áreas, como el procesamiento de cadenas de texto, protocolos de red, ingeniería del software e interacción con el sistema operativo.

Además de la programación orientada a objetos, soporta múltiples paradigmas de programación como la programación procedimental y la programación funcional y también es compatible

con diversos sistemas operativos, incluyendo Windows y múltiples variantes de Unix, como Linux y macOS.

Actualmente Python es uno de los lenguajes de programación más utilizados [**?**].

3.2. PEP 8

Python Enhancement Proposal 8, más conocida como PEP 8 [**?**], es una guía de estilo para código escrito en Python que contiene recomendaciones y convenciones dirigidas a mejorar la consistencia y legibilidad del código. Algunas de las convenciones indicadas por PEP 8 son las siguientes:

- **Tabulación**: Se debe tabular utilizando 4 espacios por cada nivel de tabulación.
- **Caracteres por línea**: Cada línea de código debe tener, como máximo, una longitud de 79 caracteres.
- **Imports**: Los módulos adicionales deben ser importados al inicio de cada fichero y sólo debe haber uno por línea.
- **Espacios antes y después de operadores**: Antes y después de cada operador debe haber un único espacio.
- **Nomenclatura**: Cada elemento tiene definidas unas normas distintas al realizar su declaración.

PEP 8 también nos indica que las distintas normas pueden ser ignoradas en determinados casos como, por ejemplo:

- Cuando se añade código a una librería ya existente que sigue un estilo distinto se debe respetar dicho estilo para mantener la consistencia dentro de dicha librería.
- En caso de que el código necesite funcionar en versiones antiguas de Python y modificarlo suponga una incompatibilidad.
- Cuando modificar el código dificulte la lectura y comprensión del mismo.

3.3. Pylint

Pylint [**?**] es una herramienta de análisis de código escrito en Python que comprueba si hay incumplimiento de la guía de estilo PEP 8. Es un analizador de código estático, lo que facilita el análisis del código ya que se realiza sin tener que ejecutarlo.

Pylint identifica los errores mediante mensajes a los que se les asigna una letra seguida de un código numérico, cada letra especifica un tipo de mensaje:

- **I**: Mensajes informativos. No penalizan en la calificación emitida el final de análisis.
- **R**: Refactorizaciones para solucionar errores en la métrica.
- **C**: Convenciones para violaciones estándar del código.
- **W**: Warnings o advertencias causadas por problemas de estilo o problemas menores de programación.
- **E**: Errores debidos a problemas importantes de programación.
- **F**: Errores fatales que impiden continuar con el proceso.

La salida de Pylint puede ser configurada mediante argumentos en el momento de ejecución o mediante un fichero de configuración llamado *pylintrc*. Dicha salida se divide en 3 secciones: análisis del código fuente, informes y calificación.

Análisis del código fuente

Primera y principal sección de la salida de Pylint. Muestra los errores que Pylint ha encontrado al analizar todas las líneas del código. Por defecto, para cada error detectado indica el tipo correspondiente, en qué línea y columna se ha encontrado y el mensaje correspondiente a dicho error. Esta salida es configurable mediante un template, permitiendo mostrar los siguientes elementos a la información de cada error detectado:

- *path*: Ruta relativa del fichero analizado.
- *abspath*: Ruta absoluta del fichero analizado.
- *line*: Número de línea.

- *column*: Número de columna.
- *module*: Nombre del módulo.
- *obj*: Nombre del objeto dentro del módulo (si corresponde).
- *msg*: Texto del mensaje.
- *msg_id*: Código del mensaje.
- *symbol*: Nombre del mensaje.
- *C*: Letra identificativa de la categoría del mensaje.
- *category*: Nombre completo de la categoría del mensaje.

Informes

Segunda sección de la salida de Pylint. Por defecto, muestra un conjunto de informes acerca de distintos aspectos del código analizado, pero puede ser desactivado mediante un argumento en el momento de ejecutar el análisis. Los informes mostrados son los siguientes:

- Numero de módulos analizados.
- Para cada módulo, porcentaje de errores y warnings.
- Número total de errores y warnings.
- Porcentaje de clases, funciones y módulos que incluyen docstrings y comparación con la ejecución previa del análisis.
- Porcentaje de clases, funciones y módulos con un nombre correcto según el estándar y una comparación con la ejecución previa del análisis.
- Lista de dependencias externas encontradas en el código y dónde aparecen.

Calificación

Tercera y última sección de la salida de Pylint. Muestra una calificación global, siendo 10.0 el máximo, para el código analizado y lo compara con la calificación obtenida en la ejecución previa del análisis. Al igual que los informes, puede ser desactivado mediante un argumento en el momento de ejecutar el análisis.

3.4. Django

Django[**?**] es un framework de desarrollo de aplicaciones web de código abierto escrito en Python. Django está basado en una variación de la arquitectura MVC (Model-View-Controller) a la que denominan MVT (Model-View-Template):

- **Model**: Datos con los que el sistema trabaja. Se definen en el fichero models.py.
- **View**: Describe qué datos se presentan mediante los ficheros views.py y urls.py.
- **Template**: Definido por las plantillas HTML y CSS que describen cómo se presentan los datos.

3.5. PostgreSQL

PostgreSQL [**?**] es un sistema de gestión de bases de datos relacionales de código abierto basado en POSTGRES. PostgreSQL es compatible con gran parte del estándar SQL, añade algunas características propias, como el control de concurrencias multiversión o la compatibilidad con consultas complejas, y permite al usuario añadir nuevos tipos de datos, funciones u operadores.

3.6. HTML

HTML (Hypertext Markup Language) [**?**] es el lenguaje de marcado estándar para la creación de páginas web. Este estándar está a cargo del World Wide Web Consortium (W3C) y es compatible con todos los navegadores web. Con HTML se define la estructura de las páginas web mediante la utilización de distintas etiquetas.

3.7. CSS

CSS (Cascading Style Sheets) [**?**] es un lenguaje de diseño web utilizado para definir la apariencia y el estilo visual de una página web. Al igual que HTML, es un estándar a cargo del World Wide Web Consortium (W3C). CSS establece cómo se muestran los elementos declarados en un documento HTML definiendo propiedades como los colores, fuentes y diseño. Para ello CSS hace uso de reglas, formadas por selectores, que indican sobre qué elementos HTML se aplican, y declaraciones, que definen y modifican las propiedades. CSS es independiente de HTML y se puede llevar a cabo desde un fichero separado, lo que facilita el mantenimiento y la reutilización de las hojas de estilo.

3.8. Bootstrap

Bootstrap [**?**] es un framework de código abierto dirigido al desarrollo front-end de páginas web. Es una biblioteca que contiene herramientas HTML, CSS y JavaScript que facilitan el diseño y ayudan a realizar modificaciones de manera más sencilla en el estilo visual. Bootstrap contiene plantillas predefinidas con diferentes estilos de botones, formularios, desplegables o pestañas, entre otros componentes. Una de las características más importantes de Bootstrap es que también aporta el diseño responsive, que adapta automáticamente la presentación de la página web a diferentes tipos de dispositivos teniendo en cuenta el tamaño de pantalla y la resolución, lo que facilita la visualización y aumenta la compatibilidad de las páginas web.

