ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

DOBLE TITULACIÓN: INGENIERÍA SUPERIOR DE TELECOMUNICACIONES + INGENIERÍA TÉCNICA EN INFORMÁTICA DE SISTEMAS

PROYECTO FIN DE CARRERA

CREACIÓN DE UNA API Y UNA INTERFAZ WEB PARA GENERAR UN DASHBOARD A PARTIR DE LA HERRAMIENTA PERCEVAL

Autor : Pablo Fernández Salguero
Tutor : Jesús M. González-Barahona

Curso Académico 2016/2017

Dedicado a
mis padres / mi familia / mi novia
pero sobre todo, a mi madre.

Agradecimientos

Hace ya casi doce años que llegué a Madrid para empezar la carrera. Antes de llegar me decanté por la Universidad Rey Juan Carlos, porque ofrecía una doble titulación que nadie más tenia, Ingeniería superior de telecomunicaciones + Ingeniería técnica en informática de sistemas. A día de hoy, ya cerca de terminar, sé que tomé la decisión adecuada. Esta decisión ha marcado la forma de vivir todos estos años, desde el lugar dónde trabajo, hasta la gente con la que comparto mi día a día.

Mi paso por la universidad, ha sido un poco especial, ya que más de la mitad del tiempo que he estado matriculado en ella, no he asistido a clase por motivos laborales, esto hace que no tenga una relación especial con ninguno de los profesores, pero esto no quita que su comportamiento fuese ejemplar durante estos años, desde el primer profesor que me dio la bienvenida, hasta Jesús que ha sido mi tutor para realizar el proyecto.

También tengo que agradecer a todos los compañeros, muchos amigos, que he tenido desde que empezó todo, porque en mayor o menos medida estoy seguro de que me han ayudado para llegar hasta aquí, y que posiblemente yo también les ayudase a ellos.

Por último y más importante, tengo que dar todos los agradecimientos posibles a mis padres, **Mercedes y Juan Manuel**, que hicieron posible, en todos los aspectos, tener la opción de acceder a la universidad, y en especial a mi madre, que ya no podrá compartir conmigo el momento de terminar. También tengo que agradecer a mi novia, **Alba**, que me ha ayudado durante estos últimos años a llevar hacia delante el hecho de trabajar y estudiar, y por supuesto al resto de mi familia que han estado ahí cuando les he necesitado.

Muchas gracias.

Resumen

El objetivo principal del proyecto es facilitar al usuario el proceso de creación de un panel de representación de datos, con la información acerca de un determinado proyecto. Para ello la base será la herramienta Perceval creada por Bitergia. Por lo tanto, la intención es dotar a un posible usuario, con desconocimiento total de las tecnologías involucradas en el proyecto, de una herramienta a la que proporcionando la información necesaria sobre el elemento a analizar, cree un panel para la representación de los datos de éste.

En resumen, se trata, de hacer transparente el proceso que transcurre desde la obtención de los datos del proyecto a analizar hasta que se crea el panel de representación de datos.

El desarrollo trabajará con información de proyectos alojados en Git o GitHub, que son las plataformas de desarrollo colaborativo más habituales hoy en día.

Para poder hacer transparente este proceso, el proyecto debe crear la funcionalidad necesarias para que las principales tecnologías que participan interactúen de forma correcta entre ellas. Las principales tecnologías son Elasticsearch, Kibana, Python + Django y JSON.

La herramienta resultado del desarrollo, debe cumplir las pretensiones de dos tipos de usuario.

En primer lugar las de un posible desarrollador que pueda estar interesado en integrar la funcionalidad que se ofrece en un desarrollo posterior, e incluso un posible desarrollo tomando como partido este proyecto. Para este primer usuario el producto final debe ofrecer una API (del inglés, Application Programming Interface) para ofrecer abstracción sobre dicha funcionalidad y permitir la integración en otro software

En segundo lugar las de un posible usuario que sólo está interesado en la representación gráfica de los datos obtenidos de un determinado proyecto. Para esto es necesario construir una interfaz web capaz de proporcionar al usuario la abstracción necesaria, para que el proceso de creación sea completamente transparente desde la solicitud la información del proyecto a analizar, hasta la representación gráfica de los datos obtenidos.

Summary

The main objective of this thesis is to enable users to create a dashboard with information about GitHub projects. Since GitHub is the most used platform for free, open source software (FOSS) development, the user will be enabled to analyze most of the popular FOSS projects. The system will be based on Perceval and GrimoireELK, Python modules which are part of GrimoireLab. Those modules will retrieve the relevant data from GitHub APIs, and will store and organize it in a way suitable to produce a dashboard. Other technologies used are Django, ElasticSearch, and Kibana.

The whole process of producing a dashboard is completely transparent to users, who will only need to authenticate themselves with GitHub, and specify the projects to analyze. Once that is done, the system will automatically produce an interactive dashboard, ready for browsing, with all relevant data about commits, issues and pull requests of the project.

The system can be used via a REST API, suitable for being commanded from scripts, or via a web application, suitable for end users from a web browsers.

Índice general

1. Introducción **17**
1.1. Objetivo general . . . 17
1.2. Objetivos específicos . . . 19
1.3. Requisitos . . . 20
1.4. Planificación temporal . . . 20
1.5. Estructura de la memoria . . . 21

2. Estado del arte y tecnologías **23**
2.1. GitHub . . . 23
2.1.1. Historia . . . 23
2.1.2. Características . . . 23
2.1.3. API de GitHub . . . 25
2.2. Python . . . 26
2.2.1. Historia . . . 27
2.2.2. Características . . . 27
2.3. Django . . . 28
2.3.1. Historia . . . 29
2.3.2. Características . . . 29
2.4. Elasticsearch . . . 29
2.4.1. Uso de Elasticsearch . . . 30
2.4.2. Historia . . . 30
2.4.3. Características . . . 31
2.5. Kibana . . . 31
2.5.1. Historia . . . 32
2.5.2. Características . . . 32
2.6. JSON . . . 32
2.6.1. Historia . . . 33
2.6.2. Características . . . 33
2.6.3. Comparación con XML . . . 33
2.7. Python Social Auth . . . 34
2.7.1. Historia . . . 34
2.7.2. Características . . . 34
2.8. Requests . . . 35
2.8.1. Historia . . . 35
2.8.2. Características . . . 35
2.9. Eclipse . . . 36

2.9.1. Historia . . . 36
2.9.2. Características . . . 36
2.10. Perceval . . . 37
2.10.1. Historia . . . 37
2.10.2. Características . . . 37
2.11. GrimoireELK . . . 38
2.11.1. Historia . . . 38
2.11.2. Características . . . 38
2.12. Bootstrap . . . 38
2.12.1. Historia . . . 38
2.12.2. Características . . . 39
2.13. CSS . . . 40
2.13.1. Historia . . . 40
2.13.2. Características . . . 41
2.14. LaTeX . . . 41
2.14.1. Historia . . . 42
2.14.2. Características . . . 42

3. Desarrollo **45**
3.1. Iteración 1. Conocimiento de las herramientas . . . 45
3.1.1. Objetivos . . . 45
3.1.2. Conocer GrimoireLab . . . 45
3.1.3. Familiarizarse con Elasticsearch . . . 46
3.1.4. Script resumen de lo aprendido . . . 46
3.2. Iteración 2. Primera aproximación con Django . . . 47
3.2.1. Objetivos . . . 47
3.2.2. Django . . . 47
3.2.3. La aplicación con Django . . . 47
3.2.3.1. Creación del formulario POST . . . 47
3.2.3.2. Almacenamiento en Elasticsearch de los datos obtenidos con el POST . . . 49
3.2.3.3. Adaptación del script para la creación del dashboard . . . 50
3.3. Iteración 3. Integración de la aplicación con GitHub . . . 51
3.3.1. Objetivos . . . 51
3.3.2. Aplicaciones en GitHub . . . 51
3.3.3. La API de GitHub . . . 53
3.3.4. Python Social Auth . . . 55
3.3.4.1. Instalación de *Python Social Auth*: . . . 56
3.3.4.2. Configuración de *Python Social Auth*: . . . 56
3.3.4.3. Configuración en GitHub . . . 57
3.3.5. Interfaz de prueba: . . . 58
3.3.6. Obtención del access_token: . . . 59
3.4. Iteración 4. Consolidación de la aplicación . . . 61
3.4.1. Objetivos . . . 61
3.4.2. Campos adicionales en el modelo . . . 61
3.4.3. Almacenamiento en Elasticsearch del access_token y el usuario . . . 62

3.4.4. Query en Elasticsearch 64
3.4.5. Listado de tareas pendientes de ejecutar para el usuario 65
3.4.6. Listado de tareas ya ejecutadas del usuario 65
3.4.7. Conocer los repositorios que posee un usuario determinado 66
3.4.8. Actualización del script de ejecución 67
3.5. Iteración 5. Interfaz con Bootstrap, estadísticas y Kibana 69
3.5.1. Objetivos 69
3.5.2. Interfaz web mediante Bootstrap 69
3.5.2.1. Instalación y configuración 69
3.5.2.2. Elección de plantilla predefinida 70
3.5.2.3. Definición de plantilla base 71
3.5.2.4. Páginas principales de la interfaz 72
3.5.3. Generación de estadísticas y visualización 75
3.5.4. Enlace del sistema con Kibana 76

4. Descripción del producto final **77**
4.1. Descripción de la aplicación: versión con interfaz web 77
4.1.1. Entrada a la aplicación 77
4.1.2. Proceso de autenticación a través de GitHub 78
4.1.3. Pantalla principal de la aplicación 79
4.1.4. Funcionalidad de la aplicación 80
4.1.4.1. Añadir tarea 80
4.1.4.2. Mostrar lista de tareas totales de la aplicación 80
4.1.4.3. Mostrar tareas pendientes de ejecutar del usuario 81
4.1.4.4. Mostrar tareas ejecutadas del usuario 82
4.1.4.5. Mostrar los repositorios de un usuario determinado de GitHub 83
4.1.4.6. Enlace con Kibana 85
4.2. Descripción de la aplicación: versión API 85
4.2.1. Usuarios para la ejecución 86
4.2.2. Añadir tareas al sistema 87
4.2.3. Listado general de tareas del sistema 89
4.2.4. Listado de tareas pendientes de ejecutar para un usuario 89
4.2.5. Listado de tareas ya ejecutadas para un usuario 90
4.2.6. Estadísticas de las tareas ya ejecutadas 91
4.2.7. Lista de repositorios pertenecientes a un usuario GitHub dado 91
4.3. Implementación 93
4.3.1. Arquitectura general 93
4.3.2. Archivos Django: settings.py 95
4.3.3. Archivos Django: models.py 96
4.3.4. Archivos Django: urls.py 98
4.3.5. Archivos Django: signals.py 99
4.3.6. Archivos Django: views.py 99
4.3.7. Archivos Django: search.py 104
4.3.8. Construcción del dashboard: handler.py 105

5. Conclusiones **111**
5.1. Consecución de objetivos 111
5.2. Aplicación de lo aprendido 112
5.3. Lecciones aprendidas 113
5.4. Trabajos futuros 114
5.5. Valoración personal 115

Bibliografía **117**

Índice de figuras

1.1. Ejemplo de dashboard creado con Kibana 18

2.1. Página principal de un proyecto en GitHub. 24
2.2. Estadísticas de los lenguajes de programación en GitHub. 25
2.3. Gráfico de contribuciones a un repositorio GitHub. 25
2.4. Entrada de la API correspondiente a la creación de un proyecto. 26
2.5. Biblioteca estándar de Python. 28
2.6. Ejemplo de consulta al servidor de Elasticsearch. 30
2.7. Ejemplo de dashboard creado con Kibana 32
2.8. Comparación entre JSON y XML. 33
2.9. Interfaz del IDE Eclipse. 37
2.10. Ejemplo de plantilla predefinida por Bootstrap. 40
2.11. Ejemplo de parte de una plantilla CSS. 41
2.12. Texto escrito en LaTeX y su resultado una vez compilado. 43

3.1. Página del módulo de administración de Django. 48
3.2. Página que muestra las instancias creadas del modelo. 48
3.3. Página que muestra el contenido de una instancia del modelo. 49
3.4. Clase DocType generada para nuestro proyecto. 49
3.5. Código Python del método *.indexing()*. 50
3.6. Pantalla inicial para el registro de una aplicación de desarrollador. 51
3.7. Pantalla de toma de datos para el registro de una aplicación en GitHub. 52
3.8. Página inicial de la API de GitHub. 53
3.9. Página de la API de GitHub. 53
3.10. Proceso de autenticación con *Oauth Web Flow*. 54
3.11. Página de la documentación de *Python Social Auth*. 55
3.12. Fragmento de *settings.py* con las aplicaciones instaladas. 56
3.13. Fragmento de *settings.py* con el apartado MIDDLEWARE. 56
3.14. Fragmento de *settings.py* con el apartado TEMPLATES. 56
3.15. Fragmento de *settings.py* con el apartado de BACKENDS. 57
3.16. Fragmento de *settings.py* con los campos. 57
3.17. Fragmento de *urls.py* con la librería. 57
3.18. Introducción de la *Authorization callback URL*. 57
3.19. Valores en *settings.py*. 58
3.20. Pantalla inicial de la interfaz de prueba. 58
3.21. Pantalla de registro en GitHub. 58
3.22. Pantalla de autorización en GitHub. 59

3.23. Redirección a la pantalla principal de la aplicación. 59
3.24. Fragmento Python de petición HTTP con Requests. 60
3.25. Modelo Django *tareas* actualizado. 62
3.26. Actualizaciones complementarias al modelo *tareas*. 62
3.27. Actualizaciones complementarias al modelo *usuarioapp*. 63
3.28. Fragmento de código para almacenar el token de usuario en Elasticsearch. . . . 64
3.29. Página de la documentación relativa a las Query. 64
3.30. Fragmento de código con el proceso de creación de una Query. 65
3.31. Fragmento de código para la obtención de las tareas pendientes. 65
3.32. Fragmento de código para la obtención de las tareas ejecutadas. 65
3.33. Fragmentos de la API de GitHub para obtener los respositorios de un usuario. . 66
3.34. Fragmento de código para la obtención de los repositorios de un usuario. 67
3.35. Código Python del script de ejecución. 68
3.36. Página de descarga de Bootstrap. 69
3.37. Sección de plantillas predefinidas en la página de Bootstrap. 70
3.38. Fragmento de la hoja CSS correspondiente a nuestra interfaz. 71
3.39. Código HTML para insertar la barra superior en la plantilla base. 72
3.40. Barra de navegación perteneciente a la plantilla base. 72
3.41. Código HTML para crear la primera fila, que se divide en tres columnas. 73
3.42. Panel que incluye la funcionalidad de la aplicación. 73
3.43. Código HTML para insertar un formulario en la plantilla base. 74
3.44. Código HTML para insertar una tabla con los datos de Elasticsearch. 74
3.45. Método para generar los datos que se mostrarán en las estadísticas pendientes. . 75
3.46. Método para generar los datos que se mostrarán en las estadísticas ejecutadas. . 76
3.47. Código HTML correspondiente a los botones situados en la barra de navegación. 76
3.48. Código HTML correspondiente a los botones situados en el panel principal. . . 76

4.1. Pantalla de inicio de la interfaz web. 77
4.2. Zona de información de la pantalla de inicio de la interfaz web. 78
4.3. Página de registro de usuario en GitHub. 78
4.4. Página de autorización de la aplicación en GitHub. 79
4.5. Pantalla principal de la interfaz web de la aplicación. 79
4.6. Pantalla para añadir una tarea al sistema. 80
4.7. Pantalla donde se muestra el listado general de tareas de la aplicación. 81
4.8. Pantalla con el listado de tareas pendientes de ejecutar del usuario. 81
4.9. Pantalla con las estadísticas de una de las tareas pendientes de ejecutar. 82
4.10. Pantalla con el listado de tareas ejecutadas del usuario. 83
4.11. Pantalla con las estadísticas de una de las tareas ejecutadas. 83
4.12. Pantalla para introducir el usuario del que se quieren conocer los repositorios. . 84
4.13. Listado de repositorios pertenecientes al usuario dado. 84
4.14. Ejemplo de dashboard creado por el sistema para la información de Git. 85
4.15. Ejemplo de dashboard creado por el sistema para la información de GitHub. . . 85
4.16. Contenido del índice de Elasticsearch de usuarios autenticados. 86
4.17. Dos pares de usuario-token almacenados en variables independientes. 87
4.18. Respuesta de la API a la verificación del token de usuario. 87
4.19. Contenido del índice Elasticsearch tras la ejecución del script. 88

4.20. Respuesta de la API con el listado general 89
4.21. Respuesta de la API a las tareas pendientes de ejecutar para cada usuario. . . . 90
4.22. Respuesta de la API a las tareas ya ejecutadas para cada usuario. 90
4.23. Respuesta de la API a la consulta sobre las estadísticas de una tarea ejecutada. . 91
4.24. Respuesta de la API GitHub a la consulta sobre un usuario dado. 92
4.25. Respuesta de la API a la consulta sobre un usuario determinado. 92
4.26. Diagrama que representa la arquitectura de la aplicación. 93
4.27. Diagrama que representa el proceso de añadir una tarea así como las tecnologías que intervienen en él. 94
4.28. Contenido del fichero settings.py de la aplicación. 96
4.29. Código del modelo Django *tareas* 97
4.30. Código del modelo Django *usuariosapp* 98
4.31. Archivos urls.py de nuestro proyecto. 98
4.32. Código del fichero signals.py. 99
4.33. Líneas de importación en views.py 100
4.34. Código correspondiente al método para mostrar la pantalla de inicio. 100
4.35. Código correspondiente al método para mostrar la pantalla principal. 102
4.36. Código correspondiente al método para añadir una tarea al sistema. 102
4.37. Código correspondiente al método para ver el listado general de tareas del sistema. 103
4.38. Código correspondiente al método para ver el listado de repositorios de un usuario dado. 103
4.39. Código correspondiente a los métodos para ver el listado de tareas pendientes/ejecutadas de un usuario. 104
4.40. Código correspondiente al archivo search.py. 105
4.41. Código correspondiente a la parte de comprobación de la existencia de un fichero. 106
4.42. Código para hacer la consulta de tareas pendientes de ejecutar. 106
4.43. Código para la obtención de información del creador. 106
4.44. Código para la actualización de campos. 107
4.45. Código con la ejecución de los scripts. 108
4.46. Resultado de la ejecución del script con el dashboard de GitHub. 108
4.47. Código correspondiente al archivo handler.py. 109

Capítulo 1

Introducción

El desarrollo de software es una actividad en la que trabajan diariamente millones de personas en el mundo. La necesidad de desarrollar:

- Aplicaciones web
- Aplicaciones para smartphones
- Big data

y otras múltiples necesidades hacen que en un contexto de código abierto, el desarrollo colaborativo sea imprescindible. Para dar soporte a la colaboración entre desarrolladores existen varias plataformas, como por ejemplo GitHub, que da alojamiento a proyectos de desarrolladores, así como la posibilidad de que el resto de usuarios, si el creador lo cree conveniente, vean y propongan mejoras o correcciones de errores. Toda esta interacción entre usuarios queda registrada por GitHub, que ofrece estadísticas sobre las veces que alguien ha propuesto una modificación o el creador ha subido código nuevo.

Cómo toda estadística, necesita un sistema que se encargue de mostrar los datos relacionados con un proyecto, un usuario o un grupo de usuarios. El sistema de representación de datos crea una visualización para mostrarlos.

Este proyecto trata de facilitar a un posible usuario las acciones previas a la creación de dicha visualización, es decir, una interfaz para que el usuario introduzca el repositorio/proyecto del que se obtendrán los datos que contenga la visualización, para una vez creado informarle de cual es su ubicación para que pueda acceder al él.

Entre la petición de datos al usuario y la creación de la visualización, deben ponerse varias tecnologías en juego, Python, HTML, Elasticsearch, Kibana o Django.

1.1. Objetivo general

El objetivo general de este proyecto es la creación de una aplicación que permita a un usuario obtener una visualización de los datos de repositorios alojados en GitHub, para ello debe autenticar al usuario a través de GitHub, obtener los datos del repositorio a partir del que crear la visualización, crearlo y finalmente indicarle donde puede ver el resultado.

Lo que hasta ahora hemos llamado visualización de datos, se conoce como dashboard, y será la forma en la que lo denominaremos a partir de ahora. Un dashboard, en el sentido general,

es una interfaz/panel en la cual se muestran enlaces a los elementos principales de un sistema. Sin embargo, basándose en esta definición y según en el contexto en el que se encuentre la palabra, tiene definiciones aproximadas pero adaptadas a dicho contexto. Para nuestro proyecto, la adaptación de la definición de dashboard será, la de un panel en el que se representarán los datos que ofrecen las estadísticas de un proyecto/repositorio, para dicha representación se utilizarán múltiples tipos de representaciones gráficas, como gráficos de barras, líneas o puntos, además de los elementos necesarios para poder interactuar con dichos gráficos. Por lo tanto el dashboard se encargará de agrupar y conciliar este conjunto de gráficos y mostrarlos de forma agradable para el usuario.

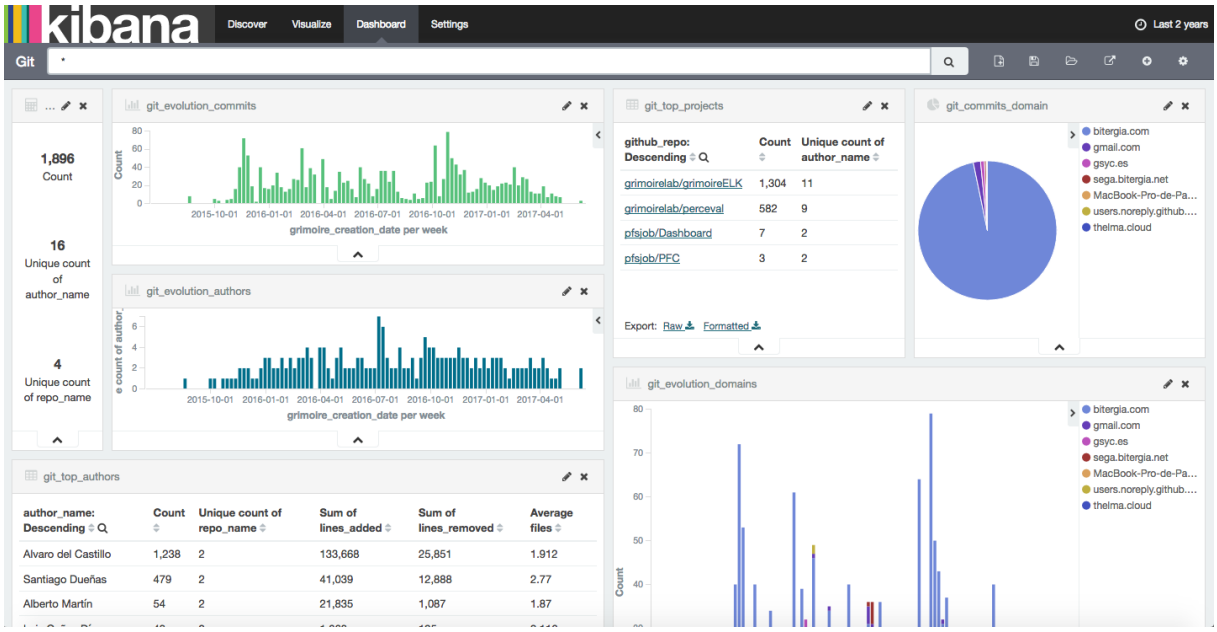


Figura 1.1: Ejemplo de dashboard creado con Kibana

Hasta ahora tenemos el punto de partida, el repositorio/proyecto del que queremos obtener la información, y el elemento final, el dashboard donde se representarán los datos finalmente. El proceso para llegar de un punto al otro será el que tendrá que llevar a cabo nuestro proyecto.

El objetivo general será comunicar la distintas tecnologías que se pondrán en juego para completar el proceso. Dicho proceso tendrá las siguientes partes.

- El punto de partida será el uso del módulo de Python Perceval, a partir de cual obtendremos los datos que posteriormente se plasmarán en el dashboard.
- Para poder empezar con la obtención de datos, hay que solicitarle al usuario de la aplicación los datos del proyecto/repositorio, para este proceso se creará una aplicación Django.
- Toda la información solicitada al usuario, así como información adicional, deberá ser almacenada, para su posterior ejecución. Por lo tanto la aplicación Django anteriormente mencionada tener soporte para almacenamiento de datos.

- En paralelo con la aplicación Django, debe existir un manejador que se encargue de ejecutar las tareas solicitadas por el usuario.

Como hemos comentado al inicio del apartado, la aplicación tomará los datos a visualizar de GitHub, qué es una plataforma para el desarrollo colaborativo de software, tomando el repositorio/proyecto que el usuario solicite. Como ya hemos indicado el proceso de obtención de datos lo llevará a cabo Perceval, pero GitHub ofrece una limitación para la descarga a quien no es usuario registrado de la plataforma. Por lo tanto, si queremos que la aplicación no quede inservible el tiempo en el que el ancho de banda esté agotado, necesitamos que el usuario que solicita la información esté registrado en GitHub, para de esta forma tener ancho de banda ilimitado.

Para que nuestra aplicación pueda interaccionar con GitHub necesitaremos la intervención de la API de desarrolladores que ofrece la plataforma.

La API de GitHub ofrece la funcionalidad necesaria para interaccionar con GitHub desde una aplicación de un tercero, o llevar a cabo las distintas operaciones que ofrece sin la necesidad de usar un navegador. El funcionamiento está basado en peticiones HTTP(GET, PUT, POST,...) acompañados de una serie de parámetros que llevarán la información de la tarea a realizar, así como las credenciales necesarias.

Para nuestro caso, necesitaremos por un lado verificar que el usuario está registrado en la plataforma, y posteriormente obtener un identificador del mismo para evitar la limitación del ancho de banda.

Finalmente todos los recursos del proyectos, tales como:

- Código fuente.
- Memoria.
- Resto de documentos o archivos necesarios.

quedarán alojados en la siguiente URL: Proyecto fin de carrera Pablo Fernández Salguero - https://pfsjob.github.io/.

1.2. Objetivos específicos

Una vez descrito el objetivo de la aplicación pasamos a ver cuales son los objetivos específicos de cada parte relevante del flujo de la aplicación.

- **Integración con el sistema de almacenamiento:** La aplicación usará Kibana para la visualización del dashboard, y Perceval para la creación del mismo, por lo que, usando ya información en formato JSON, lo ideal seria usar Elasticsearch. Por lo tanto es necesario integrar Elasticsearch con el lenguaje de programación que usemos, en este caso será Python con Django.
- **Integración con GitHub:** Una vez que somos capaces de almacenar información en Elasticsearch a través de Django, hay que añadir la funcionalidad necesaria para que el usuario sea autenticado a través de GitHub y una vez autenticado obtener el token de usuario para operar con él en la creación del dashboard.

- **Funcionalidad útil para el usuario:** La aplicación no debe limitarse únicamente a la creación del dashboard, además de esto hay que añadir funcionalidad relativa al proceso de ejecución, cómo que creaciones tiene pendiente un usuario, o cuales tiene ya ejecutadas.

- **Creación de una API:** El primer soporte que ofrecerá la aplicación será el de una API con la funcionalidad anteriormente comentada, para su posterior integración en otros proyectos.

- **Interfaz de usuario:** Además de la creación de una API, el proyecto debe ofrecer una interfaz web para que el usuario pueda fácilmente y sin un aprendizaje previo, crear un dashboard con la información del repositorio GitHub que considere oportuno.

1.3. Requisitos

Los requisitos que debe cumplir el proyecto serán:

- **Interfaz:** La aplicación deberá tener una interfaz que sea rápida, intuitiva y agradable para el usuario, de tal forma que en unos pocos pasos sea capaz de ofrecer el resultado esperado por el usuario, un dashboard con la representación gráfica de los datos del repositorio que ha solicitado.

- **Transparencia:** Todos los procesos que transcurren desde que el usuario llega a la página principal de la aplicación hasta que obtiene el dashboard final, deben ser transparentes al usuario, es decir, deben estar ocultos. Cuando decimos que deben estar ocultos, no sólo nos referimos a que el usuario visualmente no debe apreciarlos, sino que la experiencia de éste sea altamente fluida, para evitar que en hipotéticas transiciones entre las partes que componen la aplicación el usuario pueda pensar que se están realizando tareas sin darle opción a interaccionar durante la ejecución de las mismas.

- **Integración con GitHub:** Al ser una aplicación de análisis de repositorios alojados en GitHub, es conveniente, para dar seguridad y confianza al usuario, que sea dicha plataforma quien autentique a éste, obteniendo de esta forma el soporte para la obtención de los datos.

- **Velocidad en la obtención de resultados:** El proceso debe ser lo suficientemente rápido para que el usuario considere que es práctico su uso.

1.4. Planificación temporal

El contexto personal en el que se desarrolla el trabajo, es el de una persona con un trabajo a tiempo completo, que nada tiene que ver con las telecomunicaciones o la informática, pero que ocupa gran parte de la jornada diaria.

Debido a todo esto el proyecto se va realizando en cortos periodos de tiempo, sobre todo por la tarde-noche, lo que hace que el periodo de tiempo total sea muy extenso, aunque el tiempo efectivo dedicado es totalmente diferente.

Para obtener el producto final, se van ejecutando por separado las diferentes partes de las que se compone, integrando cada nuevo desarrollo en el anterior, es decir, el proyecto se compone de varias etapas intermedias. Dichas etapas se planifican mediante una reunión con el tutor del proyecto, en la cual se determinan las tareas, los objetivos y los plazos de la misma.

El tiempo que hay entre una reunión y otra es bastante variable en función de la disponibilidad del alumno, llevándose a cabo principalmente a través de videoconferencia.

1.5. Estructura de la memoria

Para la documentación del proyecto se redacta esta memoria, que tendrá la siguiente estructura:

- **Introducción:** En este apartado se informará al lector acerca del contexto en el que está enmarcado el proyecto, para que de esta forma pueda entender su función. Además se contará cual es el objetivo general, así como los objetivos específicos. También se indicarán los requisitos que debe cumplir el producto final. Por último se mostrará un esquema de la memoria, explicando brevemente el contenido de cada parte.

- **Estado del arte y tecnologías:** El apartado de estado del arte y tecnología tendrá, para cada elemento y tecnología usada durante el proyecto, una descripción con:

  - Aspectos generales.
  - Historia.
  - Características.

- **Desarrollo:** Para el proceso de desarrollo del proyecto se utilizará una metodología basada en iteraciones, es decir, etapas en las que se entregarán productos parciales, relacionados entre sí, por lo que en este apartado se documentaran los procesos de dichas etapas.

- **Descripción del producto final:** El apartado de descripción del producto final contendrá el análisis del resultado del proceso de desarrollo. La documentación de dicho análisis se realizará desde el punto de vista del uso de la aplicación y desde el punto de vista de la implementación del mismo, es decir, de la arquitectura.

- **Conclusiones:** Las conclusiones contendrán un resumen de los conocimientos aprendidos, los conocimientos aplicados y los siguientes pasos en los que podría seguir desarrollándose el proyecto.

Capítulo 2

Estado del arte y tecnologías

2.1. GitHub

GitHub es una plataforma de desarrollo colaborativo de software que permite al usuario, previamente registrado, almacenar proyectos. Está basado en el sistema de control de versiones Git.

GitHub no sólo permite almacenar proyectos, ya que al ser colaborativo, permite que el resto de usuarios pueda interactuar con los proyectos del resto.

La plataforma GitHub es ideal para nuestros requisitos, principalmente porque la herramienta Perceval que sirve como punto de partida es capaz de obtener los datos de GitHub, y aunque Perceval puede interactuar con otras plataformas como Gerrit o StackOverflow, GitHub cuenta con un uso más extendido, además de una API para dar soporte a desarrolladores.

2.1.1. Historia

GitHub nace en el año 2008, en una oficinas del valle de San Francisco, de la mano de Tom Preston-Werner (creador de Gravatar) y otros dos desarrolladores.

Pero para entender el nacimiento de GitHub es necesario conocer el origen del sistema en el que está basado, Git. Git es el sistema de control de versiones que crea Linus Torvalds para poder controlar su enorme proyecto de código abierto, Linux, es el año 2005.

2.1.2. Características

Por lo tanto GitHub ofrece al usuario:

- La posibilidad de almacenar el código fuente perteneciente al usuario, organizado de forma similar a un sistema de archivos común. Para almacenar estos archivos el usuario debe crear un proyecto, que puede declarar como público o como privado, es decir, visible para el resto de usuarios o no.
- Además de esto, el usuario podrá acceder y consultar los proyectos, siempre que sean públicos, del resto de usuarios, ya que además de poder ver el código fuente, ofrece la posibilidad de tener una página de explicación sobre el mismo.
- Para que pueda ser colaborativo, el usuario además de poder consultar el repositorio, tiene la opción de hacer *fork*, que consiste en crear una copia en su cuenta (de forma local o en

la versión online), para de esta forma hacer las modificaciones que crea necesarias, con el objetivo de mejorar el proyecto original.

- Una vez realizadas las modificaciones, se le puede informar al propietario del proyecto de éstas. Para ello se realiza un *pull_request*, que permitirá al propietario supervisar y, en el caso de considerar oportuna la modificación, añadirla al proyecto original.

- Existe también la opción de los *Issues*, que dan soporte para informar al propietario del proyecto de los posibles errores que contenga el mismo.

La imagen de debajo corresponde con la página principal de un proyecto en GitHub, en ella pueden verse todas las opciones comentadas anteriormente. En la zona que corresponde a README.md, será la página explicativa sobre el proyecto, que estará en el archivo con el mismo nombre.

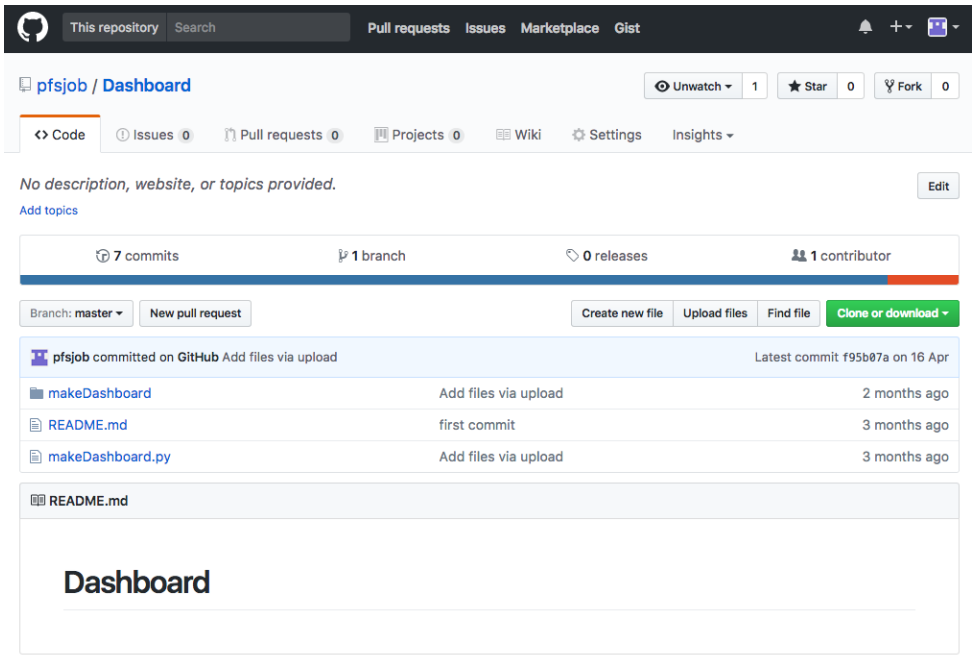


Figura 2.1: Página principal de un proyecto en GitHub.

Una de las partes más interesantes que ofrece GitHub, son los valores estadísticos de las acciones anteriormente comentadas. Éstas ofrecen la posibilidad de realizar un seguimiento de las mismas, para ver la actividad de los desarrolladores con los proyectos alojados en la plataforma.

A continuación vemos dos ejemplos de los datos estadísticos de GitHub que almacena un proyecto.

En primer lugar vemos una representación gráfica del uso de lenguajes de programación en GitHub, con el año de creación del primer repositorio o los repositorios activos de un lenguaje de programación determinado. En nuestro caso hemos optado por mostrar los datos de Python.

