



ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE
TELECOMUNICACIÓN

MÁSTER UNIVERSITARIO EN INGENIERÍA DE
TELECOMUNICACIÓN

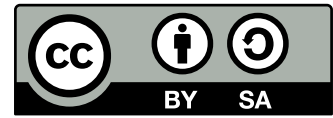
TRABAJO FIN DE MÁSTER

**GRIMOIREBOTS: SISTEMA SAAS PARA EL ANÁLISIS
DE PROYECTOS SOFTWARE**

Autor: Sergio Merino Hernández
Tutor: Dr. Jesus M. González-Barahona

Curso académico 2022/2023

Esta obra está bajo una licencia Creative Commons
«Atribución-CompartirIgual 4.0 Internacional».



Resumen

El desarrollo de *software* es una disciplina que ha sufrido una gran evolución en los últimos años. Hoy en día, esta materia no está ligada única y exclusivamente a lo que su nombre sugiere, sino que abarca una infinidad de actividades diferentes que conforman un ecosistema cada vez más grande y complejo. Estudiar estos entornos se ha convertido en una auténtica ciencia, donde la analítica de los datos asociados a ella posibilita un mayor entendimiento del desarrollo de *software*.

Para este fin, existen herramientas como GrimoireLab que permite la obtención automática de estos datos. Sin embargo, su uso no es tan sencillo, y esto supone en ocasiones una limitación para perfiles no técnicos.

En este contexto, este Trabajo de Fin de Máster se centra en el desarrollo de un sistema que facilite el uso de GrimoireLab, primero mediante el estudio e investigación de herramientas ya existentes para este fin, luego mediante el diseño e implementación de un sistema automatizado de obtención de datos referentes al desarrollo de *software*, y finalmente mediante la integración de este sistema con herramientas de visualización de datos para la construcción de métricas y gráficas que ayuden a comprender mejor este ecosistema.

Índice general

Resumen	III
1. Introducción	1
1.1. Motivación	1
1.2. Estructura de la memoria	1
1.3. Objetivos	2
1.3.1. Objetivo general	2
1.3.2. Objetivos específicos	2
2. Tecnologías utilizadas	5
2.1. Python	5
2.1.1. Poetry	5
2.2. HTTP	6
2.2.1. API REST	6
2.2.2. Postman	7
2.3. Django	7
2.3.1. Django REST Framework	8
2.3.2. Gunicorn	8
2.4. GrimoireLab	9
2.4.1. Cauldron.io	10
2.5. FastAPI	11
2.5.1. Uvicorn	12
2.6. OpenSearch	13
2.6.1. OpenSearch Dashboards	13
2.7. PostgreSQL	14
2.8. Docker	15
2.9. GitHub	16
2.9.1. Dependabot	17

3. Desarrollo del Proyecto	19
3.1. Arquitectura del software	19
3.1.1. Sprint 0: Etapa de análisis	19
3.1.2. Sprint 1: Grimoirebots I	21
3.1.3. Sprint 2: Grimoirebots II	24
3.2. Problemas encontrados	27
3.3. Tiempo dedicado	28
4. Resultados	29
4.1. Descripción técnica del sistema	29
4.1.1. Definición y funcionalidades	29
4.1.2. Modelos de datos	29
4.1.3. Descripción de la API	31
4.1.4. Entradas y salidas del sistema	34
Entradas del sistema	34
Salidas del sistema	35
4.1.5. Cliente automático	35
4.1.6. Flujos de ejecución	36
Flujo de ejecución del usuario	36
Flujo de ejecución del cliente	36
4.2. Despliegue	37
4.2.1. Requisitos	37
4.2.2. Configuración	38
4.2.3. Despliegue de componentes	40
4.3. Ejemplo de uso	41
4.3.1. Creación de una petición de análisis	41
4.3.2. Visualización de datos en OpenSearch	45
5. Conclusiones	51
5.1. Consecución de objetivos	51
5.2. Conocimientos adquiridos	51
5.3. Líneas de desarrollo futuras	52
Bibliografía	55

Índice de figuras

2.1. Python (Logo)	5
2.2. Postman (Logo)	7
2.3. Django (Logo)	7
2.4. DRF (Logo)	8
2.5. Gunicorn (Logo)	9
2.6. GrimoireLab (Logo)	9
2.7. GrimoireLab (Esquema)	10
2.8. Cauldron (Logo)	10
2.9. Cauldron (Gráficas)	11
2.10. FastAPI (Logo)	12
2.11. Uvicorn (Logo)	12
2.12. OpenSearch (Logo)	13
2.13. OpenSearch Dashboards	14
2.14. PostgreSQL (Logo)	14
2.15. Docker (<i>Workflow</i>)	15
2.16. Octocat	16
2.17. Dependabot (Logo)	17
3.1. Cauldron (Métricas)	21
3.2. Grimoirebots I (modelos)	21
3.3. Grimoirebots I (API)	22
3.4. Grimoirebots (<i>Frontend</i>)	23
3.5. Grimoirebots I (<i>Backend</i>)	24
3.6. Grimoirebots II (modelos)	25
3.7. Grimoirebots II (API)	25
3.8. Grimoirebots II (<i>Backend</i>)	26
4.1. Grimoirebots (modelos)	30

4.2. Grimoirebots API (orders)	31
4.3. Grimoirebots API (reports)	33
4.4. Grimoirebots (flujo del usuario)	36
4.5. Grimoirebots (flujo del cliente)	37
4.6. OpenSearch Dashboards (<i>Login</i>)	41
4.7. Grimoirebots (Creación de análisis)	42
4.8. Grimoirebots (Respuesta de creación de análisis)	42
4.9. Grimoirebots (Solicitud de análisis)	43
4.10. Análisis en Grimoirebots (Fichero de repositorios)	43
4.11. Análisis en Grimoirebots (Fichero de configuración)	44
4.12. Grimoirebots (Informe)	45
4.13. OpenSearch (Selección de <i>tenant</i>)	46
4.14. OpenSearch (Índices)	47
4.15. OpenSearch (Creación de <i>Index Pattern</i>)	48
4.16. OpenSearch (Creación de <i>Index Pattern</i>) Cont.	48
4.17. OpenSearch (Sección <i>Discover</i>)	49

Lista de Abreviaciones

API	Application Programming Interface
ASGI	Asynchronous Server Gateway Interface
AWS	Amazon Web Services
CHAOSS	Community Health Analytics in Open Source Software
CI / CD	Continuous Integration / Continuous Deployment
DRF	Django REST Framework
HTTP	Hypertext Transfer Protocol
IP	Internet Protocol
JSON	JavaScript Object Notation
LDAP	Lightweight Directory Access Protocol
PEP	Python Enhancement Proposal
PSFL	Python Software Foundation License
REST	Representational State Transfer
SaaS	Software as a Service
SDK	Software Development Kit
SQL	Structured Query Language
URL	Uniform Resource Locators
UTC	Universal Time Coordinated
UUID	Universally Unique Identifier
WSGI	Web Server Gateway Interface
YAML	Yet Another Markup Language

1 Introducción

1.1. Motivación

Es de sobra conocida la importancia de la analítica de datos en nuestro tiempo, así también lo es la referida al desarrollo de *software*. Herramientas como GrimoireLab posibilitan este fin.

Para acercar su uso a perfiles no técnicos, los creadores de GrimoireLab idearon Cauldron, un SaaS de GrimoireLab que funciona como una aplicación web. Con un par de clics, un usuario es capaz de poner en marcha el sistema y obtener datos referidos al desarrollo de *software* fácilmente.

Sin embargo, esta plataforma no está exenta de defectos. Como parte del equipo desarrollador de Cauldron, el autor de este Trabajo de Fin de Máster conoce de primera mano los puntos de mejora del sistema.

Con este pretexto comienza el desarrollo de una herramienta que facilite la analítica de datos referente al desarrollo de *software*, soslayando las dificultades encontradas durante el desarrollo de Cauldron.

1.2. Estructura de la memoria

La estructura que esta memoria sigue es la siguiente:

- En el Capítulo 2 se explican varios conceptos y tecnologías que ya existen y que se han usado para llevar a cabo el proyecto.
- En el Capítulo 3 se detalla el desarrollo del proyecto, desde el trabajo previo al comienzo del proyecto hasta los problemas encontrados durante el desarrollo, pasando por los diferentes prototipos de la aplicación.

- En el Capítulo 4 se exponen las especificaciones del sistema final, incluyendo instrucciones detalladas para poner la aplicación en funcionamiento, así como un caso de uso.
- En el Capítulo 5 se finaliza la memoria haciendo una reflexión sobre los resultados y las posibles líneas de desarrollo futuras.

1.3. Objetivos

1.3.1. Objetivo general

El objetivo principal de este proyecto es el desarrollo de una plataforma similar a Cauldron, que permita el análisis de ecosistemas de desarrollo de *software*, modificando algunas partes de su funcionamiento original, añadiendo nuevas características, y resolviendo varios problemas que el proyecto original llevaba arrastrando desde su comienzo.

1.3.2. Objetivos específicos

Para llevar a cabo el objetivo principal del proyecto, se han elaborado una serie de objetivos más específicos:

- El esquema de funcionamiento en Cauldron gira en torno a la generación de informes alterables, esto es, cuando un usuario analiza una o varias fuentes de datos es capaz de modificar (incluso cuando el análisis no ha finalizado) la lista de objetos a analizar. De la misma manera, un informe realizado hace meses continúa actualizando los datos mostrados si otro usuario (o el creador) lo solicita. Esta característica, si bien permitía no tener que analizar varias veces un mismo repositorio para tener datos actualizados, generaba bastantes problemas de desarrollo al tener que lidiar con múltiples factores que podían alterar los datos de un informe. Uno de los objetivos que persigue este proyecto es que la nueva plataforma genere informes inalterables, creando un flujo más simple desde la solicitud por parte del usuario al *output* de la aplicación.
- Muchos de los servicios que componen Cauldron tienen una fuerte dependencia entre ellos, de manera que el fallo en uno puede ocasionar el fallo general del

sistema. En un entorno en el que el desarrollo *serverless* y las arquitecturas basadas en micro servicios son cada vez más frecuentes, es necesaria la creación de sistemas desacoplados que funcionen en su conjunto para dar un servicio. Este proyecto persigue el desarrollo de varios componentes *software* independientes que colaboren entre ellos para dar un servicio al usuario.

- Cauldron utiliza GrimoireLab (más concretamente la herramienta conocida como SirMordred) para realizar las tareas de recogida y enriquecimiento de datos. La forma en la que Cauldron opera con la herramienta es mediante un contenedor Docker personalizado, el cual deriva de uno proporcionado por GrimoireLab, lo que añade más complejidad al incluir un componente extra que necesita ser mantenido. Este objetivo pretende simplificar el uso de GrimoireLab mediante la ejecución del contenedor Docker original de GrimoireLab. Aparte, otro de los componentes de GrimoireLab, llamado Sorting Hat, añade limitaciones al sistema (o complejidad, según se mire) al ser requisito inalterable para su funcionamiento el uso de una base de datos MariaDB. Debido a esto, y a que el RGPD¹ limita el valor que se le puede dar a una plataforma como esta, se ha decidido eliminar este componente del esquema de la nueva plataforma.
- El último objetivo que busca este proyecto está relacionado con mejoras menores respecto a Cauldron, como el uso de Poetry para la gestión de dependencias, la sustitución del servidor WSGI de Django por Unicorn, o la detección de actualizaciones de dependencias con Dependabot.

¹Reglamento General de Protección de Datos

2 Tecnologías utilizadas

2.1. Python

Python es un lenguaje de programación orientada a objetos de alto nivel. Fue desarrollado originalmente por Guido van Rossum en el final de la década de 1980, y actualmente es gestionado por la *Python Software Foundation* bajo una licencia *open source* propia conocida como PSFL (*Python Software Foundation License*). [25]

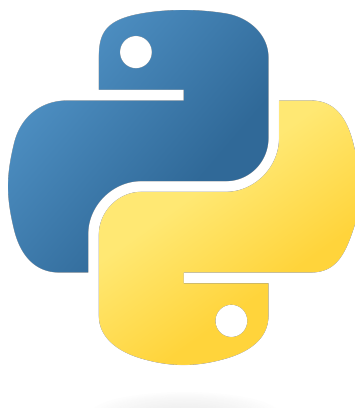


FIGURA 2.1: Logo de Python [30]

Python es actualmente uno de los lenguajes de programación más populares. Esto se debe en gran parte a su modularidad, la cual permite agregar fácilmente nuevas piezas de código a aplicaciones ya desarrolladas. [27]

2.1.1. Poetry

Poetry es un gestor de paquetes y dependencias para Python. Entre sus características más destacadas se encuentra la generación de un fichero con la especificación exacta de dependencias (incluidas las dependencias de dependencias), lo que asegura instalaciones idénticas en diferentes entornos. [18]

Además de esto, Poetry incorpora la mayoría de reglas establecidas en PEP¹, como el uso del fichero *pyproject.toml* para escribir la meta información de los paquetes, o el uso del esquema *major.minor.micro* para los identificadores de versión. [18]

2.2. HTTP

Hypertext Transfer Protocol (HTTP) es un protocolo de nivel de aplicación para sistemas de información distribuidos.

Es un protocolo genérico y sin estado que es usado para múltiples tareas como la transferencia de hipertexto o la representación de sistemas de ficheros.

Una característica de HTTP es la negociación de la representación de los datos, lo que permite construir sistemas independientemente de los datos que se vayan a transmitir. [10]

2.2.1. API REST

Una API (*Application Programming Interface*) es un contrato entre dos equipos sobre como deben comunicarse entre ellos. Con el uso de APIs se pretende abstraer los detalles acerca de como un sistema funciona internamente, haciendo públicas solo aquellas partes que los desarrolladores pudieran encontrar interesantes. De hecho, aunque la lógica interna cambiara, la API debería permanecer intacta. [23]

REST (*Representational State Transfer*) es un estilo de arquitectura *software* para sistemas distribuidos que operan con contenido de diverso tipo. La idea tras este estilo es que cada respuesta del servidor debe ser una representación de un recurso (comúnmente en formato JSON), siendo las llamadas al servidor solicitudes para la creación, actualización o eliminación de un recurso. [11]

Por tanto, una API REST sería aquella que sigue los principios arquitectónicos que le dan nombre.

