TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

UNIVERSIDAD DE ALMERIA

ESCUELA SUPERIOR DE INGENIERÍA

Construcción de infraestructura Big Data para el procesamiento y visualización de datos de Twitter

Curso 2018/2019

Alumno/a:

Mª Francisca Gálvez Lao

Director/es:

Manuel Torres Gil

IN LUMINE UNIVERSITAS

UNIVERSIDAD DE ALMERÍA

Resumen/Abstract

Twitter se ha convertido en la red social de referencia para medir el impacto de ciertos temas y noticias justo cuando estos se producen. Explorar la frecuencia o el sentimiento con el que los usuarios hablan sobre una tecnología, una marca o un personaje público es útil para conocer su prestigio y popularidad y así poder actuar en consecuencia.

En este trabajo se utiliza Apache Spark Streaming junto con Apache Kafka para extraer información de Twitter sobre los 10 motores de bases de datos más populares del momento. El resultado final será una aplicación web en la que el usuario podrá visualizar los tweets más recientes dispuestos sobre un mapa, en función de su ubicación y el tema que tratan.

Palabras clave: Twitter, Spark Streaming, Apache Kafka.

Twitter has become the reference social network to get a measure of the impact of certain topics and news just when they are born. Exploring the frequence or the sentiment with which users talk about a technology, a brand or a celebrity is useful in order to know its prestige and popularity, and so being able to act accordingly.

In this project, Apache Spark Streaming and Apache Kafka are used together to extract Twitter information about the top 10 database engines at the moment. The final result will be a website in which the user will be able to visualize the most recent tweets in a map, according to their location and the topic they talk about.

Keywords: Twitter, Spark Streaming, Apache Kafka.

GRADO EN INGENIERÍA INFORMÁTICA, 2018/2019

Trabajo Fin de Grado

Construcción de infraestructura Big Data
para el procesamiento y visualización de datos de Twitter

Alumno/a: Mª Francisca Gálvez Lao
Director: Manuel Torres Gil

Grado en Ingeniería Informática
Escuela Superior de Ingeniería
Universidad de Almería

Curso 2018/2019

Notas de la autora

La plantilla utilizada para realizar este informe es una adaptación de la publicada por José Manuel Requena Plens para el Grado en Tecnología Multimedia de la Universidad de Alicante[1].

Todos los ejemplos incluidos en este informe, así como la totalidad del código del proyecto pueden ser consultados en los siguientes repositorios:

- https://github.com/francisgalvez/twitter-analysis (módulo Spark-Kafka)
- https://github.com/francisgalvez/twitter-analysis-web (aplicación web)

[1] Disponible en GitHub: https://github.com/lcg51/tfg/

Dedicado a mi familia.

Agradecimientos

Gracias a mi tutor, Manolo, por su cercanía y por haber aportado tantísimo a este proyecto. Por las reuniones interminables en las que nos sobraban ideas para mejorar el proyecto. También por su importante labor en el Departamento de Informática y, por supuesto, como profesor.

Gracias a mi familia, por ser mi motor. Por enseñarme lo que no se aprende en ninguna escuela.

A mi madre, por su ejemplo de mujer fuerte y luchadora. Por su sensibilidad, por enseñarme el valor de la familia y el sentido del humor.

A mi padre, gracias por tu alegría y tus ganas. Gracias por enseñarme a ser independiente y pelear por lo que quiero.

A mi hermano, por ser mi compañero de batallas. Gracias por tu generosidad.

A mis compañeros de carrera, gracias por estos 4 años llenos de momentos inolvidables.

A mis amigos, por estar ahí en los buenos y malos momentos. Por apoyarme y hacerme sentir querida.

A todos mis profesores, por educarme y enseñarme a valorar cuán complicada e importante es su profesión. En especial, a todos mis profesores del Grado, gracias por vuestro cariño y confianza.

Uno nunca se da cuenta de lo que se ha hecho;
uno solo puede ver lo que queda por hacer.

Marie Curie.

Resumen

Twitter se ha convertido en la red social de referencia para medir el impacto de ciertos temas y noticias justo cuando estos se producen. Explorar la frecuencia o el sentimiento con el que los usuarios hablan sobre una tecnología, una marca o un personaje público es útil para conocer su prestigio y popularidad y así poder actuar en consecuencia.

En este trabajo se utiliza Apache Spark Streaming junto con Apache Kafka para extraer información de Twitter sobre los 10 motores de bases de datos más populares del momento[2]. El resultado final será una aplicación web en la que el usuario podrá visualizar los tweets más recientes dispuestos sobre un mapa, en función de su ubicación y el tema que tratan.

Palabras clave: Twitter, Spark Streaming, Apache Kafka.

[2] De acuerdo con la clasificación de DB-Engines (DB-Engines, marzo de 2019).

Abstract

Twitter has become the reference social network to get a measure of the impact of certain topics and news just when they are born. Exploring the frequence or the sentiment with which users talk about a technology, a brand or a celebrity is useful in order to know its prestige and popularity, and so being able to act accordingly.

In this project, Apache Spark Streaming and Apache Kafka are used together to extract Twitter information about the top 10 database engines at the moment[3]. The final result will be a website in which the user will be able to visualize the most recent tweets in a map, according to their location and the topic they talk about.

Keywords: Twitter, Spark Streaming, Apache Kafka.

[3] According to DB-Engines ranking (DB-Engines, marzo de 2019).

Índice general

1. Introducción 1
1.1. Motivación 1
1.2. Objetivos 2
1.3. Estructura de la memoria 3

2. Metodología de trabajo 5
2.1. Scrum 5
2.1.1. Roles 6
2.1.2. Artefactos 6
2.1.3. Eventos 6
2.2. Definición inicial del Product Backlog 7
2.3. Planificación de sprints 8
2.4. Resumen del capítulo 9

3. Herramientas y tecnologías 11
3.1. Python 11
3.2. API de Twitter 11
3.3. Apache Kafka 11
3.3.1. Apache Zookeeper 13
3.4. Apache Spark 14
3.4.1. Spark Streaming 15
3.5. Motores de base de datos 16
3.5.1. Elasticsearch 17
3.5.2. MongoDB 17
3.5.3. Redis 19
3.6. API 20
3.6.1. Node.js 20
3.6.2. Express 21
3.6.3. JSON Web Token 21
3.6.4. Passport 21
3.7. Docker 22
3.7.1. Docker Compose 22
3.8. Aplicación web 22
3.8.1. HTML, CSS y JavaScript 22
3.8.2. EJS 22
3.8.3. Leaflet.js 23
3.9. Postman 23
3.10. Cloud DI 23
3.10.1. Openstack 23

3.10.2. Openshift 23
3.10.2.1. Kubernetes 24
3.10.3. Redmine 25
3.11. TeamGantt 25
3.12. LaTeX 25
3.13. Editores de código e IDEs 25
3.14. Sistema de control de versiones 26
3.15. Resumen del capítulo 26

4. Desarrollo del proyecto 27
4.1. Investigación, propuesta y formación 27
4.2. Twitter Streaming API 27
4.2.1. POST statuses/filter 28
4.3. Bases de datos 32
4.3.1. Elasticsearch 33
4.3.2. Docker Compose: MongoDB y Redis 34
4.3.2.1. Creación del archivo *docker-compose.yml* 34
4.3.2.2. *Dockerfile* 36
4.3.2.3. Inicialización de bases de datos 38
4.3.2.4. Despliegue 40
4.3.2.5. *Cron job* para eliminar registros de las BBDD 40
4.4. Construcción y despliegue de clusters 41
4.4.1. Cluster Spark 41
4.4.1.1. Creación y configuración de instancias 42
4.4.1.2. Puesta en marcha del *cluster* 43
4.4.2. Cluster Kafka 44
4.5. Creación de *producer* y *consumer* 46
4.5.1. *Producer* 46
4.5.2. *Consumer* 48
4.5.2.1. *spark-submit* 53
4.6. Desarrollo de la API 54
4.6.0.1. Autorización y autenticación: JWT y Passport 57
4.6.0.1.1. Añadir JWT a la API 57
4.6.0.1.2. Añadir autenticación mediante Passport 58
4.6.0.2. Modelos 60
4.6.0.3. Rutas 62
4.6.0.3.1. Aplicación web 62
4.6.0.3.2. API REST 63
4.6.0.3.3. GeoJSON 67
4.6.1. Cliente web 68
4.6.1.1. Construcción del mapa Leaflet 69
4.6.2. Despliegue en Openshift 73
4.7. Vista previa y manejo de la aplicación final 75
4.8. Resumen del capítulo 77

5. Conclusiones y trabajo futuro **79**
5.1. Conclusiones 79
5.2. Trabajo futuro 79

Bibliografía **81**

A. Diagrama de Gantt **83**

B. Gráficas Burndown **87**

C. Modelos y esquemas Mongoose **91**

D. Capturas de funcionamiento de la aplicación **93**

E. Diagramas de colaboración **101**

F. Diagrama de casos de uso **103**

Índice de figuras

1.1. Top 10 Ranking DB-Engines, Marzo 2019. 2
1.2. Estructura de la aplicación a construir. 3

2.1. Flujo de trabajo de un proyecto Scrum. 5
2.2. Planificación inicial de los Sprints del proyecto. 9

3.1. *Topic* con múltiples particiones. 12
3.2. Estructura de un sistema de procesamiento Apache Kafka. 13
3.3. Regresión logística en Hadoop y Spark. 14
3.4. Ecosistema Spark. 15
3.5. Ejemplo de ejecución de Spark Streaming. 16
3.6. Bases de datos del sistema. 20

4.1. Ejemplo web UI de Spark. 43
4.2. Estructura de carpetas de la aplicación Node. 56
4.3. Ejemplo de JSON Web Token. 57
4.4. Vista del Service Catalog de Openshift. 73
4.5. Vista del proyecto Openshift. 74
4.6. Vista inicial de la aplicación web. 75
4.7. Vista detalle de un *tweet*. 76

B.1. Gráfico Burndown del Sprint 1. 87
B.2. Gráfico Burndown del Sprint 2. 88
B.3. Gráfico Burndown del Sprint 3. 88
B.4. Gráfico Burndown del Sprint 4. 89
B.5. Gráfico Burndown del Sprint 5. 89
B.6. Gráfico Burndown del Sprint 6. 90
B.7. Gráfico Burndown del Sprint 7. 90

D.1. Vista inicial de la aplicación web. 93
D.2. Vista detalle de un *tweet*. 94
D.3. Vista de ubicación del usuario. 94
D.4. Vista del mapa en modo nocturno. 95
D.5. Vista de búsqueda de dirección. 95
D.6. Vista *About*. 96
D.7. Vista *API Documentation*. 96
D.8. Vista *Login/Sign up*. 97
D.9. Vista registro correcto. 97
D.10. Vista registro incorrecto: Email ya existente. 98
D.11. Vista registro incorrecto: Contraseña con menos de 5 caracteres. 98

D.12. Vista registro incorrecto: Contraseñas diferentes. 99
D.13. Vista inicio de sesión incorrecto. 99
D.14. Vista *dashboard*. 100

E.1. Diagrama de colaboración: Clic sobre marcador del mapa. 101
E.2. Diagrama de colaboración: Clic sobre filtro de tiempo del mapa. 101
E.3. Diagrama de colaboración: Registro. 101
E.4. Diagrama de colaboración: Inicio de sesión. 102
E.5. Diagrama de colaboración: Cambio de contraseña. 102
E.6. Diagrama de colaboración: Cierre de sesión. 102
E.7. Diagrama de colaboración: Cron job. 102

F.1. Diagrama de casos de uso de la aplicación web. 103

Índice de tablas

4.1. Parámetros de filtrado de *tweets*. 28
4.2. Parámetros de respuesta de la API de Twitter. 30
4.3. Tipos de datos en bases de datos de *tweets*. 53
4.4. *Endpoints* de la API REST. 66

Índice de Códigos

3.1. Ejemplo de documento BSON en MongoDB. 18

4.1. Fragmento de respuesta de la API de Twitter. 31
4.2. Creación de índice ”twitter”. 33
4.3. Fragmento del archivo docker-compose.yml para la creación de los contenedores de bases de datos. 35
4.4. *Script* para creación de usuarios en bases de datos Mongo. 36
4.5. Dockerfile para la creación de bases de datos. 37
4.6. Script de inicialización de bases de datos. 38
4.7. Comandos para la construcción y el despliegue de Docker Compose. 40
4.8. Script de eliminación de registros de bases de datos. 40
4.9. Comandos para la construcción y el despliegue de Docker Compose. 41
4.10. *Script* para instalación de Spark y sus dependencias en instancia Openstack. . . . 42
4.11. Declaración de *hosts* en cada nodo. 42
4.12. Inicialización del nodo maestro desde el directorio /opt/spark. 43
4.13. Inicialización de nodos esclavos. 43
4.14. Inicialización de Zookeeper desde el directorio de instalación de Kafka. 45
4.15. Fragmento del archivo conf/server.properties. 45
4.16. Inicialización de servidor Kafka. 45
4.17. Creación de *topic* ”twitter”. 45
4.18. Seguimiento de *topic* ”twitter”. 46
4.19. *Script producer* de *tweets*. 46
4.20. Método *main* de la aplicación Spark. 49
4.21. Fragmento del método *parse_json()*. 50
4.22. Método *get_coordinates()*. 51
4.23. Método *write_to_databases()*. 52
4.24. Comando de ejecución de trabajo Spark. 54
4.25. Archivo *config.js*. 58
4.26. Archivo *service.js*. 58
4.27. Archivo *passport.js*. 59
4.28. Fragmento del archivo *index.js*. 59
4.29. Archivo *user.js*. 61
4.30. Fragmento del archivo *routes.js*. 62
4.31. Creación de clientes de bases de datos en *tweets.routes.js*. 64
4.32. Restricción de ruta a usuarios administradores. 64
4.33. Ejemplo de *endpoints* del archivo *tweets.routes.js*. 65
4.34. Ejemplo de respuesta GeoJSON. 67
4.35. Código HTML de la vista principal. 69
4.36. Creación de mapa Leaflet 70

4.37. Asignación de marcadores a capas y filtros. 71
4.38. Carga del *widget* de Twitter. 72

C.1. Modelo *tweet.js* 91
C.2. Modelo *databases.js* 91
C.3. Modelo *user.js* 92

1. Introducción

En este primer capítulo se expondrá la motivación principal por la que se decidió desarrollar este proyecto, así como los objetivos principales que se persiguen mediante su desarrollo. También se dará una visión preliminar de las tecnologías que se utilizarán durante el trabajo.

1.1. Motivación

En el año 2017 se generaron de media 2,5 quintillones de bytes al día y se estima que para 2020 cada uno de nosotros genere 1,7 megabytes de datos cada segundo (Domo, 2018). Es evidente, por tanto, que vivimos en una sociedad digital en la que Internet es la fuente principal de comunicación e información.

Sin embargo, estos datos que generamos no sólo tienen valor para nosotros como consumidores, sino que también son un activo fundamental para las empresas. Conocer el comportamiento de sus clientes y ser capaces de identificar patrones y tendencias en ellos les permite desarrollar estrategias de marketing mucho más concretas y personalizadas.

Una de las plataformas en las que más volumen de datos generamos diariamente es Twitter. El estudio de Domo, mencionado anteriormente, revela que, al día, se envían una media de 456.000 *tweets* por minuto.

Twitter es un servicio de microblogging creado en 2006 que cuenta ya con más de 500 millones de usuarios en todo el mundo. Su carácter instantáneo y global lo convierte en una de las redes sociales más importantes del momento, ya no solo para sus usuarios, sino también para las empresas, que ven en ella un ”termómetro” del comportamiento del mercado y de sus clientes.

La motivación de este Trabajo Fin de Grado, por tanto, es ser capaces de procesar *tweets* en tiempo real, en base a ciertas palabras clave por las que los filtraremos. Dichas palabras clave estarán relacionadas con un tema concreto: los 10 motores de bases de datos más populares según el ranking de DB-Engines (DB-Engines, marzo de 2019). Estos datos serán procesados y almacenados, y accesibles desde una aplicación web en la que se podrán visualizar a partir de un mapa, en función de la ubicación desde la que fueron escritos y del tema que tratan.[1]

La motivación final del proyecto, que se culmina con el Complemento de este Trabajo Fin de Grado, es poder analizar el histórico de datos valiéndonos de una herramienta de visualización, y así poder extraer conclusiones sobre los diferentes motores en materia de Inteligencia de Negocio.

[1]Nota: Dada la coincidencia entre Redis, el nombre de un motor de base de datos, y el verbo del idioma francés con el mismo nombre, se decide eliminarlo del conjunto para así no perjudicar los resultados no sólo de este TFG, sino también del análisis que se realizará en el Complemento del Trabajo Fin de Grado.

345 systems in ranking, March 2019

Rank			DBMS	Database Model	Score		
Mar 2019	Feb 2019	Mar 2018			Mar 2019	Feb 2019	Mar 2018
1.	1.	1.	Oracle	Relational, Multi-model	1279.14	+15.12	-10.47
2.	2.	2.	MySQL	Relational, Multi-model	1198.25	+30.96	-30.62
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model	1047.85	+7.79	-56.94
4.	4.	4.	PostgreSQL	Relational, Multi-model	469.81	-3.75	+70.46
5.	5.	5.	MongoDB	Document	401.34	+6.24	+60.82
6.	6.	6.	IBM Db2	Relational, Multi-model	177.20	-2.23	-9.47
7.	↑ 9.	7.	Microsoft Access	Relational	146.20	+2.18	+14.26
8.	↓ 7.	8.	Redis	Key-value, Multi-model	146.12	-3.32	+14.90
9.	↓ 8.	9.	Elasticsearch	Search engine, Multi-model	142.79	-2.46	+14.25
10.	10.	↑ 11.	SQLite	Relational	124.87	-1.29	+10.06

Figura 1.1: Top 10 Ranking DB-Engines, Marzo 2019.

1.2. Objetivos

El objetivo principal de este Trabajo Fin de Grado es la construcción de una infraestructura Big Data en la nube, capaz de procesar *streams* de datos (tweets) y almacenarlos una vez transformados. Estos datos, como se menciona en el apartado anterior, serán consumidos por una aplicación web que se encargará de la visualización de los mismos a través de un mapa.

Para la consecución del objetivo principal, habrán de llevarse a cabo una serie de objetivos secundarios:

- Estudio de herramientas y tecnologías Big Data para procesamiento en streaming.
- Selección de API en tiempo real de la que consumir datos de Twitter.
- Estudio de plataformas de procesamiento de datos en tiempo real y selección de la más adecuada.
- Investigar opciones de almacenamiento: selección del motor(es) de base de datos.
- Construcción de una API REST para realizar operaciones CRUD sobre los datos almacenados. Uso de métodos de autorización para proteger el acceso a los *endpoints* de la API.
- Investigación y selección de herramientas y/o librerías especializadas para el desarrollo de mapas interactivos.
- Estudio y selección de herramientas *open-source* de despliegue de aplicaciones en contenedores que mejor se adapten a cada uno de los módulos del proyecto.

De esta manera, se propone construir una infraestructura como la que puede observarse en la Figura 1.2.

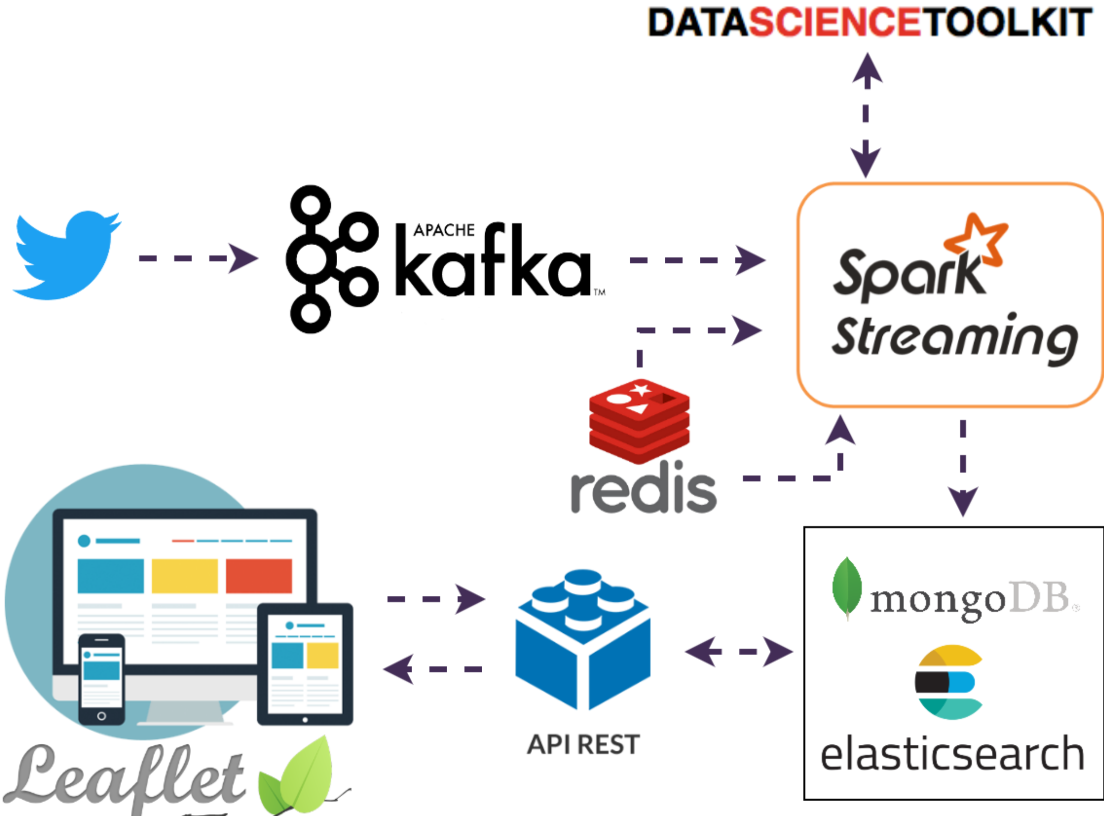

Figura 1.2: Estructura de la aplicación a construir.

1.3. Estructura de la memoria

El resto de este documento se organiza como sigue.

En el Capítulo 2 se describe la metodología en la que se ha basado el desarrollo del proyecto, así como una planificación temporal inicial.

El Capítulo 3 está dedicado a las herramientas y tecnologías utilizadas para lograr los objetivos de este Trabajo. Se introducirán y se comentará cuál es su cometido dentro del sistema.

En el Capítulo 4 se describe el proceso de desarrollo del proyecto, organizado de manera cronológica. Además, se incluye una sección con los resultados del trabajo realizado.

Por último, el Capítulo 5 basa su contenido en las conclusiones tras la finalización del proyecto y en las posibles líneas de trabajo futuro.

Además, se cuenta con varios Anexos. En el Anexo A se incluye un Diagrama de Gantt con la distribución real de las tareas en el tiempo. El Anexo B, por su parte, cuenta con todas las ventanas de la aplicación.

2. Metodología de trabajo

Tras conocer los objetivos del proyecto, es momento de definir la metodología que se seguirá para culminarlos. Desde un primer momento se toma partido por las metodologías ágiles, dado el carácter flexible y cambiante del proyecto y a la posibilidad que estas metodologías nos otorgan a la hora de redefinir ciertas tareas o funcionalidades del proyecto en pleno desarrollo.

Entre las metodologías ágiles más comunes tenemos Extreme Programming (XP), Kanban y Scrum. Se decide utilizar esta última dado su carácter incremental y dinámico.

2.1. Scrum

Scrum es una metodología ágil para el desarrollo de proyectos que se basa en una estrategia incremental, esto es, divide el producto final en pequeños productos funcionales en sí mismos, con intención de evitar una planificación completa del desarrollo al inicio del proyecto. De esta manera, no se sigue un desarrollo en cascada, sino que todas las fases del desarrollo están presentes en las diferentes iteraciones.

Scrum define una serie de elementos propios y definidos detalladamente: Roles, Artefactos y Eventos. La Figura 2.1 representa un diagrama de flujo de cualquier proyecto Scrum.

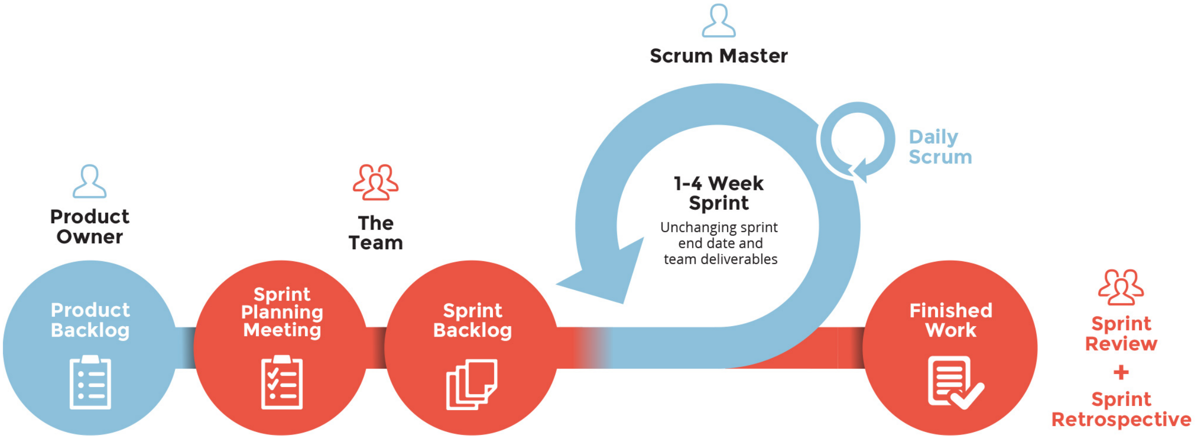

Figura 2.1: Flujo de trabajo de un proyecto Scrum.

En las siguientes subsecciones se expondrán las características principales de cada uno de estos elementos.

2.1.1. Roles

Scrum está recomendado para trabajos en equipo, en los que deben ser reconocibles los siguientes roles:

- **Product Owner:** Es el responsable de maximizar el valor del producto que resulta del trabajo del equipo de desarrollo. Además, es la persona encargada de la gestión del Product Backlog, esto es, definir sus elementos, ordenarlos y asegurarse de su entendimiento, claridad y visibilidad. En este caso, este rol es ostentado por dos personas: el tutor y la alumna trabajarán juntos para definir y manejar el Product Backlog, así como para dar lugar a un producto final de calidad y conforme a los objetivos iniciales.
- **Equipo de desarrollo (Development Team)**: Está formado por profesionales en el campo de aplicación del producto a desarrollar. Son los encargados de realizar las tareas que se les asignan durante cada *sprint*. Los equipos de desarrollo son auto-organizados, y sus miembros no pertenecen a un departamento/especialidad determinada: simplemente forman parte del equipo y pueden hacer frente a cualquiera de las tareas del Product Backlog.

  Este rol es asumido por la propia alumna.
- **Scrum Master:** Este miembro del proyecto se centra en asegurar que Scrum se lleva a cabo conforme a las directrices del Scrum Guide (Schwaber y Sutherland, 2017) y de que todos los miembros del equipo comprenden todos sus conceptos.

  Este rol será competencia del tutor de este Trabajo.

2.1.2. Artefactos

Son todos los elementos que representan algún valor para el proyecto. Encontramos 3 tipos:

- **Product Backlog:** Lista ordenada de todos los requisitos, características o mejoras del producto. Se ordena el base al valor que el Product Owner le otorga a cada tarea. Además, puede ver alterado su contenido en los períodos entre sprints.

  El responsable de este artefacto es el Product Owner.
- **Sprint Backlog:** Conjunto de tareas extraídas del Product Backlog que el equipo de desarrollo se compromete a realizar durante una iteración concreta. A diferencia del el Product Backlog, es estático, es decir, no se pueden añadir nuevas tareas durante el desarrollo del *sprint*.
- **Increment:** Es la versión del producto resultante de completar todas las tareas del Sprint Backlog. Se obtiene al final de cada iteración y debe ser funcional, con el fin de que el cliente pueda probarlo y dar su opinión sobre posibles mejoras o cambios.

2.1.3. Eventos

Los eventos en Scrum establecen cierta regularidad en el ciclo de vida de un proyecto, con el fin de poder controlar el trabajo en las diferentes iteraciones y establecer pequeños compromisos en cada una de ellas. Hay cinco eventos principales: el Sprint, el Sprint Planning, el Daily Scrum, el Sprint Review y el Sprint Retrospective.

- **Sprint:** Es el elemento principal de Scrum. Debe tener una duración de entre 2 y 4 semanas y tener como salida una versión del producto totalmente funcional en base a un objetivo fijado al comienzo de la iteración (Sprint Goal). No se puede realizar ningún cambio en el Sprint que pueda poner en peligro su objetivo.

- **Sprint Planning:** En esta reunión, celebrada antes del comienzo de cada Sprint, los diferentes miembros del equipo de desarrollo definen el objetivo del próximo Sprint y las tareas que se deben llevar a cabo para lograrlo. Todo ello bajo la supervisión del Scrum Master.

- **Daily Scrum:** Reunión diaria de los miembros del equipo de desarrollo con una duración aproximada de 15 minutos. En ella comentan los problemas experimentados el día anterior y su planificación para el día que comienza.

  En este caso no existen tales reuniones por constar el equipo de desarrollo de una sola persona.

- **Sprint Review:** Al final de cada Sprint, se celebra una reunión para revisar el trabajo realizado en la que participan todos los roles: el Scrum Master, el Product Owner y el equipo de desarrollo. En ella se explica el trabajo que se ha podido realizar y el que no, se realiza una demostración de la versión funcional del producto y, además, el equipo de desarrollo responde las dudas que puedan tener los *stakeholders* (parte interesada en el producto). También se discute si es necesario incluir alguna tarea más en el Product Backlog o si hay que cambiar o añadir alguna funcionalidad al producto para que se ajuste mejor a las necesidades de mercado de ese momento.

  El resultado de esta reunión es un Product Backlog revisado y preparado para el próximo Sprint.

- ***Sprint Retrospective*:** Oportunidad para que cada miembro del equipo realice una introspección y reflexione sobre cómo transcurrió el Sprint anterior, qué mejoras se pueden realizar y cómo, con qué problemas se encontraron, etc.

2.2. Definición inicial del Product Backlog

En cuanto al desarrollo del proyecto que nos ocupa, en el primer Sprint Planning, se definieron una serie de tareas y características que el producto final debía tener. De esta manera, se tiene una primera versión del Product Backlog con los siguientes elementos:

- **Creación de clusters (Spark y Kafka):** Creación y aprovisionamiento de instancias de máquinas virtuales para la creación de dos clusters: uno para Spark (con un nodo maestro y dos esclavos) y otro para Kafka (tres servidores o *brokers*).

- **Instalación y configuración de Zookeeper y Apache Kafka** en las máquinas del cluster.

- **Uso de MongoDB como motor de base de datos**

- **Desarrollo de *producer* y *consumer* de *tweets*:** Creación del *script* que generan los mensajes (*tweets*) y los publican al *topic* correspondiente y del trabajo Spark que los consume, los transforma y los almacena.
- **Crear aplicación web:** Desarrollo *responsive* a partir de Bootstrap.
- **Crear mapa Leaflet.js**
- **Desarrollar API con Node y Express**
- **Documentación del proyecto**

A estas tareas iniciales podrán ir añadiéndose nuevas a lo largo del ciclo de vida del proyecto. De hecho, las siguientes tareas fueron añadidas a posteriori:

- **Instalación y uso de Elasticsearch** como motor de base de datos para almacenar el histórico de *tweets*.
- **Uso de Redis** como caché para almacenar direcciones y coordenadas.
- **Despliegue de bases de datos con Docker-Compose:** Diseño y despliegue de las bases de datos Mongo y Redis como aplicación multi-contenedor.
- **Despliegue de API en Openshift:** Despliegue del módulo Node.js + Express en Kubernetes mediante Openshift.
- **Implementar seguridad en la API mediante JWT (JSON Web Token):** Asegurar ciertos *endpoints* de la API para controlar el acceso a los mismos.
- **Permitir autenticación de usuarios mediante registro e inicio de sesión**
- **Uso de EJS** para crear plantillas de elementos que se repiten en las diferentes secciones de la aplicación web.

2.3. Planificación de sprints

En Scrum, los *sprints* deben tener todos la misma duración. Ya que la fecha de inicio del proyecto es el 6 de febrero de 2019 (Sprint 0) y que se necesitan, a priori, 450 horas para completarlo (junto con el Complemento del Trabajo Fin de Grado, estrechamente ligado a este proyecto), se decide que este va a constar de 8 *sprints* con una duración de 2 semanas cada uno.

De esta manera, tenemos que, de media, cada *sprint* tendrá una duración de 56,25 horas, aunque esto es algo incierto, dado que las tareas de un Sprint pueden ver acortada o alargada su duración estimada, la cual se define al incluirlas en el Product Backlog.

En la Figura 2.2 podemos observar la planificación inicial de los *sprints* que se pretenden llevar a cabo.

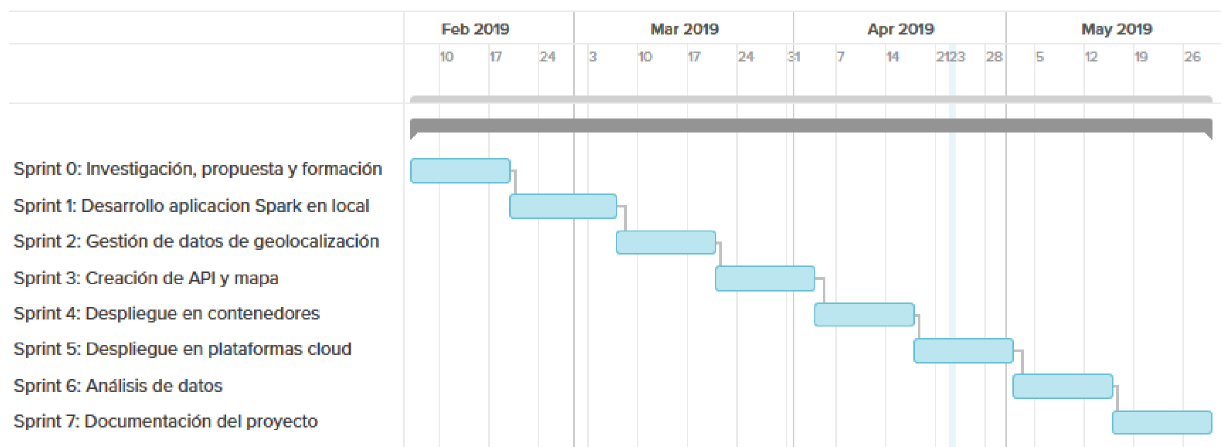

Figura 2.2: Planificación inicial de los Sprints del proyecto.