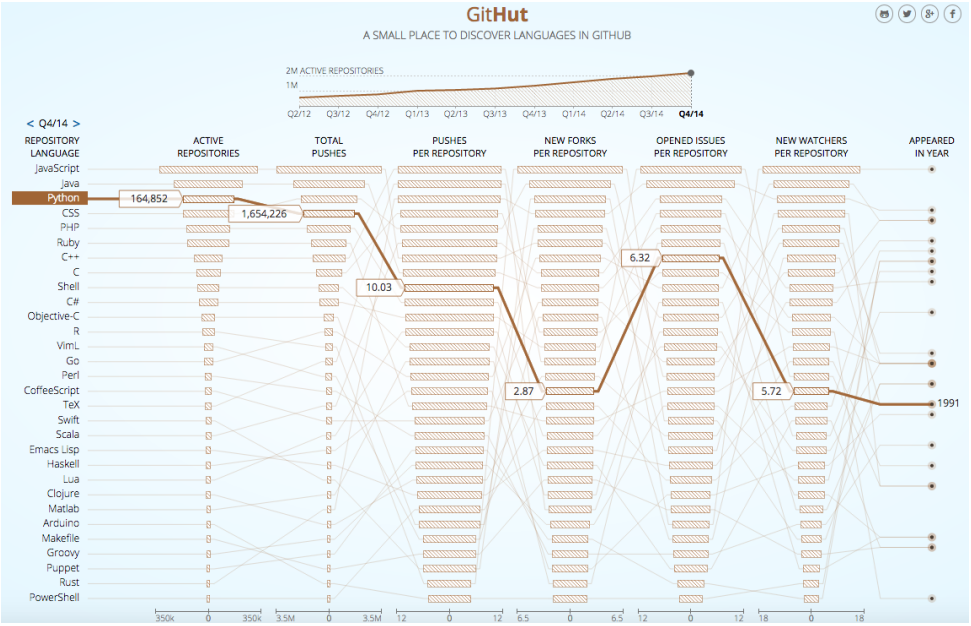


Figura 2.2: Estadísticas de los lenguajes de programación en GitHub.

En segundo lugar vemos un gráfico con la contribuciones que se han realizado sobre un repositorio determinado.

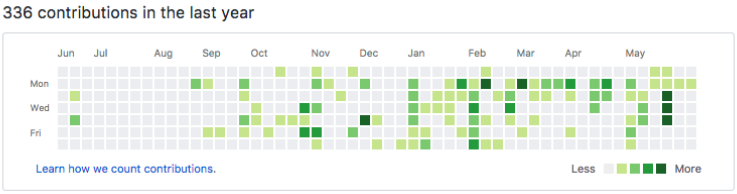


Figura 2.3: Gráfico de contribuciones a un repositorio GitHub.

2.1.3. API de GitHub

GitHub proporciona una API[1] para desarrolladores, con una extensa funcionalidad que permite a éstos trabajar con GitHub sin acceder a su interfaz web, desde crear un repositorio, hacer un *pull_request* o herramientas para buscar un repositorio dentro de la plataforma.

Esta API, también ofrece la posibilidad de autenticar a un usuario a través del protocolo Oauth u otros métodos de autenticación.

La API funciona con el protocolo HTTP, es decir, mediante peticiones GET, PUT, POST o PATCH. La principal ventaja de usar el protocolo HTTP es la fácil integración con las lenguajes de programación habituales, que generalmente incluyen soporte para el manejo de forma simple de HTTP. Para usar la API necesitaremos conocer la url correspondiente al método que queramos ejecutar, así como los parámetros obligatorios que debe contener el cuerpo de la petición HTTP, además de éstos generalmente las peticiones admiten algunos parámetros opcionales que aumentan la versatilidad de la ejecución.

Todas las peticiones a la API de GitHub tienen una parte de la url en común, y una parte en la que varia el texto según lo que queramos hacer. Una petición tendrá la siguiente estructura: *https://api.github.com + url del método*.

Por lo tanto, si queremos crear un nuevo repositorio mediante la API de GitHub, buscaremos su entrada en la API.

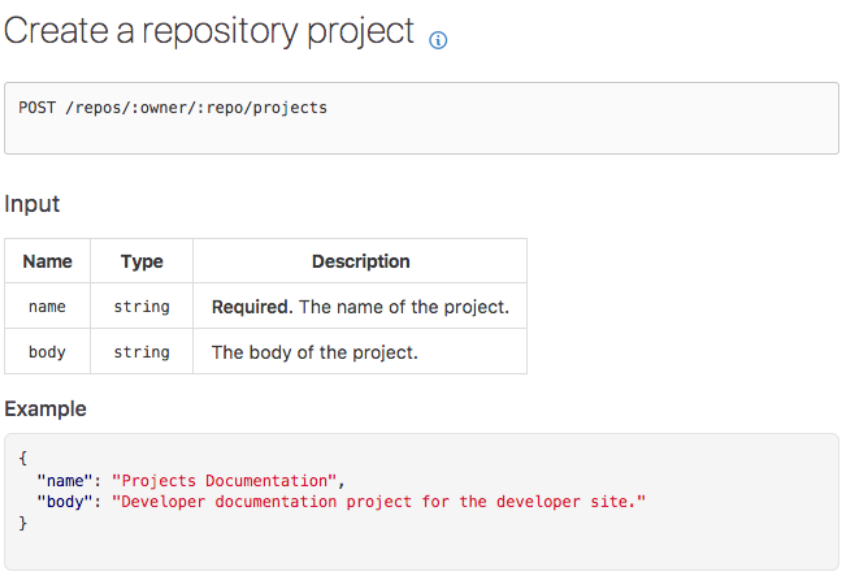


Figura 2.4: Entrada de la API correspondiente a la creación de un proyecto.

Como podemos ver en la imagen, la url de este método es */repos/:owner/:repo/projects*, que en la página de información nos indica justo delante de la URL el tipo de petición con el que funciona, en este caso POST. En cuanto a los parámetros que admite, es imprescindible incorporar un nombre y opcional un cuerpo con información acerca del mismo.

Por último hay que señalar que la salida de los distintos métodos de la API tienen un formato JSON, que hacen fácil la integración con los lenguajes de programación habituales. Tanto la petición, vía HTTP, como la salida en formato JSON, hacen de la API de GitHub altamente integrable en casi todos los desarrollos existentes.

2.2. Python

Python[2] es un lenguaje de programación con una sintaxis que favorezca un código legible. Se trata de un lenguaje multiparadigma, es decir, admite varios estilos de programación:

- Programación orientada a objetos.
- Programación imperativa.
- Programación funcional (en menor medida).

Las características principales de Python son:

- Tiene tipado dinámico.

- Es un lenguaje interpretado.

- En un lenguaje multiplataforma.

La elección de Python para el uso en este proyecto está fuertemente relacionada con la necesidad de implementar una aplicación web con Django, que está escrito en Python. Existen otros frameworks para la creación de aplicaciones web, basados en diferentes lenguajes de programación, pero tanto Django como Python son conocidos por el alumno, lo que facilita su manejo y elimina el periodo previo de aprendizaje.

Además de lo anteriormente señalado, Python ofrece soporte suficiente para las tecnologías que, a priori, se consideran necesarias para llevar a cabo el proyecto, como JSON o Elasticsearch.

2.2.1. Historia

Python tiene su origen entre los últimos años de la década de los 80 y los primeros años de la década de los 90, cuando Guido Van Rossum (trabajador del CWI, centro de investigación holandés de carácter oficial) decide, en su tiempo libre, crear el proyecto para dar continuidad al lenguaje de programación ABC. La primera publicación de Python es del mes de febrero del año 1991.

Python recibe este nombre por la afición de Guido Van Rossum al grupo musical los Monty Python.

2.2.2. Características

Python tiene licencia de código abierto, denominada *Python Software Foundation License*. Lo que lo convierte en un lenguaje que permite distribuir aplicaciones basadas en él sin tener que pagar ninguna licencia. Incluso al ser de código abierto, cualquiera puede contribuir a su desarrollo.

La posibilidad que ofrece Python de desarrollar software para distintos ámbitos (científico, red, GUI o aplicaciones web) lo convierte en uno de los lenguajes de programación de mayor uso en la actualidad (4º en Junio de 2017 según el índice TIOBE).

Uno de los puntos fuertes de Python se basa en el uso de las bibliotecas, que pueden ser de dos orígenes:

- **Biblioteca estándar de Python:** la biblioteca estándar de Python[3] pertenece al propio lenguaje, y destaca por ser extensa, es decir, de ofrecer funcionalidad para el manejo de fechas, ficheros JSON, concurrencia o http.

Figura 2.5: Biblioteca estándar de Python.

- **Bibliotecas externas:** se trata de bibliotecas desarrolladas por terceros para el manejo de funcionalidad no recogida en la biblioteca estándar, como por ejemplo la librería *requests*, que mejora el manejo del protocolo HTTP.

2.3. Django

Django es un framework de alto nivel para la creación de aplicaciones web escrito en Python, que respeta el patrón de diseño conocido como *modelo–vista–controlador*.

El modelo MVC (*modelo–vista–controlador*) es un patrón de arquitectura de software, que separa los siguientes aspectos:

- Los datos y la lógica de negocio.
- La interfaz de usuario.
- Módulo encargado de gestionar los eventos y las comunicaciones.

El modelo MVC define tres componentes, el Modelo, el Controlador y la Vista, pero Django en la aplicación del modelo tiene inconsistencias en la parte de la nomenclatura. A continuación vemos una descripción general de los componentes de MVC y su adaptación Django.

- **Modelo:** el Modelo es la parte del patrón que se encarga de gestionar el acceso a la información con la que el sistema trabaja. En este apartado Django encaja perfectamente con el patrón.
- **Controlador:** se encarga de gestionar los eventos, pudiendo hacer peticiones al modelo o enviar comandos a la vista asociada. En este apartado Django al controlador lo llama Vista dentro de su nomenclatura.
- **Vista:** ssta parte se encarga de presentar la información que gestiona el modelo con un formato que sea adecuado para el usuario. En este apartado Django a la vista la denomina/gestiona con las templates.

Como ya hemos indicado en la sección de Python, la elección de Django está principalmente basada en su relación con este lenguaje y el conocimiento previo del framework por parte del alumno. Además la relativa facilidad para desplegar la aplicación web y su capacidad de integración con el resto de tecnologías del proyecto lo hacen ideal para el mismo.

2.3.1. Historia

El origen de Django se remonta al World Company de Lawrence, en Kansas, sitio de noticias, que desarrolló Django para gestionar de forma ágil y rápida la subida y modificacion de éstas al servidor.

2.3.2. Características

La característica más importante de Django es la facilidad para crear aplicaciones web, y sobre todo, la facilidad que ofrece para modificar las mismas una vez creadas.

El resto de características son:

- API para bases de datos muy robusta.
- Sistema de vistas genéricas, que ofrecen una serie de funcionalidad ya definida para evitar escribir el código de tareas comunes.
- Sistema extensible de plantillas, con herencia de plantillas, para que la representación de la aplicación se homogénea, es decir, se mantenga un patrón común entre todas sus páginas.
- Un manejador de URLs, basado en expresiones regulares.
- Soporte de administración para la aplicación web, con un módulo de administrador que permite manejar los datos almacenados en los modelos.
- Soporte para la gestión de usuarios, en el módulo de administración, que permite la gestión de los usuarios que se conectan.

Todas estas características están recogidas en la documentación[4].

2.4. Elasticsearch

Elasticsearch es un servidor de búsqueda basado en Lucene, es decir, es una base de datos distribuida enfocada a búsquedas en documentos estructurados. Al ser distribuida la hace altamente escalable y tolerante a fallos. Elasticsearch está desarrollado en Java, publicado como código abierto bajo las condiciones de la licencia Apache.

La elección de Elasticsearch para nuestro proyecto se debe principalmente a dos factores, en primer lugar el uso de JSON como sistema de escritura de los datos y en segundo lugar la potencia para la búsqueda de estos datos una vez indexados.

2.4.1. Uso de Elasticsearch

Elasticsearch es un sistema distribuido, es decir, se encuentra ubicado en una dirección IP determinada y escuchando peticiones en un puerto determinado. Las consultas se realizan mediante el protocolo HTTP, ya sea con una interfaz sobre el navegador o usando por ejemplo el comando *curl* desde el terminal de usuario. Las consultas que se realizan al servidor de Elasticsearch pueden devolver la estructura de una instancia o el contenido de la misma.

Figura 2.6: Ejemplo de consulta al servidor de Elasticsearch.

En la imagen se muestra una consulta desde el terminal con el comando *curl*, en este caso, viendo la estructura de la instancia tareas, que nos devuelve los parámetros que contiene así como el tipo de dato que se almacena en cada uno. Como puede verse en la línea del comando *curl* se realiza la consulta al *localhost* en el puerto 9200 que es donde por defecto arranca el servidor de Elasticsearch, con el nombre de la instancia *tareas* y por último la instrucción *pretty*, que devuelve la salida formateada, con una apariencia tabulada.

2.4.2. Historia

En el año 2004 Shay Banon ante la dificultad de trabajar directamente con Lucene, comenzó a trabajar en una capa de abstracción en JAVA para facilitar las búsquedas, lo llamó Compass y

estaba basado en código abierto. En posteriores versiones de Compass, más concretamente en la tercera, vio la necesidad de crear una solución de búsqueda escalable, lo que hacia necesario reescribir gran parte del código de Compass, nació entonces Elasticsearch, cuya primera versión es del año 2010.

2.4.3. Características

Las principales características de Elasticsearch son:

- Aporta acceso en tiempo real a los datos que se están modificando.
- Escalabilidad, es decir, posibilidad de escalar de forma horizontal.
- Gran porcentaje de disponibilidad gracias a la capacidad para detectar nodos con fallos.
- Posibilidad de actuar sobre varios índices al mismo tiempo.
- Permite trabajar sin una estructura fija de bases de datos.
- La información se almacena en formato JSON.
- Tiene a disposición del desarrollados una potente API, tanto para JSON como para múltiples lenguajes.
- Búsquedas basadas en texto, soportando geolocalización o autocompletado.
- Previene de la pérdida de datos si se modifican registros simultáneamente.

Debido al uso de JSON como sistema de almacenamiento de información, hace que Elasticsearch sea compatible con un gran número de lenguajes de programación.

Para elaborar la sección de Elasticsearch, principalmente para conocer sus características, se ha utilizado el libro *Elasticsearch: The Definitive Guide: A Distributed Real-Time Search and Analytics Engine*[5].

2.5. Kibana

Kibana es una herramienta de código libre para visualización de datos de forma sencilla y entendible para personas sin conocimiento técnico.

La elección de Kibana para la representación de los datos se debe a que la base del proyecto es Perceval. Como ya hemos indicado anteriormente Perceval se encarga de obtener datos de repositorios de GitHub, almacenándolo posteriormente en Elasticsearch,y ofreciendo soporte para crear la visualización de datos con Kibana, ya que Elasticsearch y Kibana funcionan de manera estrechamente relacionada.

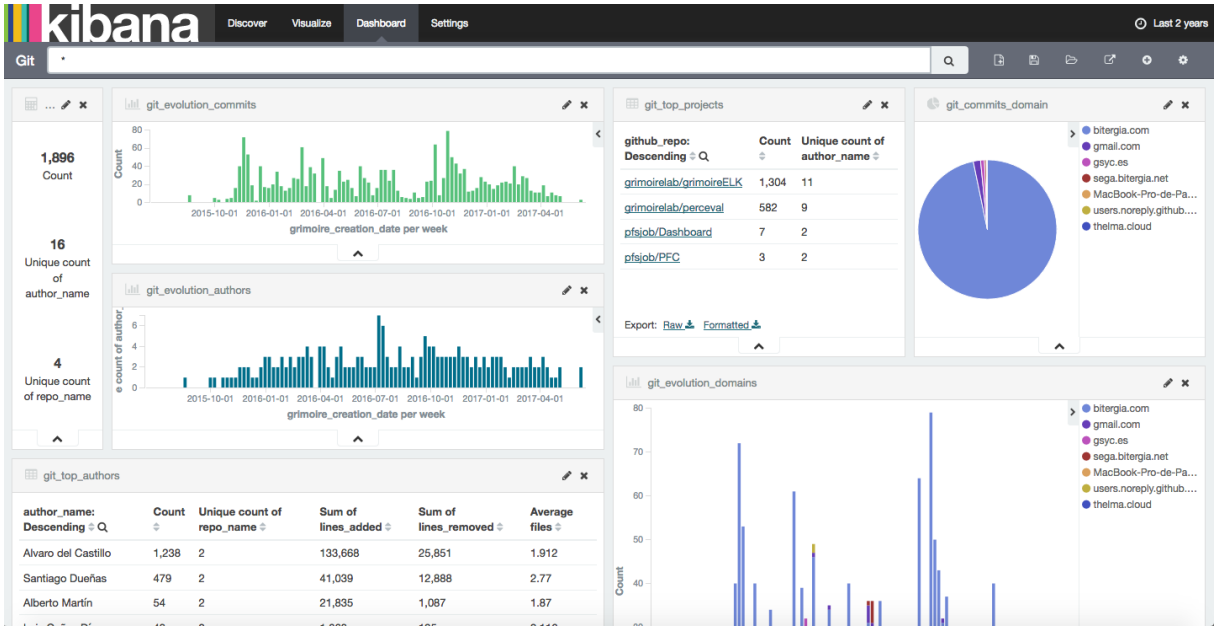


Figura 2.7: Ejemplo de dashboard creado con Kibana

2.5.1. Historia

Kibana nace por la necesidad de visualizar los datos almacenados en Elasticsearch, de hecho, son los desarrolladores de esta herramienta los encargados de crear Kibana, es decir, Kibana pertenece a Elasticsearch.

2.5.2. Características

Según la documentación de Kibana[6], éste ofrece al usuario la posibilidad de manejar grandes volúmenes de datos visualizándolos en diferentes tipos de gráficos:

- Barras
- Líneas
- Dispersiones
- Circulares
- Mapas

2.6. JSON

JSON es el acrónimo de *JavaScript Object Notation*, es un formato de texto ligero para intercambio de datos. JSON describe los datos con una sintaxis dedicada, que se usa para identificar y gestionar los datos.

Teniendo en cuenta las distintas tecnologías que intervienen en el proyecto el uso de JSON se hace casi necesario, ya que como hemos visto tanto Elasticsearch como Kibana utilizan datos

en JSON, o la API de GitHub devuelve las consultas que se le hacen en JSON, a esto hay que sumarle que es fácilmente integrable con el resto de tecnologías del proyecto. Otra razón es el potente soporte que ofrece Python para manejar datos en JSON, almacenada en la librería estándar de Python[7].

2.6.1. Historia

JSON nace para solucionar el problema de agilidad y rapidez de XML con grandes volúmenes de datos de la mano de Douglas Crockford y sus colaboradores a principios de los años 2000. Douglas Crockford durante el tiempo que pasó en State Software popularizó JSON, consiguiendo el dominio *json.org* en el año 2002.

2.6.2. Características

Las principales características que hacen de JSON un lenguaje muy usado son:

- Javascript está entre los lenguajes más usados (el séptimo según el índice TIOBE en Junio de 2017), por lo que la facilidad para usar JSON en este lenguaje, le proporciona muchos usuarios.
- Técnicamente escribir un analizador sintáctico (parser) para JSON, es muy sencillo, por lo que los lenguajes se decantan por él antes que por XML.
- JSON puede ser manejado por casi todos los lenguajes de programación.

JSON se utiliza en escenarios en los que el tamaño del intercambio de datos entre el servidor y el cliente es de vital importancia y la fuente de los mismo es de fiar, por eso en la actualidad XML ofrece mayor seguridad, que junto con los procesadores nativos de los navegadores actuales hacen que JSON y XML puedan convivir simultáneamente en la misma aplicación.

2.6.3. Comparación con XML

```
{
  "id": 123,
  "title": "Object Thinking",
  "author": "David West",
  "published": {
    "by": "Microsoft Press",
    "year": 2004
  }
}
```

(a) JSON

```
<?xml version="1.0"?>
<book id="123">
  <title>Object Thinking</title>
  <author>David West</author>
  <published>
    <by>Microsoft Press</by>
    <year>2004</year>
  </published>
</book>
```

(b) XML

Figura 2.8: Comparación entre JSON y XML.

A la izquierda tenemos un documento en formato JSON y a la derecha tenemos uno en formato XML, a simple vista parecen similares en extensión, cada uno con su sintaxis, pero al contar el número de caracteres puede ver que en el JSON hay 140 caracteres y en el de XML

hay 167. La diferencia es sólo de 27 caracteres, pero hay que tener en cuenta la simplicidad del ejemplo, si se escala a grandes volúmenes de datos la diferencia será muy grande, de ahí la ventaja de usar JSON para grandes tamaños de datos.

2.7. Python Social Auth

Python Social Auth es un sistema que facilita la integración del mecanismo de autorización/autenticación para redes sociales, en aplicaciones de terceros.

En nuestro caso, como ya hemos comentado, para darle confianza y seguridad al usuario en el uso de la aplicación, la autenticación debe ser a través de la red social correspondiente, en nuestro caso GitHub, no siendo necesaria la creación de una cuenta de usuario por éste en la aplicación, eliminando por lo tanto la implementación de dicha gestión.

Autenticar al usuario a través de cualquier red social nos ofrece dos ventajas, la primera es la verificación de éste con un mecanismo de fiabilidad demostrada y la segunda es el acceso a toda la funcionalidad de la que dispone un usuario de la red social correspondiente.

Existen otras opciones para poder facilitar la tarea de autorización/autenticación, pero siendo un proyecto que trabaja con GitHub, que a su vez trabaja con Oauth2 para la autenticación y autorización y estando además escrito en Python con Django, la utilización de Python Social Auth es ideal, debido a su extenso manual[8] para instalarlo y configurarlo con los protocolos anteriores.

2.7.1. Historia

El código base de Python Social Auth deriva de django-social-auth, y nace para generalizar el proceso de autenticación/autorización en los frameworks actuales para desarrollo de aplicaciones web, así como ofrecer las herramientas necesarias para los nuevos frameworks.

2.7.2. Características

Las características principales de Python Social Auth son:

- Python Social Auth trabaja con diferentes protocolos de autenticación, es decir, engloba a múltiples protocolos y por lo tanto a muchas redes sociales, como:
  - Oauth, tanto 1 como 2.
  - OpenId.
  - Otros.
- Otro motivo por el que Python Social Auth llega a un gran número de de usuarios, es la integración de múltiples frameworks, como:
  - Django.
  - Flask.
  - CherryPy.
  - Pyramid.

    - Webpy.

- Por último, la última característica que ayuda a la popularidad del sistema, Python Social Auth está preparada para dar soporte a las redes sociales que más se utilizan, como:

    - Google.
    - Twitter.
    - Yahoo.
    - Github.
    - Otros.

2.8. Requests

Requests es una librería para el uso de HTTP escrita en Python, cuyo nombre completo es: *Requests: HTTP for Humans* o *Requests: HTTP para Humanos* en español.

Para nuestro proyecto, la librería Requests facilita en gran medida el proceso de peticiones HTTP (GET, POST o PUT), generalmente en las interacciones con la API de GitHub. Dichas peticiones HTTP, son de fácil uso mediante la documentación[9] de Requests.

2.8.1. Historia

La biblioteca estándar de Python contiene el módulo urllib2, que contiene la funcionalidad necesaria para el manejo de HTTP. Pero está anticuada, surgió para un web diferente y a pesar de contener la funcionalidad completa, su manejo es muy complicado, llegando en algunos casos a tener que reescribir parte de los métodos ya existentes. Es por esto que Kenneth Reitz empieza a desarrollar Requests haciendo que la integración con servicios web sea transparente.

2.8.2. Características

El módulo Requests está preparado para la web de hoy en día, las características son:

- URLs y dominios internacionales.
- Keep-Alive y Agrupamiento de conexiones (Connection Pooling).
- Sesiones con Cookies persistentes.
- Verificación SSL al estilo navegador.
- Autenticación Básica y Digest.
- Elegantes Cookies en pares Llave/Valor.
- Descompresión automática.
- Cuerpos de respuestas Unicode.
- Subida de archivos Multiparte.

- Tiempos de espera de conexión.
- Soporte para .netrc.
- Python 2.6 – 3.6.
- Seguridad para programación en hilos (Thread-safety).

2.9. Eclipse

Eclipse en un entorno de desarrollo integrado (en inglés, IDE), para ser extensible de forma indefinida mediante plug-ins. Está desarrollado en código abierto.

2.9.1. Historia

Eclipse nace como un proyecto de IBM Canadá. Su desarrollador principal fue OTI (Object Technology International) para reemplazar a VisualAge también desarrollado por OTI. En el año 2001 se formó un consorcio para el desarrollo de Eclipse como código abierto, y en el año 2003 se separa finalmente de IBM. La primera versión pública de eclipse sale a la luz en Junio del año 2004, bajo el nombre de Eclipse 3.0. La versión actual (Junio 2016) es Eclipse 4.6 o Neon.

2.9.2. Características

Eclipse no está pensado para el uso de un lenguaje de programación específico, pero es habitualmente usado para Java, ya que el plug-in JDT (Java Development Toolkit) viene por defecto con la distribución base del IDE.

Las principales características de Eclipse son:

- **Gran colección de plug-ins:** hay disponibles una gran cantidad, creados por la fundación de eclipse o por terceros. Prácticamente todos los lenguajes tienen disponible el suyo.
- **Perspectivas, editores y vistas:** en Eclipse el concepto de trabajo se basa en perspectivas, que es una preconfiguración de ventanas y editores, relacionadas entre sí, permitiéndonos trabajar en un determinado entorno de trabajo de forma óptima.
- **Gestión de proyectos:** el desarrollo en Eclipse se basa en proyectos, que son los recursos relacionados entre sí, como el código fuente, la documentación, los ficheros de configuración o el árbol de directorios. El IDE tiene un creador de proyectos que se encarga de abrir la perspectiva adecuada para él.
- **Depurador de código:** el IDE incluye un potente depurador fácil e intuitivo que visualmente ayuda a mejorar el código del usuario. Por supuesto se incluye una perspectiva de depuración, donde se muestra de forma ordenada toda la información para dicha tarea.

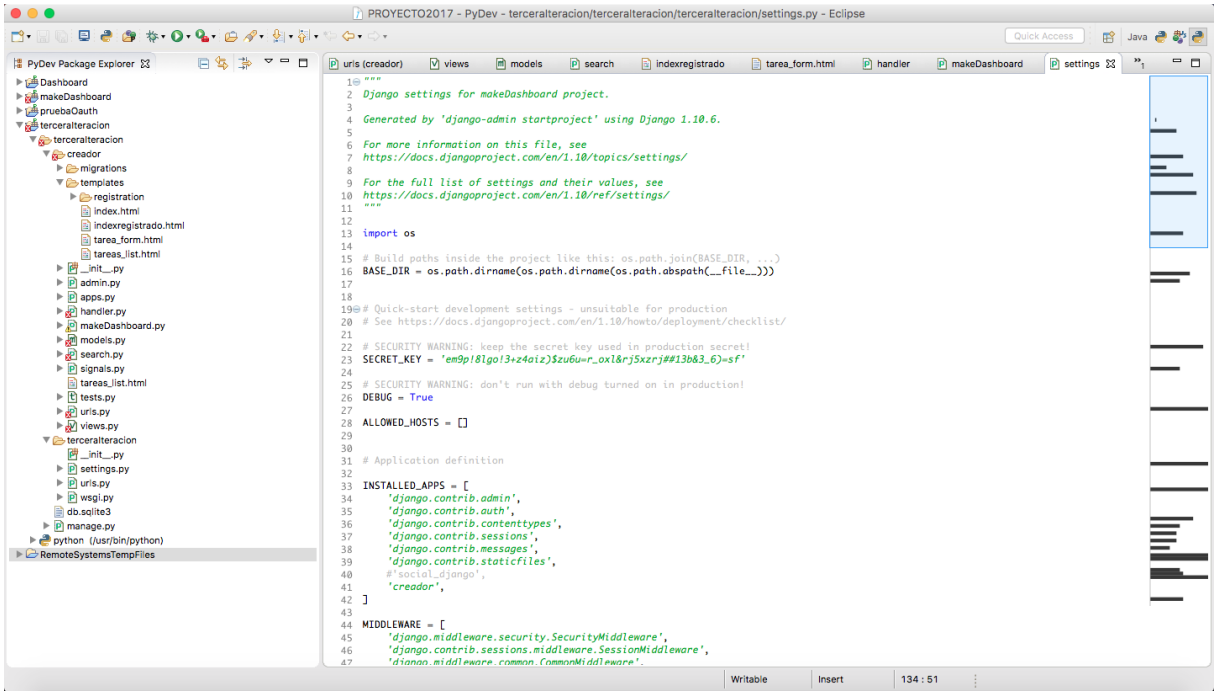


Figura 2.9: Interfaz del IDE Eclipse.

2.10. Perceval

Perceval es un módulo Python para recuperar datos de repositorios relacionados con el desarrollo de software.

2.10.1. Historia

Perceval pertenece al software de código libre para análisis de desarrollo software Grimoire Lab, que es una evolución del trabajo realizado durante más de 10 años por Bitergia, grupo de software libre de la URJC.

2.10.2. Características

Perceval es capaz de analizar datos de más de 20 fuentes diferentes. Algunas de las más usadas y conocidas son:

- Repositorios Git
- Proyectos GitHub
- Listas de correo
- Gerrit
- StackOverflow

Los datos que obtiene de las diferentes fuentes las devuelve en formato JSON, lo que lo hace muy práctico para la integración con otras tecnologías.

2.11. GrimoireELK

GrimoireELK es un sistema orientado a producir dashboards basados en Kibana mediante Perceval. Proporcionando un módulo Python (grimoire_elk) con facilidades para manejar Perceval.

2.11.1. Historia

GrimoireELK pertenece al software de código libre para análisis de desarrollo software Grimoire Lab, que es una evolución del trabajo realizado durante más de 10 años por Bitergia, grupo de software libre de la URJC.

2.11.2. Características

La abstracción que ofrece GrimoreELK sobre Perceval, ofrece la posibilidad de enriquecer los datos que se obtienen con éste y posteriormente cargarlo/descargarlo de Elasticsearch, es decir, complementa la funcionalidad de Perceval con un sistema de almacenamiento como Elasticsearch y la posibilidad de enriquecer los datos para mostrarlos mediante Kibana.

Para poder hacer esto GrimoireELK se basa principalmente en dos scripts Python:

- **p2o.py:** Usa Perceval para obtener los datos de cualquiera de las fuentes con las que es compatible, las almacena en formato JSON en Elasticsearch, crea una copia de los mismos, los prepara para Kibana y vuelve a almacenarlos en Elasticsearch.

- **kidash.py:** Toma los datos preparados para Kibana por p2o.py de Elasticsearch, y crea el dashboard.

2.12. Bootstrap

Bootstrap es un framework o conjunto de herramientas, basado en la filosofía de código abierto. A partir de un conjunto formado por plantillas, botones, formularios, cuadros, menús, tablas y una larga lista de elementos HTML con CSS, junto con extensiones de Javascript hacen de Bootstrap el proyecto más popular dentro de GitHub.

En nuestro caso, se selecciona usar Bootstrap, debido a que sus elementos ya definidos, así como sus plantillas base hacen que la creación de una interfaz web sea relativamente sencilla, teniendo una apariencia aceptable, sirviendo como base para un desarrollo más profundo.

2.12.1. Historia

Bootstrap fue desarrollado por Mark Otto y Jacbod Thornton pertenecientes a la empresa Twitter, debido a la necesidad de aunar en una sola librería el desarrollo de interfaces de usuario, que hasta ese momento se realizaban con varias, lo que provocaba inconsistencias entre ellas, así como una gran carga de mantenimiento.

El lanzamiento de la primera versión tuvo lugar en agosto del año 2011, colocándose como el proyecto de desarrollo más popular en GitHub en menos de un año, febrero de 2012.

2.12.2. Características

Como ya hemos indicado anteriormente, la principal característica que hace de Bootstrap un framework tan usado es su facilidad para obtener en un periodo de tiempo reducido una página web con una apariencia aceptable, gracias a su sistema de plantillas predefinidas y fácilmente modificables. Para que esto sea posible, Bootstrap ofrece además:

- **Plantillas:** hay dos tipos de plantillas para Bootstrap, las definidas por los propios creadores, y que se muestran en su página web, y las definidas por terceros a partir de los elementos predefinidos que proporciona Bootstrap. La existencia de estas plantillas colabora en la facilidad del diseño.

- **Elementos predefinidos:** el framework tiene predefinidos una serie de elementos como botones, formularios o tablas. Para el uso de estos elementos es necesario importar los ficheros que Bootstrap pone al servicio del desarrollador, siendo necesario al menos los archivos *bootstrap.min.css* con las definiciones de estilo y *jquery.min.js* para las definiciones de Javascript realizadas con jQuery.

  Éstos elementos se añaden de forma sencilla al código HTML, debido a que sus definiciones de estilo ya están importadas, e indicando el nombre concreto en el parámetro *class* del elemento, éste adquiere las propiedades y la apariencia predeterminada.

- **Diseño basado en rejilla:** para definir el *layout* con el que se presentará la página, Bootstrap recurre a las rejillas, es decir, divide la estructura de la página en doce columnas que se adaptan según sea el tamaño de ésta.

  Para el manejo de la rejilla, se definen clases para las filas y las columnas, éstas últimas de dividen en varias según las divisiones que ocupen de las doce comentadas anteriormente. Además ofrece clases adicionales para la mejor integración con tablets y móviles.

- **Compatibilidad con la mayoría de navegadores:** el funcionamiento en la mayoría de los navegadores, es otra razón por la que Bootstrap alcanza la popularidad que tiene, debido a que desarrolladores de diferentes lenguajes y navegadores pueden usarlo sin restricciones.

- **Plug-ins de Javascript:** la integración de Bootstrap con Javascript se basa en la librería jQuery, que proporciona elementos adicionales para la interfaz de usuario, como diálogos, autocompletados y una gran lista de elementos dinámicos que mejoran la experiencia del usuario.

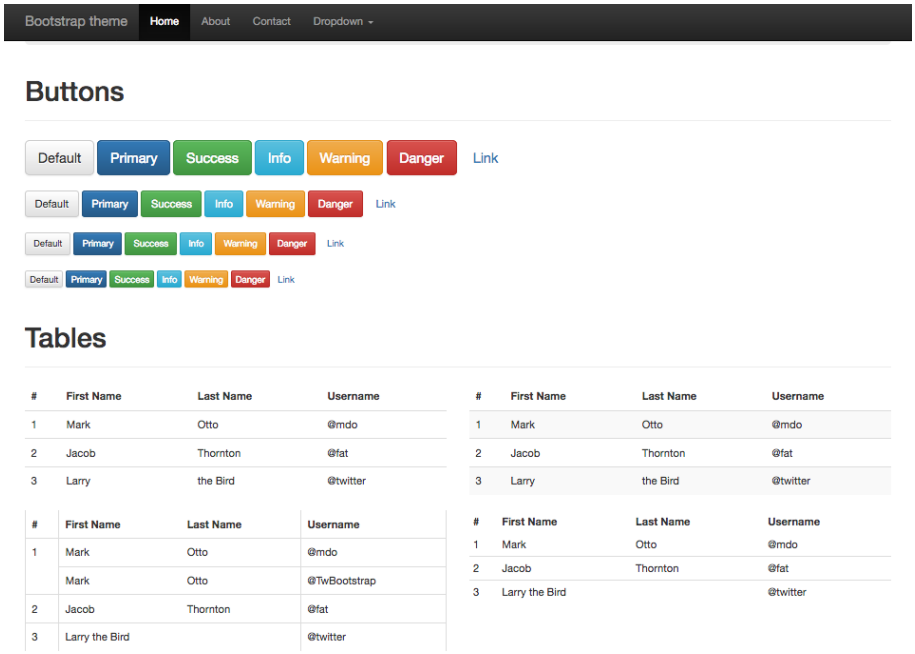


Figura 2.10: Ejemplo de plantilla predefinida por Bootstrap.

2.13. CSS

El nombre completo de CSS es hoja de estilo en cascada (del inglés Cascading Stylesheets, CSS). Se trata de un lenguaje de diseño gráfico para definir y crear la presentación de un documento estructurado escrito en un lenguaje de marcado. Generalmente CSS se muestra acompañado de HTML, pero es compatible con otros lenguajes como SVG, XUL o RSS.

Para nuestro caso el uso de CSS se hace totalmente necesario en el momento que se decide hacer la implementación de la interfaz web con Bootstrap, ya que éste lo utiliza por definición.