¹PEP (*Python Enhancement Proposal*) recoge y describe las principales guías y recomendaciones de la comunidad respecto a nuevas características y procesos para cualquier desarrollo con Python.

2.2.2. Postman

Postman es una plataforma utilizada para simplificar el uso y creación de APIs, así como para facilitar la colaboración entre desarrolladores. Entre sus características más importantes se encuentra la posibilidad de construir colecciones de llamadas a APIs, de manera que el *testing* de estas se acelera. Además, es posible compartir y publicar estas colecciones como ficheros de texto, permitiendo una colaboración muy fluida entre diferentes personas. [20]



FIGURA 2.2: Logo de Postman [29]

2.3. Django

Django es un *web framework* desarrollado para Python y que, al igual que este, cuenta con una licencia *open source*. Fue desarrollado originalmente por Adrian Holovaty y Simon Willison, dos programadores web que trabajaban en un periódico. [1]



FIGURA 2.3: Logo de Django [6]

Django sigue los principios de modularidad que caracterizan a Python, además de incluir muchas otras características interesantes como un sistema de vistas, modelos y plantillas que permite agilizar el desarrollo de aplicaciones web. Esto, sumado a un servidor web simple para desarrollos locales, una interfaz administrativa intuitiva, y la posibilidad de construir *middleware* de todo tipo, convierten a Django en uno de los *frameworks* para desarrollo web más famosos y usados de los últimos años. [1]

2.3.1. Django REST Framework

Django REST Framework (DRF) es un módulo para Django que amplía las características de este, y que facilita la creación de APIs REST manteniendo todas las ventajas de Django. [9]



FIGURA 2.4: Logo de DRF [4]

A diferencia del *framework* original, DRF se centra principalmente en la parte *backend* que compone una aplicación web. Para ello, ofrece al usuario muchas más opciones para serializar la entrada y salida de datos, además de ampliar las conocidas *Generic Views* de Django con mejoras relacionadas con la construcción de APIs. [9]

Permite además interaccionar fácilmente con la API construida a través de una interfaz web básica, y es capaz de generar documentación dinámica para la API siguiendo la especificación de OpenAPI². [9]

2.3.2. Gunicorn

Gunicorn ("*Green Unicorn*") es un servidor HTTP, escrito en Python, que implementa WSGI³ como protocolo de llamadas.

Funciona utilizando un esquema centralizado, donde un nodo maestro gestiona y orquesta diferentes tipos de nodos trabajadores. Estos pueden ser utilizados para procesar peticiones síncronas o asíncronas, y su número escala en función de la demanda que tenga el servidor. [14]

²<https://spec.openapis.org/oas/v3.1.0>

³WSGI (*Web Server Gateway Interface*) establece cómo deben ser llevadas las peticiones del servidor web a la aplicación web escrita en Python, de manera síncrona. [24]



FIGURA 2.5: Logo de Gunicorn [28]

2.4. GrimoireLab

GrimoireLab es el resultado de años de investigación y análisis de comunidades *open source* y procesos de desarrollo. Desarrollado originalmente por miembros del equipo *LibreSoft* (que más tarde fundarían Bitergia), actualmente forma parte de CHAOSS, una iniciativa de la *Linux Foundation* centrada en crear y modelar métricas sobre comunidades *open source*. [2]



FIGURA 2.6: Logo de GrimoireLab [2]

Se trata de un conjunto de herramientas diseñadas para recoger, enriquecer, consumir y visualizar datos de casi cualquier fuente relacionada con el desarrollo *Open Source*.

Cada una de estas herramientas sirve a un propósito específico y, en su conjunto, proveen el servicio antes mencionado:

- Perceval es el encargado de recoger los datos crudos de cada fuente de datos compatible con GrimoireLab. Ya sea utilizando herramientas como Git o haciendo uso de la API de la fuente analizada.
- GrimoireELK enriquece los datos recogidos por Perceval. Genera, a partir de estos, nuevos datos que son de interés para estudiar las comunidades *open source*.
- Sorting Hat permite gestionar la información relacionada con identidades, incluso entre diferentes fuentes de datos.

- SirMordred es la herramienta orquestadora para GrimoireLab. Se encarga de sincronizar a todas las componentes para dar servicio.
- ¡Y muchas más herramientas increíbles!

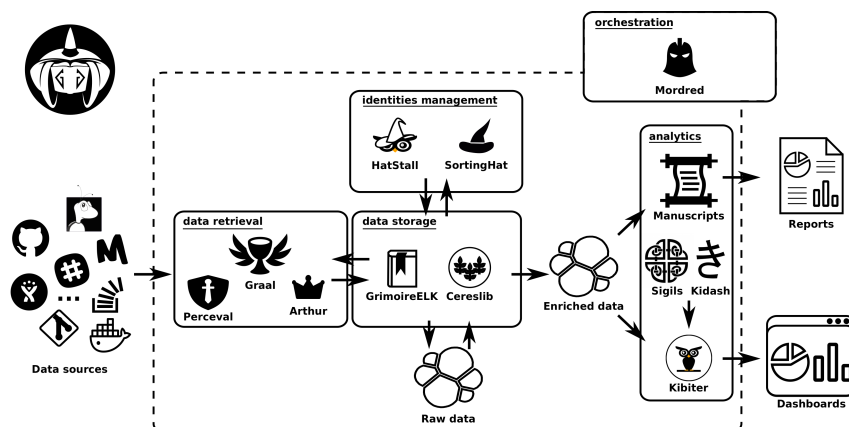


FIGURA 2.7: Esquema de GrimoireLab [2]

2.4.1. Cauldron.io

Cauldron.io (o simplemente Cauldron) es una plataforma web que permite a sus usuarios utilizar GrimoireLab de manera sencilla y rápida, necesitando únicamente un navegador web para su uso. [3]



FIGURA 2.8: Logo de Cauldron [3]

Cauldron nace como un SaaS⁴ de GrimoireLab, y actualmente tiene compatibilidad con algunas de las fuentes de datos más importantes dentro de los ecosistemas *open source*, como GitHub, Meetup, o StackExchange. Está compuesto por una aplicación Django detrás de un servidor Nginx, y utiliza Open Distro for Elasticsearch y Kibana para almacenar y visualizar los datos generados con GrimoireLab, respectivamente.

⁴Software as a Service

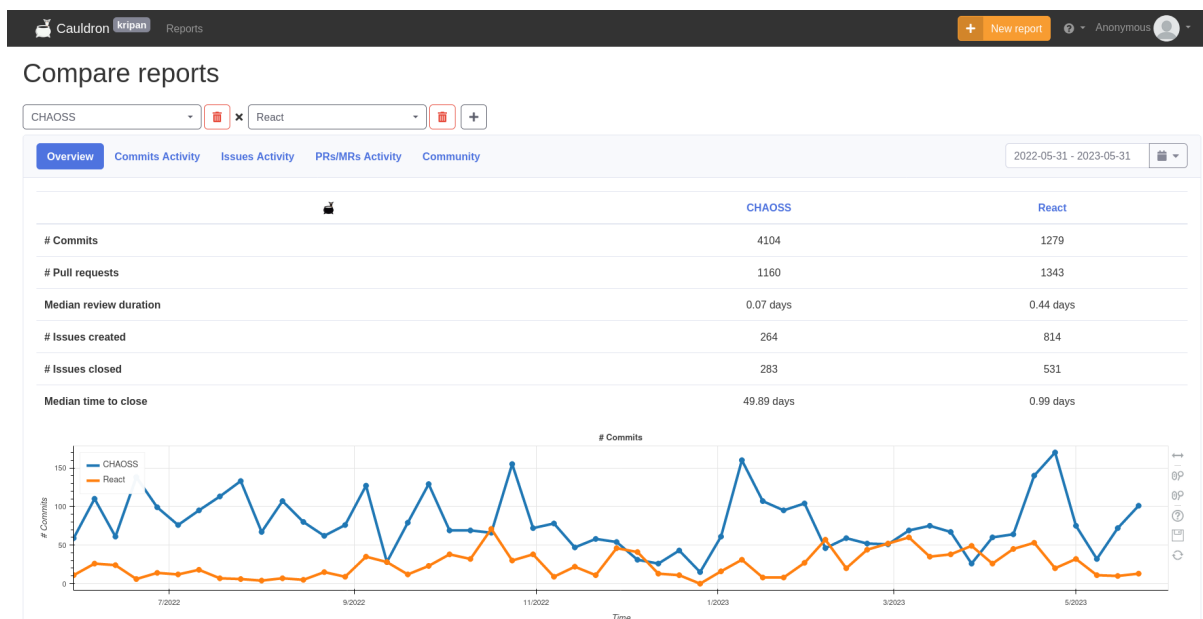


FIGURA 2.9: Gráficas en Cauldron [3]

Además de las visualizaciones generadas en Kibana, Cauldron ofrece a sus usuarios otras formas de dar valor a sus análisis. Como se observa en la Figura 2.9, permite comparar métricas y gráficas de diferentes proyectos. Además de esto, permite generar gráficas personalizadas dentro de Kibana gracias a las funcionalidades extra de Open Distro for Elasticsearch.

2.5. FastAPI

FastAPI es un *web framework* desarrollado para Python, orientado principalmente a la construcción de APIs utilizando las anotaciones de tipos estándar de Python. [21]

FastAPI nace después de que su autor no pudiera cubrir las necesidades de sus proyectos personales con los *web frameworks* actuales en ese momento. Después de investigar las especificaciones de OpenAPI, JSON, OAuth2, etc., decide construir su propia solución teniendo como base Pydantic⁵ y Starlette⁶. [22]

⁵Pydantic es una herramienta que permite aprovechar las anotaciones de Python para informar de errores en la validación de datos.

⁶Starlette es un framework que permite construir servicios web asíncronos en Python.



FIGURA 2.10: Logo de FastAPI [21]

FastAPI hace uso de las anotaciones de tipos en Python para conseguir *parsing* y validación de datos, documentación automática, y soporte en editores (comprobación de errores, auto completado, etc). Hace uso de ASGI⁷ para llamadas asíncronas por defecto, es compatible con la mayoría de bases de datos relacionales y no relacionales, y permite tener un servicio web funcional en pocos pasos. [21]

2.5.1. Uvicorn

Uvicorn es un servidor HTTP, para aplicaciones Python, que procesa las llamadas de manera asíncrona implementando ASGI. En combinación con Gunicorn, se obtiene un servidor asíncrono multi nodo. [26]



FIGURA 2.11: Logo de Uvicorn [26]

⁷ASGI (*Asynchronous Server Gateway Interface*) es similar a WSGI, pero para llamadas asíncronas.

2.6. OpenSearch

OpenSearch es un *suite* de herramientas *open source* que permite el almacenamiento, búsqueda, y análisis de datos. Se trata de un proyecto formado principalmente por dos componentes: el propio motor OpenSearch y OpenSearch Dashboards, siendo estos un *fork*⁸ de los proyectos Elasticsearch y Kibana, respectivamente. [15]



FIGURA 2.12: Logo de OpenSearch [15]

OpenSearch está basado en el motor de búsqueda Apache Lucene, lo que le confiere una gran capacidad de indexación en volúmenes de datos grandes. Muchas de las características que definen a OpenSearch provienen de plugins para Elasticsearch, pero que el primero integra de manera nativa. Entre estas características destacan:

- Un módulo avanzado de seguridad que permite la autenticación de usuarios mediante *Active Directory*, LDAP, y otros protocolos similares. Igualmente, permite mayor granularidad sobre los permisos en índices y otras estructuras de datos.
- Un módulo de gestión de índices, con el que definir políticas y automatismos en índices.
- Un analizador de eficiencia que es capaz de obtener diversas métricas del *cluster* de OpenSearch asociadas a la eficiencia, como la cantidad de memoria utilizada por los nodos, o el número de operaciones de lectura y escritura.
- Un sistema de alertas que permite establecer límites en gráficas y métricas, los cuales envían una notificación al ser rebasados.

2.6.1. OpenSearch Dashboards

OpenSearch Dashboards es la herramienta de visualización de datos de OpenSearch. Está basado en Kibana, tratándose de hecho de un *fork* de esta. [16]

⁸Uno de los principales motivos de su creación fue la decisión de modificar la licencia de Elasticsearch y Kibana a una no *open source* por parte de sus creadores.

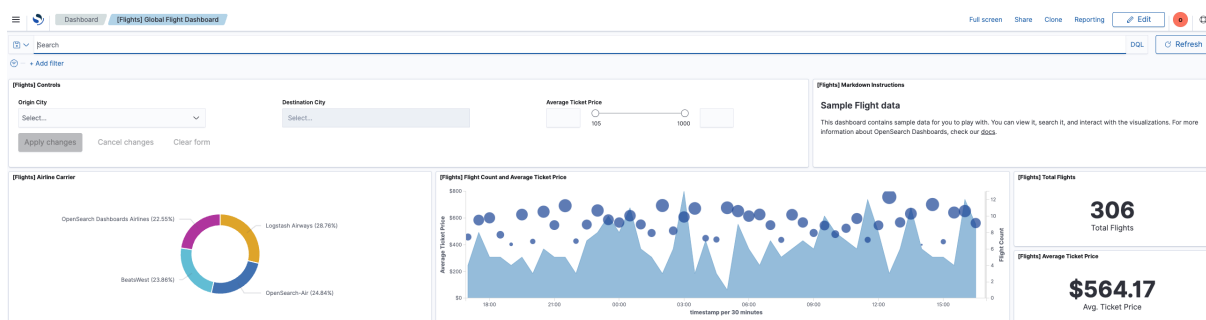


FIGURA 2.13: OpenSearch Dashboards [16]

Posee varias herramientas para interactuar con los datos almacenados en el *cluster* y crear, con estos, gráficos y métricas como los que aparecen en la Figura 2.13. Permite también la construcción de alias y patrones para índices, y la visualización de los datos directamente en formato JSON (tal y como están almacenados en el *cluster*).