Nótese que esta planificación inicial va en contra de los principios de Scrum y que se trata de un hecho excepcional de este Trabajo Fin de Grado, cuya duración y tareas han de ser justificadas de antemano a través del documento del Anteproyecto.

Por su parte, en el Anexo A se incluye otro diagrama de Gantt; en este caso, el que se ha generado tras el desarrollo del proyecto, con todos los sprints y las tareas que se han incluido en cada uno de ellos.

Finalmente, en el Anexo B tenemos todas las gráficas Burndown que dan información sobre el número de tareas completadas en cada iteración. Recordemos que estos diagramas representan las tareas que quedan por hacer durante un período de tiempo. En el caso de Scrum, representan la evolución diaria de las tareas restantes dentro de un Sprint.

2.4. Resumen del capítulo

En este capítulo hemos introducido la metodología de desarrollo que se seguirá durante el proyecto: Scrum, una metodología ágil que nos dará flexibilidad a la hora de organizar las tareas y modificar el producto final durante el proceso.

De esta manera, se han detallado las tareas que conformaron la primera versión del Product Backlog, así como las que se fueron incluyendo tras el comienzo del proyecto.

Aunque es una práctica contraria a los principios de Scrum, se ha incluido también un Diagrama de Gantt con la planificación inicial que se previó con la entrega del Anteproyecto.

En relación con este Capítulo, se tienen los Anexos A y B, que constan del Diagrama de Gantt con la organización cronológica real de las tareas y de las gráficas Burndown asociadas a cada Sprint, respectivamente.

3. Herramientas y tecnologías

Una vez tenemos claros los objetivos del proyecto y las diferentes tareas a acometer, es el momento de seleccionar las herramientas y/o tecnologías de las que nos valdremos para llevar este Trabajo a cabo.

3.1. Python

Python será el lenguaje principal del proyecto. Con él desarrollaremos la parte de obtención y procesamiento de *tweets*, gracias a que las tecnologías que usamos (Apache Kafka y Apache Spark) proporcionan soporte para este lenguaje.

Python se ha convertido en uno de los lenguajes de programación más populares, tal y como rezan los resultados de la última encuesta de Stack Overflow (Stack Overflow, 2018, 2018). Gran parte de esta popularidad se debe a su gran potencial en el campo del análisis de datos, siendo casi imprescindible para cualquier experto en este campo. Por ello y por su facilidad de uso y comprensión se elige como lenguaje para construir la parte de generación, procesado y almacenamiento de datos, así como para la generación de pequeños *scripts* destinados a la configuración del ecosistema.

En este contexto, se utilizan librerías como ***tweepy*** (Tweepy, *Tweepy Documentation*) o ***pymongo*** (PyMongo, *PyMongo Documentation*), entre otras.

3.2. API de Twitter

El proyecto basa su propósito en la extracción de información de *tweets* sobre ciertos motores de bases de datos. Por tanto, tendremos una fuente de datos única: la API de Twitter.

En concreto usaremos una de las dos APIs que nos proporcionan *tweets* en tiempo real: la Standard Streaming API (Twitter, *Filter realtime Tweets: POST statuses/filter*). Se escoge esta dado su carácter gratuito, ya que la otra API ofrecida por Twitter (Twitter, *Filter realtime Tweets: PowerTrack API*) es una API de pago que a su vez otorga más posibilidades de filtrado y provee mayor número de *tweets* por segundo. Sin embargo, la versión gratuita será más que suficiente para poder obtener los datos que buscamos.

3.3. Apache Kafka

Apache Kafka es un sistema de procesamiento de mensajes publicador/suscriptor distribuido, particionado y replicado (Saphira, Palino y Narkhede, 2017).

Fue inicialmente desarrollado por LinkedIn en 2010 para cubrir sus necesidades de construcción de tuberías (*pipelines*) que fuesen capaces de producir *streams* de datos o, simplemente, de encolarlos. En 2012 pasó a formar parte de la *Apache Software Foundation*.

En la actualidad, es usado por empresas como Netflix, Spotify, PayPal o Cisco, entre otras.

La unidad básica en Kafka son los mensajes que son, en realidad, lotes (*batches*) de datos (en nuestro caso de *tweets*). Cada mensaje pertenece a uno o varios *topics*, los cuales están normalmente divididos en particiones con el fin de asegurar la escalabilidad y redundancia de los datos (Figura 3.1). Los mensajes y los *topics* en Kafka pueden entenderse como las filas y las tablas de una base de datos, respectivamente.

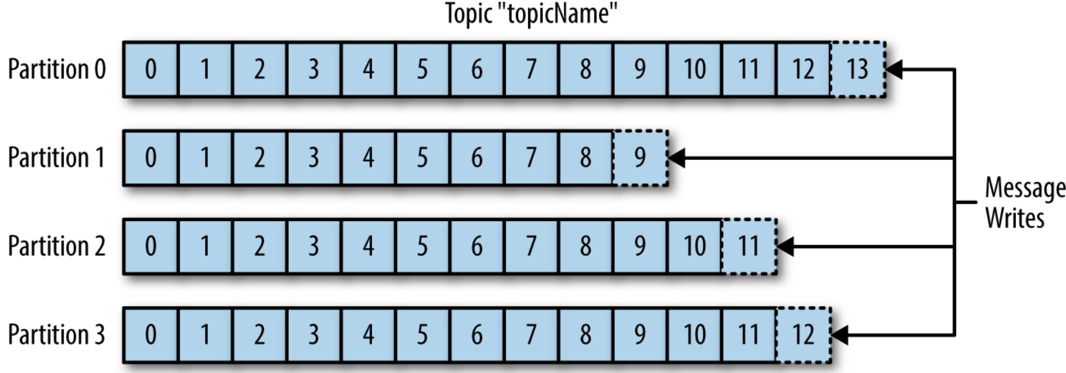

Figura 3.1: *Topic* con múltiples particiones.

De esta manera, un productor (que puede ser un script o un simple mensaje por consola) publica uno o varios mensajes a un *topic* determinado, generándose una cola de mensajes. Por defecto, el productor no dirige cada mensaje a una partición concreta, sino que se balancean para que cada partición tenga la misma carga de datos.

Por otro lado tenemos al consumidor, que estará suscrito a uno o varios *topics* y será el encargado de la ingesta y procesamiento de los mismos.

La utilidad de Kafka reside en su capacidad de admitir multiples productores y consumidores. En el primer caso, aunque por el momento solo vayamos a tener un productor (el que consulta y filtra los *tweets* de la API de Twitter), es probable que en un futuro el sistema crezca y se quieran incorporar nuevas fuentes de datos. Kafka nos garantiza la posibilidad de escalar nuestra aplicación y admitir nuevos *producers* en la misma. Por otro lado, el hecho de que Kafka admita múltiples consumidores nos permite procesar más rápidamente los *tweets* que se generan y poder ejecutar múltiples scripts en paralelo para ello.

Además, otro de sus puntos fuertes es la escalabilidad, que le permite ser capaz de manejar cualquier cantidad de datos. Es probable que, dada la escala de este Trabajo, fuera suficiente con un solo servidor o *broker* Kafka (aunque se construirá un *cluster* de 3 instancias para lograr el objetivo de construir una infraestructura Big Data). Pero en caso de que éste creciera en

tamaño, sería posible aumentar los nodos del *cluster* de manera realmente sencilla con el fin de hacer frente al procesamiento de tal cantidad de información. Esto también nos aseguraría una mayor tolerancia a fallos, pues en caso de que uno de los servidores fallase, otro de los *brokers* del cluster podría asumir las tareas del primero.

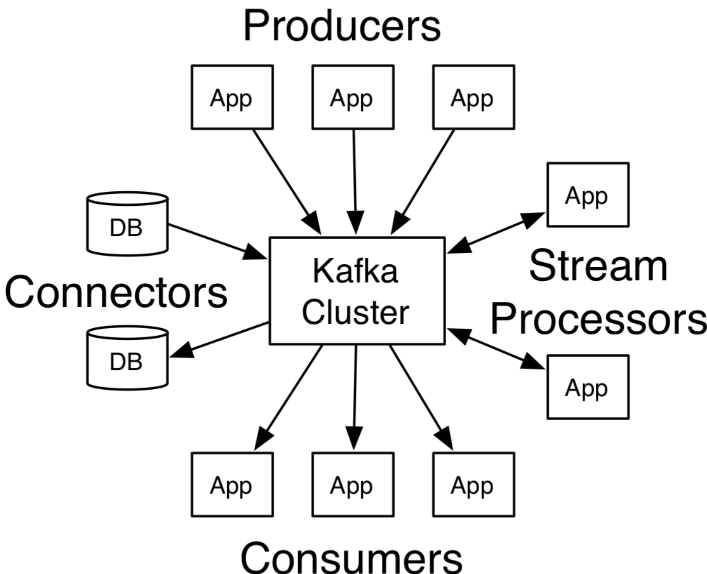

Figura 3.2: Estructura de un sistema de procesamiento Apache Kafka.

3.3.1. Apache Zookeeper

Zookeeper es un sofware desarrollado por Apache que actúa como un servicio centralizado donde se realiza el mantenimiento, configuración y sincronización de sistemas distribuidos. Kafka lo utiliza para realizar el seguimiento sobre el estado de los nodos del *cluster*, de los diferentes *topics*, sus particiones, etc. Por ejemplo, si un nodo del *cluster* falla, Zookeeper es el encargado de seleccionar otro que se encargue de sustituir al primero.

Así, las tareas principales que realiza Zookeeper en un *cluster* Kafka son:

- **Controlador:** Almacena los nodos de los que consta el *cluster* y permite el acceso a los mismos.

- **Gestión de la configuración:** Almacena la configuración de los nodos, sirviendo como copia de seguridad cuando uno de ellos falla y ha de restablecer sus servicios.

- **Sincronización:** Decide qué nodo debe ejecutar cada tarea y almacena los datos que estos han de utilizar.

3.4. Apache Spark

Spark es un motor de computación unificado y un conjunto de librerías para procesar datos de manera distribuida (Chambers y Zaharia, 2018). Actualmente es el motor de procesamiento *open source* más activo, siendo de vital conocimiento para cualquier experto en *Big Data*.

Hasta el 2005, los problemas de procesamiento de datos, simplemente, no existían. Los procesadores cada vez incrementaban más su velocidad y eran capaces de hacer frente a las necesidades de los usuarios. Sin embargo, a partir de esa fecha la tendencia de hacer los procesadores más rápidos e incluso añadir varios *cores* para el procesamiento en paralelo se hizo imposible de continuar. Era el momento de aplicar ese paralelismo a nivel de programación para mejorar la velocidad de los procesos, lo que favoreció la aparición de Spark.

Apache Spark nació como un proyecto de investigación de la Universidad de Berkeley en 2009. En ese momento, Hadoop (con su *framework* MapReduce) era la tecnología dominante para el procesamiento distribuido. Sin embargo, tenía ciertas limitaciones y dificultades. Esa fue la semilla que hizo nacer el proceso Spark: Matei Zaharia, uno de sus creadores, trabajó con usuarios de Hadoop para entender las necesidades que tenían y dicha herramienta no era capaz de proveerles. Por tanto, Spark fue desarrollado para mejorar a su predecesor y cubrir sus carencias.

Uno de los aspectos que más preocupaban a los usuarios de Hadoop era la ineficiencia de MapReduce para construir aplicaciones complejas, ya que realiza el procesamiento leyendo y escribiendo en disco. Paradójicamente, la velocidad de procesamiento se ha convertido en uno de los puntos fuertes de Spark, que, según su propia documentación[1], es hasta 100 veces más rápido que su antecesor, tal y como demuestra la Figura 3.3. Esta diferencia se debe a que Spark realiza el procesamiento en memoria.

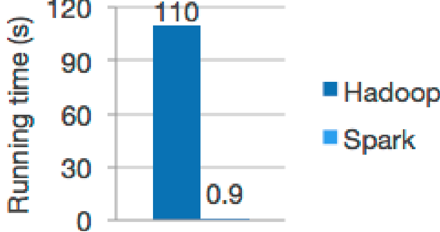

Figura 3.3: Regresión logística en Hadoop y Spark.

Así, MapReduce fue y sigue siendo adoptado como solución para procesar grandes cantidades de datos en paralelo. Para ello, descompone la información en *batches*, que son procesados en diferentes nodos que devuelven parte del resultado final. Todos estos resultados parciales se componen automáticamente para dar lugar al resultado final. También se considera una opción económica y factible cuando el tiempo de procesado no es un factor a tener en cuenta.

Por su parte, Spark es la solución más adecuada para procesamiento rápido, iterativo o en tiempo real. También para la computación de grafos y para *machine learning*.

[1]https://spark.apache.org/

En 2013, Spark ya tenía más de 100 contribuyentes de más de 30 organizaciones externas. Fue en ese año cuando Spark pasó a formar parte de la Apache Software Foundation. Desde ese momento, han visto la luz las versiones 1.0 (2014) y 2.0 (2016).

En la actualidad, Spark se ha convertido en la tecnología base para resolver problemas de datos a gran escala en compañías tan importantes como Netflix, Uber o la NASA.

Aparte de su popularidad, una de las principales razones por las que se ha escogido Apache Spark como tecnología de procesamiento es por su flexibilidad: Spark soporta multitud de lenguajes de programación (Python, Java, Scala y R), de entre los cuales se ha escogido Python para el desarrollo de este proyecto. Además, también incluye librerías para diferentes propósitos (creación de grafos, *machine learning*...), tal y como puede observarse en la Figura 3.4.

En nuestro caso, utilizaremos la librería Spark Streaming para procesar los *tweets* que consumiremos de Kafka.

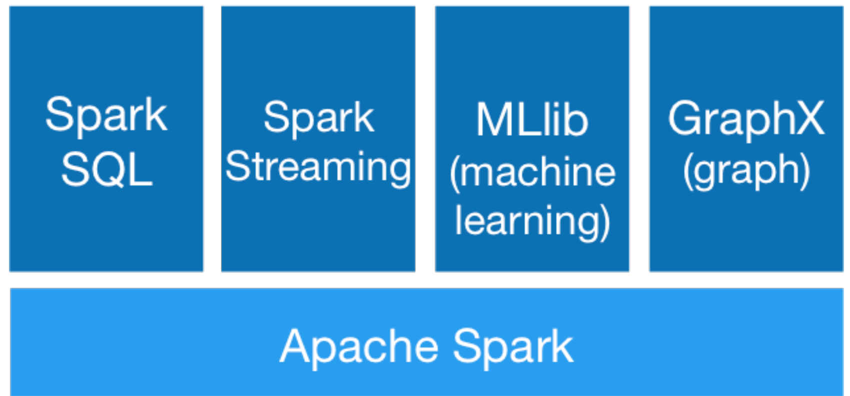

Figura 3.4: Ecosistema Spark.

3.4.1. Spark Streaming

Apache Spark Streaming es un sistema de procesamiento distribuido en tiempo real con gran tolerancia a fallos, que se usa para procesar y analizar grandes cantidades de datos. Forma parte del ecosistema Spark, tal y como podemos ver en la Figura 3.4.

Spark Streaming puede procesar datos provenientes de múltiples fuentes: Kafka, Flume, Kinesis o, incluso, de sensores y/o dispositivos conectados por TCP. Su estrategia de procesamiento reside en transformar flujos continuos de datos en pequeñas colecciones de datos sobre los que poder realizar operaciones, tal y como vemos en la Figura 3.5.

Estos pequeños bloques de datos son RDDs (Resilient Distributed Dataset), la principal abstracción de datos de Spark. Por su parte, el canal de comunicación permanente por el que los RDDs pasan al proceso Spark se conocen como *Discretized Streams* o *DStreams*.

Los RDDs se generan en función de un intervalo de tiempo determinado por el desarrollador. Por ejemplo, si establecemos el *batch interval* en 5 segundos, obtendremos los RDDs generados en los últimos 5 segundos (que pueden ser uno, varios o ninguno).

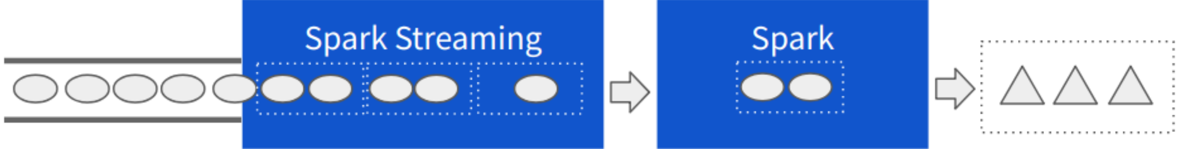

Figura 3.5: Ejemplo de ejecución de Spark Streaming.

En definitiva, los *DStreams* no son más que una abstracción proporcionada por Spark Streaming para que el motor Spark los procese y analice sin saber que, realmente, está procesando un flujo de datos y no RDDs.

Aunque la velocidad de ingesta de datos puede ser menor que la de otras tecnologías que realmente se comunican mediante *streams*, el procesamiento discretizado tiene también sus ventajas:

- **Balanceado dinámico:** Dividir los datos en pequeños fragmentos permite balancear de manera más eficiente su procesamiento entre los diferentes nodos disponibles. En un procesamiento tradicional, en el que cada mensaje se procesa según llega, podrían producirse cuellos de botella si esa partición es demasiado exigente computacionalmente.

- **Rápida recuperación ante fallos:** Spark Streaming ofrece una gran tolerancia a fallos, propiciando que cuando uno de sus nodos cae, los RDDs que le correspondía procesar se reparten equitativamente entre los nodos disponibles. Esto es posible gracias al sistema de *logs* con el que cuenta. De esta manera, cualquier otro nodo puede consultar los registros del nodo fallido y retomar la tarea que se estuviera realizando sobre los datos.

- **Interoperabilidad entre *DStreams* y *batches*:** Dado que un *DStream* es simplemente un conjunto de RDDs, es posible la interacción entre este y otras estructuras de datos estáticos, como también lo es, por tanto, la interacción entre esta librería y el resto que componen el ecosistema Spark.

- **Rendimiento:** Según sus propios desarrolladores, Spark Streaming es capaz de igualarse e incluso superar a otros sistemas de procesamiento en tiempo real (Das, Zaharia y Wendell, 2015). Además, la abstracción de los *DStreams* a menudo significa que se necesitan menos máquinas para manejar la misma carga de trabajo.

3.5. Motores de base de datos

Para la realización de este proyecto se utilizaron hasta 3 motores de base de datos distintos, cada uno con un cometido claro y justificado.

3.5.1. Elasticsearch

Elasticsearch es un motor de búsqueda especializado en texto que utiliza una potente API REST para exponer sus funcionalidades. Está orientado a documentos JSON y no utiliza esquemas, aunque pueden llegar a definirse. Además, está programado en Java y se distribuye como software de código abierto bajo las condiciones de la licencia Apache 2.0. Puede desplegarse en sistemas distribuidos.

Esta tecnología se usará para almacenar la base de datos con el histórico de *tweets*, teniendo en cuenta las necesidades de análisis posterior. De esta manera, implementaremos Kibana, una potente herramienta de visualización exclusiva para Elasticsearch, con objeto de obtener información sobre los datos recogidos a través de diferentes gráficas (Complemento del Trabajo Fin de Grado). Nos valdremos, además, de su eficaz búsqueda de texto para encontrar los temas más comentados, las palabras que más se repiten en los *tweets*, etc.

Las consultas se realizan en formato JSON a la API REST que provee Elasticsearch, aunque, en realidad, Elasticsearch está construido sobre Lucene, una API para recuperación de información programada en Java y muy utilizada en la implementación de motores de búsqueda, como en este caso. Así, las consultas en Elasticsearch son en realidad una interfaz para realizar consultas Lucene.

Una de sus principales ventajas se denomina Near Realtime (NRT o ”cerca del tiempo real”), lo cual significa que hay muy poca latencia entre el momento en que la base de datos recibe el documento y el que dicho documento puede ser buscado. Gracias a esto, podremos consultar los datos almacenados con gran rapidez y, en consecuencia, el *dashboard* de Kibana estará siempre actualizado a la última versión de los datos.

Se usará el paquete de Elasticsearch para Spark SQL (otra de las librerías Spark), que permite realizar operaciones sobre este tipo de bases de datos a través de los DataFrames SQL.

3.5.2. MongoDB

MongoDB es un sistema de base de datos NoSQL de código abierto. Al igual que Elasticsearch, está orientado a documentos, aunque en forma BSON. Este formato es una serialización codificada en binario de documentos JSON, especialmente indicada para el almacenamiento y acceso a documentos dado que, en estos casos, es más eficiente que JSON, el cual está más orientado a la transmisión de documentos en un formato más legible para el usuario. Un ejemplo de documento BSON sobre uno de los *tweets* almacenados se incluye en el Código 3.1.

Además, sus documentos no requieren de ningún esquema, sino que este es dinámico, evitando al desarrollador tener que predefinir todos los campos y sus tipos. Ello ha evitado muchos problemas en este caso, en el que después del comienzo del proyecto se fueron añadiendo nuevos campos a los registros sobre *tweets*.

En este proyecto, MongoDB almacenará las bases de datos que nutrirán al mapa con el que el usuario puede visualizar los *tweets* más recientes. Así, tendremos una base de datos Mongo por

cada filtro horario que queramos añadir al mapa.

```
{
    "_id": "5cefc19d5e9c223b7ee1e6c9",
    "id": "1134062455485865985",
    "topics": [
        "MySQL"
    ],
    "text": "Thoughtful Web Development Course: HTML, Vue.js, PHP, MySQL\n\n https://t.co/sUDaPuCY2r\n\n#php #laravel\n\nSyhWRVNV6jN https://t.co/f798pJlr0t",
    "source": "Otros",
    "hashtags_count": 2,
    "user_mentions_count": 0,
    "user_name": "LaravelTutoriaz",
    "followers": 382,
    "friends": 332,
    "verified": false,
    "geo_enabled": false,
    "location": [
        -97.997166,
        29.972907
    ],
    "longitude": -97.997166,
    "latitude": 29.972907,
    "sensitive": false,
    "lang": "en",
    "timestamp": "1559216533906",
    "date": "2019-05-30 11:42:13"
}
```

Código 3.1: Ejemplo de documento BSON en MongoDB.

Se aprovechará así una de las características principales de MongoDB: la indexación. Los índices en Mongo ”son estructuras de datos que almacenan una pequeña parte de los datos de la colección (lo que conocemos como tabla en una base de datos relacional) de tal manera que ésta pueda ser inspeccionada fácilmente” (MongoDB, *MongoDB Documentation*). De esta manera, nuestras bases de datos MongoDB se indexarán por el campo *timestamp*, con el fin de facilitar el borrado de los registros caducados.

Estas bases de datos estarán en constante cambio, dado que cada minuto se ejecutará un *script* que compruebe si todos sus registros están dentro de la franja horaria que cada una cubre y que elimine los que hayan caducado. Este es el principal motivo por el que se elige MongoDB en lugar de Elasticsearch: para tamaños pequeños de documentos, el primero tiene mejor rendimiento que el segundo[2], mientras que Elasticsearch es considerablemente más eficaz para grandes cantidades de documentos y, sobre todo, para la búsqueda de texto (algo que en el caso de estas bases de datos no es de utilidad).

[2]ver comparativa en https://bit.ly/2HOWkyt

En concreto usaremos el conector de MongoDB para Spark y Python[3]. Al igual que el paquete de Elasticsearch para Spark, este paquete se sirve de Spark SQL y de sus DataFrames para realizar operaciones sobre bases de datos Mongo.

También se utilizarán bases de datos Mongo para almacenar datos sobre configuración de la aplicación y sobre los usuarios de la misma.

Nótese la conveniencia de utilizar bases de datos orientadas a documentos, máxime teniendo en cuenta el carácter incremental del sistema a construir dado que la metodología de desarrollo es ágil. Cuando se tiene un proyecto de alcance desconocido, en el que no se tiene un esquema claro de los datos y/o módulos del sistema que se van a construir, este tipo de bases de datos facilitan el desarrollo, evitando tener que establecer compromisos respecto a los campos que tendrán los registros almacenados. De hecho, durante este desarrollo se han ido añadiendo progresivamente nuevos campos a los documentos sobre *tweets*, sin que ello afectase de ningún modo a los documentos que ya existían en las bases de datos.

Además, se escoge MongoDB por su sencillez y madurez respecto a opciones como CouchDB.

3.5.3. Redis

Redis es un almacén de datos en memoria, basado en una estructura de tipo diccionario que relaciona una clave con un valor. Opcionalmente, puede ser usada como base de datos persistente. Al igual que el resto de bases de datos utilizadas en este proyecto, Redis es un software de código abierto, en este caso bajo licencia BSD.

Una de las ventajas de Redis respecto a otros motores basados en estructuras de tipo diccionario es que no sólo soporta campos de tipo String, sino que también puede contener listas, *sets* y/o tablas *hash* de Strings. En este caso, utilizaremos este motor de base de datos como caché de direcciones en el que la clave será el literal de la dirección y el valor será un String que contenga el array con su par de coordenadas correspondiente. Es por la posibilidad de crear estructuras de datos avanzadas por lo que se escoge Redis frente a opciones como Memcached.

Cuando un usuario escribe un *tweet*, su ubicación puede tomar dos valores (no excluyentes entre sí): el lugar exacto desde el que escribe el *tweet* (coordenadas) o la dirección que especifica en su perfil. Dado que existen multitud de usuarios que no tienen habilitada la opción de ubicar sus *tweets* y que para poder visualizarlos en el mapa se necesitan su latitud y longitud, se decide hacer uso de una API que transforme una dirección (literal) en coordenadas[4].

Para evitar un elevado número de llamadas a la API se usa Redis, que para cada dirección consultada la almacenará como clave y le asignará sus coordenadas como valor. Así, si se recibe otro *tweet* de ese usuario o de otro con esa misma dirección, la tendremos cacheada en Redis y no hará falta consultar a la API. Esto supondrá un gran ahorro de tiempo a la hora de procesar cada RDD, sobre todo por la bajísima latencia de las consultas en Redis.

[3] MongoDB Connector for Spark: https://docs.mongodb.com/spark-connector/master/python-api
[4] http://www.datasciencetoolkit.org/

De esta manera, la estructura del sistema respecto a sus bases de datos quedaría como se muestra en la Figura 3.6.

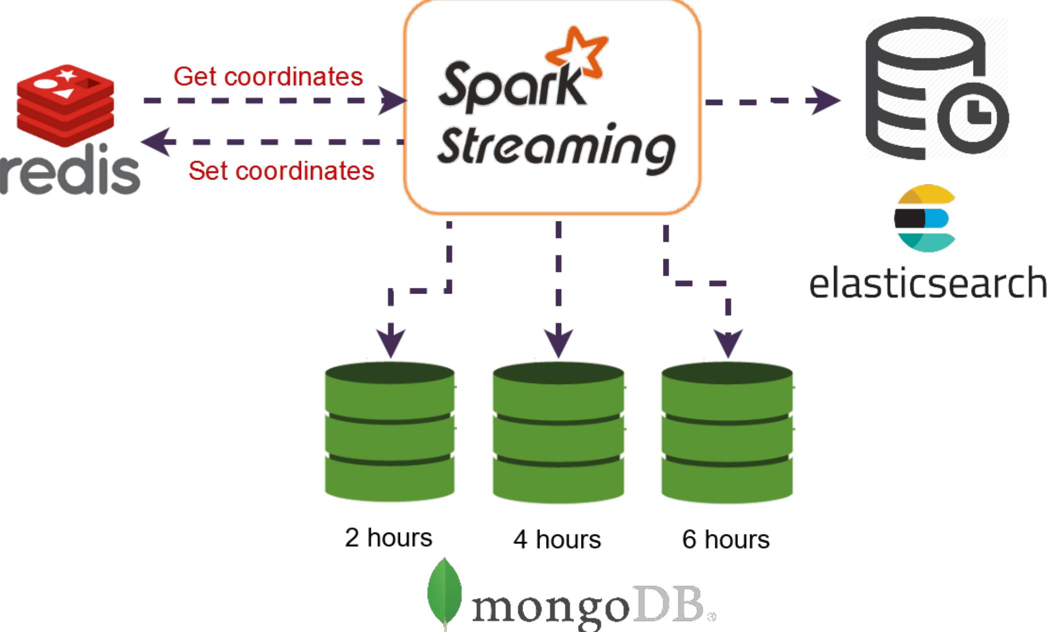

Figura 3.6: Bases de datos del sistema.

3.6. API

Para suministrar los datos a la aplicación web en la que se construirá el mapa se ha diseñado una API con Node.js, un *framework* de Javascript, y Express, su librería subyacente para la creación de rutas.

3.6.1. Node.js

Node.js es un entorno en tiempo de ejecución que permite compilar y ejecutar Javascript en el lado del servidor. Además, está orientado a eventos asíncronos, lo que le permite lograr un gran rendimiento gracias a la ejecución paralela de las tareas. En este sentido, Node funciona con un único hilo de ejecución en el que las entradas y salidas asíncronas se ejecutan concurrentemente, evitando así los cambios de contexto.

Es una tecnología multiplataforma, portable y ampliamente soportada por la mayoría de proveedores de alojamiento web.

Se escoge esta tecnología para construir la API por su velocidad, en comparación con el desarrollo en Python, Java o PHP.

3.6.2. Express

Express, según sus creadores, es una infraestructura de aplicaciones web Node.js mínima y flexible (Express.js, *Referencia de API*). Es el *framework* web más popular de Node, gratuito y de código abierto bajo licencia MIT. Proporciona mecanismos para:

- Escritura de métodos para administrar diferentes tipos de peticiones HTTP en determinadas rutas.
- Integración con motores de renderización de vistas para la inclusión de plantillas.
- Establecer ajustes de aplicaciones web, como establecimiento de puertos o localización de vistas.
- Añadir procesamiento de peticiones *middleware*.

Su éxito reside en la sencillez de su uso, además de su robustez, rapidez y flexibilidad.

En nuestra API se empleará para la creación y definición de rutas, que realizarán operaciones POST y GET sobre las diferentes bases de datos del proyecto.

3.6.3. JSON Web Token

JSON Web Token (JWT) es un estándar *open source* bajo la denominación RFC 7519 que, mediante claves o *tokens* de acceso, establece ciertos privilegios para usuarios autorizados.

JWT nos permitirá proteger los *endpoints* de la API y restringirla únicamente a usuarios registrados. Se ocupará del registro de usuarios, mientras que Passport, comentado en el apartado siguiente, se encargará del inicio de sesión.

3.6.4. Passport

Passport.js es un *middleware* para aplicaciones Node.js y Express que permite la autenticación de usuarios mediante nombre y contraseña o mediante cuentas externas como Facebook.

Se decide usar Passport como forma de permitir el acceso de los usuarios a su perfil, en el que encontrarán, aparte de información sobre su cuenta, el *token* (JWT) que les corresponde tras haberse registrado en el sistema.

Así, cada vez que un usuario se registre se le asignará un JSON Web Token con el que podrá hacer peticiones a determinadas rutas de la API. Para conocer dicho *token*, tendrá que iniciar sesión con los datos con los que se registró.

Para perpetuar las sesiones de los usuarios autenticados se utiliza el módulo ***express-session***.

3.7. Docker

Docker es una tecnología que nos permite crear y desplegar aplicaciones en contenedores, los cuales proporcionan al desarrollador una capa de abstracción respecto al sistema operativo en el que son ejecutadas. Es un proyecto de código abierto que hace uso de características de aislamiento de recursos del kernel Linux, como *cgroups* y *namespaces* para permitir que varios contenedores independientes se ejecuten dentro de una sola instancia de Linux (Docker, *Docker Documentation*).

El uso de Docker evita el mantenimiento de máquinas virtuales para el despliegue de una única aplicación. Por el contrario, favorece su existencia independiente y centralizada en una sola máquina a través de contenedores.

3.7.1. Docker Compose

Estrechamente relacionado con Docker, tenemos Docker Compose, una herramienta para definir y ejecutar aplicaciones multi-contenedores. Mediante la definición de un archivo YAML, se especificarán los servicios (contenedores Docker) que se desplegarán en el mismo Docker Compose.

Docker Compose se ha utilizado para desplegar las bases de datos MongoDB y Redis en una de las máquinas de las que consta este trabajo.

3.8. Aplicación web

3.8.1. HTML, CSS y JavaScript

HTML, CSS y JavaScript se han utilizado para el desarrollo de las vistas de la aplicación web.

Como sabemos, HTML es un conocido lenguaje para el desarrollo de páginas web basado en etiquetas. Es el lenguaje que determina el contenido de la página, mientras que CSS establece el diseño y presentación de sus elementos.

No comenzaremos el diseño de la aplicación desde cero, sino que nos valdremos de Bootstrap, conocida biblioteca de plantillas CSS, para facilitar la creación de una página *responsive* y con una apariencia atractiva para los usuarios.

Por su parte, JavaScript es un lenguaje de programación orientado a objetos que nos ayudará a manejar la parte lógica del lado del cliente. Por ejemplo, para verificar la validez de los formularios cumplimentados por el usuario. En concreto, se usarán librerías como jQuery y Leaflet.js, que será explicada en subsecciones posteriores.

3.8.2. EJS

EJS (Embedded JavaScript Templates) es un motor de plantillas basado en JavaScript.

Gracias a sus virtudes y a su facilidad de uso, nos aseguraremos la reutilización del código común a todas las vistas a través de diferentes plantillas.

3.8.3. Leaflet.js

Leaflet es una librería de Javascript de código abierto que permite crear mapas interactivos. El mapa que construiremos a partir de ella será el elemento principal de nuestra aplicación web, en el que cada *tweet* estará representado por un marcador en la localización en la que fue escrito. El usuario podrá interactuar con cada uno de los marcadores para ver el detalle del *tweet* seleccionado.

Es, sin duda, la tecnología más extendida para la construcción de mapas gracias a su simplicidad, rendimiento y usabilidad, además de la gran cantidad de *plugins* con los que cuenta. Esto ha hecho que un extenso número de empresas, como GitHub o Facebook, confíen en sus capacidades.

3.9. Postman

Esta herramienta se ha utilizado para realizar operaciones básicas sobre la base de datos Elasticsearch (recordemos que ofrece sus funcionalidades a partir de peticiones a su API) así como para realizar pruebas sobre la API construida durante el proyecto.

3.10. Cloud DI

La Universidad de Almería, y en concreto el Departamento de Informática, cuenta con una nube o *cloud* propio y privado (Cloud DI) que ofrece multitud de herramientas y servicios al mencionado Departamento. Muchos de ellos han sido utilizados en este proyecto y ha resultado imprescindibles para la consecución de sus objetivos.