2.13.1. Historia

La existencia de las hojas de estilo se remontan al poco de la creación del lenguaje de etiquetas SGML, por la necesidad de aplicar estilos a los diferentes documentos electrónicos. El crecimiento del uso de internet y el lenguaje HTML, junto con la diversidad de navegadores hacían necesaria la estandarización de la aplicación de estilos a los documentos electrónicos. Por este motivo el W3C (World Wide Web Consortium) propuso la creación de un lenguaje de hojas de estilo específico para HTML. A la propuesta se presentaron nueve opciones de las cuales dos fueron elegidas. La primera opción era CHSS realizada por Håkon Wium Lie y la segunda SSP propuesta por Bert Bos, pero pronto unieron ambas propuestas para crear CSS (Cascading Style Sheets).

En el año 1995, W3C apuesta por CSS y lo incluye en el grupo de trabajo de HTML, saliendo a la luz a finales del año 1996 la primera versión conocida como: CSS nivel 1.

2.13.2. Características

La combinación de CSS, con HTML y Javascript es hoy en día el conjunto más utilizado para el desarrollo de páginas web atractivas para el usuario. Según la documentación de CSS[10], éste requiere de las siguientes características:

- **Separación de contenido y forma:** la separación del contenido del documento de la forma de presentarlo hace posible que cualquier documento HTML pueda representarse con diferentes apariencias según el fichero CSS que se le aplique, o por el contrario que varios documentos HTML (por ejemplo una aplicación web) pueda compartir la misma forma de presentación, ya sea de forma completa o de forma parcial.
- **Jerarquía:** el lenguaje CSS define un sistema de jerarquía en cascada para que, en el caso de que existan varias reglas CSS para un elemento determinado, el sistema sea capaz de decidir qué regla tiene prioridad sobre las demás.
- **Eficiencia:** recoger la información de estilo en un único fichero, hacer que se reduzcan los datos que se transmiten, ya que generalmente la hoja de estilo CSS queda almacenada en la caché del navegador. Además al contener el fichero CSS la información para varias documentos HTML, no es necesario repetir la información y por lo tanto hace que estos documentos sean más ligeros.

```
1  /* Move down content because we have a fixed navbar that is 50px tall */
2  .body {
3    padding-top: 50px;
4    padding-bottom: 20px;
5  }
6
7  .navbar-header {
8      width: 100%;
9      text-align: center;
10 }
11
12 .navbar-header > li {
13     float: none;
14     display: inline-block;
15 }
16
17 .ppal{
18     background-image: url(../imagenes/index.jpg);
19     background-size: cover;
20     width: 100vw;
21     height: 100vh;
22 }
```

Figura 2.11: Ejemplo de parte de una plantilla CSS.

2.14. LaTeX

LaTeX es un sistema de composición de textos, principalmente enfocado a la creación de documentos con alta calidad tipográfica. Por sus características se usa en documentos científicos y técnicos, en los que su contenido incluye expresiones matemáticas y técnicas. Como el resto de tecnologías implicadas en el proyecto LaTeX es software libre.

En nuestro caso, la elección de LaTeX no fue segura hasta que se dedicaron cinco minutos a su conocimiento, consultado su documentación[**?**], tiempo necesario para desechar la idea de hacer la memoria con un procesador de texto. La facilidad para guardar el estilo dentro del

documento y la calidad del resultado final son motivos más que suficientes para seleccionar LaTeX como elemento para la generación de la memoria del proyecto.

2.14.1. Historia

LaTeX tiene su origen en un compositor de texto más primitivo llamado TeX. TeX fue creado por Donald Knuth, tras el descontento con la impresión de uno de sus libros, comenzando a trabajar en él en mayo de 1977, apoyado por la American Mathematical Society.

Posteriormente, al inicio de los años ochenta, Leslie Lamport, basándose en TeX, comenzó a desarrollar un sistema con el objetivo de conseguir un mayor nivel de abstracción sobre los comando de TeX, para que los autores se preocuparan más de la estructura que de los formatos. Los formatos quedaron a cargo de las hojas de estilo. Leslie Lamport lanzó la versión inicial en el año 1984.

2.14.2. Características

LaTeX trabaja con un fichero de texto que incluye un conjunto de instrucciones que acompañan al texto que quiere imprimirse. Las instrucciones serán las encargadas de colocar la información en un determinado lugar, de una determinada forma, añadir una imagen o la creación de índice a partir del conjunto de instrucciones determinadas para ello. Por supuesto el formato del resultado de compilar el fichero *.tex* con las instrucciones y la información, será un documento *.pdf*, aunque LaTeX también puede trabajar en otros formatos. Las principales características de LaTeX son:

- **Igual respuesta para diferentes entornos:** la respuesta que se obtiene de un documento en LaTeX, es decir, la salida del compilador, es la misma independientemente del entorno en el que se ejecute. No importa la pantalla o la impresora, ni siquiera el sistema operativo en el que se esté ejecutando el compilador, el resultado debe ser el mismo.

- **Diferentes formatos de salida:** como ya hemos señalado anteriormente el formato de salida, es habitualmente el formato *.pdf*, pero LaTeX ofrece la posibilidad de producir la salida en otros formatos como HTML, RTF o Postscript.

- **Capacidad de representación gráfica:** la elevada popularidad de la que dispone LaTeX se debe en gran parte a su capacidad para la representación de fórmulas matemáticas, ecuaciones o notación científica. Además LaTeX también ofrece soporte para la inclusión de imágenes de forma estructurada, acompañadas de anotaciones.

- **Complementos:** al tratarse de un sistema basado en el código abierto, permite que desarrolladores externos además de los propios creadores del sistema, proporcionen funcionalidad adicional para el manejo de diferentes campos, como la notación científica avanzada (la básica está incluida por defecto), diagrama UML, fórmulas matemáticas, manejo de imágenes o creación de gráficas en 2D y 3D.

```latex
\documentclass{book}

\usepackage{graphicx}
\graphicspath{ {imágenes/} }

\usepackage[utf8]{inputenc}
\usepackage[spanish]{babel}
\usepackage{caption}

\begin{document}

\section{Sección}

Aquí se cuenta lo correspondiente a esta sección.

\subsection{Una}

En esta \textbf{zona} se habla de \emph{COSAS}.

\begin{center}
    \includegraphics[width=0.4\textwidth]{lion-logo}
    \begin{figure}[h]
        \caption{Pie informativo relativo a la foto.}
    \end{figure}
\end{center}

\end{document}
```

(a) Texto en LaTeX

0.1. Sección

Aquí se cuenta lo correspondiente a esta sección.

0.1.1. Subsección

En la **subsección** se habla de *COSAS*.

Figura 1: Pie informativo relativo a la foto.

(b) Resultado de compilar el texto en LaTeX

Figura 2.12: Texto escrito en LaTeX y su resultado una vez compilado.

Capítulo 3

Desarrollo

3.1. Iteración 1. Conocimiento de las herramientas

3.1.1. Objetivos

El objetivo de la primera iteración es familiarizarse con las distintas herramientas y tecnologías que usaremos para llevar a cabo el proyecto. Para ello seguiremos el siguiente esquema.

- Conocer GrimoireLab.
- Familiarizarse con Elasticsearch.
- Script resumen de lo aprendido.

3.1.2. Conocer GrimoireLab

Para el conocimiento de las herramientas se sigue el tutorial: "GrimoireLab Training Tutorial".

En primer lugar se explica la herramienta Perceval, módulo Python a través del cual se obtienen datos de repositorios relacionados con desarrollo software en formato JSON. Perceval funciona con Git o GitHub además de otros. Nos centramos en el uso de GitHub, en el que la herramienta Perceval, puede usarse de diferentes formas:

1. **Sin credenciales:** Acceso básico a la API de GitHub y limitación del ancho de banda disponible.
2. **Con usuario y contraseña:** Ancho de banda ilimitado aunque no funciona si el usuario tiene activado el doble factor de autenticación.
3. **Con token de usuario:** Ancho de banda ilimitado.

A continuación, mediante GrimoireELK, sigo los pasos para obtener un dashboard en Kibana, para ello me baso en dos scripts de Python:

- **p2o.py**, que se encarga de obtener los datos con Perceval y guardarlos en Elasticsearch. Además prepara estos datos para usarlos con Kibana y vuelve a guardarlos en Elasticsearch.
- **kidash.py**, obtiene los datos preparados y crea el dashboard en Kibana.

3.1.3. Familiarizarse con Elasticsearch

Hasta el momento, cuando he necesitado una base de datos para el desarrollo de software en la carrera, he usado una basada en SQL (mySQL o SQLlite).

La información en Elasticsearch se guarda en un servidor, en formato JSON, por lo tanto es necesario conocer los módulos de Python adecuados para el manejo de los datos en Elasticsearch. Este manejo puede hacerse mediante *HTTP urllib* o *HTTP Requests*, pero por comodidad se usaran los módulos .elasticsearchz .elasticsearch_dsl"que aportan un nivel superior de abstracción cada uno. Para afianzarse en el uso de dichas librerías se generan varios scripts, en los que se crean, modifican o borran los índices en Elasticsearch.

3.1.4. Script resumen de lo aprendido

Finalmente, una vez completado el tutorial, plasmo lo aprendido en un script. La funcionalidad del script será: recibiendo un propietario y un repositorio GitHub, creará un dashboard en Kibana a partir de ellos, con la opción de añadirlo a los ya existentes o borrar los anteriores y crearlo.

Los pasos de ejecución del script son:

1. **Evaluación de los argumentos que recibe el script:** mediante el módulo .argparseçomprobamos los argumentos que recibe el script, el usuario después de la opción -u, el repositorio después de la opción -r, y la existencia o no de la opción -add para añadir.

2. **Borrar o mantener los índices actuales:** en función de la existencia de la opción -add, se borraran o se mantendrán los índices dónde p2o.py guarda la información.

3. **Ejecución de p2o.py y kidash.py:** finalmente se ejecutan los scripts, comprobando entre la ejecución de ambos la existencia de un fichero auxiliar necesario, para en caso de no existir descargarlo.

3.2. Iteración 2. Primera aproximación con Django

3.2.1. Objetivos

Una vez completado el tutorial de GrimoireLab, y por lo tanto, conocidos los entornos y herramientas principales que usaremos en adelante, llega el momento de ponerlos en práctica mediante una aplicación en Django.

El objetivo es crear una aplicación en Django que pida al usuario mediante un POST la información sobre el propietario/organización y el repositorio que quiere analizar, para que posteriormente el servidor almacene esta información en Elasticsearch. De forma paralela, una adaptación del script creado en la iteración 1, debe acceder al contenido del índice Elasticsearch para ir ejecutando las tareas pendientes, creando finalmente en Kibana el dashboard correspondiente.

3.2.2. Django

Django es un framework para aplicaciones web escrito en Python, y que nos va a dar soporte para crear nuestro lado del servidor. Durante una de las asignaturas del proyecto utilizamos Django para crear una aplicación web, pero desde entonces prácticamente no he vuelto a utilizarlo, por lo que antes de empezar con el desarrollo de la app, consulto un par de tutoriales[11] para volver a familiarizarme con el framework.

Para retomar el uso de Django, seguiremos las siguientes etapas:

- Creación del proyecto.
- Manejo de los archivos del proyecto.
- Uso de plantillas.

3.2.3. La aplicación con Django

Para desarrollar esta primera aplicación marcaremos tres etapas:

- Creación del formulario POST.
- Almacenamiento en Elasticsearch de los datos obtenidos con el POST.
- Adaptación del script para la creación del dashboard.

3.2.3.1. Creación del formulario POST

En primer lugar crearemos un proyecto Django que llamaremos *makeDashboard*, en cuyo interior tendremos una app Django con el nombre de *creador*.

Antes de darle al cliente una interfaz para poder darle información a la aplicación, es necesario crear el modelo en Django dónde se guardará la información para posteriormente almacenarla en Elasticsearch.

A este conjunto de datos lo llamaremos *tarea*, y constará de:

- Campo con el nombre del propietario/organización.

- Campo con el nombre del repositorio.
- Campo con el estado de la tarea.

Esta información será la que se solicita al cliente mediante un formulario POST en Django.

Para comprobar que la información ha quedado almacenada en el modelo Django necesitamos crear un usuario para el sitio de administración de Django, lo haremos con *python manage.py createsuperuser*. Una vez creado, y con el servidor activo (*python manage.py runserver*), accedemos al sitio de administración *http://127.0.0.1:8000/admin/*, mostrando la siguiente página:

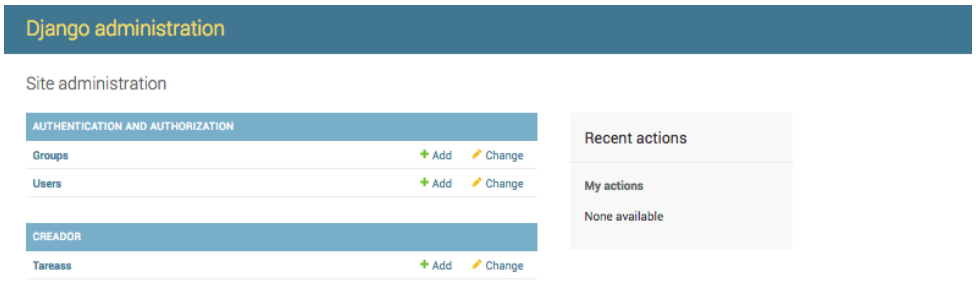


Figura 3.1: Página del módulo de administración de Django.

en la imagen vemos el apartado de la aplicación *creador* y una línea correspondiente al modelo *tarea*, si hacemos click sobre él:

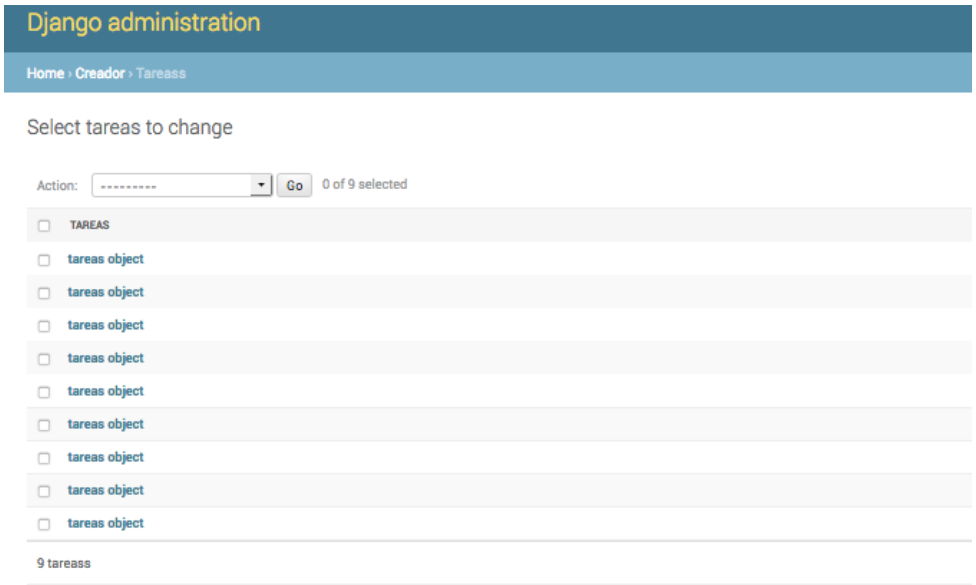


Figura 3.2: Página que muestra las instancias creadas del modelo.

en el que cada línea corresponde a las distintas instancias creadas del modelo. Si hacemos click en alguna de las líneas veremos el contenido de la instancia.

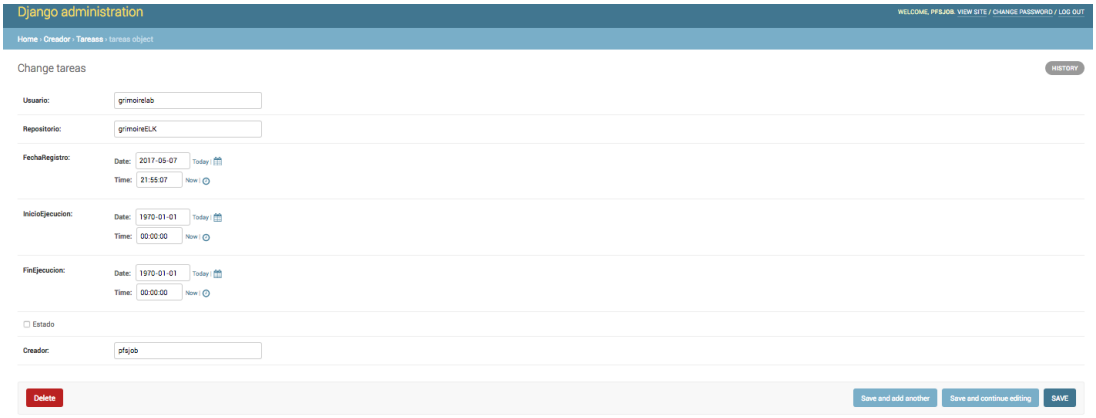


Figura 3.3: Página que muestra el contenido de una instancia del modelo.

Por lo tanto, ya tenemos un sistema que toma los datos al usuario y los guarda en el modelo correspondiente.

3.2.3.2. Almacenamiento en Elasticsearch de los datos obtenidos con el POST

Llega el momento de almacenar los datos de forma persistente, para ello vamos a utilizar Elasticsearch como sistema de almacenamiento. Seguiremos un tutorial[12], que nos ayudará en la implementación.

En primer lugar hay que conocer que librerías Python dan soporte para el uso de Elasticsearch. Usaremos elasticsearch_ dsl.

Revisando la documentación[13] de la librería vemos que para crear un índice en Elasticsearch debemos crear una clase Python de tipo *DocType*, similar al modelo creado inicialmente.

```
14 class tareasIndex(DocType):
15       usuario = String()
16       repositorio = String()
17       estado = Boolean()
```

Figura 3.4: Clase DocType generada para nuestro proyecto.

esta clase *tareasIndex* tomará los valores del modelo Django *tareas*, y al pertenecer a la librería *DocType* de elasticsearch_dsl tiene la funcionalidad para poder crear una instancia en Elasticsearch.

Por lo tanto necesitamos tener la parte del código que se encarga de asignar los datos del modelo Django al elemento *DocType* que hemos creado. Para ello dentro del modelo crearemos una función con un objeto *tareasIndex* al que le asignamos los datos del modelo y finalmente gracias a la funcionalidad comentada guardemos en Elasticsearch.

```
36
37  class tareas(models.Model):
38      usuario=models.CharField(max_length=100)
39      repositorio=models.CharField(max_length=100)
40      estado=models.BooleanField(default=False)
41
42
43      def indexing(self):
44          obj = tareasIndex(
45              meta={'type': "doc_type", 'id': self.usuario+"-"+self.repositorio},
46              usuario=self.usuario,
47              estado=self.estado,
48          )
49          obj.save(index='tareas')
50          return obj.to_dict(include_meta=True)
51
```

Figura 3.5: Código Python del método *.indexing()*.

Los elementos *DocType*, además de los campos definidos en su creación poseen una parte de información llamada *meta* dentro de la cual se encuentran los campos que necesita Elasticsearch para su instanciación, como el identificador, que en este caso será una combinación del usuario y el repositorio.

Para terminar, es necesario activar la ejecución de esta parte del código. Para ello usaremos el concepto de señales en Django. Las señales en Django son acciones predefinidas que al ejecutarse, Django da la opción de manejar su respuesta, por ejemplo, a la hora de guardar una instancia del modelo, lo que en nuestro código será el momento adecuado para además de guardar el modelo, crear una instancia en Elasticsearch de dicho modelo.

3.2.3.3. Adaptación del script para la creación del dashboard

Una vez que el sistema es capaz de almacenar la información, hay que adaptar el script para que ejecute lo almacenado.

Si el script de la iteración 1 recibía los datos del repositorio y el usuario/propietario a través de sus argumentos, éste tendrá que incluir la funcionalidad para hacer una consulta a Elasticsearch e ir ejecutando el script con la información obtenida.

Mediante la librería elasticsearch_dsl, conseguimos la funcionalidad para pedir a Elasticsearch el contenido del índice *tareas*, y a partir de este darle la información al script para crear el dashboard.

3.3. Iteración 3. Integración de la aplicación con GitHub

3.3.1. Objetivos

Como vimos en el tutorial ”GrimoireLab Training Tutorial”, GitHub ofrece la posibilidad de proporcionar un token de usuario con el que la cantidad de información que nos da GitHub es ilimitada, siendo el uso sin token limitado, es decir, si usando perceval pido información a GitHub sin token, llegará un momento en el que alcanzaremos el límite, y tendremos que esperar un determinado tiempo para volver a solicitar información, esto con un token de usuario no pasaría.

Por lo tanto el objetivo de esta iteración sera el de dotar a la aplicación de la funcionalidad necesaria para que a través de GitHub (incluso con su interfaz web) el cliente se autentique y obtengamos su token de usuario para las posteriores peticiones.

3.3.2. Aplicaciones en GitHub

Un usuario registrado en GitHub tiene la posibilidad de registrar en su configuración de desarrollador una aplicación propia, la cual, podrá tener acceso a la autenticación en GitHub.

Por lo tanto el primer paso será registrar la aplicación dentro del perfil de usuario de GitHub, en el apartado *OAuth applications*.

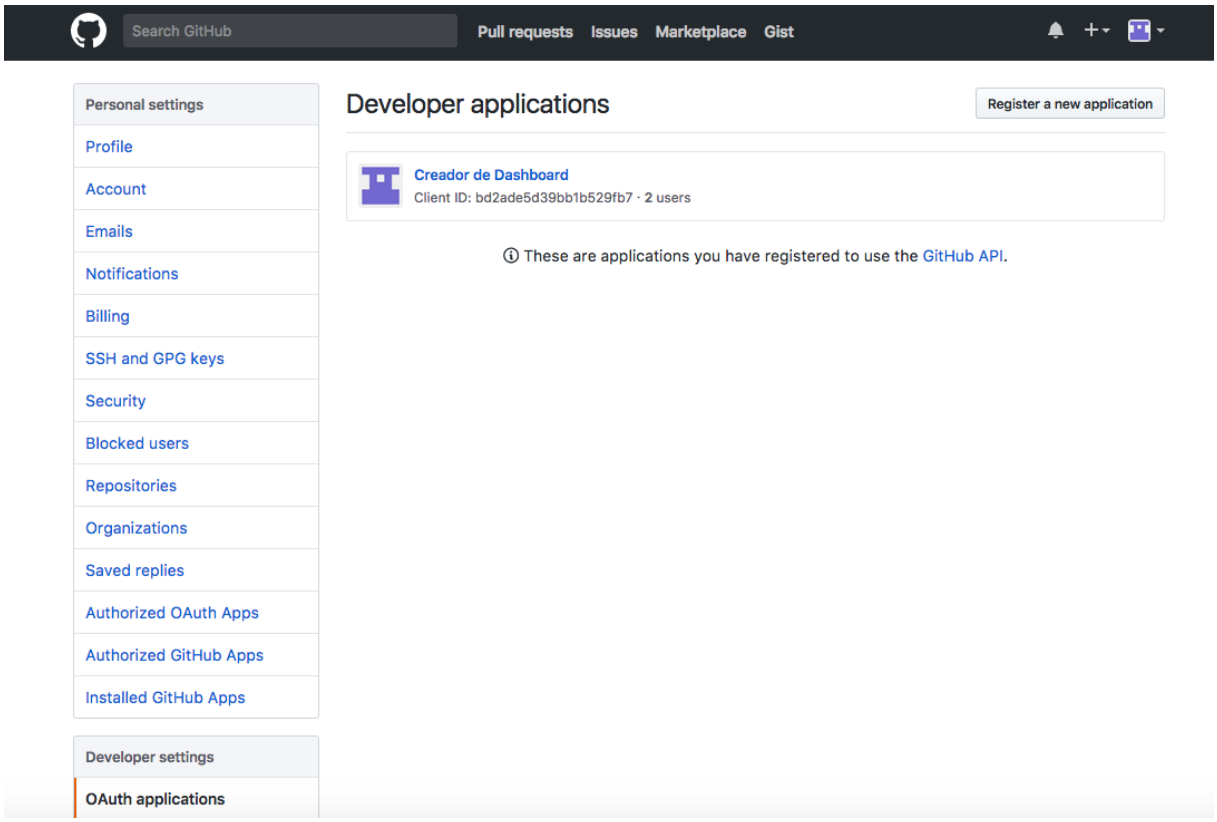


Figura 3.6: Pantalla inicial para el registro de una aplicación de desarrollador.

Si hacemos click sobre el botón *Register a new application*, accederemos a la pantalla de toma de datos para registrar la aplicación.

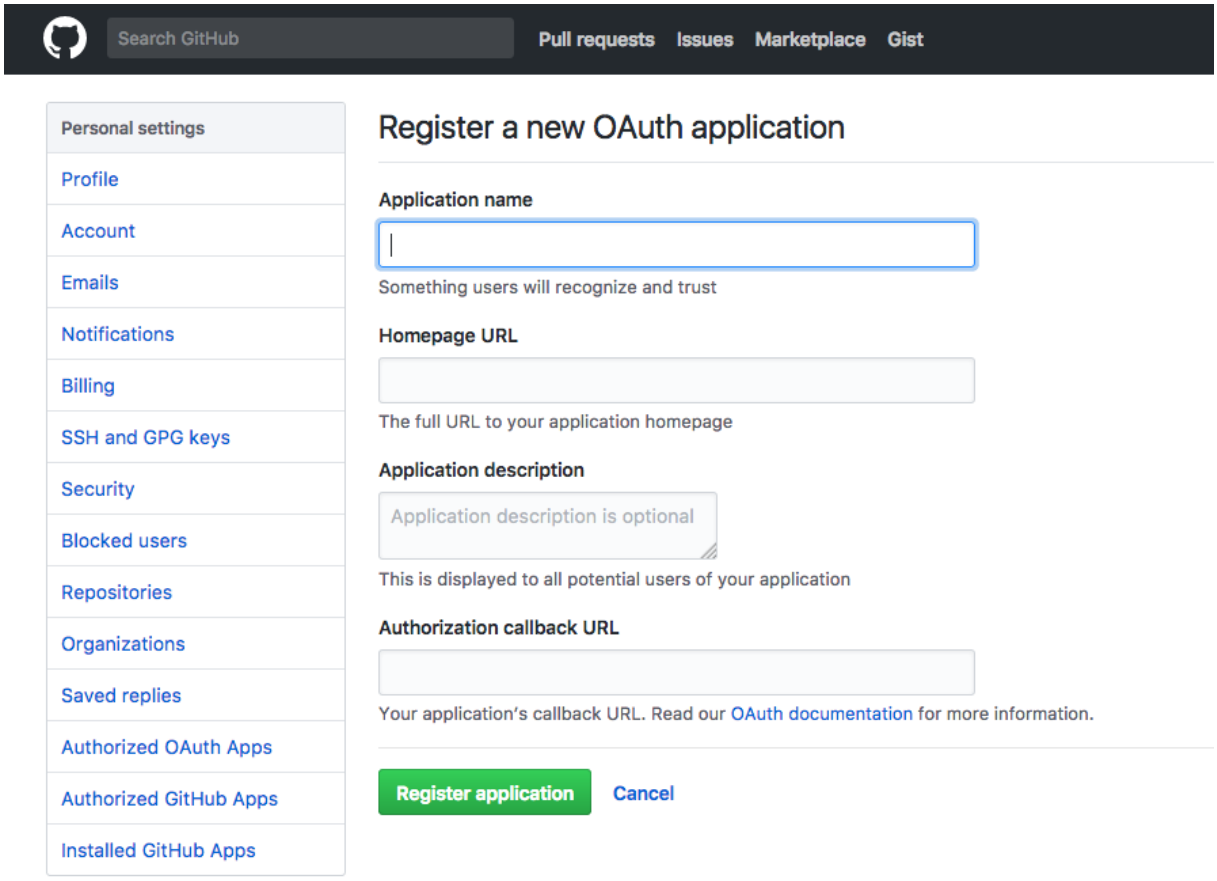


Figura 3.7: Pantalla de toma de datos para el registro de una aplicación en GitHub.

Los campos son los siguientes:

- **Application name:** nombre que le daremos a nuestra aplicación para GitHub.
- **Homepage URL:** página principal de nuestra aplicación.
- **Authorization callback URL:** página que solicitaremos para iniciar el proceso de autorización.

Una vez registrada la aplicación GitHub proporcionará dos identificadores propios para la aplicación y el usuario:

- Client ID.
- Client Secret.

Llegados a este punto, tenemos la aplicación registrada en GitHub, y debemos dotarla de la funcionalidad adecuada para hacer uso de la autenticación a través de GitHub, así como la obtención del token de usuario.

3.3.3. La API de GitHub

Para poder dotar a la app de la funcionalidad debemos consultar la *API* que proporciona GitHub, en el apartado para la autenticación con Oauth.

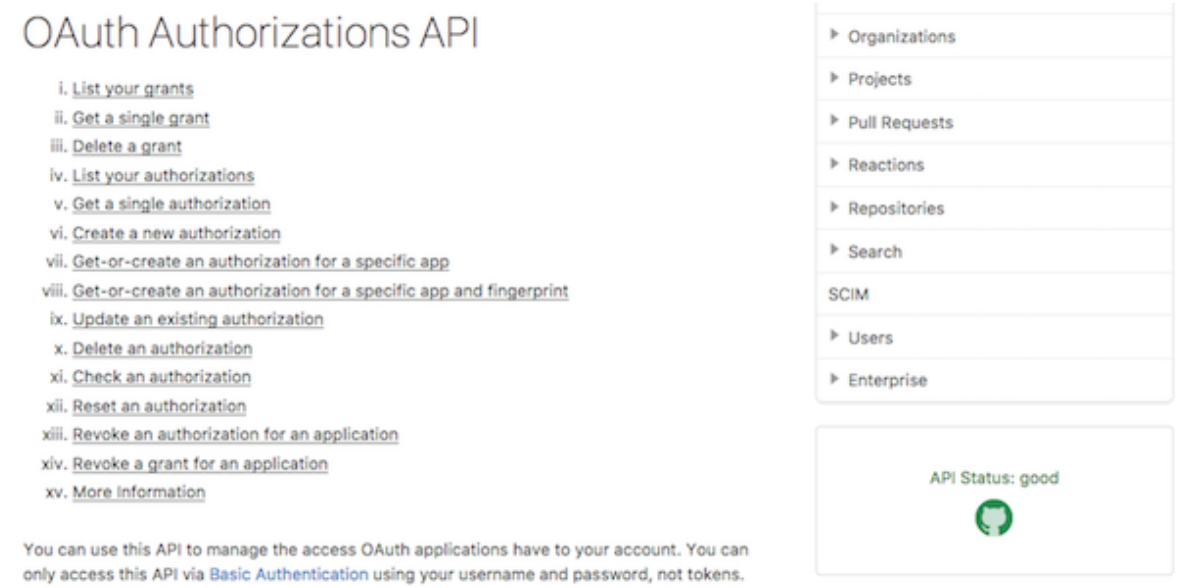


Figura 3.8: Página inicial de la API de GitHub.

El conjunto de funcionalidad que ofrece el apartado de Oauth es bastante amplio, pero tratan sobre la obtención de diferentes elementos a través de peticiones al servidor de GitHub, sin una interfaz web, que no es lo que buscamos. Por lo tanto debemos encontrar este proceso con interfaz web, para ello analizamos otros apartados de la API.

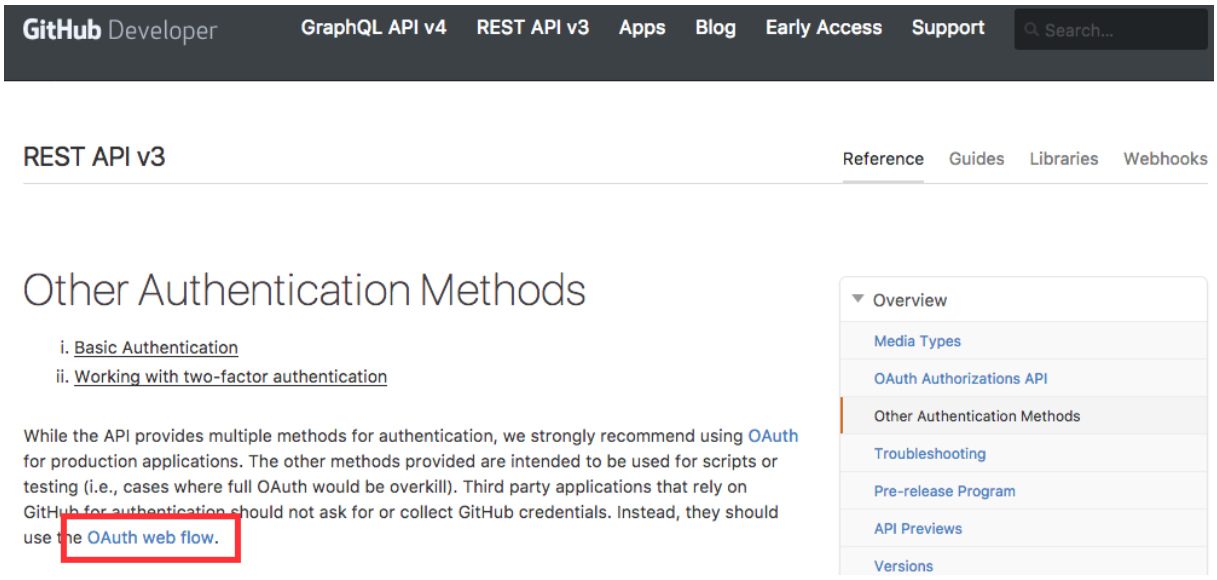
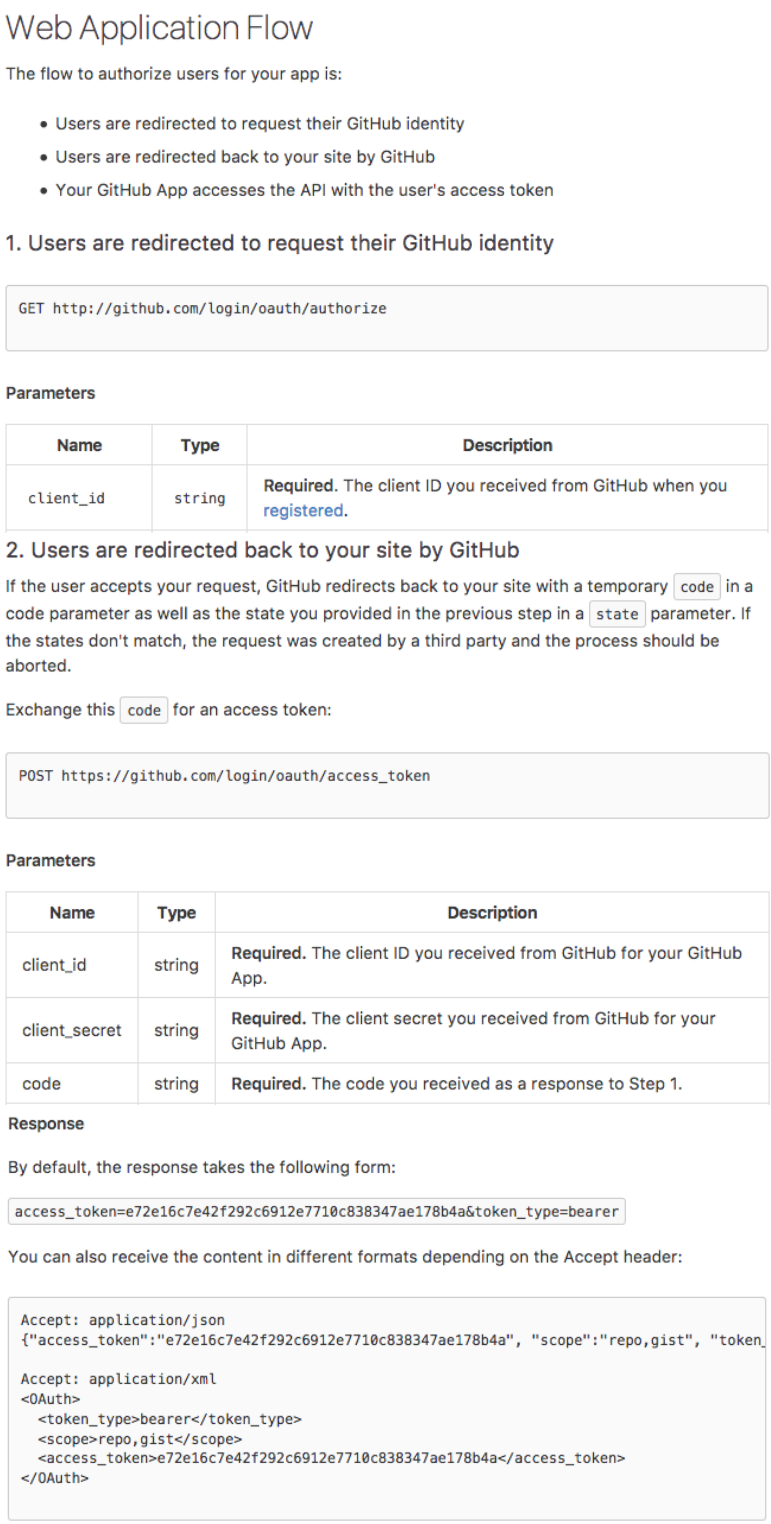


Figura 3.9: Página de la API de GitHub.