2.7. PostgreSQL

PostgreSQL es un sistema de base de datos relacional con mucha historia (fue desarrollado originalmente en 1996). Cuenta con licencia *open source*, y está gestionado por una comunidad colaborativa de desarrolladores conocida como *PostgreSQL Global Development Group*. [19]

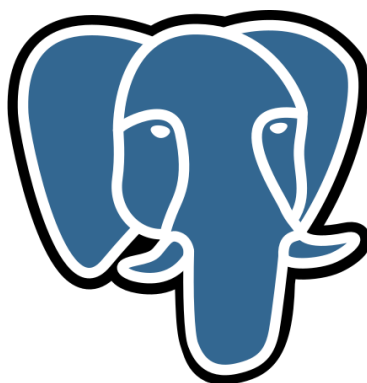


FIGURA 2.14: Logo de PostgreSQL [19]

PostgreSQL es la opción recomendada para Django. Esto es debido a las muchas características que lo hacen una opción interesante, como la diversidad de estructuras

nativas (*arrays*, direcciones IP, etc), su capacidad para alta concurrencia, o la posibilidad de ejecutar bloques de código en el servidor.

2.8. Docker

Docker es el sistema gestor de contenedores más utilizado actualmente, debido a la sencillez de su uso y a la cantidad de funcionalidades que ofrece. Docker se basa en un esquema cliente-servidor donde el cliente es utilizado por los usuarios para comunicarse con el demonio de Docker (el servidor), el cual se encarga de la construcción de contenedores, su ejecución, y su distribución. [7]

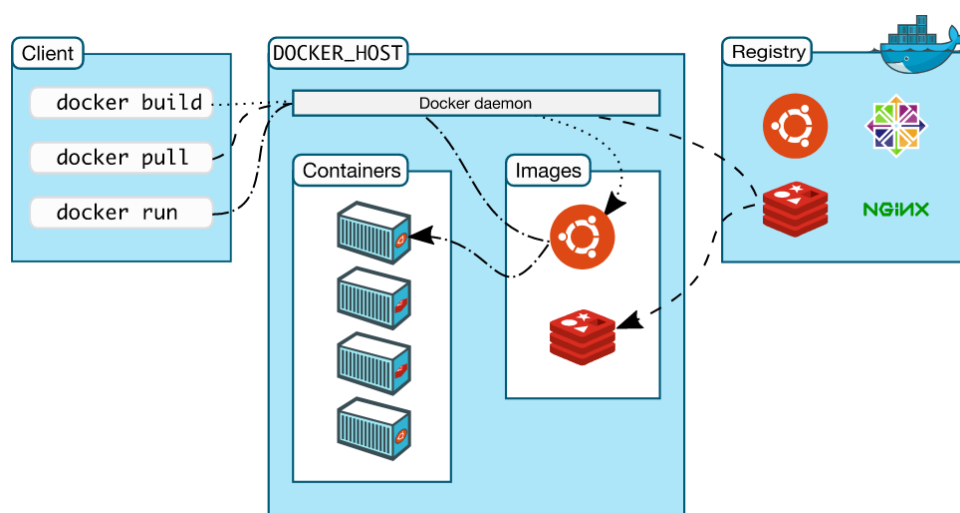


FIGURA 2.15: Flujo de trabajo en Docker [7]

Como se observa en la Figura 2.15, además del cliente y el demonio antes mencionados, en el despliegue y distribución de un contenedor en Docker intervienen varios elementos:

- **Imágenes:** Una imagen en Docker es una plantilla que detalla las instrucciones para la construcción de un contenedor. Un usuario tiene la posibilidad de crear sus propias imágenes o utilizar las que otros usuarios hayan publicado en un registro. Para el primer caso, un usuario que quiera generar su propia imagen deberá crear un fichero *Dockerfile* en el que se detallan las instrucciones (comandos) necesarias para la construcción del contenedor.

- **Contenedores:** Un contenedor, en Docker, es una instancia ejecutable de una imagen. Pueden ser creados, iniciados, parados, o eliminados utilizando el cliente de Docker. Se pueden conectar a una o más redes y se les puede agregar volúmenes para ampliar su espacio de almacenamiento.
- **Registros:** Un registro es un servidor donde los usuarios de Docker pueden distribuir sus imágenes, de manera pública o privada.

2.9. GitHub

GitHub es un servicio *online* que permite a los desarrolladores y a sus equipos llevar un control de versiones sobre sus proyectos *software*. Fue adquirido en 2018 por Microsoft y actualmente es la plataforma *online* con mayor cantidad de repositorios de código del mundo. [12]



FIGURA 2.16: Octocat (mascota de GitHub) [17]

GitHub es básicamente un SaaS de Git, pero con funcionalidades añadidas como la creación de *issues* para el seguimiento de errores, la solicitud de cambios a código (conocidas como *pull requests*), la inclusión de mecanismos automatizados (CI/CD), o la creación de *wikis*, entre otros. [13]

2.9.1. Dependabot

Dependabot es una herramienta que permite a los desarrolladores despreocuparse de actualizar las dependencias de sus proyectos *software*. Se encarga de analizar los ficheros de dependencias de un proyecto⁹ y buscar nuevas actualizaciones de estas, notificando al usuario y, en ocasiones, sugiriendo los cambios necesarios al código para su inclusión. [8]



FIGURA 2.17: Logo de Dependabot [5]

Dependabot fue adquirido por GitHub en 2019, lo que ha permitido una integración más estrecha con la plataforma. Mediante la creación de un fichero *dependabot.yml*, los desarrolladores pueden establecer qué dependencias analizar, la frecuencia de estos análisis, y la fuente desde donde se buscan actualizaciones. [8]

⁹Por ejemplo, en un proyecto Python estas dependencias están definidas en el fichero *requirements.txt* o *pyproject.toml*.

3 Desarrollo del Proyecto

3.1. Arquitectura del software

El objetivo de este proyecto es proporcionar un servicio que permita analizar proyectos *software* de una manera sencilla y automatizada. Para ello, se ha llevado a cabo el desarrollo de varios *sprints*, cada uno resolviendo los problemas descubiertos en el anterior:

- El primer *sprint* se centra en realizar un análisis de Cauldron.io, una herramienta ya existente para el análisis de ecosistemas de desarrollo de *software*.
- En el segundo *sprint* se recogen las lecciones aprendidas en el anterior y se centra en el desarrollo de un primer prototipo que sienta las bases de la arquitectura de la aplicación. En este prototipo, la API consta de algunas llamadas básicas para la creación de solicitudes de análisis y el cliente es capaz de procesarlas y retornar este resultado al usuario.
- El tercer y último *sprint* recoge el testigo del anterior y se centra en la integración de la API con GrimoireLab y OpenSearch. En este nuevo prototipo, la API incluye nuevas estructuras y llamadas y el cliente ejecuta tareas de análisis con GrimoireLab y sube los resultados a OpenSearch para su visionado.

3.1.1. Sprint 0: Etapa de análisis

Durante la primera etapa que comprende el desarrollo de este proyecto, se ha realizado un análisis de la herramienta Cauldron.io, de su proceso de desarrollo, funcionalidades y puntos de mejora, los cuales han servido como inspiración para las posteriores etapas. Se trata de una etapa de aprendizaje que sienta las bases del resto del desarrollo.

Cauldron.io nace con la idea de facilitar el uso de GrimoireLab, y permitir usar este a perfiles no técnicos. Es por ello que el desarrollo de este *SaaS* comienza como una aplicación web, utilizando Django como base y Elasticsearch y Kibana para el manejo de datos. Cauldron.io comienza en sus inicios con pocas fuentes de datos disponibles, únicamente Git, aunque muy temprano comienza a incluir otras muchas como GitHub, GitLab, o Meetup, entre otras.

Cauldron.io está ideado para generar informes modificables. Esto quiere decir que un usuario puede, en un momento dado, solicitar el análisis de ciertas fuentes de datos, e.g., repositorios Git, y una vez terminado el análisis de esta incluir una nueva fuente de datos, como Meetup. Este esquema, si bien dota a la plataforma de mayor versatilidad, acarrea varios problemas en su desarrollo y mantenimiento. Durante el desarrollo de esta característica se necesitó realizar múltiples iteraciones en los modelos de la aplicación para conseguir que solo se analizaran las nuevas fuentes de datos, y no las ya analizadas. Estas iteraciones han resultado en un esquema de modelos complejo que hace difícil su modificación.

Cauldron.io hace uso de un sistema de *workers* para realizar los análisis. Cada uno de estos *workers* es un contenedor personalizado de GrimoireLab, el cual utiliza el SDK de GrimoireLab para Python en la ejecución de los análisis. Estos *workers* se están ejecutando continuamente, y se establece su número al comienzo del despliegue del *cluster* de *workers*. Las tareas de análisis que estos *workers* realizan son gestionadas por un complejo componente personalizado conocido como *Cauldron Pool Scheduler*, el cual realiza una asignación de recursos de análisis equitativa entre todos los usuarios de la plataforma. A pesar de los buenos resultados que este sistema genera, añade una excesiva complejidad a la asignación de tareas, lo que dificulta realizar cualquier tipo de modificación o mantenimiento a este.

Además de las métricas y gráficas que Cauldron.io ofrece a través de Elasticsearch y Kibana, existen otras muchas en la página de cada análisis (Figura 3.1). Estas están desarrolladas usando la librería Boleh para JavaScript, la cual añade una excesiva complejidad al código al necesitar incluir grandes bloques a las plantillas de Django. Además de esto, debido a la gran cantidad de datos que algunos análisis manejan, en ocasiones la recogida de estos datos desde el *cluster* de Elasticsearch hace vencer un *timeout* establecido, lo que provoca el fallo de la página.

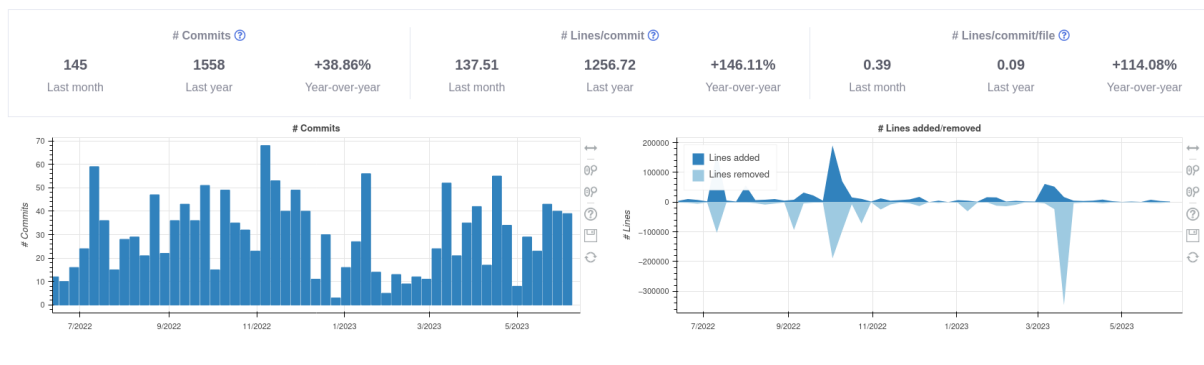


FIGURA 3.1: Métricas en Cauldron

Este análisis, sumado a la experiencia durante el desarrollo y mantenimiento de Cauldron.io han resultado en un excelente valor didáctico, y han motivado varios de los objetivos que persigue este proyecto, los cuales serán desarrollados en los siguientes *sprints*.

3.1.2. Sprint 1: Grimoirebots I

El primer prototipo de la aplicación está creado utilizando el *framework* Django REST Framework para Python, y Poetry como gestor de paquetes y dependencias. Esta primera versión establece la arquitectura base de la aplicación y el esquema de comunicación entre servidor y cliente.

Para este fin, se han desarrollado los siguientes modelos¹:

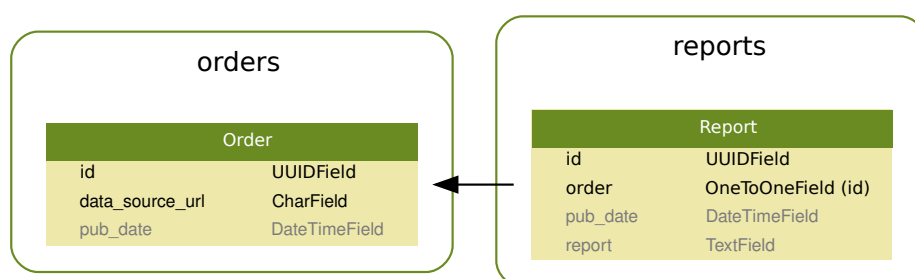


FIGURA 3.2: Modelos usados en Grimoirebots (Versión 1)

- **Order** – Este modelo representa la solicitud hecha por un usuario. Está formado por un identificador UUID único, un *string* representando la URL del repositorio

¹Los campos no obligatorios aparecen coloreados en gris.

git que se quiere analizar, y la fecha de creación de la instancia con un campo `datetime`. (Figura 3.2)

- Report – Este modelo representa el resultado de procesar una solicitud hecha por un usuario. Está formado por un identificador UUID único, una referencia a la solicitud (Order) que originó su creación, la fecha de creación de la instancia con un campo `datetime`, y un *string* representando el procesamiento de la solicitud. (Figura 3.2)

También se han desarrollado los siguientes *paths* en la API:

orders		^
GET	/orders/	▼
POST	/orders/	▼
GET	/orders/pending/	▼
GET	/orders/{id}/	▼
PUT	/orders/{id}/	▼
PATCH	/orders/{id}/	▼
DELETE	/orders/{id}/	▼
reports		^
GET	/reports/	▼
POST	/reports/	▼
GET	/reports/{id}/	▼
PUT	/reports/{id}/	▼
PATCH	/reports/{id}/	▼
DELETE	/reports/{id}/	▼

FIGURA 3.3: API de Grimoirebots (Versión 1)

- En la *app* orders se han incluido *paths* que permiten obtener la lista de solicitudes, la lista de solicitudes pendientes, y crear, modificar y eliminar solicitudes. (Figura 3.3)

- En la *app* reports se han incluido llamadas para listar informes, y para crear, modificar y eliminar los mismos. (Figura 3.3)

Adicionalmente, se ha creado una imagen Docker que instala las dependencias del proyecto y lo lanza utilizando Gunicorn como servidor WSGI, y se ha incluido el uso de Dependabot para la comprobación de cualquier actualización en las dependencias del proyecto y avisar de esto al equipo desarrollador.

