Social Media Analytics using Apache Spark

Application to Market Research

Mauro Gómez Parada

Director: Rubèn Tous
Codirector: Jordi Torres

Màster en Enginyeria Informàtica

Facultat d'Informàtica de Barcelona

Universitat Politècnica de Catalunya

4 Febrero, 2016

"Big Data is like teenage sex:
everyone talks about it,
nobody really knows how to do it,
everyone thinks everyone else is doing it,
so everyone claims they are doing it."

– Dan Ariely –

Agradecimientos

Quiero agradecerle a todas aquellas personas que me han ayudado, de una manera u otra, durante la realización del Trabajo Final de Máster.

Primero de todo agradecerle a Rubèn Tous y Jordi Torres por la paciencia y confianza que han puesto en mi, así como cada uno de sus consejos, ideas y sugerencias que han aportado a este proyecto. Han sido los pilares fundamentales para que esto se pudiera realizar.

También me gustaría agradecérselo a mi familia y a Rebe por la paciencia y el apoyo brindado durante todo este proceso, pero sobre todo durante los fines de semanas o vacaciones en las que he estado trabajando mientras el resto disfrutaba.

Resumen

Cada día más, las redes sociales son la plataforma más destacada y popular para compartir todo tipo de información. Instagram es una plataforma con la que se pueden compartir fotos del día a día, con un total de *40 billones* de fotos compartidas, un promedio de *80 millones* de fotos al día y *400 millones* de usuarios[1]. Los usuarios proporcionan una gran cantidad de información a través de cada imagen que suben, no solo por el contenido de la imagen, sino también con los *hashtags* que añaden a cada una de ellas. Con todo esto, y junto con los metadatos que se almacenan - lugar de captura, lugar de subida, hora - la cantidad de estadísticas e información que se pueden generar son incalculables.

En este trabajo se intentará generar una herramienta de marketing, de la que se pueda obtener información que puede no estar implícita en Instagram, con la que luego poder optimizar las campañas de publicidad que se hagan en esta res social. Para esto, se definirán una serie de eventos y canales con los que se obtendrán en tiempo real, tanto a través de *hashtags* como de *geolocalizaciones*, las imágenes y el contenido que los usuarios vayan subiendo a la red. Esto nos proporcionará una serie de metadatos que el usuario final no puede obtener, así como la posibilidad de analizar las imágenes para obtener más información y presentar una serie de gráficas que aporten valor al usuario final.

Para realizar todo esto, se ha desarrollado un sistema escalable, capaz de adaptarse a flujos enormes de información. Se han utilizado herramientas distribuidas de procesado (*Apache Spark*) y almacenamiento de datos (*Apache Cassandra*), diseñadas para expandirse fácilmente sobre múltiples nodos de cómputo si el volumen de datos lo requiere. Se ha incorporado un módulo de *visual recognition (OpenCV)* - elaborado por Sana Imtiaz - que nos proporcionará información sobre el contenido de cada una de las

imágenes. En un futuro próximo, se prevén optimizaciones con el fin de obtener más información sobre las imágenes, así como profundizar en el análisis de los metadatos y generar más estadísticas con las que aportar ampliar el conocimiento del usuario final.

Contenido

1 Introducción **1**

1.1 Motivación 2

1.2 Objetivos 2

2 Conceptos Previos **4**

2.1 Big Data 4

2.1.1 Nuevo Paradigma 5

2.1.2 Apache Spark 7

2.1.2.1 KMeans 10

2.2 Cassandra 11

2.3 *Visión por Computador* 13

2.4 Teoría de Grafos 14

3 Estado del Arte **18**

3.1 Sistemas existentes 18

3.2 Aplicaciones prácticas 19

4 Especificación **20**

4.1 Requisitos Funcionales 20

4.2 Requisitos no Funcionales 22

4.3 Workflow 23

4.4 Arquitectura 24

5 Diseño **28**

5.1 Frameworks 28

5.2 Capa de presentación 30

5.3 Capa de negocio 33

5.4 Capa de datos 36

6 Implementación **40**
6.1 Capa de presentación 40
6.1.1 Librerías externas 40
6.1.2 Diagrama de Componentes 41
6.2 Capa de negocio 42
6.2.1 Librerías externas 42
6.2.2 Diagrama de Componentes 43
6.2.3 Diagrama de Clases 44
6.2.4 Modulo de Visión 46
6.3 Capa de datos 47
6.3.1 Clúster Cassandra 47
6.3.2 Clúster Spark 47
6.3.2.1 Voronoi script 48