En el apartado de otros métodos de autenticación, además de hablar de la autenticación básica y de la de doble factor, recomiendan en la introducción que para aplicaciones de terceros (nuestro caso), se use el método *Oauth Web Flow*.



Web Application Flow

The flow to authorize users for your app is:

- Users are redirected to request their GitHub identity
- Users are redirected back to your site by GitHub
- Your GitHub App accesses the API with the user's access token

1. Users are redirected to request their GitHub identity

```
GET http://github.com/login/oauth/authorize
```

Parameters

Name	Type	Description
client_id	string	**Required.** The client ID you received from GitHub when you registered.

2. Users are redirected back to your site by GitHub

If the user accepts your request, GitHub redirects back to your site with a temporary `code` in a code parameter as well as the state you provided in the previous step in a `state` parameter. If the states don't match, the request was created by a third party and the process should be aborted.

Exchange this `code` for an access token:

```
POST https://github.com/login/oauth/access_token
```

Parameters

Name	Type	Description
client_id	string	**Required.** The client ID you received from GitHub for your GitHub App.
client_secret	string	**Required.** The client secret you received from GitHub for your GitHub App.
code	string	**Required.** The code you received as a response to Step 1.

Response

By default, the response takes the following form:

```
access_token=e72e16c7e42f292c6912e7710c838347ae178b4a&token_type=bearer
```

You can also receive the content in different formats depending on the Accept header:

```
Accept: application/json
{"access_token":"e72e16c7e42f292c6912e7710c838347ae178b4a", "scope":"repo,gist", "token_

Accept: application/xml
<OAuth>
  <token_type>bearer</token_type>
  <scope>repo,gist</scope>
  <access_token>e72e16c7e42f292c6912e7710c838347ae178b4a</access_token>
</OAuth>
```

Figura 3.10: Proceso de autenticación con *Oauth Web Flow*.

En las capturas anteriores se muestran los pasos principales y necesarios para el proceso de autenticación vía web.

1. **Redirección a la web de autenticación:** necesitaremos hacer una petición HTTP del tipo GET a *http://github.com/login/oauth/authorize* con el parámetro obligatorio del client id (que obtuve al registrar mi aplicación en GitHub).

2. **Redirección a la homepage de la app:** una vez que el usuario autoriza a la aplicación, se redirige al usuario a la página principal. En la redirección va incluido el parámetro *code*, necesario para obtener el access_token.

3. **Obtención del access_token:** para obtener el access_token debemos pasar los parámetro en un POST a *https://github.com/login/oauth/access_token*:

   a) Client id.

   b) Client secret.

   c) Code.

   En la respuesta al POST, se encuentra el access_token. Dicha respuesta se puede solicitar en diferentes formatos como JSON o XML.

Ya tenemos el proceso que debemos seguir para conseguir la autenticación del usuario de la aplicación a través de GitHub. Ahora hay que integrarlo en nuestro entorno con Django.

3.3.4. Python Social Auth

Para dar soporte a la autenticación/autorización en Python existen varias librerías, pero finalmente nos decantamos por *Python Social Auth* cuya documentación[8] seguiremos para su uso.

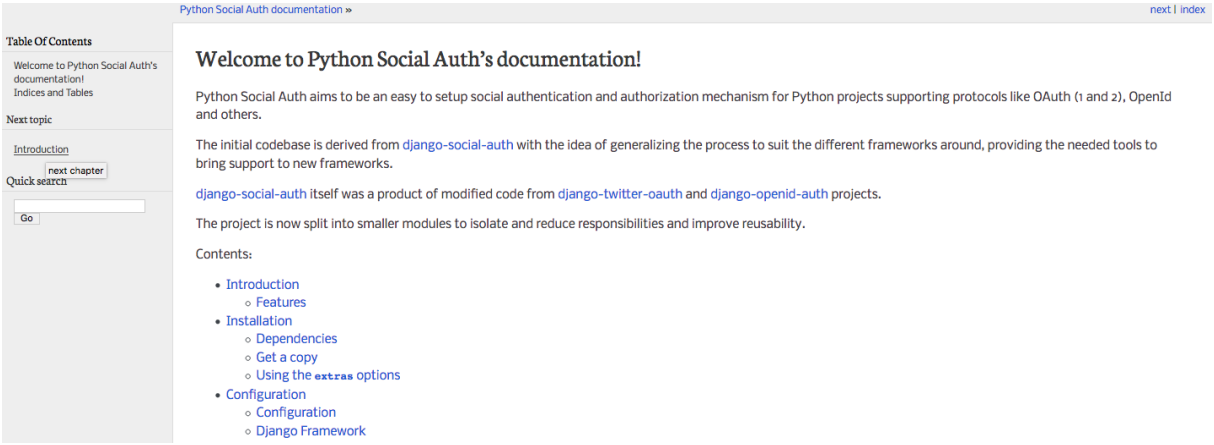


Figura 3.11: Página de la documentación de *Python Social Auth*.

La librería *Python Social Auth* incluye un apartado en su documentación[8] dedicado a su instalación en el framework Django, que seguiremos para aplicarlo en nuestra sistema.

Además de seguir la instalación de *Python Social Auth*, nos apoyaremos en un tutorial[14] para completar el proceso. Los pasos a seguir serán lo siguientes:

3.3.4.1. Instalación de *Python Social Auth*:

empezaremos con la instalación de la librería mediante el comando *pip*, añadiendo en el archivo *settings.py*, en el apartado ÏNSTALLED_APPS.[el] nombre de la librería.

```
31 # Application definition
32
33 INSTALLED_APPS = [
34     'django.contrib.admin',
35     'django.contrib.auth',
36     'django.contrib.contenttypes',
37     'django.contrib.sessions',
38     'django.contrib.messages',
39     'django.contrib.staticfiles',
40     'social_django',
41     'creador',
42 ]
```

Figura 3.12: Fragmento de *settings.py* con las aplicaciones instaladas.

3.3.4.2. Configuración de *Python Social Auth*:

para la configuración de la librería añadiremos diferentes líneas en el archivo *settings.py*:

- MIDDLEWARE: Añadimos la línea para ”SocialAuthExceptionMiddleware”.

```
43
44 MIDDLEWARE = [
45     'django.middleware.security.SecurityMiddleware',
46     'django.contrib.sessions.middleware.SessionMiddleware',
47     'django.middleware.common.CommonMiddleware',
48     'django.middleware.csrf.CsrfViewMiddleware',
49     'django.contrib.auth.middleware.AuthenticationMiddleware',
50     'django.contrib.messages.middleware.MessageMiddleware',
51     'django.middleware.clickjacking.XFrameOptionsMiddleware',
52     'social_django.middleware.SocialAuthExceptionMiddleware',
53 ]
```

Figura 3.13: Fragmento de *settings.py* con el apartado MIDDLEWARE.

- TEMPLATES: Actualizamos también la zona de templates.

```
56
57 TEMPLATES = [
58     {
59         'BACKEND': 'django.template.backends.django.DjangoTemplates',
60         'DIRS': [os.path.join(os.path.split(os.path.abspath(os.path.dirname(__file__)))[0],'.'),],
61         'APP_DIRS': True,
62         'OPTIONS': {
63             'context_processors': [
64                 'django.template.context_processors.debug',
65                 'django.template.context_processors.request',
66                 'django.contrib.auth.context_processors.auth',
67                 'django.contrib.messages.context_processors.messages',
68                 'social_django.context_processors.backends',
69                 'social_django.context_processors.login_redirect',
70             ],
71         },
72     },
73 ]
```

Figura 3.14: Fragmento de *settings.py* con el apartado TEMPLATES.

- AUTHENTICATION_BACKENDS: Actualizamos también la zona de backends con las líneas correspondientes a GitHub y Oauth.

```
106
107 AUTHENTICATION_BACKENDS = (
108     'social_core.backends.open_id.OpenIdAuth',
109     #'social_auth.backends.contrib.github.GithubBackend',
110     'social_core.backends.github.GithubOAuth2',
111     'django.contrib.auth.backends.ModelBackend',
112 )
```

Figura 3.15: Fragmento de *settings.py* con el apartado de BACKENDS.

- LOGIN_URL, LOGOUT_URL, LOGIN_REDIRECT_URL: Añadimos por último los valores por defecto de estos campos.

```
135
136 LOGIN_URL = 'login'
137 LOGOUT_URL = 'logout'
138 LOGIN_REDIRECT_URL = '/tareas/index/'
```

Figura 3.16: Fragmento de *settings.py* con los campos.

También hay que introducir en el archivo *urls.py*, la entrada para la librería Django Social Auth.

```
22 urlpatterns = [
23     url(r'^creador/', include('creador.urls')),
24     url(r'^admin/', admin.site.urls),
25     url(r'^tareas/', include('creador.urls', namespace="utareas")),
26     url(r'^oauth/complete/github/', views.indexregistrado ),
27     url(r'^oauth/', include('social_django.urls', namespace='social')),
28     #url(r'^oauth/complete/github/', views.indexregistrado ),
29 ]
```

Figura 3.17: Fragmento de *urls.py* con la librería.

3.3.4.3. Configuración en GitHub

Llegados a este punto es conveniente tener registrada en GitHub la aplicación, cosa que ya hicimos en iteraciones anteriores, por lo que sólo tendremos que actualizar el apartado de *Authorization callback URL* para adaptarlo a *Python Social Auth*. La url que pondremos será la del servidor donde esté alojada la aplicación seguida de: */oauth/complete/github/*.

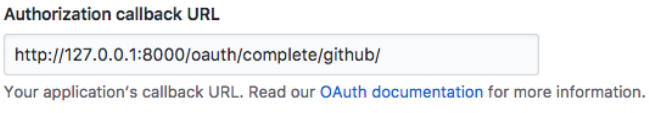


Figura 3.18: Introducción de la *Authorization callback URL*.

En nuestro caso el servidor será localhost.

Además tenemos que almacenar en *settings.py*, los valores de Client_ID y Client_Secret.

```
131
132 SOCIAL_AUTH_GITHUB_KEY = 'bd2ade5d39bb1b529fb7'
133 SOCIAL_AUTH_GITHUB_SECRET = '35b46ffbac1f02ea2ca84f44d2450fd00ffd6f40'
```

Figura 3.19: Valores en *settings.py*.

Para mantener a salvo estos valores es conveniente ocultarlos lo máximo posible, por lo que en iteraciones posteriores intentaremos buscar la forma de hacerlo.

3.3.5. Interfaz de prueba:

Para terminar crearemos una plantilla html, para comprobar el funcionamiento de la implementación.

PANTALLA INICIAL

Figura 3.20: Pantalla inicial de la interfaz de prueba.

PANTALLA REGISTRO EN GITHUB

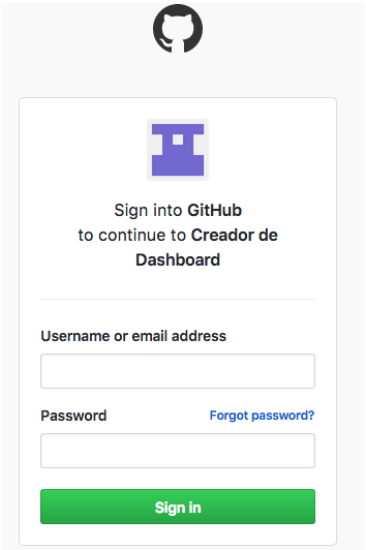


Figura 3.21: Pantalla de registro en GitHub.

PANTALLA AUTORIZACIÓN GITHUB

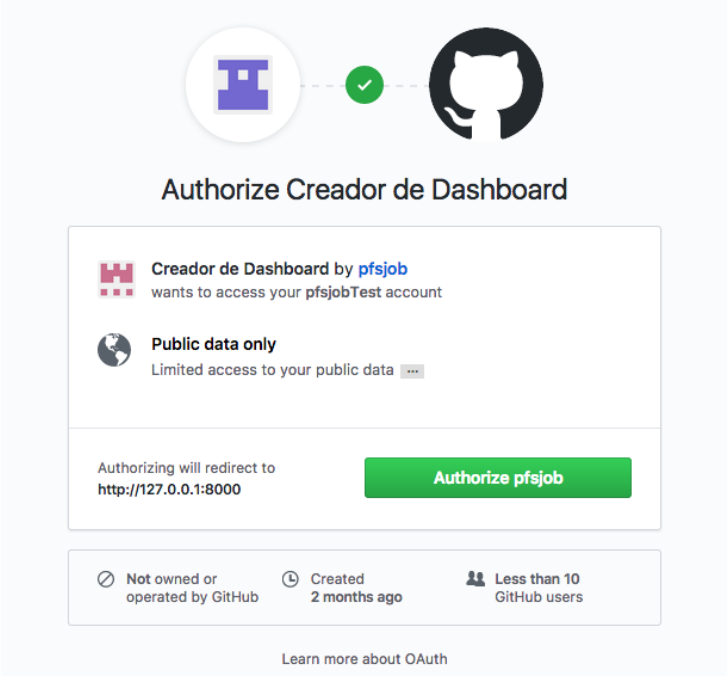


Figura 3.22: Pantalla de autorización en GitHub.

REDIRECCIÓN A APP

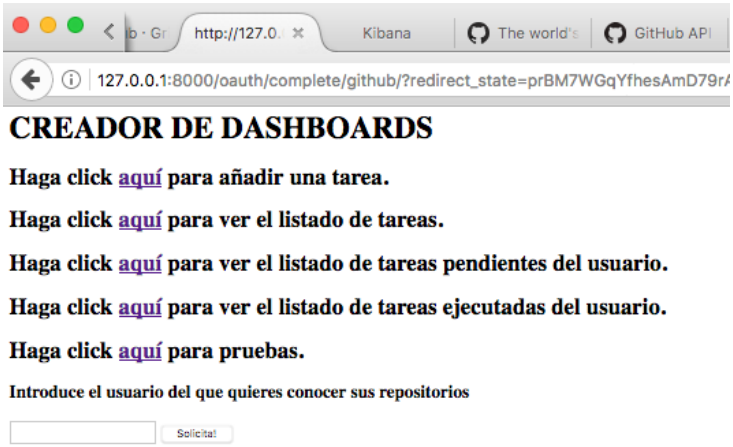


Figura 3.23: Redirección a la pantalla principal de la aplicación.

Esta última pantalla muestra las opciones que el usuario de la app tiene disponible una vez autenticado a través de GitHub.

3.3.6. Obtención del access_token:

Una vez autenticado el cliente a través de GitHub, debemos solicitar el token del usuario, para poder operar con él. Para ello, como ya vimos en el apartado *Web Application Flow* de la

API de GitHub. Una vez que el cliente autoriza a la aplicación, el servidor, en la respuesta, devuelve un parámetro code, que junto con el Client_ID y el Client_Secret usaremos para solicitar el access_token.

Para solicitar el token debemos hacer un POST a *https://github.com/login/oauth/access_token*, con los parámetros anteriormente nombrados.

Esta acción debemos llevarla a cabo en Python, y para ello usamos la librería Requests, que nos da soporte para hacer peticiones HTTP (POST, GET, PUT,...).

```
51    url = 'https://github.com/login/oauth/access_token'
52    header = {'content-type':'application/json', 'Accept': 'application/json'}
53    payload = {}
54    payload['client_id']=client_id
55    payload['client_secret']=client_secret
56    payload['code']=codigo
57
58    res = requests.post(
59        url,
60        data = json.dumps(payload),
61        headers=header
62        )
```

Figura 3.24: Fragmento Python de petición HTTP con Requests.

En la imagen vemos el uso de la librería Requests, si esto lo trasladamos a lo visto en la API de GitHub:

- En el campo url, está la dirección que GitHub nos indica para obtener el access_token, *https://github.com/login/oauth/access_token*.
- En el campo headers están las cabeceras Http de la petición, en este caso, para que la respuesta sea en formato JSON.
- En el campo data, en el interior de una lista, están almacenados los parámetros necesarios para obtener la respuesta deseada.

3.4. Iteración 4. Consolidación de la aplicación

3.4.1. Objetivos

Con la autenticación a través de GitHub, tenemos la estructura base para nuestra app, pero es necesario dotarla de funcionalidad adicional para poder ofrecer una correcta experiencia al usuario:

- Campos adicionales en el modelo.
- Almacenamiento en Elasticsearch del access_token y el usuario.
- Query en Elasticsearch.
- Listado de tareas pendientes de ejecutar del usuario.
- Listado de tareas ya ejecutadas del usuario.
- Conocer los repositorios que posee un usuario determinado.
- Actualización del script de ejecución.

3.4.2. Campos adicionales en el modelo

Como vimos en la segunda iteración, el modelo inicialmente creado para el almacenamiento y posterior creación de los dashboards, incluía los siguientes campos:

- Campo con el nombre del propietario/organización.
- Campo con el nombre del repositorio.
- Campo con el estado de la tarea.

Con esta información la app sólo es capaz de ejecutar el dashboard pero no puede dar al usuario información de cuándo lo ha hecho, cuanto ha tardado o en que momento ha terminado. Por lo tanto debemos agregar campos al modelo para poder registrar y posteriormente informar. Los campos que añadiremos son:

- Campo para registrar el momento en el cual el usuario ha añadido la tarea a la aplicación. El nombre será *fechaRegistro*. Tomará el valor por defecto de la fecha y hora del momento en que se toman los datos.
- Campo para registrar el momento en que se inicia la creación del dashboard, el nombre será *inicioEjecucion*. Por defecto tendrá el valor del 1 de Enero de 1970, este campo no se modificará hasta que el script de ejecución no inicie la misma.
- Campo para registrar el momento en que se finaliza la creación del dashboard, el nombre será *finEjecucion*. Por defecto tendrá el valor del 1 de Enero de 1970, este campo no se modificará hasta que el script de ejecución no finalice la misma.
- Campo para registrar el usuario que solicita la creación del dashboard, el nombre será *creador*.

Por lo tanto la nueva versión del modela quedará:

```
36
37 class tareas(models.Model):
38     usuario=models.CharField(max_length=100)
39     repositorio=models.CharField(max_length=100)
40     fechaRegistro=models.DateTimeField(default=timezone.now)
41     inicioEjecucion=models.DateTimeField(default=horainicio)
42     finEjecucion=models.DateTimeField(default=horainicio)
43     estado=models.BooleanField(default=False)
44     creador=models.CharField(max_length=100, default="None")
```

Figura 3.25: Modelo Django *tareas* actualizado.

Además de esto también hay que actualizar la funcionalidad relacionada con el almacenamiento en Elasticsearch, como la función *indexing* o la clase *tareasIndex*.

```
48
49     def indexing(self):
50         obj = tareasIndex(
51             meta={'type': "doc_type", 'id': self.usuario+"-"+self.repositorio},
52             usuario=self.usuario,
53             repositorio=self.repositorio,
54             fechaRegistro=self.fechaRegistro,
55             inicioEjecucion=self.inicioEjecucion,
56             finEjecucion=self.finEjecucion,
57             estado=self.estado,
58             creador=self.creador,
59         )
60         obj.save(index='tareas')
61         return obj.to_dict(include_meta=True)
```

```
13
14 class tareasIndex(DocType):
15     usuario = String()
16     repositorio = String()
17     fechaRegistro = Date()
18     inicioEjecucion = String()
19     finEjecucion = Date()
20     estado = Boolean()
21     creador = String()
```

Figura 3.26: Actualizaciones complementarias al modelo *tareas*.

3.4.3. Almacenamiento en Elasticsearch del access_token y el usuario

Cuando el usuario se autentica a través de GitHub, la aplicación solicita a éste el access_token, con el que operaremos cada vez que creemos un dashboard. Esta información debemos almacenarla en Elasticsearch, junto con el nombre del usuario propietario del access_token. Se hace necesario crear un nuevo modelo Django con su correspondiente instancia en Elasticsearch para almacenar dicha información.

El nuevo modelo Django se llamará *usuariosapp* y constará de dos campos:

- Campo con el nombre del usuario propietario del access_token, lo llamaremos *usuario*.

- Campo con el valor del access_token, lo llamaremos *token*.

Al crear un nuevo modelo, hay que acompañarlo también de la funcionalidad necesaria para almacenarlo en Elasticsearch, es decir, como ya hemos visto, la función *indexing* y la clase *usuariosAppIndex*.

```
22
23  class usuariosapp(models.Model):
24      usuario=models.CharField(max_length=100)
25      token=models.CharField(max_length=100)
26
27      def indexing(self):
28          obj = usuariosAppIndex(
29              meta={'type': "doc_type", 'id': self.usuario},
30              usuario=self.usuario,
31              token=self.token,
32          )
33          obj.save(index='usuariosapp')
34          return obj.to_dict(include_meta=True)
```

```
 9
10  class usuariosAppIndex(DocType):
11      usuario = String()
12      token = String()
```

Figura 3.27: Actualizaciones complementarias al modelo *usuarioapp*.

Hasta ahora la información que se almacenaba en Elasticsearch se guardaba sólo en el momento en el que el usuario añadía una tarea que ejecutar en la aplicación mediante la función *indexing*, y no era necesario ni modificar ni crear una instancia en Elasticsearch en otro momento de la ejecución.

Por lo tanto necesitamos crear la funcionalidad necesaria para modificar o añadir información. Lo haremos a partir de la librería elasticsearch_dsl, a través del siguiente proceso:

1. Después de obtener el access_token y el nombre del propietario hay que crear una instacia de la clase *usuariosapp*, para poder usar posteriormente la función *indexing*.

2. Abrimos una conexión con Elasticsearch y mediante la clase *Search* hacemos una búsqueda de nuestro índice de Elasticsearch donde tenemos los usuarios, *usuariosapp*, para posteriormente con la clase *scan*, preparar la información de salida para ser recorrida en un bucle.

3. Con la información ya preparada para el bucle, iniciamos el mismo, comprobando en cada iteración si el usuario que vamos almacenar está o no en la lista, si está actualizamos su token, y si no está creamos una nueva entrada en el índice con la información. Para la creación de la nueva entrada usaremos la función *indexing*.

4. En teoría ya estaría completa la funcionalidad para la creación de un nuevo índice o la actualización de uno ya existente, pero hay que comprobar si el índice está vació, ya que de lo contrarío al intentar recorrer el bucle saltaría una excepción de *TransportError*, la cual debemos capturar, creando un nuevo índice en el caso de que aparezca.

```
86
87      ##############################
88      nuevoUsuario = usuariosapp(
89          usuario=login,
90          token=access_token,
91      )
92      es = Elasticsearch()
93      req = elasticsearch_dsl.Search(using=es, index='usuariosapp')#, doc_type='summary')
94      resp = req.scan()
95      try:
96          for x in resp:
97              if x['usuario']==login:
98                  entrada = usuariosAppIndex.get(id=x.meta.id, using=es, index='usuariosapp')
99                  entrada.update(using=es, token=access_token)
100             else:
101                 nuevoUsuario.indexing()
102     except TransportError:
103         nuevoUsuario.indexing()
104     ##############################
```

Figura 3.28: Fragmento de código para almacenar el token de usuario en Elasticsearch.

3.4.4. Query en Elasticsearch

Para ser más eficientes en las consultas a Elasticsearch, debemos obtener sólo aquellas partes del índice que nos interesen, como por ejemplo, las tareas de un usuario concreto y no obtener el índice completo y recorrerlo con un bucle.

Para hacer estas consultas eficientes la librería elasticsearch_dsl nos dá la clase Query:

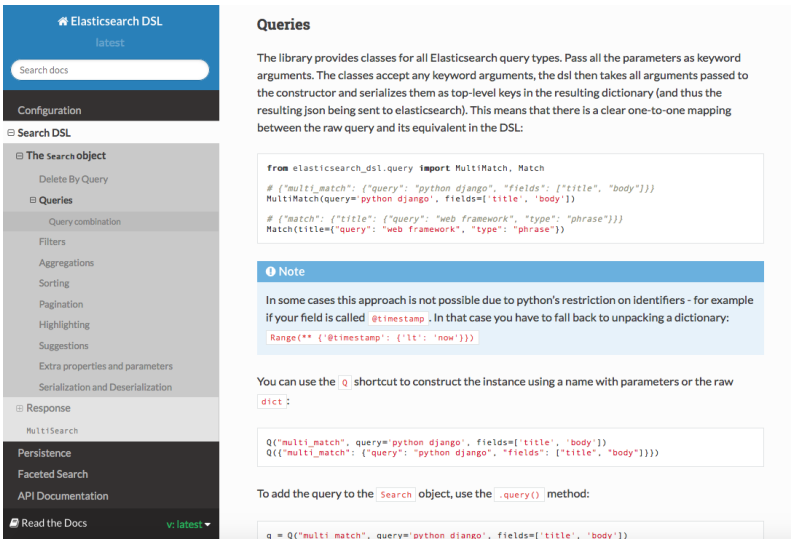


Figura 3.29: Página de la documentación relativa a las Query.

Analizando la documentación y algún ejemplo vemos que la forma de realizar una Query en Elasticsearch debe seguir los siguientes pasos:

1. **Paso 1:** creamos la Query, es decir, usamos la instrucción que nos permite dar incluir los valores de los parámetros determinados, por ejemplo, las tareas con el campo estado de valor False (pendientes de ejecutar).

2. **Paso 2:** abrimos una conexión con Elasticsearch.

3. **Paso 3:** ejecutamos una consulta Search, pero le aplicamos la función *.query(q)*, donde q es la Query creada en el paso 1.

4. **Paso 4:** preparamos la salida para poder trabajar con ella mediante la función *.execute()*.

```
131     q = Q('bool', must=[Q("match", creador=request.COOKIES['login']), Q("match", estado='true')])
132     es = Elasticsearch()
133     req = elasticsearch_dsl.Search(using=es, index='tareas').query(q)
134     resp = req.execute()
```

Figura 3.30: Fragmento de código con el proceso de creación de una Query.

3.4.5. Listado de tareas pendientes de ejecutar para el usuario

Gracias a la introducción de las Query para Elasticsearch, la consulta de tareas pendientes para un usuario determinado es prácticamente inmediata. Bastará con crear la Query en la que el campo *estado* debe ser False y el campo *creador* debe coincidir con el usuario que está en la aplicación en ese momento.

```
138
139 def lista_tareas_pendientes(request):
140     q = Q('bool', must=[Q("match", creador=request.COOKIES['login']), Q("match", estado='false')])
141     es = Elasticsearch()
142     req = elasticsearch_dsl.Search(using=es, index='tareas').query(q)
143     resp = req.execute()
144     salida = json.dumps(resp.to_dict(), indent=2)
145
146     return HttpResponse(salida, content_type='application/json')
```

Figura 3.31: Fragmento de código para la obtención de las tareas pendientes.

Hay que señalar que esta vista Django devuelve su salida en formato JSON, de ahí la existencia de la instrucción *json.dumps()* en la que convertimos la salida de la consulta al formato JSON.

3.4.6. Listado de tareas ya ejecutadas del usuario

Para obtener las tareas ya ejecutadas de un usuario seguiremos el mismo método que para las pendientes, pero con el campo *estado* con el valor True.

```
129
130 def lista_tareas_ejecutadas(request):
131     q = Q('bool', must=[Q("match", creador=request.COOKIES['login']), Q("match", estado='true')])
132     es = Elasticsearch()
133     req = elasticsearch_dsl.Search(using=es, index='tareas').query(q)
134     resp = req.execute()
135     salida = json.dumps(resp.to_dict(), indent=2)
136
137     return HttpResponse(salida, content_type='application/json')
```

Figura 3.32: Fragmento de código para la obtención de las tareas ejecutadas.

3.4.7. Conocer los repositorios que posee un usuario determinado

Otra funcionalidad interesante es la consulta de que repositorios tiene disponibles un usuario determinado de GitHub, para ello debemos acudir de nuevo a la API de GitHub y buscar la forma de obtener ese listado.

Para ello accedemos a la parte de repositorios en la API.

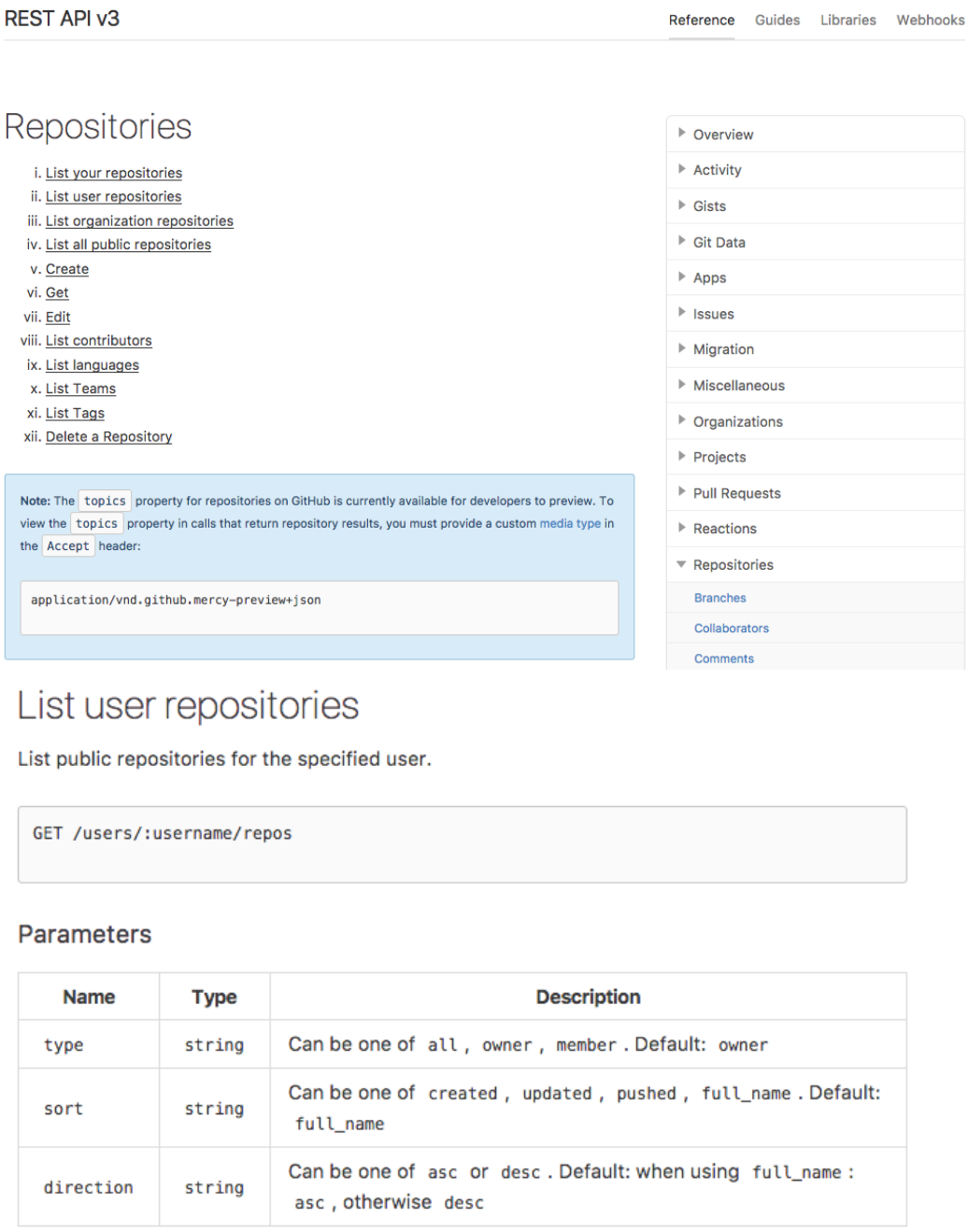


Figura 3.33: Fragmentos de la API de GitHub para obtener los respositorios de un usuario.

Aquí nos explican cómo obtener los repositorios de un usuario determinado, que consistirá simplemente en hacer un GET a la dirección */users/:username/repos*, donde :username será el

propietario del que solicita la información nuestro usuario.

Por lo tanto, para aplicar este proceso a Python seguiremos un proceso similar al de la petición del access_token. Mediante la librería Requests solicitamos con la función GET a la url anteriormente mencionada el listado de repositorios.

```
114
115 def lista_tareas_usuario(request):
116     usuario=request.POST.get('usuario')
117     url='https://api.github.com/users/'+usuario+'/repos'
118     res = requests.get(
119         url,
120         )
121     salidaaux=json.loads(res.content)
122     salida=json.dumps(salidaaux,indent=2)
123     for x in salidaaux:
124         print(x['name'])
125     return HttpResponse(salida, content_type='application/json')
```

Figura 3.34: Fragmento de código para la obtención de los repositorios de un usuario.

La líneas adicionales corresponden con:

- Mediante un formulario Django y HTML le solicitamos al cliente de la aplicación el nombre del usuario del que quiere conocer sus repositorios.
- Una vez que obtenemos el listado, se transforma a JSON para poder mostrarlo en la salida de la vista. En primer lugar usamos la función *.loads()* para convertir un flujo de datos en un objeto Python, y finalmente con *.dumps()* convertimos el objeto Python en JSON. El parámetro *indent*, añade tabulación a la salida para su mejor visualización.

3.4.8. Actualización del script de ejecución

Finalmente hay que modificar el script para la creación de los dashboards, y adaptarlo a los nuevos cambios en los modelos Django.

1. **Paso 1:** con la introducción de las Query para Elasticsearch, lo primero será solicitar sólo aquellas entradas en el índice pendientes de ejecutar. Para ello usaremos lo aprendido en el apartado de Query, solicitando las que tienen el campo *estado* a False.
2. **Paso 2:** recorremos la lista resultante de la consulta mediante Query, y en cada iteración debemos hacer otra consulta para obtener el access_token de quien pidió la tarea.
3. **Paso 3:** antes de empezar la ejecución de *p2o.py* y *kidash.py* debemos actualizar el valor del campo *inicioEjecucion* con la fecha del instante previo, a partir de un objeto *DocType*, usando la función *.get()*, obtenemos del índice *tareas* el nodo correspondiente a la petición de creación del dashboard del usuario y repositorio dados. Una vez que lo tenemos, mediante la función *.update()* actualizamos el valor con *datetime.now()*.
4. **Paso 4:** a continuación hay que ejecutar los scripts citados anteriormente, para su ejecución usaremos una nueva librería, Subprocess, cuya documentación[15] nos dará soporte para lanzar la ejecución en hilos diferentes y de esta manera poder controlar cuando terminan, y actualizar el resto de campos del modelo. Lanzamos la ejecución de los scripts

con la función *subprocess.Popen()* y posteriormente con la instrucción *wait()* esperaremos la finalización de ambos, para terminar actualizando los valores de los campos *estado* y *finEjecucion* del mismo modo que en el Paso 3.

```
16
17 es = Elasticsearch()
18 q = Q("match", estado='false')
19 requ = elasticsearch_dsl.Search(using=es, index='tareas').query(q)
20 resp = requ.execute()
21
22 for commit in resp:
23     qitem = Q("match", usuario=commit.creador)
24     reqitem = elasticsearch_dsl.Search(using=es, index='usuariosapp').query(qitem)
25     respitem = reqitem.execute()
26     for x in respitem:
27         print(x.token)
28
29         print("Empezando a ejecutar...")
30         req = tareasIndex.get(id=commit.usuario+"-"+commit.repositorio, using=es, index='tareas')
31         req.update(using=es, inicioEjecucion=datetime.now())
32         repo_url = 'https://github.com/'+commit.usuario+'/'+commit.repositorio+'.git'
33         cmd = "p2o.py --enrich --index git_raw --index-enrich git \-e http://localhost:9200 --no_inc --debug \git "+repo_url+""
34         cmd = shlex.split(cmd)
35         p1 = subprocess.Popen(cmd)
36 # #
37         cmd3 = "kidash.py --elastic_url-enrich http://locahost:9200 \--import /tmp/git-dashboard.json"
38         cmd3 = shlex.split(cmd3)
39         p2 = subprocess.Popen(cmd3)
40         p1.wait()
41         p2.wait()
42
43         req.update(using=es, estado=True, finEjecucion=datetime.now())
```

Figura 3.35: Código Python del script de ejecución.