3.9. JavaScript

JavaScript [**?**] es un lenguaje de programación interpretado, orientado a objetos, imperativo, multiparadigma, de tipado débil y dinámico y basado en prototipos. Tiene una sintaxis inspirada en la de Java y C++, con sentencias que funcionan igual que en esos lenguajes. Su uso principal es el scripting en páginas web dirigido a establecer el comportamiento de éstas al suceder un evento determinado. La ejecución de estos scripts se realiza en el lado del cliente, en el navegador web, sin tener acceso al lado del servidor. El uso de JavaScript está ampliamente extendido debido a que es compatible con cualquier navegador y sistema operativo.

3.10. jQuery

jQuery [**?**] es una librería de JavaScript de código abierto que simplifica la interacción entre JavaScript y HTML. jQuery facilita la manipulación del HTML, el manejo de eventos, el desarrollo de animaciones y el uso de AJAX y es compatible con cualquier navegador web y sistema operativo actual.

3.11. Chart.js

Chart.js [**?**] es una librería de JavaScript de código abierto dirigida a la representación y visualización de datos. Permite generar gráficos de distintos tipos, como barras, líneas, circulares o de dispersión, entre otros, y personalizar cómo se muestran dichos gráficos. Actualmente es una de las librerías de visualización de datos en JavaScript más populares en GitHub.

3.12. JSON

JSON [**?**] (JavaScript Object Notation) es un formato de intercambio de datos fácil de leer y escribir para humanos y fácil de generar e interpretar para las máquinas. Como su nombre indica, está basado en la sintaxis de objetos de JavaScript, teniendo una estructura formada por pares de nombre/valor. JSON es independiente de cualquier lenguaje y, por ello, es compatible con la gran mayoría de los lenguajes de programación actuales.

3.13. API-REST

Una API (Application Programming Interface) es un conjunto de instrucciones y protocolos ofrecidos por determinadas aplicaciones para dar acceso a su propia información o funciones. La API actúa como intermediario entre la propia aplicación y el solicitante y devuelve a éste último una respuesta con el resultado de la operación realizada o con la información solicitada.

Una API REST es un tipo de API en el que el acceso está basado en arquitectura REST (Representational State Transfer), por lo que las llamadas a dicha API se llevan a cabo mediante peticiones HTTP y el uso de objetos XML o JSON.

3.14. Git

Git[**?**] es un software de código abierto de control de versiones diseñado por Linus Torvald para facilitar la gestión y el trabajo en equipo en proyectos de desarrollo de software. Para un proyecto almacenado en un repositorio, Git permite que cada integrante del equipo tenga una versión en local del proyecto y vaya subiendo sus modificaciones al repositorio.

El funcionamiento de Git se basa en la creación, uso y mezcla de ramas dentro del proyecto, existiendo una rama principal, conocida como master, y creándose ramas secundarias para realizar las modificaciones en el código. Una vez se realicen las modificaciones, la rama secundaria se mezcla con la rama master y cuando otros miembros del equipo actualicen su rama master descargarán los cambios realizados sin que esto afecte a su trabajo.

3.15. GitHub

GitHub [**?**] es un servicio web, propiedad de Microsoft, que almacena repositorios de proyectos de desarrollo software que utilizan el sistema de control de versiones Git, siendo actualmente la plataforma de almacenamiento de repositorios más usada. Permite la creación de repositorios públicos, que podrán ser vistos y descargados por cualquier usuario, y repositorios privados, que únicamente serán visibles por los colaboradores de dicho repositorio.

Pull-Request

Cualquier usuario puede colaborar en proyectos públicos mediante la creación peticiones conocidas como Pull-Requests, que permiten que un usuario informe al dueño de un repositorio de los cambios que ha realizado para dicho repositorio. Para esto, un usuario externo a un repositorio crea un fork del mismo, crea una nueva rama dentro de ese fork para llevar a cabo las modificaciones y, una vez realizado los commits y push, puede crear la Pull-Request. Durante la creación de una Pull-Request se le da un nombre y una descripción a la misma y se tiene la posibilidad de visualizar y revisar los commits incluidos junto a las diferencias entre el código del repositorio original y el de la Pull-Request. Una vez creada una Pull-Request el usuario que la ha creado puede seguir realizando commits sobre la misma y además otros usuarios pueden participar en las modificaciones llevadas a cabo y sugerir cambios, añadir comentarios

o realizar nuevos commits sobre la Pull-Request. Cuando el propietario del repositorio original esté de acuerdo con los cambios propuestos en la Pull-Request, puede aceptarla, haciendo *merge* y fusionando así el contenido de la Pull-Request con el del repositorio original.

3.16. GitLab

GitLab [**?**] es un servicio web de código abierto que almacena repositorios de proyectos de desarrollo software que utilizan el sistema de control de versiones Git. Al igual que GitHub, permite la creación de repositorios públicos y privados y tiene un sistema de Pull-Requests que, en este caso, se denominan Merge-Requests. GitLab permite la instalación en un servidor privado para no hacer uso de la plataforma pública que se encuentra abierta a todo el mundo.

3.17. Heroku

Heroku [**?**] es una plataforma como servicio (PaaS) que permite alojar aplicaciones en la nube. Es compatible con aplicaciones desarrolladas en Node.js, Ruby, Python, Java, PHP, Go, Scala y Clojure y con bases de datos PostgreSQL, MongoDB y Redis.

Capítulo 4

Diseño e implementación

En este capítulo se describe de manera detallada el diseño y la implementación del proyecto.

4.1. Arquitectura general

La arquitectura de la aplicación se basa en el modelo cliente-servidor representado en la figura 4.1. En este modelo existen dos partes diferenciadas: cliente y servidor.

Cliente y servidor puede encontrarse en la misma máquina, pero en el caso de este proyecto siempre van a estar en máquinas separadas, ya que la aplicación se encuentra desplegada en Heroku y la comunicación entre ambas partes se lleva a cabo a través de internet mediante protocolo HTTP.

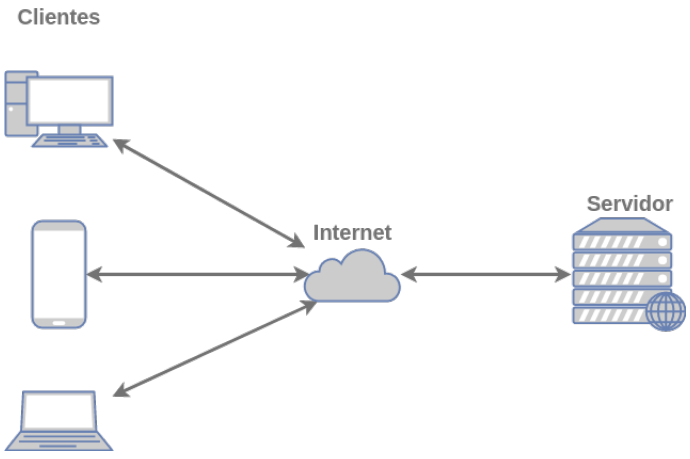

Figura 4.1: Modelo cliente-servidor.

4.1.1. Cliente

La parte cliente está formada por el navegador web desde el que el usuario accede a la aplicación. El navegador recibe la página web en forma de código HTML y lo interpreta para mostrarlo al usuario para su interacción. Las acciones que el usuario lleve a cabo sobre la página web son convertidas en peticiones HTTP por el navegador web y se envían al servidor, que las procesa y devuelve una respuesta al cliente en forma de código HTML. Desde la parte cliente no se realiza ninguna ejecución a excepción de los scripts JavaScript que puedan encontrarse dentro del HTML.