3.10.1. Openstack

Openstack es una plataforma de *Cloud Computing* que permite crear nubes públicas o privadas que ofrecen a sus usuarios Infraestructuras como Servicio (IaaS). Es un proyecto de código abierto que cuenta con la financiación de multitud de empresas multinacionales, como Intel, Huawei, Cisco o Dell. Además, multitud de servicios *cloud* de importantes compañías son Openstack; es el caso de los *clouds* de IBM, Oracle o del CERN, entre otros.

Todas las máquinas en las que se han desplegado los diferentes módulos del sistema han sido proporcionadas por el Openstack del Cloud DI bajo demanda.

3.10.2. Openshift

Openshift, y en concreto la Openshift Container Platform, proporciona entornos de Kubernetes para el despliegue y administración de aplicaciones. Es lo que se conoce como PaaS (Platform as a Service). Además de poder incorporar nuestros propios archivos .yaml, también cuenta con

plantillas predefinidas para las tecnologías más populares, como es el caso de Node.js.

Gracias a esta plantilla, se ha podido desplegar la API desarrollada en un cluster de Kubernetes totalmente escalable y configurable.

3.10.2.1. Kubernetes

Kubernetes (k8s) es una herramienta de código abierto que nos permite automatizar, configurar y manejar *clusters* de contenedores. Fue desarrollado por Google y lanzado inicialmente el 7 de junio de 2014.

Esta tecnología facilita sobremanera el mantenimiento y el escalado de las aplicaciones desplegadas en contenedores. Su unidad básica son los *pods*, que almacenan las instancias de uno o más contenedores. La creación y eliminación de estos *pods* es dinámica, haciendo que sólo estén activos cuando realmente estén realizando una función. La labor conjunta de uno o varios *pods* permite desplegar un servicio (*Service*). Además, cada *pod*, durante su ciclo de vida, tiene un volumen asociado, el cual contiene los directorios y/o datos que sus contenedores necesitan para ejecutarse.

Por otro lado, en un *cluster* de Kubernetes pueden definirse uno o varios *namespaces*, que permiten separar recursos del mencionado *cluster* entre varios usuarios.

Cada *pod* tiene asignada una dirección IP dentro de cada *cluster*. Un cluster tiene los siguientes nodos:

- **Master**: Nodo maestro en el que se ejecutan tres procesos: *kube-apiserver* (servidor API para interactuar con el *cluster*), *kube-controller-manager* (para crear y replicar *pods*) y *kube-scheduler* (encargado de asignar *pods* a nodos).
- **Nodos:** Máquinas que ejecutan las aplicaciones, controladas por el nodo *master*. Ejecutan, además, dos procesos: *kubelet*, para comunicarse con el nodo *master*, y *kube-proxy*, un proxy de red que implementa los servicios de red de Kubernetes en cada nodo (Kubernetes, *Documentación de Kubernetes*).

Por último, las características principales de Kubernetes son:

- **Escalado:** En función del uso de CPU del *cluster*, se permite el aumento o reducción de *pods*, ya sea de forma manual o automática (previa definición de los indicadores).
- **Balanceo de carga:** Kubernetes garantiza la distribución de la carga de trabajo de una aplicación de forma equilibrada entre todos los nodos.
- **Autorreparación:** Cuando un contenedor falla en su ejecución o inicialización, puede ser inmediatamente reiniciado o reemplazado de manera automática.
- **Despliegues y *rollbacks* automáticos:** A la hora de actualizar una aplicación, Kubernetes se asegura de que el servicio se mantenga activo en todo momento. Para ello, va

"matando" los *pods* progresivamente mientras se asegura de que son inicializados correctamente. En caso contrario, realiza un *rollback*, esto es, vuelve al estado anterior en el que la aplicación y sus nodos funcionaban correctamente.

Kubernetes es compatible con distintos tipos de contenedores, como Docker o Rkt.

3.10.3. Redmine

Redmine ha sido otra de las herramientas del Cloud DI que se han usado en este Trabajo Fin de Grado. Esta aplicación web para la gestión de proyectos nos ha ayudado a organizar y realizar un seguimiento de los diferentes *sprints* y de sus tareas (*issues*) asociadas gracias a las posibilidades que ofrece en relación con las metodologías ágiles.

3.11. TeamGantt

TeamGantt es un servicio *online* para la creación y exportación de diagramas de Gantt. Se ha utilizado la versión gratuita de esta herramienta para generar tanto el diagrama de planificación inicial como el diagrama de Gantt definitivo, el cual puede ser localizado en el Anexo A.

3.12. LaTeX

LaTeX es un sistema de composición de textos muy utilizado en la generación de textos científicos.

Gracias a la multitud de paquetes, comandos y funcionalidades que ofrece, abstrae al usuario de la ardua tarea de formatear los documentos y le permite centrarse únicamente en el contenido.

En concreto, utilizaremos una plantilla creada por Jose Manuel Requena Plens para el Grado en Tecnologías Multimedia de la Universidad de Alicante[5] y adaptada para este Trabajo Fin de Grado.

3.13. Editores de código e IDEs

Para el desarrollo de este Trabajo se han utilizado dos editores de código:

- **JetBrains PyCharm Community Edition:** Se ha utilizado este IDE para el desarrollo con Python, esto es, para el desarrollo del módulo Kafka-Spark.
- **Visual Studio Code:** Editor de código fuente utilizado para el desarrollo de la API y de la aplicación web.
- **TeXstudio:** Entorno de desarrollo de textos LaTeX, de código abierto y multiplataforma. Utilizado para la redacción del presente informe.

[5] https://github.com/lcg51/tfg

3.14. Sistema de control de versiones

Para gestionar los cambios y versiones del proyecto se utiliza Git y, en concreto, el servicio GitHub. Esto ha permitido tener siempre a salvo todo el código que se ha ido generando durante el desarrollo de este Trabajo Fin de Grado, así como sincronizar las versiones del sistema entre todas las máquinas de las que consta.

Los repositorios en los que se puede encontrar todo el código relativo a este proyecto son:

- https://github.com/francisgalvez/twitter-analysis (módulo Spark-Kafka)
- https://github.com/francisgalvez/twitter-analysis-web (aplicación web)

3.15. Resumen del capítulo

En este tercer Capítulo se han introducido todas las tecnologías que se han usado durante el desarrollo del proyecto.

Así, tenemos que Python va a ser el lenguaje principal con el que se crearán los *scripts* Kafka y Spark. Estas dos serán las tecnologías que se utilizarán para el procesado y almacenamiento de *tweets*, que serán provistos por la Standard Streaming API de Twitter.

Los estados de Twitter se almacenarán a la vez en 4 bases de datos diferentes: una Elasticsearch, que albergará el histórico de *tweets*, y 3 MongoDB, cada una en el contexto de una franja horaria determinada. Además, se usará una caché Redis como diccionario dirección-coordenadas. Tanto las bases de datos Mongo como la caché serán desplegadas en contenedores a través de Docker Compose.

La API que nutrirá la aplicación web será desarrollada con Node.js y Express. También se usará JWT y Passport para la autorización y autenticación de usuarios. Postman será la herramienta que nos ayude a realizar pruebas sobre la API construida.

Por su parte, la aplicación web será desarrollada a partir de HTML, CSS y JavaScript. Se usará EJS como motor de plantillas y Leaflet.js para la construcción del elemento principal de la aplicación: el mapa en el que se visualizarán los *tweets*. Dicha aplicación será desplegada en Kubernetes gracias a la plataforma Openshift-DI.

Además, Openstack-DI proveerá a este proyecto de todas las instancias necesarias para su desarrollo.

Finalmente, se usará LaTeX como sistema de composición de textos de la presente memoria. En concreto, TeXStudio será la herramienta que permita documentar el proyecto de esta manera. Además, se usarán otros IDEs durante el desarrollo: PyCharm, para los desarrollos con Python, y Visual Studio Code, para la construcción de la API y la aplicación web.

Todo el código será accesible desde los dos repositorios del proyecto.

4. Desarrollo del proyecto

En este capítulo se expondrán las principales tareas de las que ha constado el proyecto, explicando detalladamente los pasos que se han seguido para completarlas.

4.1. Investigación, propuesta y formación

Durante esta fase inicial, se investigaron las diferentes opciones existentes en el mercado para poder abordar el problema inicial. Tras descubrir en Apache Spark, y más concretamente en Spark Streaming, la mejor solución para satisfacer las necesidades del proyecto, se realizó la correspondiente propuesta al tutor de este Trabajo Fin de Grado, quien dio su visto bueno.

Teniendo claro el fin del proyecto y los medios que se iban a emplear para llevarlo a cabo, se cerró esta fase con la formación de la alumna en las diferentes tecnologías y herramientas a utilizar durante el mismo.

Estas tareas constituyeron el *Sprint* 0 del desarrollo, cuyo principal objetivo era definir la idea original del TFG e identificar las herramientas de las que se iba a hacer uso para lograr su consecución. Además, sirvió para formar a los diferentes miembros del equipo en dichas herramientas, muchas de las cuales eran desconocidas, y así darles el nivel necesario para poder desempeñar su trabajo de manera fluida durante el resto de *sprints*.

Del mismo modo, tras este *Sprint* inicial se definió por primera vez el Product Backlog con todas las tareas que entrarían en juego en el *Sprint* 1 y sucesivos.

4.2. Twitter Streaming API

El proyecto tiene como fuente de datos principal la API de Twitter, más concretamente la API que nos permite filtrar *tweets* en tiempo real. En este sentido, se ha escogido la versión gratuita (Standard Streaming API), que nos permite filtrar por hasta 400 palabras clave, algo más que suficiente para nuestro caso.

Dado su carácter restringido, ha sido necesario un registro previo en la plataforma de desarrolladores de Twitter[1]. En este proceso, además de asociar una cuenta existente en dicha red social a la cuenta de desarrollador, hemos de informar a Twitter sobre qué uso se pretende dar a su API.

Tras la confirmación de registro, el siguiente paso fue la creación de una *app*, que nos proporcionaría una serie de claves de autenticación para incluirlas en el *script* que realiza las peticiones a la mencionada API.

[1] https://developer.twitter.com

4.2.1. POST statuses/filter

Con este nombre, Twitter denomina a la versión Standard de su API en tiempo real. Según su documentación, este *endpoint* devuelve estados (*tweets*) públicos que se ajustan a uno o más filtros. Los parámetros de filtrado que pueden incluirse en las peticiones son los que se especifican en la Tabla 4.1.

Nombre	Requerido	Descripción
follow	opcional	Lista de IDs de usuarios separados por comas, que indican los usuarios de los que se quieren obtener sus estados.
track	opcional	Lista de palabras clave a seguir.
locations	opcional	Conjunto de localizaciones a seguir.
delimited	opcional	Número máximo de caracteres por *tweet*.
stall_warnings	opcional	Valor *booleano* que permite a la API enviar advertencias de fallo de conexión como respuesta.

Tabla 4.1: Parámetros de filtrado de *tweets*.

En nuestro caso, el único parámetro de filtrado que incluiremos en nuestras peticiones será ”track”. Mediante este filtro, estableceremos qué palabras clave deseamos seguir, que serán aquellas relacionadas con los motores de base de datos a analizar.

Las respuestas a las peticiones tienen formato JSON y devuelven un diccionario cuyos niveles raíz son los que se muestran en la Tabla 4.2.

Atributo	Tipo	Descripción
created_at	String	Fecha (UTC) en la que se creó el *tweet*.
id	Int64	Identificador único del *tweet*.
id_str	String	Representación del ID en formato String.
text	String	Texto del estado.
source	String	Herramienta o aplicación desde la que se ha publicado (Mac, Android, iPhone, Web...).
truncated	Boolean	Indica si el texto está acortado, es decir, si excede los 140 caracteres.

Atributo	Tipo	Descripción
in_reply_to_status_id	Int64	ID del *tweet* original, en caso de que se trate de una respuesta (puede ser nulo).
in_reply_to_status_id_str	String	ID del *tweet* al que se responde, en formato String.
in_reply_to_user_id	Int64	ID del usuario al que se responde.
in_reply_to_user_id_str	String	ID del usuario al que se responde, en formato String.
in_reply_to_screen_name	String	Nombre de usuario al que se responde.
user	Objeto User	Usuario autor del *tweet*. Incluye datos sobre él, como su ID, su nombre de usuario, si está verificado, el número de seguidores, la localización que figura en su perfil, etc.
coordinates	Coordenadas	Localización geográfica del *tweet*. Puede ser nulo.
place	Places	Polígono con 4 coordenadas que especifica la región espacial desde la que el usuario escribió el estado. Puede ser nulo.
quoted_status_id	Int64	ID del *tweet* original, en caso de que el estado actual sea una cita del mismo.
quoted_status_id_str	String	ID del estado original en formato String.
is_quote_status	Boolean	Indica si el *tweet* es una cita de otro.
quoted_status	Tweet	Objeto que representa el estado citado.
retweeted_status	Tweet	*Tweet* original del que se realizó el *retweet* (*tweet* actual).
quote_count	Integer	Número de veces que el estado ha sido citado.
reply_count	Int	Número de respuestas del *tweet*.

Atributo	Tipo	Descripción
retweet_count	Int	Número de *retweets*.
favorite_count	Integer	Número de veces que otros usuarios han marcado el *tweet* como favorito.
entities	Entities	Este diccionario contiene valores como *hashtags*, URLs que aparecen en el *tweet*, usuarios mencionados, etc.
extended_entities	Extended Entities	Entidades adicionales, como fotos, vídeos, GIFs.
favorited	Boolean	Indica si el usuario que realiza la petición ha marcado el *tweet* como favorito.
retweeted	opcional	Indica si el usuario que realiza la petición ha hecho *retweer* de este estado.
possibly_sensitive	Boolean	Este parámetro sólo toma valor cuando el *tweet* contiene una URL, e indica si ésta puede tener un contenido sensible o inofensivo.
filter_level	String	Valor del nivel de filtrado a través del cual se ha obtenido esta respuesta.
lang	String	Lenguaje detectado en el *tweet*.
matching_rules	Array of Rule Objects	Conjunto de reglas que satisfacen el filtro especificado por el usuario (ID, etiquetas, etc.).

Tabla 4.2: Parámetros de respuesta de la API de Twitter.

Los parámetros más significativos dado el cariz del proyecto, además de la información sobre el usuario y sobre el texto, son los que tienen que ver con la localización del usuario que escribe el *tweet*. Lo ideal sería contar con la ubicación exacta desde la que se publicó el estado (*coordinates*), pero dado que esta es una opción que los usuarios de Twitter han de habilitar expresamente (y muy pocos lo hacen), tenemos que establecer otras vías para poder determinar la localización del autor.

Una solución es utilizar el parámetro *place*. Aunque se podría hallar el punto medio entre los cuatro pares de coordenadas dados, se opta, por ser la opción más sencilla, por acceder al primer elemento del diccionario de coordenadas que contiene y establecerlo como ubicación del *tweet*.

Otra opción es utilizar la dirección que el usuario establece en su biografía y que se incluye como parte del diccionario contenido en la clave *user*. De esta manera, presuponemos que el autor escribe el *tweet* desde su ubicación habitual.

Como necesitamos unas coordenadas exactas para poder posicionar el *tweet* sobre el mapa, en caso de tener que usar el parámetro *location* (localización según biografía) se habrá de utilizar una API que convierta una dirección (literal) en una latitud y una longitud concretas. La API en cuestión es Data Science Toolkit y en concreto su *endpoint* ”Street Address to Coordinates” (Data Science Toolkit, *Data Science Toolkit Documentation*).

```
1 {
2     [...]
3     "user": {
4     "id": 3001969357,
5     "id_str": "3001969357",
6     "name": "Jordan Brinks",
7     "screen_name": "furiouscamper",
8     "location": "Manhattan",
9     "url": "http:\/\/indigofiddle.com",
10    "description": "Alter Ego - Twitter PE",
11    "coordinates": {
12        "type": "Point",
13        "coordinates": [-73.9998279, 40.74118764]
14    },
15    "place": {
16        "id": "01a9a39529b27f36",
17        "url": "https://api.twitter.com/1.1/geo/id/01a9a39529b27f36.json",
18        "place_type": "city",
19        "name": "Manhattan",
20        "full_name": "Manhattan, NY",
21        "country_code": "US",
22        "country": "United States",
23        "bounding_box": {
24            "type": "Polygon",
25            "coordinates": [
26                [
27                    [-74.026675, 40.683935],
28                    [-74.026675, 40.877483],
29                    [-73.910408, 40.877483],
30                    [-73.910408, 40.683935]
31                ]
32            ]
33        },
34        "attributes": {}
35    }
36 }
```

Código 4.1: Fragmento de respuesta de la API de Twitter.

Dada la gran cantidad de información sobre cada *tweet* que nos devuelven las peticiones a la API, se decide mantener y almacenar únicamente aquellos campos que realmente resultan de utilidad para poder plasmarlos en el mapa y también para poder analizarlos a posteriori. Los campos de los que constarán nuestros *tweets* son:

- **id:** ID del t*tweet*.
- **text:** Texto (completo) del estado.
- **tweet_topics:** Array que contiene los temas de los que habla ese estado en relación con los filtros (palabras clave) aplicados.
- **source:** Desde qué plataforma ha sido escrito el *tweet*. Este campo es especialmente interesante en términos analíticos. Habrá 4 posibilidades: escrito desde Android, desde iPhone, desde la Web o desde ”otro” dispositivo.
- **hashtags_count:** Número de hashtags que incluye el *tweet*.
- **user_mentions_count:** Número de usuarios a los que se menciona en el estado.
- **user_name:** Nombre de usuario del autor del *tweet*.
- **followers:** Número de seguidores del autor del estado.
- **friends:** Número de personas a las que sigue el autor.
- **verified:** Valor *booleano* para indicar si es un usuario verificado por Twitter.
- **geo_enabled:** Valor que indica si el usuario tiene o no activada la ubicación de sus *tweets*.
- **location:** Localización del *tweet*. En caso de existir el campo *coordinates*, será ese el valor de este parámetro. Si su valor fuese nulo pero tuviéramos información sobre la variable *place*, cogeríamos la primera posición del diccionario *coordinates*, dentro del parámetro *bounding_box*.

  Si no tenemos ubicación exacta desde la que se escribió el estado pero sí sobre la ubicación indicada por el usuario en su perfil (*location*), extraeremos las coordenadas de dicha dirección. En caso de no obtener valores satisfactorios, este campo sería nulo y el *tweet* en cuestión no sería representado en nuestro mapa, aunque sí se almacenaría en el histórico para fines analíticos.
- **sensitive:** Almacenamos si el *tweet* es sensible u ofensivo.
- **lang:** Lenguaje en el que se escribió el estado.
- **timestamp:** Fecha en la que se publicó, en formato UNIX.
- **date:** Fecha en la que se publicó, en formato yyyy/mm/dd HH:MM:ss.

4.3. Bases de datos

Para almacenar la información sobre los *tweets*, se escogieron tres tipos de bases de datos, tal y como se mencionó en la Sección 3.5: Elasticsearch, MongoDB y Redis.

4.3.1. Elasticsearch

Esta base de datos estará dedicada en exclusiva al almacenamiento del histórico de *tweets*. En este sentido, se instaló la versión 7.0.1 de Elasticsearch en una máquina virtual con el sistema operativo Ubuntu 18.04, con 8 GB de RAM, 4 VCPUs y 80 GB de disco duro. En esa misma máquina se instaló también la misma versión de Kibana, aprovechando que se utilizaría posteriormente en el Complemento de este Trabajo Fin de Grado. Ambas herramientas se instalaron en sus puertos por defecto, es decir, el 9200 y el 5601, respectivamente.

Con vistas al posterior análisis de datos en Kibana, fue necesario definir un esquema (lo que en el entorno de Elasticsearch se conoce como *mapping*) para ciertos campos de la base de datos. Es el caso del texto, la localización y la fecha del *tweet*.

El texto se definió mediante el tipo ”text” y se estableció la propiedad *fielddata* a *true*. Este último valor es clave para poder usar las potentes funcionalidades de Elasticsearch en materia de texto: búsqueda, ordenación y agregación.

Por su parte, las coordenadas relativas a la localización del estado de Twitter se definieron del tipo *geo_point*, un tipo de datos que Elasticsearch reserva para almacenar pares de latitud y longitud. Esto nos permitirá realizar análisis sobre la geolocalización de los *tweets* en Kibana (Complemento del Trabajo Fin de Grado).

Por último, la fecha que resulta de convertir el *timestamp* (en formato UNIX) se define como tipo *date*. Además, fue necesario indicar el formato de la misma.

Así, se define el índice ”twitter_map” y su *mapping* tal y como indica el Código 4.2.

```
1  PUT "<ip>:9200/twitter_map"
2  {
3      "mappings":{
4          "properties":{
5              "text":{
6                  "type": "text",
7                  "fielddata": "true"
8              },
9              "location":{
10                 "type": "geo_point"
11             },
12             "date":{
13                 "type": "date",
14                 "format": "yyyy-MM-dd HH:mm:ss"
15             }
16         }
17     }
18 }
```

Código 4.2: Creación de índice ”twitter”.

Nótese que los *index* en Elasticsearch distan del concepto tradicional de índice dentro de una base de datos. En este caso, un *index* es lo que conocemos como base de datos.

4.3.2. Docker Compose: MongoDB y Redis

Además de la base de datos Elasticsearch, que almacena todos los registros procesados por nuestro sistema desde el inicio de su ciclo de vida, tenemos otros tipos de bases de datos, cada uno con un claro cometido dentro del proyecto.

Por una parte tenemos las bases de datos MongoDB, que almacenarán *tweets* por franjas horarias. Esto es, se crearán tantas bases de datos Mongo como filtros horarios queramos implementar en el mapa. Inicialmente, se cuenta con 3 filtros: *tweets* publicados en las últimas 2 horas, en las últimas 4 horas o en las últimas 6 horas. Esto significa que tendremos 3 bases de datos de tiempo.

También se usará MongoDB para construir otras dos bases de datos: una para almacenar la configuración sobre el sistema (en concreto sobre *topics* y bases de datos existentes) y otra que albergará la información sobre los usuarios de la aplicación web.

Por otro lado, existe una caché Redis que almacena los datos relativos a las direcciones y sus coordenadas. Recordemos que Redis se basa en el almacenamiento de pares clave-valor, los cuales en nuestro caso serán los literales de las ubicaciones de los usuarios y sus respectivas coordenadas. De esta manera, nos evitamos un número considerable de llamadas a la API para convertir direcciones en latitud y longitud, algo que enlentecería el procesado de datos. Al guardar los datos en memoria, Redis nos garantiza una baja latencia en las consultas y, por tanto, una mayor rapidez al procesar direcciones que ya se encuentren almacenadas.

En lugar de desplegar estas bases de datos de la misma manera que lo hicimos con Elasticsearch, se decidió utilizar Docker Compose para este cometido.

Desde su lanzamiento en 2013, Docker ha revolucionado el desarrollo de aplicaciones, proporcionando una capa de aislamiento respecto a los sistemas operativos que permite desplegar los contenedores de cualquier aplicación en cualquier entorno. En concreto, Docker Compose permite crear aplicaciones multicontenedores.

Dada su popularidad, el interés por su uso y la facilidad con la que permite crear, desplegar y ejecutar aplicaciones, se decide usar esta tecnología para desplegar las bases de datos MongoDB y Redis. Se instala la versión 18.08.6 de Docker y la versión 1.21.2 de Docker Compose en una máquina Ubuntu 18.04 con 4 GB de RAM, 2 VCPUs y 40 GB de disco duro.

4.3.2.1. Creación del archivo *docker-compose.yml*

La versión del archivo Compose es la 3.3, tal y como se indica en el Código 4.3. Tras definir esta propiedad en el archivo YAML, se especifican todos los servicios (contenedores) de los que constará nuestra aplicación, la cual se desplegará en otra máquina virtual Ubuntu 18.04 con 4 GB de RAM, 2 VCPUs y 40 GB de disco.

Tendremos 5 contenedores con la imagen mongo, cada uno de los cuales se encuentra escuchando en el puerto por defecto de su respectivo contenedor, pero en puertos distintos (sucesivos a partir del 27017, hasta el 27021) de la máquina en la que se despliega la aplicación.

```
version: '3.3'

services:
    mongo-system:
        image: mongo
        container_name: mongo-system
        environment:
            - 'MONGO_DB=settings'
            - 'MONGO_INITDB_ROOT_USERNAME=${MONGO_USER}'
            - 'MONGO_INITDB_ROOT_PASSWORD=${MONGO_PASSWORD}'
        volumes:
            - ./data:/data/db/system
            - ./init-mongo-system.sh:/docker-entrypoint-initdb.d/init-mongo-system.sh
        ports:
            - "27017:27017"
        restart: always

    mongo-2hours:
        image: mongo
        container_name: mongo-2hours
        environment:
            - 'MONGO_DB=twitter_2hours'
            - 'MONGO_INITDB_ROOT_USERNAME=${MONGO_USER}'
            - 'MONGO_INITDB_ROOT_PASSWORD=${MONGO_PASSWORD}'
        volumes:
            - ./data:/data/db/2hours
            - ./init-mongo-2hours.sh:/docker-entrypoint-initdb.d/init-mongo-2hours.sh
        ports:
            - "27018:27017"
        restart: always

    [...]

    mongo-seed:
        container_name: mongo-seed
        build: ./mongo-seed
        links:
            - mongo-system

    redis:
        image: redis
        container_name: redis
        environment:
            - 'REDIS_PASSWORD=${REDIS_PASSWORD}'
        ports:
            - '6379:6379'
        volumes:
            - ./data:/data/db/cache
        restart: always
```

Código 4.3: Fragmento del archivo docker-compose.yml para la creación de los contenedores de bases de datos.

Aunque podría haberse creado un contenedor único con todas las bases de datos MongoDB, mediante la creación de un contenedor para cada base de datos y la asignación de puertos distintos para cada uno conseguimos incrementar la eficacia de las operaciones sobre las mismas. De esta manera, las conexiones no tendrán que encolarse todas en el puerto 27017 de la máquina

virtual, sino que cada contenedor tendrá su puerto concreto, permitiendo el paralelismo entre conexiones a diferentes bases de datos.

Además, en el puerto de 6379 de la máquina se encuentra el servicio *redis*, que despliega la imagen de Redis en otro contenedor.

Cada uno de los contenedores de la aplicación tiene asociado un volumen en el que almacenará toda su información y que garantizará la persistencia de los datos ante cualquier posible caída del servicio. Estos volúmenes son rutas del sistema operativo sobre el que corre Docker Compose, a las que el contenedor acudirá cuando se realice un *build* para recuperar e incorporar el contenido del directorio asignado.

En este caso, los volúmenes de almacenamiento de la información siguen la ruta */data/data/db/*, seguida del directorio específico para cada contenedor. Por ejemplo, en el caso de la base de datos Redis, el volumen asociado está en la ruta */data/data/db/cache*.

Por otro lado, se definen una serie de variables de entorno (*environment*) para añadir autenticación a nuestras bases de datos. Así, mediante los archivos *init-mongo-<database>.sh* se ejecuta un comando para crear dichas bases de datos y añadirles un usuario y una contraseña, además de un rol de lectura y escritura.

Los valores de las variables de entorno están presentes en el archivo *secret.env*, el cual se mantiene fuera del sistema de control de versiones utilizado.

```
1  mongo -- "twitter_2hours" <<EOF
2      var user = '$MONGO_INITDB_ROOT_USERNAME';
3      var passwd = '$MONGO_INITDB_ROOT_PASSWORD';
4      var twitter_2hours = db.getSiblingDB('twitter_2hours');
5      twitter_2hours.auth(user, passwd);
6      db.createUser({user: user, pwd: passwd, roles: ["readWrite"]});
7  EOF
```

Código 4.4: *Script* para creación de usuarios en bases de datos Mongo.

Finalmente, para cada contenedor se establece una política de restablecimiento común (*restart*), que, mediante el valor "always" indica que siempre que los contenedores vean interrumpida su ejecución, inmediatamente restablezcan sus servicios. Esto excluye las situaciones en las que el usuario los interrumpe de forma manual y voluntaria.

4.3.2.2. *Dockerfile*

Como en todo proyecto que utilice contenedores Docker, ha de definirse un Dockerfile, esto es, un documento de texto que contiene una serie de comandos que el desarrollador considere necesarios para construir sus contenedores.

En esta aplicación, tendremos dos Dockerfiles. El primero, relativo al módulo Docker Compose, que simplemente creará un directorio para la *app* y copiará el contenido de la carpeta del proyecto

en dicho directorio para que todos los contenedores que lo necesiten puedan hacer uso de estos archivos. En segunda instancia, tendremos el Dockerfile del contenedor *mongo-seed* (Código 4.5) que tiene como propósito ejecutar un script Python que cree una serie de bases de datos Mongo:

- **settings:** Base de datos de configuración, que tendrá dos colecciones:
  - **databases:** Almacena la información sobre las bases de datos del sistema (motor, nombre, *host*, nombre de la colección...).
  - **topics:** Todas las palabras clave por las que se filtrarán los *tweets*. Los campos de esta colección son: "topics" (array con las palabras que se usarán en el filtro), "name" (nombre con el que se almacenarán las referencias a ese *topic* en la base de datos) y "keywords" (palabras clave a partir de las que se comprobará a qué *topic* o *topics* pertenece un *tweet*).
- **twitter_2hours:** Base de datos con los *tweets* de las últimas 2 horas. Indexada por el campo *timestamp*.
- **twitter_4hours:** Base de datos con los *tweets* de las últimas 4 horas. Indexada por el campo *timestamp*.
- **twitter_6hours:** Base de datos con los *tweets* de las últimas 6 horas. Indexada por el campo *timestamp*.

Aparece en este momento el concepto "indexar". Los índices son campos de los documentos que permiten ordenarlos dentro de una colección. Esto provoca que las consultas ejecutadas sobre colecciones con índices definidos sean más eficientes, como en este caso lo será cuando se quiera buscar por *timestamp*.

En este caso, es claro que las bases de datos que contienen los *tweets* dentro de una determinada franja horaria han de ser indexadas por un campo relacionado con la fecha. Ese campo es *timestamp*, que almacena la fecha de publicación del *tweet* en formato UNIX. Esto nos ayudará a realizar más rápidamente las labores de búsqueda y eliminación de los registros que ya no deban pertenecer a cierta base de datos dada su antigüedad.

```
FROM python:latest

RUN mkdir -p /usr/src/app
WORKDIR /usr/src/app
COPY databases-creation.py /usr/src/app/
COPY secret.py /usr/src/app/

RUN easy_install pymongo

ENTRYPOINT ["python3"]

CMD ["databases-creation.py"]
```

Código 4.5: Dockerfile para la creación de bases de datos.

Así, el Dockerfile del contenedor *mongo-seed* utiliza la imagen más reciente (*latest*) de Python para ejecutar el Código 4.6. Para ello, ha creado antes el directorio "app" en la ruta absoluta

/usr/src/ y copiado en él el *script* que va a ejecutar cada vez que se construya esta aplicación. También se ha tenido que instalar el módulo *pymongo*, el cliente de MongoDB para Python, indispensable para poder realizar las operaciones de creación e indexación de las diferentes bases de datos.

Como vemos, *mongo-seed* es un contenedor que, en cada construcción (*build*) de Docker Compose, ejecuta su Dockerfile y, en consecuencia, ejecuta el *script* deseado. Esta es la forma más sencilla de ejecutar código sin tener que modificar el *entrypoint* de los demás contenedores.

4.3.2.3. Inicialización de bases de datos

En *databases-creation.py*, definimos dos estructuras de documentos JSON: *databases* y *topics*, cada una con sus pertinentes campos. En el caso de la primera, no todos los campos son fijos, sino que van variando en función de la información que se necesite de los diferentes motores, en relación con los diferentes clientes que interactúan con ellas.

A continuación, se crea un cliente que se conecta al puerto 27017 (puerto por defecto de MongoDB) del contenedor *mongo-system*, definido en el *docker-compose.yml* (Código 4.3) y comprueba si la base de datos *settings* existe. En caso afirmativo, borra todo su contenido. Al tratarse de un nuevo *build* que redefinirá toda la información que antes hubiere acerca de bases de datos y *topics*, no nos interesa su contenido previo.

Se crea, entonces, la mencionada base de datos de configuración y sus colecciones, para después insertar los documentos JSON definidos al principio del *script* en sus respectivos destinos.

Finalmente, se crean las bases de datos de tiempo (en caso de que no existieran ya) y se indexan por el campo *timestamp*. Nótese que en este caso no importa que la base de datos existiera con antelación, es más, sería recomendable para así mantener la integridad de los *tweets* cotejados durante las últimas ventanas de tiempo.

```
1  from pymongo import MongoClient
2  from secret import MONGO_USER, MONGO_PASSWORD, REDIS_PASSWORD
3
4  databases = [
5                  {
6                      "engine": "elasticsearch",
7                      "name": "mainDbES",
8                      "URI": "<ip>:9200",
9                      "host": "<ip>",
10                     "port": "9200",
11                     "index": "twitter",
12                     "doc_type": "tweet"
13                 },
14
15                 {
16                     "engine": "redis",
17                     "name": "redis",
18                     "URI": "<ip>:6379/",
19                     "database_name": 0,
20                     "host": "<ip>",
21                     "port": "6379"
22                 },
```

```
23
24                     {
25                         "engine": "mongo",
26                         "name": "2hours",
27                         "URI": "<ip>:27018/",
28                         "database_name": "twitter_2hours",
29                         "collection": "coll",
30                         "time": 120
31                     },
32
33                     [...]
34             ]
35
36     topics = [
37                 {
38                     "topics": ["SQL Server", "SQLServer"],
39                     "name": "SQLServer",
40                     "keywords": ["sql server", "sqlserver"]
41                 },
42
43                 {
44                     "topics": ["PostgreSQL", "Postgres"],
45                     "name": "PostgreSQL",
46                     "keywords": ["postgres"]
47                 },
48
49                 {
50                     "topics": ["Elasticsearch"],
51                     "name": "Elasticsearch",
52                     "keywords": ["elasticsearch", "elastic"]
53                 },
54
55                 [...]
56             ]
57
58     # Crear BD "System"
59     client = MongoClient('mongodb://' + MONGO_USER + ':' + MONGO_PASSWORD + '@' + 'mongo-
           system:27017/')
60
61     dbnames = client.list_database_names()
62
63     if 'settings' in dbnames:
64         client.drop_database('settings')
65
66     settings = client['settings']
67
68     # Crear colección Databases
69     databases_coll = settings['databases']
70
71     # Crear colección topics
72     topics_coll = settings['topics']
73
74     databases_coll.insert_many(databases)
75     topics_coll.insert_many(topics, ordered=False)
76
77     twitter_2hours = client['twitter_2hours']['coll']
78     twitter_4hours = client['twitter_4hours']['coll']
79     twitter_6hours = client['twitter_6hours']['coll']
80
81     # Indexar BDs de tiempo
82     twitter_2hours.create_index("timestamp")
83     twitter_4hours.create_index("timestamp")
84     twitter_6hours.create_index("timestamp")
```

Código 4.6: Script de inicialización de bases de datos.