Por otra parte, en este *sprint* también se ha desarrollado el cliente (o bot automático) de Grimoirebots, que es una aplicación escrita en Python que, haciendo uso de la librería *api-client*, recoge periódicamente aquellas solicitudes pendientes de procesar y realiza un procesamiento básico de ellas. En esta parte del desarrollo aún no se había incluido GrimoireLab, por lo que el procesamiento que el cliente hace de la solicitud es la construcción de un *string* que contiene el identificador y el repositorio de la solicitud.

Para ilustrar el comportamiento de este prototipo, se han creado las siguientes infografías:



FIGURA 3.4: *Frontend* de Grimoirebots (Versiones 1 y 2)

Como muestra la Figura 3.4, el usuario crea nuevas solicitudes de análisis realizando una llamada POST a `/orders/`. El usuario puede listar los informes creados mediante una llamada GET a `/reports/` y acceder a cada uno de ellos mediante una llamada GET a `/reports/<uuid>`.

Por su parte, en la Figura 3.5 se incluyen las interacciones que realiza el cliente con la API para procesar las solicitudes de los usuarios. Periódicamente, el cliente pide

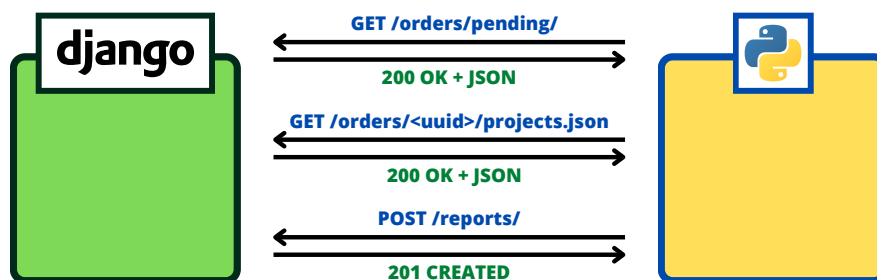


FIGURA 3.5: Backend de Grimoirebots (Versión 1)

al servidor las solicitudes que aún no hayan sido procesadas. Por cada una de estas, pide el repositorio solicitado para analizar haciendo una llamada GET a `/orders/<uuid>/projects.json`. Posteriormente, realiza el procesamiento básico comentado anteriormente y realiza una llamada POST a `/reports/` para crear el informe y asociarlo con la solicitud del usuario.

3.1.3. Sprint 2: Grimoirebots II

Durante este *sprint* se utilizan las mismas tecnologías mencionadas en el anterior, y algunas añadidas, para el desarrollo de un nuevo prototipo. Como gran aditivo, en este prototipo se incluye el uso de GrimoireLab y OpenSearch para realizar el análisis de proyectos *software*.

Debido a esto, este prototipo actualiza los modelos de la aplicación de la siguiente forma²:

- **Order** – Este modelo sigue representando la solicitud hecha por un usuario. El campo `data_source_url` ha sido eliminado y se ha incluido un campo `title` para que el usuario pueda identificar más fácilmente la solicitud. (Figura 3.6)
- **Setup** – Este modelo representa la configuración utilizada por GrimoireLab en el análisis de proyectos *software*. Está enlazado con una solicitud específica mediante el campo `order`. (Figura 3.6)
- **Projects** – Este modelo representa la lista de repositorios git que un usuario ha solicitado analizar. Está compuesto por un identificador numérico único, el

²Igual que en el apartado anterior, los campos no obligatorios aparecen coloreados en gris.

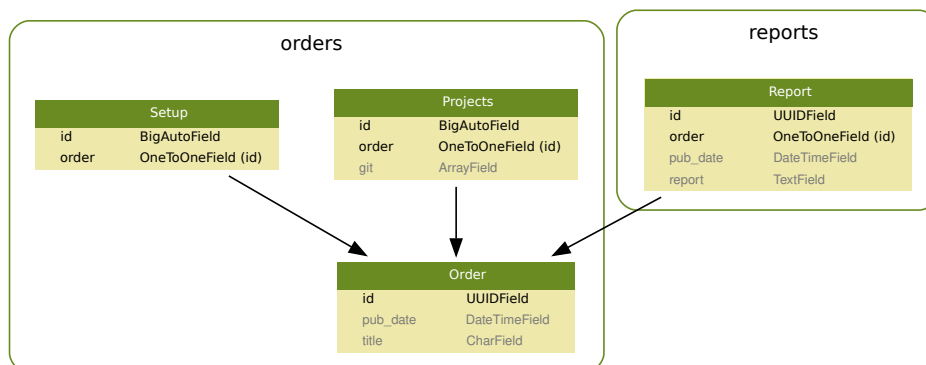


FIGURA 3.6: Modelos usados en Grimoirebots (Versión 2)

identificador de la solicitud, y una lista de *strings* con las URLs de los repositorios. (Figura 3.6)

- Report – Este modelo no ha sufrido cambios en este prototipo. (Figura 3.6)

Los *paths* que incluye la API de este prototipo son los mismos que en el anterior, a excepción de:

orders		^
POST	/orders/	▼
GET	/orders/{id}/projects.json	▼
GET	/orders/{id}/setup.cfg	▼

FIGURA 3.7: API de Grimoirebots (Versión 2)

- La llamada POST a /orders/ ahora incluye los campos title, projects, y setup en el cuerpo de la petición. De esta forma, el usuario utiliza una única llamada para crear su solicitud de análisis. (Figura 3.7)
- Se han incluido las llamadas GET para /orders/<uuid>/projects.json y para /orders/<uuid>/setup.cfg para obtener los ficheros con la lista de repositorios y la configuración para GrimoireLab, respectivamente. (Figura 3.7)

El cliente es modificado en este prototipo para incluir el uso de GrimoireLab. La forma en que el cliente hace uso de esta herramienta es mediante un contenedor Docker³ lanzado desde el mismo cliente gracias a la librería docker.

Por otro lado, en este prototipo se incluye el uso de OpenSearch y OpenSearch Dashboards para almacenar y visionar los datos generados por GrimoireLab. Este sistema es lanzado mediante Docker Compose⁴, y se ha construido un fichero de configuración con las características que necesita el sistema.

Igual que en el prototipo anterior, se han creado algunas infografías para ilustrar su comportamiento:

En el caso de la interacción entre usuario y servidor, el comportamiento es el mismo que en el prototipo 1. (Véase 3.4)

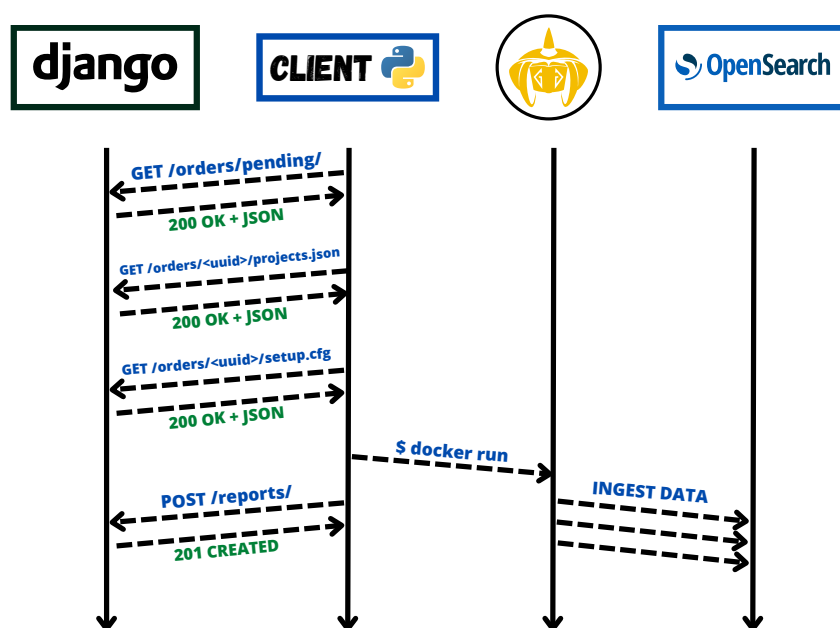


FIGURA 3.8: Backend de Grimoirebots (Versión 2)

³<https://hub.docker.com/r/grimoirelab/grimoirelab>

⁴Docker Compose es una herramienta que permite lanzar múltiples contenedores de manera simultánea, mediante ficheros de configuración.

El *backend*, sin embargo, sufre varios cambios. Como muestra la Figura 3.8, el cliente obtiene la lista de solicitudes pendientes. Para cada una de ellas obtiene los ficheros de repositorios y configuración. A continuación, utiliza la librería *docker* para lanzar un contenedor de GrimoireLab por cada solicitud, incluyendo los ficheros antes mencionados en el volumen que utiliza cada contenedor. Por último, cada contenedor realizará el análisis de los repositorios indicados en la solicitud e ingestará los datos al *cluster* de OpenSearch, desde donde el usuario es capaz de operar con ellos.

3.2. Problemas encontrados

Durante el desarrollo del proyecto, surgieron algunas situaciones que dificultaron la finalización del mismo.

Se quiso utilizar, al comienzo, la versión estándar de Django incluyendo su sistema de *templates*. Esto era debido a que la aplicación contaba en un principio con un *frontend* con interfaz gráfica, la cual sería accesible desde un navegador web. Debido a las ventajas que suponía el uso de Django REST Framework (como el uso de *serializers*) se decidió utilizar este para parte del *backend*. Sin embargo, este módulo funcionaba de manera bastante mala con el sistema de *templates* de Django. Finalmente, se decidió continuar usando Django REST Framework y dejar de lado la interfaz gráfica, debido a la gran ventaja que suponía el primero frente al segundo.

Uno de los ficheros que GrimoireLab utiliza, el llamado fichero de configuración, utiliza un formato con una estructura muy similar al que puede encontrarse en los ficheros INI usados en Microsoft Windows. Este formato no es soportado de forma nativa por Django REST Framework, y causó varios retrasos tratando de implementar una solución para hacerlo viable. Finalmente se decidió hacer que la API devolviera el fichero con formato JSON y luego, en el cliente, convertir este al formato necesario usando la librería *configparser*. Esto fue posible debido a que ambos formatos comparten similitudes que hacen viable su conversión.

Durante la última etapa del desarrollo, se quiso realizar una implementación de Grimoirebots desarrollada con FastAPI, con el fin de comparar este con Django REST Framework. Sin embargo, este prototipo nunca llegó a ver la luz por complicaciones en su desarrollo, aunque se realizaron algunas pruebas que pueden encontrarse en GitHub⁵.

⁵<https://github.com/merinhunter/grimoirebots-fastapi>

Además de todo lo anterior, existieron otros pequeños problemas que ralentizaron el desarrollo del proyecto, como la complejidad de ejecutar contenedores desde otro contenedor, o la falta de documentación en algunos módulos de GrimoireLab. Sin embargo, todos estos incidentes tienen un alto valor didáctico, y su documentación permitirá no volver a cometerlos en el futuro.

3.3. Tiempo dedicado

El desarrollo de cada *sprint* ha requerido un tiempo de trabajo distinto, dedicado a diferentes actividades. En esta sección se detalla el tiempo invertido en cada etapa:

- El primer *sprint*, al tratarse de una etapa de aprendizaje, se ha centrado exclusivamente en la investigación. Debido a que ya se contaba con un conocimiento inicial sobre la herramienta analizada, no ha supuesto un número excesivo de horas, finalizando este sprint a las tres semanas de iniciar el proyecto. Con una media diaria de dos horas, esta etapa ha supuesto un total de **42 horas**.
- El segundo *sprint* marca el comienzo del desarrollo del primer prototipo. Aunque principalmente se ha dedicado tiempo a la creación de componentes software, también se ha dedicado una pequeña parte del tiempo al estudio de diferentes herramientas que pudieran ayudar al desarrollo del prototipo. En total, la finalización de esta etapa ha supuesto dos semanas de investigación y cuatro de desarrollo, con una media de dos horas diarias, haciendo un total de **84 horas**.
- El tercer *sprint*, al partir de la base desarrollada en el sprint anterior, no debería haber requerido demasiado tiempo de trabajo. Sin embargo, los problemas que se detallan en el Capítulo 3.2, han incrementado el tiempo dedicado a esta etapa. Esta fase ha requerido utilizar dos semanas para investigación sobre cómo realizar la integración de GrimoireLab y OpenSearch con el proyecto, y cinco semanas de desarrollo para su implementación. Con una media diaria de dos horas, esta etapa se ha finalizado en **98 horas**.

En total, la finalización del proyecto ha requerido aproximadamente **224 horas** de trabajo, con mayor tiempo dedicado al desarrollo.

4 Resultados

4.1. Descripción técnica del sistema

En esta sección se detallan las especificaciones del sistema resultante tras el desarrollo del proyecto. Se detallan las prestaciones de este, así como los modelos de datos finales, y las entradas y salidas.

4.1.1. Definición y funcionalidades

Grimoirebots es el resultado del desarrollo expuesto en el Capítulo 3. Se trata de un complejo sistema de análisis de repositorios git que, de forma automática, obtiene y genera datos sobre el desarrollo de *software*, permitiendo su uso mediante la creación de métricas y gráficas.

Las principales funcionalidades de este sistema son:

- Ofrece una API desde la que crear, modificar, recoger y eliminar instancias de los distintos modelos que componen el sistema (Sección 4.1.2)
- Acceso a una instancia personal de OpenSearch Dashboards, desde la que visualizar y crear métricas y gráficas con los datos obtenidos de los análisis
- Cliente que, de forma automática, recoge todas las solicitudes pendiente de análisis y ejecuta un contenedor Docker con GrimoireLab para su análisis
- Herramientas de despliegue automático de la aplicación, tales como ficheros de configuración para Docker Compose

4.1.2. Modelos de datos

Los distintos modelos que ofrece la API están divididos en dos grandes aplicaciones: orders y reports. Los modelos de la primera están principalmente enfocados en todos

aquellos elementos necesarios para la creación de solicitudes de análisis; y los modelos de la segunda en el resultado de estos análisis. En la Figura 4.1 se puede encontrar una representación visual de estos modelos.