4.1.2. Servidor

La parte servidor está desplegada en Heroku y contiene la aplicación desarrollada en Django y una base de datos PostgreSQL tal y como se ve reflejado en la figura 4.2.

El servidor desplegado en Heroku recibe las peticiones HTTP realizadas sobre una determinada URL y Django las procesa realizando las acciones correspondientes para la petición recibida y, si es necesario, realiza una consulta a la base de datos PostgreSQL pudiendo ser ésta para obtener o modificar datos almacenados en dicha base de datos. Por último, el servidor devuelve al cliente el código HTTP con la respuesta a la petición realizada.

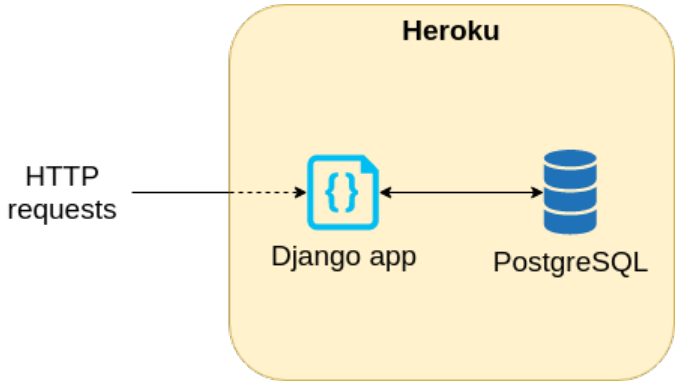

Figura 4.2: Arquitectura del servidor.

4.2. Diseño e implementación del servidor

El funcionamiento global de la aplicación y sus etapas son las representadas en la figura 4.3.

El inicio del uso de la aplicación es la introducción en un formulario de la URL del repositorio que se va analizar. Una vez comprobado que la URL introducida apunta a un repositorio, se descargan los datos del mismo y se lleva a cabo la primera de las etapas principales, el análisis del código mediante la utilización de Pylint. Una vez se ha analizado el código y se ha confirmado que se quieren llevar a cabo las modificaciones correspondientes a los errores detectados por Pylint, se realiza un fork del repositorio, se descarga el código de dicho fork y se realiza la segunda etapa principal, la corrección de los errores. Cuando se ha terminado con la corrección se llega al final del proceso con la realización de una Pull-Request/Merge-Request con las modificaciones realizada en el código.

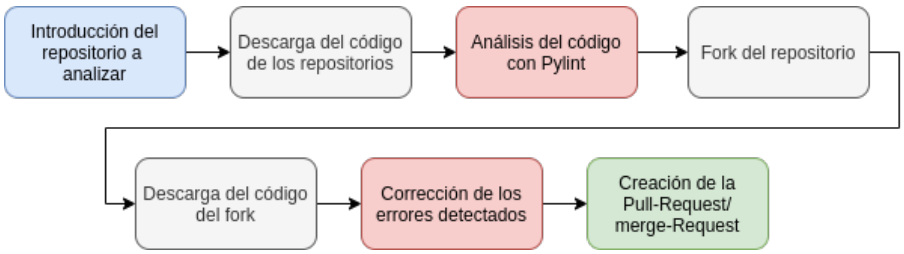

Figura 4.3: Etapas del uso de la aplicación.

4.2.1. Django

Como ya se ha explicado en la sección 3.4, Django se basa en una arquitectura MVT como la representada en la figura 4.4.

Las peticiones HTTP son procesadas por la parte *Views* que realiza las acciones correspondientes y, si es necesario, se comunica *Model* para obtener o modificar la información almacenada en la base de datos. Una vez realizado el procesamiento, es la parte *Templates* la que se encarga de construir la respuesta HTTP a partir de los datos proporcionados por *Views* y las plantillas HTML.

Esta respuesta HTTP es enviada de vuelta al cliente, que visualiza la página construida por *Templates*, mostrando los datos de *Model* que han sido procesados por *Views*.

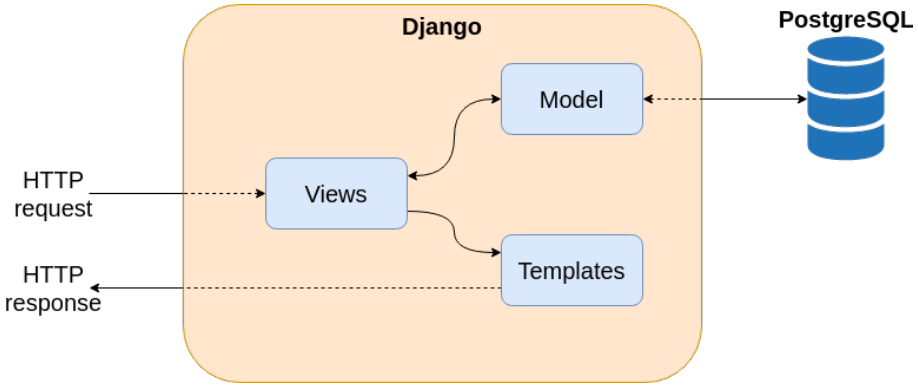

Figura 4.4: Arquitectura Django.

Directorios y ficheros

En la figura 4.5 se pueden observar los directorios y ficheros que componen la parte Django de la aplicación. Los ficheros y directorios más importantes y sus funciones son los siguientes:

- **models.py**: Define el modelo de datos a utilizar en la base de datos de la aplicación. Dentro de la arquitectura MVT de Django, pertenece a *Model*.

- **views.py**: En él se definen las vistas encargadas de leer y procesar el contenido de las peticiones HTTP realizadas a la aplicación para después realizar las acciones correspondientes. Como su propio nombre indica, se encuentra dentro de la parte *Views* definida en el modelo de arquitectura MVT.

- **urls.py**: A través de este fichero se gestiona el acceso a los recursos (URLs) de la aplicación. En él se indican, mediante el uso de expresiones regulares, los recursos que son accesibles en la aplicación y se asigna a cada uno una función definida en el fichero views.py. Junto a views.py, pertenece a la parte *Views* definida en el modelo de arquitectura MVT.

- **pylint_errors.py**: Fichero que contiene las funciones encargadas de corregir los errores analizados por Pylint.

- **tests.py**: Fichero que contiene los tests unitarios desarrollados para comprobar que las funciones de la aplicación se comportan correctamente.

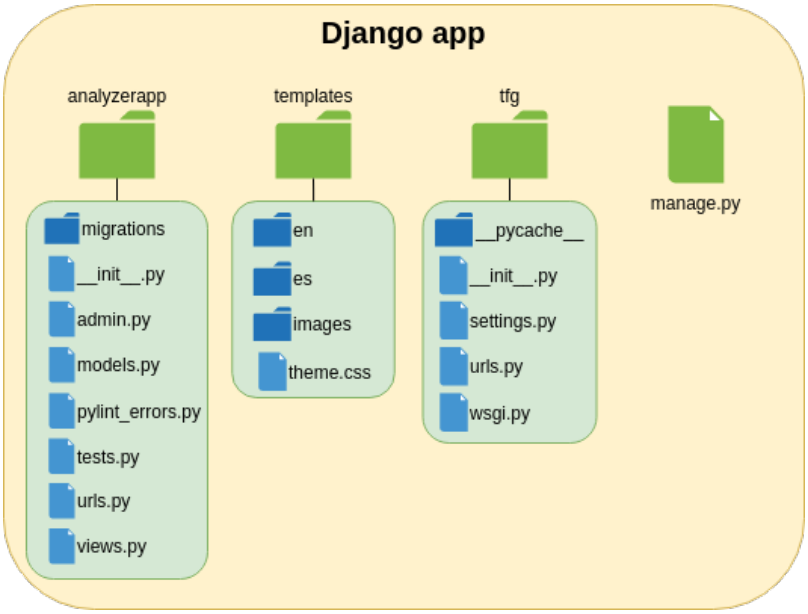

Figura 4.5: Directorios y ficheros de Django.