El motivo principal por el que se crea la base de datos *system* es la escalabilidad que esto proporciona a nuestro sistema. Como se verá en esta misma Sección y en sucesivas, a la hora de realizar operaciones sobre las distintas bases de datos no es necesario conocer qué base de datos se está tratando en cada momento (salvando las diferencias entre los distintos motores de bases de datos), en qué *host* o puerto se encuentra cada una, cuál es su nombre, por qué *topics* han de filtrarse los *tweets*... Simplemente se realizan operaciones de consulta sobre esta base de datos de configuración y se automatiza la creación de clientes, objetos o arrays para el filtrado de *tweets*, de manera que cuando se quiera extender el sistema sea tan sencillo como cambiar el contenido de sus documentos y volver a crear sus colecciones.

4.3.2.4. Despliegue

Una vez instaladas las mencionadas versiones de Docker y Docker Compose en la máquina virtual, nos desplazamos al directorio donde se encuentra el archivo YAML y ejecutamos los pertinentes comandos para construir y desplegar nuestra aplicación multicontenedores. Tras ello, ya tendríamos nuestras bases de datos preparadas para comenzar a almacenar información.

```
$ docker-compose build

$ docker-compose up
```

Código 4.7: Comandos para la construcción y el despliegue de Docker Compose.

4.3.2.5. *Cron job* para eliminar registros de las BBDD

En la misma máquina en la que desplegamos la aplicación Docker Compose, aprovechamos para programar un *cron job* que cada minuto de cada día ejecute un *script* Python que realiza una petición POST sobre la API que opera con las bases de datos. Esta petición se refiere a la eliminación de los registros ”anticuados” de las bases de datos de tiempo.

El código Python realizará un GET sobre el *endpoint* que devuelve la información sobre las bases de datos existentes en el sistema (previa identificación mediante un JWT que lo acredite como Administrador) y almacenará dicho documento JSON en una variable. De esa variable extraeremos la columna ”database_name” para cada registro a través de un bucle *for*, a la vez que realizamos la mencionada petición POST para que sea la API quien se ocupe de comprobar si hay registros que ya no se enmarcan en la franja horaria de esa base de datos y, en caso afirmativo, los elimine.

```python
import requests
from secret import TOKEN

databases = requests.get(url = 'https://whosbest-twitter-map.app.di.ual.es/api/tweets/databases', headers
    ={'Authorization': 'Bearer ' + TOKEN}).json()

for v in databases:
    if v['name']=='2hours' or v['name']=='4hours' or v['name']=='6hours':
        requests.post(url = 'https://whosbest-twitter-map.app.di.ual.es/api/tweets/delete/db/' + v['name'],
            headers={'Authorization': 'Bearer ' + TOKEN})
```

Código 4.8: Script de eliminación de registros de bases de datos.

Recordemos que *cron* es una utilidad específica del sistema operativo Unix que ejecuta procesos en segundo plano con cierta frecuencia. Cada usuario del sistema tiene una *crontab* asignada que determina los procesos a ejecutar y los intervalos de tiempo en los que hacerlo. Para ejecutar este archivo Python durante cada día de la semana, a cada hora y cada minuto, se añadió la siguiente línea en el fichero *crontab* (/etc/crontab):

```
1   * * * * * /usr/bin/python /home/ubuntu/twitter-analysis/config/databases-deletion.py
```

Código 4.9: Comandos para la construcción y el despliegue de Docker Compose.

4.4. Construcción y despliegue de clusters

En esta sección se detallará cómo se construyeron los *clusters* de máquinas virtuales para la ejecución distribuida de los *scripts* productor (Kafka) y consumidor (Spark).

4.4.1. Cluster Spark

Spark puede ser desplegado junto con herramientas de administración de *clusters*, como YARN o Mesos, pero también puede hacerlo en modo *standalone*, es decir, de manera independiente.

Apache Hadoop YARN (Yet Another Resource Negociator) es una tecnología de administración de *clusters* lanzada en la versión 2 de Hadoop. Se caracteriza como un sistema operativo distribuido a gran escala para aplicaciones Big Data.

Por su parte, Mesos, también de la Apache Software Foundation, es un administrador de *cluster* caracterizado por su abstracción respecto a los recursos del mismo, permitiendo así sistemas eficaces y de fácil construcción.

La principal diferencia entre usar estos *managers* para desplegar un cluster Spark reside en la política de distribución de trabajos entre nodos y en la manera en la que se restablecen tras cualquier fallo.

Spark por sí mismo utiliza una cola FIFO para distribuir las aplicaciones entre los nodos esclavos disponibles, mientras que Apache Mesos (Kakadia, 2015) trabaja con procesos *master* y *slave*, de manera que el primero ofrece los recursos disponibles a la aplicación que se va a ejecutar. En caso de no haber recursos suficientes, la aplicación no se ejecutará.

Por su parte, Apache Hadoop YARN (Murthy, 2014) también relaciona la programación de los trabajos con los recursos disponibles (y no con los nodos en sí mismos), de manera que los distribuye de forma igualitaria entre las aplicaciones que estén a la espera de ejecutarse. En este caso, Spark nos ofrece la opción más adecuada a nuestro problema: dado que se trata de una aplicación en *streaming*, queremos evitar la espera de los diferentes *batches* para ser asignado a uno u otro proceso en función de sus recursos. Al contrario, queremos encolar los trabajos Spark para que sean procesados por los nodos disponibles tan pronto como sea posible.

Tanto Spark como Mesos y YARN garantizan una alta disponibilidad del cluster y son capaces de recuperar los nodos en estado fallido gracias a ZooKeeper (Zookeeper, *Zookeeper Documentation*). También permiten monitorizar sus *clusters*.

Dadas las capacidades de Spark, la facilidad de construcción de un *cluster standalone* y las garantías que ofrece, se opta por este tipo de despliegue.

4.4.1.1. Creación y configuración de instancias

El primer paso para la construcción del *cluster* Spark es la creación de las instancias en Openstack. Se crean 3 instancias (un maestro y dos esclavos) con el sistema operativo Ubuntu 18.04 y el *flavour* "medium" (4 GB de RAM, 2 VCPUs y 40 GB de disco).

Para agilizar la creación de las máquinas virtuales, Openstack permite especificar un *script* que se ejecutará en la instancia una vez sea lanzada. Se aprovechó esta opción para instalar la versión 2.4.3 de Spark y sus dependencias, así como el sistema de gestión de paquetes Python 3, pip 3, y algunos módulos necesarios para la ejecución del trabajo Spark.

Además, se establecen las variables de entorno en el archivo ".bashrc", el cual determina las configuraciones para la terminal. Tendremos que incluir el directorio en el que se encuentra Spark, así como dónde se encuentra el fichero *bash* que habrá de ejecutarse cada vez que iniciemos un trabajo Spark mediante el comando *spark-submit*. También se incluirá una línea para establecer la versión Python con la que se ejecutarán los trabajos PySpark (en concreto, la versión de Python instalada en las instancias es la 3.6.7).

```
1   #!/bin/bash
2   apt update
3   apt -y upgrade
4   apt install openjdk-8-jdk
5   curl -O http://apache.rediris.es/spark/spark-2.4.3/spark-2.4.3-bin-hadoop2.7.tgz
6   tar xvf spark-2.4.3-bin-hadoop2.3.tgz
7   mv spark-2.4.3-bin-hadoop2.7/ /home/ubuntu/opt/spark
8   apt install -y python3-pip
9   pip3 install redis unidecode pymongo pandas requests
10  echo "export SPARK_HOME=/opt/spark
11            export PATH=$PATH:$SPARK_HOME/bin:$SPARK_HOME/sbin
12            export PYSPARK_PYTHON=python3" >> /home/ubuntu/.bashrc
```

Código 4.10: *Script* para instalación de Spark y sus dependencias en instancia Openstack.

De esta manera, automatizamos el proceso y evitamos tener que conectarnos a cada una de las tres máquinas para ejecutar los mismos comandos por consola.

```
1   21.0.0.15 spark-master
2   21.0.0.4 spark-worker
3   21.0.0.16 spark-worker2
```

Código 4.11: Declaración de *hosts* en cada nodo.

Finalmente, tendremos que acceder a cada una de las instancias para modificar el contenido del archivo */etc/hosts*, con el fin de determinar un nombre para cada uno de los nodos. De

esta manera, será mucho más fácil referirnos a cada uno de ellos sin tener que especificar su IP concreta.

4.4.1.2. Puesta en marcha del *cluster*

Con Spark instalado en cada uno de los nodos, el primero que se puso en marcha fue el nodo *master*[2]. En el directorio *sbin* de la carpeta de instalación de Spark nos encontramos con un archivo *bash* (*start-master.sh*) con cuya simple ejecución habremos iniciado un servidor maestro. En los registros o *logs* de dicho nodo, puede comprobarse si efectivamente el nodo se ha creado correctamente y se encuentra escuchando en el puerto 7077 (el puerto por defecto de Spark).

```
1    $./sbin/start-master.sh
```

Código 4.12: Inicialización del nodo maestro desde el directorio /opt/spark.

También se encuentra habilitado el Web UI de Spark en el puerto 8080. Esta interfaz de usuario nos da algunos detalles sobre los nodos *workers* que tiene nuestro *cluster*, el número de *cores*, la memoria que está en uso, las aplicaciones que se encuentran ejecutándose y las completadas, etc. Incluso puede accederse a una vista detalle de las diferentes aplicaciones y de los nodos del *cluster* (incluidos sus *logs*).

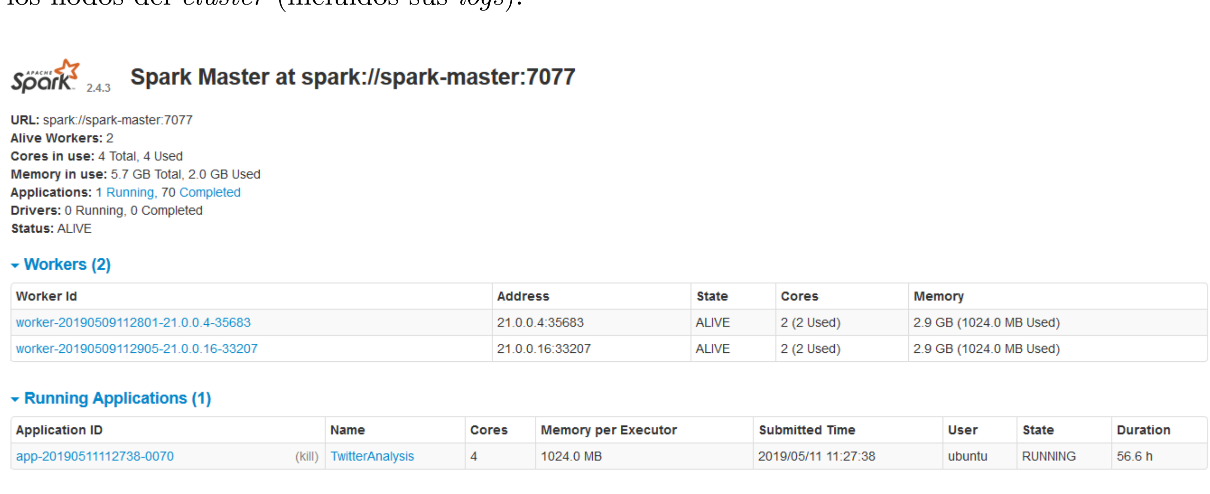

Figura 4.1: Ejemplo web UI de Spark.

A continuación, se iniciaron los nodos esclavos ejecutando el correspondiente archivo junto con la URL del nodo *master*, tal y como podemos observar en el Código 4.13. Tras comprobar la correcta creación del *cluster* en los *logs* de los nodos *slaves*, se da por concluida la puesta en marcha de nuestro *cluster* Spark.

```
1    $./sbin/start-slave.sh spark://spark-master:7077
```

Código 4.13: Inicialización de nodos esclavos.

[2]Nota: todas las conexiones a las instancias utilizadas en este proyecto se han realizado mediante SSH, previa adición del par de claves correspondiente en la configuración de la instancia en Openstack.

En el directorio *sbin* tenemos, además de los mencionados ejecutables, otros archivos que permiten realizar diferentes operaciones sobre el cluster:

- **start-slaves.sh:** Inicia todos los nodos esclavos especificados en el archivo *conf/slaves.*
- **start-all.sh:** Inicializa todos los nodos del cluster.
- **stop-master.sh:** Detiene el nodo maestro.
- **stop-slaves.sh:** Detiene la ejecución de todos los nodos esclavos.
- **stop-all.sh:** Detiene todos los nodos del cluster.

Dada la importancia del fichero *spark/conf/slaves*, es conveniente añadir una línea por cada nodo esclavo de nuestro *cluster* en el correspondiente archivo del nodo *master*. De esta manera, podremos controlar su inicialización y detención sin necesidad de conectarnos a ellos. Bastará con incluir sus nombres tal y como los definimos en el archivo */etc/hosts* para poder empezar a manejar nuestro *cluster* desde el nodo maestro.

4.4.2. Cluster Kafka

Para la construcción del *cluster* Kafka, se crearon 3 máquinas idénticas a las descritas en la subsección anterior, con la salvedad de que en este caso los comandos a ejecutar tras el lanzamiento de las instancias está relacionado con la instalación de Apache Kafka (aunque la instalación del JDK 8 también es necesaria). También se instalarán los módulos ***tweepy***, para acceder a la API de Twitter, ***kafka*** y ***pymongo***.

La versión de Kafka instalada es la 2.1.1, que usa Scala 2.12. Para la configuración de estas máquinas también ha sido necesario modificar el archivo */etc/hosts* e incluir las IPs y los nombres no sólo de los nodos de este *cluster*, sino también de los nodos Spark. Igualmente, estos últimos tienen entre sus *hosts* conocidos los *brokers* Kafka.

Nótese la conveniencia de elegir un número impar de nodos a partir de 3. Es una práctica muy común dentro de las estructuras maestro/esclavo, en las que uno de los nodos ha de ostentar el cargo de *master* obligatoriamente. En caso de fallar, otra de las instancias del *cluster* deberá postularse para el cargo y convertirse en el maestro, en ausencia del anterior quien, cuando fuese restablecido, perdería su condición previa y pasaría a ser esclavo. En consecuencia, si tuviésemos 2 nodos en nuestro *cluster* y uno de ellos fallase, se produciría un conflicto en el nodo restante.

Las distribuciones de Apache Kafka incluyen la herramienta Apache Zookeeper para la administración y mantenimiento de sus *clusters*. Normalmente, se inicializa el servidor de Zookeeper en el nodo *master*, en concreto en su puerto 2181.

Iniciamos Apache Zookeeper de manera similar a como lo hicimos con los nodos Spark: con un ejecutable *bash*. Lo haremos en modo *detached*, es decir, liberando la consola (-*daemon*), para así poder ejecutar otras órdenes posteriormente.

```
1    $./bin/zookeeper-server-start.sh -daemon config/zookeeper.properties
```

Código 4.14: Inicialización de Zookeeper desde el directorio de instalación de Kafka.

A continuación, se modificaron las propiedades de los diferentes servidores del *cluster* a través del archivo *config/server.properties*. En cada uno se tuvo que indicar un identificador único y la IP del nodo en el que se encontraba el servidor Zookeeper. También se añadieron parámetros relacionados con los *topics*: número de particiones, factor de replicación, etc. Un fragmento del mencionado archivo de una de las máquinas puede observarse en el Código 4.15

```
1    broker.id=1
2    port=9092
3    log.dirs=/tmp/kafka-logs
4    zookeeper.connect=kafka-1:2181
5    num.partitions=3
6    offsets.topic.replication.factor=3
7    transaction.state.log.replication.factor=3
8    transaction.state.log.min.isr=3
```

Código 4.15: Fragmento del archivo conf/server.properties.

Crearemos un total de 3 particiones por cada *topic*, lo cual nos garantizará un mayor número de ”consumiciones” en paralelo. El factor de replicación se establece, del mismo modo, en 3. De esta manera, cada *topic* se replicará hasta 3 veces en los nodos del cluster, por lo que garantizaremos una alta disponibilidad de los mensajes.

Modificados todos los archivos de configuración de las 3 máquinas, ha de ponerse en marcha el servidor Kafka de cada uno de los nodos ejecutando el correspondiente *bash*.

```
1    $./bin/kafka-server-start.sh -daemon config/server.properties
```

Código 4.16: Inicialización de servidor Kafka.

Una vez inicializado el *cluster* y comprobadas las conexiones entre los nodos, se creó el *topic* ”twitter”, que nos permitirá encolar los diferentes mensajes (*tweets*) generados por un *script* productor para luego consumirlos mediante la aplicación Spark.

```
1    $./bin/kafka-topics.sh --create --zookeeper kafka-1:2181 --topic twitter
```

Código 4.17: Creación de *topic* ”twitter”.

Con esto ya tendríamos completamente montado el *cluster* Kafka, a la espera de nuevos mensajes que el *producer* introdujese en el *topic* creado.

Un comando interesante para el seguimiento de un *topic* es el que se especifica en el Código 4.18. Tras ejecutarlo, se nos mostrarán por consola todos los mensajes de ese *topic* desde su creación, además de los que se van produciendo en el momento.

```
1 $./bin/kafka-console-consumer.sh --zookeeper kafka-1:2181 --topic twitter --from-beginning
```

Código 4.18: Seguimiento de *topic* ”twitter”.

Como vemos, este ejecutable crea un simple *consumer* a partir de la línea de comandos.

4.5. Creación de *producer* y *consumer*

Una vez se tiene la infraestructura completamente instalada y funcionando, es el momento de crear las aplicaciones en las que se basan la extracción y la ingesta de *tweets*: el productor (*producer*) y el consumidor (*consumer*). Como vimos en la Sección 3.3, éstos son los elementos en los que se centra el funcionamiento de Apache Kafka. Un proceso productor se encarga de generar mensajes y publicarlos en uno o varios *topics*, mientras que los procesos consumidores, suscritos a esos *topics*, los reciben.

4.5.1. *Producer*

En nuestro productor, desarrollado en Python, crearemos, en primer lugar, una instancia de *KafkaProducer*. En el constructor le pasaremos como parámetro una lista con las IPs y los puertos de las máquinas que conforman el *cluster* Kafka. Esta clase, junto con muchas otras, forma parte del cliente Python para Kafka (módulo ***kafka***).

En este proyecto, el *producer* se conectará a la API de Twitter gracias a la librería *tweepy*, específica para Python, que cuenta con multitud de métodos para establecer la conexión. Como vemos en el Código 4.19, se define una función para crear una variable de autenticación a la API, utilizando para ello el método *OAuthHandler*, que, a partir de las claves proporcionadas por Twitter para nuestra app, establece una conexión mediante el protocolo OAuth (Open Authorization).

A pesar de que *tweepy* tiene una clase propia para crear una sesión persistente con la Streaming API de Twitter (*StreamListener*), es conveniente redefinirla y ajustarla a los requisitos de nuestro *producer*. De esta manera, la clase *MyStreamListener* hereda de *StreamListener* y redefine los métodos *on_data()* y *on_error()*. El primero se asegurará de que cualquier dato que llegue a través del *stream* sea enviado al productor Kafka mediante la función asíncrona *send()*, que tiene como parámetros el *topic* al que se publicarán los mensajes y la codificación de los mismos (UTF-8).

El método *on_error()*, por su parte, define el comportamiento de la clase cuando se produce un error a la hora de recuperar los *tweets* del flujo de datos. En este caso, optamos por que se muestre el estado por pantalla.

A continuación, crearemos un objeto *Stream* incluyendo como parámetros el objeto API creado a partir del método *get_auth* y una instancia del objeto *Listener* personalizado.

```
import tweepy
from kafka import KafkaProducer
from pymongo import MongoClient
from secret import consumer_key, consumer_secret, access_token, access_token_secret, MONGO_USER, MONGO_PASSWORD

def get_auth():
    auth = tweepy.OAuthHandler(consumer_key, consumer_secret)
    auth.set_access_token(access_token, access_token_secret)
    return auth

class MyStreamListener(tweepy.StreamListener):
    def on_data(self, data):
        producer.send('twitter', data.encode('utf-8'))
        return True

    def on_error(self, status):
        print(status)

if __name__ == '__main__':
    producer = KafkaProducer(bootstrap_servers=['21.0.0.6:9092', '21.0.0.12:9092', '21.0.0.13:9092'])

    auth = get_auth()
    api = tweepy.API(auth)

    myStreamListener = MyStreamListener()
    myStream = tweepy.Stream(auth=api.auth, listener=myStreamListener)

    client = MongoClient('mongodb://' + MONGO_USER + ':' + MONGO_PASSWORD + '@' + '21.0.0.11:27017')
    topics = client['settings']['topics'].find()

    keywords = []

    for topic in topics:
        for name in topic['topics']:
            keywords.append(name)

    myStream.filter(track=keywords)
```

Código 4.19: *Script producer* de *tweets*.

Para que el *Stream* pueda obtener y filtrar *tweets*, es necesario contar con las *keywords* que definimos en la base de datos de configuración, en concreto en la colección *topics*. Por tanto, se creará una conexión con dicha base de datos y se extraerán todos los documentos de la colección.

Tal y como vimos en el Código 4.6, las palabras clave por las que deseamos obtener los comentarios sobre cada motor de base de datos no son una en todos los casos. Por ejemplo, para PostgreSQL no sólo buscaremos *tweets* con esta palabra, sino que buscaremos también "Postgres", que es una abreviatura con la que también suele referirse dicho motor. Por esto, a la hora de extraer las palabras por las que filtrar recorremos el array "topics" de cada documento y añadimos cada uno a la lista *keywords*.

Con todas las *keywords* en un array, creamos el filtro de tipo *track* mediante el correspondiente método de nuestro *Stream*.

Para ejecutar el *producer*, bastará con ejecutar el *script* Python mediante la terminal con el comando *python producer.py*.

4.5.2. *Consumer*

El consumidor de los mensajes de Twitter será un trabajo Spark que, mediante la librería Spark Streaming establecerá un *stream* permanente con el *topic* "twitter" e irá procesando y transformando los mensajes según vayan siendo publicados.

Los módulos Python que utiliza la mencionada aplicación son:

- ***pyspark:*** API de Spark para Python. En concreto, se utilizarán los módulos ***streaming*** (que incluye un subpaquete para interactuar con Apache Kafka) y ***sql***, que nos ayudará en la tarea de exportar los *tweets* a las bases de datos.
- ***requests:*** Para hacer peticiones a la API "Street Adress to Coordinates".
- ***json:*** Librería que permitirá interpretar las respuestas de la API "Street Adress to Coordinates" en formato JSON.
- ***datetime:*** Usado para convertir el campo *timestamp* en un formato más amigable.
- ***redis:*** Cliente de Redis para Python con el que realizaremos las conexiones a la caché.
- ***unidecode:*** Utilizado para codificar las direcciones y eliminar los caracteres especiales.
- ***string:*** Se usará su función *punctuation* para ayudarnos a eliminar los signos de puntuación de las direcciones.

Si nos fijamos en el método *main* de la aplicación (Código 4.20), vemos que lo primero que hace es crear cuatro objetos de tipo *SparkConf*, *SparkContext*, *StreamingContext* y *SparkSession*.

SparkConf nos permite especificar algunas de las configuraciones básicas de nuestro trabajo Spark, como su nombre, el modo de ejecución (cliente o *cluster*), el nodo *master* que se ocupará de la administración del *cluster*... Todos estos parámetros se pueden configurar tanto mediante esta instancia como a la hora de ejecutar el proceso (*spark-submit*). En este caso, sólo indicaremos el nombre de la aplicación para más tarde incluir más información en el comando de ejecución.

Con la mencionada configuración, creamos un contexto Spark, el cual representa las condiciones en las que nuestra aplicación va a ser ejecutada. Es la forma de conectar la aplicación al *cluster* existente. En este sentido, también tendremos que inicializar el contexto *Streaming* (*StreamingContext*) a través de un constructor proporcionado por el paquete *pyspark.streaming*. Como parámetros tendremos que aportar el *SparkContext* de la aplicación y una ventana temporal (lo que se conoce como *batch interval*). Esta ventana temporal será la frecuencia en segundos con la que el *DStream* será seccionado y enviado al proceso Spark para su tratamiento. Cada 10 segundos, el trabajo Spark recibirá todos los RDDs acumulados y los distribuirá entre los nodos para su procesamiento.

Por último, la sesión Spark (*SparkSession*) nos permitirá acceder a la API DataFrame para poder, a través de ella, realizar operaciones sobre las bases de datos MongoDB de nuestro sistema. Es una de las recomendaciones de los propios desarrolladores de Mongo para garantizar la satisfactoria lectura y escritura de DataFrames en sus bases de datos a través de PySpark. Se describirán más detalladamente estas estructuras de datos en párrafos sucesivos.

```
1  if __name__ == '__main__':
2      conf = SparkConf().setAppName('TwitterAnalysis')
3      sc = SparkContext(conf=conf)
4      ssc = StreamingContext(sc, 10)
5
6      spark = SparkSession \
7          .builder \
8          .appName('TwitterAnalysis') \
9          .getOrCreate()
10
11     topics = spark.read.format("com.mongodb.spark.sql.DefaultSource").option("uri", 'mongodb://' + ↩
           ↪ MONGO_USER + ':' + MONGO_PASSWORD + '@' + '21.0.0.11/settings.topics').load()
12     databases = spark.read.format("com.mongodb.spark.sql.DefaultSource").option("uri", 'mongodb://' + ↩
           ↪ MONGO_USER + ':' + MONGO_PASSWORD + '@' + '21.0.0.11/settings.databases').load()
13
14     # Conversion a DataFrame Pandas
15     topics_pandas = topics.toPandas()
16     databases_pandas = databases.toPandas()
17
18     kafkaStream = KafkaUtils.createDirectStream(ssc, topics=['twitter'], kafkaParams={'metadata.broker.list'↩
           ↪ : '21.0.0.6:9092, 21.0.0.12:9092, 21.0.0.13:9092'})
19
20     parsedJSON = kafkaStream.map(lambda x: parse_json(json.loads(x[1]), topics_pandas))
21
22     parsedJSON.foreachRDD(lambda rdd: write_to_databases(spark.createDataFrame(rdd, tweet_schema), ↩
           ↪ databases_pandas))
23
24     # Empezar ejecución del Stream
25     ssc.start()
26     ssc.awaitTermination()
```

Código 4.20: Método *main* de la aplicación Spark.

A continuación se cargan en sendas variables los contenidos de las dos colecciones de la base de datos *settings* y se convierten en DataFrames Pandas (previamente eran DataFrames SQL) para un manejo más sencillo de los mismos. A grandes rasgos, ***pandas*** es una biblioteca Python orientada principalmente al manejo y análisis de datos. Ofrece estructuras de datos, como los DataFrames, fáciles de transformar y manipular.

En la variable *kafkaStream* tendremos un flujo de datos cuya fuente es el *topic* "twitter" y en concreto la lista de servidores (*brokers*) que almacenan las particiones de dicho *topic*. A partir del método *map* realizamos una operación de transformación sobre todos los RDDs del *stream*. Este método retorna nuevos RDDs, resultado de aplicarles la función lambda. Esta función, como vemos consiste en transformar los RDDs a la forma JSON y, aparte, aplicarles el método *parse_json()*.

Dicho método extraerá de cada *tweet* los campos que se desean almacenar (especificados en la Sección 4.2.1). En concreto, resulta interesante la forma en la que se determinan los temas

sobre los que trata un *tweet*.

```
def parse_json(df, topics):

    [...]

    # Comprobación de topics
    for index, row in topics.iterrows():
        for keyword in row['keywords']:
            if keyword in text.lower():
                if row['name'] not in tweet_topics:
                    tweet_topics.append(str(row['name']))

    [...]

    if df['coordinates'] is not None:
        location = df['coordinates']['coordinates']
    else:
        if df['place'] is not None:
            location = df['place']['bounding_box']['coordinates'][0][0]
        else:
            # Si no tenemos la localización del tweet, cogemos la del usuario autor del tweet
            if df['user']['location'] is not None:
                location = get_coordinates(df['user']['location'])
            else:
                location = None
    [...]
```

Código 4.21: Fragmento del método *parse_json()*.

Como vemos en el Código 4.21, utilizamos el DataFrame que contiene la información sobre los *topics* para comprobar si alguna de sus palabras clave relacionadas están presentes en el texto. Para evitar diferencias en cuanto a la forma de dichas *keywords* en el texto (mayúsculas y minúsculas), se normaliza el texto convirtiéndolo a minúsculas. Cabe recordar que todos los elementos del campo ”keywords” de los documentos que recogen la información sobre los *topics* están, del mismo modo, en minúscula.

Para cada fila del DataFrame (que coincidiría con un documento de la base de datos) se recorren sus *keywords*. Si alguna de ellas está presente en el texto, se añade el nombre del *topic* correspondiente a la lista de *topics* sobre los que habla ese *tweet* (si es que no estuviera ya presente).

Otro aspecto destacado de este método *parse_json()* es la forma de determinar la ubicación del *tweet*. Como ya anticipamos en la Sección 4.2.1, la estrategia a seguir es la siguiente:

- **Si existen coordenadas exactas del *tweet***, es decir, si el campo *coordinates* no es nulo, la localización será esta.
- **En caso contrario:**
  - **Si tenemos un valor de *place***, es decir, si el *tweet* está localizado en una región delimitada por cuatro pares de coordenadas, cogemos la primera de ellas como ubicación.
  - **En caso contrario**, intentaremos obtener las coordenadas a partir del literal que el

usuario ha especificado como ubicación en su biografía. Si no la tuviera, tendríamos que marcar el campo *location* como nulo.

En este último supuesto, se llamaría al método *get_coordinates()* pasándole como parámetro la dirección del usuario. En él, realizamos también una serie de operaciones de normalización, como eliminar signos de puntuación, convertirla a minúsculas o eliminar los caracteres especiales. Esto hará que, al guardarla posteriormente en la caché, sea más probable encontrarla en futuras peticiones de direcciones muy similares que sólo se diferencien de esta en algún signo de puntuación, tildes o mayúsculas.

```
def get_coordinates(address):
    # Parsed address
    encoded_location = address.lower().translate(str.maketrans('', '', string.punctuation))
    # Eliminamos caracteres especiales
    decoded_location = unidecode.unidecode(encoded_location)

    response = get_cached_location(str(decoded_location))

    if response is not None:
        try:
            return json.loads(response)
        except json.decoder.JSONDecodeError:
            return None
    else:
        api_response = requests.get('http://www.datasciencetoolkit.org/maps/api/geocode/json?address=' + ↩
            ↪ str(decoded_location))

    if api_response is not None:
        try:
            api_response_dict = api_response.json()
        except json.decoder.JSONDecodeError:
            return None

        if api_response_dict['status'] == 'OK':
            latitude = api_response_dict['results'][0]['geometry']['location']['lat']
            longitude = api_response_dict['results'][0]['geometry']['location']['lng']
            set_cached_location(decoded_location, longitude, latitude)
            location = [float(longitude), float(latitude)]

            # Restringir localizaciones ficticias
            if location is not [0, 0]:
                return location
            else:
                return None
        else:
            set_cached_location(address, None, None)
            return None
    else:
        set_cached_location(address, None, None)
        return None
```

Código 4.22: Método *get_coordinates()*.

Una vez tenemos el literal normalizado, intentamos localizarlo entre las claves de nuestra caché. Si se encuentra, devolvemos las coordenadas de dicha ubicación. Si no, tendremos que realizar la correspondiente petición a la API "Street Address to Coordinates" para que nos devuelva una latitud y una longitud de la dirección que buscamos. Si la respuesta devuelve un estado exitoso, además de devolver dichas coordenadas, las incluiremos en nuestra caché junto

con la dirección a la que pertenece para así evitar futuras consultas a la API para esa misma dirección.

También si la API devuelve una respuesta nula habrá que tenerla en cuenta. Querrá decir, entonces, que esa localización que el usuario ha especificado es ficticia y no existe realmente, por lo que las próximas veces que se reciba esa misma dirección ya sabremos que no es real y no hará falta consultar a la API nuevamente.

Por último, y dado que nuestro *stream* está formado por RDDs, tendremos que ejecutar un método que recorra dicho *stream* y añada todos sus RDDs a las bases de datos del sistema. Para ello, será necesario convertirlos en DataFrames, con el fin de poder utilizar los métodos SQL que nos permiten realizar inserciones en Elasticsearch y MongoDB.

Como vemos en la línea 22 del Código 4.20, utilizamos el método *forEachRDD()* para aplicar una función lambda sobre cada RDD. Esta función consiste, por una parte, en convertirlos en DataFrames mediante la función *createDataFrame()*, perteneciente a la librería SparkSQL, que recibe como parámetros el propio RDD y un esquema del futuro DataFrame. Este esquema deberá definir una estructura (*StructType*) de columnas (*StructField*) definidas a partir de su nombre, tipo y un valor *booleano* que indique si puede tomar o no un valor nulo.

```
1 def write_to_databases(tweet, databases):
2     for index, row in databases.iterrows():
3         if row['engine'] == "elasticsearch":
4             tweet.write.format('org.elasticsearch.spark.sql').mode('append').option('es.nodes', row['host']).↩
              ↪ option('es.port', int(row['port'])).option('es.resource', row['index'] + "/" + row['doc_type↩
              ↪ ']).save()
5         elif row['engine'] == "mongo":
6             URI = str(row['URI'] + row['database_name'] + "." + row['collection'])
7             tweet.write.format('com.mongodb.spark.sql.DefaultSource').mode('append').option('uri', URI).↩
              ↪ save()
```

Código 4.23: Método *write_to_databases()*.

El esquema aplicado puede observarse en la Tabla 4.3.

Los DataFrames SQL resultantes se incluirán junto con el DataFrame *databases* como parámetros del método *write_to_databases()*.

Así, de manera similar a como se recorrían los *topics*, se van recorriendo los registros con los nombres y ubicaciones de las bases de datos. Ya que cada motor de base de datos tiene su propio formato de escritura y sus propias opciones personalizables, tendremos que distinguirlas por el campo *engine* y ejecutar el correspondiente método *write* en cada caso.

A pesar de todo esto, sin las últimas dos líneas de nuestro *consumer* no habrá comenzado el procesamiento de los datos. Mediante el método *start()* del *StreamingContext*, iniciamos la aplicación, que no terminará hasta indicación expresa del usuario, tal y como señala la llamada al método *awaitTermination()*.