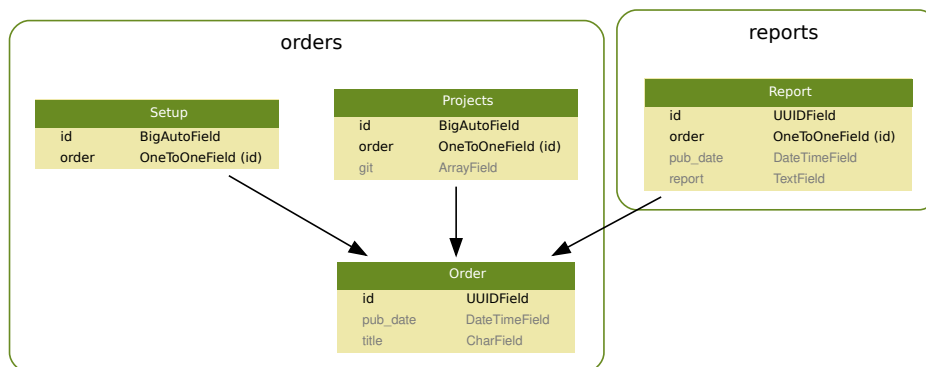


FIGURA 4.1: Modelos usados en Grimoirebots

Los modelos de datos creados dentro de la aplicación `orders` son los siguientes:

- **Order** – Este modelo representa una solicitud de análisis. Por tanto, es el elemento raíz que el cliente de Grimoirebots solicita para llevar a cabo un análisis. Está compuesto por un identificador único, un título, y una fecha de creación, y es que el resto de elementos necesarios para el análisis están contenidos en otros modelos.
- **Projects** – Este modelo representa la lista de repositorios git solicitados para analizar. Lo compone un identificador único, una referencia al modelo `Order` del cual depende, y un *array* de *strings* con todas las URLs de repositorios git que se quieren analizar.
- **Setup** – Este modelo representa la configuración de análisis para GrimoireLab, esto es, qué estudios se quieren incluir, las credenciales de OpenSearch, etc. Está compuesto por un identificador único y una referencia al modelo `Order` que lo originó. Al ser objeto de análisis de este proyecto únicamente los repositorios git, la configuración usada en cada análisis es la misma. Sin embargo, el modelo es bastante versátil, y permite la inclusión de nuevas opciones fácilmente.

Los modelos de datos creados dentro de la aplicación `reports` son los siguientes:

- Report – Este modelo representa el resultado de procesar una solicitud de análisis. La creación de una instancia de este modelo implica el comienzo de un análisis. Está compuesto por un identificador único, una referencia a la instancia Order que lo originó, la fecha de creación del mismo, y un pequeño texto donde se representa el procesamiento de la solicitud.

4.1.3. Descripción de la API

En esta sección se van a detallar los distintos *paths* que componen la API de Grimoirebots, sus entradas y su salidas. Igual que con los modelos, la API está dividida en dos grandes secciones: orders y reports.

Los *paths* disponibles para la aplicación orders son los siguientes (Figura 4.2):

orders		^
GET	/orders/	✓
POST	/orders/	✓
GET	/orders/pending/	✓
GET	/orders/{id}/	✓
PUT	/orders/{id}/	✓
PATCH	/orders/{id}/	✓
DELETE	/orders/{id}/	✓
GET	/orders/{id}/projects.json/	✓
GET	/orders/{id}/setup.cfg/	✓

FIGURA 4.2: API de Grimoirebots para orders

- El *path* /orders/ permite interactuar con la colección completa de Orders. Permite realizar llamadas de tipo GET y POST para recoger todos los elementos de la colección y crear nuevos elementos en esta, respectivamente. La petición POST, además de crear un elemento de tipo Order, también crea un elemento de tipo

Setup y Project y los asocia al elemento Order creado anteriormente. La petición GET no tiene parámetros de entrada, pero la petición POST necesita que se incluya el siguiente cuerpo en el *payload* de la petición:

```
{
  "title": "string",
  "projects": {
    "git": [
      "string"
    ]
  },
  "setup": {}
}
```

- El *path* /orders/pending/ permite recoger aquellas instancias Order que no tienen una instancia Report asociada, o lo que es lo mismo, aquellas solicitudes de análisis que aún no han sido analizadas.
- El *path* /orders/{id}/ permite interactuar con una instancia Order en particular, identificada por el parámetro id en la solicitud. Permite realizar llamadas de tipo GET, PUT, PATCH y DELETE para recoger, actualizar, actualizar parcialmente y borrar una instancia, respectivamente. Las llamadas GET y DELETE no necesitan parámetros adicionales más allá del campo id incluido en la solicitud, pero las llamadas PUT y PATCH necesitan que se incluya el siguiente cuerpo en el *payload* de la llamada:

```
{
  "title": "string",
  "projects": {
    "git": [
      "string"
    ]
  },
  "setup": {}
}
```

- Por último, los *paths* `/orders/{id}/projects.json` y `/orders/{id}/setup.cfg` permiten recoger una representación de los modelos `Project` y `Setup`, respectivamente. Y se recalca lo de representación porque incluyen más datos de los que los modelos originalmente tienen. Esto es debido a que GrimoireLab necesita un formato específico para sus ficheros de configuración, y la API los devuelve prácticamente listos para ser utilizados por esta herramienta.

Los *paths* disponibles para la aplicación `reports` son los siguientes (Figura 4.3):

reports		^
GET	/reports/	▼
POST	/reports/	▼
GET	/reports/{id}/	▼
PUT	/reports/{id}/	▼
PATCH	/reports/{id}/	▼
DELETE	/reports/{id}/	▼

FIGURA 4.3: API de Grimoirebots para `reports`

- El *path* `/reports/` permite interactuar con la colección completa de `Reports`. Permite realizar llamadas de tipo `GET` y `POST` para recoger todos los elementos de la colección y crear nuevos elementos en esta, respectivamente. La petición `GET` no tiene parámetros de entrada, pero la petición `POST` necesita que se incluya el siguiente cuerpo en el *payload* de la petición:

```
{
  "order": "string",
  "report": "string"
}
```

- El *path* `/reports/{id}/` permite interactuar con una instancia `Report` en particular, identificada por el parámetro `id` en la solicitud. Permite realizar llamadas de tipo `GET`, `PUT`, `PATCH` y `DELETE` para recoger, actualizar, actualizar parcialmente y borrar una instancia, respectivamente. Las llamadas `GET` y `DELETE` no

necesitan parámetros adicionales más allá del campo `id` incluido en la solicitud, pero las llamadas PUT y PATCH necesitan que se incluya el siguiente cuerpo en el *payload* de la llamada:

```
{
  "order": "string",
  "report": "string"
}
```

4.1.4. Entradas y salidas del sistema

En la sección anterior ya se detallaron algunas de las entradas y salidas de la API. Sin embargo, aquí se incluye esta información algo más estructurada.

Entradas del sistema

- Las peticiones de tipo GET y DELETE no tienen ningún tipo de entrada necesaria, más allá del campo `id` contenido en algunos *paths*.
- Las peticiones de tipo PUT, PATCH y POST necesitan incluir en el *payload* de la petición una representación del modelo que quiere crear o actualizar (a excepción de PATCH, que permite introducir parcialmente esta información). Este cuerpo es diferente en función de si el elemento a crear / actualizar es de tipo Order o Report, siendo estos:

```
{
  "title": "string",
  "projects": {
    "git": [
      "string"
    ]
  },
  "setup": {}
}
```

```
{
  "order": "string",
  "report": "string"
}
```

}

Salidas del sistema

- Las peticiones de tipo GET, PUT y PATCH responden con un código 200 indicando que la solicitud ha sido un éxito, y con la representación del modelo en el *payload*.
- Las peticiones de tipo POST responden con un código 201 indicando que el elemento ha sido creado, e incluyen una representación de este en el *payload*.
- Las peticiones de tipo DELETE responden con un código 204, indicando que la petición no tiene contenido en el *payload*.

4.1.5. Cliente automático

Otro de los componentes de Grimoirebots es su sistema automático de lanzamiento de contenedores, o también conocido como cliente de Grimoirebots. Esta pieza se encarga de recoger periódicamente todas aquellas solicitudes de análisis pendientes de analizar, recoge sus ficheros de configuración y de repositorios, y lanza un contenedor Docker con GrimoireLab para realizar el análisis.

Para ello, hace uso de la librería `api-client`, de la que hereda la clase `APIClient` para la construcción de un cliente API para Grimoirebots. En él tiene configurados todos los *endpoints* de la API para poder interaccionar fácilmente con ellos.

La forma en que lanza los contenedores de GrimoireLab es mediante el uso de la librería `docker`, la cual permite interaccionar con Docker. Debido a que el cliente necesita lanzar múltiples contenedores, y que cada uno de estos debe llevar a cabo un análisis diferente, el cliente utiliza diferentes volúmenes para almacenar los documentos de configuración de cada análisis.

Además, estos contenedores necesitan tener visible el *cluster* de OpenSearch, por lo que es necesario desplegarlos en el *host*. Esta situación está descrita en el Capítulo 3.2, y es que el propio cliente está desplegado en forma de contenedor, por lo que hubo que configurar los ajustes de red del mismo para que fuera capaz de lanzar contenedores usando el mismo servicio que lo había levantado a él.

4.1.6. Flujos de ejecución

Dentro de la plataforma se diferencian dos flujos de ejecución diferentes: el que el usuario lleva a cabo para solicitar un análisis y visualizar los datos, y el que realiza el cliente de Grimoirebots para lograr este análisis. En esta sección se repasará cada uno de ellos.

Flujo de ejecución del usuario

Un usuario de Grimoirebots lo primero que realiza es una petición de tipo POST a la API, concretamente al path `/orders/`, indicando en el *payload* de la petición los requisitos de este. (Figura 4.4)



FIGURA 4.4: Flujo de ejecución de un usuario en Grimoirebots

A partir de aquí, el análisis se llevaría a cabo, y el usuario podría acceder a los datos del mismo usando OpenSearch Dashboards, el cual estaría disponible a través de un navegador web.

Flujo de ejecución del cliente

El cliente automático se encarga periódicamente de llevar a cabo peticiones de tipo GET al path `/orders/pending/`, de donde recoge todas aquellas solicitudes pendiente de análisis.

Por cada una de ellas, recoge los ficheros de configuración de GrimoireLab, realizando una petición de tipo GET a los *paths* `/orders/{id}/projects.json` y `/orders/{id}/setup.cfg`, y lanza un contenedor Docker con esa configuración. Finalmente, genera un elemento de tipo Report realizando una petición POST al path `/reports/`.

Al finalizar, el contenedor se apaga y vuelve a ejecutar unos segundos después gracias a la opción restart de Docker. El flujo descrito puede encontrarse en la Figura 4.5

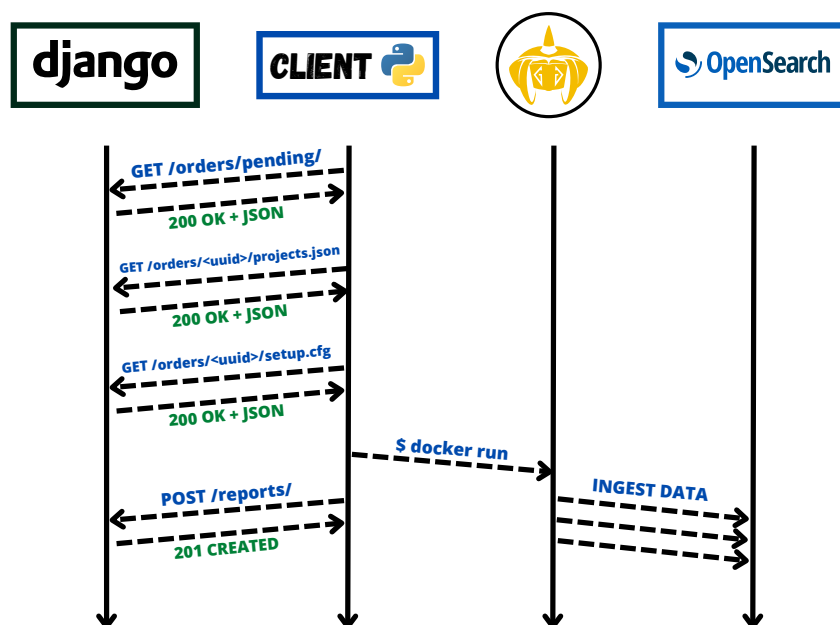


FIGURA 4.5: Flujo de ejecución del cliente en Grimoirebots

Estos flujos quedan bastante representados en el posterior Capítulo 4.3, donde se lleva a cabo un caso de uso y se pasa por todas las etapas descritas.

4.2. Despliegue

4.2.1. Requisitos

En esta sección se describen todas las herramientas y *software* que es necesario instalar en el sistema antes de la ejecución de Grimoirebots.

- **Docker** – La ejecución de Grimoirebots se realiza usando contenedores Docker, por lo que su instalación es obligatoria. La instalación de Docker es diferente en función del Sistema Operativo utilizado, encontrándose una guía para los más utilizados en su página de documentación¹.

¹<https://docs.docker.com/get-docker/>

- **Docker Compose** – La herramienta Docker Compose permite ejecutar varios contenedores al mismo tiempo, gracias a ficheros de configuración. Su instalación es necesaria para la ejecución del *cluster* de OpenSearch y de la base de datos PostgreSQL. De nuevo, su instalación varía en función del Sistema Operativo, encontrándose una guía de instalación en su página de documentación².

Al tratarse de una aplicación *dockerizada*, todos los demás requisitos están incluidos en los contenedores. A pesar de no ser necesaria su instalación, y simplemente para dar a conocerlos, estos son:

- **Python** – Para la ejecución tanto del servidor como del cliente se ha utilizado Python 3.11.
- **Poetry** – La gestión (e instalación) de paquetes en Grimoirebots se realiza con la versión más actualizada de esta herramienta.
- **Dependencias de los proyectos Python** – Esto incluye las últimas versiones disponibles de las librerías Django, gunicorn, djangorestframework, psycopg, py-yaml, uritemplate, api-client y configparser.
- **PostgreSQL** – La base de datos SQL que utiliza Django. La versión que utiliza el proyecto es la 15.1.
- **OpenSearch** – Se utiliza para almacenar los resultados de los análisis con GrimoireLab. La versión que utiliza el proyecto es la versión más actualizada de la versión 1³.
- **OpenSearch Dashboards** – Se utiliza la misma versión que para OpenSearch.