- **template**: Directorio que contiene las plantillas HTML que van a formar el cuerpo de la respuesta HTTP que se da al cliente. En su interior se encuentran los directorios *es*, que contiene las plantillas en castellano, *en*, que contienen las plantillas en inglés e *images*, que contiene las imágenes utilizadas por las plantillas HTML. También se encuentra el fichero *themes.css*, que actúa como hoja de estilo para toda la aplicación. Los ficheros y directorios de su interior confirman la parte *Templates* del modelo de arquitectura MVT.

Recursos

Los recursos accesibles definidos en *urls.py* son los siguientes:

- **/**: Recurso que devuelve la página principal de la aplicación. Llama a la vista *views.main*.
- **/repo/id**: Muestra la información correspondiente al repositorio del id indicado y permite llevar a cabo el análisis del mismo y la corrección de los errores detectados. Llama a la vista *views.repo*.
- **/list**: Muestra la información de todas las Pull-Requests/Merge-Requests que se han llevado a cabo. Llama a la vista *views.list*.

- **/error-list**: Muestra dos gráficas, una con el recuento del número de veces que se ha corregido cada error y otra con el estado actual de las Pull-Requests/Merge-Requests realizadas. Llama a la vista *views.error_list*.
- **/guide**: Contiene una guía de uso de la aplicación. Llama a la vista *views.guide*.
- **/contact**: Muestra los datos de contacto. Llama a la vista *views.contact*.
- **/es**: Cambia el idioma del usuario a castellano. Llama a la vista *views.es*.
- **/en**: Cambia el idioma del usuario a inglés. Llama a la vista *views.en*.

Vistas

Las vistas de Django definidas en el fichero *views.py* y sus funciones son las siguientes:

- **main**: Si recibe un GET, devuelve el template *main.html*. Mientras que si recibe un POST evalúa el contenido introducido en el formulario y, dependiendo de éste, pueden producirse varias situaciones:
  - El contenido está vacío: se devuelve el template *error.html*.
  - El contenido no corresponde a un repositorio: se devuelve el template *error_repo.html*.
  - Se ha introducido un repositorio que tiene una Pull-Request/Merge-Request realizada por la aplicación en estado abierto: se devuelve el template *request_exists.html*.
  - Se ha introducido un repositorio sin ninguna Pull-Request/Merge-Request realizada por la aplicación en estado abierto: se redirige al recurso */repo/id*.
- **repo**: Si recibe un GET, devuelve el template *repo_data.html*. Mientras que si recibe un POST evalúa el nombre del formulario que ha realizado la petición y, dependiendo de éste, pueden producirse varias situaciones:
  - *pylint*: Se lanza la función asíncrona que ejecuta el análisis del código utilizando Pylint y se devuelve el template *running_pylint.html*.
  - *running*: Espera un tiempo determinado y comprueba si la función asíncrona que ejecuta el análisis del código ha finalizado. En caso de que haya finalizado, devuelve

el template *repo_data_pylint.html*, mientras que si el análisis sigue en ejecución, devuelve otra vez el template *running_pylint.html*.

- *fix_errors*: Se lanza la función asíncrona que ejecuta la corrección de los errores detectados por Pylint y se devuelve el template *fixing_errors.html*.
- *fixing*: Espera un tiempo determinado y comprueba si la función asíncrona que ejecuta la corrección del código ha finalizado. En caso de que haya finalizado, devuelve el template *repo_data_success.html*, mientras que si el análisis sigue en ejecución, devuelve otra vez el template *fixing_errors.html*.

- **list**: Obtiene la lista de repositorios analizados y devuelve el template *list.html*.
- **error_list**: Obtiene el número de errores corregidos y el estado de las Pull-Requests/Merge-Requests realizadas y devuelve el template *error_list.html*.
- **guide**: Devuelve el template *guide.html*.
- **contact**: Devuelve el template *contact.html*.
- **es**: Cambia el valor de la variable de sesión correspondiente al idioma para visualizar el sitio en castellano y redirige a la página principal.
- **en**: Cambia el valor de la variable de sesión correspondiente al idioma para visualizar el sitio en inglés y redirige a la página principal.

Además, todas las vistas tienen una parte común al recibir un GET: comprueban el estado de las Pull-Requests/Merge-Requests para actualizarlo en la base de datos si éste ha cambiado y también comprueban el valor de la variable de sesión correspondiente al idioma para así devolver las templates en castellano o en inglés según corresponda.

Templates

Las plantillas HTML del proyecto se encuentran divididas en los directorios *templates/en* y *templates/es*. Las plantillas de un directorio son prácticamente iguales a las del otro, diferenciándose únicamente en el idioma de los textos.

Las plantillas y sus contenidos son las siguientes:

- **base.html**: Fichero base que extienden el resto de plantillas. Define el banner, el fondo y el menú lateral y su funcionamiento mediante un script.

- **main.html**: Contiene el formulario de introducción de repositorios.

- **error.html**: Muestra el mensaje de error que se le pase como parámetro.

- **error_repo.html**: Muestra un mensaje de error indicando que el formulario no se ha rellenado con una URL que pertenezca a un repositorio.

- **request_exists.html**: Informa de que ya existe una Pull-Request/Merge-Request abierta para el repositorio indicado y muestra una tabla con la información de dicho repositorio y la petición realizada.

- **repo_data.html**: Muestra una tabla con la información del repositorio introducido y un formulario para ejecutar el análisis con Pylint. Dicho formulario es sustituido por una imagen en movimiento al ser pulsado.

- **running_pylint.html**: Muestra una tabla con la información del repositorio introducido y una imagen y mensaje informando de que se está realizando el análisis. Mediante un script, al terminar la carga realiza una redirección ejecutando un POST al recurso */repo/id* con un body con el mensaje "running".

- **repo_data_pylint.html**: Muestra una tabla con la información del repositorio introducido, los errores detectados en el análisis realizado por Pylint y un botón para ejecutar la corrección. Mediante pestañas se puede filtrar para ver todos los errores o sólo los errores corregibles por la aplicación.

- **fixing_errors.html**: Muestra una tabla con la información del repositorio introducido y una imagen y mensaje informando de que se está realizando la corrección del código. Mediante un script, al terminar la carga realiza una redirección ejecutando un POST al recurso */repo/id* con un body con el mensaje "fixing".

- **repo_data_success.html**: Muestra una tabla con la información del repositorio introducido y añade una nueva fila con el enlace a la Pull-Request/Merge-Request realizada.

- **list.html**: Muestra una tabla con la lista de las Pull-Requests/Merge-Requests realizadas por la aplicación. Mediante pestañas se puede realizar un filtrado según el estado de cada una.

- **error_list.html**: Hace uso de la librería de JavaScript Chart.js 3.11 para representar dos gráficas, una con el recuento de errores y otra con el estado de las Pull-Requests/Merge-Requests realizadas.