Nombre	**Tipo**	**Nulo**
id	String	No
topics	Array de Strings	No
text	String	No
source	String	No
hashtags_count	Integer	No
user_mentions_count	Integer	No
user_name	String	No
followers	Integer	No
friends	Integer	No
verified	Boolean	No
geo_enabled	Boolean	No
location	Array de Double	Sí
longitude	Double	Sí
latitude	Double	Sí
sensitive	Boolean	Sí
lang	String	No
timestamp	String	No
date	String	No

Tabla 4.3: Tipos de datos en bases de datos de *tweets*.

4.5.2.1. *spark-submit*

Una vez se tiene el *consumer* preparado para su ejecución, los *clusters* Kafka y Spark están disponibles y el *script* productor ejecutándose, es momento de lanzar el trabajo Spark. Para ello, se ejecutará un comando desde la consola que incluirá bastante información sobre configuración y paquetes necesarios para ejecutar el proceso.

Como veíamos en párrafos anteriores, la configuración de la aplicación puede realizarse tanto a partir del *SparkConf* como del propio *spark-submit*, esto es, el comando de ejecución de un trabajo Spark.

```
1 $ /opt/spark/bin/spark-submit --master spark://spark-master:7077 --packages org.apache.spark:spark-↩
    ↪ streaming-kafka-0-8_2.11:2.2.0,org.mongodb.spark:mongo-spark-connector_2.11:2.4.0,org.↩
    ↪ elasticsearch:elasticsearch-spark-20_2.11:7.0.0 local:///home/ubuntu/twitter-analysis/spark-↩
    ↪ consumer.py
```

Código 4.24: Comando de ejecución de trabajo Spark.

El que se muestra en el Código 4.24 es el comando utilizado para la ejecución del *consumer*. A continuación se listan y especifican sus partes:

- ***–master:*** Mediante esta opción, indicamos la URL del nodo *master* del *cluster* en el que se va a ejecutar la aplicación. Va seguido de la URL con la forma *spark://<host>:<port>*.

- ***–deploy-mode:*** Puede tomar dos valores: *cluster* o *client*, que dependen de la localización del controlador del proceso. En el primer caso, se delega la tarea en uno de los nodos esclavos del *cluster*, por lo que la consola desde la que se ejecuta el trabajo Spark queda liberada y los resultados o *logs* del mismo se almacenan en el nodo correspondiente. Por su parte, el modo cliente hace que sea la máquina desde la que se lanza el proceso la que actúe como *driver*. Este modo está especialmente recomendado para aquellas situaciones en las que la máquina ”ejecutora” está en la misma red que las instancias del *cluster*, mientras que el modo *cluster* está diseñado para minimizar la latencia de red entre los *drivers* y los nodos que ejecutan el código Spark, por lo que es usado para lanzar trabajos a máquinas alejadas de los nodos trabajadores (p.e. si se tiene un *cluster* en AWS).

- ***–packages:*** Esta opción permite añadir paquetes que sean necesarios para la ejecución del *script*. En este caso, se añaden las dependencias relacionadas con Kafka, MongoDB y Elasticsearch.

- ***local://:*** Ruta absoluta del *script* a ejecutar, que debe estar presente en todos los nodos *workers*. También puede tomar otros valores, como *file*, para rutas locales servidas mediante HTTP a los diferentes nodos, *hdfs*, si se usa este sistema de archivos, o *http* si está localizado en una URL específica.

4.6. Desarrollo de la API

Para la construcción de la API se han utilizado Node.js y Express en conjunción con varias bases de datos Mongo.

El primer paso para comenzar el desarrollo fue instalar Node en el equipo personal de la alumna. En concreto, se instaló la versión 10.13.0. Junto con Node, se instaló el sistema de instalación de paquetes **npm**, que sirvió para instalar posteriormente todas las dependencias de esta parte del proyecto.

Algunos de los módulos que se han utilizado son:

- ***bcryptjs:*** Librería para encriptación de literales, utilizada para encriptar las contraseñas de los usuarios (que se encuentran almacenadas en una base de datos). Versión 2.4.3.

- ***body-parser:*** *Middleware* que permite manejar peticiones POST y extraer la información contenida en el *body*. Versión 1.19.0.
- ***cookie-parser:*** Paquete para el manejo de *cookies* a través de Express. Versión 1.4.4.
- ***elasticsearch:*** Librería para creación de clientes Elasticsearch. Versión 15.4.1.
- ***express:*** *Framework* para Node que permite construir APIs. En la actualidad es el servidor estándar de facto para Node.js. Versión 4.16.4.
- ***express-session:*** Permite crear sesiones en Express. Versión 1.16.1.
- ***geojson:*** Módulo para la generación de documentos GeoJSON, formato de respuesta a las peticiones GET. Versión 0.5.0.
- ***jsonwebtoken:*** Permite generar JWT y comprobar la autorización de un determinado usuario dado cierto *token*. Versión 8.5.1.
- ***mongoose:*** Herramienta para modelado de objetos orientada a MongoDB y a ecosistemas asíncronos. Versión 5.4.19.
- ***morgan:*** Utilizado para obtener detalles sobre los *logs* de Express. Versión 1.9.1.
- ***nodemon:*** Herramienta de desarrollo para la ejecución de Node que restablece la aplicación automáticamente tras detectar cambios, entre otras muchas funciones. Versión 1.18.10.
- ***passport:*** *Middleware* para autenticación de usuarios en Express. Versión 0.4.0.

La estructura de carpetas que fue necesaria para el desarrollo de la API se detalla en la Figura 4.2, con mayor nivel de detalle sobre la carpeta *server*, relativa a la parte servidor del sistema. La parte cliente se expondrá en Secciones posteriores.

Tras la instalación de las dependencias del proyecto, se procedió a escribir el código de la API, comenzando por el archivo *index.js*. En él, se crea el servidor Express bajo la denominación ”app”, variable a la que se van añadiendo parámetros como el puerto en el que debe estar escuchando, la IP o el directorio donde se encuentran las vistas o los archivos estáticos. También se definen los archivos que contienen rutas de las que va a constar la API.

En dicho archivo también se importa el módulo *passport* (*controllers/passport.js*), en el cual se definen las funciones para autenticar un usuario que intenta iniciar sesión (*local-login*).

Por otra parte, en la carpeta *models* tenemos todos los modelos de datos que se usan en la aplicación: databases, user y tweet. Cada uno tiene un esquema definido mediante sus correspondientes variables. Mongoose utilizará este esquema para conectarse a cada una de las bases de datos que albergan estos ”objetos” y así poder tratarlos y manejarlos como tal. Mediante la expresión *module.exports* podremos exportar el modelo de este objeto para que pueda ser utilizado en otros archivos Node.

Finalizando con la parte del servidor, tenemos la carpeta *routes* con dos archivos: *routes.js* y *tweets.routes.js*. El primero define las rutas de lo que será la aplicación cliente, esto es, define

las URLs en las que se encontrarán las vistas. Por su parte, *tweet.routes* define los *endpoints* de la API REST del sistema y que se especifican en la Tabla 4.4. Estas rutas van precedidas por la ruta */api/tweets*.

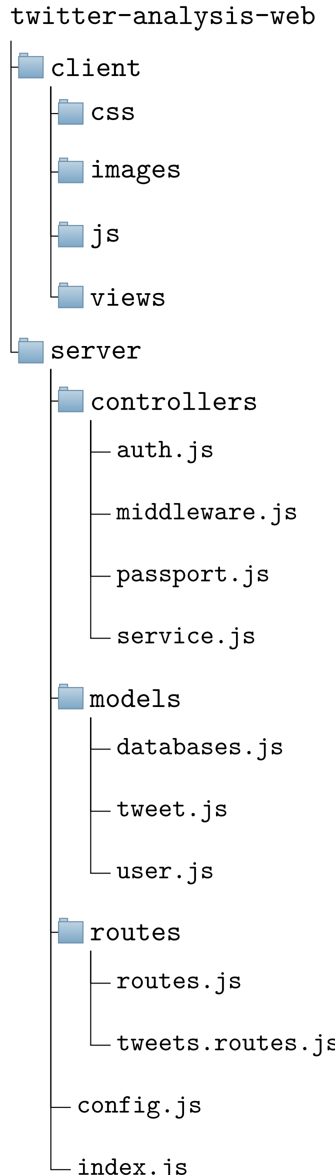

Figura 4.2: Estructura de carpetas de la aplicación Node.

En los siguientes apartados se darán más detalles sobre cada una de las partes de la aplicación anteriormente comentadas.

4.6.0.1. Autorización y autenticación: JWT y Passport

Como se comentó en anteriores secciones, las tecnologías utilizadas para la autorización y la autenticación de la API son JSON Web Token y Passport, respectivamente.

Antes de entrar en detalle, hemos de tener claras las diferencias entre ambos términos. La autenticación se refiere en la identificación de un usuario mediante unas credenciales. Normalmente permiten a los usuarios el acceso a un sistema en el que se tiene información sobre ellos mismos, información que no debería ser accesible a otros usuarios. Por su parte, la autorización es la permisión de acceso a un usuario a una determinada información o funcionalidad. Si su rol o identidad no le permiten el acceso, se le cerrará la puerta a esa determinada parte del sistema.

De esta manera, la autorización nos servirá para registrar a los usuarios en el sistema y asignarles un *token* (JSON Web Token) que les permita acceder a los *endpoints* de la API que permitan el acceso a usuarios no administradores.

Con los datos introducidos para registrarse, los usuarios podrán iniciar sesión, es decir, autenticarse, consultar sus datos (entre los que se encuentra la clave de autorización) e incluso modificarlos.

Con *express-session* manejaremos la sesión de los usuarios que se autentican (en la parte del servidor), mientras que JWT nos permite mantener la sesión a la hora de acceder a los *endpoints* de la API REST, en este caso en la parte del cliente.

Los JWT constan de tres partes separadas por puntos (Figura 4.3):

- **Cabecera o *header*:** JSON con información sobre el tipo de algoritmo de *hashing* utilizado y el tipo de token (en este caso, JWT).
- ***Payload:*** Contiene información sobre el usuario y sus datos.
- **Firma o *signature*:** Cadena resultante de codificar la cabecera y el *payload* junto con una clave (*secret*) que sirve para verificar la validez del *token*.

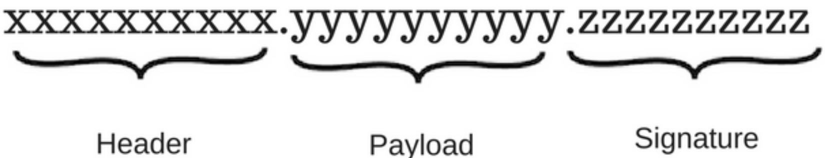

Figura 4.3: Ejemplo de JSON Web Token.

4.6.0.1.1. Añadir JWT a la API En primer lugar, se definirá la clave secreta en el archivo *config.js* y se exportará para utilizarla en la creación de los *tokens*.

```
module.exports = {
    TOKEN_SECRET: process.env.TOKEN_SECRET || "francisgalvez",
};
```

Código 4.25: Archivo *config.js*.

A continuación, se desarrolló el código que generaría un *token* para cada usuario al registrarse: el método *createToken*. Así, el *payload* de los JWT generados por nuestra API contendría el nombre del usuario (email), la fecha en que se registró (en formato UNIX) y el rol que ostenta (*user* para usuarios corrientes o *admin* para usuarios administradores). También podría añadirse un campo para especificar la caducidad del *token*, aunque en este caso se ha optado por JWT que no caduquen, de manera que cada usuario que pida acceso a la API, es decir, que se registre, tenga su propia clave para poder acceder a ella cuando lo necesite.

```
exports.createToken = function(user, role) {
    var payload = {
        sub: user,
        iat: moment().unix(),
        rol: role
    };
    return jwt.sign(payload, config.TOKEN_SECRET);
};
```

Código 4.26: Archivo *service.js*.

El mencionado método de creación de *tokens* será llamado por el método *signup*, definido en el archivo *auth.js*, en caso de que no haya ya ningún usuario registrado con ese email. Así, una vez creados el usuario y el JWT se almacenan ambos en la base de datos MongoDB habilitada para ello.

Por último, el *middleware* se encargará de verificar las peticiones que se reciban. Esto es, será el filtro que permitirá o no el acceso de los usuarios a los *endpoints* a los que estén realizando peticiones.

En subsecciones posteriores se especificará la ruta a través de la cual un usuario podrá registrarse en el sistema.

4.6.0.1.2. Añadir autenticación mediante Passport Dado que el registro se lleva a cabo a través de JWT, Passport se encarga del inicio de sesión de los usuarios en base al email y la contraseña que se almacenaron sobre ellos en dicho proceso.

Así, tenemos el archivo *passport.js* dentro de la carpeta *controllers*. Este archivo exporta una función que se encarga de validar los usuarios previo inicio de sesión. En caso de invalidez, devuelve un mensaje mediante el módulo *connect-flash*, muy útil para mostrar mensajes en aplicaciones Express.

```
const LocalStrategy = require('passport-local').Strategy;

const User = require('../models/user');

module.exports = function (passport) {

    [...]

    passport.use('local-login', new LocalStrategy({
        usernameField: 'email',
        passwordField: 'password',
        passReqToCallback: true
    },

    function (req, username, password, done) {
        User.findOne({'email': username}, function (err, user) {
            if (err) { return done(err); }
            if (!user || !user.validatePassword(password)){
                return done(null, false, req.flash('loginMessage', 'Invalid user or password'));
            }
            return done(null, user);
        });
    }));
}
```

Código 4.27: Archivo *passport.js*.

Nótese cómo ha sido necesario utilizar el módulo *passport-local* para poder definir lo que se denomina una estrategia local (*Local Strategy*) con el fin de establecer los campos ”email” y ”password” como los campos por defecto para la autenticación de usuarios.

```
// Passport session
app.use(session({
    secret: 'whosbest',
    resave: false,
    saveUninitialized: false
}));
app.use(passport.initialize());
app.use(passport.session());
app.use(flash());

[...]

/* ----- Routes ----- */
app.use('/api/tweets', require('./routes/tweets.routes'));
require('./routes/routes')(app, passport);
```

Código 4.28: Fragmento del archivo *index.js*.

Finalmente, sólo nos queda especificar en el archivo *index.js* la necesidad de que nuestra aplicación utilice sesiones e inicializar Passport, tal y como se muestra en el Código 4.28.

4.6.0.2. Modelos

Para la realización de la API ha sido necesario definir tres modelos:

- **Databases:** Dado que se tiene una ruta que devuelve la información sobre las diferentes bases de datos del sistema (colección perteneciente a la base de datos *settings*, mencionada en secciones anteriores), se definió un esquema específico para estos documentos.
- **Tweet:** Es el objeto principal de cuantos modelos tenemos, en el cual se basan los *endpoints* relacionados con el filtrado por franjas horarias.
- **User:** Esquema que permite modelar los usuarios del sistema y que recoge su información principal: email, contraseña, rol y *token*.

Todos estos modelos pueden consultarse más en profundidad en el Anexo C.

Dado que los esquemas de *databases* y *tweets* contienen los mismos campos que se han mostrado en Códigos y Secciones anteriores (ver Sección 4.3.2.3 y Sección 4.2.1), conviene exponer cómo se definió el modelo User.

Mediante el método *connect()* de Mongoose, especificamos la URI en la que se encuentra la base de datos donde almacenar los objetos User. Nótese cómo se importa el archivo *config.js* con el fin de obtener la información sobre el usuario y la contraseña que garantizan el acceso a dicha base de datos.

Acto seguido, se define el esquema del modelo que, como ya anticipábamos, contiene los campos *email*, *password*, *role* y *token*. Todos ellos son requeridos y de tipo String, aunque el rol, de no especificarse, toma por defecto el valor *user*. Es decir, todo usuario que no se especifique como administrador, será registrado como un usuario sin privilegios para acceder a ciertos *endpoints*.

Tanto el email como la contraseña deberán tener entre 5 y 10 caracteres. Además, el rol define su valor en base a un tipo enumerado, que le permite tomar los valores *user* o *admin*.

Especial atención merecen los métodos *pre("save")* y *validatePassword*. El primero se encarga de ejecutar ciertas acciones previas al guardado del usuario en la base de datos. En concreto, sus acciones tienen que ver con la contraseña del usuario. Ya que la vista del perfil del usuario tendrá un formulario de cambio de contraseña, hemos de comprobar si esa operación sobre el usuario es una inserción o una modificación. Esto se hará a través del método *isModified('password')*. Así, si la contraseña especificada no es distinta a la que se tiene de ese usuario que solicita el registro, no hará falta cambiarla y se "escapará" del método. En caso contrario, es decir, si se trata de un registro o de una modificación de contraseña, hará falta encriptarla, labor de la que se encargará el módulo *bcrypt*.

El método de encriptado incluye un número (10), denominado *salt*, que se usará para generar el *hash* asociado a la contraseña y evitará que para dos contraseñas iguales se genere el mismo.

```
const mongoose = require('mongoose');
const bcrypt = require('bcryptjs');
const config = require('../config');

mongoose.connect('mongodb://' + config.MONGO_USER + ':' + config.↩
    ↪ MONGO_PASSWORD + '@' + '<ip>:27021/users', { useNewUrlParser: true });

const userSchema = mongoose.Schema({
    email: {
        type: String,
        required: true,
        minlength: 5,
        maxlength: 50
    },
    password: {
        type: String,
        required: true,
        minlength: 5,
        maxlength: 50
    },
    role: {
        type: String,
        enum: ['admin', 'user'],
        default: 'user'
    },
    token: {
        type: String,
        required: true
    }
});
userSchema.pre("save", function (next) {
    var user = this;

    if (!user.isModified('password')) {
        return next();
    }

    bcrypt.hash(this.password, 10, (err, hash) => {
        this.password = hash;
        next();
    });
});

userSchema.methods.validatePassword = function(password){
    const user = this;
    return bcrypt.compareSync(password, user.password);
};
module.exports = mongoose.model('User', userSchema);
```

Código 4.29: Archivo *user.js*.

4.6.0.3. Rutas

Como vimos en la estructura de carpetas de la Figura 4.2, tenemos dos archivos que definen rutas de nuestra aplicación web (*routes.js*) y de la API REST vinculada (*tweets.routes.js*).

4.6.0.3.1. Aplicación web En *routes.js* definimos las rutas que se corresponden con las diferentes secciones de la aplicación web: la vista de inicio, una sección que versa sobre los objetivos del proyecto ("/about"), otra que sirve como documentación de la API REST ("/apidocs") y, por último, la ventana de inicio de sesión y/o registro ("/auth"). El acceso a esta última sección estará condicionado por el estatus actual del usuario que visita la página. Si está autenticado y su sesión sigue activa, al ir a acceder a la zona de registro se le redirigirá a su perfil, al igual que cuando inicia sesión, que se le redirigirá a dicho *dashboard* (ruta "/dashboard").

```
var ctrlAuth = require('../controllers/auth');
module.exports = (app, passport) => {
    app.get('/auth', isAlreadyLoggedIn, function(req, res){
        res.render('login', {
            message: req.flash('loginMessage')
        });
    });

    app.post('/auth/signup', ctrlAuth.signup);
    app.post('/auth/login', passport.authenticate('local-login', {
        successRedirect: '/dashboard',
        failureRedirect: '/auth',
        failureFlash: true
    }));

    app.get('/dashboard', isLoggedIn, (req, res) => {
        res.render('dashboard', {
            user: req.user
        });
    });

    [...]
};

function isLoggedIn (req, res, next) {
    if (req.isAuthenticated()) { return next(); }
    res.redirect('/');
}

function isAlreadyLoggedIn (req, res, next) {
    if (!req.isAuthenticated()) { return next(); }
    res.redirect('/dashboard');
}
```

Código 4.30: Fragmento del archivo *routes.js*.

Las redirecciones del usuario en función de si está o no autenticado se realizan tal y como se observa en el Código 4.30. Las funciones *isLoggedIn* e *isAlreadyLoggedIn* establecen sendas comprobaciones sobre el estatus del usuario en la página para actuar de la siguiente manera:

- Si el usuario intenta acceder a la ruta */auth* y no está autenticado, se cargará la vista de inicio de sesión/registro. Si lo está, por el contrario, se le redirigirá a su *dashboard*, esto es, a la vista de su perfil.
- Si el usuario intenta acceder a la ruta */dashboard* sin estar autenticado, se le redirigirá a la vista raíz.

Otra de las redirecciones que se realiza es cuando el usuario cierra su sesión en la página ("/logout"). En este caso, también se redirige al usuario a la vista raíz, en la que se encuentra el mapa que visualiza los *tweets*.

Prestemos atención también a la ruta "/auth/login", de tipo POST, a la que se realiza una petición cuando el usuario completa el formulario de inicio de sesión. Mediante el método *authenticate* de Passport, comprobamos que los datos son correctos. En caso afirmativo (*sucessRedirect*), se redirige al usuario a su *dashboard*. De lo contrario, se le redirigirá a la vista origen, */auth*, en la que se mostrará un mensaje de error para informar al usuario de que las credenciales introducidas no son correctas.

4.6.0.3.2. API REST En el archivo *tweet.routes.js* se definen las rutas de la API REST creada tanto para nutrir el mapa de la aplicación web como para satisfacer las peticiones de usuarios y/o aplicaciones externas.

Cabe mencionar, antes de nada, que todas las respuestas de estas rutas se realizan en formato GeoJSON, un formato estándar derivado de JSON para representar elementos geográficos. Es el tipo de datos requerido por Leaflet para importar información sobre elementos geolocalizados.

Los módulos principales importados en este archivo son los que tienen que ver con las conexiones con las bases de datos: *mongoose*, para MongoDB, y *elasticsearch*, para Elasticsearch.

En este sentido, se demuestra la utilidad de la base de datos de configuración y de su colección *databases*, mediante la que abstraemos la información sobre las bases de datos del sistema a todos sus módulos. Así, creamos un diccionario, *Tweets*, en el que las claves serán los nombres de las distintas bases de datos y el valor será el propio cliente para poder realizar operaciones sobre las mismas. Mediante una declaración if/else, haremos una distinción según el motor de base de datos (*engine*) para así crear el cliente correspondiente a cada uno de los dos motores de bases de datos utilizados en esta parte del proyecto.

Nótense las diferencias en cuanto a la creación de las conexiones con Mongoose cuando necesitamos acceder a colecciones concretas. En lugar de exportar el modelo en el correspondiente archivo, tenemos que importar los esquemas en el archivo en que queramos realizar la conexión y utilizar los métodos *createConnection* (en lugar de *connect*) y *model* (especificando en este caso el nombre del modelo, el esquema y el nombre de la colección).

```
const tweetSchema = require('../models/tweet');
const databasesSchema = require('../models/databases');

var settings = mongoose.createConnection('mongodb://' + config.MONGO_USER + ':' + config.MONGO_PASSWORD + '@' + '<ip>:27017/settings', { useNewUrlParser: true });
var databases = settings.model('Databases', mongoose.Schema(databasesSchema.DatabasesSchema), 'databases');

var db;
var Tweets = {};
var dbs = databases.find().lean().exec(function (err, docs) {
    for (var database in docs){
        if(docs[database].engine == "elasticsearch"){
            Tweets[docs[database].name] = new elasticsearch.Client({host: docs[database].URI, log: 'trace'});
        } else if (docs[database].engine == "mongo"){
            db = mongoose.createConnection('mongodb://' + config.MONGO_USER + ':' + config.MONGO_PASSWORD + '@' + docs[database].URI + docs[database].database_name, { useNewUrlParser: true });
            Tweets[docs[database].name] = db.model('Tweet', mongoose.Schema(tweetSchema.TweetSchema), docs[database].collection);
        }
    }
});
```

Código 4.31: Creación de clientes de bases de datos en *tweets.routes.js*.

```
router.get('/databases', middleware.ensureAuthenticated, async (req, res) => {
    if(req.role != 'admin'){
        return res.sendStatus(403);
    }
    response = await databases.find().lean();
    res.jsonp(response);
});
```

Código 4.32: Restricción de ruta a usuarios administradores.

En el Código 4.33, podemos observar dos de los *endpoints*, uno relacionado con cada tipo de base de datos en los que almacenamos *tweets*. En realidad, se trata de la misma consulta (aquella que devuelve los *tweets* según si se tiene o no información geográfica sobre ellos), con la única diferencia de que estamos consultando bases de datos distintas (con sus correspondientes motores), con las diferencias que ello conlleva a nivel de consultas.

Cabe destacar las diferencias de las consultas a Elasticsearch y a MongoDB en términos de complejidad y extensión. Las consultas a Elasticsearch tienen una estructura tipo JSON, en la que ha de establecerse un *body* en el que incluir nuestra *query*. También han de especificarse el *index* y el tipo al que nos referimos. Por su parte, las consultas a MongoDB desde Mongoose

son muy similares a las consultas a las que se harían desde la consola.

```
router.get('/geolocation/:option', middleware.ensureAuthenticated, async (req,
    res) => {
    var response;

    if(req.params.option == "true"){
        response = await Tweets["mainDbES"].search({
            index: 'twitter',
            type: 'tweet',
            body: {
                query: {
                    exists : { "field" : "location" }
                },
                size: req.query.size
            }
        });
    } else if (req.params.option == "false") {
        response = await Tweets["mainDbES"].search({
            index: 'twitter',
            type: 'tweet',
            body: {
                query: {
                    bool: {
                        must_not: { exists: {"field": "location"} }
                    }
                },
                size: req.query.size
            }
        });
    }

    var tweets = response.hits.hits.map(hit => hit._source);
    res.jsonp(geojson.parse(tweets, { Point: 'location' }));
});

router.get('/geolocation/:option/since/:hours', middleware.ensureAuthenticated
    , async (req, res) => {
    var tweets;
    var option = (req.params.option == 'true');

    tweets = await Tweets[req.params.hours].find({location : {$exists: option
        }}).lean();

    res.jsonp(geojson.parse(tweets, { Point: 'location' }));
});
```

Código 4.33: Ejemplo de *endpoints* del archivo *tweets.routes.js*.

Incluyendo como parámetro de las rutas el método del *middleware* que comprueba la veracidad del JWT, exigimos la identificación del usuario que realiza las peticiones mediante la inclusión del *token* en el campo Authorization del cuerpo de las mismas.

Endpoint	**Tipo**	**Descripción**
/all	GET	Devuelve todos los *tweets* almacenados. Restringido a administradores.
/geolocation/:option	GET	Devuelve los *tweets* según el valor de su ubicación (*true*, para *tweets* con ubicación, *false* en caso contrario).
/topics/:list/condition/:op/geolocation/:opt?	GET	*Tweets* que contengan ciertos *topics* (:op = ”and” para búsqueda exclusiva, ”or” para inclusiva).
/geolocation/:option/since/:hours	GET	*Tweets* en base a la existencia de su ubicación filtrados por franja horaria (”2hours”, ”4hours” o ”6hours”.)
/topics/:list/condition/:op/geolocation/:opt/since/:hrs	GET	*Tweets* con presencia de ciertos topics, en base a la existencia de geolocalización y su antigüedad.
/databases	GET	Devuelve la información sobre las bases de datos del sistema. Restringido a usuarios administradores.
/delete/db/:db	POST	Elimina los registros caducados de las bases de datos por franja horaria. Restrigido a usuarios administradores.

Tabla 4.4: *Endpoints* de la API REST.

Todos los *endpoints* están protegidos y necesitan autorización para poder ser accedidos. Además, ciertas rutas están restringidas a usuarios administradores por el peligro que supondría que fuesen accesibles para cualquier usuario. Se trata de las rutas */all*, que devuelve todos los *tweets*

almacenados durante el ciclo de vida del sistema, */databases*, que devuelve la información sobre las bases de datos del sistema y */delete/db/:db*, que ejecuta el borrado de los registros caducados sobre las bases de datos de tiempo.

La restricción a usuarios administradores se realiza como se muestra en el Código 4.32, enviando un estado HTTP 403 (Forbidden) a usuarios no autorizados.

Todas las peticiones a las diferentes rutas de esta API REST se realizarán de manera asíncrona, aprovechando así las capacidades que ofrece Node.js para ejecutar procesos en paralelo.

Finalmente, se incluye una tabla resumen de todos los *endpoints* de la API REST, su tipo y una pequeña descripción sobre ellos (Tabla 4.4).

4.6.0.3.3. GeoJSON GeoJSON (Internet Entineering Task Force, *RFC 7946 - The GeoJSON Format*) es un estándar de codificación de estructuras de datos geográficos que se usará para la mayoría de *endpoints* de nuestra API REST.

```
{
    "type": "FeatureCollection",
    "features": [
    {
        "type": "Feature",
        "geometry": {
            "type": "Point",
            "coordinates": [-83.060184,42.347021]
        },
        "properties": {
            "_id": "5ce1baca5e9c224a9508b94c",
            "id": "1130206829089296385",
            "topics": ["MongoDB"],
            "text": "RT @codewallblog: NoSQL: Creating Collections using MongoDB
                [...]",
            "source": "Otros",
            "user_name": "bobbidigi0",
            "sensitive": false,
            "lang": "en",
            "timestamp": "1558297280939",
            "date": "2019-05-19 20:21:20"
        }
    },
    [...]
}
```

Código 4.34: Ejemplo de respuesta GeoJSON.

Se basa en una estructura con tres campos raíz:

- **type**, relacionado con el número de objetos geográficos que se incluyen. Puede tomar los valores *Feature*, para un único elemento, o *FeatureCollection*, para colecciones de elementos.
- **geometry**, que incluye los objetos geométricos de los que se provee información. Puede tomar los valores *Point*, *LineString*, *Polygon*, *MultiPoint*, *MultiLineString* y *MultiPolygon*.
- **properties:** Cualquier dato sobre el objeto geográfico. En formato clave-valor.

Así, cada respuesta de la API REST será una *FeatureCollection*, esto es, una colección de *tweets* cuyo elemento central son las coordenadas (*Point*) que lo relacionan con la ubicación desde la que se publicó. Las propiedades de cada *Feature* serán el resto de datos que almacenamos sobre los estados de Twitter: id, texto, fecha, idioma, etc.

Un ejemplo de respuesta GeoJSON de nuestra API REST se incluye en el Código 4.34.

4.6.1. Cliente web

La API Node tendrá una parte cliente cuyo componente principal será el mapa en el que se visualizarán los *tweets* almacenados en las bases de datos por franja horaria. Además, contará con diferentes secciones relacionadas con la información del proyecto, la documentación de la API REST y una zona de inicio de sesión y registro, que permitirá a los usuarios acceder a su perfil y consultar su *token* de acceso a la API REST.

Las diferentes vistas se han construido mediante HTML en conjunción con Bootstrap, que nos ha facilitado la tarea de diseño de las mismas. Además, mediante CSS se han especificado reglas concretas para esta aplicación.

EJS ha permitido crear plantillas para ciertas partes HTML comunes a todas las vistas, evitando así la duplicación de código. De hecho, en la carpeta *views* se incluye la carpeta *partials* en la que tenemos los módulos relativos a la cabecera, el pie de página y las etiquetas comunes del *<head>* de los archivos HTML. En este sentido, podemos ver cómo se importan estas plantillas en cada vista a través del ejemplo en el Código 4.35.

Véase cómo, aparte del título de la página y del cuerpo de la misma (distintos en cada caso), el resto del código es reutilizado gracias a EJS. Mediante su opción *include*, previa indicación de que se trata de una orden EJS (a partir de los elementos de inicio y cierre, "<%" y "%>"), podremos importar cuantos módulos tengamos predefinidos como plantillas de este tipo. Eso sí, el tipo de los archivos HTML y de las plantillas será ".ejs". Además, en el archivo *index.js*, donde especificábamos toda la configuración de la aplicación web, tendremos que establecer EJS como motor de plantillas (*view engine*).

En cada vista se añade un pequeño *script* que, mediante Jquery, añade la clase Bootstrap "active" al elemento de lista de la barra de navegación que se corresponde con la vista actual. Esto hace que el color del título de esta sección se resalte sobre el resto, haciéndole ver al usuario en qué sección se encuentra.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Who is best?</title>
  [...]
  <% include partials/head %>
</head>
<body>
  <% include partials/navbar %>
  <div id="map"></div>
  <script src="js/map.loader.min.js"></script>
  <script>
    $(function() {
      $('#home').addClass('active');
    });
  </script>
</body>
</html>
```

Código 4.35: Código HTML de la vista principal.

4.6.1.1. Construcción del mapa Leaflet

La vista inicial que se mostrará en la ruta padre de la API es el mapa de visualización de *tweets*. Este mapa se ha construido con Leaflet.js (Leaflet, *Leaflet Documentation*), una librería JavaScript especializada en este campo.

Una de las principales virtudes de Leaflet es la gran comunidad de desarrolladores que contribuyen a su crecimiento con la construcción de *plugins* que aumentan las funcionalidades de sus mapas. En este sentido, han sido varios los *plugins open-source* que se han utilizado en nuestro mapa:

- **Awesome Markers:** Marcadores de múltiples colores que nos permitirán distinguir los *topics* de cada *tweet*.
- **Leaflet Fullscreen:** Icono situado sobre el mapa que permite la vista del mismo a pantalla completa.
- **Leaflet Search:** Control para buscar direcciones concretas sobre el mapa.
- **Locate Control:** Permite localizar la ubicación del usuario y sitúa la vista sobre ésta.
- **Marker Cluster:** Este *plugin* permite crear *clusters*, de manera que la vista del mapa quede menos saturada de marcadores y, de un simple vistazo, se puedan ver las zonas con mayor y/o menor número de marcadores.
- **Timeline Slider:** Filtro que permite al usuario determinar los *tweets* que se muestran: los de las últimas 6, 4 o 2 horas.