7 Resultados **51**
7.1 Restricciones 51
7.1.1 Rendimiento 51
7.1.2 Instagram API 52
7.2 Resultados visuales 53

8 Conclusiones y futuro trabajo **57**
8.1 Conclusiones generales del proyecto 57
8.1.1 Conclusiones personales 58
8.2 Trabajo futuro 59

Bibliography **61**

Figuras

1.1 Tiempo de uso en las redes sociales 1
1.2 Crecimiento de los usuarios de Instagram 1

2.1 Crecimiento de los datos en los últimos 10 años 4
2.2 Entorno de Spark 9
2.3 Estrategia de replicación simple (RF = 3) 12
2.4 Estrategia de replicación en red (RF = 1) 12
2.5 Representación de una red social 15
2.6 Representación del Betweenness Centrality en un grafo 16
2.7 Representación del Degree Centrality en un grafo 16
2.8 Representación del Closeness Centrality en un grafo 17

4.1 Flujo de ejecución para visualizar un evento 23
4.2 Flujo de ejecución para una actualización de Instagram 24
4.3 Arquitectura del sistema 25
4.4 Arquitectura del sistema 27

5.1 Arquitectura AngularJS 30
5.2 Arquitectura final Angular JS 32
5.3 Ejemplo de uso de directivas 32
5.4 Endpoints de la API REST 34
5.5 Diseño del backend del sistema 35
5.6 Diseño del backend del sistema 37
5.7 Diseño del backend del sistema 38

6.1 Diagrama de Componentes del módulo de AngularJS 42
6.2 Diagrama de Componentes de la capa de lógica 44
6.3 Diagrama de Clases de la capa de lógica 45
6.4 Resultados ejecución 46
6.5 Estado del clúster de Cassandra 47

6.6 Arquitectura del clúster de Spark 48
6.7 Ejemplo mapa de Voronoi 49

7.1 Resumen del Dashboard 53
7.2 Resumen del evento 54
7.3 Últimas 8 imágenes del evento 54
7.4 Visualización del mapa de *Voronoi* a mayor escala 55
7.5 Visualización de las imágenes recibidas como album 55
7.6 Resumen de canales de *hashtags* y formulario 56
7.7 Resumen de canales de *geolocalización* y formulario 56

Capítulo 1

Introducción

Las redes sociales están teniendo un gran impacto en la sociedad, creciendo día a día tanto en el número de usuario que las usan, como en el tiempo de uso que se le dedican [2], tal y como muestra en los siguientes gráficos 1.1 y 1.2.