- **guide.html**: Contiene un resumen sobre la aplicación, una breve guía de uso definiendo cada una de las páginas y una tabla con los errores detectados por Pylint que son corregibles por la aplicación.

- **contact.html**: Contiene los datos de contacto.

- **theme.css**: Hoja de estilo para personalizar el aspecto de los elementos HTML de la aplicación.

4.2.2. Errores de Pylint

En el fichero *pylint_errors.py* se definen qué errores detectados por Pylint van a ser corregidos y cómo va a llevarse a cabo la corrección. El listado de errores corregibles por la aplicación se puede visualizar en la tabla 4.2.2.

Pylint indica dónde aparecen los errores mediante el número de línea en el que se encuentran, por lo que para realizar la corrección se ha hecho uso de placeholders que se añaden al código e indican si una determinada línea tiene un error y de qué tipo es. De esta forma no se cambian las líneas de posición y se evitan posibles modificaciones en las líneas incorrectas. Una vez terminada la colocación de los placeholders para todo el código, se vuelve a escanear el fichero línea por línea y se aplica la corrección correspondiente para cada placeholder encontrado.

4.2.3. Base de datos

La estructura de la base de datos queda definida en el fichero *models.py* de Django y tiene la forma representada en la figura 4.6. Está compuesta por las siguientes tablas:

Error ID	Error Name	Message
C0303	trailing-whitespace	Trailing whitespace. Used when there is whitespace between the end of a line and the newline.
C0304	missing-final-newline	Final newline missing. Used when the last line in a file is missing a newline.
C0321	multiple-statements	More than one statement on a single line. Used when more than on statement are found on the same line.
C0326	bad-whitespace	%s space %s %s %s. Used when a wrong number of spaces is used around an operator, bracket or block opener.
C0410	multiple-imports	Multiple imports on one line (%s). Used when import statement importing multiple modules is detected.
C0411	wrong-import-order	%s should be placed before %s. Used when PEP8 import order is not respected (standard imports first, then third-party libraries, then local imports).
C0413	wrong-import-position	Import " %s"should be placed at the top of the module. Used when code and imports are mixed.
W0404	reimported	Reimport %r (imported line %s). Used when a module is reimported multiple times.
W0611	unused-import	Unused %s. Used when an imported module or variable is not used.

Cuadro 4.1: Errores detectados por Pylint corregibles por la aplicación.

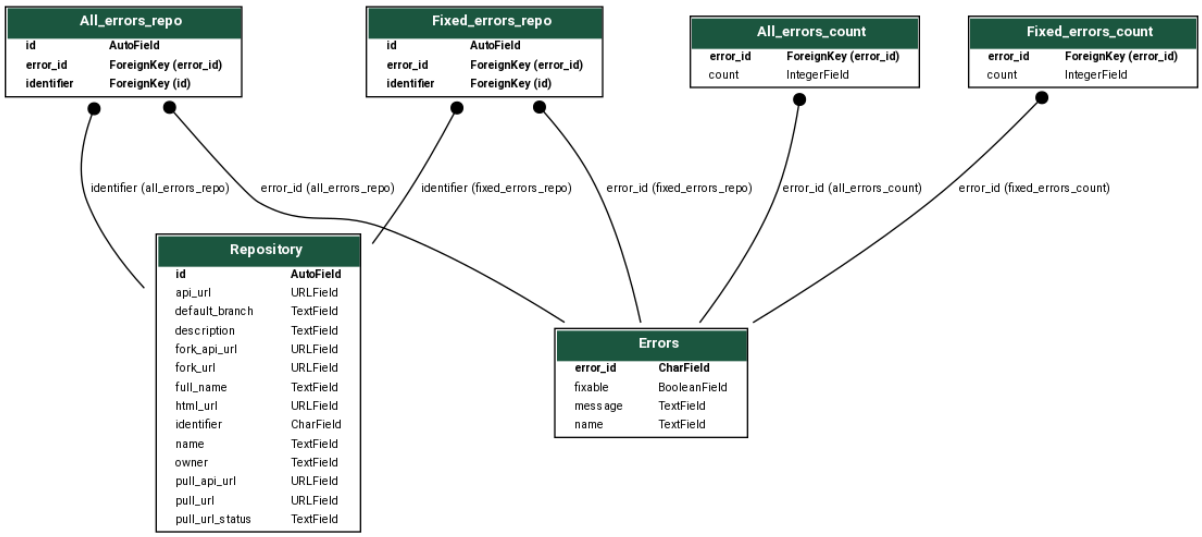

Figura 4.6: Modelo de datos.

- **Errors**: Contiene la información relativa a todos los errores detectables por Pylint y si son corregibles o no por la aplicación.

- **Repository**: Almacena la información de todos los repositorios que han sido analizados por la aplicación y las Pull-Requests/Merge-Requests que se han llevado a cabo.

- **All_errors_repo**: Guarda los errores que se han detectado en cada repositorio.

- **Fixed_errors_repo**: Guarda los errores que se han detectado en cada repositorio y además han sido corregidos por la aplicación.

- **All_errors_count**: Almacena el número de veces que se ha detectado cada error.

- **Fixed_errors_count**: Almacena el número de veces que se ha corregido cada error.

Es una base de datos PostgreSQL provisionada por Heroku y externa a la aplicación Django.

4.2.4. Despliegue

El despliegue de la aplicación se ha configurado para que se lleve a cabo de manera automática mediante el uso de las GitHub Actions y Heroku.

Github

Cuando se realiza un push y se suben los nuevos commits a GitHub, se ejecutan automáticamente las GitHub Actions definidas en el directorio */.github/workflows/*.

El fichero *main.yml*, ubicado en dicho directorio, hace uso de la GitHub Action *akhileshns/heroku-deploy@v3.12.12* [**?**], logrando así que la aplicación se despliegue automáticamente en Heroku cada vez que se hace push al repositorio que contiene el código.

Heroku

Para llevar a cabo el despliegue de la aplicación en Heroku es necesaria la configuración de dos ficheros, *Procfile* y *requirements.txt*.

Procfile especifica los comandos que se van a ejecutar durante el despliegue de la aplicación en Heroku. Para esta aplicación se especifica que se deben realizar las migraciones del modelo de datos en el momento del lanzamiento, o *release*, por si se hubiese llevado a cabo alguna modificación en *models.py*, y se ejecuta el comando *runserver* como un proceso de tipo web para que sea accesible y pueda recibir peticiones HTTP desde fuera de Heroku.

El fichero *requirements.txt* hace que Heroku identifique automáticamente la aplicación como una aplicación desarrollada en Python y en su interior se definen las dependencias, incluyendo la versión, que deben ser instaladas antes del arranque de la aplicación en Heroku. Para esta aplicación *requirements.txt* contiene paquetes Python como *Django*, *Pylint*, o *requests*, entre otros.

Además, para que Django funcione correctamente al desplegarse en Heroku, hay que realizar cambios dentro del código del proyecto. El fichero *settings.py* se modifica para que Django utilice la base de datos PostgreSQL proporcionada por Heroku y se modifica el fichero *views.py* para que haga uso de las variables de configuración de Heroku, que contendrán datos como los token de GitHub y Gitlab.