4.2.2. Configuración

Antes de realizar el despliegue de la aplicación y sus componentes, es necesario configurar estos. En esta sección se detallan los pasos necesarios para preparar un entorno local para el despliegue de Grimoirebots⁴.

Primero, es necesario descargar o clonar el repositorio de Grimoirebots:

²<https://docs.docker.com/compose/install/>

³GrimoireLab no soporta versiones superiores a la 2, por lo que el proyecto está limitado por esta parte.

⁴Se asume que el despliegue se realizará en una máquina Linux local.


```
$ git clone https://github.com/merinhunter/grimoirebots.git
```

A continuación, se debe construir la imagen Docker:

```
$ cd grimoirebots/  
$ docker build -t grimoirebots .
```

Una vez hecho esto, es necesario descargar o clonar el repositorio con los ficheros de configuración para desplegar Grimoirebots con Docker Compose:

```
$ git clone https://github.com/merinhunter/grimoirebots-deployment.git
```

En este repositorio es posible encontrar ficheros YAML con la configuración para desplegar tanto la base de datos PostgreSQL como el servidor Django.

Los ficheros de configuración están preparados para funcionar una vez descargados, pero es recomendable modificar las claves por defecto en los siguientes ficheros:

```
$ cd grimoirebots-deployment/  
$ vim grimoirebots/docker-compose.yml  
$ vim postgresql/docker-compose.yml
```

Las variables que se debe modificar son `DJANGO_SECRET_KEY` y `DATABASE_PASSWORD`⁵.

A continuación, es necesario descargar o clonar el repositorio del cliente de Grimoirebots:

```
$ git clone https://github.com/merinhunter/grimoirebots-client.git
```

E igual que en pasos anteriores, construir la imagen Docker:

```
$ cd grimoirebots-client/  
$ docker build -t grimoirebots-client .
```

Por último, el *cluster* de OpenSearch requiere que, en el *host*, el valor de la variable `vm.max_map_count` sea de, al menos, 262144:

```
$ sudo vim /etc/sysctl.conf
```

⁵Esta variable aparece dos veces en el fichero `grimoirebots/docker-compose.yml`.

```
# Include the following line at the end of the file
vm.max_map_count=262144

$ sudo sysctl -p
```

4.2.3. Despliegue de componentes

Una vez realizada la correcta configuración de todos los componentes, es posible llevar a cabo su despliegue. En esta sección se detallan todos los pasos necesarios para poner Grimoirebots en funcionamiento.

El primer paso es poner en marcha el *cluster* de OpenSearch:

```
$ cd grimoirebots-deployment/opensearch/
$ docker compose up -d
```

A continuación, se despliega la base de datos PostgreSQL:

```
$ cd grimoirebots-deployment/postgresql/
$ docker compose up -d
```

Se continúa con el despliegue del cliente de Grimoirebots:

```
$ cd grimoirebots-deployment/grimoirebots-client/
$ docker compose up -d
```

Y, finalmente, se despliega el servidor de Grimoirebots:

```
$ cd grimoirebots-deployment/grimoirebots/
$ docker compose up -d
```

En este punto, Grimoirebots debería estar escuchando peticiones en el *endpoint* `http://localhost:8000`, y debería ser posible acceder a OpenSearch Dashboards en el *endpoint* `http://localhost:5601`.

Para comprobarlo, es posible realizar una petición al servidor a través de la terminal:

```
$ curl http://localhost:8000
openapi: 3.0.2
info:
```

```
title: Grimoirebots
version: 0.1.0
description: Run GrimoireLab in the cloud!
...
```

Accediendo a la dirección `http://localhost:5601` usando un navegador web, se debería ver la página de *login* de OpenSearch Dashboards tal como aparece en la Figura 4.6.



FIGURA 4.6: Página de *login* en OpenSearch Dashboards

4.3. Ejemplo de uso

Para ilustrar el funcionamiento de Grimoirebots, en esta sección se va a mostrar un caso de uso en el que un usuario quiere analizar una serie de repositorios git, y después visualizar los datos generados en este análisis.

Para ello se va a hacer uso de la herramienta Postman, que permite realizar fácilmente peticiones HTTP a diferentes *endpoints*.

4.3.1. Creación de una petición de análisis

El primer paso será realizar una petición POST al *endpoint* `http://localhost:8000/orders/`, con la configuración del análisis en el campo *body* de la solicitud (Figura 4.7).

La solicitud dará como respuesta un **201 Created** (Figura 4.8).

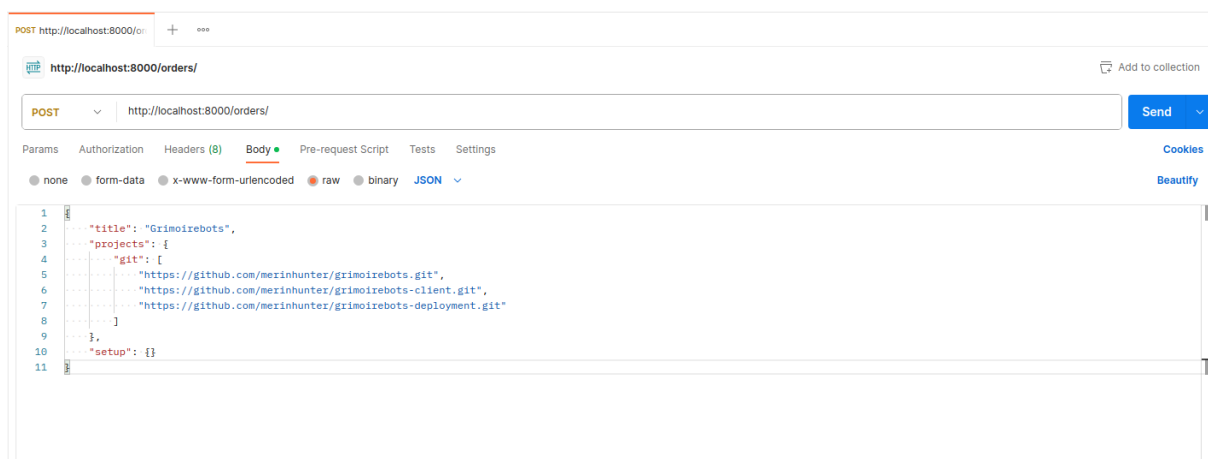


FIGURA 4.7: Creación de una solicitud de análisis en Grimoirebots

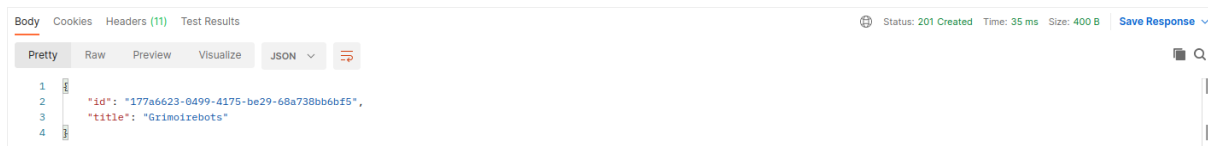


FIGURA 4.8: Respuestas a la creación de una solicitud de análisis en Grimoirebots

Si se realiza una petición GET al *endpoint* `http://localhost:8000/orders/<order_id>`, sustituyendo `<order_id>` por el identificador de la solicitud de análisis enviada, se puede obtener el objeto creado (Figura 4.9).

Y realizando una petición GET a los *endpoints* `http://localhost:8000/orders/<order_id>/projects.json` y `http://localhost:8000/orders/<order_id>/setup.cfg` se pueden obtener los ficheros de configuración de GrimoireLab para el análisis (Figura 4.10 y Figura 4.11).

A continuación, el cliente de Grimoirebots solicitará al servidor estos ficheros. Se pueden comprobar los logs del contenedor para asegurarnos de ello:

```

$ docker logs grimoirebots-client
2023-06-18 17:00:54,511 - INFO - Retrieving pending orders from http://
    ↪ localhost:8000
2023-06-18 17:00:54,523 - INFO - Retrieved 1 orders
2023-06-18 17:00:54,523 - INFO - Processing order 177a6623-0499-4175-be29
    ↪ -68a738bb6bf5

```

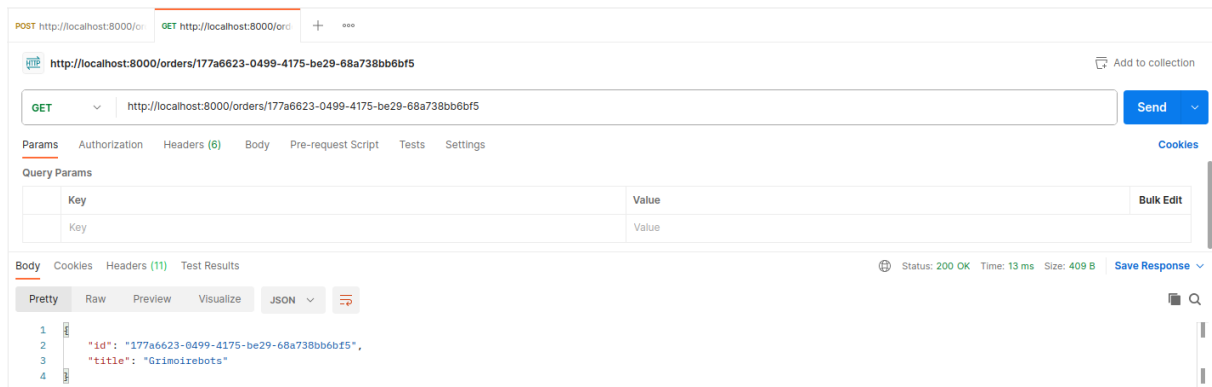


FIGURA 4.9: Solicitud de análisis en Grimoirebots

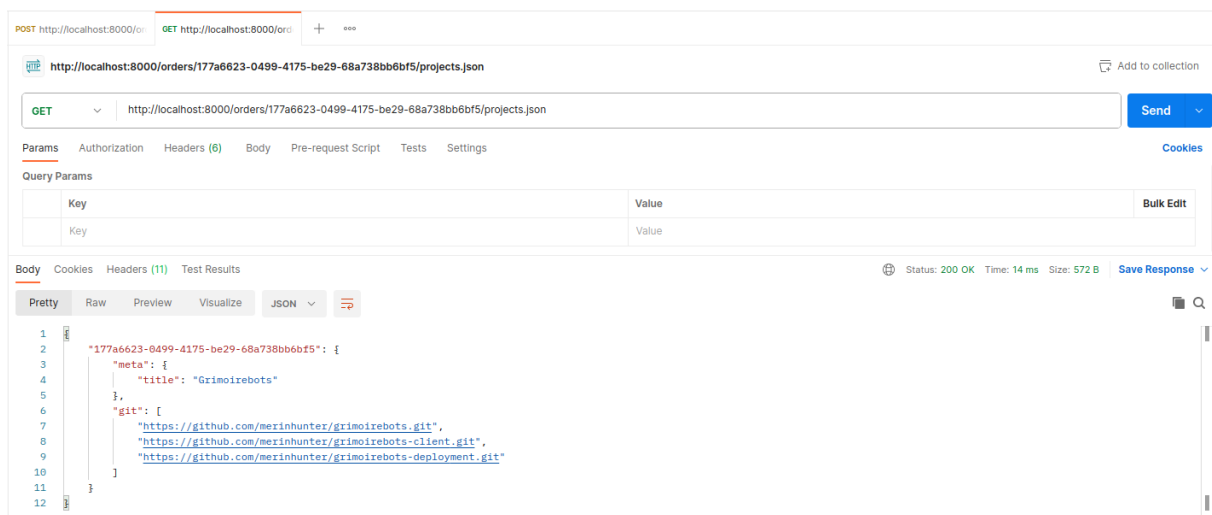


FIGURA 4.10: Fichero de repositorios de un análisis en Grimoirebots

```

2023-06-18 17:00:54,523 - INFO - Retrieving projects.json for order 177
    ↳ a6623-0499-4175-be29-68a738bb6bf5
2023-06-18 17:00:54,536 - INFO - File projects.json for order 177a6623
    ↳ -0499-4175-be29-68a738bb6bf5 saved at reports/177a6623-0499-4175-
    ↳ be29-68a738bb6bf5/projects.json
2023-06-18 17:00:54,536 - INFO - Retrieving setup.cfg for order 177a6623
    ↳ -0499-4175-be29-68a738bb6bf5
2023-06-18 17:00:54,550 - INFO - File setup.cfg for order 177a6623
    ↳ -0499-4175-be29-68a738bb6bf5 saved at reports/177a6623-0499-4175-
    ↳ be29-68a738bb6bf5/setup.cfg