3.5. Iteración 5. Interfaz con Bootstrap, estadísticas y Kibana

3.5.1. Objetivos

El objetivo principal de esta iteración será la de dotar a la aplicación de una interfaz web que permita al usuario llegar al objetivo final de obtener un dashboard por medio de una interfaz agradable e intuitiva. Además de esto añadiremos al sistema algunos campos de estadísticas temporales para informarle, por ejemplo, del tiempo que ha tardado en ejecutarse una tarea determinada. Por último será necesario que la aplicación proporcione enlaces para que el usuario pueda acceder a los dashboard que se han creado, tantos para Git por un lado, como para GitHub por otro.

3.5.2. Interfaz web mediante Bootstrap

Como ya se ha comentado en las iteraciones previas, el diseño ha tenido una interfaz web, con el único objetivo de probar el resultado de la funcionalidad creada en cada iteración. Esta interfaz es exclusivamente para probar la aplicación no cumpliendo el requisito previo de abstraer al usuario del proceso de creación y las tecnologías participantes.

Por lo tanto, una vez finalizado el diseño en cuanto a funcionalidad, se decide implementar una interfaz web que cumpla con el requisito establecido. Para ello se utiliza Bootstrap, por su facilidad para crear una interfaz simple pero intuitiva en un periodo razonable de tiempo.

3.5.2.1. Instalación y configuración

Para comenzar el diseño es necesaria la instalación de Bootstrap, para ello accedemos a su página web *getbootstrap.com*, y a continuación a la sección *Getting started*[16].

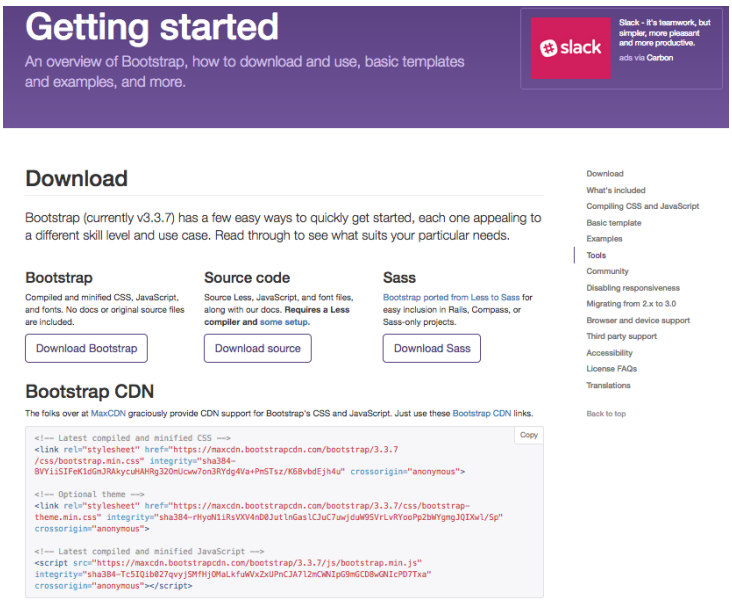


Figura 3.36: Página de descarga de Bootstrap.

La página ofrece tres tipos de descargas, la primera es una versión reducida, con los archivos justos para su ejecución, sin incluir por ejemplo las plantillas predeterminadas o el código fuente, es decir, solo contiene lo mínimo para la ejecución. La segunda es la versión completa del sistema con el código fuente y las plantillas, pero para su manejo es necesario compilarlo y configurarlo. La última es una versión para integrar con algunos frameworks para el desarrollo de aplicaciones, como Rails o Compass, entre los que no está Django. Además también se ofrece *Bootstrap CDN*, en la que la empresa CDN aloja una copia de los ficheros necesarios para Bootstrap, que será necesario solicitar vía HTTP en la zona de cabeceras.

Por lo tanto en nuestro caso usaremos versión CDN, ya que para nuestra complejidad será suficiente. En el caso de que Django estuviese entre los frameworks en los que puede integrarse Bootstrap nos decantaríamos por esta opción.

Por lo tanto no será necesario realizar configuración o instalación alguna, así que buscaremos una plantilla que nos sirva de base para nuestra interfaz.

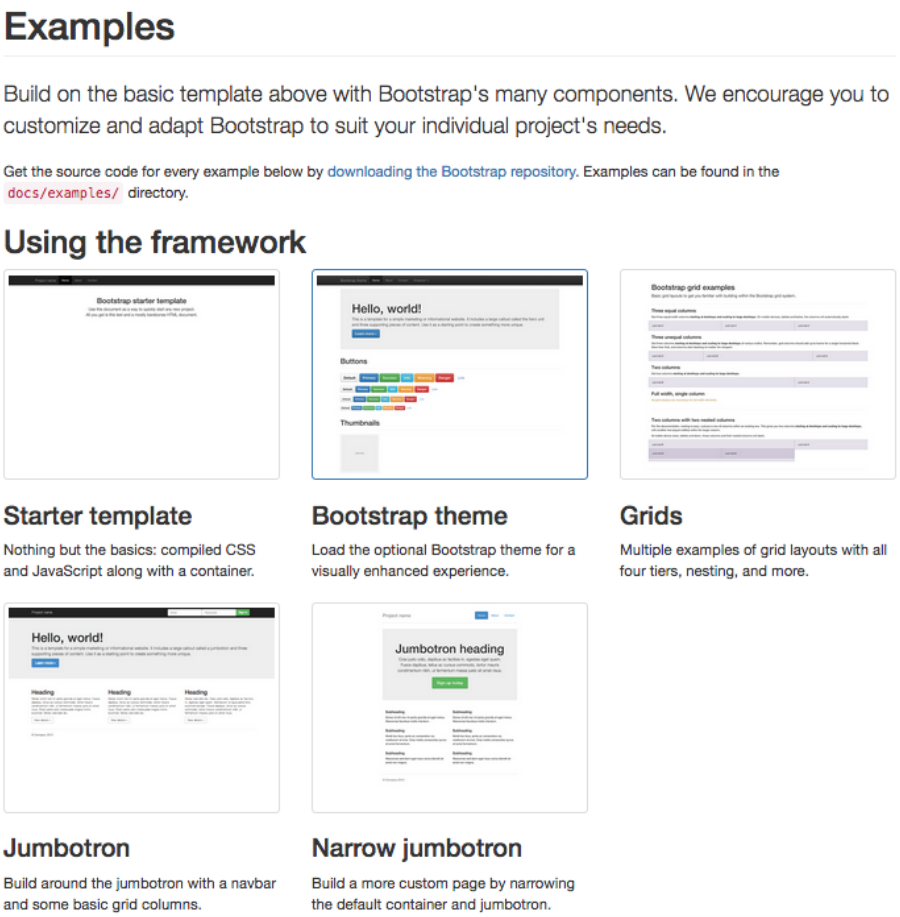


Figura 3.37: Sección de plantillas predefinidas en la página de Bootstrap.

3.5.2.2. Elección de plantilla predefinida

Finalmente nos decantaremos por la plantilla que se llama *Jumbotron*, que básicamente ofrece una apariencia donde los elementos más importantes para nuestro diseño son:

- **Navbar:** barra superior de navegación, fija en la parte de arriba, que proporcionará alojamiento para los botones de navegación entre páginas.
- **Jumbotron:** se trata de un elemento que se utiliza para mostrar contenidos de forma destacada. Consiste en un elemento *div* con estilo CSS predefinido por Bootstrap para darle relevancia.
- **Row:** elemento *div* que gestiona filas, que pueden colocarse de forma consecutiva. A su vez en su interior pueden dividirse en hasta doce columnas, permitiéndose la gestión de estas columnas mediante cualquiera de las versiones de *class="md"*, en divisiones desde dos hasta doce.

3.5.2.3. Definición de plantilla base

Necesitamos entonces definir una plantilla base, que se repita en todas las páginas de la aplicación, sobre la que cada una de ellas mostrará la parte de información que le corresponde.

Los elementos que compondrán esta página base serán los siguientes:

1. **Hoja de estilos:** aunque no forma parte de la zona visible de la plantilla base, será la hoja donde estén recogidas todas las reglas CSS que afecten a la totalidad de la interfaz.

```
/* Move down content because we have a fixed navbar that is 50px tall */
.body {
  padding-top: 50px;
  padding-bottom: 20px;
}

.navbar-header {
    width: 100%;
    text-align: center;
}

.navbar-header > li {
    float: none;
    display: inline-block;
}

.ppal{
    background-image: url(../imagenes/index.jpg);
    background-size: cover;
    width: 100vw;
    height: 100vh;
}
```

Figura 3.38: Fragmento de la hoja CSS correspondiente a nuestra interfaz.

2. **Fondo general de la interfaz:** la idea es mostrar una foto que ocupe la parte visible de la página, y que se repita en todas las secciones de la interfaz. Esta propiedad debe ser definida en la hoja de estilo comentada anteriormente, definida para la *section class="ppal"*, con las propiedades adecuadas para que quede centrada y ocupando el total de la zona visible de la página.
3. **Barra de navegación superior:** basándonos en la barra de navegación superior de la plantilla predefinida, eliminaremos algunos de los elementos que aparecen en ella y añadiremos tantos botones como funcionalidad tenga el sistema, para que de esta forma el usuario pueda moverse de una página a otra con facilidad. Los botones que se van a utilizar

en la barra, y prácticamente en el resto de localizaciones de la interfaz, también están ya predefinidos en Bootstrap, *class="btn btn-primary btn-lg"*, unos botones están alineados a la izquierda y otros a la derecha, para separar su funcionalidad.

```
40    <section class="ppal">
41        <div class="jumbotron">
42            {% block content %}
43            <form class="formulario" action="{% url 'utareas:tadd' %}" method="post">
44                {% csrf_token %}
45                {{ form.as_p }}
46                <input class="btn btn-primary btn-lg" type="submit" value="Añadir tarea" />
47            </form>
48            {% endblock %}
49        </div>
50    </section>
```

Figura 3.39: Código HTML para insertar la barra superior en la plantilla base.

El resultado del código anterior mostrará la siguiente barra en el navegador.

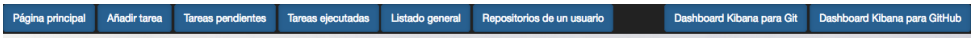


Figura 3.40: Barra de navegación perteneciente a la plantilla base.

3.5.2.4. Páginas principales de la interfaz

A continuación se describirán las páginas que incluyen una implementación diferente de las demás ya que, por ejemplo, hay dos que muestran formularios, actuando de manera muy similar, o varias que muestran tablas con listados de tareas, basadas en la misma forma de visualización. Todas las páginas que a continuación se describen, toman como plantilla base la descrita en el apartado anterior.

- **Página de inicio:** en la página de inicio se ofrecen al usuario dos cosas, el botón para comenzar la autenticación a través de GitHub, y un par de botones que enlazan con la parte inferior de la página donde podrán encontrar información sobre las tecnologías que intervienen en el proyecto, así como información acerca del autor.

  La forma de diseñar la página es sencilla. Un elemento *class="jumbotron"* contendrá, para que quede resaltada, la parte para que el usuario pueda empezar el proceso de autenticación a través de GitHub, con unas líneas de texto y un botón para comenzar el proceso. Justo debajo, perteneciendo a la misma sección un elemento *class=row"* alojará unos botones para enlazar con la parte no visible de la página donde estará la información de la app y del creador.

  En otra sección, cada parte dentro de un elemento *class=row"*, tendremos información sobre la aplicación e información sobre el creador de la misma.

- **Página principal:** esta será la encargada de dar la bienvenida al usuario una vez que ha sido autenticado por GitHub, poniendo a su disposición toda la funcionalidad que ofrece la aplicación. Partiendo de la plantilla base, necesitaremos mostrar de forma ordenada esta funcionalidad. Para ello crearemos tres elementos *class=row"*, colocados uno debajo de

otro. Vamos a mostrar siete enlaces, por lo que los repartiremos de la siguiente forma: tres elementos en la primera fila, dos en la siguiente y dos en la última.

Para dividir cada elemento *class=row"* se usará mediante la clase *class=çol-md-(nº)"*, en donde (nº) se sustituirá por la cantidad de columnas que queremos ocupar de la división de doce que tiene por definición el elemento. Por lo tanto, para la primera fila tendremos en su interior, tres elementos del tipo *class=çol-md-4"*, y tanto para la segunda como para la tercera tendremos dos *class=çol-md-6"*, es decir, se tomarán seis unidades de las doce, teniendo como resultado dos divisiones.

A continuación se muestra un fragmento de código, de la construcción del panel, concretamente el correspondiente a la primera fila del panel, que a su vez se divide en tres columnas.

```
<div class="row">
  <div class="col-md-4">
    <h2>AÑADIR UNA TAREA AL SISTEMA</h2>
    <p><strong>Añada una tarea al sistema para que éste lo añada a su cola de ejecución.</strong></p>
    <p><a class="btn btn-primary btn-lg active" href="{% url 'utareas:tadd'  %}" role="button">Añadir tarea &raquo;</a></p>
  </div>
  <div class="col-md-4">
    <h2>LISTADO DE TAREAS PENDIENTES DEL USUARIO</h2>
    <p><strong>Mostrará el listado de tareas pendientes que el usuario añadió al sistema.</strong></p>
    <p><a class="btn btn-primary btn-lg active" href="{% url 'utareas:tlistpend'  %}" role="button">Tareas pendientes &raquo;</a></p>
 </div>
  <div class="col-md-4">
    <h2>LISTADO DE TAREAS EJECUTADAS DEL USUARIO</h2>
    <p><strong>Mostrará el listado de tareas ejecutadas que el usuario añadió al sistema.</strong></p>
    <p><a class="btn btn-primary btn-lg active" href="{% url 'utareas:tlistexec'  %}" role="button">Tareas ejecutadas &raquo;</a></p>
  </div>
</div>
```

Figura 3.41: Código HTML para crear la primera fila, que se divide en tres columnas.

Éste será el resultado de la ejecución del código anterior, junto con el del resto de filas que conforman el panel.

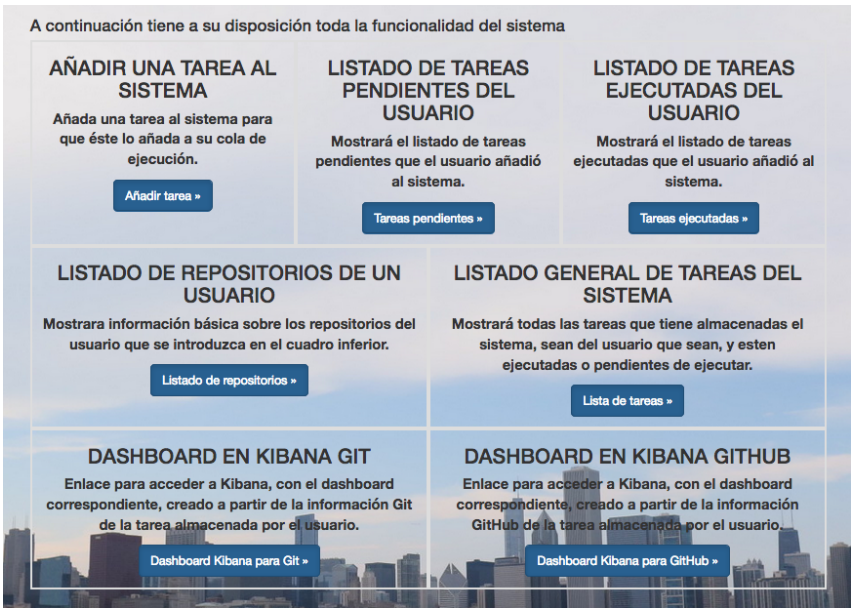


Figura 3.42: Panel que incluye la funcionalidad de la aplicación.

- **Formularios y listas de tareas:** el resto de las páginas que sirve la interfaz se dividen en dos tipos, unos para la entrada de datos mediante formularios y otros que mediante tablas predefinidas en Bootstrap muestran los contenidos de los índices de Elasticsearch.

  El primero de los tipos, los formularios, será sencillo de implementar. Tan sólo hay que insertar el código HTML correspondiente a un formulario habitual.

```
40    <section class="ppal">
41        <div class="jumbotron">
42            {% block content %}
43            <form class="formulario" action="{% url 'utareas:tadd' %}" method="post">
44                {% csrf_token %}
45                {{ form.as_p }}
46                <input class="btn btn-primary btn-lg" type="submit" value="Añadir tarea" />
47            </form>
48            {% endblock %}
49        </div>
50    </section>
```

Figura 3.43: Código HTML para insertar un formulario en la plantilla base.

El segundo tipo de página que se muestra es el correspondiente a la información almacenada en Elasticsearch. Tras analizar varias opciones, se decide usar las tablas predefinidas en Bootstrap para mostrar la información de forma ordenada. Dentro de las tablas que están predefinidas en Bootstrap seleccionamos una combinación de ellas, el tipo *class="table-hover"* que resalta la línea al pasar sobre ella con el ratón y así facilitar la lectura de la misma y el tipo *class="table-bordered"* que colorea los bordes de la misma.

```
40    <section class="ppal">
41        <div class="jumbotron">
42            {% block content %}
43            <form class="formulario" action="{% url 'utareas:tadd' %}" method="post">
44                {% csrf_token %}
45                {{ form.as_p }}
46                <input class="btn btn-primary btn-lg" type="submit" value="Añadir tarea" />
47            </form>
48            {% endblock %}
49        </div>
50    </section>
```

Figura 3.44: Código HTML para insertar una tabla con los datos de Elasticsearch.

El elemento *table class="table table-hover table-bordered"*, se divide en dos subelementos internos *thead* y *tbody*, el primero contedrá la información sobre el encabezamiento de la tabla, es decir, el nombre de los campos que se van a mostrar. El segundo de los elementos llevará la información que contiene Elasticsearch, es decir, los valores con los que se rellenará la tabla.

A su vez en el interior de ambos elementos, hay dos nuevas etiquetas, *tr*, en cuyo interior estará lo que se inserta en una fila y *th* para indicar el valor de cada posición de la tabla.

Antes de terminar hay que señalar que los parámetros englobados entre llaves pertenecen al objeto que se le pasa a la plantilla desde Django, sirviendo estas llaves para acceder a su contenido.

3.5.3. Generación de estadísticas y visualización

Una vez que tenemos una visualización más intuitiva para el usuario, vemos la necesidad de que el sistema genere algún dato propio, como pueden ser las estadísticas, principalmente, de las tareas ejecutadas, mostrándolas con las visualizaciones creadas anteriormente.

Para que el usuario tenga acceso a dicha funcionalidad, se añade un botón dentro de las tablas que representan las tareas pendientes de ejecutar y las que ya están ejecutadas, además de crear una plantilla para mostrar las estadísticas de cada una de ellas. Los botones que se añaden a las tablas funcionan como un formulario pero sin solicitar datos al usuario, sino que tomará los campos de propietario y de nombre del repositorio de la fila a la que corresponde el botón, para poder hacer la consulta a Elasticsearch en *views.py*.

Para crear la funcionalidad que genera las estadísticas, tendremos que crear dos entradas en el archivo *views.py* de Django, una para generar las correspondientes a las tareas ejecutadas y otra para las tareas pendientes de ejecutar.

- **Tareas pendientes de ejecutar:** además de generar las estadísticas, que en el caso de las tareas pendientes, se reduce al cálculo de la antigüedad de la tarea en el sistema, es decir, cuando tiempo ha pasado desde que se añadió, al obtener los datos de la tarea correspondiente, se manipularán para que la visualización de los campos relacionados con el tiempo, sea agradable para el usuario.

  Para generar el campo de antigüedad bastará con restar el valor del campo *fechaRegistro*, con el momento actual que obtendremos mediante *datetime.now()*.

```
179
180  def estadisticasPend(request):
181      lista=[]
182      if request.POST.get('usuario')==None:
183          lista=request.COOKIES['lista']
184      else:
185          q = Q('bool', must=[Q("match", usuario=request.POST.get('usuario')), Q("match", repositorio=request.POST.get('repositorio'))])
186          es = Elasticsearch()
187          req = elasticsearch_dsl.Search(using=es, index='tareas').query(q)
188          resp = req.execute()
189          for x in resp:
190              nodo={}
191              nodo['propietario']=x.usuario
192              nodo['repositorio']=x.repositorio
193              nodo['fechaRegistro']=x.fechaRegistro
194              nodo['inicioEjecucion']=x.inicioEjecucion
195              nodo['finEjecucion']=x.finEjecucion
196              nodo['tRespuesta'] = str(datetime.now() - datetime.strptime(x.fechaRegistro, '%Y-%m-%dT%H:%M:%S.%f'))
197              lista.append(nodo)
198
199      respuesta=render(request, 'statsPend.html', {'object_list': lista})
200      respuesta.set_cookie('lista',lista)
201      return respuesta
```

Figura 3.45: Método para generar los datos que se mostrarán en las estadísticas pendientes.

- **Tareas pendientes de ejecutar:** además de generar las estadísticas, que en el caso de las tareas ejecutadas, además de obtener la antigüedad, obtendremos el tiempo de ejecución efectiva y el tiempo total de respuesta. De igual forma que en las tareas pendientes de ejecutar, al obtener los datos de la tarea correspondiente, se manipularán para que la visualización de los campos relacionados con el tiempo, sea agradable para el usuario.

  El campo antigüedad se generará de igual forma que en el caso de las tareas pendientes.

  El campo de tiempo de ejecución efectiva calculará el tiempo transcurrido desde que se empieza a ejecutar la tarea hasta que se termina, es decir, el tiempo que realmente el sistema dedica a la ejecución. Para calcularlo restaremos el valor del campo *finEjecucion* del valor del campo *inicioEjecucion*.

El campo tiempo total de respuesta calculará el tiempo que transcurre desde que el usuario añade una tarea al sistema, hasta que pueden consultar su resultado, es decir, el tiempo de respuesta, que se compone del tiempo que pasa hasta que se inicia su ejecución y el tiempo de ejecución efectiva. Para calcularlo restaremos el valor del campo *finEjecucion* del valor del campo *fechaRegistro*.

```
def estadisticasExec(request):
    lista=[]
    if request.POST.get('usuario')==None:
        lista=request.COOKIES['lista']
    else:
        q = Q('bool', must=[Q("match", usuario=request.POST.get('usuario')), Q("match", repositorio=request.POST.get('repositorio'))])
        es = Elasticsearch()
        req = elasticsearch_dsl.Search(using=es, index='tareas').query(q)
        resp = req.execute()
        for x in resp:
            nodo={}
            nodo['propietario']=x.usuario
            nodo['repositorio']=x.repositorio
            aux=datetime.strptime(x.fechaRegistro, '%Y-%m-%dT%H:%M:%S.%f')
            aux=aux.strftime('Creado a las %Hh %Mmin %Sseg del %d-%m-%Y')
            nodo['fechaRegistro']=aux
            aux=datetime.strptime(x.inicioEjecucion, '%Y-%m-%dT%H:%M:%S.%f')
            aux=aux.strftime('Iniciado a las %Hh %Mmin %Sseg del %d-%m-%Y')
            nodo['inicioEjecucion']=aux
            aux=datetime.strptime(x.finEjecucion, '%Y-%m-%dT%H:%M:%S.%f')
            aux=aux.strftime('Finalizado a las %Hh %Mmin %Sseg del %d-%m-%Y')
            nodo['finEjecucion']=aux
            nodo['tEjecucion'] = str(datetime.strptime(x.finEjecucion, '%Y-%m-%dT%H:%M:%S.%f') - datetime.strptime(x.inicioEjecucion, '%Y-%m-%dT%H:%M:%S.%f'))
            nodo['tProceso'] = str(datetime.strptime(x.finEjecucion, '%Y-%m-%dT%H:%M:%S.%f') - datetime.strptime(x.fechaRegistro, '%Y-%m-%dT%H:%M:%S.%f'))
            nodo['tRespuesta'] = str(datetime.now() - datetime.strptime(x.fechaRegistro, '%Y-%m-%dT%H:%M:%S.%f'))
            lista.append(nodo)

    respuesta=render(request, 'statsExec.html', {'object_list': lista})
    respuesta.set_cookie('lista',lista)
    return respuesta
```

Figura 3.46: Método para generar los datos que se mostrarán en las estadísticas ejecutadas.

3.5.4. Enlace del sistema con Kibana

Como último paso de esta iteración, es necesario que el usuario pueda acceder a Kibana para ver el resultado del procesamiento del sistema. Para ello, simplemente tenemos que añadir un par de botones, tanto en la página principal como en la barra de navegación. Los botones deberán llevar al usuario directamente al dashboard correspondiente, según haga click en el botón para ir a Git o para ir a GitHub.

Los botones incluidos en la barra de navegación, tienen la misma configuración que cualquier otro botón de la interfaz, pero éstos están dentro del elemento *li class="pull-right"* para que queden alineados a la derecha y de ésta forma separarlos de los correspondientes a la navegación.

```
        <li class="pull-right">
            <a href="http://localhost:5601/app/kibana#/dashboard/Git" target="_blank" class="btn btn-primary btn-lg active" role="button">Dashboard Kibana para Git</a>
            <a href="http://localhost:5601/app/kibana#/dashboard/GitHub" target="_blank" class="btn btn-primary btn-lg active" role="button">Dashboard Kibana para GitHub</a>
        </li>
```

Figura 3.47: Código HTML correspondiente a los botones situados en la barra de navegación.

Los botones del panel principal van incluidos dentro de un elemento *class=row"*, que a su vez ha sido dividido en dos columnas, una para Git y otra para GitHub. Tanto en estos botones como en los de la barra de navegación, se ha añadido el parámetro *target="_blank"* al botón para que al pinchar sobre él, la nueva ventana de abra en otra pestaña.

```
<div class="row">
  <div class="col-md-6">
  <h2>DASHBOARD EN KIBANA GIT</h2>
    <p><strong>Enlace para acceder a Kibana, con el dashboard correspondiente, creado a partir de la información Git de la tarea almacenada por el usuario.</strong></p>
    <p><a class="btn btn-primary btn-lg active 2" href="http://localhost:5601/app/kibana#/dashboard/Git" target="_blank" role="button">Dashboard Kibana para Git &raquo;</a></p
  </div>
  <div class="col-md-6">
    <h2>DASHBOARD EN KIBANA GITHUB</h2>
    <p><strong>Enlace para acceder a Kibana, con el dashboard correspondiente, creado a partir de la información GitHub de la tarea almacenada por el usuario.</strong></p>
    <p><a class="btn btn-primary btn-lg active 2" href="http://localhost:5601/app/kibana#/dashboard/GitHub" target="_blank" role="button">Dashboard Kibana para GitHub &raquo;<
  </div>
</div>
```

Figura 3.48: Código HTML correspondiente a los botones situados en el panel principal.

Capítulo 4

Descripción del producto final

4.1. Descripción de la aplicación: versión con interfaz web

Aunque la aplicación también ofrece un servicio API, para poder usar la funcionalidad implementada, y no esperar que alguien la integre mediante la API, se ha construido una interfaz HTML, basada en Bootstrap, que permita representar una experiencia de usuario con la aplicación.

4.1.1. Entrada a la aplicación

La pantalla inicial de la aplicación es una pantalla de bienvenida, es decir, la única funcionalidad real que ofrece es el enlace para autenticarse a través de GitHub, el resto de lo que muestra la pantalla inicial es información acerca de, el mensaje de presentación y bienvenida,información acerca de los pasos para avanzar en el uso de la aplicación, información del desarrollador e información acerca de las tecnologías utilizadas.

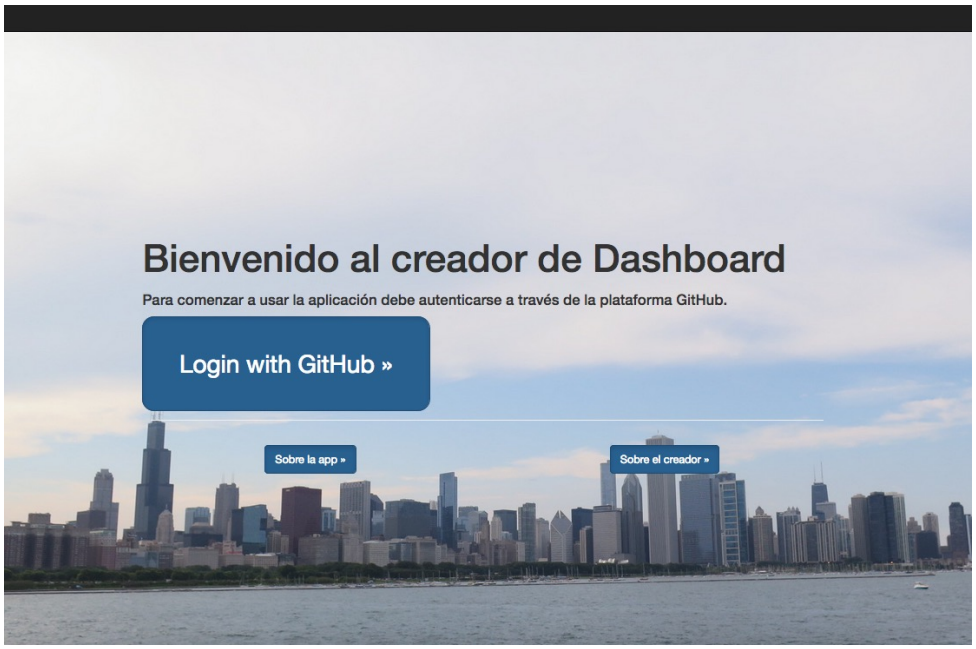


Figura 4.1: Pantalla de inicio de la interfaz web.

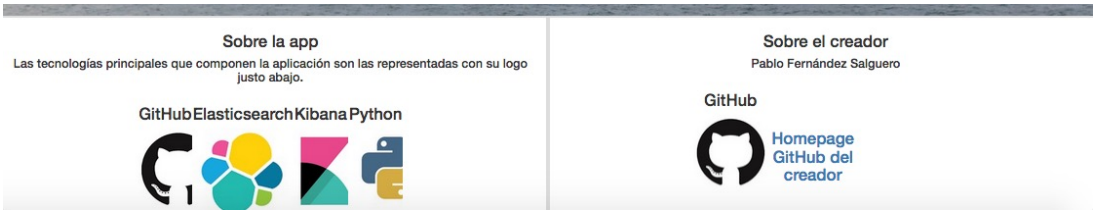


Figura 4.2: Zona de información de la pantalla de inicio de la interfaz web.

4.1.2. Proceso de autenticación a través de GitHub

Una vez que el usuario ha visualizado la página de inicio, como ya hemos comentado la única opción para usar la aplicación es pinchar sobre el enlace para la autenticación a través de GitHub. Una vez que el cliente pincha sobre dicho enlace el proceso será el siguiente:

1. **Pantalla de registro en GitHub:** En primer lugar, GitHub mostrará la pantalla para el registro del usuario de nuestra aplicación en nuestro sistema, en ella solicita su usuario y su contraseña, explicando la necesidad de registrarse en GitHub para pasar al proceso de autorización de nuestra aplicación.

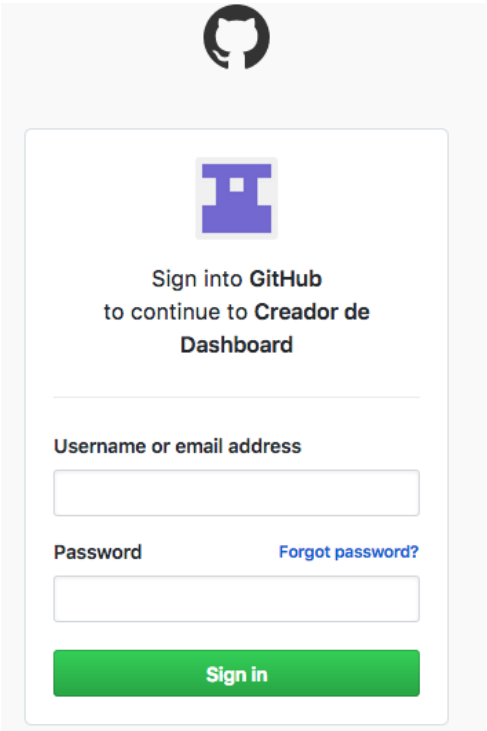


Figura 4.3: Página de registro de usuario en GitHub.

2. **Pantalla de autorización de la aplicación en GitHub:** El siguiente paso será que el usuario de autorización a nuestra aplicación para poder acceder, por ejemplo, a su token de usuario. Para ello GitHub muestra una pantalla con un primer recuadro mostrando el nombre de la aplicación y el nombre de su propietario, así como el tipo de acceso que se

le va a permitir. En ese mismo recuadro informa de la página a la que será redireccionado el usuario y el botón para autorizar. Además de esto existe justo debajo un cuadro con alguna información estadística sobre la aplicación, como el número de usuarios, la fecha de creación y si es de creación o operado por GitHub.

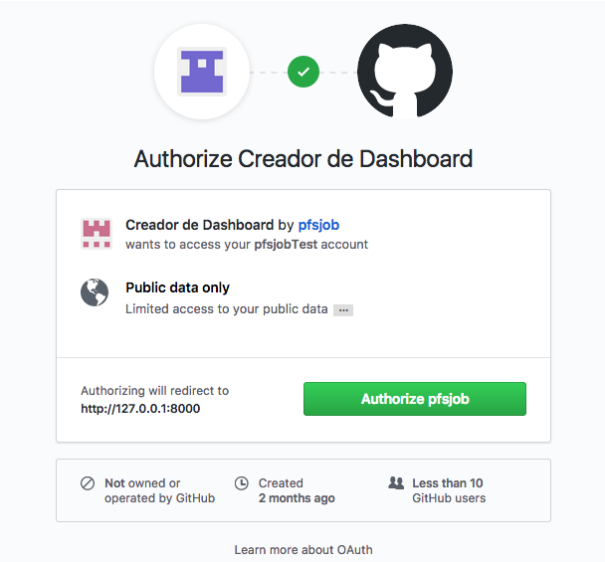


Figura 4.4: Página de autorización de la aplicación en GitHub.

4.1.3. Pantalla principal de la aplicación

La pantalla principal de la aplicación es la encargada de mostrar al usuario la funcionalidad que tiene a su servicio, es decir, todas las opciones que puede ejecutar una vez que ya se ha autenticado a través de GitHub. La pantalla principal es el centro de la aplicación,

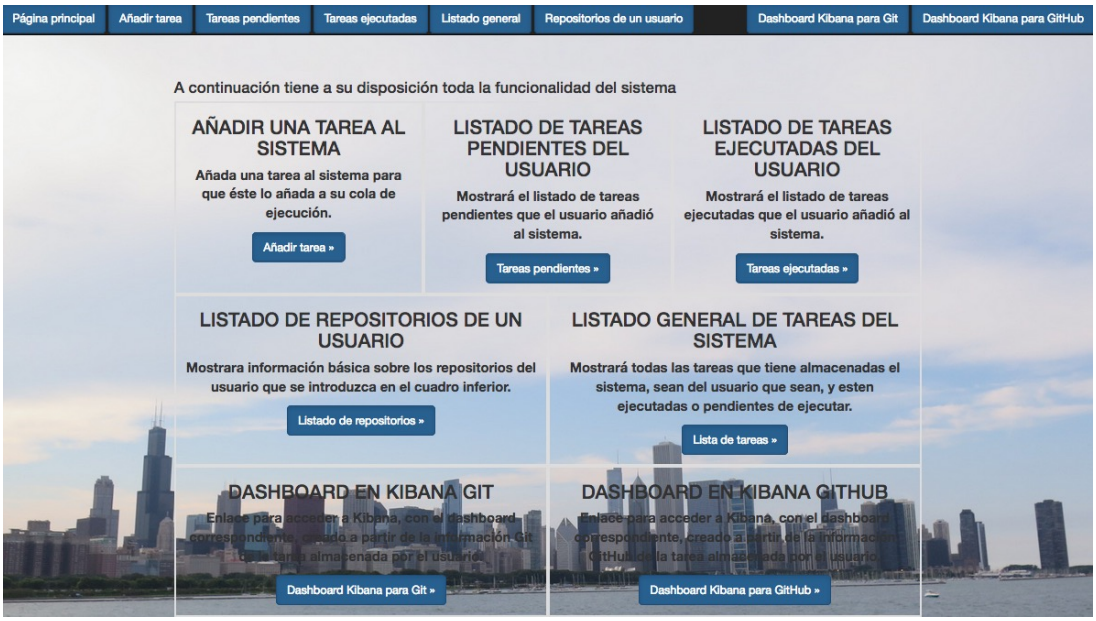


Figura 4.5: Pantalla principal de la interfaz web de la aplicación.