Capítulo 5

Experimentos y validación

5.1. Tests unitarios

Para comenzar las pruebas del correcto funcionamiento de la aplicación se ha hecho uso de la herramienta de tests unitarios incluida en Django. Dichos tests unitarios se encuentran en el fichero */analyzerapp/tests.py* y están diseñados para comprobar que las funciones correspondientes a cada error son capaces de situar correctamente el placeholder indicado en las líneas afectadas, y que las funciones que corrigen los errores al encontrar un placeholder son capaces de hacerlo correctamente.

```
test_c0303 (analyzerapp.tests.PylintErrorsTestCase)
'Trailing whitespace' eliminados correctamente ... ok
test_c0304 (analyzerapp.tests.PylintErrorsTestCase)
'Final newline missing' corregido correctamente ... ok
test_c0321 (analyzerapp.tests.PylintErrorsTestCase)
'More than one statement in a single line' marcados correctamente ... ok
test_c0326 (analyzerapp.tests.PylintErrorsTestCase) ... ok
test_c0410 (analyzerapp.tests.PylintErrorsTestCase)
'Multiple imports in one line' marcados correctamente ... ok
test_c0411 (analyzerapp.tests.PylintErrorsTestCase)
'wrong-import-order: %s comes before %s' marcados correctamente ... ok
test_c0413 (analyzerapp.tests.PylintErrorsTestCase)
'Import should be placed at the top of the module' marcados correctamente ... ok
test_placeholder_external (analyzerapp.tests.PylintErrorsTestCase)
Placeholder #EXT eliminado y línea corregida correctamente ... ok
test_placeholder_importsplit (analyzerapp.tests.PylintErrorsTestCase)
Placeholder #IMPORTSPLIT eliminado y línea corregida correctamente ... ok
test_placeholder_split (analyzerapp.tests.PylintErrorsTestCase)
Placeholder #SPLIT eliminado y línea corregida correctamente ... ok
test_placeholder_top (analyzerapp.tests.PylintErrorsTestCase)
Placeholder #TOP eliminado y línea corregida correctamente ... ok
test_w0404 (analyzerapp.tests.PylintErrorsTestCase)
'Reimport' marcado correctamente ... ok
test_w0611 (analyzerapp.tests.PylintErrorsTestCase)
'Unused import' marcado correctamente ... ok

----------------------------------------------------------------------
Ran 13 tests in 0.008s

OK
```

Figura 5.1: Salida de la ejecución de los tests unitarios en Django.

La salida de la ejecución de los tests unitarios desde el propio Django puede ser observada en la figura 5.1. En ella se refleja que la ejecución de los tests unitarios es exitosa tanto para los nueve tests correspondientes a las funciones que añaden los placeholders, como para los cuatro tests correspondientes a la corrección de los errores y borrado de los placeholders.

5.2. Análisis y corrección de proyectos en local

Tras comprobar el correcto funcionamiento de las funciones de corrección de errores mediante los tests unitarios, el siguiente paso ha sido realizar pruebas con proyectos en local para comprobar que se realiza correctamente tanto el análisis con Pylint como la corrección de los errores detectados. En estas primeras pruebas no se ha ejecutado la aplicación utilizando Django, se ha creado un script con el contenido de las funciones del fichero *views.py* y se ha ejecutado el análisis y la corrección ejecutando dicho script.

El primer paso ha sido realizar las pruebas con un único fichero, para ello se ha creado dos ficheros con las mismas sentencias de Python. El primero de ellos contiene errores corregibles por la aplicación, mientras que el segundo de ellos cumple con todas las normas de estilo que indican los errores que la aplicación es capaz de corregir.

Es durante esta prueba cuando se observa que, al realizar las correcciones del código al mismo tiempo que se recorre el fichero con la salida de Pylint, algunas sentencias pueden sufrir cambios en su número de línea, lo que puede provocar que se realicen modificaciones en las líneas equivocadas y el fichero no pueda ejecutarse por errores de sintaxis. Para solucionar este problema se toma la decisión de no realizar acciones que puedan cambiar los números de línea de ninguna sentencia mientras se lee la salida de Pylint y, para ello, únicamente se llevará a cabo la colocación de placeholders durante esta lectura. Estos placeholders indican qué acción hay que realizar sobre la línea en la que se encuentran y, si fuese necesario, un número indicando en qué posición de la línea se encuentra el problema. Al terminar de situar los placeholders en el fichero se vuelve a recorrer el mismo buscando dichos placeholders y realizando la acción correspondiente para corregir el error.

Tras solventar dicho problema, se procede a ejecutar la aplicación sobre el fichero que contiene los errores y se observa que ambos fichero son totalmente idénticos, por lo que se confirma que la aplicación ha funcionado correctamente al ser ejecutada sobre un único fichero con erro-

res.

Después de comprobar que el análisis y la corrección sobre un único fichero ha tenido éxito, el siguiente paso es realizarlo sobre un proyecto que contenga distintos directorios y ficheros, incluyendo tanto ficheros con extensión *.py* como ficheros con otras extensiones o sin extensión. Para ello se han creado copias del fichero inicial que contenía los errores y se han distribuido en diferentes directorios dentro de un mismo proyecto realizando cambios en el nombre para poder tener varios en el mismo directorio y eliminando la extensión para comprobar que no se ven afectados por el análisis. Tras ejecutar la aplicación sobre el directorio raíz del proyecto se comprueba que ésta funciona correctamente, ya que todos los ficheros con extensión *.py* contenidos en los diferentes directorios y subdirectorios han sido corregidos y son idénticos al fichero inicial sin errores corregibles, mientras que los ficheros sin extensión *.py* no se han visto modificados por la aplicación.

5.3. Análisis y corrección de repositorios en GitHub y GitLab

Para poder comprobar que la aplicación al completo funciona correctamente se ha dejado de usar el script indicado en la sección 5.2 y se ha levantado la aplicación Django en local. Tras tener la aplicación Django funcionando correctamente, se ha creado un repositorio en GitHub que contiene un único fichero como el indicado en la primera prueba exitosa de la sección 5.2 y se ha introducido la URL de dicho repositorio en el formulario de la página principal de la aplicación. Una vez terminado el proceso, se comprueba que se ha realizado la Pull-Request y que ésta contiene un commit modificando el único fichero del repositorio que, en caso de realizarse el *merge* de la Pull-Request con el repositorio, pasa a ser idéntico al fichero inicial sin los errores corregibles por la aplicación. También se comprueba que la Pull-Request realizada contiene en su descripción un breve mensaje con un enlace a la aplicación y a la documentación de la guía de estilo PEP8 y después el listado con los errores que han sido corregidos por la aplicación tal y como los muestra Pylint en su salida.

En este proceso se recibe un mensaje de GitHub notificando que la autenticación mediante usuario y contraseña al usar la API va a ser deshabilitada próximamente, por lo que se modifica el método de autenticación para usar un token de acceso y se comprueba que el proceso de creación de la Pull-Request sigue funcionando correctamente.

Tras esta corrección y la creación de la Pull-Request de manera exitosa, se realiza el mismo proceso utilizando el proyecto con múltiples directorios y ficheros que se ha usado en la segunda parte de la sección 5.2. Al finalizar el proceso se observa que se ha realizado la creación de la Pull-Request de manera adecuada, modificando correctamente todos los ficheros con extensión *.py* y dejando intactos los que no tienen ninguna extensión. Por último, se prueba a aceptar una de las Pull-Requests y a cancelar la otra y se comprueba que la aplicación realiza la consulta a la API de GitHub para obtener el estado de las Pull-Requests y lo actualiza correctamente en la base de datos.

Estos dos mismos experimentos se repiten utilizando un repositorio alojado en el GitLab de la ETSIT de la URJC en lugar de GitHub y, como en el caso anterior, ambas pruebas concluyen de manera satisfactoria.