```
var map = L.map('map', {
    maxZoom: 16, layers: [oracle, mysql, sqlserver, postgres, mongo, ibm, ↩
        ↪ access, redis, elasticsearch, sqlite]
}).setView([20.0,0.0], 2);

var lightMap = L.tileLayer('https://{s}.basemaps.cartocdn.com/light_all/{z}/{x↩
    ↪ }/{y}.png', {
    attribution: '&copy; <a href="https://www.openstreetmap.org/copyright">↩
        ↪ OpenStreetMap</a> & &copy; <a href="https://carto.com/attributions">↩
        ↪ CARTO</a>',
    maxZoom: 16,
    minZoom: 2.5,
    continuousWorld: false,
}).addTo(map);
```

Código 4.36: Creación de mapa Leaflet

Como vemos en el Código 4.35, para incluir un mapa Leaflet en un archivo HTML, bastará con definir un <div> con un id concreto que se usará en la programación del mapa, el cual se define mediante un archivo JavaScript. Además, en la cabecera de la página se incluyen todos los archivos CSS y todos los *scripts* relativos a los diferentes *plugins* utilizados. En este sentido, se ha intentado reducir el tiempo de carga de la página teniendo el máximo número posible de estos archivos descargados en el propio servidor. También se han ”minificado” todos los archivos JavaScript y CSS con este mismo fin.

Entre los archivos JavaScript de nuestra aplicación se encuentra el que carga el mapa Leaflet en la página. Así, las primeras líneas de este código se definen las capas del mapa. Cada capa se corresponderá con uno de los *topics* de los que versan los *tweets* almacenados y se nombrará en relación a los mismos.

A continuación, se crea el mapa mediante la función *L.map()*, que recibe como parámetros el id del <div> que contendrá el mapa y una lista de parámetros de configuración. En concreto, en este caso se especifica el zoom máximo que se podrá realizar sobre el mapa y las capas previamente definidas.

Todavía es necesario añadir un mapa base para poder visualizar el mapa. Se ha escogido la versión *light* del mapa gratuito de Carto (Carto, *Basemaps Data Services*). Aparte de las referencias al desarrollador del mapa, se incluyen el zoom máximo y el mínimo y se establece la variable *continuousWorld* a *false*. Esto hará que el mapa del mundo no se repita a izquierda y a derecha, sino que sea finito.

Otro de los aspectos destacados en la construcción del mapa es la forma en la que se filtran los *tweets* por su antigüedad, gracias al *plugin* Timeline Slider. A través de la función getDataAddMarkers, la cual es llamada cada vez que el usuario selecciona un filtro de tiempo distinto. Así, comprobando el valor del *label* seleccionado, se identifica el filtro y se realiza la correspondiente llamada a la API REST. En concreto, queremos que devuelva los *tweets* geolocalizados de la

base de datos con la antigüedad especificada, por lo que accederemos a la ruta */geolocation/-true/since/:hours*.

```
getDataAddMarkers = function ({label, value, map, exclamation}){
    for (layer in enginesOverlay) {
        enginesOverlay[layer].clearLayers();
    }

    var hours;

    if(label == "2 hours"){
        hours = "2hours";
    } else if (label == "4 hours"){
        hours = "4hours";
    } else if (label == "6 hours"){
        hours = "6hours";
    }

    tweets = $.ajax({
        url: "https://whosbest-twitter-map.app.di.ual.es/api/tweets/geolocation/↩
            ↪ true/since/" + hours,
        headers: { 'Authorization': 'Bearer token'},
        dataType: "json",
        error: function(xhr) {
            console.log(xhr.statusText);
        }
    });

    $.when(tweets).done(function() {
        var geojson = L.geoJson(tweets.responseJSON, {
            onEachFeature: function (feature, layer) {
            if(jQuery.inArray("Oracle", feature.properties.topics) !== -1){
                layer.setIcon(L.AwesomeMarkers.icon({icon: 'null', markerColor: '↩
                    ↪ red'}));
                layer.addTo(oracle);
            } else if [...]

            layer.bindPopup("<blockquote class=twitter-tweet data-cards=hidden data↩
                ↪ -conversation=none data-lang=es" + "><p lang=" + feature.↩
                ↪ properties.lang + "dir=ltr>" + feature.properties.text + "</p>&↩
                ↪ mdash;" + feature.properties.user_name + "<a href=https://↩
                ↪ twitter.com/" + feature.properties.user_name + "/status/" + ↩
                ↪ feature.properties.id + ">" + feature.properties.date + "</a></↩
                ↪ blockquote>", {minWidth: 215, autoClose: true, closeOnClick: ↩
                ↪ true}).on('click', clickZoom);
        });
    });
}
L.control.layers(null, enginesOverlay, {collapsed: false}).addTo(map);
```

Código 4.37: Asignación de marcadores a capas y filtros.

La mencionada petición a la API REST se realiza a través del método Ajax.

El método *when()* de Jquery nos permite crear una promesa (*promise*) que espera a que la petición *tweets* se haya completado con éxito (*done()*). En ese caso, se ejecuta una función que carga las diferentes *Features* del GeoJSON en los correspondientes objetos Leaflet. Habrá que comprobar la pertenencia de cada *feature* a cada capa (*topic*) mediante varias declaraciones if/else. Cuando una instancia pertenezca a una capa, se le añadirá su marcador (cada capa tiene uno asociado, con un color distintivo) y se le asignará a la capa o capas que le correspondan (recordemos que un *tweet* puede tener varios *topics*, es decir, comentar varios motores de bases de datos a la vez).

Además de tener un marcador situado en cada par de coordenadas, cada marcador tendrá asociado un *popup*, de manera que, cuando el usuario haga clic sobre uno de ellos, un cuadro de texto emerja sobre él y muestre el contenido del *tweet*. Asociaremos dicho *popup* con su marcador aplicando el método *bindPopup()* sobre cada marcador. Asimismo, se especificarán algunos parámetros sobre estas ventanas: no podrán tener menos de 215 píxeles de ancho y se cerrarán automáticamente cuando el usuario pulse en cualquier otra parte del mapa.

Como vemos en el Código 4.37, al *popup* se le asocia directamente el código HTML que albergará. En concreto, se hace uso del widget oficial de Twitter, diseñado precisamente para integrar *tweets* en sitios web. Twitter provee a sus usuarios de una URL que contiene el código JavaScript del *widget*. Así, con el método *on("click")*, se determina la ejecución de la función *clickZoom*, asociada con la ejecución de dicho *script*, cada vez que el usuario pulse sobre un marcador.

```
function clickZoom(e) {
    $.getScript("https://platform.twitter.com/widgets.js");
    $(".leaflet-popup").hide();
    setTimeout(function() {
        $(".leaflet-popup").show();
    }, 1050);
}

map.on('popupopen', function(e) {
    var px = map.project(e.popup._latlng);
    px.y -= e.popup._container.clientHeight/2;
    map.panTo(map.unproject(px), {animate: true});
});
```

Código 4.38: Carga del *widget* de Twitter.

Con el método *getScript* de Jquery podremos obtener el archivo JavaScript. Dado que la carga del *script* y, en consecuencia, del *widget* requiere de unos pocos segundos, se esconderá el *popup* que el usuario desea desplegar. Esto sólo durará apenas un segundo y, a continuación, se mostrará el contenido del *tweet* con una interfaz idéntica a la de sus clientes web y móvil. Este retraso será casi imperceptible para el usuario, máxime cuando al pulsar sobre un mar-

cador (*map.on('popupopen')*) ocurren una serie de "movimientos" en la vista del mapa. Estos movimientos tienen que ver con el ajuste de la vista para que el marcador elegido esté centrado en la pantalla.

4.6.2. Despliegue en Openshift

Teniendo ya la parte cliente y la parte del servidor listas, se optó por desplegar la aplicación en Openshift-DI, que nos proporciona una plataforma para el despliegue y administración de aplicaciones en Kubernetes. Es lo que se conoce como una Plataforma como Servicio (PaaS).

Para el despliegue de nuestra aplicación en Openshift se han usado las plantillas predefinidas para aplicaciones, las cuales se muestran en la Figura 4.4. En el *Service Catalog* de Openshift tenemos una vista general de todas las plantillas, entre las cuales se escogió la relativa a Node.js.

Para crear una aplicación Node, bastará con especificar un nombre y el repositorio (público) en el que se encuentra el código de la misma. El único requisito que se exige es que en la carpeta raíz del proyecto se encuentren los archivos *package-lock.json* y *package.json*, con el fin de que todas las dependencias del proyecto puedan ser identificadas e instaladas.

En el momento en que apliquemos la configuración de la aplicación, inmediatamente comenzará la construcción (*build*) y, a continuación, el despliegue (*deployment*). Cada uno de ellos será etiquetado con un número a modo de identificador.

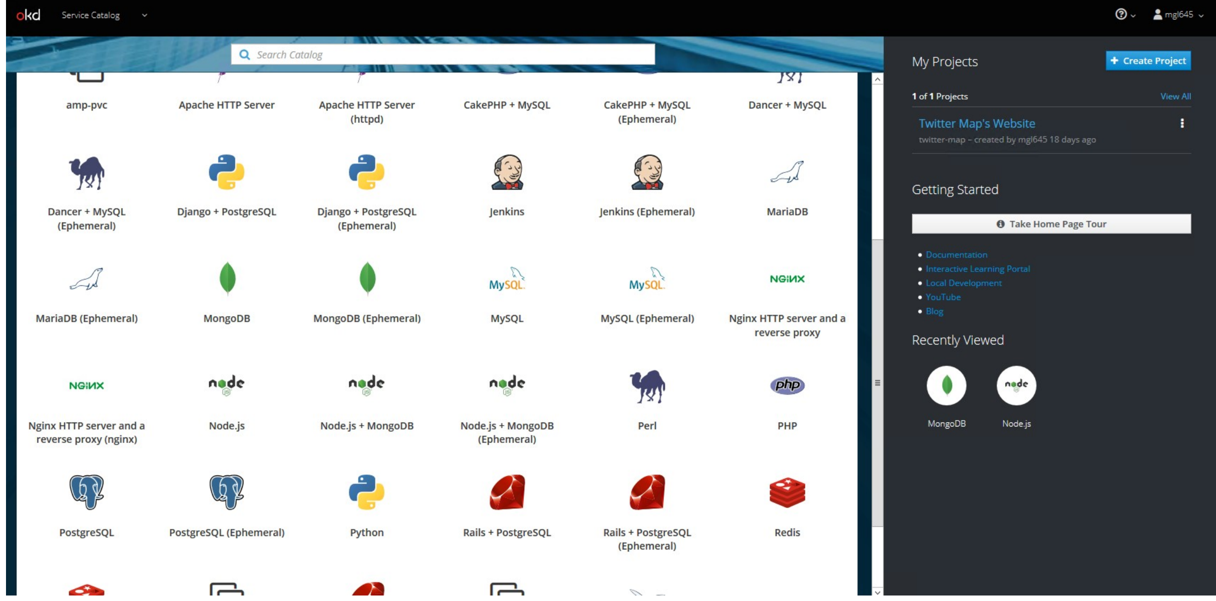

Figura 4.4: Vista del Service Catalog de Openshift.

Como vemos en la Figura 4.5, correspondiente a la vista de proyecto (en concreto, a la sección de vista general, *Overview*), la aplicación se encontrará escuchando las peticiones TCP en el

puerto 8080 del *cluster* de Kubernetes. El tráfico externo se procesa a través de la URL **https://whosbest-twitter-map.app.di.ual.es**, protegida mediante certificado SSL, que reúne tanto el nombre que se le dio al proyecto como el de la *app*. Esta ruta será accesible únicamente desde la red de la UAL.

En el resumen informativo de la aplicación *whosbest* observamos, en la zona derecha, un elemento circular con un número en su interior y dos flechas a su derecha, una en sentido ascendente y otra en sentido descendente. Este conjunto hace referencia al número de *pods* de la aplicación. Mediante las mencionadas flechas podremos escalar el tamaño del *cluster* de contenedores en el que se encuentra desplegado este servicio. Como vemos en la mencionada Figura, en el momento de la captura el número de *pods* se encuentra escalándose hasta 5.

Durante el proceso de escalado del *cluster*, no se detiene en ningún momento la ejecución de la aplicación, que seguirá funcionando sin interrupción. En este sentido, la aplicación mantendrá siempre el número especificado de *pods*, de manera que, si alguno fallase, rápidamente Kubernetes desplegaría otro en su lugar.

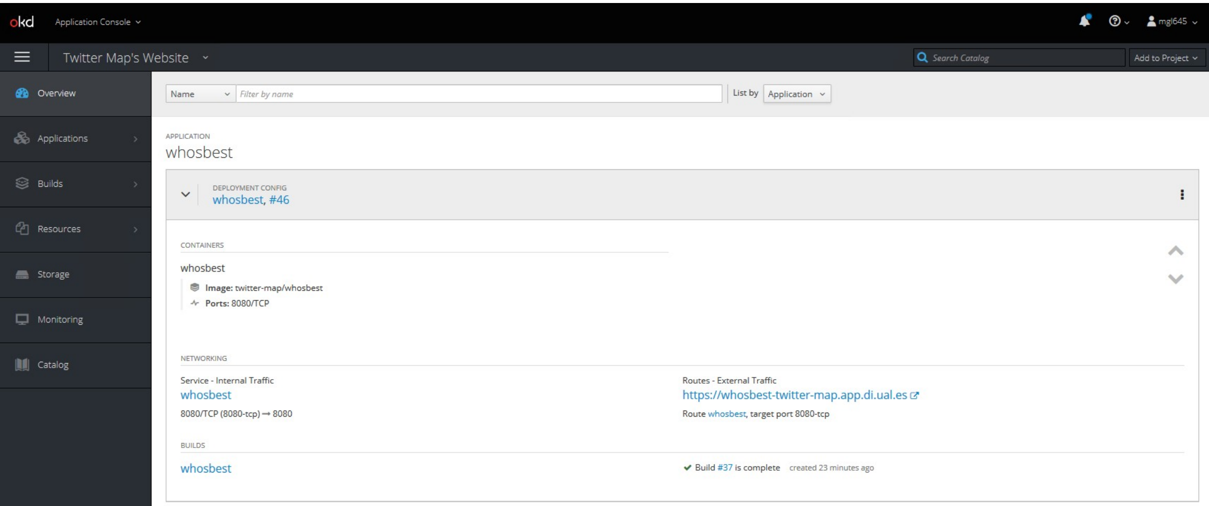

Figura 4.5: Vista del proyecto Openshift.

Además de la sección Overview, se tienen las siguientes secciones:

- **Applications:** Nos da información sobre los *deployments* llevados a cabo, los *pods* activos e inactivos, los servicios o las rutas de la aplicación o aplicaciones que formen parte del proyecto. En concreto, resulta muy útil el acceso al detalle de los *pods* y el poder consultar sus *logs*, con el fin de poder resolver cualquier fallo en la construcción y/o despliegue de los mismos.

- **Builds:** En esta sección tendremos acceso a toda la información relativa a la construcción de las aplicaciones.

- **Resources:** Relacionada con los recursos (memoria, CPU, etc.) y los usuarios del proyecto.

- **Storage:** Información sobre los volúmenes persistentes del proyecto, no utilizados en este caso.
- **Monitoring:** Provee una vista general del estado del *cluster*, de modo similar a la interfaz de usuario de Kubernetes. Da información sobre los *pods* activos y los últimos *builds* y *deployments*.
- **Catalog:** Vista del catálogo de plantillas para añadir al proyecto.

4.7. Vista previa y manejo de la aplicación final

En esta última sección del capítulo se mostrará la apariencia y funcionalidades de la aplicación final, ya desplegada en Openshift-DI y lista para ser usada.

Como veíamos en el apartado anterior, Openshift definió una ruta para el tráfico externo. Se trata de la URL **https://whosbest-twitter-map.app.di.ual.es/**. Si accedemos a la misma, veremos la sección principal, esto es, el mapa en el que se sitúan los *tweets* según la ubicación desde la que se publicaron.

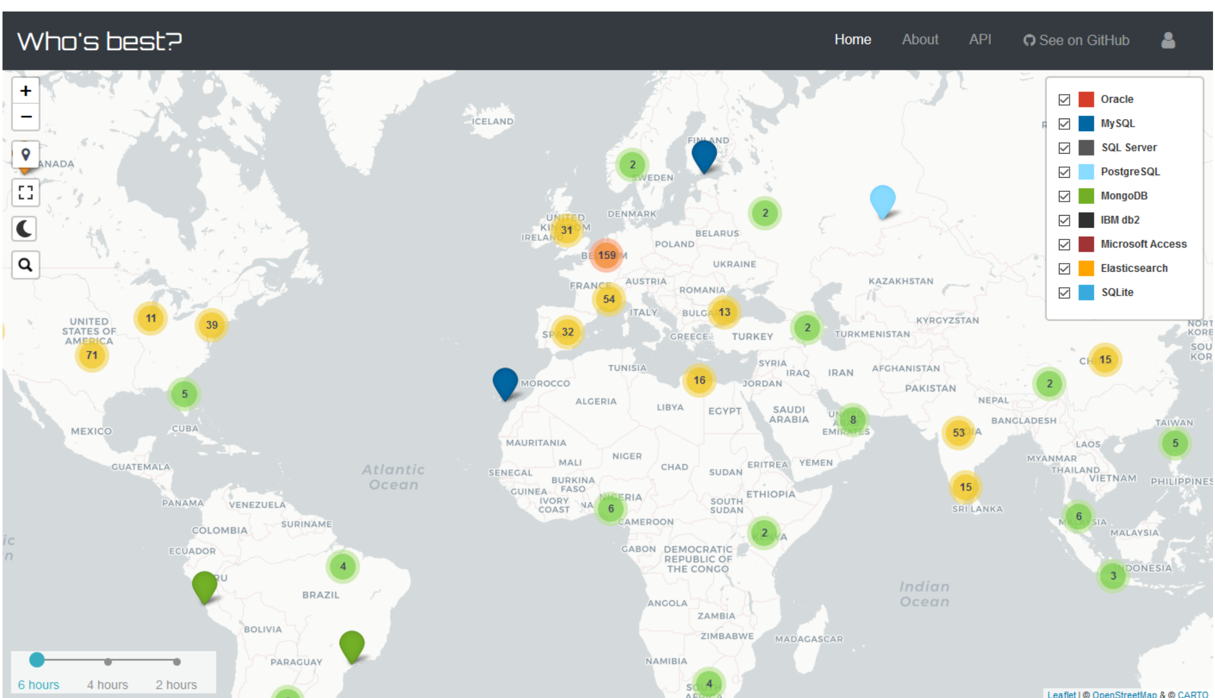

Figura 4.6: Vista inicial de la aplicación web.

En la Figura 4.6 vemos los *clusters* de marcadores, creados gracias al *plugin* Marker Cluster que comentábamos en la Sección 4.6.1.1. Así, las zonas con más afluencia de *tweets* se presentan en colores más rojizos, mientras que los más ”vacíos” tienen color verde. Incluso, algunos marcadores aparecen por sí mismos, señal de que no hay más marcadores en un radio cercano. En

cada *cluster*, además, vemos el número exacto de marcadores que engloba.

A medida que vamos haciendo zoom sobre el mapa, los *clusters* se van desagregando y reconfigurando. Si hacemos clic sobre uno de ellos, nos acercará al seleccionado y, en caso de que se encuentren en la misma zona (o en un radio demasiado pequeño), se desplegarán los marcadores de los que consta.

El usuario puede personalizar la vista mediante el filtro de capas situado en la zona superior derecha del mapa. En este filtro se muestran los diferentes temas por los que se han filtrado los *tweets*. Junto a ellas se muestra el color de los marcadores que las representan. Inicialmente están todas habilitadas por defecto, pero el usuario puede activarlas o desactivarlas a su gusto. Para ello tendrá que presionar el *checkbox* situado a la izquierda de cada capa.

Al hacer clic sobre un marcador, se abrirá su *popup* asociado y nos mostrará la información del *tweet*, tal y como se observa en la Figura 4.7.

En la zona superior derecha del mapa vemos los controles relativos al zoom, la localización del usuario (Locate Control), pantalla completa (Leaflet Fullscreen), modo noche (un control personalizado) y búsqueda de localizaciones (Leaflet Search). Se pueden encontrar capturas sobre sus funcionalidades en el Anexo D.

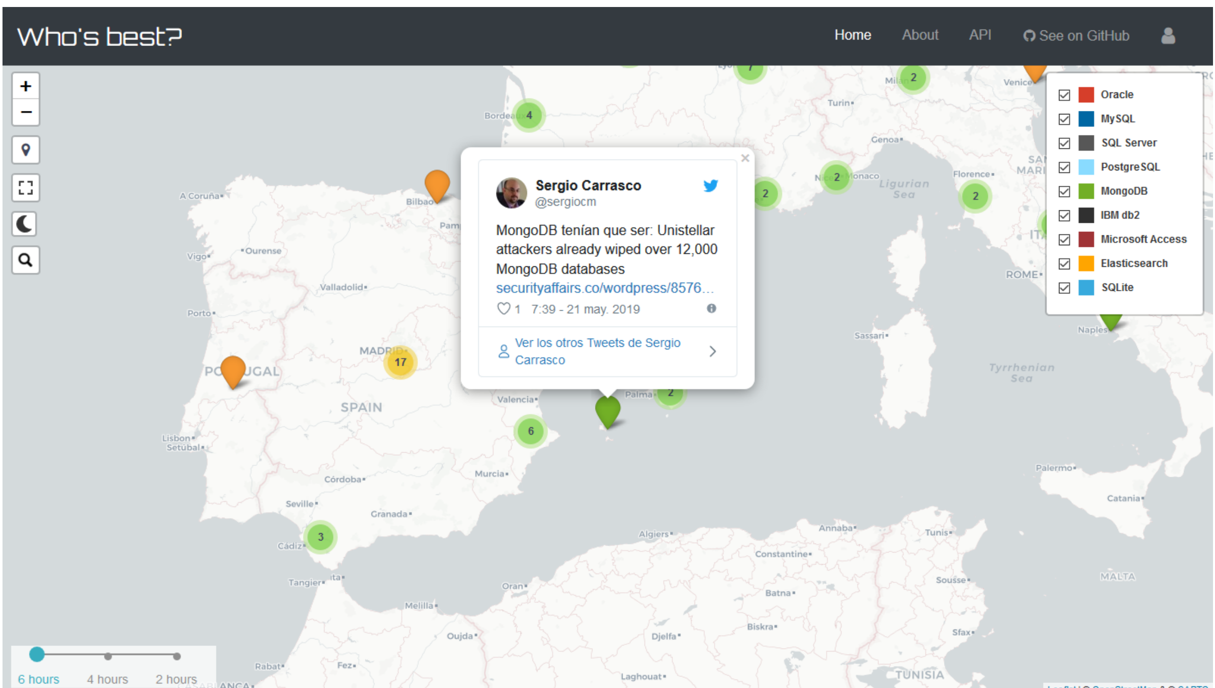

Figura 4.7: Vista detalle de un *tweet*.

Finalmente, en la zona inferior izquierda tenemos el filtro de tiempo (Timeline Slider), incialmente establecido en 6 horas. El usuario podrá, del mismo modo, ir variando la antigüedad de

los *tweets* que desea ver en el mapa.

Como se anticipaba en apartados anteriores, tenemos además otras vistas en la aplicación relativas a la información del proyecto, la documentación de la API REST y el inicio de sesión y registro. También se tiene la vista *dashboard*, en la que el usuario registrado podrá ver detalles sobre su perfil. Las capturas de funcionamiento de estas vistas se exponen en el Anexo D.

Puede observarse en las mencionadas imágenes que el lenguaje utilizado en la aplicación es el inglés. Esto le dará el carácter global con el que ya cuenta a través de el mapa del mundo que constituye su figura central. Además, permitirá que más usuarios puedan comprender su funcionamiento y su objetivo y, por tanto, utilizarla y en algunos casos adaptarla a sus necesidades, ya que el código es público y accesible para cualquier desarrollador.

4.8. Resumen del capítulo

Este Capítulo 4 es el capítulo principal de la memoria, en el que se han expuesto todas y cada una de las tareas que han llevado a cumplir los objetivos mencionados en el Capítulo 1.

Por un lado, se ha expuesto toda la parte de obtención, procesamiento y almacenado de *tweets* a partir de la Standard Streaming API de Twitter, Kafka, Spark Streaming y los diferentes motores de bases de datos del sistema. Así, se tiene el *index* ”twitter” en Elasticsearch, para el que se ha definido un *mapping*, centrado en los campos *text*, *location* y *date*. Dicho índice estará en una máquina Ubuntu dedicada en exclusiva a Elasticsearch y Kibana, que será la herramienta de análisis que se utilizará para el Complemento del Trabajo Fin de Grado.

El resto de bases de datos (MongoDB y Redis) se han desplegado en otra instancia como aplicación Docker Compose. Dicha instancia, además, tendrá un *cron job* que se ejecutará cada minuto de cada hora y cada día y que consistirá en una petición POST a la API para eliminar los registros caducados de las bases de datos de tiempo (Mongo).

Se han detallado las construcciones de los *clusters* Spark y Kafka y se han explicado detenidamente los *scripts* productor y consumidor.

Igualmente, se dedicó una sección al desarrollo de la API, la cual se basó en la implementación de la autorización y autenticación de usuarios, la definición de modelos y rutas y la creación de la aplicación web. Del mismo modo, se comentó el despliegue de la misma en Openshift.

Finalmente, se incluyó una sección para explicar el uso y apariencia de las diferentes vistas de la aplicación.

5. Conclusiones y trabajo futuro

En este último capítulo se comentarán los resultados del trabajo realizado durante este Trabajo Fin de Grado y se propondrán futuras líneas de desarrollo del mismo.

5.1. Conclusiones

Este Trabajo Fin de Grado tenía como objetivo principal la construcción de una infraestructura Big Data para el procesado de *tweets* en tiempo real y, como parte visible, una interfaz en forma de mapa en la que poder consultarlos según su ubicación. Así, se ha encontrado en Spark Streaming y en Kafka dos tecnologías que, de manera fiable, distribuida y con una gran tolerancia a fallos, han cooperado para lograr el objetivo.

También ha de ponerse en valor la multitud y variedad de tecnologías que han asistido a los objetivos de este TFG y que lo han convertido en un proyecto multidisciplinar y transversal. En concreto, se ha aprovechado este proyecto para probar distintas herramientas y tecnologías de actualidad relacionadas con la búsqueda, tratamiento y almacenamiento de datos en el contexto de Big Data.

Por otro lado, la gran repercusión que ostentan las redes sociales, y en concreto Twitter, en la actualidad ha hecho que el sistema construido sea de interés y constituya una fuente fiable sobre la popularidad de ciertos temas en cada momento. En este sentido, cabe recordar que, gracias a la abstracción de la configuración del sistema y, en concreto, de las palabras clave por las que se filtran los estados de Twitter, resultaría muy sencillo cambiar el contexto del sistema y realizar el seguimiento de cualquier otro tema.

5.2. Trabajo futuro

Tras la finalización de este Trabajo Fin de Grado, el proyecto desarrollado podría mejorar o avanzar su recorrido en varias direcciones.

En primer lugar, y tal y como se justificó en la Sección 1.1, debería desarrollarse algún método para mejorar el filtrado de los *tweets*. Redis tuvo que retirarse del conjunto de *topics* dada su coincidencia con un verbo del francés y, en muchos otros conjuntos de temas relacionados (véase lenguajes de programación, marcas, etc.), muchos de los nombres coinciden con verbos, adjetivos o sustantivos de diferentes idiomas. De esta manera, se hace muy difícil obtener resultados fiables para según qué filtros, y aunque la API de Twitter permite especificar ciertos grados de rigurosidad, no resultan suficientes para resolver este problema.

Así, se propone como posible línea de trabajo futuro el desarrollo de un algoritmo que, mediante ***machine learning***, sea capaz de interpretar el significado de la palabra clave por la que se filtran los *tweets*. Nótese que este análisis debe ser posterior a la obtención del estado de Twitter y debería ejecutarse, por tanto, en el *script* Spark. En este sentido, resultaría interesante el uso de la librería Spark MLlib para la construcción del algoritmo.

Otra mejora del presente Trabajo tendría que ver con la cantidad de *tweets* procesados. Es evidente que el uso de la API Standard de Twitter no nos devuelve la totalidad de estados que se generan en todo el mundo, por lo que para implementaciones cuya finalidad sea obtener datos mucho más precisos **sería recomendable el uso de la PowerTrack API**.

Asimismo, se ha detectado pérdida de información relativa a la ubicación de los usuarios cuya causa es la precisión de la API que se utiliza para convertir las direcciones en pares de coordenadas. Se ha observado que, en ocasiones, confunde localidades con el mismo nombre pero de distintas regiones e incluso da como respuesta las coordenadas [0, 0] a literales generalistas del tipo ”Global” o ”Mundo”. Sería conveniente, por tanto, un **cambio a una API de mayor precisión**, como la de Google Maps, aunque este cambio supondría una inversión económica, ya que la mayoría de servicios gratuitos de este tipo tienen ciertas limitaciones en cuanto al número de peticiones diarias.

En cuanto a la definición de los temas o palabras clave por las que filtrar los *tweets*, aunque el sistema se ha intentado hacer lo más encapsulado posible, es cierto que el administrador sigue necesitando modificar código para cambiar dichos parámetros. Por ello, se propone la **construcción de una interfaz de usuario que permita añadir y/o eliminar temas** que actualmente están siendo objeto de filtrado de *tweets*.

Finalmente, se propone utilizar todos los datos almacenados desde el inicio del ciclo de vida del sistema con **fines analíticos**, para así poder obtener conclusiones a partir de los mismos y poder cotejar la popularidad de los motores de bases de datos analizados en Twitter. Esta será la línea que se continuará mediante el Complemento de este Trabajo Fin de Grado.

Bibliografía

[1] Carto. *Basemaps Data Services*. Disponible en: https://carto.com/location-data-services/basemaps/.

[2] B. Chambers y M. Zaharia. *Spark: The Definitive Guide*. O'Reilly Media, Inc., 2018. ISBN: 9781491912218.

[3] Nova Ciencia. *CloudDi: bienvenidos a la nube de la UAL*. Febrero de 2018. Disponible en: https://novaciencia.es/clouddi-nube-ual/.

[4] T. Das, M. Zaharia y P. Wendell. *Diving into Apache Spark Streaming's Execution Model*. Julio de 2015. Disponible en: https://databricks.com/blog/2015/07/30/diving-into-apache-spark-streamings-execution-model.html.

[5] Data Science Toolkit. *Data Science Toolkit Documentation*. Disponible en: http://www.datasciencetoolkit.org/developerdocs.

[6] Docker. *Docker Compose*. Disponible en: https://docs.docker.com/compose/.

[7] Docker. *Docker Documentation*. Disponible en: https://docs.docker.com/.

[8] Domo. *Data Never Sleeps 6.0*. 2018. Disponible en: https://www.domo.com/learn/data-never-sleeps-6#/.

[9] Elastic. *Elastic Stack and Product Documentation*. Disponible en: https://www.elastic.co/guide/index.html.

[10] DB-Engines. *DB-Engines Ranking: March 2019*. Marzo de 2019. Disponible en: https://db-engines.com/en/ranking (visitado 26 de marzo de 2019).

[11] Express.js. *Referencia de API*. Disponible en: http://expressjs.com/es/api.html.

[12] Internet Entineering Task Force. *RFC 7946 - The GeoJSON Format*. Disponible en: https://tools.ietf.org/html/rfc7946.

[13] D. Kakadia. *Apache Mesos Essentials: Build and execute robust and scalable applications using Apache Mesos*. Packt Publising, 2015. ISBN: 9781783288779.

[14] Kubernetes. *Documentación de Kubernetes*. Disponible en: https://kubernetes.io/es/docs/.

[15] LaTeX. *LaTeX Documentation*. Disponible en: https://www.latex-project.org/help/documentation.

[16] Leaflet. *Leaflet Documentation*. Disponible en: https://leafletjs.com/reference-1.4.0.html.

[17] G. Maas y F. Garillot. *Stream Processing with Apache Spark*. O'Reilly Media, Inc., 2019. ISBN: 9781491944240.

[18] W. McKinney. *Python for Data Analysis*. 2nd Edition. O'Reilly Media, Inc., 2017. ISBN: 9781491957660.

[19] MongoDB. *MongoDB Documentation*. Disponible en: https://docs.mongodb.com/.

[20] A. Murthy. *Apache Hadoop YARN: Moving beyond MapReduce and batch processing with Apache Hadoop 2*. Addisson-Wesley Professional, 2014. ISBN: 9780133441925.

[21] Node.js. *Node.js Documentation*. Disponible en: https://nodejs.org/es/docs/.

[22] Openshift. *Openshift Container Platform*. Disponible en: https://www.openshift.com/products/container-platform/.

[23] Openstack. *What is Openstack?* Disponible en: https://www.openstack.org/software/.

[24] PyMongo. *PyMongo Documentation*. Disponible en: https://api.mongodb.com/python/current/.

[25] Redmine. *Redmine Documentation*. Disponible en: https://www.redmine.org/.

[26] G. Saphira, T. Palino y N. Narkhede. *Kafka: The Definitive Guide*. O'Reilly Media, Inc., 2017. ISBN: 9781491936160.

[27] K. Schwaber y J. Sutherland. *The Scrum Guide*. Noviembre de 2017. Disponible en: https://www.scrum.org/resources/scrum-guide.

[28] Stack Overflow. *Developer Survey Results*. 2018. Disponible en: https://insights.stackoverflow.com/survey/2018#technology.

[29] Tweepy. *Tweepy Documentation*. Disponible en: https://tweepy.readthedocs.io/en/latest/.

[30] Twitter. *Docs - Twitter Developers*. Disponible en: https://developer.twitter.com/en/docs.html.

[31] Twitter. *Filter realtime Tweets: POST statuses/filter*. Disponible en: https://developer.twitter.com/en/docs/tweets/filter-realtime/api-reference/post-statuses-filter.html.

[32] Twitter. *Filter realtime Tweets: PowerTrack API*. Disponible en: https://developer.twitter.com/en/docs/tweets/filter-realtime/api-reference/powertrack-stream.

[33] Apache Zookeeper. *Zookeeper Documentation*. Disponible en: https://zookeeper.apache.org/.

A. Diagrama de Gantt

2/19	3/19	4/19	5/19
5, 10, 17, 24	3, 10, 17, 24, 31	7, 14, 21, 28	5, 12, 19, 26

TFG + CTFG

Sprint 0: Investigación, propuesta y formación (34 h)
- Reunión inicial (1,5 h)
- Cursos (30 h)
- Estudio de la API de Twitter (1 h)
- Sprint Planning (1,5 h)

Sprint 1: Desarrollo aplicacion Spark en local (19,25 h)
- Crear producer (1 h)
- Crear consumer (6,5 h)
- Puesta en marcha de Zookeeper y Kafka (4 h)
- Plasmar datos en un mapa (6,25 h)
- Sprint Review (1,5 h)

Sprint 2: Gestión de datos de geolocalización (9,5 h)
- Sprint Planning (0,5 h)
- Trasladar comprobación de ubicación a consumer (1,25 h)
- Exportar datos en GeoJSON (2,25 h)
- Crear caché Redis (4,5 h)
- Guardar timestamp con formato fecha (0,5 h)
- Sprint Review + Sprint Retrospective (1 h)

Sprint 3: Creación de API y mapa (26,5 h)
- Sprint Planning (0,5 h)
- Identificación de topics (3,5 h)
- Desarrollar mapa Leaflet (17 h)
- Construir API Node (4,5 h)
- Sprint Review + Sprint Retrospective (1 h)

Sprint 4: Despliegue en contenedores (24 h)
- Sprint Planning (0,5 h)
- Crear bases de datos Mongo indexadas por tiempo (6,5 h)
- Optimizar página web (3,5 h)
- Crear Docker-Compose (4 h)
- Crear filtros y capas de tweets por tiempo (2,5 h)
- Refactorización de código (6 h)
- Sprint Review + Sprint Retrospective (1 h)

Sprint 5: Despliegue en clusters (90,75 h)
- Sprint Planning (0,5 h)
- Añadir secciones a web (14 h)
- Migración de histórico a Elasticsearch (15 h)
- Documentación API REST (1 h)
- Redacción Anteproyecto (3 h)
- Crear y desplegar cluster Spark (34 h)
- Crear y desplegar cluster Kafka (2 h)
- Despliegue en Openshift (2 h)
- Despliegue Docker-Compose (18,25 h)
- Sprint Review + Sprint Retrospective (1 h)

Created with Free Edition

	2/19	3/19	4/19	5/19
	5 10 17 24	3 10 17 24 31	7 14 21 28	5 12 19 26
Sprint 6: Análisis de datos (110,5 h)				
Sprint Planning (1,5 h)				
Proteger acceso a la API (5 h)				
Login y registro web (9 h)				
Añadir seguridad a BBDD (4 h)				
Instalar Kibana (3 h)				
Cambios en BDs (6h)				
Construcción gráficas Kibana (70 h)				
Sprint Review + Sprint Retrospective (2 h)				
Sprint 7: Documentación (135,5 h)				
Sprint Planning (1,5 h)				
Documentación TFG (76 h)				
Documentación CTFG (57 h)				
Crear y añadir dashboard a webapp (10 h)				
Sprint Review + Sprint Retrospective (2 h)				

B. Gráficas Burndown

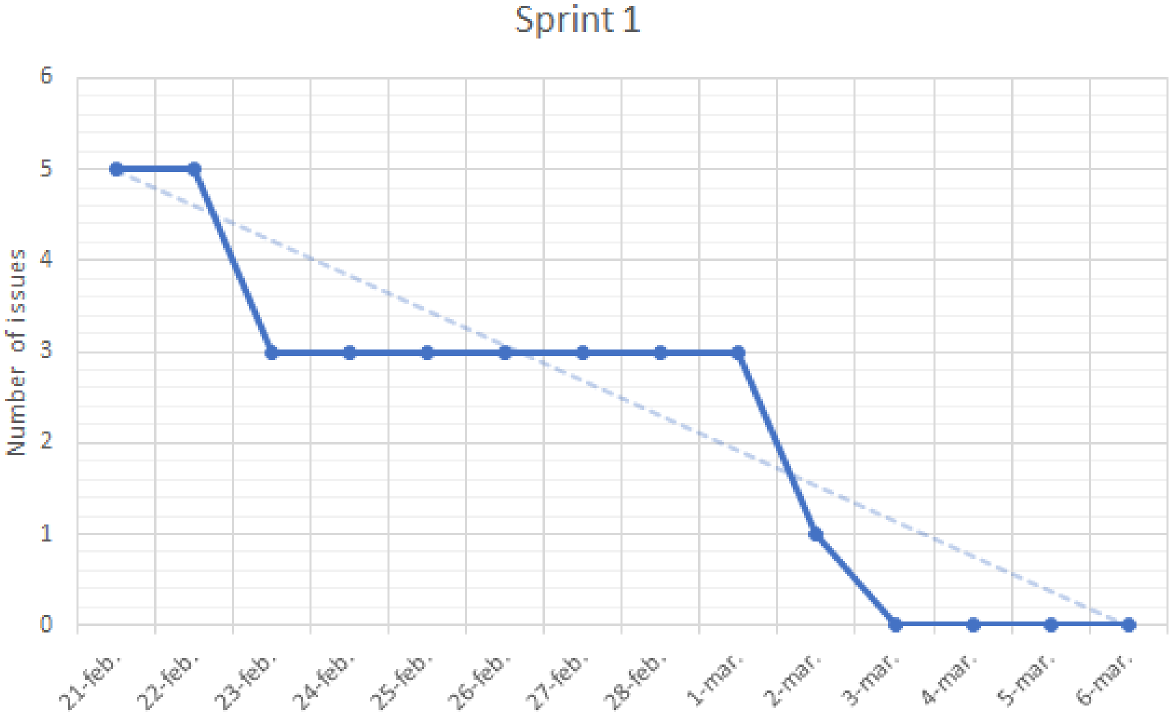

Figura B.1: Gráfico Burndown del Sprint 1.

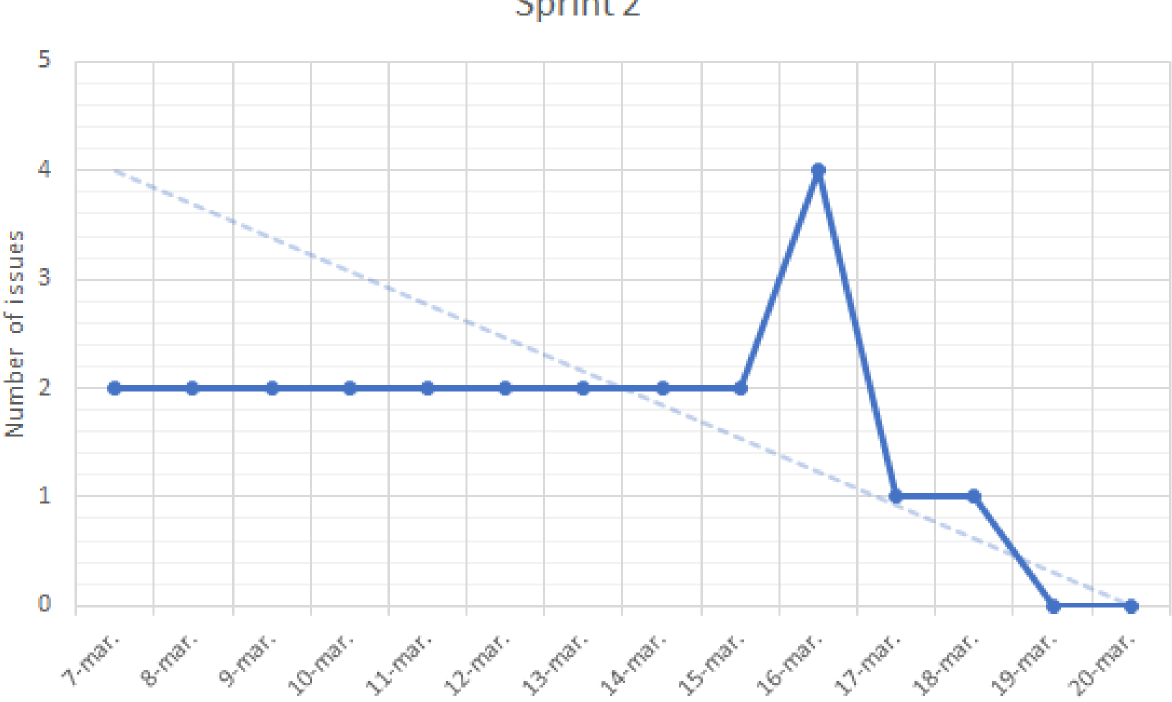

Figura B.2: Gráfico Burndown del Sprint 2.

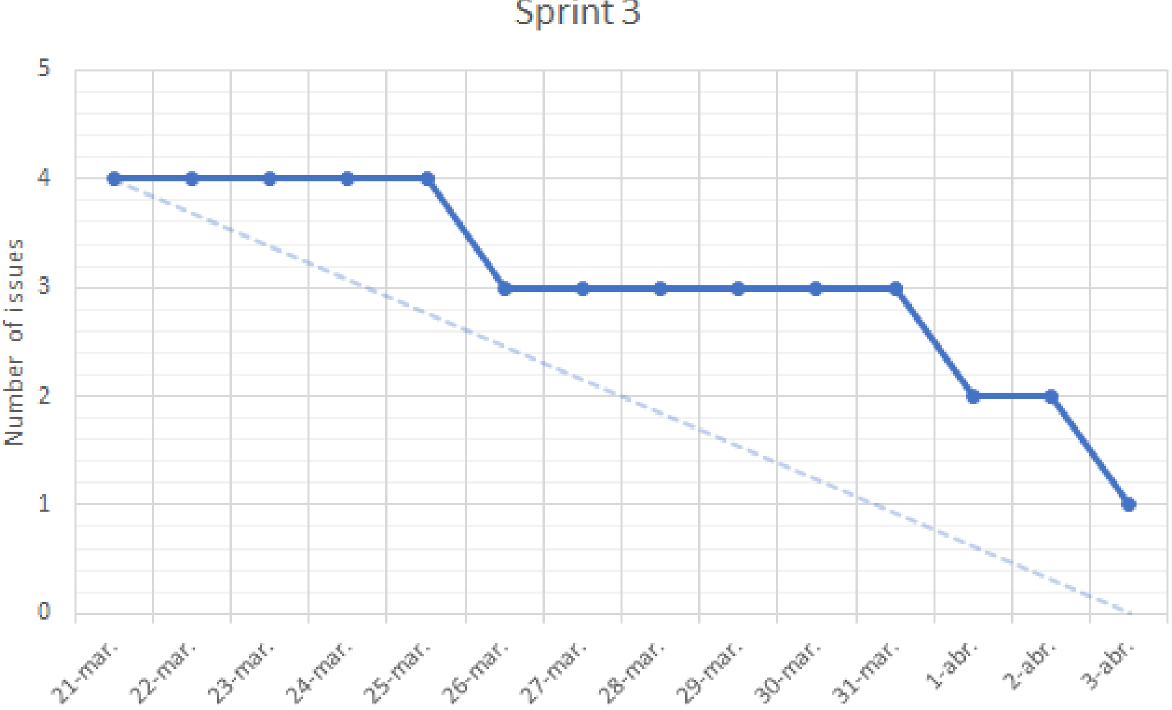

Figura B.3: Gráfico Burndown del Sprint 3.

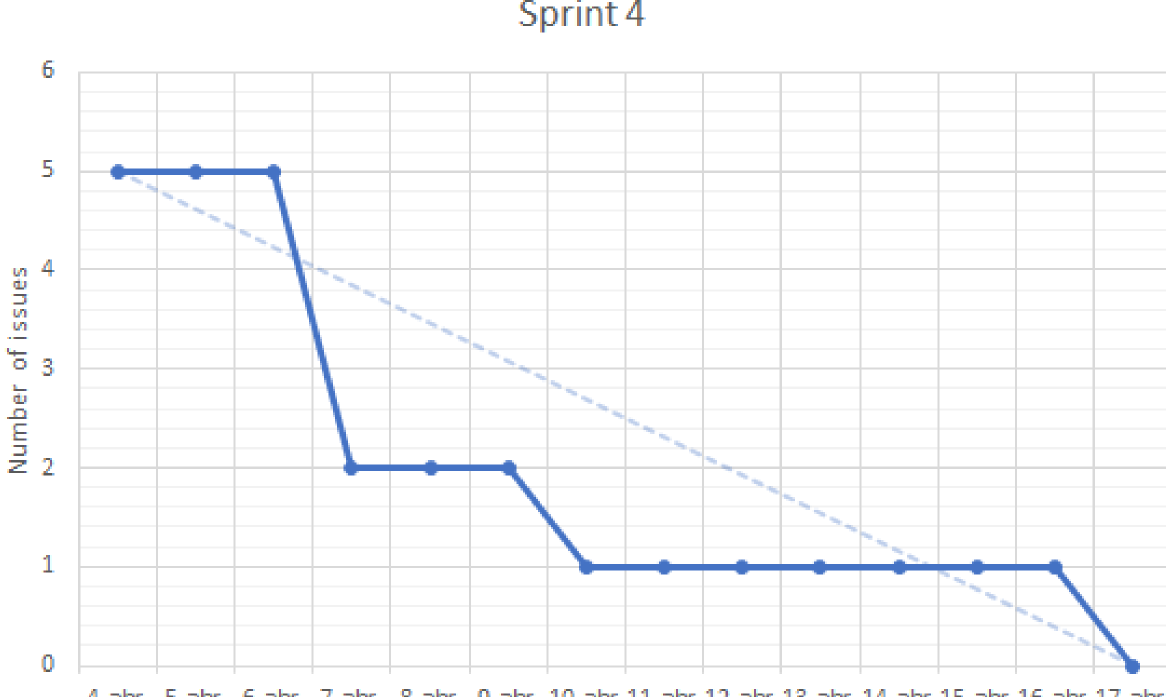

Figura B.4: Gráfico Burndown del Sprint 4.

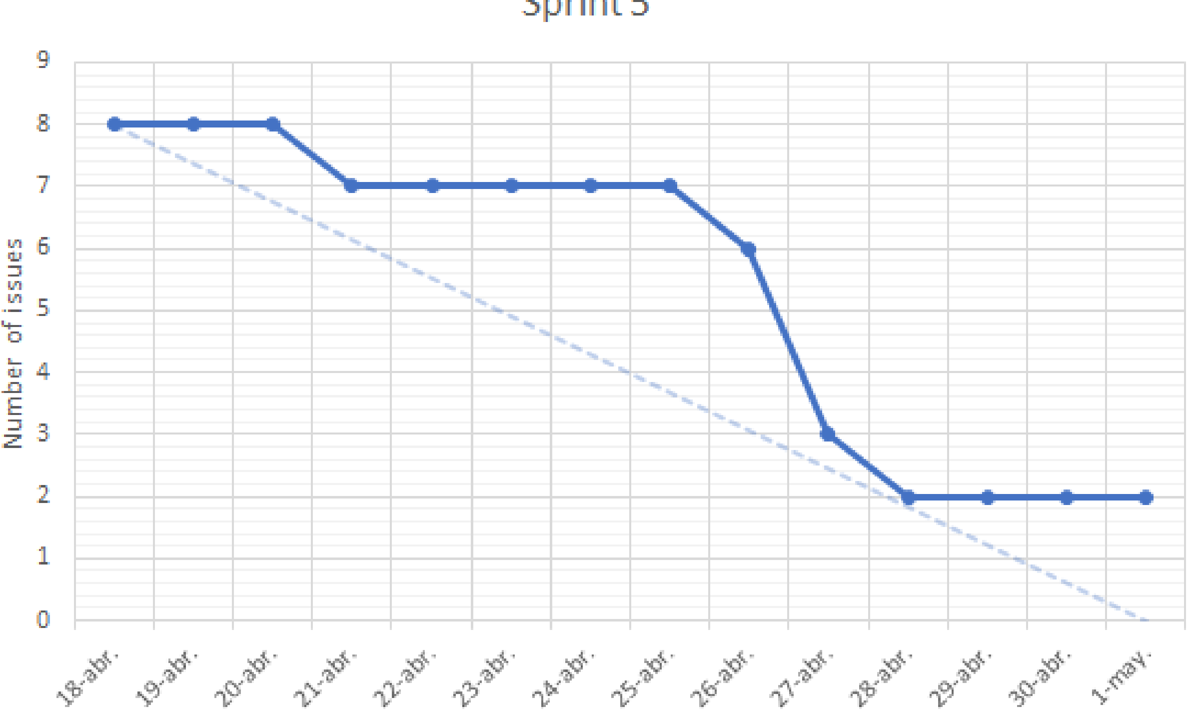

Figura B.5: Gráfico Burndown del Sprint 5.

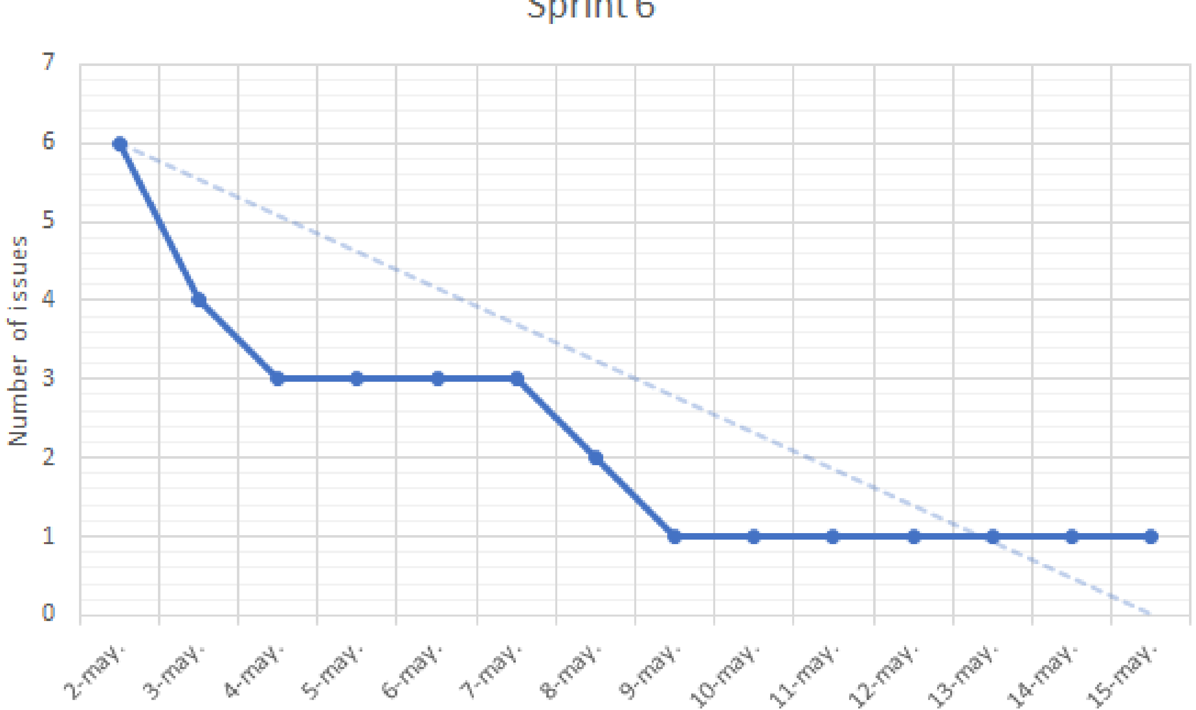

Figura B.6: Gráfico Burndown del Sprint 6.

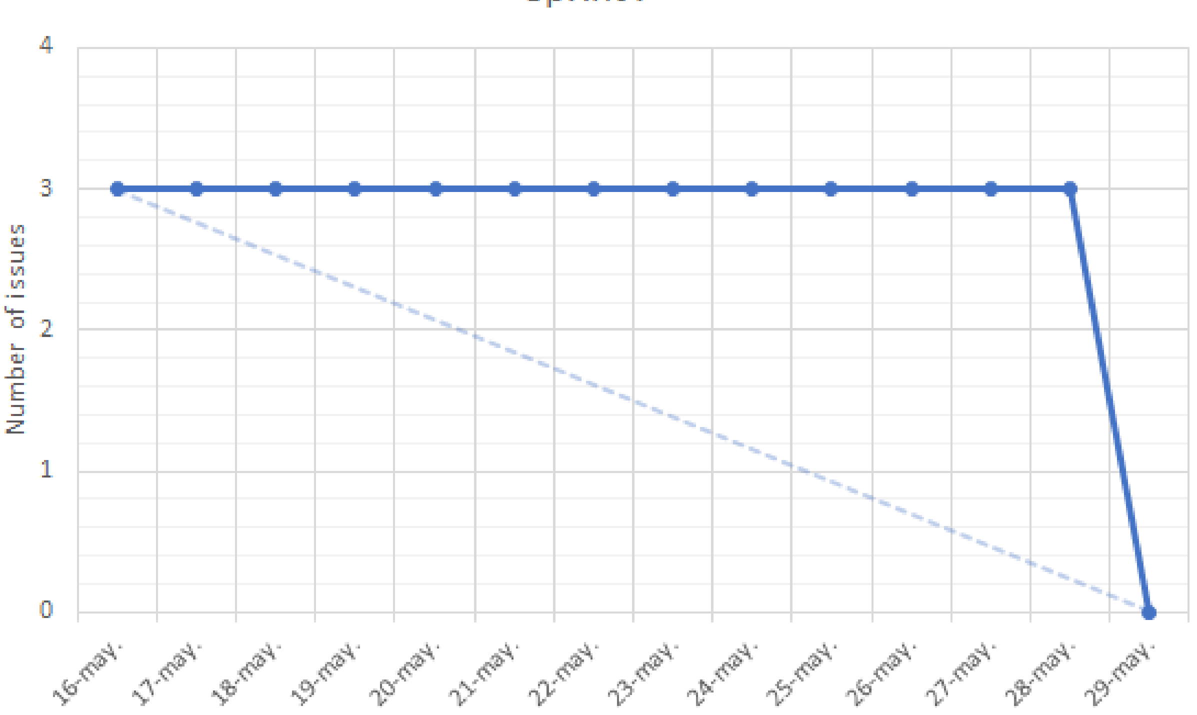

Figura B.7: Gráfico Burndown del Sprint 7.

C. Modelos y esquemas Mongoose