```

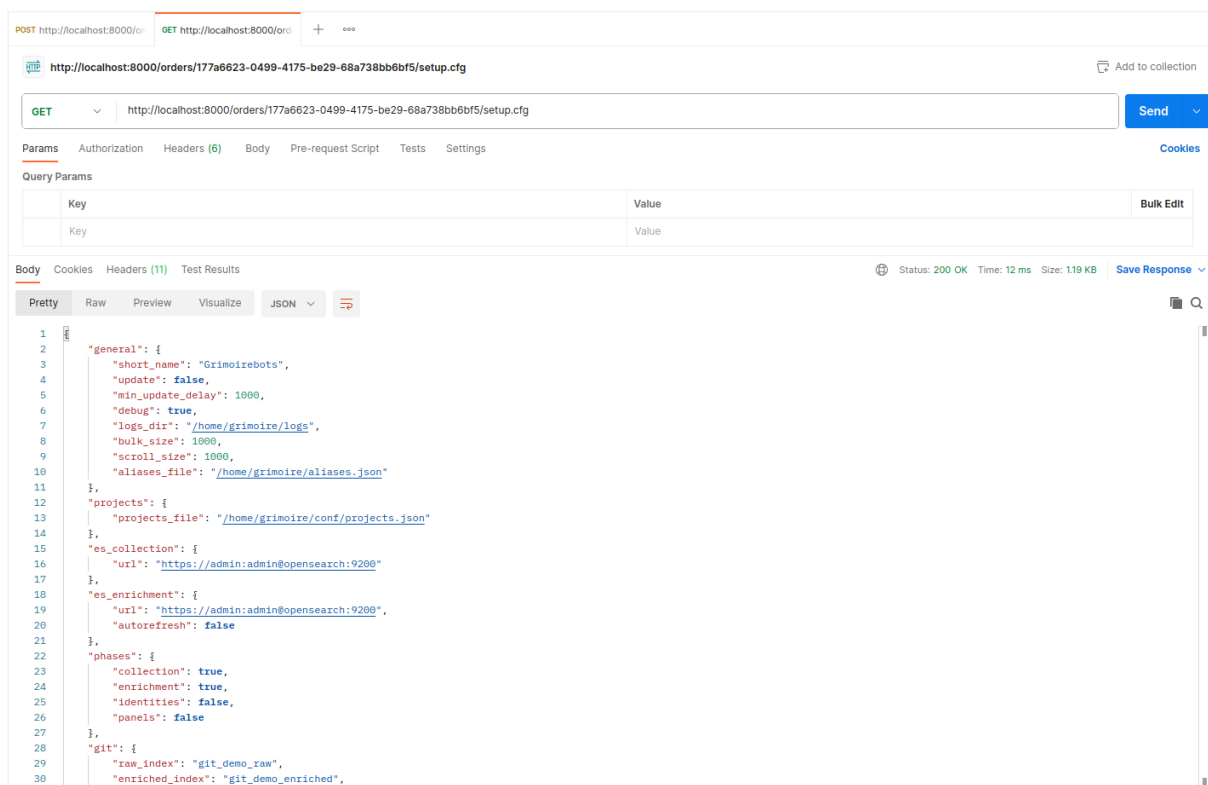


FIGURA 4.11: Fichero de configuración de un análisis en Grimoirebots

```

2023-06-18 17:00:54,550 - INFO - Starting GrimoireLab container
2023-06-18 17:00:54,676 - INFO - Analysis for 177a6623-0499-4175-be29-68
    ➔ a738bb6bf5 has finished
2023-06-18 17:00:54,676 - INFO - Creating report for order 177a6623
    ➔ -0499-4175-be29-68a738bb6bf5
2023-06-18 17:00:54,690 - INFO - Report 9ddf60a6-964f-41d5-9493-
    ➔ f058cbdc2c41 created for order 177a6623-0499-4175-be29-68a738bb6bf5
  
```

Y una vez que el cliente haya comenzado el análisis, se puede realizar una petición GET al *endpoint* `http://localhost:8000/reports/` para comprobar que existe un informe asociado a la petición inicial (Figura 4.12).

A partir de este momento, el análisis se está llevando a cabo en un contenedor de GrimoireLab. El seguimiento de este análisis se puede realizar revisando en tiempo real los logs del contenedor, necesitando localizar a este primero:

```
$ docker ps
```

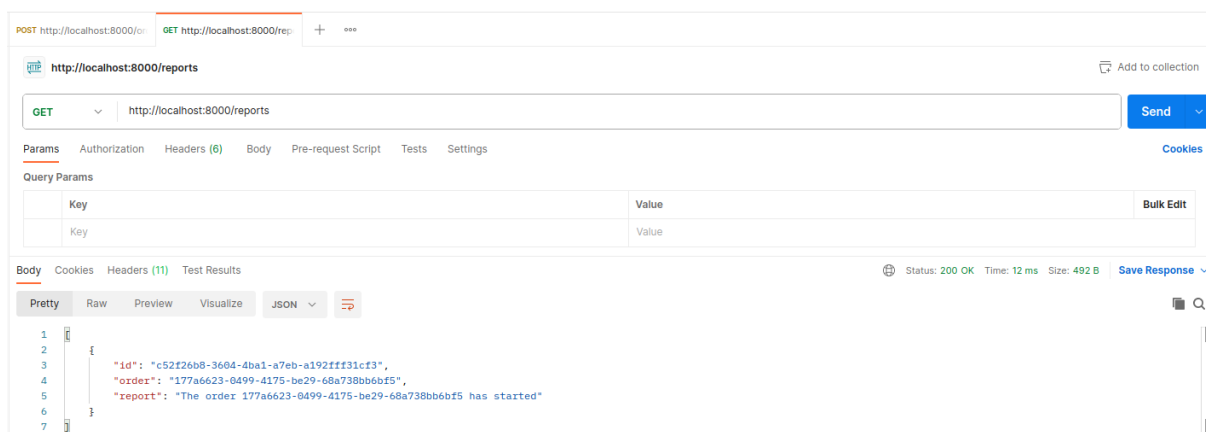


FIGURA 4.12: Informe en Grimoirebots

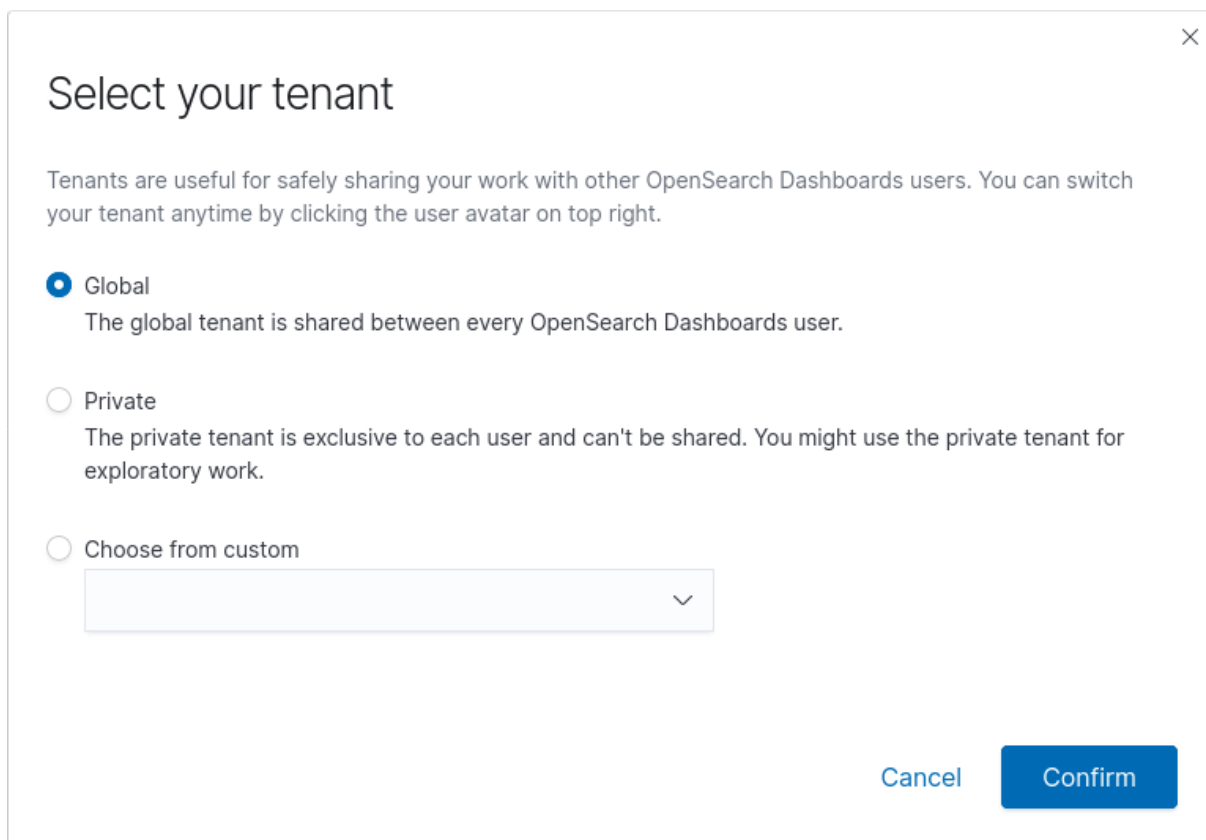
```
CONTAINER ID IMAGE ... NAMES
b0a1f6acf822 grimoirelab/grimoirelab:0.10.0 ... grimoirelab-177a6623
    ↪ -0499-4175-be29-68a738bb6bf5
...

$ docker logs -f grimoirelab-177a6623-0499-4175-be29-68a738bb6bf5
2023-06-18 17:00:55,882 - sirmordred.sirmordred - INFO -
2023-06-18 17:00:55,882 - sirmordred.sirmordred - INFO -
    ↪ -----
2023-06-18 17:00:55,882 - sirmordred.sirmordred - INFO - Starting
    ↪ SirMordred engine ...
2023-06-18 17:00:55,882 - sirmordred.sirmordred - INFO -
    ↪ -----
2023-06-18 17:00:55,885 - urllib3.connectionpool - DEBUG - Starting new
    ↪ HTTPS connection (1): localhost:9200
...
```

4.3.2. Visualización de datos en OpenSearch

Una vez que el análisis con GrimoireLab haya concluido, es posible visualizar los datos del análisis y construir gráficos y métricas con ellos accediendo a OpenSearch Dashboards. Este servicio debería estar disponible navegando a la dirección `http://localhost:5601` con un navegador web.

Las credenciales por defecto son **admin** para el usuario y la contraseña. Al acceder, lo primero que se pedirá es seleccionar el *tenant* de trabajo. Para el ejemplo se seleccionará el *tenant* global (Figura 4.13).



Select your tenant

Tenants are useful for safely sharing your work with other OpenSearch Dashboards users. You can switch your tenant anytime by clicking the user avatar on top right.

☒ Global
The global tenant is shared between every OpenSearch Dashboards user.

☐ Private
The private tenant is exclusive to each user and can't be shared. You might use the private tenant for exploratory work.

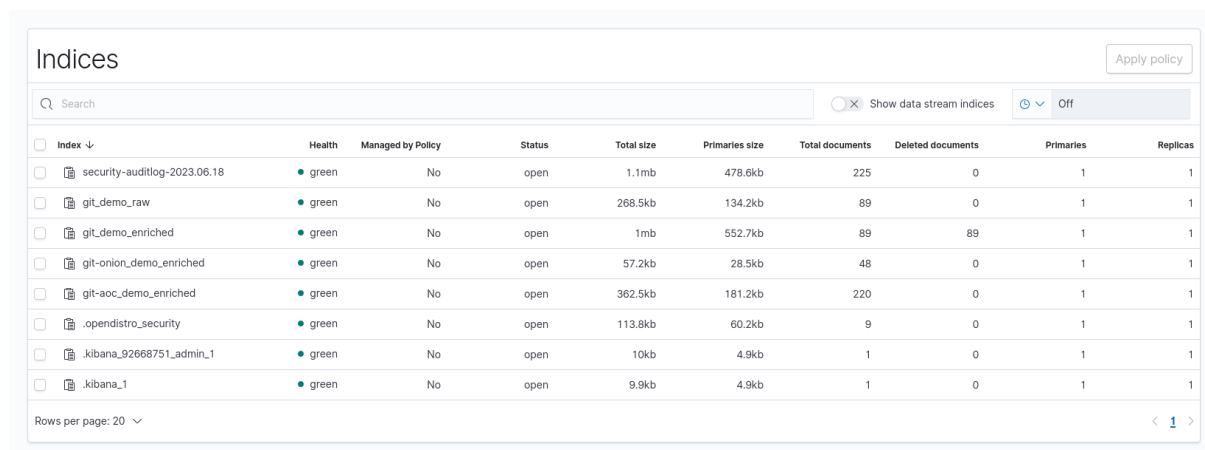
☐ Choose from custom

Cancel Confirm

FIGURA 4.13: Selección de *tenant* en OpenSearch

Si se accede a la sección Index Management / Indices es posible comprobar que existen algunos índices originados por GrimoireLab, como `git_demo_raw` o `git_demo_enriched` (Figura 4.14).

Para visualizar los datos, es necesario primero crear un *index pattern*. Esto sirve para que OpenSearch sepa qué campos de los documentos debe utilizar en la indexación de la información. Para ello, se debe hacer clic en la pestaña *Discover*, en la barra lateral, desde donde se solicitará la creación de un *index pattern*. Para este ejemplo se utilizará como *index pattern* el valor `git`, que concuerda con uno de los índices existentes (Figura 4.15).



The screenshot shows the OpenSearch Indices management interface. At the top, there is a search bar and a toggle for 'Show data stream indices' (currently off). Below the search bar is a table listing various indices. Each row includes a checkbox, an index icon, the index name, its health status (green), whether it's managed by a policy (No), its status (open), total size, primary size, total documents, deleted documents, number of primaries, and number of replicas. The table is paginated, showing 20 rows per page.

<input type="checkbox"/>	Index ↓	Health	Managed by Policy	Status	Total size	Primaries size	Total documents	Deleted documents	Primaries	Replicas
<input type="checkbox"/>	security-auditlog-2023.06.18	● green	No	open	1.1mb	478.6kb	225	0	1	1
<input type="checkbox"/>	git_demo_raw	● green	No	open	268.5kb	134.2kb	89	0	1	1
<input type="checkbox"/>	git_demo_enriched	● green	No	open	1mb	552.7kb	89	89	1	1
<input type="checkbox"/>	git-onion_demo_enriched	● green	No	open	57.2kb	28.5kb	48	0	1	1
<input type="checkbox"/>	git-aoc_demo_enriched	● green	No	open	362.5kb	181.2kb	220	0	1	1
<input type="checkbox"/>	.opendistro_security	● green	No	open	113.8kb	60.2kb	9	0	1	1
<input type="checkbox"/>	.kibana_92668751_admin_1	● green	No	open	10kb	4.9kb	1	0	1	1
<input type="checkbox"/>	.kibana_1	● green	No	open	9.9kb	4.9kb	1	0	1	1

Rows per page: 20

FIGURA 4.14: Índices en OpenSearch

Como campo temporal se utilizará el campo `utc_commit`, que corresponde con la hora UTC en la que cada *commit* se realizó (Figura 4.16).