5.4. Aplicación desplegada en Heroku

Como última prueba, se realiza el despliegue de la aplicación en Heroku y se repiten las pruebas utilizando repositorios con subdirectorios y múltiples ficheros alojados en GitHub y en el GitLab de la ETSIT de la URJC. Estas pruebas tienen el mismo resultado que al haber desplegado la aplicación Django en local, por lo que se puede validar el funcionamiento correcto de la aplicación completa ya desplegada en Heroku y siendo accesible desde Internet, finalizando así las pruebas y tests ejecutados sobre la aplicación.

Capítulo 6

Resultados

Una vez terminada la fase de pruebas, se borra el contenido de la base de datos para no mezclar los datos de las pruebas con los datos reales y se procede a utilizar la aplicación con repositorios reales alojados en GitHub. Para ello se escogen algunos proyectos correspondientes a la práctica final de la asignatura Servicios y Aplicaciones Telemáticas realizadas por compañeros de la carrera y se buscan proyectos Python en GitHub que tengan actividad reciente.

Al introducir el primero de estos proyectos, debido a su tamaño, el tiempo de análisis con Pylint es superior al de las pruebas realizadas previamente y se produce un *timeout* ya que el servidor tarda más de treinta segundos en devolver una respuesta HTTP al cliente.

Para solucionarlo se intenta cambiar el valor de dicho *timeout*, pero tras revisar la documentación de Heroku se observa que no es posible modificarlo. Por lo tanto, se procede a programar el análisis con Pylint como una tarea asíncrona con el servidor comprobando periódicamente si dicha tarea ha finalizado o no. Dicha solución se aplica también para la tarea de corrección de los errores detectados, ya que puede provocar el mismo fallo.

Tras realizar estos cambios se vuelve a introducir el repositorio que ha provocado el fallo y se comprueba que el análisis y la corrección se llevan a cabo de manera satisfactoria y que se permite que el tiempo de los mismos sea superior a treinta segundos sin provocar un *timeout*. También se comprueba que la Pull-Request se ha generado correctamente y ha quedado abierta a la espera de que el propietario del repositorio la acepte o la cancele.

Una vez realizados diez Pull-Requests sobre repositorios de proyectos reales, se procede a analizar cuáles son los errores más corregidos y más detectados por la aplicación.

En la figura 6.1 se puede observar una gráfica en la que se representa el número de errores

detectados por la aplicación para los diez más frecuentes en los repositorios analizados. Entre estos diez errores más frecuentes se encuentran dos de los que son corregibles por la aplicación: *C0326: bad-whitespace* y *C0303: trailing-whitespace*. El resto de errores se encuentran junto a su ID y su mensaje de error en la tabla 6.

Entre los errores detectados por la aplicación, el más repetido es *C0326: bad-whitespace*, que además es corregible, habiendo sido detectado un total de 2541 veces. Los dos siguientes errores también superan las mil repeticiones totales para los diez análisis realizados. Estos errores son *C0103: invalid-name*, repetido 1465 veces y *C0116: missing-function-docstring*, con 1146 detecciones. El resto de errores detectados se encuentran por debajo de las 600 repeticiones. Para los diez análisis realizados, se han detectado 108 errores distintos, de los 349 posibles, con un recuento total de 9566 detecciones.

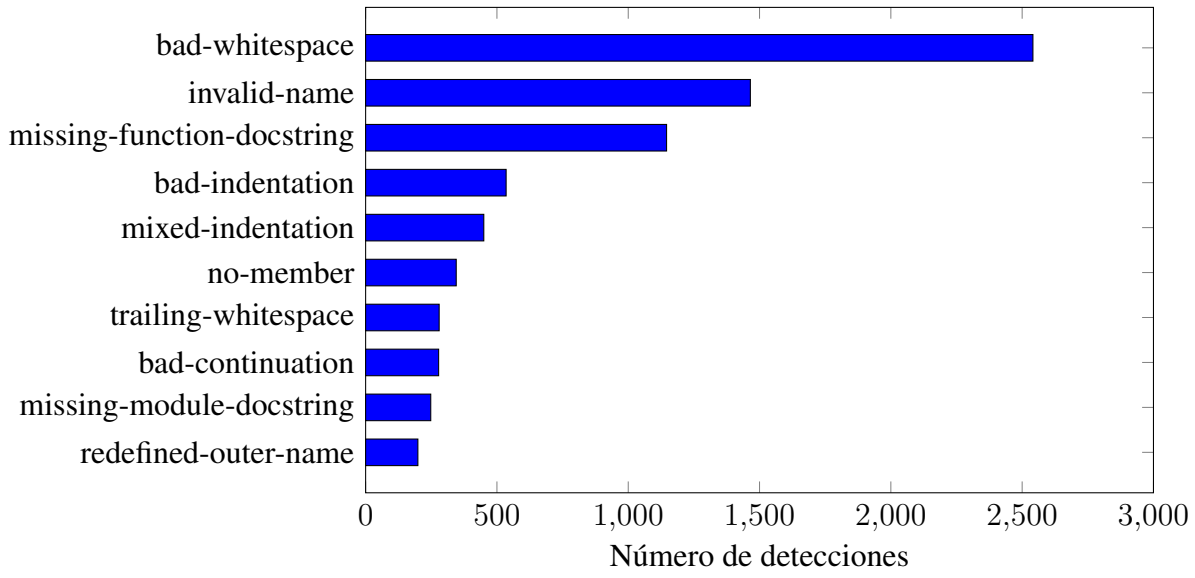

Figura 6.1: Cantidad de detecciones por error.

Error ID	Error Name	Message
C0103	invalid-name	%s name ” %s”doesn’t conform to %s. Used when the name doesn’t conform to naming rules associated to its type (constant, variable, class...).
C0116	missing-function-docstring	Missing function or method docstring. Used when a function or method has no docstring.Some special methods like __init__ do not require a docstring.
W0311	bad-indentation	Bad indentation. Found %s %s, expected %s. Used when an unexpected number of indentation’s tabulations or spaces has been found.
W0312	mixed-indentation	Found indentation with %ss instead of %ss. Used when there are some mixed tabs and spaces in a module.
E1101	no-member	%s %r has no %r member %s. Used when a variable is accessed for an unexistent member.
C0330	bad-continuation	Wrong %s indentation %s %s.
C0114	missing-module-docstring	Missing module docstring. Used when a module has no docstring.Empty modules do not require a docstring.
W0621	redefined-outer-name	Redefining name %r from outer scope (line %s). Used when a variable’s name hides a name defined in the outer scope.

Cuadro 6.1: Errores detectados por Pylint corregibles por la aplicación.

En la figura 6.2 se puede observar una gráfica en la que se representa el número correcciones que se han llevado a cabo para cada uno de los errores que son corregibles por la aplicación.

Los 9 errores corregibles por la aplicación han sido detectados y solucionados al menos una vez durante los 10 análisis realizados. Los dos errores más corregidos son los que ya aparecían en la lista de errores más detectados, con 2541 repeticiones para *C0326: bad-whitespace* y 280 para *C0303: trailing-whitespace*. Para los siguientes errores corregidos, *C0321: multiple-statements* se ha repetido 190 veces, *C0304: missing-final-newline* ha tenido 113 apariciones

y *W0611: unused-import* ha sido corregido 108 veces. Con menos de 100 correcciones se encuentran *C0411: wrong-import-order*, con 84, y *C0413: wrong-import-position*, que ha sido corregido 64 veces. Mientras que los dos errores con menos correcciones son *W0404: reimported* y *C0410: multiple-imports* con 9 y 7 correcciones cada uno respectivamente Estos 9 errores suman un total de 3396 correcciones para los 10 análisis que se han llevado a cabo.