```
const mongoose = require('mongoose');
const { Schema } = mongoose;

const TweetSchema = new Schema({
    id: { type: String, required: true },
    topics: { type: Array, required: true },
    text: { type: String, required: true },
    source: { type: String, required: true },
    hashtags_count: { type: Number, required: true },
    user_mentions_count: { type: Number, required: true },
    user_name: { type: String, required: true },
    followers: { type: Number, required: true },
    friends: { type: Number, required: true },
    verified: { type: Boolean, required: true },
    geo_enabled: { type: Boolean, required: true },
    location: { type: String, required: false },
    longitude: { type: Number, required: true },
    latitude: { type: Number, required: true },
    sensitive: { type: Boolean, required: false },
    lang: { type: String, required: false },
    timestamp: { type: String, required: true },
    date: { type: String, required: true }
});

TweetSchema.index({ topics: 1 });
```

Código C.1: Modelo *tweet.js*

```
const mongoose = require('mongoose');
const { Schema } = mongoose;

const DatabasesSchema = new Schema({
    name: { type: String, required: true },
    URI: { type: String, required: true },
    database_name: { type: String, required: true },
    collection: { type: String, required: true },
    time: { type: Number, required: false },
});
```

Código C.2: Modelo *databases.js*

```
const mongoose = require('mongoose');
const bcrypt = require('bcryptjs');
var config = require('../config');

mongoose.connect('mongodb://' + config.MONGO_USER + ':' + config.
    MONGO_PASSWORD + '@' + 'IP:PORT/users', { useNewUrlParser: true });

const userSchema = mongoose.Schema({
    email: {
        type: String,
        required: true,
        minlength: 5,
        maxlength: 50
    },
    password: {
        type: String,
        required: true,
        minlength: 5,
        maxlength: 50
    },
    role: {
        type: String,
        enum: ['admin', 'user'],
        default: 'user'
    },
    token: {
        type: String,
        required: true
    }
});

userSchema.pre("save", function (next) {
    var user = this;

    if (!user.isModified('password')) {
        return next();
    }

    bcrypt.hash(this.password, 10, (err, hash) => {
        this.password = hash;
        next();
    });
});

userSchema.methods.validatePassword = function(password){
    return bcrypt.compareSync(password, this.password);
};