4.1.4. Funcionalidad de la aplicación

La opciones de funcionalidad de la aplicación, como ya hemos comentado, está recogida en la pantalla principal. Tanto la apariencia de ésta, como la apariencia del resto de las pantallas de la aplicación serán similares. La franja de color negro se mantendrá en todas las páginas de la aplicación, que además incluye botones para acceder a cualquier página de la aplicación. El fondo de la página será la misma en todas las secciones.

4.1.4.1. Añadir tarea

Cuando el usuario solicita añadir una tarea, el sistema servirá una página web con un formulario para que éste pueda rellenar los campos necesarios y crear una instancia en Elasticsearch acerca del repositorio del que quiere crear un dashboard.

La pantalla muestra los campos que el usuario puede solicitar: el nombre del propietario del repositorio y el nombre del repositorio. Para enviar la información la página muestra un botón, que al pulsar, llevará al cliente a la página principal, ejecutando el proceso de almacenar la información de forma transparente al usuario.

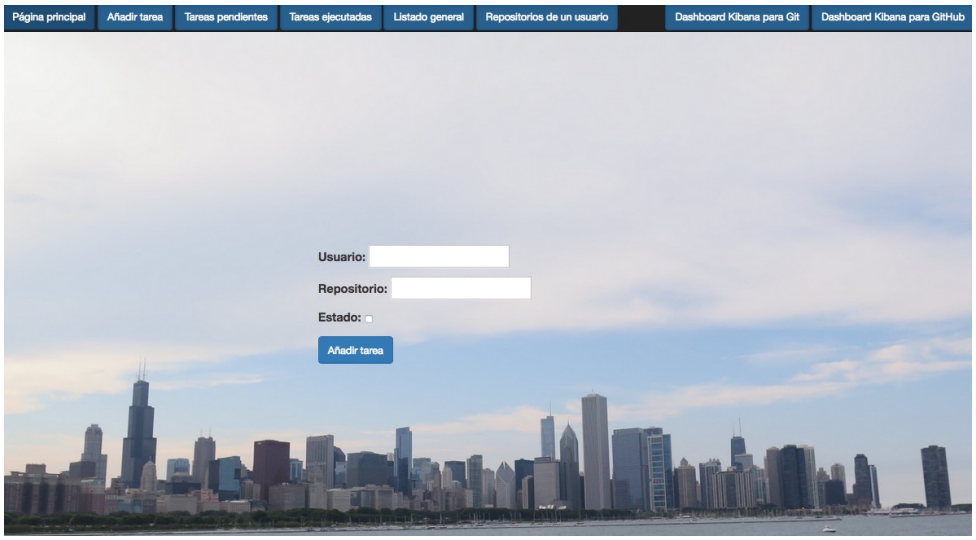


Figura 4.6: Pantalla para añadir una tarea al sistema.

4.1.4.2. Mostrar lista de tareas totales de la aplicación

Si el usuario lo desea, puede tener acceso al listado total de tareas que tiene el sistema almacenadas, tanto si están ejecutadas como si no, y sean del usuario que sean. De esta forma se tiene un acceso al histórico del sistema, de carácter informativo.

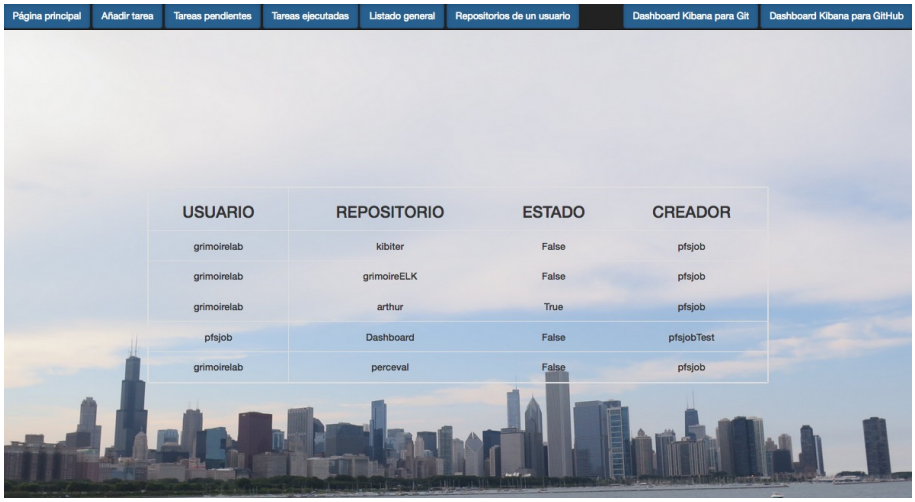


Figura 4.7: Pantalla donde se muestra el listado general de tareas de la aplicación.

4.1.4.3. Mostrar tareas pendientes de ejecutar del usuario

En todo momento el usuario puede conocer la relación de tareas que en algún momento ha añadido al sistema pero que están esperando para ser ejecutadas. Para ello el sistema mostrará una pantalla que contiene una tabla con los principales campos de cada tarea.

En la última casilla de cada fila tendremos un botón para ver más al detalle los diferentes campos de los que consta la tarea, así como estadísticas. En el caso de las fechas, antes de mostrarlas, se realiza un proceso previo en Python para que sean lo más fáciles de entender. No se muestra el campo de quien ha creado la tarea, ya que son las tareas del usuario que está activo en el sistema, es decir, se sobreentiende que las tareas de las que se muestra el detalle son de este usuario.

Para este caso, como se trata de tareas que están pendientes de ejecutar, como estadística adicional a los campos se ofrece el tiempo que ha transcurrido desde que la tarea fue añadida al sistema, es decir, la antigüedad de la misma.

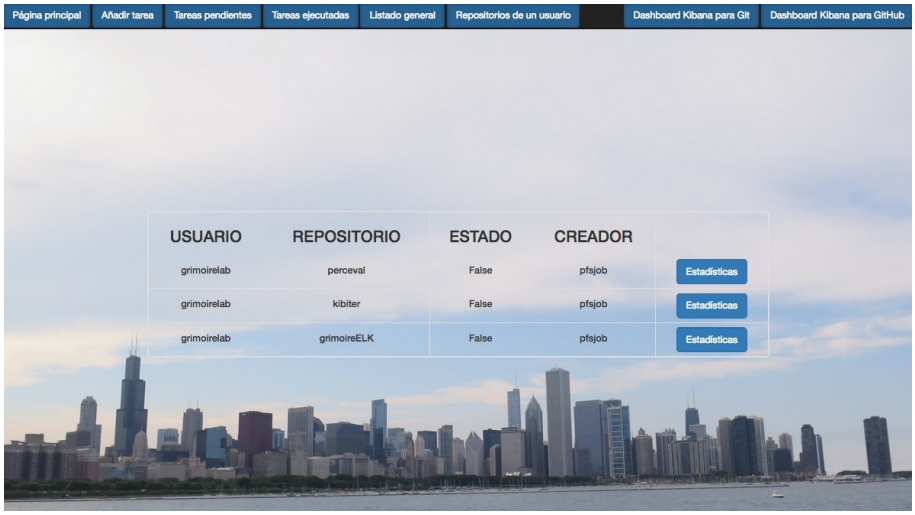


Figura 4.8: Pantalla con el listado de tareas pendientes de ejecutar del usuario.

Figura 4.9: Pantalla con las estadísticas de una de las tareas pendientes de ejecutar.

4.1.4.4. Mostrar tareas ejecutadas del usuario

En todo momento el usuario puede conocer la relación de tareas que en algún momento ha añadido al sistema y que ya han sido ejecutadas. Para ello el sistema mostrará una pantalla que contiene una tabla con los principales campos de cada tarea.

En la última casilla de cada fila tendremos un botón para ver más al detalle los diferentes campos de los que consta la tarea, así como estadísticas. En el caso de las fechas, antes de mostrarlas, se realiza un proceso previo en Python para que sean lo más fáciles de entender.No se muestra el campo de quien ha creado la tarea, ya que son las tareas del usuario que está activo en el sistema, es decir, se sobreentiende que las tareas de las que se muestra el detalle son de este usuario.

Para este caso, al tratarse de tareas ya ejecutadas, como estadística adicional a los campos se ofrece:

- **Tiempo efectivo de ejecución:** Se trata del tiempo exacto que el sistema dedica a una petición del usuario, es decir, desde que empieza a ejecutar la creación del dashboard, hasta que termina.

- **Tiempo de procesamiento:** Es el tiempo total que transcurre desde que el cliente añade una tarea al sistema, hasta que puede acceder a su dashboard.

- **Antigüedad de la tarea:** Muestra el tiempo que la tarea lleva en el sistema, respecto del momento actual, es decir su antigüedad en el sistema.

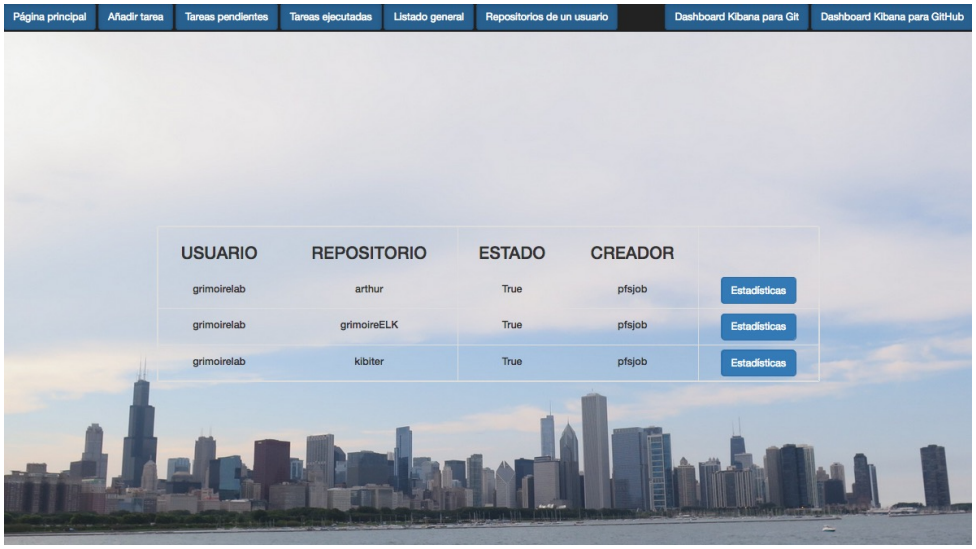


Figura 4.10: Pantalla con el listado de tareas ejecutadas del usuario.

Figura 4.11: Pantalla con las estadísticas de una de las tareas ejecutadas.

4.1.4.5. Mostrar los repositorios de un usuario determinado de GitHub

La aplicación ofrece al usuario la posibilidad de conocer los repositorios de un usuario determinado, es decir, partiendo del nombre de un usuario conocido se mostrará un listado de los repositorios que pertenecen a éste.

El proceso para solicitar el listado de nombres de repositorios, comienza con un página de formulario solicitando el nombre del usuario de GitHub del que queremos conocer sus repositorio. Y una vez que enviamos esta información tendremos una página que mostrará el resultado. Para la visualización, el sistema mostrará una tabla resumen, ya que la información que proporciona la API de GitHub por cada repositorio es muy extensa, en esta tabla resumen estarán los siguientes campos:

- **Nombre del propietario:** es el nombre del propietario del repositorio, es decir, el nombre que nuestro usuario proporcionó al sistema para hacer la búsqueda mediante la API de GitHub.
- **Nombre del repositorio:** mostrará el nombre que el propietario ha dado a su repositorio en GitHub.
- **Nombre completo del repositorio:** en la plataforma GitHub el nombre completo del repositorio se compone del nombre del propietario, seguido del nombre del repositorio, separados con una barra vertical, por ejemplo, *usuario/repositorio*.
- **Dirección HTML del repositorio:** además de la información sobre el nombre y el repositorio, se ofrece al cliente la dirección HTML donde el propietario tiene alojado su repositorio.

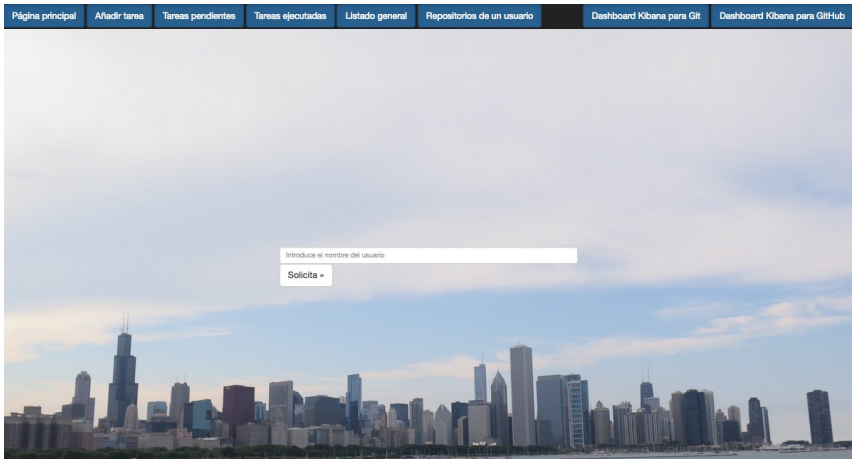


Figura 4.12: Pantalla para introducir el usuario del que se quieren conocer los repositorios.

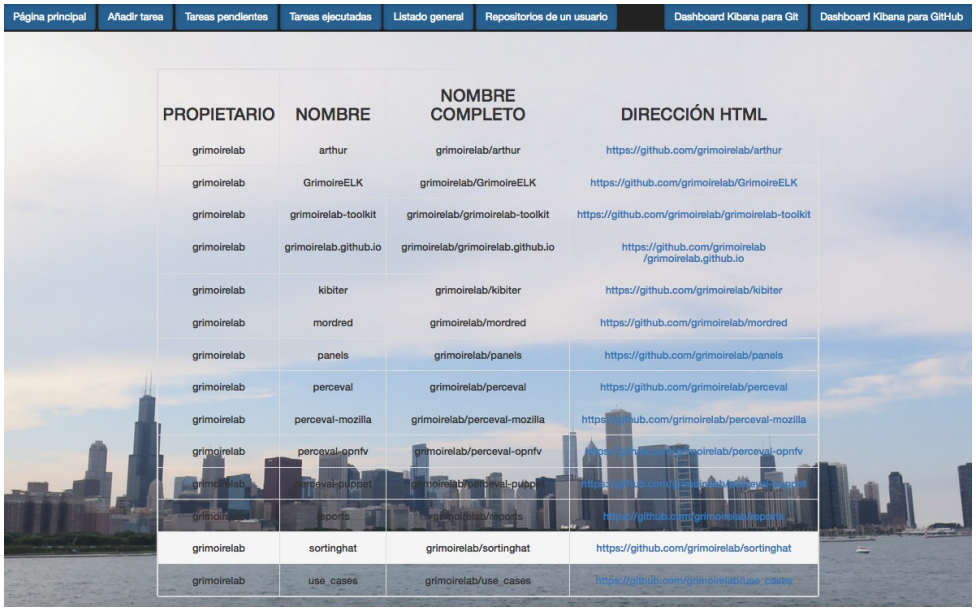


Figura 4.13: Listado de repositorios pertenecientes al usuario dado.

4.1.4.6. Enlace con Kibana

Utilizando cualquiera de los dos enlaces a Kibana, ya sea a través de la barra de navegación superior o desde el panel central de la página principal, se abrirá una nueva pestaña para ver el resultado de la ejecución del sistema, los dashboard de Kibana. Tendremos dos dashaboard, uno para Git y otro para GitHub.

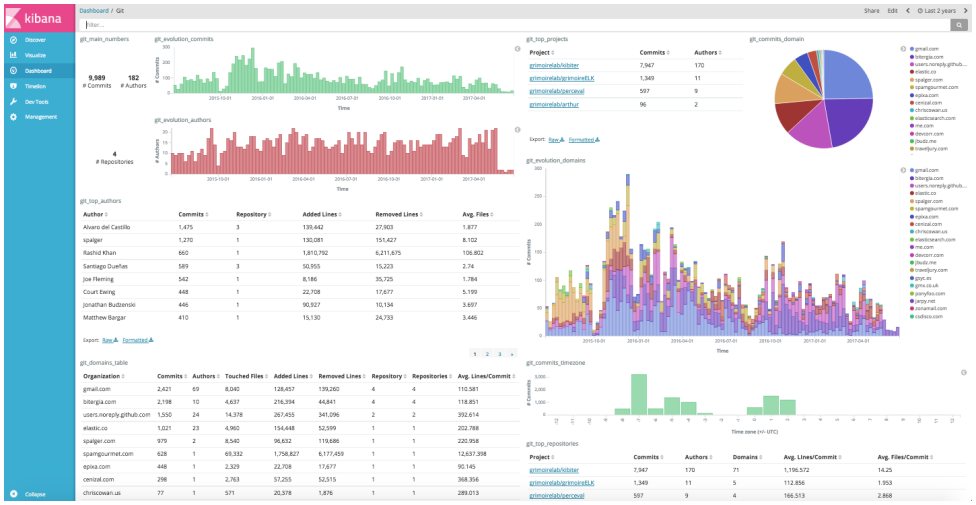


Figura 4.14: Ejemplo de dashboard creado por el sistema para la información de Git.

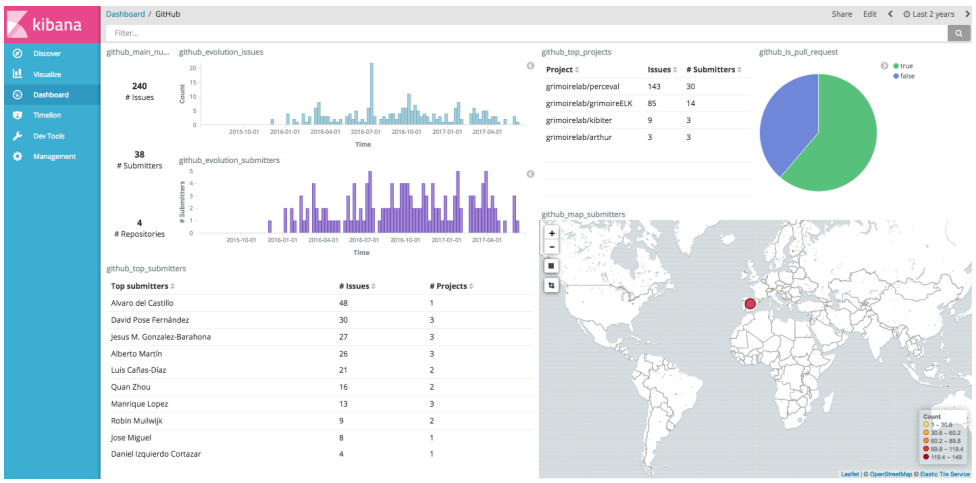


Figura 4.15: Ejemplo de dashboard creado por el sistema para la información de GitHub.

4.2. Descripción de la aplicación: versión API

Como hemos visto en el uso de GitHub, el hecho de que exista una API, hace que la capacidad de interactuar con otros desarrolladores sea total. Por lo tanto, nuestro proyecto estará basado en la creación de una API con la funcionalidad descrita anteriormente, y a partir de ella se ha construido la interfaz web que hemos descrito anteriormente.

Para poder probar y mostrar el funcionamiento de la API implementada se construye un script, que a paso a paso, vaya ejecutando la distinta funcionalidad. Para hacer las pruebas usaremos el comando *curl*, que se encarga de llevar a cabo las peticiones HTTP, admitiendo diferentes opciones como:

- **Tipo de petición:** El comando *curl* acepta un parámetro que será el tipo de petición que se quiere usar, GET, POST, PUT o cualquier otro.
- **Cookies:** También acepta un parámetro correspondiente a las cookies, que necesitaremos ya que nuestro sistema las utiliza para pasarse el usuario que lo está usando en un momento determinado. Para añadir este parámetro hay que poner el flag *-b* seguido de el par *nombre=valor*.
- **Variables para un formulario:** Si la intención es hacer POST a un formulario, necesitaremos darle valores a los campos que solicita el formulario, para ello hay que poner el flag *-F* seguido del par *nombre=valor*, si se quieren enviar varias campos del formulario, hay que repetir el flag y el par tantas veces como campos necesitemos.

4.2.1. Usuarios para la ejecución

Para poder probar la funcionalidad es necesario tener usuarios autenticados en GitHub con su correspondiente token de usuario, esta información la tenemos almacenada en Elasticsearch, por lo que el script lo primero que debe hacer es acceder al índice de Elasticsearch y obtener dos usuarios para hacer las pruebas de la API.


```
{
  "took" : 2,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 2,
    "max_score" : 1.0,
    "hits" : [ {
      "_index" : "usuariosapp",
      "_type" : "usuarios_app_index",
      "_id" : "pfsjob",
      "_score" : 1.0,
      "_source" : {
        "usuario" : "pfsjob",
        "token" : "52a32a92c1c0668f17c266f6ab6f75a1a61043e4"
      }
    }, {
      "_index" : "usuariosapp",
      "_type" : "usuarios_app_index",
      "_id" : "pfsjobTest",
      "_score" : 1.0,
      "_source" : {
        "usuario" : "pfsjobTest",
        "token" : "3d49a13a619085c6e46f90880204d52920d15c23"
      }
    } ]
  }
}
```


Figura 4.16: Contenido del índice de Elasticsearch de usuarios autenticados.

Por lo tanto el script debe extraer de este índice dos pares de usuario-token para hacer las posteriores pruebas, almacenando cada elemento en una variable independiente.

```
Usuario 1: pfsjob Token 1: 52a32a92c1c0668f17c266f6ab6f75a1a61043e4
Usuario 2: pfsjobTest Token 2: 3d49a13a619085c6e46f90880204d52920d15c23
```

Figura 4.17: Dos pares de usuario-token almacenados en variables independientes.

Para comprobar que efectivamente estos usuarios están autenticados en el sistema, haremos uso de la API de GitHub, mediante *curl* con la siguiente instrucción: *curl https://api.github.com/user?access_to TOKEN*, donde *OAUTH-TOKEN*, será el token de usuario que hemos extraído del índice de Elasticsearch.


```
{
  "login": "pfsjob",
  "id": 5685987,
  "avatar_url": "https://avatars3.githubusercontent.com/u/5685987?v=3",
  "gravatar_id": "",
  "url": "https://api.github.com/users/pfsjob",
  "html_url": "https://github.com/pfsjob",
  "followers_url": "https://api.github.com/users/pfsjob/followers",
  "following_url": "https://api.github.com/users/pfsjob/following{/other_user}",
  "gists_url": "https://api.github.com/users/pfsjob/gists{/gist_id}",
  "starred_url": "https://api.github.com/users/pfsjob/starred{/owner}{/repo}",
  "subscriptions_url": "https://api.github.com/users/pfsjob/subscriptions",
  "organizations_url": "https://api.github.com/users/pfsjob/orgs",
  "repos_url": "https://api.github.com/users/pfsjob/repos",
  "events_url": "https://api.github.com/users/pfsjob/events{/privacy}",
  "received_events_url": "https://api.github.com/users/pfsjob/received_events",
  "type": "User",
  "site_admin": false,
  "name": null,
  "company": null,
  "blog": "",
  "location": null,
  "email": null,
  "hireable": null,
  "bio": null,
  "public_repos": 3,
  "public_gists": 0,
  "followers": 0,
  "following": 0,
  "created_at": "2013-10-14T20:37:15Z",
  "updated_at": "2017-06-18T08:23:09Z"
}
```


(a) Usuario 1


```
{
  "login": "pfsjobTest",
  "id": 27086597,
  "avatar_url": "https://avatars3.githubusercontent.com/u/27086597?v=3",
  "gravatar_id": "",
  "url": "https://api.github.com/users/pfsjobTest",
  "html_url": "https://github.com/pfsjobTest",
  "followers_url": "https://api.github.com/users/pfsjobTest/followers",
  "following_url": "https://api.github.com/users/pfsjobTest/following{/other_user}",
  "gists_url": "https://api.github.com/users/pfsjobTest/gists{/gist_id}",
  "starred_url": "https://api.github.com/users/pfsjobTest/starred{/owner}{/repo}",
  "subscriptions_url": "https://api.github.com/users/pfsjobTest/subscriptions",
  "organizations_url": "https://api.github.com/users/pfsjobTest/orgs",
  "repos_url": "https://api.github.com/users/pfsjobTest/repos",
  "events_url": "https://api.github.com/users/pfsjobTest/events{/privacy}",
  "received_events_url": "https://api.github.com/users/pfsjobTest/received_events",
  "type": "User",
  "site_admin": false,
  "name": null,
  "company": null,
  "blog": "",
  "location": null,
  "email": null,
  "hireable": null,
  "bio": null,
  "public_repos": 0,
  "public_gists": 0,
  "followers": 0,
  "following": 0,
  "created_at": "2017-04-08T18:37:49Z",
  "updated_at": "2017-06-18T09:07:51Z"
}
```


(b) Usuario 2

Figura 4.18: Respuesta de la API a la verificación del token de usuario.

4.2.2. Añadir tareas al sistema

Teniendo ya usuarios con los que realizar las pruebas, pasamos a añadir tareas al sistema para su posterior ejecución y creación del dashboard. Para ello tendremos que hacer POST con *curl* a la dirección *http://127.0.0.1:8000/tareas/add*. Al tratarse de un POST a un formulario será necesario enviar con la petición los valores de los campos que solicita el formulario. Para ello necesitaremos añadir como vimos al inicio de la sección el siguiente parámetro *-F 'usuario=usuarioprueba' -F 'repositorio=repositorioprueba'*. Además de esto por la arquitectura de nuestro proyecto necesitamos añadir un valor a la cookie. Nuestro sistema usa la cookie para tener identificado al usuario que está conectado a la aplicación en un momento determinado, y de esta forma saber quien añade las tareas al sistema o de quien tiene que mostrar la información, para ello *curl* necesita el siguiente parámetro *-b 'login=nombreusuario'*. Finalmente la instrucción completa será:

curl POST -b 'login=usuarioprueba' -F 'usuario=usuarioprueba' -F 'repositorio=repositorioprueba' http://127.0.0.1:8000/tareas/add/.

Repetiremos esta instrucción para añadir al índice de Elasticsearch dos instancias, una para cada usuario de los almacenados en el apartado anterior. Por lo tanto, suponiendo que el índice estaba vacío antes de la ejecución del script, su contenido será:


```
{
  "took" : 1,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 2,
    "max_score" : 1.0,
    "hits" : [ {
      "_index" : "tareas",
      "_type" : "tareas_index",
      "_id" : "usuarioprueba-repositorioprueba",
      "_score" : 1.0,
      "_source" : {
        "usuario" : "usuarioprueba",
        "repositorio" : "repositorioprueba",
        "fechaRegistro" : "2017-06-18T10:10:49.801966+00:00",
        "inicioEjecucion" : "1970-01-01T00:00:00",
        "finEjecucion" : "1970-01-01T00:00:00",
        "estado" : false,
        "creador" : "pfsjob"
      }
    }, {
      "_index" : "tareas",
      "_type" : "tareas_index",
      "_id" : "usuarioprueba2-repositorioprueba2",
      "_score" : 1.0,
      "_source" : {
        "usuario" : "usuarioprueba2",
        "repositorio" : "repositorioprueba2",
        "fechaRegistro" : "2017-06-18T10:10:59.783638+00:00",
        "inicioEjecucion" : "1970-01-01T00:00:00",
        "finEjecucion" : "1970-01-01T00:00:00",
        "estado" : false,
        "creador" : "pfsjobTest"
      }
    } ]
  }
}
```


Figura 4.19: Contenido del índice Elasticsearch tras la ejecución del script.

Cada instancia del índice se compone de varios campos, más de los que se han añadido mediante la instrucción con *curl*. Los campos a los que no se han dado valor tienen valores, esto es posible porque en la definición del modelo Django se incluye un parámetro llamado *default*, cuyo valor será el valor por defecto con el que se creará la instancia en el caso de que no se especifique uno.

Para el ejemplo que hemos representado, el campo *fechaRegistro* tiene por defecto el instante de tiempo en el que se creó la instancia.

Lo que aparece antes de la parte que comienza por *"source"*, es la parte de los metadatos, dónde se encuentran los datos que Elasticsearch almacena de forma automática aunque pueden modificarse, como el identificador o el nombre del índice donde está almacenada la instancia, que en nuestro caso es *tareas*.

4.2.3. Listado general de tareas del sistema

El listado general de tareas del sistema nos ofrece un listado completo de las instancias que se han añadido al índice de Elasticsearch, estén o no ejecutadas, sean del usuario que sean. Esta funcionalidad puede no parece útil, a priori, para la parte de visualización, pero informa de la cantidad de tareas que el sistema tiene, tanto ejecutadas como pendientes de ejecutar.

Los argumentos a utilizar con *curl* en este caso son más reducidos, ya que no tenemos que informar al sistema de que usuario está usando la aplicación, debido a que se trata de un listado general. La url sobre la que debemos hacer la petición es *http://127.0.0.1:8000/tareas/lista/*, y la instrucción completa sera:

curl GET http://127.0.0.1:8000/tareas/lista/


```
{
  "_index": "tareas",
  "_type": "tareas_index",
  "_id": "usuarioprueba-repositorioprueba",
  "_score": 1.0,
  "_source": {
    "usuario": "usuarioprueba",
    "repositorio": "repositorioprueba",
    "fechaRegistro": "2017-06-18T10:10:49.801966+00:00",
    "inicioEjecucion": "1970-01-01T00:00:00",
    "finEjecucion": "1970-01-01T00:00:00",
    "estado": false,
    "creador": "pfsjob"
  }
},
{
  "_index": "tareas",
  "_type": "tareas_index",
  "_id": "usuarioprueba2-repositorioprueba2",
  "_score": 1.0,
  "_source": {
    "usuario": "usuarioprueba2",
    "repositorio": "repositorioprueba2",
    "fechaRegistro": "2017-06-18T10:10:59.783638+00:00",
    "inicioEjecucion": "1970-01-01T00:00:00",
    "finEjecucion": "1970-01-01T00:00:00",
    "estado": false,
    "creador": "pfsjobTest"
  }
},
{
  "_index": "tareas",
  "_type": "tareas_index",
  "_id": "usuarioprueba3-repositorioprueba3",
  "_score": 1.0,
  "_source": {
    "usuario": "usuarioprueba3",
    "repositorio": "repositorioprueba3",
    "fechaRegistro": "2017-06-18T10:41:08.562933+00:00",
    "inicioEjecucion": "1970-01-01T00:00:00",
    "finEjecucion": "1970-01-01T00:00:00",
    "estado": true,
    "creador": "pfsjob"
  }
},
{
  "_index": "tareas",
  "_type": "tareas_index",
  "_id": "usuarioprueba4-repositorioprueba4",
  "_score": 1.0,
  "_source": {
    "usuario": "usuarioprueba4",
    "repositorio": "repositorioprueba4",
    "fechaRegistro": "2017-06-18T10:41:36.105154+00:00",
    "inicioEjecucion": "1970-01-01T00:00:00",
    "finEjecucion": "1970-01-01T00:00:00",
    "estado": true,
    "creador": "pfsjobTest"
  }
}
```


Figura 4.20: Respuesta de la API con el listado general

4.2.4. Listado de tareas pendientes de ejecutar para un usuario

Una vez que se añaden tareas al sistema, es conveniente que el usuario pueda saber el estado en el que se encuentran, y para ello se le proporciona la posibilidad de consultar que tareas están pendientes de ejecutar. Para ello, siguiendo la misma metodología que con los anteriores apartados necesitaremos hacer uso del comando *curl* a la dirección *http://127.0.0.1:8000/tareas/listapendientes/*.

Para este caso también necesitaremos el uso de la cookie, por lo que tendremos que añadir *-b 'login=nombreusuario'*. El comando final será:

curl GET -b 'login=nombreusuario' http://127.0.0.1:8000/tareas/listapendientes/

En la imagen están las cuatro instancias que hay actualmente en el sistema. Se han añadido dos instancias más para que existan instancias tanto de usuarios diferentes como ejecutadas y sin ejecutar.

```
"hits": [
  {
    "_index": "tareas",
    "_type": "tareas_index",
    "_id": "usuarioprueba-repositorioprueba",
    "_score": 0.4339554,
    "_source": {
      "usuario": "usuarioprueba",
      "repositorio": "repositorioprueba",
      "fechaRegistro": "2017-06-18T10:10:49.801966+00:00",
      "inicioEjecucion": "1970-01-01T00:00:00",
      "finEjecucion": "1970-01-01T00:00:00",
      "estado": false,
      "creador": "pfsjob"
    }
  }
]
```

```
"hits": [
  {
    "_index": "tareas",
    "_type": "tareas_index",
    "_id": "usuarioprueba2-repositorioprueba2",
    "_score": 1.724915,
    "_source": {
      "usuario": "usuarioprueba2",
      "repositorio": "repositorioprueba2",
      "fechaRegistro": "2017-06-18T10:10:59.783638+00:00",
      "inicioEjecucion": "1970-01-01T00:00:00",
      "finEjecucion": "1970-01-01T00:00:00",
      "estado": false,
      "creador": "pfsjobTest"
    }
  }
]
```


(a) Usuario 1 (b) Usuario 2

Figura 4.21: Respuesta de la API a las tareas pendientes de ejecutar para cada usuario.

4.2.5. Listado de tareas ya ejecutadas para un usuario

Una vez que se añaden tareas al sistema, es conveniente que el usuario pueda saber el estado en el que se encuentran, y para ello se le proporciona la posibilidad de consultar que tareas están ya ejecutadas. Para ello, siguiendo la misma metodología que con los anteriores apartados necesitaremos hacer uso del comando *curl* a la dirección *http://127.0.0.1:8000/tareas/listaejecutados/*. Para este caso también necesitaremos el uso de la cookie por lo que tendremos que añadir *-b 'login=nombreusuario'*. El comando final será:

curl GET -b 'login=nombreusuario' http://127.0.0.1:8000/tareas/listaejecutados/

```
"hits": [
  {
    "_index": "tareas",
    "_type": "tareas_index",
    "_id": "usuarioprueba3-repositorioprueba3",
    "_score": 1.724915,
    "_source": {
      "usuario": "usuarioprueba3",
      "repositorio": "repositorioprueba3",
      "fechaRegistro": "2017-06-18T10:41:08.562933+00:00",
      "inicioEjecucion": "1970-01-01T00:00:00",
      "finEjecucion": "1970-01-01T00:00:00",
      "estado": true,
      "creador": "pfsjob"
    }
  }
]
```

```
"hits": [
  {
    "_index": "tareas",
    "_type": "tareas_index",
    "_id": "usuarioprueba4-repositorioprueba4",
    "_score": 1.4142135,
    "_source": {
      "usuario": "usuarioprueba4",
      "repositorio": "repositorioprueba4",
      "fechaRegistro": "2017-06-18T10:41:36.105154+00:00",
      "inicioEjecucion": "1970-01-01T00:00:00",
      "finEjecucion": "1970-01-01T00:00:00",
      "estado": true,
      "creador": "pfsjobTest"
    }
  }
]
```


(a) Usuario 1 (b) Usuario 2

Figura 4.22: Respuesta de la API a las tareas ya ejecutadas para cada usuario.

4.2.6. Estadísticas de las tareas ya ejecutadas

La salida que ofrece el listado de tareas ya ejecutadas, sólo contiene información sobre los tiempos de ejecución de dichas tareas. Por lo tanto es interesante añadir un método que se encargue de calcular una serie de estadísticas que informen al usuario de los tiempos de ejecución:

- Tiempo de ejecución efectiva.
- Tiempo de respuesta.
- Antigüedad en el sistema.

Para llevar a cabo esta ejecución, seguiremos la misma metodología que con los anteriores apartados. Necesitaremos usar el comando *curl* con *http://127.0.0.1:800/tareas/estadisticasExec/*. Para este caso hay que indicar el propietario y el nombre del repositorio, mediante *-F 'usuario=grimoirelab' -F 'repositorio=perceval'*. El comando final será:

curl POST -F 'usuario=grimoirelab' -F 'repositorio=perceval'
http://127.0.0.1:800/tareas/estadisticasExec/


```
[
  {
    "propietario": "grimoirelab",
    "repositorio": "perceval",
    "fechaRegistro": "Creado a las 18h 45min 48seg del 27-06-2017",
    "inicioEjecucion": "Iniciado a las 19h 04min 21seg del 27-06-2017",
    "finEjecucion": "Finalizado a las 19h 05min 48seg del 27-06-2017",
    "tEjecucion": "0:01:26.930040",
    "tProceso": "0:19:59.909037",
    "tRespuesta": "22:51:05.961682"
  }
```


Figura 4.23: Respuesta de la API a la consulta sobre las estadísticas de una tarea ejecutada.