Una vez creado el *index pattern*, es posible hacer clic en *Discover* y visualizar algunos de los datos generados por el análisis. Si se ajusta la fecha al último año, se obtiene una serie temporal con los *commits* realizados en los diferentes repositorios analizados a lo largo del último año (Figura 4.17).

Create index pattern

An index pattern can match a single source, for example, `filebeat-4-3-22`, or **multiple** data sources, `filebeat-*`.
[Read documentation](#)

Step 1 of 2: Define an index pattern

Index pattern name

git

Next step >

Use an asterisk (*) to match multiple indices. Spaces and the characters `\,/,?,",<,>|` are not allowed.

☐ Include system and hidden indices

✓ Your index pattern matches 1 source.

git	Alias
-----	-------

Rows per page: 10 ▾

FIGURA 4.15: Creación de *Index Pattern* en OpenSearch

Create index pattern

An index pattern can match a single source, for example, `filebeat-4-3-22`, or **multiple** data sources, `filebeat-*`.
[Read documentation](#)

Step 2 of 2: Configure settings

Specify settings for your **git** index pattern.

Select a primary time field for use with the global time filter.

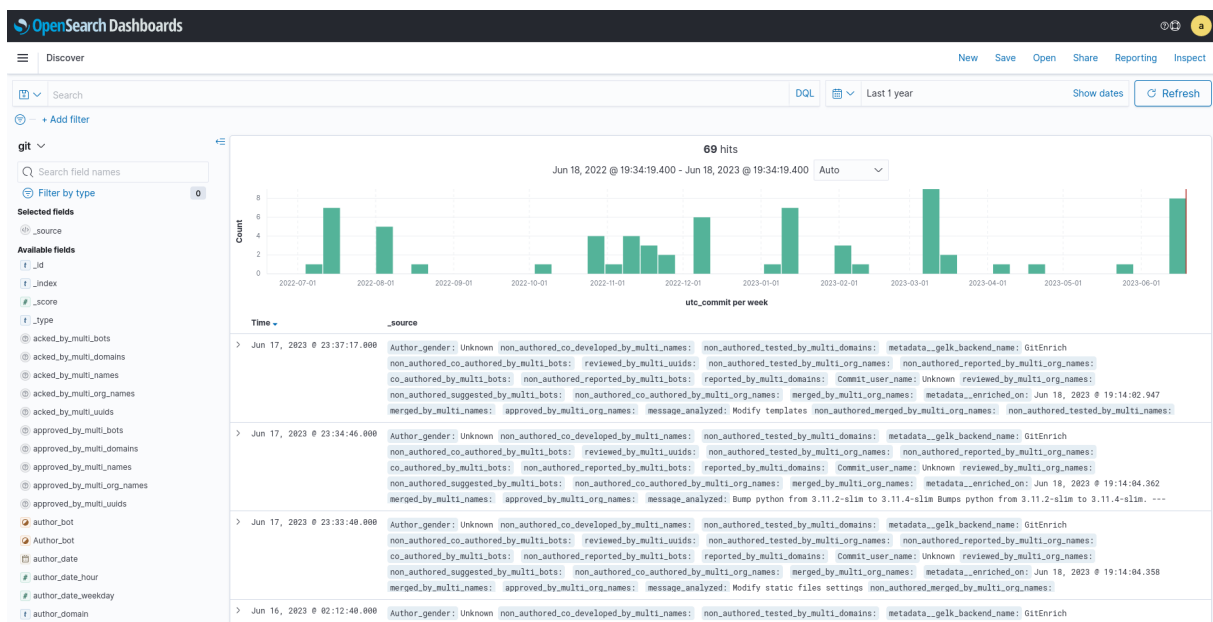
Time field Refresh

utc_commit ▾

> [Show advanced settings](#)

[< Back](#) [Create index pattern](#)

FIGURA 4.16: Creación de *Index Pattern* en OpenSearch (Cont.)

FIGURA 4.17: Sección *Discover* en OpenSearch

5 Conclusiones

5.1. Consecución de objetivos

Los objetivos propuestos al comienzo del proyecto eran:

- Desarrollar un sistema que genere informes inalterables, creando un flujo más simple desde la solicitud por parte del usuario al *output* de la aplicación.
- Desarrollar múltiples componentes *software* independientes que colaboren entre ellos para dar un servicio al usuario.
- Simplificar el uso de GrimoireLab mediante la ejecución del contenedor Docker original de GrimoireLab.
- Realizar varias mejoras menores respecto a Cauldron.

Estos objetivos se han llevado a cabo de manera satisfactoria. El sistema desarrollado permite el análisis de repositorios *software* utilizando un esquema de informes inalterables y con una arquitectura desacoplada entre sus componentes.

Sin embargo, el objetivo principal que perseguía este proyecto era también el desarrollo de un sistema que mejorase las prestaciones de Cauldron. Este objetivo no se ha cumplido: Cauldron ofrece muchas más características que Grimoirebots a día de hoy, como la existencia de usuarios, la comparación de análisis o un mayor número de fuentes de datos disponible.

5.2. Conocimientos adquiridos

Desde la concepción del proyecto hasta la finalización de esta memoria, he estado inmerso en un continuo aprendizaje. A pesar de tener un conocimiento previo en la mayoría de las tecnologías utilizadas, otras han resultado un descubrimiento y una ampliación en experiencia y sabiduría.

Los conocimientos adquiridos más relevantes son:

- Reforzar y ampliar el conocimiento sobre el desarrollo de APIs REST mediante el estudio y uso herramientas novedosas en la creación de APIs REST como Django REST Framework o FastAPI, así como el estudio e investigación del estándar OpenAPI.
- Profundizar en el desarrollo de contenedores con Docker, al tener que elaborar esquemas de funcionamiento complejos con estos.
- Mejorar la destreza en el desarrollo de código con Python, mediante la escritura de código más limpio y más manejable.
- Aprender como funcionan las arquitecturas de *software* con componentes desacoplados, al haber diseñado e implementado los diferentes servicios que componen el sistema.
- Ampliar el conocimiento sobre GrimoireLab, mediante el estudio avanzado de todas sus herramientas, su configuración y su ejecución.
- Realizar la gestión del tiempo durante el desarrollo del proyecto para cumplir los plazos de entrega.

Todo el conocimiento y experiencia adquiridos me ha permitido comprender mejor el mundo del desarrollo web y las arquitecturas de *software* desacopladas. Aunque esperaba haber dotado al sistema de mayor funcionalidad, me quedo con haber sentado las bases de lo que será una gran herramienta para la analítica del desarrollo de *software* en el futuro.

5.3. Líneas de desarrollo futuras

Las líneas de desarrollo futuras se centran en mejorar algunas de las prestaciones de Grimoirebots.

Utilizar un sistema de colas con prioridades para la realización de los análisis simplificaría bastante el desarrollo necesario para la asignación de tareas, ya que existen componentes creados por terceros pensados para este propósito.

Permitir la creación de usuarios pero, a diferencia de Cauldron, que la autenticación no esté asociada a una fuente de datos, como GitHub o Meetup. Django posee un sistema

de gestión de usuarios muy sencillo y robusto, y permite extender los modelos básicos con mucha facilidad.

Utilizar y aprovechar las herramientas que ofrecen las plataformas *Cloud* podría mejorar el rendimiento general de la aplicación y reducir sus costes de operación. Servicios y herramientas como *AWS Lambda*, *AWS Fargate*, o *AWS RDS* podrían ser opciones válidas con las especificaciones originales del proyecto.

Desarrollar una interfaz de usuario que facilite el uso del servicio. Para preservar el principio de tener componentes desacoplados, esta interfaz debería ser desarrollada de manera independiente al *backend*, y debería interaccionar con este mediante llamadas a la API de Grimoirebots. Para hacerlo más interesante, este *frontend* debería diseñarse como un conjunto de páginas estáticas, las cuales conllevan mucho menor gasto operativo, al no necesitar un servidor que procese las solicitudes.

Por último, la inclusión de mecanismos de Integración Continua y Despliegue Continuo permitiría un mayor rendimiento y eficacia en el desarrollo de cada uno de los componentes, al reducir los tiempos de entrega y limitar el error humano.

Todo el código utilizado en este Trabajo de Fin de Máster está disponible en los siguientes repositorios de GitHub:

- <https://github.com/merinhunter/grimoirebots>
- <https://github.com/merinhunter/grimoirebots-client>
- <https://github.com/merinhunter/grimoirebots-deployment>

Bibliografía

- [1] J. Kaplan-Moss; et al. A. Holovaty. *The Django Book*. [Último acceso: 24 de Mayo de 2023]. 2009. URL: <https://web.archive.org/web/20180729171111/https://djangobook.com/introducing-django/>.
- [2] Bitergia. *GrimoireLab - Software Development and Community Analytics platform*. Último acceso: 30 de Mayo de 2023. 2017. URL: <https://chaoss.github.io/grimoirelab/>.
- [3] Cauldron. *Cauldron - Level Up Software Development Analytics*. Último acceso: 31 de Mayo de 2023. 2019. URL: <https://cauldron.io>.
- [4] T. Christie. *DRF logo*. [Último acceso: 29 de Mayo de 2023]. 2014. URL: <https://www.django-rest-framework.org/img/logo.png>.
- [5] Dependabot Contributors. *Dependabot logo*. Último acceso: 4 de Junio de 2023. 2017. URL: <https://github.com/dependabot>.
- [6] Django Software Foundation. *Django logo*. [Último acceso: 24 de Mayo de 2023]. 2005. URL: <https://www.djangoproject.com/community/logos/>.
- [7] Docker Contributors. *Docker overview | Docker Documentation*. Último acceso: 4 de Junio de 2023. 2013. URL: <https://docs.docker.com/get-started/overview/>.
- [8] GitHub Docs. *About Dependabot version updates*. Último acceso: 4 de Junio de 2023. 2020. URL: <https://docs.github.com/en/code-security/dependabot/dependabot-version-updates/about-dependabot-version-updates>.
- [9] DRF Docs. *Home*. [Último acceso: 29 de Mayo de 2023]. 2023. URL: <https://www.django-rest-framework.org>.
- [10] R. Fielding et al. *Hypertext Transfer Protocol – HTTP/1.1*. Inf. téc. RFC 2616. Último acceso: 4 de Junio de 2023. IETF, 1999. URL: <https://tools.ietf.org/html/rfc2616>.
- [11] Roy Thomas Fielding. «Architectural Styles and the Design of Network-based Software Architectures». Último acceso: 4 de Junio de 2023. PhD thesis. Irvine,

- CA: University of California, 2000. URL: https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm.
- [12] Ingrid Lunden Frederic Lardinois. *Microsoft has acquired GitHub for \$7.5B in stock*. Último acceso: 4 de Junio de 2023. 2018. URL: <https://techcrunch.com/2018/06/04/microsoft-has-acquired-github-for-7-5b-in-microsoft-stock/>.
- [13] GitHub Contributors. *Features | GitHub*. Último acceso: 4 de Junio de 2023. 2023. URL: <https://github.com/features>.
- [14] Unicorn Docs. *Gunicorn-Python WSGI HTTP Server for UNIX*. Último acceso: 29 de Mayo de 2023. 2023. URL: <https://gunicorn.org/>.
- [15] OpenSearch Contributors. *What is OpenSearch? - Open Source Search Engine Explained - AWS*. Último acceso: 4 de Junio de 2023. 2021. URL: <https://aws.amazon.com/what-is/opensearch/>.
- [16] OpenSearch Docs. *OpenSearch Dashboards - OpenSearch documentation*. Último acceso: 4 de Junio de 2023. 2021. URL: <https://opensearch.org/docs/latest/dashboards/index/>.
- [17] Simon Oxley. *GitHub Octodex*. Último acceso: 4 de Junio de 2023. 2007. URL: <https://octodex.github.com/original/>.
- [18] Poetry Docs. *Documentation*. [Último acceso: 27 de Mayo de 2023]. 2023. URL: <https://python-poetry.org/docs/>.
- [19] PostgreSQL Contributors. *PostgreSQL: About*. Último acceso: 4 de Junio de 2023. 1996. URL: <https://www.postgresql.org/about/>.
- [20] Postman. *What is Postman? Postman API Platform*. [Último acceso: 18 de Junio de 2023]. 2022. URL: <https://www.postman.com/product/what-is-postman/>.
- [21] Sebastián Ramírez. *FastAPI*. Último acceso: 3 de Junio de 2023. 2018. URL: <https://fastapi.tiangolo.com/>.
- [22] Sebastián Ramírez. *History, Design and Future - FastAPI*. Último acceso: 3 de Junio de 2023. 2018. URL: <https://fastapi.tiangolo.com/history-design-future/>.
- [23] Martin Reddy. *API Design for C++*. Último acceso: 4 de Junio de 2023. Elsevier Science, 2011.
- [24] Guido van Rossum, Barry Warsaw y Nick Coghlan. *Style Guide for Python Code*. PEP 3333. Último acceso: 29 de Mayo de 2023. 2010. URL: <https://peps.python.org/pep-3333/>.
- [25] C. Severance. «Guido van Rossum: The Early Years of Python». En: *Computer* 48.02 (feb. de 2015), págs. 7-9. ISSN: 1558-0814. DOI: 10.1109/MC.2015.45.

-
- [26] Uvicorn Docs. *Uvicorn*. Último acceso: 3 de Junio de 2023. 2018. URL: <https://www.uvicorn.org/>.
 - [27] B. Venners. *The Making of Python*. [Último acceso: 20 de Mayo de 2023]. Ene. de 2013. URL: <https://www.artima.com/articles/the-making-of-python>.
 - [28] Wikimedia Commons. *Gunicorn logo*. [Último acceso: 29 de Mayo de 2023]. 2010. URL: https://commons.wikimedia.org/wiki/File:Gunicorn_logo_2010.svg.
 - [29] Wikimedia Commons. *Postman logo*. [Último acceso: 17 de Junio de 2023]. 2021. URL: [https://commons.wikimedia.org/wiki/File:Postman_\(software\).png](https://commons.wikimedia.org/wiki/File:Postman_(software).png).
 - [30] Wikimedia Commons. *Python logo*. [Último acceso: 20 de Mayo de 2023]. 2008. URL: <https://commons.wikimedia.org/wiki/File:Python-logo-notext.svg>.