module.exports = mongoose.model('User', userSchema);
```

Código C.3: Modelo *user.js*

D. Capturas de funcionamiento de la aplicación

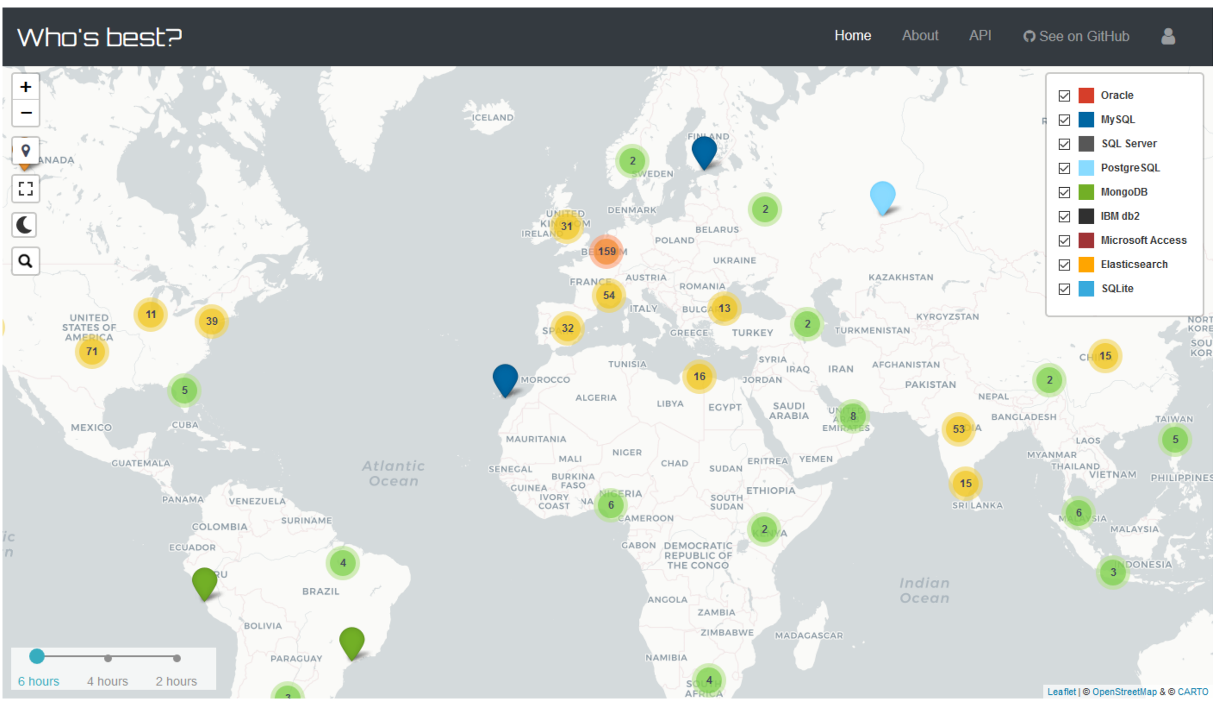

Figura D.1: Vista inicial de la aplicación web.

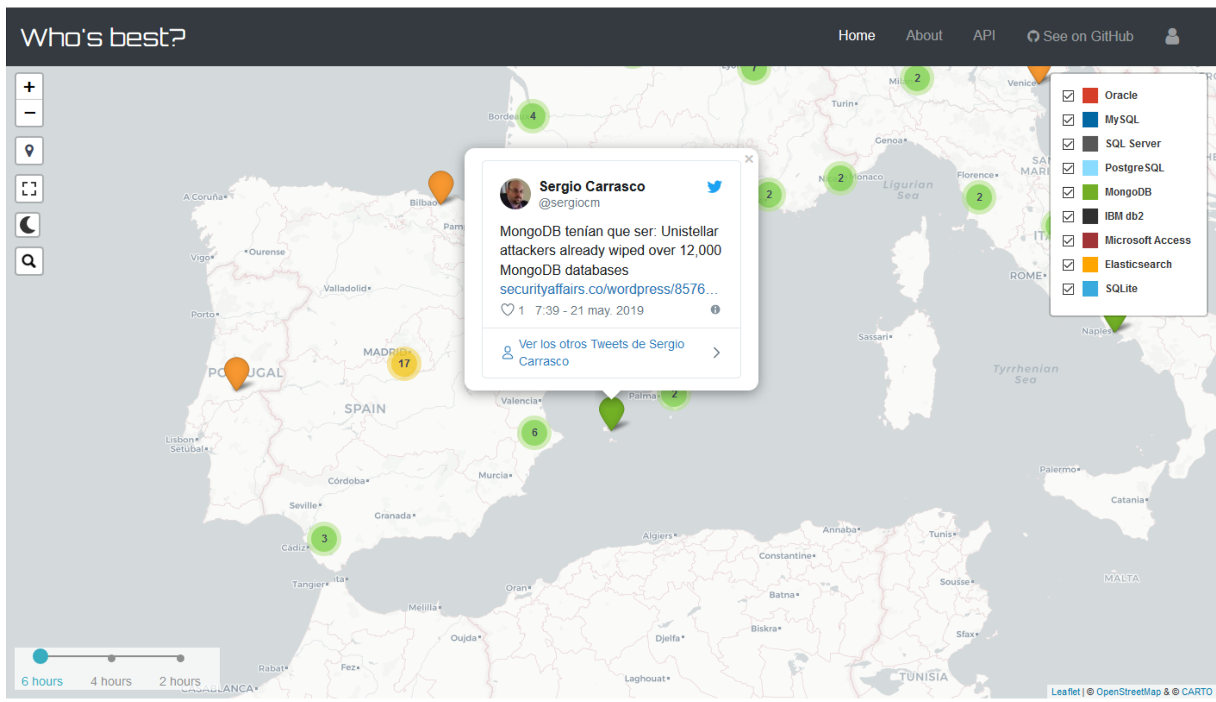

Figura D.2: Vista detalle de un *tweet*.

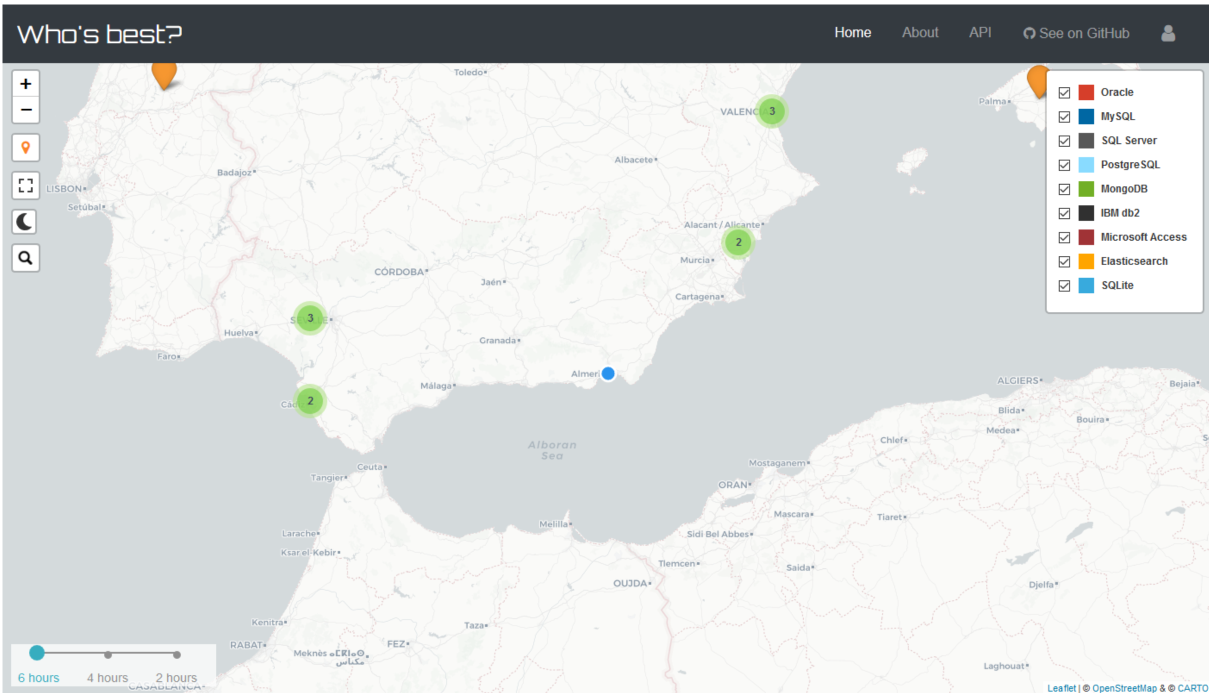

Figura D.3: Vista de ubicación del usuario.

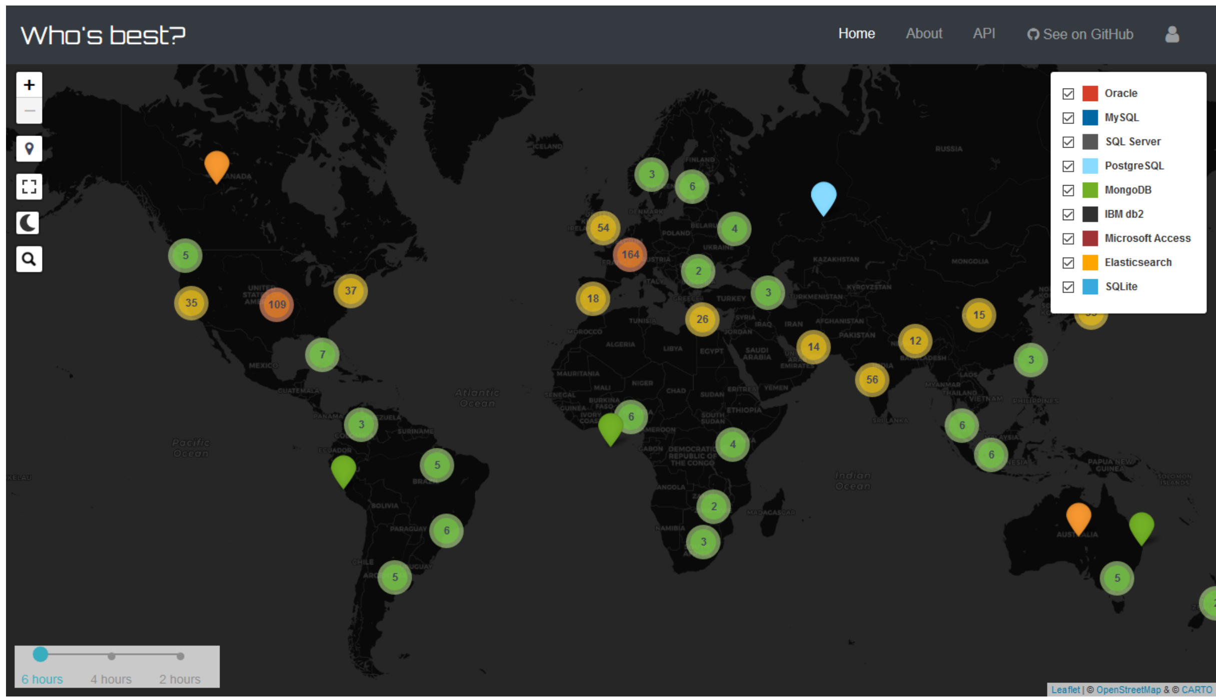

Figura D.4: Vista del mapa en modo nocturno.

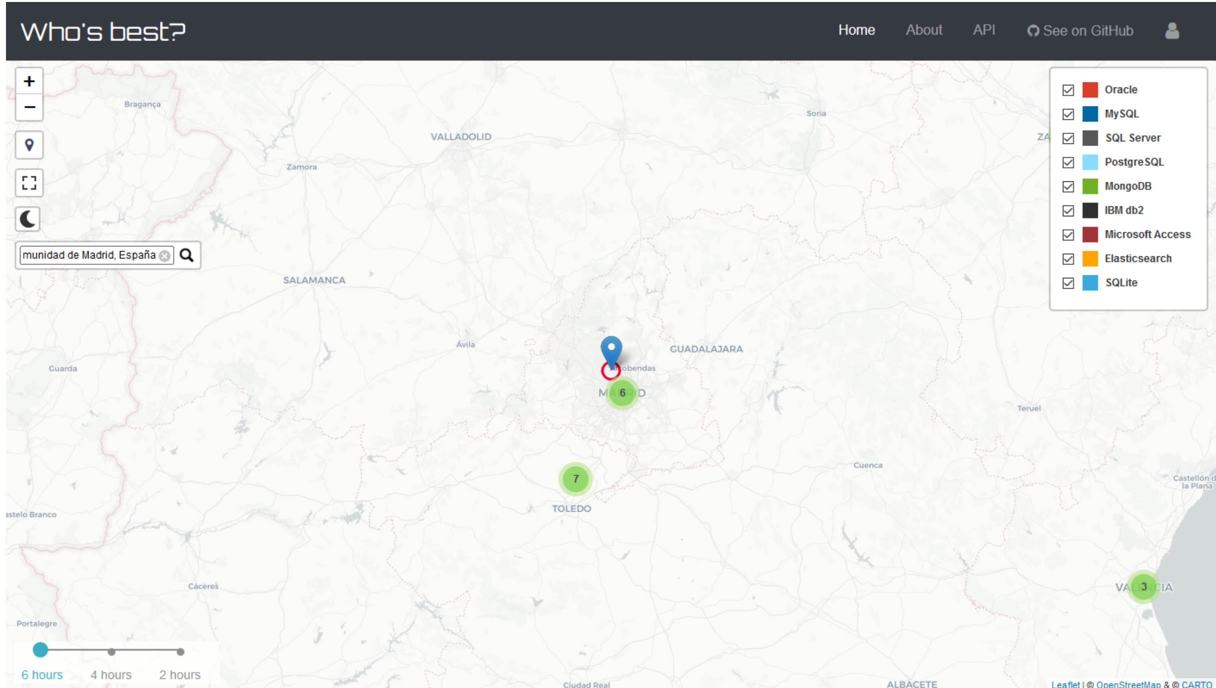

Figura D.5: Vista de búsqueda de dirección.

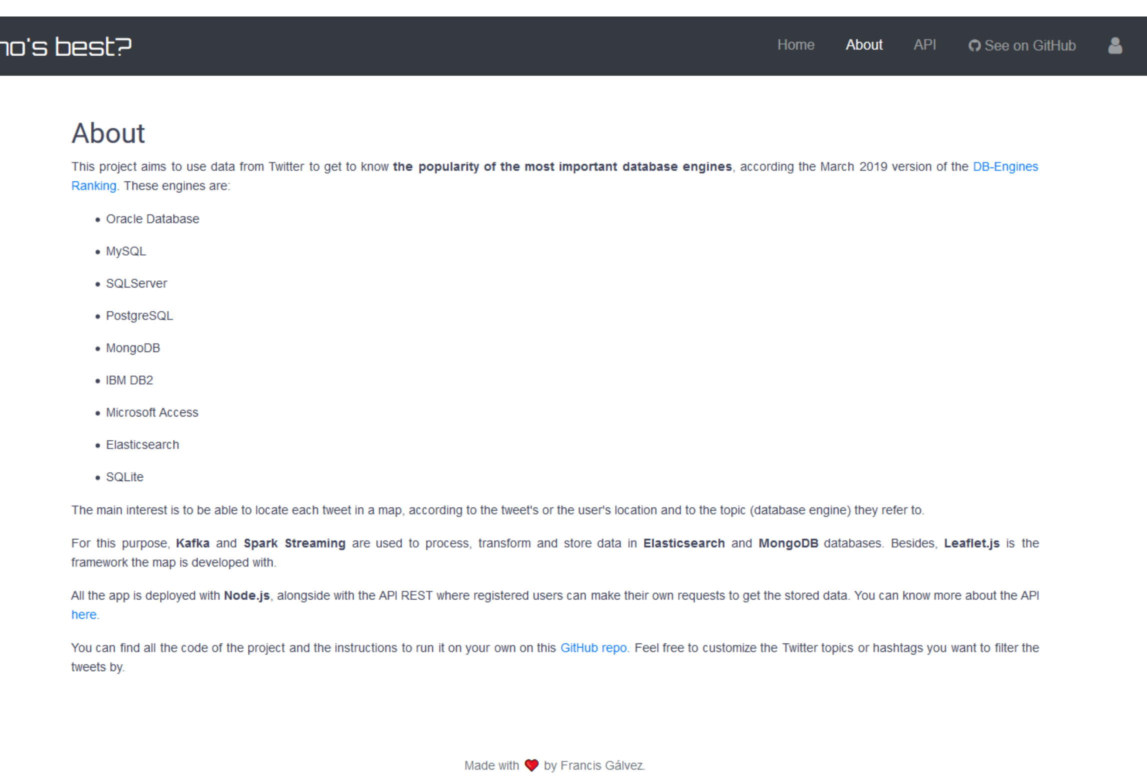

Figura D.6: Vista *About.*

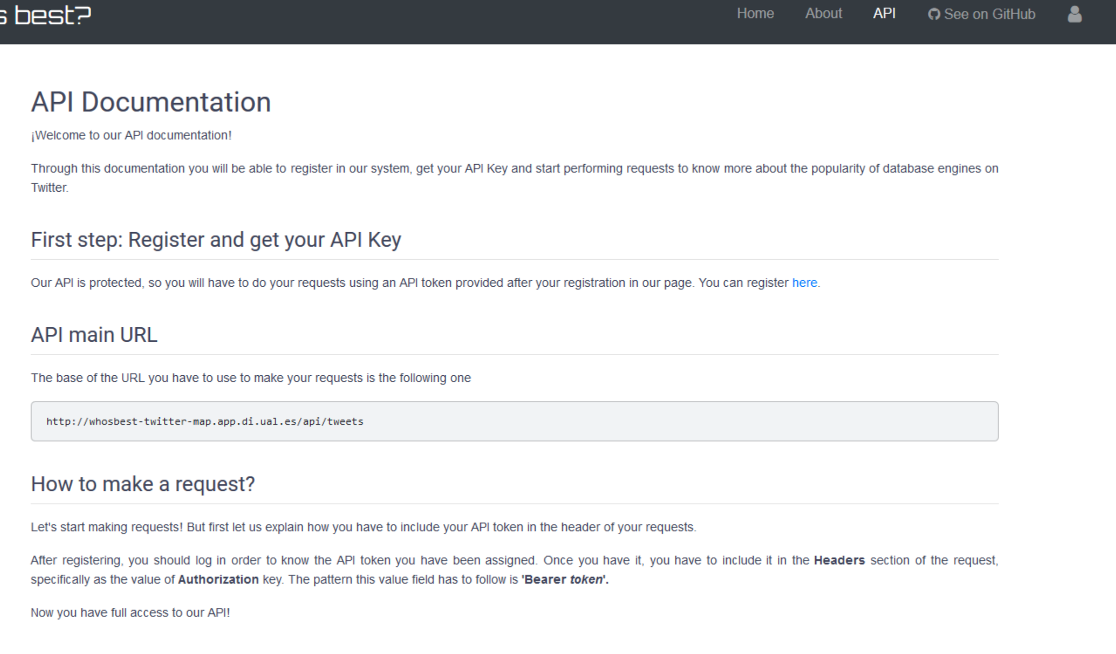

Figura D.7: Vista *API Documentation.*

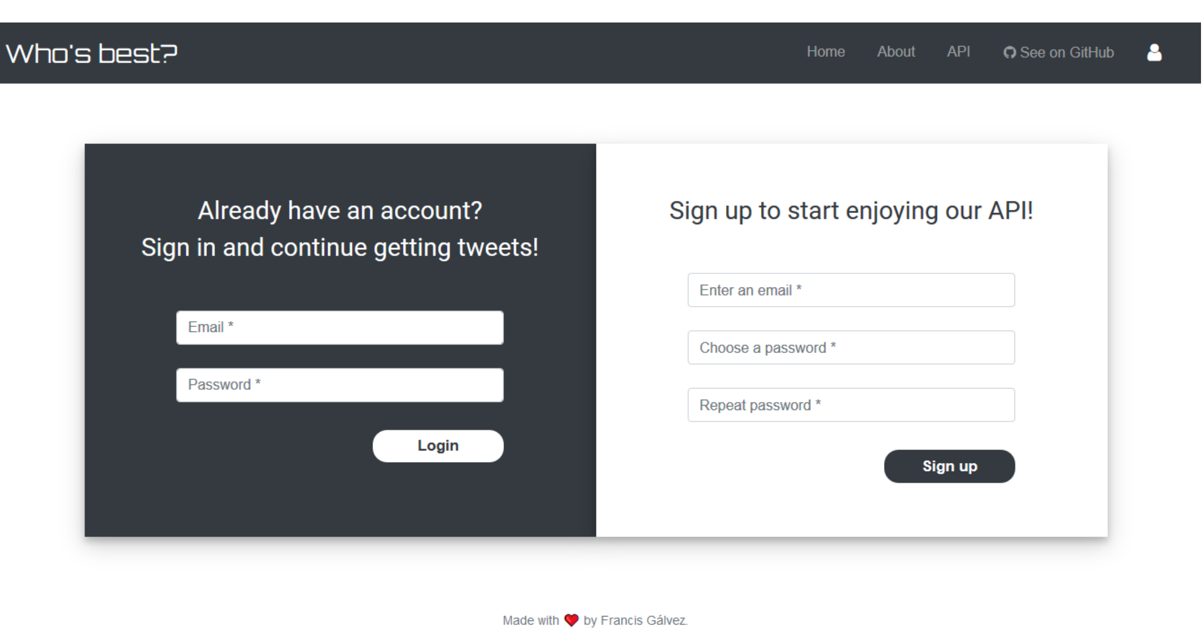

Figura D.8: Vista *Login/Sign up*.

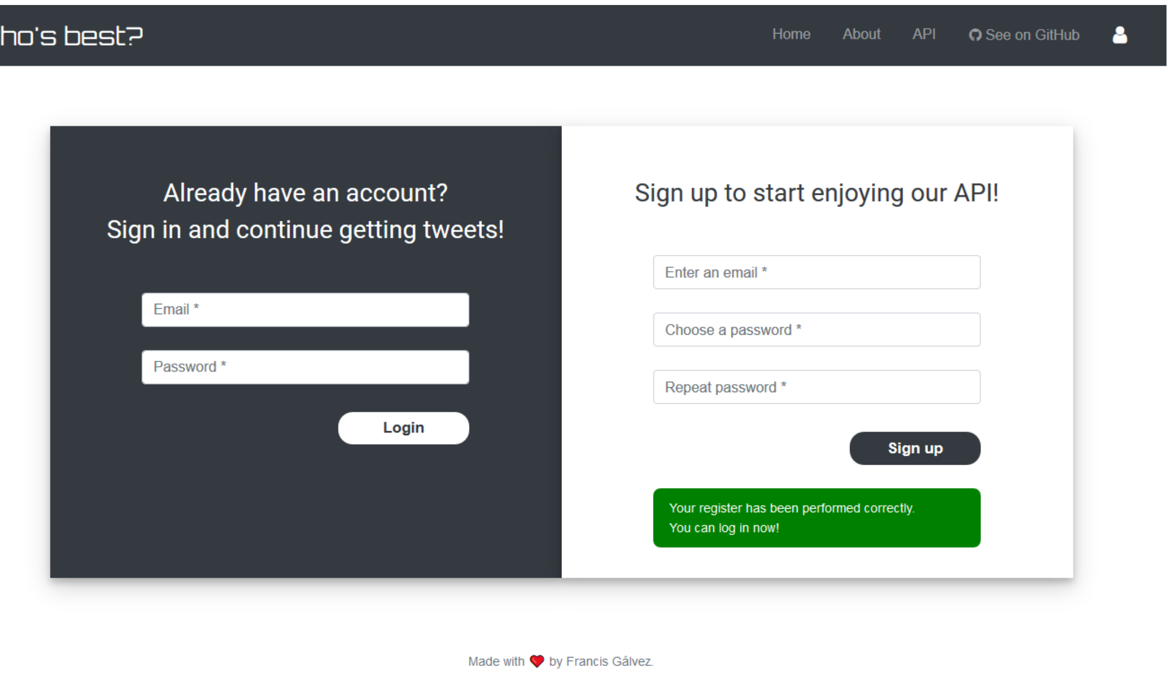

Figura D.9: Vista registro correcto.

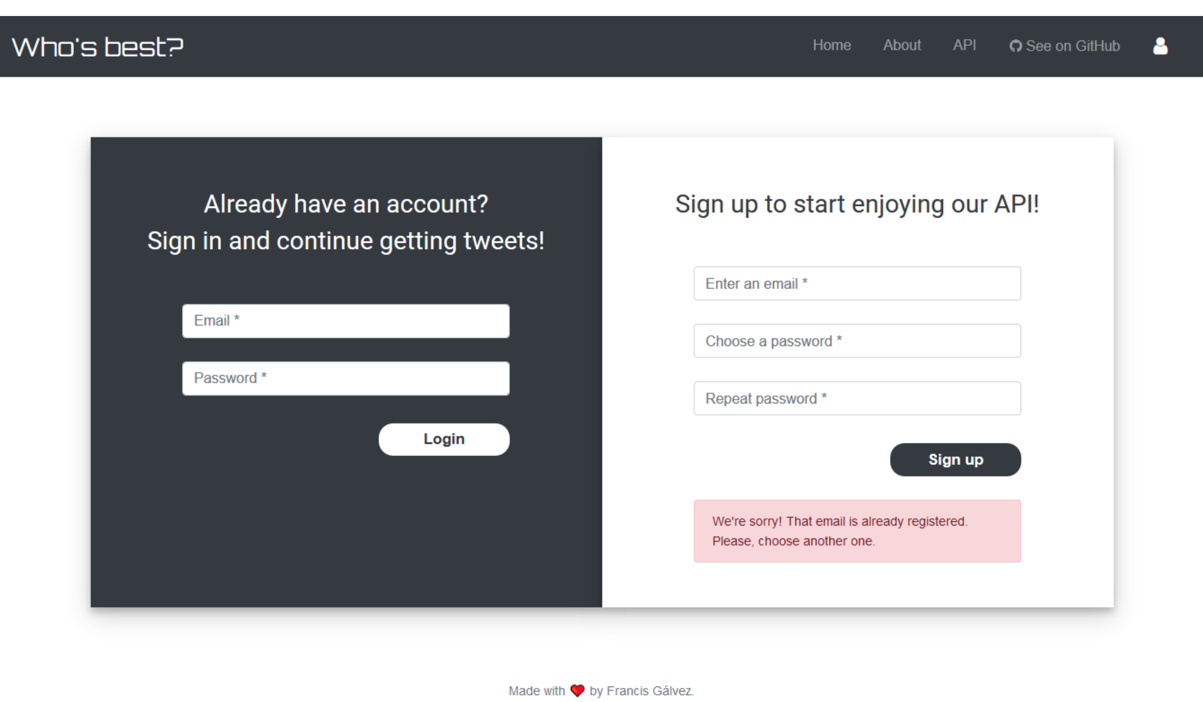

Figura D.10: Vista registro incorrecto: Email ya existente.

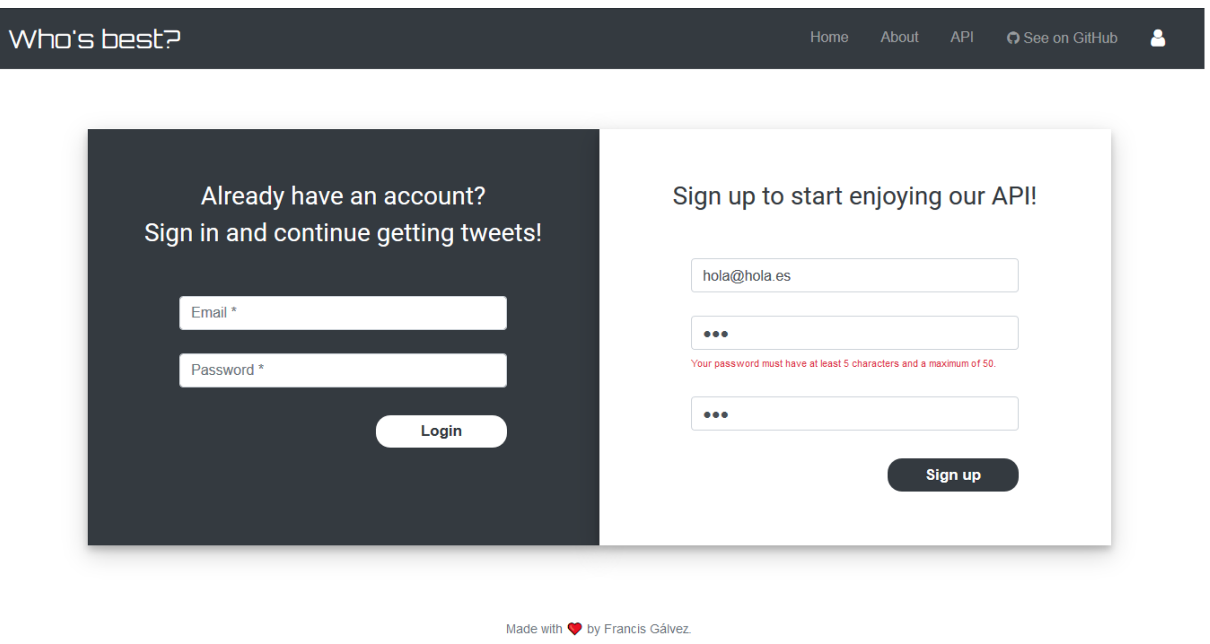

Figura D.11: Vista registro incorrecto: Contraseña con menos de 5 caracteres.

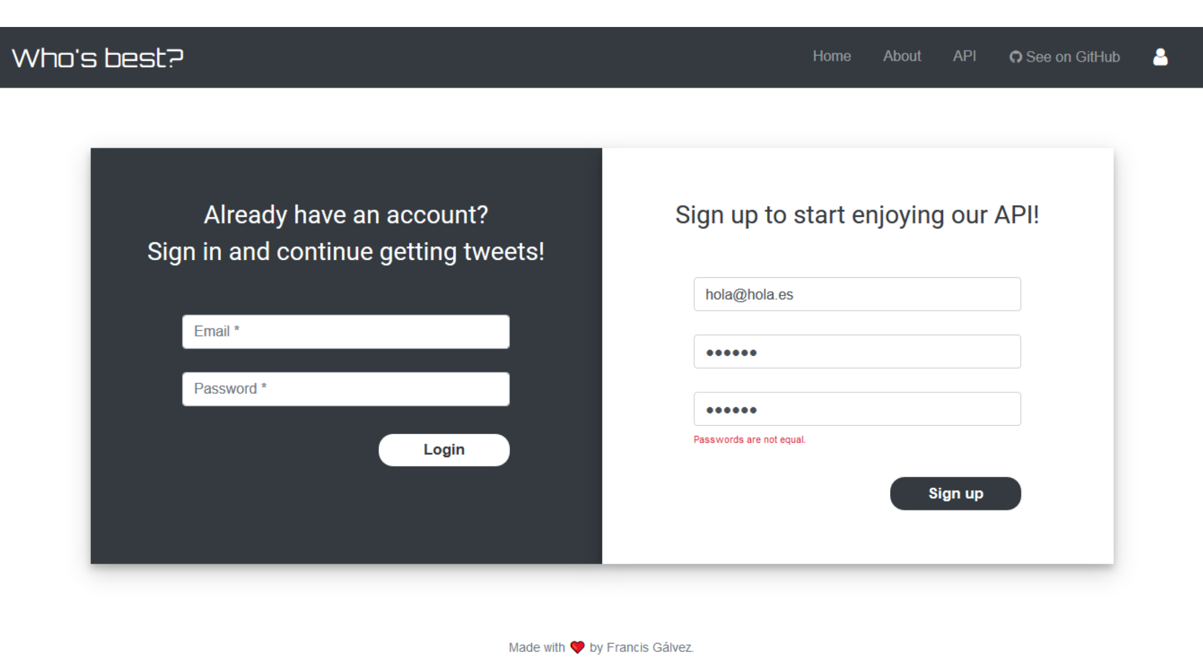

Figura D.12: Vista registro incorrecto: Contraseñas diferentes.

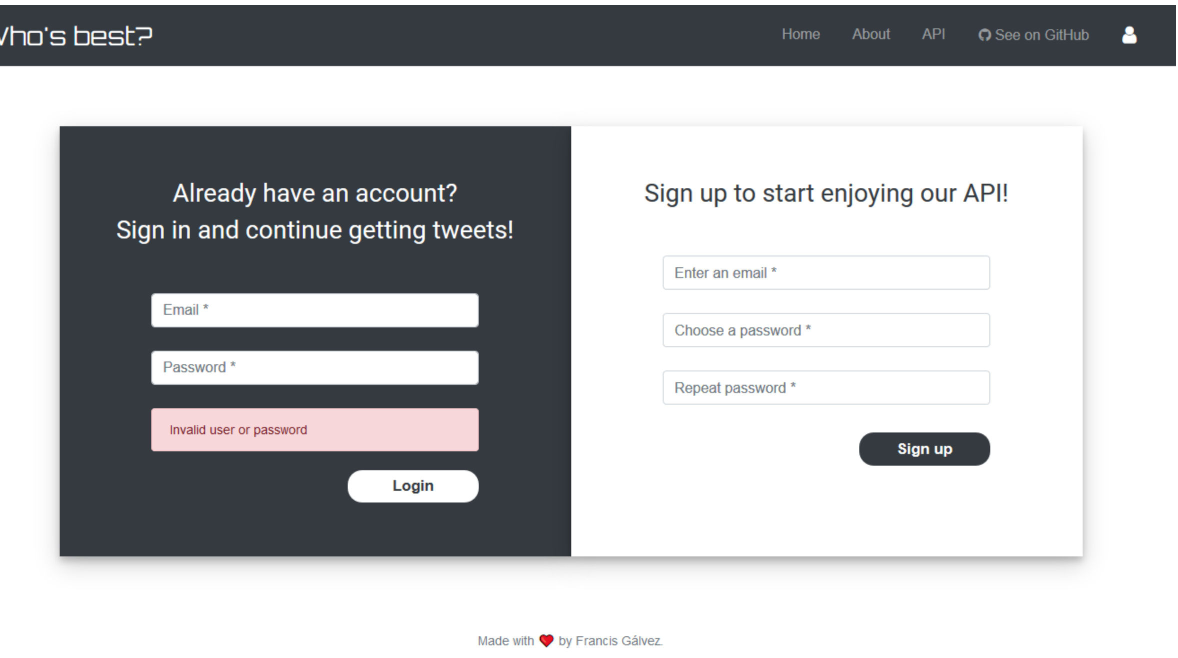

Figura D.13: Vista inicio de sesión incorrecto.

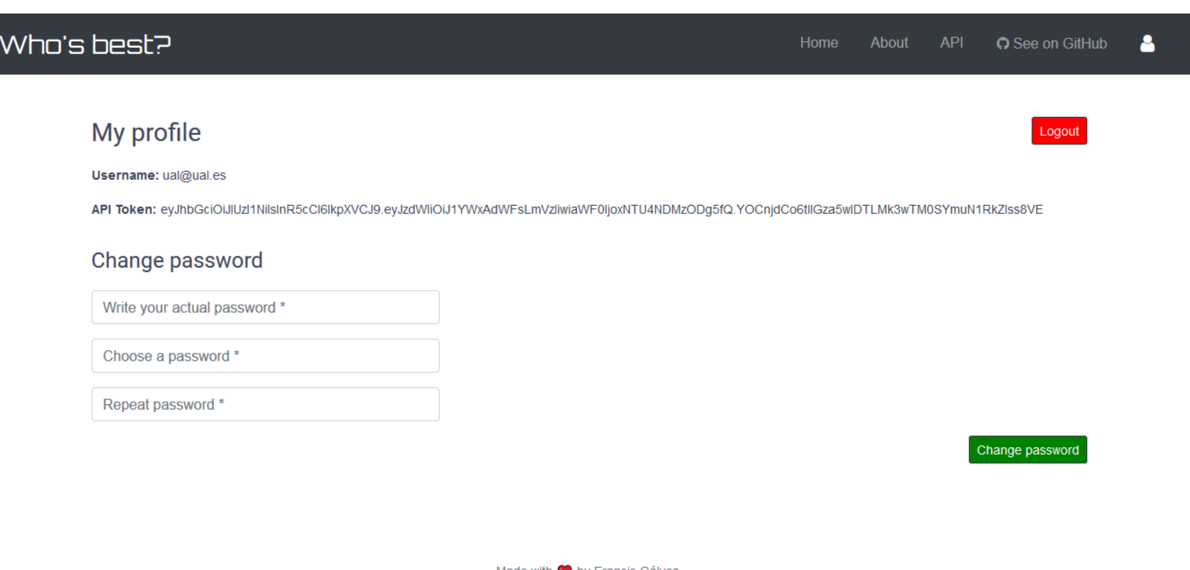

Figura D.14: Vista *dashboard*.

E. Diagramas de colaboración

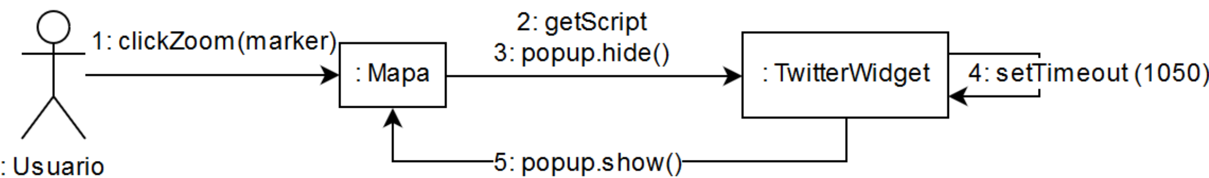

Figura E.1: Diagrama de colaboración: Clic sobre marcador del mapa.

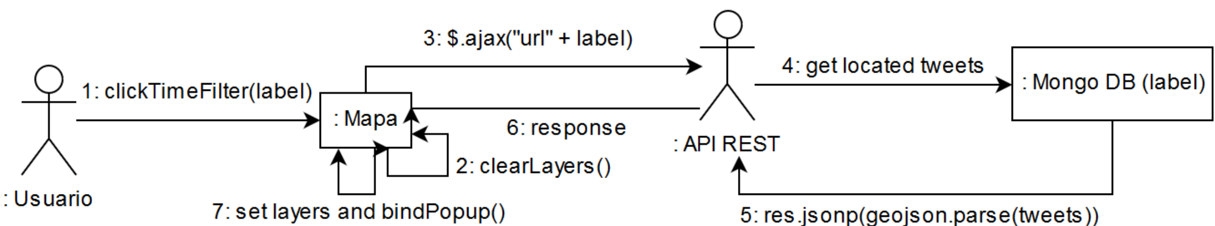

Figura E.2: Diagrama de colaboración: Clic sobre filtro de tiempo del mapa.

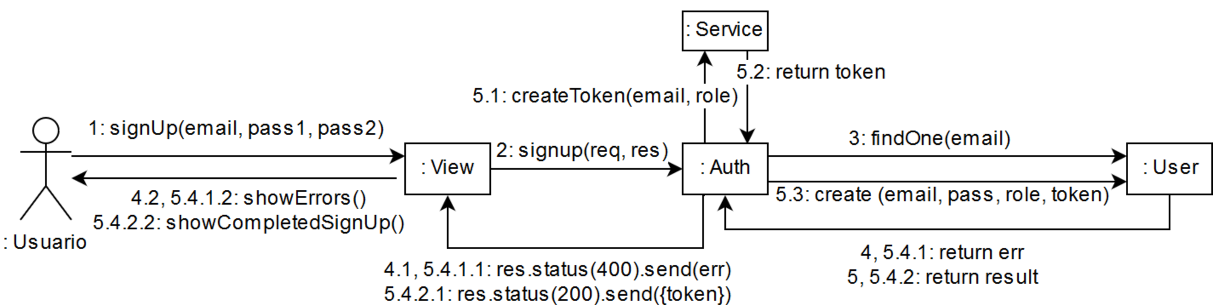

Figura E.3: Diagrama de colaboración: Registro.

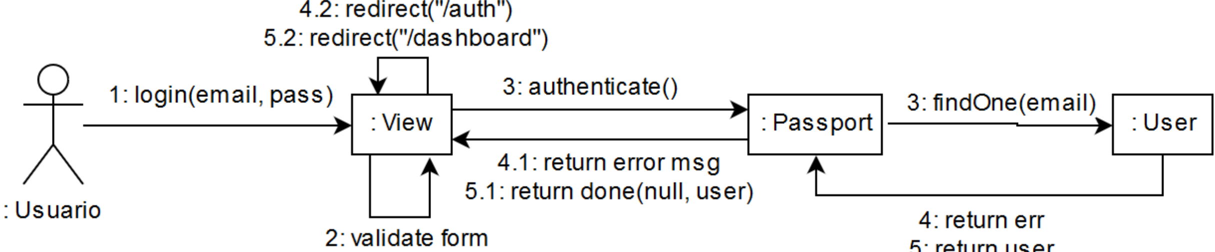

Figura E.4: Diagrama de colaboración: Inicio de sesión.

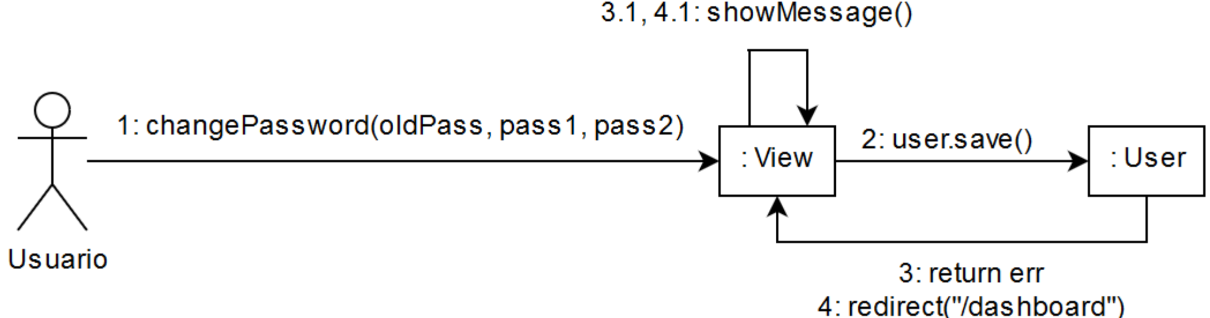

Figura E.5: Diagrama de colaboración: Cambio de contraseña.

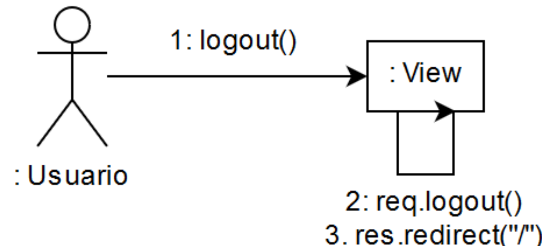

Figura E.6: Diagrama de colaboración: Cierre de sesión.

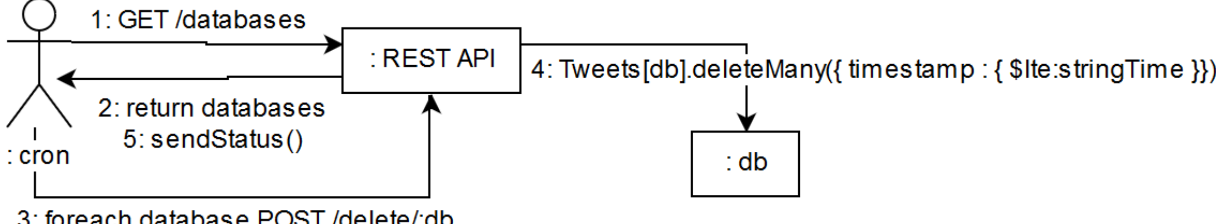

Figura E.7: Diagrama de colaboración: Cron job.

F. Diagrama de casos de uso

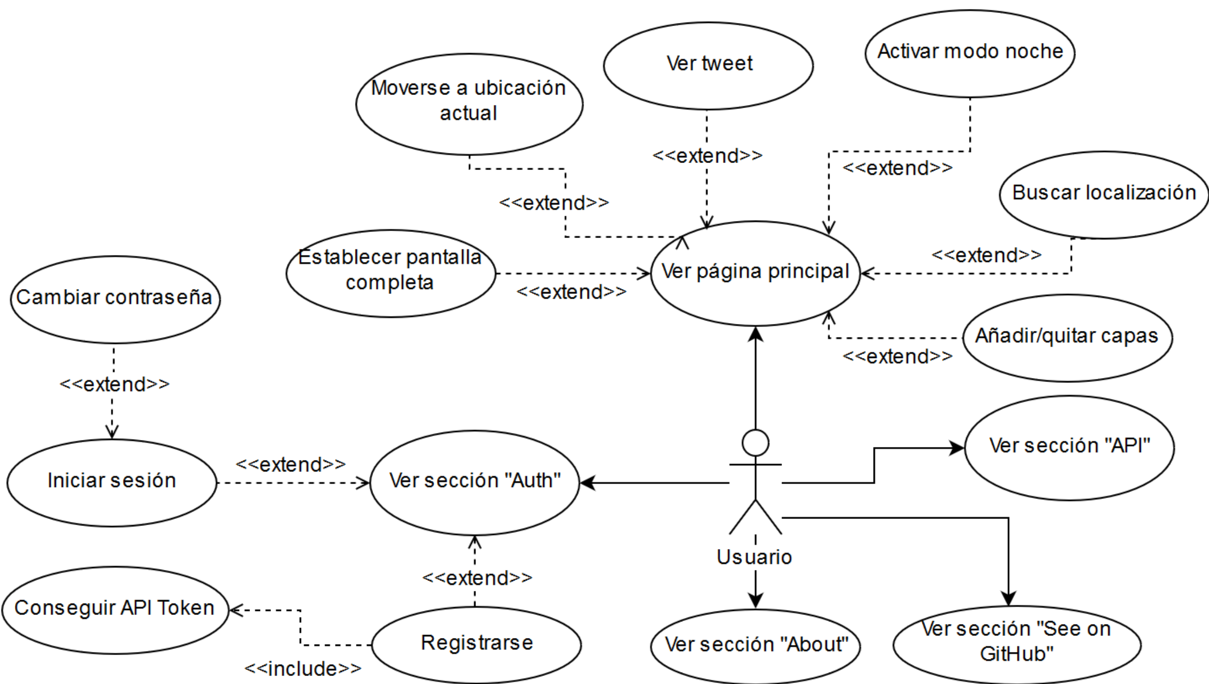

Figura F.1: Diagrama de casos de uso de la aplicación web.