4.2.7. Lista de repositorios pertenecientes a un usuario GitHub dado

Otra funcionalidad que se le ofrece al usuario, es la de a partir de un usuario cualquiera de GitHub, obtener todos aquellos repositorios que el usuario dado tenga declarados como públicos. Para ello nuestra aplicación necesitará interactuar con la API de GitHub y, a partir del resultado que obtiene de dicha interacción crear una salida para nuestra API.

La respuesta que devuelve una consulta a la API de GitHub acerca de los repositorios públicos de un usuario de GitHub determinado incluye una gran cantidad de información.


```
{
  "id": 85123655,
  "name": "Dashboard",
  "full_name": "pfsjob/Dashboard",
  "owner": {
    "login": "pfsjob",
    "id": 5685987,
    "avatar_url": "https://avatars3.githubusercontent.com/u/5685987?v=3",
    "gravatar_id": "",
    "url": "https://api.github.com/users/pfsjob",
    "html_url": "https://github.com/pfsjob",
    "followers_url": "https://api.github.com/users/pfsjob/followers",
    "following_url": "https://api.github.com/users/pfsjob/following{/other_user}",
    "gists_url": "https://api.github.com/users/pfsjob/gists{/gist_id}",
    "starred_url": "https://api.github.com/users/pfsjob/starred{/owner}{/repo}",
    "subscriptions_url": "https://api.github.com/users/pfsjob/subscriptions",
    "organizations_url": "https://api.github.com/users/pfsjob/orgs",
    "repos_url": "https://api.github.com/users/pfsjob/repos",
    "events_url": "https://api.github.com/users/pfsjob/events{/privacy}",
    "received_events_url": "https://api.github.com/users/pfsjob/received_events",
    "type": "User",
    "site_admin": false
  },
  "private": false,
  "html_url": "https://github.com/pfsjob/Dashboard",
  "description": null,
  "fork": false,
  "url": "https://api.github.com/repos/pfsjob/Dashboard",
  "forks_url": "https://api.github.com/repos/pfsjob/Dashboard/forks",
  "keys_url": "https://api.github.com/repos/pfsjob/Dashboard/keys{/key_id}",
  "collaborators_url": "https://api.github.com/repos/pfsjob/Dashboard/collaborators{/collaborator}",
  "teams_url": "https://api.github.com/repos/pfsjob/Dashboard/teams",
  "hooks_url": "https://api.github.com/repos/pfsjob/Dashboard/hooks",
  "issue_events_url": "https://api.github.com/repos/pfsjob/Dashboard/issues/events{/number}",
  "events_url": "https://api.github.com/repos/pfsjob/Dashboard/events",
  "assignees_url": "https://api.github.com/repos/pfsjob/Dashboard/assignees{/user}",
  "branches_url": "https://api.github.com/repos/pfsjob/Dashboard/branches{/branch}",
  "tags_url": "https://api.github.com/repos/pfsjob/Dashboard/tags",
  "blobs_url": "https://api.github.com/repos/pfsjob/Dashboard/git/blobs{/sha}",
  "git_tags_url": "https://api.github.com/repos/pfsjob/Dashboard/git/tags{/sha}",
  "git_refs_url": "https://api.github.com/repos/pfsjob/Dashboard/git/refs{/sha}",
  "trees_url": "https://api.github.com/repos/pfsjob/Dashboard/git/trees{/sha}",
  "statuses_url": "https://api.github.com/repos/pfsjob/Dashboard/statuses/{sha}",
  "languages_url": "https://api.github.com/repos/pfsjob/Dashboard/languages",
  "stargazers_url": "https://api.github.com/repos/pfsjob/Dashboard/stargazers",
  "contributors_url": "https://api.github.com/repos/pfsjob/Dashboard/contributors",
```

```
"subscribers_url": "https://api.github.com/repos/pfsjob/Dashboard/subscribers",
"subscription_url": "https://api.github.com/repos/pfsjob/Dashboard/subscription",
"commits_url": "https://api.github.com/repos/pfsjob/Dashboard/commits{/sha}",
"git_commits_url": "https://api.github.com/repos/pfsjob/Dashboard/git/commits{/sha}",
"comments_url": "https://api.github.com/repos/pfsjob/Dashboard/comments{/number}",
"issue_comment_url": "https://api.github.com/repos/pfsjob/Dashboard/issues/comments{/number}",
"contents_url": "https://api.github.com/repos/pfsjob/Dashboard/contents/{+path}",
"compare_url": "https://api.github.com/repos/pfsjob/Dashboard/compare/{base}...{head}",
"merges_url": "https://api.github.com/repos/pfsjob/Dashboard/merges",
"archive_url": "https://api.github.com/repos/pfsjob/Dashboard/{archive_format}{/ref}",
"downloads_url": "https://api.github.com/repos/pfsjob/Dashboard/downloads",
"issues_url": "https://api.github.com/repos/pfsjob/Dashboard/issues{/number}",
"pulls_url": "https://api.github.com/repos/pfsjob/Dashboard/pulls{/number}",
"milestones_url": "https://api.github.com/repos/pfsjob/Dashboard/milestones{/number}",
"notifications_url": "https://api.github.com/repos/pfsjob/Dashboard/notifications{?since,all,participating}",
"labels_url": "https://api.github.com/repos/pfsjob/Dashboard/labels{/name}",
"releases_url": "https://api.github.com/repos/pfsjob/Dashboard/releases{/id}",
"deployments_url": "https://api.github.com/repos/pfsjob/Dashboard/deployments",
"created_at": "2017-03-15T21:42:17Z",
"updated_at": "2017-03-15T21:47:05Z",
"pushed_at": "2017-04-16T17:49:02Z",
"git_url": "git://github.com/pfsjob/Dashboard.git",
"ssh_url": "git@github.com:pfsjob/Dashboard.git",
"clone_url": "https://github.com/pfsjob/Dashboard.git",
"svn_url": "https://github.com/pfsjob/Dashboard",
"homepage": null,
"size": 63,
"stargazers_count": 0,
"watchers_count": 0,
"language": "Python",
"has_issues": true,
"has_projects": true,
"has_downloads": true,
"has_wiki": true,
"has_pages": false,
"forks_count": 0,
"mirror_url": null,
"open_issues_count": 0,
"forks": 0,
"open_issues": 0,
"watchers": 0,
"default_branch": "master"
},
```


Figura 4.24: Respuesta de la API GitHub a la consulta sobre un usuario dado.

En los cuatro fragmentos mostrados arriba, está la salida de la consulta a GitHub sobre un usuario, y sólo sobre uno de sus repositorios. Contiene toda la información que tiene el repositorio, desde el nombre, la url, el lenguaje de programación, el número de fork realizados sobre él o el número de identificador.

Es evidente que para nuestro proyecto no es necesaria todo la información, por lo que seleccionamos los parámetros que nos parecen más adecuados:

- Propietario del repositorio.
- Nombre del repositorio.
- Nombre completo del repositorio.
- Url de acceso al repositorio.

A partir de estos parámetros construimos una estructura JSON, para que la salida de esta funcionalidad tenga el mismo formato que la del resto de la API.


```
[
  {
    "propietario": "pfsjob",
    "nombre": "Dashboard",
    "nombre_completo": "pfsjob/Dashboard",
    "direccion_html": "https://github.com/pfsjob/Dashboard"
  },
  {
    "propietario": "pfsjob",
    "nombre": "Graficas",
    "nombre_completo": "pfsjob/Graficas",
    "direccion_html": "https://github.com/pfsjob/Graficas"
  },
  {
    "propietario": "pfsjob",
    "nombre": "pfsjob.github.io",
    "nombre_completo": "pfsjob/pfsjob.github.io",
    "direccion_html": "https://github.com/pfsjob/pfsjob.github.io"
  }
```


Figura 4.25: Respuesta de la API a la consulta sobre un usuario determinado.

4.3. Implementación

En el apartado de implementación veremos la arquitectura de la aplicación, en primer lugar de forma general y luego las partes más importantes. Justo debajo se muestra un diagrama con la organización de las diferentes tecnologías que intervienen en el proyecto.

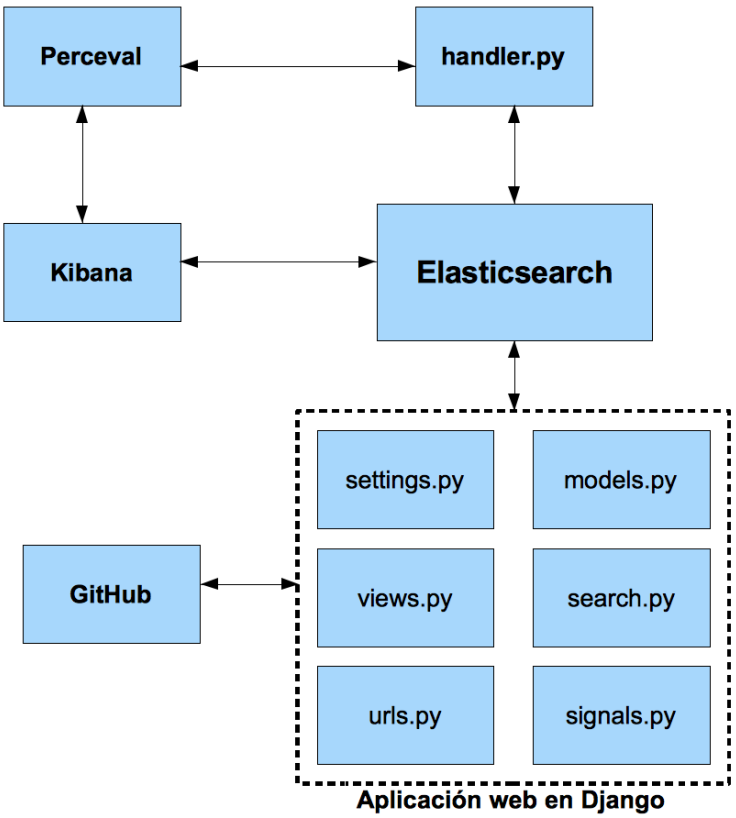


Figura 4.26: Diagrama que representa la arquitectura de la aplicación.

Las flechas del diagrama representan la interacción entre los módulos, es decir, entre que módulos existe un intercambio de información.

4.3.1. Arquitectura general

La arquitectura general de la aplicación describe la forma en la que está construida, es decir, las partes de las que se compone, y como éstas interaccionan entre ellas. Como ya hemos señalado en apartados anteriores estas partes de la aplicación se han construido por separado, de forma modular. El desarrollo se ha realizado en 4 iteraciones, donde el principio de cada una es la conexión con la iteración anterior y el final la preparación para la conexión con la siguiente.

A continuación vemos un caso de uso de la aplicación, con un diagrama que representa el flujo de ejecución y de información. En este caso el usuario quiere añadir una tarea al sistema, veremos cual es el proceso y cómo y cuando intervienen las diferentes tecnologías que empleamos.

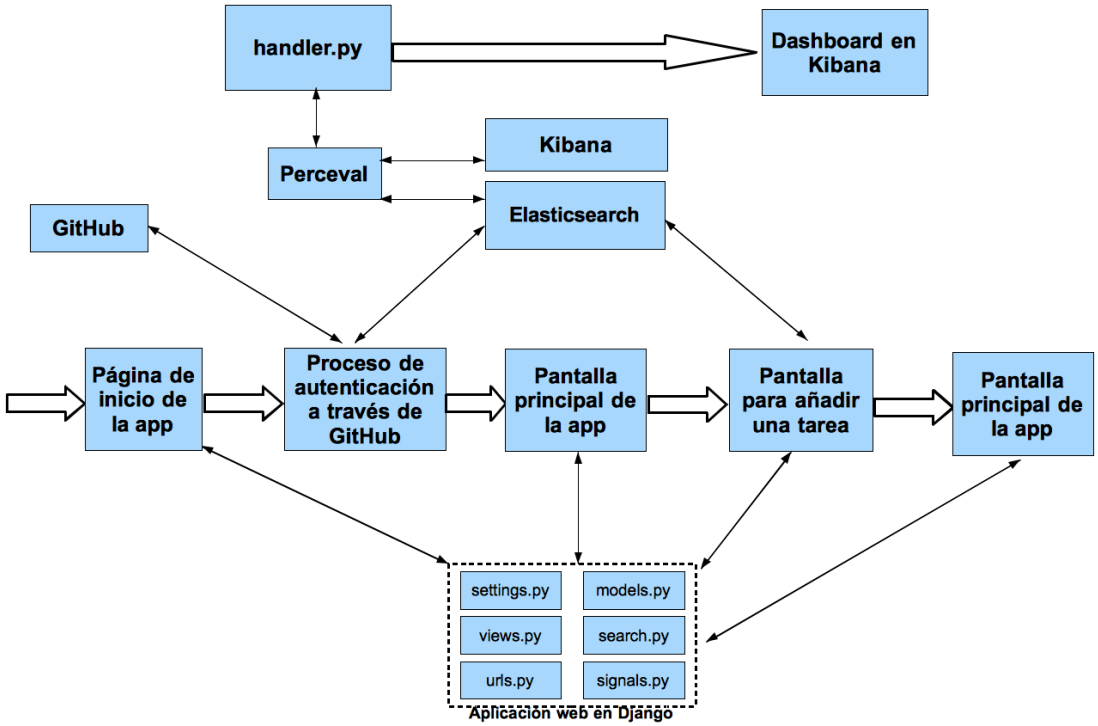


Figura 4.27: Diagrama que representa el proceso de añadir una tarea así como las tecnologías que intervienen en él.

Las flechas más anchas representan el flujo de ejecución y las finas el flujo de la información. Respecto al flujo de ejecución, existen dos procesos en paralelo:

- **Servidor WEB:** El flujo web es el encargado de servir al usuario la interfaz web para realizar las tareas, dependiendo de lo que el usuario quiera el flujo podrá variar ligeramente, en función de las tecnologías que puedan influir.
- **Creación del dashboard:** El flujo de creación es el encargado de comprobar periódicamente las tareas que están almacenadas en el sistema, y proceder según su estado. Si están pendientes de ejecutar, llevará a cabo la ejecución, actualizará la información relativa a los tiempos de ejecución e informará al usuario del lugar donde puede consultar el dashboard solicitado.

Veamos más detalladamente las etapas del caso de uso *Añadir tarea*.

1. **Pantalla inicial:** el cliente que accede a la aplicación llega directamente a la pantalla principal, donde Django sirve la interfaz web, en ésta se informa al usuario de la necesidad de autenticarse a través de GitHub, contiene información general acerca de la aplicación e información sobre el creador/propietario. En este apartado la única tecnología que interviene es Django, sirviendo la interfaz web como hemos indicado anteriormente.
2. **Proceso de autenticación a través de GitHub:** cuando el cliente hace click sobre el botón que inicia el proceso de autenticación a través de GitHub, comienza dicho proceso navegando a través de dos páginas web pertenecientes a GitHub, que ofrecen al usuario

mayor seguridad y confianza, la primera para loguearse en la plataforma y la segunda para autorizar a nuestra aplicación. Finalmente Django se encargará de mostrar la pantalla principal de la aplicación. Por lo tanto en este apartado primero se interacciona con la API de GitHub, para mostrar las páginas de autenticación, redireccionando finalmente al usuario a la página principal de la aplicación. En segundo lugar, justo antes de mostrar la página principal Django debe interaccionar con GitHub para obtener el token de usuario y con Elasticsearch para almacenar dicho token así como el usuario al que pertenece en el índice correspondiente a los usuarios.

3. **Añadir una tarea:** Cuando el usuario accede a la parte de añadir una tarea, Django le sirve un formulario para que introduzca los datos. Una vez que el cliente rellena el formulario, tiene un botón para añadir la tarea al sistema, que le redirecciona a la página principal. Antes de mostrar la página principal Django almacena la información de la tarea en el índice correspondiente de Elasticsearch.

- **Creación del dashboard:** Paralelamente a la ejecución de los pasos anteriores, el script de creación de dashboards debe interactuar con varias tecnologías para crearlo. En primer lugar accede a la información contenida en Elasticsearch sobre las tareas y el token del usuario, pidiendo solamente las pendientes de ejecutar y el token del usuario activo. En segundo lugar mediante Perceval, a través de GrimoireELK, ejecutará los scripts correspondientes a la obtención de datos y su posterior enriquecimiento, estos datos serán almacenados también en Elasticsearch. Por último antes de pasar al reposo para esperar el siguiente barrido, debe actualizar la información de tiempos de ejecución, justo antes de empezar a ejecutar y justo después. También debe cambiar el estado a ya ejecutado.

El resto de casos de uso posibles con la funcionalidad que incluye la aplicación tendrá un diagrama similar al anterior, excepto en el último paso, dónde el tipo de funcionalidad que seleccione el usuario pondrá en juego una tecnología u otra.

4.3.2. Archivos Django: settings.py

El archivo settings.py pertenece al conjunto de ficheros que forman la aplicación web de Django. En concreto este archivo contiene los valores fijos de configuración que necesita Django para la ejecución:

- **Configuración de base de datos(DATABASES):** en esta zona se configuran los valores según el tipo de base de datos que utilizará el sistema, mySQL, SQLlite o cualquier otra. Para nuestro caso no tendrá ninguna modificación, ya que usaremos Elasticsearch que no necesita configuración en este apartado.

- **Configuración de templates(TEMPLATES):** en esta parte está la información y configuración de las plantillas que usará Django para la visualización. Para nuestro caso añadiremos la información sobre la ubicación de las plantillas y un par de lineas sobre *Python Social Auth* para la autenticación a través de GitHub.

- **Aplicaciones instaladas(INSTALLED_APPS):** aquí estarán las aplicaciones que el sistema lleva instaladas por defecto, como *sessions* para dar soporte de sesiones o *admin* para ofrecer el módulo de administrador y de gestión de usuarios. Para nuestro caso es

imprescindible añadir la aplicación creada dentro del proyecto Django, que se llama *creador*.

- **Intercambio de información(MIDDLEWARE):** parte en la que se añade la información necesaria para el intercambio de información entre nuestra aplicación y el resto de aplicaciones. Para nuestro caso, tendremos que añadir información sobre *Python Social Auth*, que es el paquete de terceros que integramos en nuestro proyecto.

- **Autenticación en otros sistemas(AUTHENTICATION_BACKENDS):** zona para configurar aquellos sistemas con los que se desea que la aplicación tenga soporte para la autorización en ellos o a través de ellos. Para nuestro caso es necesario añadir el soporte para autenticación en GitHub mediante el protocolo Oauth2.

Estas son las zonas de configuración más habituales, que además son las que tienen alguna modificación para nuestra aplicación. También hay opciones de configuración para la zona horaria o el idioma.

```python
# Application definition

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'creador',
]
MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    #'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
    'social_django.middleware.SocialAuthExceptionMiddleware',
]
ROOT_URLCONF = 'terceraIteracion.urls'
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [os.path.join(os.path.split(os.path.abspath(os.path.dirname(__file__)))[0],'.'),],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
                'social_django.context_processors.backends',
                'social_django.context_processors.login_redirect',
            ],
        },
    },
]
WSGI_APPLICATION = 'terceraIteracion.wsgi.application'
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
```

```python

AUTHENTICATION_BACKENDS = (
    'social_core.backends.open_id.OpenIdAuth',
    #'social_auth.backends.contrib.github.GithubBackend',
    'social_core.backends.github.GithubOAuth2',
    'django.contrib.auth.backends.ModelBackend',
)

SOCIAL_AUTH_PIPELINE = (

)


# Internationalization
# https://docs.djangoproject.com/en/1.10/topics/i18n/

LANGUAGE_CODE = 'en-us'

TIME_ZONE = 'Europe/Madrid'

USE_I18N = True

USE_L10N = True

USE_TZ = True

LOGIN_URL = 'login'
LOGOUT_URL = 'logout'
LOGIN_REDIRECT_URL = '/tareas/index/'


# Static files (CSS, JavaScript, Images)
# https://docs.djangoproject.com/en/1.10/howto/static-files/

STATIC_URL = '/static/'
```

Figura 4.28: Contenido del fichero settings.py de la aplicación.

4.3.3. Archivos Django: models.py

En models.py está la información acerca de los modelos Django que usará la aplicación. El modelo Django no es más que un objeto especial que guardaremos en la base de datos, por lo tanto hay que definir sus características y sus acciones, que en programación orientada a objetos son sus atributos y sus métodos.

Por lo tanto es aquí donde tenemos que definir las unidades de información que vamos almacenar en Elasticsearch. Lo primero es pensar que información necesitamos para nuestra aplicación, para ello hemos seguido una construcción modular que consiste en ir añadiendo atributos según las necesidades de la iteración. La evolución es la siguiente:

1. **Primera aproximación:** lo primero era guardar la información sobre que repositorio solicita el cliente, para ello era necesario el propietario, el nombre del repositorio y un campo para determinar si ya se había creado el dashboard o no. El modelo, y la posterior instancia en Elasticsearch se llamará *tareas*.

   Ya tenemos la información necesaria, pero ahora necesitamos un método que se encargue de indexar la instancia en Elascticsearch, ese método será *.indexing()*.

2. **Información complementaria:** para completar el modelo, hay que añadir campos con información relativa a la fecha de registro, la fecha de inicio de ejecución y la fecha del final de la ejecución, con el objetivo de poder mostrar estadísticas acerca de los tiempos de ejecución.

   El método *.indexing()* debe actualizarse para incluir los nuevos campos añadidos.

3. **Modelo final:** los requerimientos de la iteración final hacen necesaria la incorporación de un modelo nuevo con información sobre los usuarios que se autentiquen a través de GitHub y su token de usuario. Para ello creamos el modelo *usuariosapp*, con los campos usuario y token, y un método *.indexing()* como el del modelo *tareas*, adaptado a los campos existentes.

   La creación de un modelo con información sobre usuarios, implica que el modelo *tareas* debe contener la información de quien añade esa tarea al sistema, para de esta forma poder relacionarlos. Por lo tanto se añade un campo llamado creador para recoger esa información.

```
27
28  class tareas(models.Model):
29      usuario=models.CharField(max_length=100)
30      repositorio=models.CharField(max_length=100)
31      fechaRegistro=models.DateTimeField(default=timezone.now)
32      inicioEjecucion=models.DateTimeField(default=horainicio)
33      finEjecucion=models.DateTimeField(default=horainicio)
34      estado=models.BooleanField(default=False)
35      creador=models.CharField(max_length=100, default="None")
36
37      def indexing(self):
38          obj = tareasIndex(
39              meta={'type': "doc_type", 'id': self.usuario+"-"+self.repositorio},
40              usuario=self.usuario,
41              repositorio=self.repositorio,
42              fechaRegistro=self.fechaRegistro,
43              inicioEjecucion=self.inicioEjecucion,
44              finEjecucion=self.finEjecucion,
45              estado=self.estado,
46              creador=self.creador,
47          )
48          obj.save(index='tareas')
49          return obj.to_dict(include_meta=True)
```

Figura 4.29: Código del modelo Django *tareas*

```
class usuariosapp(models.Model):
    usuario=models.CharField(max_length=100)
    token=models.CharField(max_length=100)

    def indexing(self):
        obj = usuariosAppIndex(
            meta={'type': "doc_type", 'id': self.usuario},
            usuario=self.usuario,
            token=self.token,
        )
        obj.save(index='usuariosapp')
        return obj.to_dict(include_meta=True)

```

Figura 4.30: Código del modelo Django *usuariosapp*

4.3.4. Archivos Django: urls.py

El archivo urls.py contiene el manejador de urls basado en expresiones regulares con el que Django procesa las peticiones a su servidor. El sistema que usa para procesar estas peticiones es *regex*, abreviaturas de expresiones regulares, muy complejo y potente.

Un proyecto Django puede tener más de un archivo urls.py, esto se debe a que el proyecto general tiene su propio urls.py y cada aplicación creada dentro de él tiene el suyo. A continuación vemos dos ejemplos:

```
from django.conf.urls import include, url
from django.contrib import admin
from social_django.urls import urlpatterns as social_dianao_urls
from creador import views


urlpatterns = [
    url(r'^creador/', include('creador.urls')),
    url(r'^admin/', admin.site.urls),
    url(r'^tareas/', include('creador.urls', namespace="utareas")),
    url(r'^oauth/complete/github/', views.indexregistrado ),
]
```

(a) Archivo del proyecto general

```
from django.conf.urls import url, include
from social_django.urls import urlpatterns as social_dianao_urls
from . import views

urlpatterns = [
    url(r'^index/', views.index, name='tindex'),
    url(r'^lista/$', views.lista_tareas, name='tlist'),
    url(r'^listaejecutados/$', views.lista_tareas_ejecutadas, name='tlistexec'),
    url(r'^listapendientes/$', views.lista_tareas_pendientes, name='tlistpend'),
    url(r'^listausuario/', views.lista_tareas_usuario, name='tlistu'),
    url(r'^add/$', views.add_tarea, name='tadd'),
    url(r'^delete/$', views.delete_tarea, name='tdelete'),
]
```

(b) Archivo de la aplicación

Figura 4.31: Archivos urls.py de nuestro proyecto.

A la vista del contenido de ambos archivos y tomando una url ejemplo como:

http://127.0.0.1:8000/nombreapp/recurso1/

donde *nombreapp* es el nombre de la aplicación creada dentro del proyecto Django y *recurso1* es el recurso que quiero servir, el archivo del proyecto general actúa como primer filtro, es decir, lo que sigue a *http://127.0.0.1:8000* será lo que se analice en primer lugar (nombreapp), comparando mediante regex las entradas que tiene el fichero. Para que el el archivo urls.py de la aplicación entre en juego, el urls.py general debe tener una línea para la aplicación, en nuestro caso *url(r'ĉreador/', include('creador.urls', namespace=ütareas"))*, lo que hará que directamente todo lo que esté detrás de *creador/* sea analizado por el urls.py de la aplicación (recurso1).

La forma en la que urls.py procesa estas peticiones es la siguiente:

1. **Análisis de la url recibida:** en primer lugar se analiza la cadena de texto que sigue a la primera barra, pasándolo por regex, para en función del resultado elegir la línea adecuada del fichero. Gracias a las expresiones regulares, el desarrollador puede evitar tener una línea en el fichero para cada recurso que quiera servir, ya que puede agrupar aquellas que tengas un patrón común bajo una sola expresión regular, por ejemplo, el caso en el que se sirvan recursos numerados como una colección de libros, con una url similar a */libro/numLibro*, no será necesario usar tantas líneas como libros existan, sino que regex ofrece la posibilidad de usar una regla que considera todos los dígitos en una sola línea.

2. **Asociación de la url con una vista:** una vez que se ha seleccionado la línea correspondiente del fichero, ésta se encargará de conectarla con la vista adecuada de views.py. Para ello cada línea lleva un enlace con la acción que se llevará a cabo cuando se solicite el recurso. En el caso de conectarse a una vista de views.py será *views.vistacorrespondiente*, donde *vistacorrespondiente* está definida dentro de views.py, aunque también pueden ser acciones predefinidas en Django o aplicaciones de terceros, como *admin.site.urls*.

4.3.5. Archivos Django: signals.py

El archivo signals.py proporciona al usuario la posibilidad de capturar ciertas acciones que se realizan en Django, e implementar funcionalidad para que se ejecute una vez que ocurran. Las señales pueden estar asociadas a varias partes de Django, por ejemplo el modelo, con señales cuando guardamos un modelo o lo borramos, justo antes de la ejecución y justo después. También existen señales asociadas a la creación de una conexión con la base de datos o al parámetro request/response de las peticiones HTTP.

Para nuestra aplicación, nos interesa saber cuando guardamos un objeto de los modelos, y para esto usaremos la señal *post_save*, es decir, la funcionalidad se ejecutará justo después de llamar al método *.save()*.

```
1  from .models import tareas
2  from django.db.models.signals import post_save
3  from django.dispatch import receiver
4  import time
5
6  @receiver(post_save, sender=tareas)
7  def index_post(sender, instance, **kwargs):
8      instance.indexing()
```

Figura 4.32: Código del fichero signals.py.

Se puede apreciar que el código es sencillo, cuando se llama a *save()*, inmediatamente después se ejecuta el contenido del método *index_post*, que se encarga de ejecutar el método *.indexing()*, que como explicamos, llevará la información del modelo a Elasticsearch.

4.3.6. Archivos Django: views.py

Es el fichero más importante de la estructura Django, contiene la funcionalidad que la aplicación ejecutará cuando el usuario solicita un recurso con entrada en el fichero urls.py, es decir,

contiene la lógica de la aplicación.

El funcionamiento del fichero consiste en responder la solicitud de un recurso por parte de un usuario determinado, para ello urls.py lo analiza y en el caso de encajar con alguna de sus líneas, lo enlaza con la acción recogida en views.py. Como la solicitud de un recurso no es más que una petición HTTP, views.py está preparado para que cada método definido en su interior pueda recibir un parámetro llamado *request*, donde estará el contenido de la petición HTTP. Esta información es valiosa porque contiene el tipo de petición (GET, POST, PUT,..), las cookies si las hubiera, los campos enviados con el POST, cabeceras y más información, que ayuda a la implementación del contenido de la vista.

En nuestro proyecto views.py debe interaccionar con otras tecnologías participantes del proyecto, como Elasticsearch, JSON o GitHub. Para ello necesitará de ayuda, es decir, necesita importar librerías (de la librería estándar de Python o de terceros), para dar el soporte necesario para trabajar con estas tecnologías.

```
1  from django.shortcuts import render, render_to_response, redirect
2
3  # Create your views here.
4  from django.http import HttpResponse, HttpResponseRedirect
5  from creador.models import tareasForm
6  from django.core.urlresolvers import reverse
7  from django.views.generic import ListView
8  from .models import tareas, usuariosapp
9  from .search import tareasIndex, usuariosAppIndex
10 from elasticsearch import Elasticsearch, TransportError
11 from elasticsearch_dsl import Q
12 import elasticsearch_dsl
13 import time
14 import requests
15 import json
```

Figura 4.33: Líneas de importación en views.py

Para la interacción con las tecnologías anteriormente señaladas, utilizaremos las últimas líneas, desde *import elasticsearch_dsl* a *import json*, el resto se utilizan para las visualizaciones, o para incluir clases o métodos desde otros archivos del proyecta, como models.py o search.py.

Las principales vistas incluidas en el views.py de nuestro proyecto son:

- **Mostrar la pantalla de inicio:** cuando el cliente solicita el recurso de entrada a la aplicación, entra en juego el método para mostrarla. Éste método es simple, sirve la plantilla correspondiente a la página de inicio, *index.html*.

```
22 def index(request):
23     return render_to_response('index.html')
```

Figura 4.34: Código correspondiente al método para mostrar la pantalla de inicio.

- **Proceso previo a mostrar la pantalla principal:** Después del proceso de autenticación a través de GitHub, la aplicación debe servir al usuario una pantalla principal, con la funcionalidad que pone a su disposición. Antes de esto, hay que llevar a cabo las siguientes tareas:

1. **Obtención del access_token:** como ya se explicó en el capitulo de desarrollo, una vez autenticado el cliente a través de GitHub, necesitamos obtener su token de usuario. Cuando GitHub redirige al usuario a la página que se indicó en el registro de la aplicación en la plataforma, el parámetro request contiene un campo llamado *code*, que necesitamos para la obtención del token. Por lo tanto, mediante la librería *Requests* realizaremos una petición HTTP a la API de GitHub, cuyo contenido debe incluir el valor de *code*, así como los parámetros correspondientes al registro de la app en GitHub, *Client ID* y *Client Secret*. Además de estos datos en el cuerpo de *Requests* pondremos la url que nos indica la API de GitHub y cabeceras para que la respuesta sea en formato JSON. Para terminar hay que convertir el contenido de la respuesta a *Requests* a un formato entendible por Python. Al solicitar la respuesta en formato JSON, sabemos que *Requests* nos lo devolverá así, pero en un array de bytes, que mediante *json.loads()* transformaremos a un objeto Python para extraer el token de usuario.

2. **Verificación del nombre de usuario:** una vez que tenemos el token de usuario, necesitamos tener el nombre de usuario de GitHub, para poder insertarlo en las cookies y tener al usuario que está conectado identificado. Para ello tendremos que realizar de nuevo una petición a la API de GitHub mediante *Requests*, esta vez con el token de usuario dentro de las cabeceras como indica la API, de nuevo con la respuesta debemos hacer el mismo proceso para que sea un objeto entendible por Python.

3. **Registro del usuario y su token en Elasticsearch:** como último paso antes de servir la página principal al usuario, debemos almacenar en Elasticsearch el nombre de éste con su token. Para ello abriremos una conexión a Elasticsearch mediante la librería elasticsearch_dsl, al índice de *usuariosapp*. Para indexar la información debemos antes aplicar el método *.scan()*, para convertir el resultado de la consulta a Elasticsearch en un objeto iterable, una vez lo tenemos, iteramos para comprobar si el usuario ya existe, en ese caso actualizamos mediante *.get()* para obtener el nodo completo y *.update()* para modificar el valor del token. Si el usuario no existe, entonces creamos una nueva instancia con el método *.indexing()*. La forma de llevar a cabo esta tarea plantea el problema de que sea el primer usuario en el índice, ya que provocaría una excepción al intentar recorrer el objeto iterable, que lógicamente está vacío. Para evitar este problema, capturamos la excepción *TransportError*, de forma que si aparece, creemos el usuario mediante el método *.indexing()*.

4. **Mostrar la página principal:** para finalizar el método, debemos insertar la cookie con el nombre de usuario y servir la página principal de la aplicación, para esto, ejecutaremos *render* con lo que cargaremos la plantilla, y antes de devolverlo como resultado del método, insertamos la cookie mediante *.set_cookie()*.

```
def indexregistrado(request):
    client_id='bd2ade5d39bb1b529fb7'
    client_secret='35b46ffbac1f02ea2ca84f44d2450fd00ffd6f40'
    codigo = request.GET.get('code')
    url = 'https://github.com/login/oauth/access_token'
    header = {'content-type':'application/json', 'Accept': 'application/json'}
    payload = {}
    payload['client_id']=client_id
    payload['client_secret']=client_secret
    payload['code']=codigo
    res = requests.post(
        url,
        data = json.dumps(payload),
        headers=header)
    salidaaux=json.loads(res.content)

    header2 = {'content-type':'application/json', 'Accept': 'application/json','Authorization': 'token '+salidaaux['access_token']+''}
    url2='https://api.github.com/user'
    payload2={}
    payload2['access_token']=salidaaux['access_token']
    res2 = requests.get(
        url2,
        headers=header2)
    salidaaux2=json.loads(res2.content)
    access_token=salidaaux['access_token']
    login=salidaaux2['login']
    ##############################
    nuevoUsuario = usuariosapp(
        usuario=login,
        token=access_token,)
    es = Elasticsearch()
    req = elasticsearch_dsl.Search(using=es, index='usuariosapp')#, doc_type='summary')
    resp = req.scan()
    try:
        for x in resp:
            if x['usuario']==login:
                entrada = usuariosAppIndex.get(id=x.meta.id, using=es, index='usuariosapp')
                entrada.update(using=es, token=access_token)
            else:
                nuevoUsuario.indexing()
    except TransportError:
        nuevoUsuario.indexing()
    ##############################
    respuesta = render(request, 'indexregistrado.html')
    respuesta.set_cookie('login',login)
    return respuesta
```

Figura 4.35: Código correspondiente al método para mostrar la pantalla principal.

- **Añadir una tarea:** para añadir una tarea, el método debe comprobar que el tipo de petición HTTP es POST, si lo es, debe crear un objeto para almacenar la información y comprobar que es válido. A continuación, antes de enviar la información a Elasticsearch, se debe incluir quien añade la tarea, y para eso utilizamos la cookie. Como antes de enviar la información vamos hacer una modificación de ésta, debemos guardar una vez, con el flag *commit=false*, para que no se guarde el modelo y así nos permita hacer la modificación, hacerla y por último guardar, para que con el proceso explicado en signals, la información quede almacenada en Elasticsearch. Si el tipo de petición no es POST, entonces se creará un objeto vacío y se volverá a llamar la página del formulario.

```
def add_tarea(request):
    if request.method == 'POST':
        form = tareasForm(request.POST)
        if form.is_valid():
            new_tarea = form.save(commit=False)
            new_tarea.creador = request.COOKIES['login']
            new_tarea.save()
            return HttpResponseRedirect(reverse('utareas:tlist'))
    else:
        form = tareasForm()


    return render(request, 'tarea_form.html', {'form': form})
```

Figura 4.36: Código correspondiente al método para añadir una tarea al sistema.