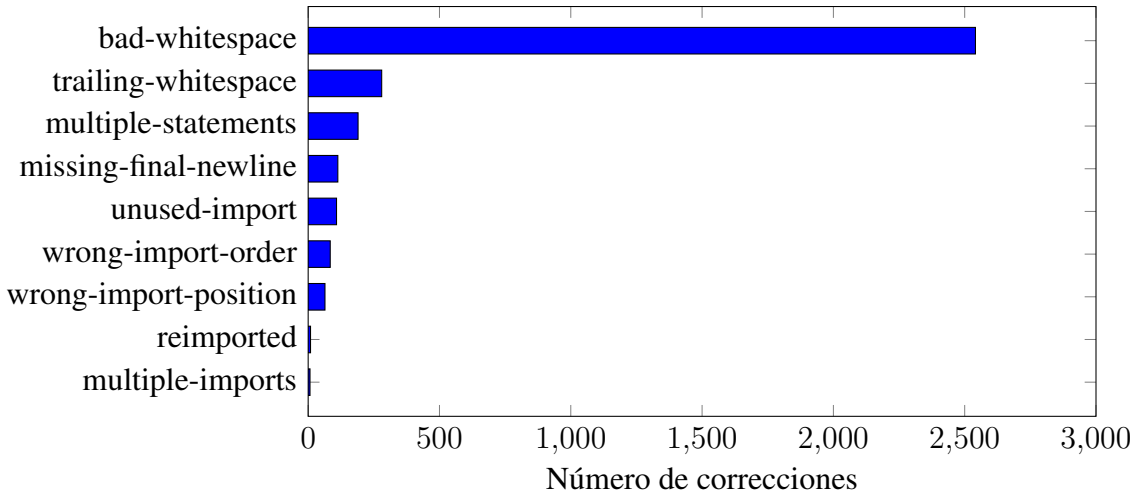

Figura 6.2: Cantidad de correcciones por error.

Capítulo 7

Conclusiones

7.1. Consecución de objetivos

Como se ha indicado en la sección 2.1, el objetivo principal del proyecto era desarrollar una aplicación que analizase mediante Pylint repositorios de GitHub con código Python y corrigiera algunos de los errores encontrados en dicho código para cumplir con la guía de estilo PEP8 realizando finalmente Pull-Requests para sugerir al propietario del repositorio las modificaciones que se han llevado a cabo. Este objetivo principal se ha cumplido en su totalidad, la aplicación es capaz de analizar código Python almacenado en GitHub y corrige los 9 errores indicados en la tabla 4.2.2 para, finalmente, subir estas modificaciones a GitHub mediante el uso de Pull-Requests.

Respecto a los objetivos específicos, el resultado para cada uno de ellos ha sido el siguiente:

- **Trabajar con proyectos reales**: Una vez finalizado el desarrollo de la aplicación, se ha podido probar su funcionamiento con proyectos reales.
- **Aplicación web**: La aplicación ha sido desarrollada como una aplicación web mediante el uso de Django.
- **Accesibilidad desde Internet**: Se ha logrado la accesibilidad desde Internet mediante el despliegue de la aplicación en Heroku. Este despliegue tiene algunas limitaciones causadas por el uso de la versión gratuita de Heroku. La principal limitación se encuentra en la base de datos que proporciona la plataforma, ya que no se permite que ésta contenga más de diez mil filas. Otra limitación provocada por la versión gratuita de Heroku es la entrada

en reposo de la aplicación pasado un determinado tiempo de inactividad. Esto causa que si un usuario accede a la aplicación cuando se encuentra en reposo, deberá esperar unos segundos mientras la aplicación arranca para poder hacer uso de la misma.

- **Compatibilidad con el servidor GitLab de la ETSIT**: La aplicación es totalmente compatible con el servidor GitLab de la ETSIT y permite realizar sobre sus repositorios las mismas acciones que sobre los repositorios de GitHub.

- **Análisis de aceptación de las Pull-Requests**: La aplicación comprueba el estado de las Pull-Requests que se han realizado y lo actualiza en la base de datos para mostrarlo correctamente en las gráficas.

7.2. Aplicación de lo aprendido

En el desarrollo de este proyecto se han aplicado conocimientos adquiridos en las siguientes asignaturas:

- **Fundamentos de la Programación**: Introducción a la programación, necesaria para poder entender las asignaturas de programación que se estudian más adelante.

- **Arquitectura de Redes de Ordenadores**: Esta asignatura introduce por primera vez los protocolos usados en las redes, como el protocolo HTTP, necesario para el funcionamiento de las aplicaciones web como la desarrollada en este proyecto.

- **Programación de Sistemas de Telecomunicación**: Se profundiza en el uso de protocolos, la programación de tareas asíncronas y se presenta la arquitectura cliente-servidor en la que se basa la aplicación.

- **Sistemas Operativos**: Asignatura que, entre otras cosas, explica el uso básico de Linux, necesario para el tratamiento de ficheros y ejecución de comandos del sistema operativo que realiza la aplicación.

- **Servicios y Aplicaciones Telemáticas**: Asignatura base para el desarrollo de este proyecto. En ella se tiene el primer contacto con git, Python, HTML, CSS y Django, tecnologías en las que se basa la aplicación desarrollada. También se presenta la guía de estilo PEP8.

- **Ingeniería de Sistemas de Información**: Presenta el uso de las bases de datos, el lenguaje SQL, los tests unitarios y el despliegue de aplicaciones en Heroku. También profundiza en el uso de git.

- **Desarrollo de Aplicaciones Telemáticas**: Se introduce JavaScript y jQuery y se muestra qué es una API y cómo comunicarse con ella.

7.3. Lecciones aprendidas

De lo aprendido durante la elaboración del Trabajo de Fin de Grado se podría destacar lo siguiente:

- La profundización en Python y Django, que ha permitido añadir conocimientos a lo aprendido previamente durante las asignaturas de la carrera.

- La posibilidad de realizar Pull-Requests en GitHub y así poder sugerir modificaciones y colaborar en proyectos de otros usuarios.

- GitLab. Desconocía la existencia de plataformas de almacenamiento de repositorios además de GitHub y que además permitía la creación de servidores privados en los que alojar los proyectos.

- Cómo funcionan las APIs de GitHub y GitLab y su sistema de autenticación mediante token.

7.4. Trabajos futuros

Aunque cumpla los objetivos propuestos, este proyecto tiene la posibilidad de ser mejorado y ampliado en el futuro, algunas de estas posibles mejoras son las siguientes:

- Añadir más errores corregibles por la aplicación.

- Mejorar la velocidad de corrección mediante la posibilidad de corregir los ficheros en paralelo en lugar de secuencialmente como se realiza ahora.

- Añadir compatibilidad con el servidor público y global de Gitlab además de con otras plataformas de almacenamiento de repositorios como Bitbucket.
- Despliegue en una plataforma sin las limitaciones producidas por Heroku actualmente.

Apéndice A

Manual de usuario

Esto es un apéndice. Si has creado una aplicación, siempre viene bien tener un manual de usuario. Pues ponlo aquí.