- **Lista general de tareas del sistema:** el usuario podrá consultar el listado de tareas que se han añadido al sistema, sea cual sea el usuario que la ha creado, así como si está ejecutada o pendiente de ejecutar. Para esto, debemos abrir una conexión con Elasticsearch, hacer una consulta, y convertir mediante el método *.execute()* la respuesta a un elemento iterable que pueda ser mostrado con la plantilla. Por último mediante *render()* serviremos la plantilla.

```
158 def lista_tareas(request):
159     es = Elasticsearch()
160     req = elasticsearch_dsl.Search(using=es, index='tareas')
161     resp = req.execute()
162 
163     return render(request, 'listatareas.html', {'object_list': resp})
```

Figura 4.37: Código correspondiente al método para ver el listado general de tareas del sistema.

- **Lista de repositorios de un usuario dado:** si un usuario conoce el nombre de un propietario pero no sus repositorio, la API de GitHub le ofrece funcionalidad. Para implementarla es necesario que la página principal muestre un formulario donde el usuario pueda introducir el nombre del propietario del que quiere conocer sus repositorios, por lo tanto, la primera acción será obtener del parámetro *request* el campo usuario que viene del POST. A continuación mediante la librería *Requests* formularemos una petición GET a la url que indica la API de GitHub, con el valor del campo usuario del POST incrustado en la url de petición, sin más parámetros. El resultado de esta petición a la API de GitHub devuelve, en formato JSON, una cantidad enorme de información de cada repositorio, por lo que tras transformarlo a un objeto entendible por Python mediante *.loads()* debemos seleccionar aquellos campos que nos interesa mostrar. Nuestra intención es devolver un JSON simplificado con estos campos y para ello crearemos una lista de diccionarios, que posteriormente transformaremos en JSON. Volvemos al objeto Python con la respuesta de la petición y comenzamos a iterarlo, almacenando en un diccionario en cada iteración los campos que queremos, para finalmente añadirlos a la lista. Una vez completada la iteración, mediante el método *.dumps()* convertimos la lista de diccionarios a JSON. El parámetro *indent=2* modelará la salida con tabulación para hacerla más vistosa. Por último devolvemos mediante render (en el caso de la versión web) o mediante HttpResponse (en el caso de la versión API) el resultado de la consulta.

```
115 def lista_tareas_usuario(request):
116     usuario=request.POST.get('usuario')
117     url='https://api.github.com/users/'+usuario+'/repos'
118     res = requests.get(
119         url,
120         )
121     salidaaux=json.loads(res.content)
122     lista=[]
123     for x in salidaaux:
124         nodo={}
125         nodo['propietario']=x['owner']['login']
126         nodo['nombre']=x['name']
127         nodo['nombre_completo']=x['full_name']
128         nodo['direccion_html']=x['html_url']
129         lista.append(nodo)
130     sal=json.dumps(lista,indent=2)
131 
132     return HttpResponse(sal, content_type='application/json')
```

Figura 4.38: Código correspondiente al método para ver el listado de repositorios de un usuario dado.

- **Listas de tareas ejecutadas/pendientes de un usuario:** otra funcionalidad que puede interesarle al usuario, es el estado de sus peticiones, es decir, de las tareas que ha solicitado al sistema, saber si están ejecutadas o pendientes de ejecutar. Hay dos formas de hacerlo, la primera es haciendo una consulta a Elasticsearch y obteniendo el índice por completo, para posteriormente iterarlo y mostrar aquellas partes que estén pendientes/ejecutadas. La segunda forma de hacerlo es con una consulta eficiente, es decir, solicitar a Elasticsearch sólo aquellas instancias que estén ejecutadas/pendientes, ahorrando consumo de datos y procesamiento de los mismos. Optamos por la segunda opción. Necesitamos hacer dicha consulta eficiente, para ello hacemos uso de las Query, incluidas dentro de la librería *elasticsearch_dsl*, éstas Query en nuestro caso serán múltiples ya que buscaremos las tareas del usuario que está en la aplicación y sólo las que están ejecutadas/pendientes. La Query requiere el campo de usuario que obtendremos de la cookie y el campo del estado que obtendremos según que recurso solicite el usuario de la aplicación. El resto es una consulta a Elasticsearch con el método *.query()* y usar el método *.execute()* para hacerlo compatible con el método *.dumps()* y crear el JSON que finalmente devolveremos. Por último devolvemos mediante render (en el caso de la versión web) o mediante HttpResponse (en el caso de la versión API) el resultado de la consulta.

```
140 def lista_tareas_ejecutadas(request):
141     q = Q('bool', must=[Q("match", creador=request.COOKIES['login']), Q("match", estado='true')])
142     es = Elasticsearch()
143     req = elasticsearch_dsl.Search(using=es, index='tareas').query(q)
144     resp = req.execute()
145     salida = json.dumps(resp.to_dict(), indent=2)
146 
147     return HttpResponse(salida, content_type='application/json')
148 
149 def lista_tareas_pendientes(request):
150     q = Q('bool', must=[Q("match", creador=request.COOKIES['login']), Q("match", estado='false')])
151     es = Elasticsearch()
152     req = elasticsearch_dsl.Search(using=es, index='tareas').query(q)
153     resp = req.execute()
154     salida = json.dumps(resp.to_dict(), indent=2)
155 
156     return HttpResponse(salida, content_type='application/json')
```

Figura 4.39: Código correspondiente a los métodos para ver el listado de tareas pendientes/ejecutadas de un usuario.

4.3.7. Archivos Django: search.py

El archivo search.py está creado especialmente para nuestro proyecto, es decir, no pertenece al conjunto de ficheros que por defecto se generan cuando se crea el proyecto Django. La creación de este fichero se hace con la intención de almacenar el código necesario para generar un objeto que posteriormente pueda ser indexado en Elasticsearch. Inicialmente además de las clases Python para poder indexar la información de tareas y usuarios en Elasticsearch, el archivo search.py contenía funcionalidad adicional, que en posteriores iteraciones se dejó de usar, o está en otra zona del proyecto. Para mayor comodidad y ante posible funcionalidad que pueda volver a ser alojada en este fichero, se ha decidido mantener como tal en el proyecto.

De igual forma que ocurre con la información que se obtiene del formulario HTML, necesitamos crear un objeto para poder indexar la información, este objeto será del tipo *DocType*,

tipo que se define en la librería *elasticsearch_dsl* para ayudar en el proceso de almacenamiento de la información. Este tipo *DocType*, tiene a su vez definidos los tipos generales de Python como *String()*, *Date()* o *Boolean()* haciéndolo compatible con el modelo Django. De esta forma se definen *tareasIndex* y *usuariosAppIndex* de igual manera que están definidos sus modelos en Django, gracias a la compatibilidad que ofrece *elasticsearch_dsl*.

```
1  from elasticsearch_dsl import DocType, Boolean, String, Date
2
3  connections.create_connection()
4
5  class usuariosAppIndex(DocType):
6      usuario = String()
7      token = String()
8
9  class tareasIndex(DocType):
10     usuario = String()
11     repositorio = String()
12     fechaRegistro = Date()
13     inicioEjecucion = String()
14     finEjecucion = Date()
15     estado = Boolean()
16     creador = String()
```

Figura 4.40: Código correspondiente al archivo search.py.

4.3.8. Construcción del dashboard: handler.py

En paralelo a la aplicación creada con Django nuestro proyecto necesita un servicio que se encargue de la revisión, ejecución y actualización de las tareas almacenadas en Elasticsearch.

Los pasos que seguirá el script para la creación del dashboard serán:

1. **Existencia de los ficheros JSON que sirven de base para la creación de los dashboard:** antes de comenzar el proceso de creación del dashboard, es necesario preparar todos los elementos necesarios para su ejecución. En primer lugar debemos tener preparados los archivos JSON que servirán como base para la creación de los dashboards. Estos ficheros contienen las visualizaciones que mostraremos en los dashboard, por medio de las cuales tomaremos los datos enriquecidos por p2o.py que están almacenados en Elasticsearch y mostrarlos según las visualizaciones del fichero en Kibana.

   Para realizar este proceso se debe comprobar, en primer lugar, si los ficheros existen en el directorio
   tmp, que será el directorio en el que trabajaremos, para esto, usaremos la librería *os*, concretamente el módulo *os.path*. La librería *os* es la encargada de las interacciones con el sistema operativo, como abrir y leer ficheros. En el módulo *os.path* existe un método que permite comprobar la existencia de un fichero, el método es *os.path.exists()*, que será el que usaremos.

   Si lo ficheros existen podemos continuar la ejecución, pero si no existen tendremos que descargarlos. Para ello usaremos *curl* con el flag *-o* que permite descargar ficheros, además tendremos que incluir la ruta donde queremos hacer la descarga y la ruta desde donde

queremos descargar el fichero. La forma de ejecutar *curl* será como se ha utilizado hasta ahora para otros comandos del sistema.

```
15
16      if os.path.exists("/tmp/git-dashboard.json"):
17          print("Existe")
18      else:
19          print("No existe")
20          cmdTmp="curl -o /tmp/git-dashboard.json https://raw.githubusercontent.com/jgbarah/GrimoireLab-training/master/grimoireelk/dashboards/git-dashboard.json"
21          cmdTmp=shlex.split(cmdTmp)
22          subprocess.call(cmdTmp)
23
24      if os.path.exists("/tmp/github-dashboard.json"):
25          print("Existe")
26      else:
27          print("No existe")
28          cmdTmp="curl -o /tmp/github-dashboard.json https://raw.githubusercontent.com/jgbarah/GrimoireLab-training/master/grimoireelk/dashboards/github-dashboard.json"
29          cmdTmp=shlex.split(cmdTmp)
30          subprocess.call(cmdTmp)
```

Figura 4.41: Código correspondiente a la parte de comprobación de la existencia de un fichero.

2. **Obtener las tareas pendientes de ejecutar:** a continuación necesitamos saber qué tareas de las almacenadas en el sistema están pendientes de ejecutar, para procesarlas. La forma de hacerlo es mediantes una consulta a Elasticsearch con las ya comentadas Querys, en este caso obtendremos solamente aquellas tareas en las que el campo estado tenga el valor *false*. Primero construimos la Query con *estado=false*, abrimos una conexión a Elasticsearch, ejecutamos la búsqueda mediante *.Search()* con la Query y finalmente *.execute()* nos devolverá un objeto que podamos iterar.

```
31
32      es = Elasticsearch()
33      q = Q("match", estado='false')
34      requ = elasticsearch_dsl.Search(using=es, index='tareas').query(q)
35      resp = requ.execute()
```

Figura 4.42: Código para hacer la consulta de tareas pendientes de ejecutar.

3. **Proceso de iteración:** el siguiente paso será realizar la iteración sobre el resultado de la consulta a Elasticsearch.

   a) **Obtener la información del usuario que creó la tarea:** para la creación del dashboard sobre GitHub, y para tener un ancho de banda adecuado, necesitamos el token de usuario de quien creó la tarea en el sistema. Para ello volvemos hacer una consulta a Elasticsearch, esta vez sobre el índice *usuariosapp*.

```
40          qitem = Q("match", usuario=commit.creador)
41          reqitem = elasticsearch_dsl.Search(using=es, index='usuariosapp').query(qitem)
42          respitem = reqitem.execute()
```

Figura 4.43: Código para la obtención de información del creador.

   b) **Actualización inicial de los campos:** cada instancia en Elasticsearch tiene campos relativos a fechas, campos que inicialmente se les dio un valor por defecto. El campo que se actualizará antes de empezar el procesamiento será *inicioEjecucion* para posteriormente generar las estadísticas.

Para la actualización usaremos una nueva funcionalidad que nos ofrece *DocType*, que como ya hemos dicho en anteriores apartados, es la clase que nos permite almacenar los datos en Elasticsearch. La clase *DocType* contiene el método *.get()* que es capaz de recuperar una instancia del índice a partir de su identificador, se diferencia de la Query en que sólo busca por el identificador, que es único para cada instancia, y no por el resto de campos, devolviendo por lo tanto una única instancia del índice. Como en nuestro caso el identificador es una concatenación del propietario y el nombre del repositorio, podemos hacer la búsqueda ya que conocemos ambos elementos. Una vez que obtenemos el nodo que buscamos, debemos actualizarlo, para ello hacemos uso del método *.update()* también dentro de *DocType*, al que le daremos como argumentos la conexión con Elasticsearch y la fecha actual asignada al campo *inicioEjecucion*.

```
45          req = tareasIndex.get(id=commit.usuario+"-"+commit.repositorio, using=es, index='tareas')
46          req.update(using=es, inicioEjecucion=datetime.now())
```

Figura 4.44: Código para la actualización de campos.

c) **Ejecución de scripts para la creación de los dashboards:** finalmente necesitaremos ejecutar los scripts *p2o.py* y *kidash.py* para crear los dashboards. Hablamos de dos dashboards ya que obtendremos uno con la información que ofrece Git (sin token de usuario) y otra con la información de GitHub (este con token de usuario). Por un lado ejecutaremos los scripts para la obtención del dahsboard para Git y por otro los mismos scripts, con algún parámetro diferente, para la obtención del dashboard para GitHub. La ejecución de estos scripts en Python se lleva a cabo mediante las librerías *shlex* y *subprocess*. La primera de ellas se encarga mediante el método *.split()* de trocear la cadena del comando en partes, más concretamente las partes que quedarían eliminando los espacios, ya que *subprocess* así lo requiere. Una vez preparada la cadena de texto que incluye el comando, pasamos a ejecutarlo, esto lo haremos mediante el método *.Popen()*, de *subprocess*. Este método mejora las prestaciones de *.call()*, ya que nos proporciona la capacidad de controlar la salida. En nuestro caso almacenaremos la salida del comando para controlar cuando termina, es decir, lanzaremos la ejecución de los dos scripts y esperaremos mediante *.wait()* a que ambos acaben para volver a ejecutar los dos siguientes. Tras la ejecución de los dos siguientes scripts volvemos a repetir el proceso de espera mediante *.wait()*. Antes de terminar la iteración del bucle debemos actualizar mediante el método *.update()* de *DocType* el campo *finEjecucion* con el momento actual.

Hay que señalar que los parámetros de ejecución de *p2o.py* requieren de la inclusión de variables correspondientes al nombre del propietario y al nombre del repositorio. En el caso de Git, es necesario construir una url que contenga ambas palabras, por ejemplo, *https://github.com/propietario/repositorio.git*, para ello crearemos la variable *repo_url* donde crearemos la url como la del ejemplo. Para el caso de GitHub, incluiremos las palabras dentro de los parámetros del comando.

```
repo_url = 'https://github.com/'+commit.usuario+'/'+commit.repositorio+'.git'
cmd = "p2o.py --enrich --index git_raw --index-enrich git \-e http://localhost:9200 --no_inc --debug \git "+repo_url+""
cmd = shlex.split(cmd)
p1 = subprocess.Popen(cmd)

cmd3 = "kidash.py --elastic_url-enrich http://locahost:9200 \--import /tmp/git-dashboard.json"
cmd3 = shlex.split(cmd3)
p2 = subprocess.Popen(cmd3)
p1.wait()
p2.wait()

cmd = "p2o.py --enrich --index github_raw --index-enrich github \-e http://localhost:9200 --no_inc --debug \github "+c.usuario+" "+c.repositorio+" \-t "+x.token+""
cmd = shlex.split(cmd)
p1 = subprocess.Popen(cmd)

cmd3 = "kidash.py --elastic_url-enrich http://locahost:9200 \--import /tmp/github-dashboard.json"
cmd3 = shlex.split(cmd3)
p2 = subprocess.Popen(cmd3)
p1.wait()
p2.wait()

req.update(using=es, estado=True, finEjecucion=datetime.now())
```

Figura 4.45: Código con la ejecución de los scripts.

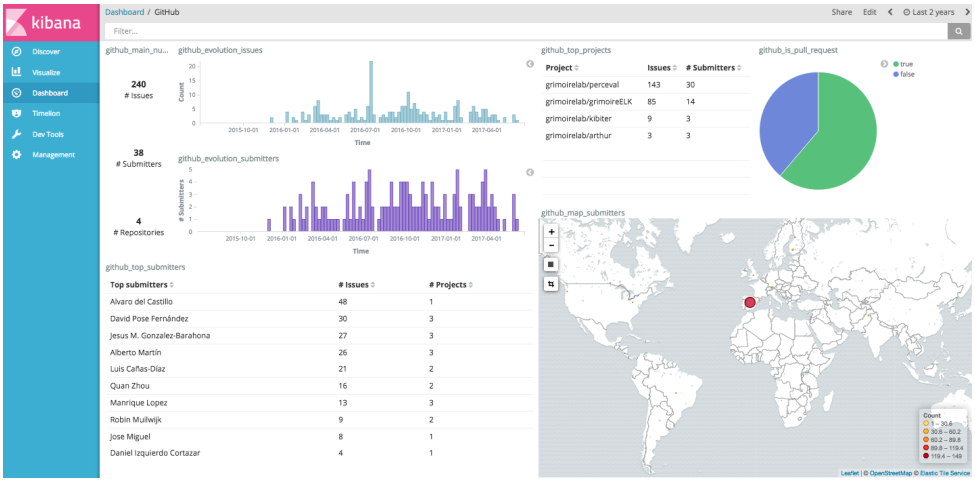


Figura 4.46: Resultado de la ejecución del script con el dashboard de GitHub.

4. **Bucles necesarios para la ejecución:** la ejecución de handler.py incluye varias iteraciones o bucles.

   El primero corresponde con un bucle infinito, ya que se trata de una aplicación de continua ejecución y por lo tanto es necesario estar permanentemente esperando nuevas tareas para ejecutar. Para no provocar un problema de eficiencia, tras cada ejecución el proceso se dormirá durante un tiempo determinado para no sobrecargar la máquina donde se ejecuta. Una vez que se despierta el script vuelve a repetir el proceso del que hemos hablado en este apartado, pudiendo encontrar tareas que ejecutar o no, en el caso de que no existan tareas que ejecutar el proceso pasará a dormirse otra vez, ya que su consulta a Elasticsearch no devolverá nada que pueda iterar.

   El siguiente bucle será el que se realiza con la respuesta a la consulta a Elasticsearch sobre las tareas pendientes, como ya hemos indicado, esta consulta puede estar vacía, por lo que el script se dormirá. En caso de que la respuesta tenga contenido entonces iteraremos sobre todas las tareas que contenga dicha respuesta, ejecutando el proceso anteriormente comentado.

   El último bucle corresponde a la consulta a Elasticsearch sobre la información del usuario que creó la tarea en el sistema, con el único fin de obtener su token de usuario. Este bucle sólo tendrá una única iteración ya que la respuesta a la consulta debe ser única.

```python
#! /usr/bin/env python3
import shlex, subprocess, os.path
from elasticsearch import Elasticsearch
import elasticsearch_dsl
from elasticsearch_dsl import DocType, String, Boolean, Q
from datetime import datetime
import time
class tareasIndex(DocType):
        usuario = String()
        repositorio = String()
        estado = Boolean()
while 1:
    if os.path.exists("/tmp/git-dashboard.json"):
        print("Existe")
    else:
        cmdTmp="curl -o /tmp/git-dashboard.json https://raw.githubusercontent.com/jgbarah/GrimoireLab-training/master/grimoireelk/dashboards/git-dashboard.json"
        cmdTmp=shlex.split(cmdTmp)
        subprocess.call(cmdTmp)
    if os.path.exists("/tmp/github-dashboard.json"):
        print("Existe")
    else:
        cmdTmp="curl -o /tmp/github-dashboard.json https://raw.githubusercontent.com/jgbarah/GrimoireLab-training/master/grimoireelk/dashboards/github-dashboard.json"
        cmdTmp=shlex.split(cmdTmp)
        subprocess.call(cmdTmp)

    es = Elasticsearch()
    q = Q("match", estado='false')
    requ = elasticsearch_dsl.Search(using=es, index='tareas').query(q)
    resp = requ.execute()
    for commit in resp:
        qitem = Q("match", usuario=commit.creador)
        reqitem = elasticsearch_dsl.Search(using=es, index='usuariosapp').query(qitem)
        respitem = reqitem.execute()
        for x in respitem:
            req = tareasIndex.get(id=commit.usuario+"-"+commit.repositorio, using=es, index='tareas')
            req.update(using=es, inicioEjecucion=datetime.now())
            repo_url = 'https://github.com/'+commit.usuario+'/'+commit.repositorio+'.git'
            cmd = "p2o.py --enrich --index git_raw --index-enrich git \-e http://localhost:9200 --no_inc --debug \git "+repo_url+""
            cmd = shlex.split(cmd)
            p1 = subprocess.Popen(cmd)
            cmd3 = "kidash.py --elastic_url-enrich http://locahost:9200 \--import /tmp/git-dashboard.json"
            cmd3 = shlex.split(cmd3)
            p2 = subprocess.Popen(cmd3)
            p1.wait() p2.wait()
            cmd = "p2o.py --enrich --index github_raw --index-enrich github \-e http://localhost:9200 --no_inc --debug \github "+c.usuario+" "+.repo+" \-t "+x.token+""
            cmd = shlex.split(cmd)
            p1 = subprocess.Popen(cmd)
            cmd3 = "kidash.py --elastic_url-enrich http://locahost:9200 \--import /tmp/github-dashboard.json"
            cmd3 = shlex.split(cmd3)
            p2 = subprocess.Popen(cmd3)
            p1.wait() p2.wait()
            req.update(using=es, estado=True, finEjecucion=datetime.now())
    time.sleep(2*60)
```

Figura 4.47: Código correspondiente al archivo handler.py.

Capítulo 5

Conclusiones

La idea inicial del proyecto era la de generar una interfaz web para la creación de un dashboard a partir de la herramienta Perceval, es decir, dar el soporte necesario a un usuario sin conocimientos avanzados de informática, para que a partir de la información del propietario del repositorio y del nombre de éste, crear un dashboard de manera transparente para el usuario. Cuando decimos de manera transparente nos referimos a que el usuario proporciona la información dicha y se le devuelve un enlace al dashboard ya completado y listo para su visionado.

Por lo tanto, después del proceso de elaboración del proyecto, podemos confirmar que se han alcanzado los objetivos propuestos inicialmente, obteniendo una interfaz que proporciona al usuario un dashboard, sólo solicitándole la información del propietario y el nombre del repositorio. Además también se entrega una API para poder integrar el software en otros desarrollos posteriores.

5.1. Consecución de objetivos

Como hemos señalado al inicio del capítulo tras la elaboración del proyecto se ha obtenido un producto capaz de satisfacer los requisitos previos. Al inicio de la elaboración se plantearon una serie de objetivos, veamos como se han superado:

1. **Integración con el sistema de almacenamiento:** la integración con el sistema de almacenamiento elegido, Elasticsearch, ha sido total. Se ha conseguido que la información proporcionada por el usuario, así como la información generada y obtenida por el sistema, quede almacenada en Elasticsearch. Además también se ha conseguido integrar toda la funcionalidad para consultar de forma eficiente o actualizar la información que está almacenada.

2. **Integración con GitHub:** se ha tratado de la fase más compleja del proyecto, ya que para poder integrar la funcionalidad que ofrece la API de GitHub para autenticar a un usuario a través de la plataforma, con Django, ha sido necesario buscar software externo ya que Django no dispone de forma predefinida de dicha funcionalidad.

   Tras buscar información acerca de varias librerías, finalmente decidimos usar *Python Social Auth*, que proporciona integración con Django, con GitHub y con el protocolo de autenticación Oauth2 que implementa GitHub. Por lo tanto en esta fase ha sido necesario relacionar Django, la librería *Python Social Auth*, Elasticsearch y la API de GitHub, con

la dificultad que conlleva hacer que interaccionen varias tecnologías entre ellas. El principal obstáculo ha sido la instalación y posterior configuración de la librería *Python Social Auth* en Django, para conseguir que nuestra aplicación ofrezca al usuario un proceso de autenticación a través de GitHub. Una vez que se ha configurado, el resto de interacciones con la API de GitHub han sido más llevaderas. Por lo tanto, con algo más de trabajo que en otras parte del desarrollo, finalmente se ha conseguido el objetivo.

3. **Funcionalidad útil para el usuario:** una vez que somos capaces de almacenar y modificar información en Elasticsearch, y que podemos autenticar al usuario a través de GitHub, así como obtener información de la API de GitHub como el token de usuario, es necesario que el proyecto, además del objetivo final de la creación del dashboard, proporcione información al usuario del estado de las tareas que le ha solicitado, si están pendientes de ejecutar, están ejecutadas, el listado de tareas general del sistema o el nombre de los repositorios de un usuario de GitHub conocido. Además también se ofrecen una serie de parámetros estadísticos sobre la ejecución de las tareas, como el tiempo de ejecución o la antigüedad en el sistema.

4. **Creación de una API:** como resultado final del producto se ofrecen dos versiones, en este caso, se plantea una API que sirva toda la funcionalidad del sistema, facilitando la integración con futuros desarrollos de software. A medida que avanza el diseño se determina que el mejor formato para que la API sirva los resultados, debido a las tecnologías que intervienen en el diseño, es el formato JSON, ya que es el que utiliza Elasticsearch y el que devuelve la API de GitHub. La implementación de esta parte del código es casi un paso previo al de construir una interfaz web para el usuario, de hecho, la interfaz web se basará en esta API para mostrar la información.

5. **Interfaz de usuario:** que como ya hemos comentado parte de la versión API para mostrar la información. En esta fase se han realizado dos implementaciones, la primera ha coincidido con las primeras cuatro iteraciones del desarrollo, tratándose de una interfaz extremadamente sencilla, que principalmente servía como plataforma para mostrar los resultados de cada nueva funcionalidad, añadir tareas al sistema o mostrar el contenido del índice almacenado en Elasticsearch. El segundo desarrollo, que se ha llevado a cabo en la última iteración del desarrollo, se trata de una interfaz para la mejora de la experiencia del usuario, con una interfaz agradable e intuitiva. Durante el planteamiento inicial del proyecto, esta fase estaba incluida, si bien es cierto que la interfaz que se proporciona es agradable e intuitiva pero básica, es decir, de baja complejidad. Como fases opcionales del proyecto se planteó inicialmente, un desarrollo de la interfaz web de usuario de forma dinámica, incluyendo Javascript, esta fase opcional no ha podido ser implementada por falta de tiempo, por que finalmente se ha seleccionado Bootstrap para el desarrollo, que ayuda a la creación de una buena interfaz base.

5.2. Aplicación de lo aprendido

La carrera ha servido como base para la elaboración de este proyecto, es decir, para poder desarrollar este proyecto de fin de carrera se han adquirido unos conocimientos previos en una serie de asignaturas a lo largo de la carrera, enumeradas en orden de importancia:

1. **Servicios y aplicaciones telemáticas (SAT):** se trata de una de las principales asignaturas relacionadas con el proyecto, ya que en ella de desarrolló una aplicación web de gestión de noticias RSS, con el framework Django.

2. **Sitemas telemáticos I (STI) y Sistemas telemáticos II (STII):** sirvieron para conocer lo necesario sobre el protocolo HTTP, TCP o UDP. En la primera de ellas se desarrolló un sistema de intercambio de ficheros.

3. **Estructura de datos y de la información (EDI):** asignatura correspondiente a la titulación de ingeniería técnica en informática de sistemas, que aporta un extenso conocimiento acerca de las diferentes estructuras de datos, como pilas, colas o listas, que forman parte de cualquier lenguaje y metodología de programación.

4. **Diseño de bases de datos y seguridad de la información (DBBDD):** asignatura correspondiente a la titulación de ingeniería técnica en informática de sistemas, incluyendo conocimientos acerca de las bases de datos y la forma de trabajar con ellas. Aunque las prácticas se realizaron con SQL los fundamentos teóricos fueron importantes.

5. **Fundamentos de sistemas operativos (FSO) y Sistemas operativos (SSOO):** han aportado los conocimientos necesarios en cuanto a la forma de ejecutar que tiene el sistema operativo, programación con hilos y concurrencia.

5.3. Lecciones aprendidas

En la sección anterior se enumeraban las asignaturas que han proporcionado de alguna forma conocimientos previos para facilitar el desarrollo del proyecto. Es el momento de enumerar lo aprendido o reforzado con el desarrollo:

1. **Python:** profundización en el lenguaje de programación Python, que ya se había usado durante la carrera, pero que se ha reforzado con el conocimiento sobre el manejo de información en formato JSON, el uso del protocolo HTTP mediante la librería *Requests*, así como el manejo de de información almacenada en Elasticsearch mediante la librería *elasticsearch_dsl*.

2. **GitHub:** el conocimiento sobre la plataforma GitHub era casi nulo, por lo que siendo una de las partes más importantes del proyecto, el conocimiento sobre esta tecnología era totalmente imprescindible. Se han adquirido conocimientos sobre la forma de estructurar los proyectos, los conceptos más relevantes (branch, fork, pull_request o issue), el registro de aplicaciones mediante el protocolo Oauth y lo más importante el manejo de su API para interaccionar con la plataforma desde nuestro proyecto.

3. **JSON:** este formato de gestión de la información era desconocido hasta el inicio del proyecto, por lo que se partía de cero con su uso. Se han adquirido conocimientos en base a la estructura y la forma de almacenar la información, al manejo del formato en Python, así como su relación con Elasticsearch, del que participa como formato de almacenamiento de la información.

4. **Elasticsearch:** el conocimiento previo sobre Elasticsearch era nulo, por lo que también se partía desde cero en el uso de esta tecnología. Se han adquirido los conocimientos necesarios para el manejo de la información que se almacena mediante Python, así como la forma de interaccionar con el servidor que almacena la información vía HTTP y la forma que tiene de almacenarla.

5. **Kibana:** al igual que con las anteriores tecnologías, Kibana era desconocido, por lo que se ha partido de cero en su conocimiento. Se han adquirido conocimientos básicos en cuanto su instalación y manejo básico para la representación de datos almacenados en Elasticsearch.

6. **Bootstrap:** partiendo de cero, se han adquirido conocimientos para el desarrollo de una interfaz básica escrita en HTML y con el complemento de estilo aportado por CSS, con una interacción mínima en la parte de Javascript.

7. **LaTeX:** sistema de composición de textos usado para la creación de la memoria, que se desconocía por completo, se han adquirido los conocimientos necesarios para descartar un procesador de texto de forma casi inmediata. Esto conocimientos tienen que ver con la inclusión de imágenes, la forma de estructurar el documento o de darle un formato determinado.

5.4. Trabajos futuros

Una de las conclusiones derivadas de la realización del proyecto ha sido la falta de tiempo para llevar a cabo todas las ideas que se iban ocurriendo a medida que se avanzaba en la elaboración del mismo. Pero para poder entregar en tiempo y forma se ha decidido acotar el proyecto a una serie de objetivos determinados. Por lo tanto vamos a enumerar una relación de futuros trabajos que pueden realizarse para mejorar-extender la funcionalidad del proyecto:

1. **Protección frente a errores, robustez:** mejorar la protección y recuperación frente a los posibles errores que puedan producirse, ya sea por motivos de mala ejecución, caídas de los diferentes servidores que están en juego (Elasticsearch, Django o Kibana). Por ejemplo podría implementarse un sistema que se encargue de comprobar el funcionamiento de los esos servidores, y en caso de fallo, reiniciar su ejecución de forma que el usuario no perciba estas caídas. Otra forma de añadir robustez, es probar las diferentes posibilidad de mal uso, es decir, aquellas situaciones excepcionales que se salen del flujo habitual del sistema, para que su respuesta sea coherente además de que permita al usuario poder seguir usando la aplicación.

2. **Desarrollo de una interfaz de usuario dinámica:** como ya hemos comentado anteriormente, una de las fases opcionales que se habían propuesto al inicio del diseño del proyecto, era la creación de una interfaz dinámica, que permitiese al sistema mostrar la información de forma activa, es decir, actualizando la interfaz web a medida que el sistema generaba nueva información, sin que el usuario tuviese que solicitar un nuevo recurso, o el mismo en el que ya se encontraba. Un ejemplo claro de esta interfaz activa seria la representación del listado de tareas pendientes de ejecutar del sistema, pudiendo tener un

espacio en la página principal, refrescándose cada vez que el sistema completa una actualización, sin que sea necesario que el usuario vuelva a solicitar el recurso que muestra las tareas pendientes de ejecutar que tiene el sistema.

3. **Aumento de la funcionalidad para mejorar la experiencia de usuario:** el sistema incorpora funcionalidad adicional al objetivo principal de la creación de un dashboard, como mostrar las tareas pendientes, las tareas ejecutadas con algunas estadísticas o el listado de nombres de repositorios de un usuario/propietario determinado. Además de esto se puede implementar nueva funcionalidad relacionada con GitHub, como por ejemplo la creación de nuevos proyectos a través de la aplicación o la modificación de éstos. También puede crearse funcionalidad para la modificación de tareas ya creadas por parte del usuario.

4. **Ampliar las formas de visualización de los dashboard creados con Kibana:** los ficheros JSON que se utilizan como base para la creación de los dashboard tanto para Git como para GitHub, son los que proporciona el tutorial seguido en la primera iteración del proyecto, *GrimoireLab Training*. Por lo tanto pueden desarrollarse nuevos dashboard, o modificar los actuales para añadir/modificar nuevas visualizaciones. Los índices almacenados por Perceval en Elasticsearch preparados para su interpretación por medio de Kibana, contienen gran cantidad de información con la que crear nuevas visualizaciones.

5.5. Valoración personal

La elección inicial de la materia se realizó siempre pensando en un proyecto basado en programación, con gran interés por aplicaciones web o aplicaciones para sistemas operativos móviles.

Para mí el desarrollo del proyecto ha sido muy gratificante, ya que ha supuesto volver al mundo, en este caso, de la programación y el desarrollo de aplicaciones web. Aunque por otro lado, haber compaginado el desarrollo del proyecto con la jornada laboral, ha sido en algunos momentos, complicado, retrasando mucho su finalización, llegando incluso al punto de tener que cambiar de materia.

Todo esto ha vuelto a despertar en mí el interés por este mundo, ya que, actualmente trabajo en algo que nada tiene que ver ni con las telecomunicaciones, ni con la informática en general, pensando incluso en retomar el mundo laboral relacionado con la titulación.

Bibliografía

[1] Api de github. https://developer.github.com/v3/. Visitado el 28/06/2017.

[2] Mark Lutz. *Learning Python: Powerful Object-Oriented Programming*. .º'Reilly Media, Inc.", 2013.

[3] Librería estándar de python. https://docs.python.org/3/library/index.html. Visitado el 28/06/2017.

[4] Documentación de django. https://docs.djangoproject.com/en/1.11/. Visitado el 28/06/2017.

[5] Clinton Gormley and Zachary Tong. *Elasticsearch: The Definitive Guide: A Distributed Real-Time Search and Analytics Engine*. .º'Reilly Media, Inc.", 2015.

[6] Guia de usuario de kibana. https://www.elastic.co/guide/en/kibana/current/index.html. Visitado el 28/06/2017.

[7] Librería python para el manejo de json. https://docs.python.org/3.6/library/json.html. Visitado el 28/06/2017.

[8] Documentación de la librería python social auth. http://python-social-auth.readthedocs.io/en/latest/. Visitado el 28/06/2017.

[9] Documentación de la librería python requests. http://docs.python-requests.org/en/master/. Visitado el 28/06/2017.

[10] Documentación de css. https://developer.mozilla.org/en-US/docs/Web/CSS/Reference. Visitado el 18/06/2017.

[11] Tutorial sobre django. http://librosweb.es/libro/django_1_0/. Visitado el 19/06/2017.

[12] Adam Wattis. Tutorial sobre la integración de django y elasticsearch, mediante la librería elasticsearch_dsl. https://medium.freecodecamp.com/elasticsearch-with-django-the-easy-way-909375bc16cb. Visitado el 09/05/2017.

[13] Documentación de la librería python elascticsearch_dsl. http://elasticsearch-dsl.readthedocs.io/en/latest/. Visitado el 20/06/2017.

[14] Vitor Freitas. Tutorial sobre el uso de python social auth y su integración tanto con github como con django. https://simpleisbetterthancomplex.com/tutorial/2016/10/24/how-to-add-social-login-to-django.html. Visitado el 12/06/2017.

[15] Documentación de la librería para python subprocess. https://docs.python.org/3/library/subprocess.html#module-subprocess. Visitado el 28/06/2017.

[16] Instalación y configuración de bootstrap. http://getbootstrap.com/getting-started/. Visitado el 28/06/2017.