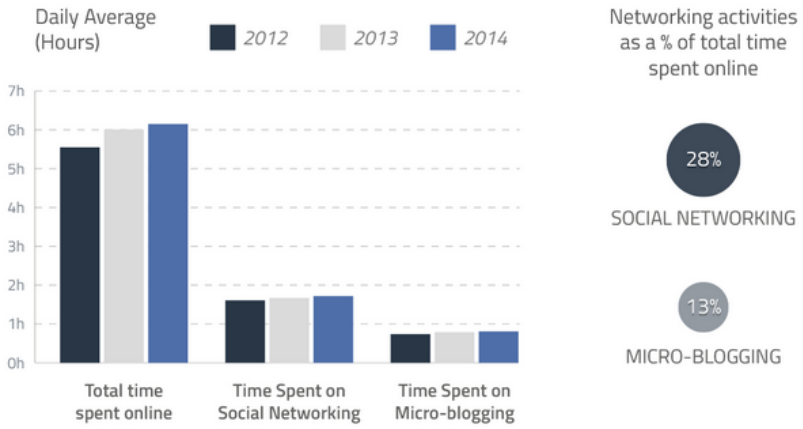

Fig. 1.1: Tiempo de uso en las redes sociales

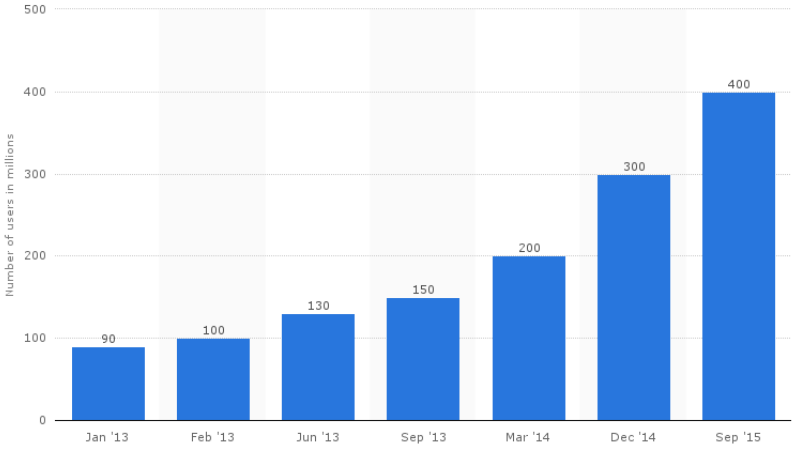

Fig. 1.2: Crecimiento de los usuarios de Instagram

Esto, unido a que el número de usuarios sigue creciendo a un ritmo descomunal [3], hace que las redes sociales - *Facebook*, *Twitter*, *Instagram*, etc.. - sean una gran fuente para obtener información sobre, por ejemplo, las tendencias de un día concreto.

1.1 Motivación

La principal motivación para realizar este proyecto ha sido la escasez de herramientas que aporten información extra a los contenidos que los usuarios suben a Instagram, así como la posibilidad reciente de crear campañas de publicidad en esta red social. También es un gran incentivo el que los usuarios de *Instagram* hayan superado a los de *Twitter*[4], haciendo que el tráfico y el contenido de esta red social aumente con el paso de las horas. Además de esto, el interés por *Apache Spark* y el *Big Data* ha hecho que este proyecto fuese un gran aliciente.

En los últimos años, el contenido de las redes sociales ha aumentado drásticamente y la información que se puede obtener de ellas, consecuentemente, también. Este proyecto intenta obtener información a partir de lo que los usuarios suben a *Instagram* y ofrecerle al usuario un resumen de ésta. Con toda esta información, el usuario del sistema podrá hacer un seguimiento de ciertos tópicos en *Instagram* y orientar sus campañas de márketing a un público más específico, así como comprobar cuales de ellos son los que realmente están teniendo movimiento por parte de los usuarios.

1.2 Objetivos

El principal objetivo, tal y como se ha dicho anteriormente, es crear una herramienta con la que se pueda obtener información no implícita sobre los metadatos y las imágenes de Instagram. Para ello, se ha creado una aplicación web desde la cual se pueden definir una serie de *eventos*, y *canales* asociados, que serán los que indiquen cuáles son los *hashtags* y *geolocalizaciones* de las cuales se quiere recibir información de Instagram.

Para la obtención de información a partir de las imágenes se ha utilizado un proyecto desarrollado por Sana Imtiaz el cual permite, a partir de unos modelos entrenados, predecir a cual de ellos pertenece una imagen. Por tanto, esto nos permite entrenar modelos sobre un determinado tema, y poder aplicarlo a las imágenes recibidas de Instagram, y así obtener más información sobre cada imagen. En el momento actual, se disponen de muy pocos modelos entrenados y solamente se pueden predecir imágenes sobre *sushi*, *paella* y *cerveza*, pero en un futuro cercano se prevé

entrenar una serie de modelos *básicos* para luego, en caso de que algún cliente necesite alguno más concreto, poder entrenarlo y ofrecer una información más concreta.

Para el caso de los metadatos recibidos de Instagram - localización, fecha, etc.. - se han usado algoritmos de *Machine Learning* a través de *Apache Spark*. Gracias a la *geolocalización* que ofrece Instagram en los metadatos, se ha podido generar un diagrama de *Voronoi* [5] en el cual los usuarios pueden observar el reparto de las publicaciones recibidas a través de sus *canales*, tal y como se explicará más adelante.

Capítulo 2

Conceptos Previos

En este capítulo se explicarán en detalle ciertos conceptos que serán necesarios para entender el proyecto que luego se describirá.

2.1 Big Data

La generación de información se ha disparado con la llegada de internet, de echo, en la actualidad cada persona es responsable de los nuevos datos que se generan, desde los teléfonos móviles hasta tarjetas de crédito. También, cada día, se generan millones de datos gracias al ya conocido *Internet of Things* [6]. De todos estos datos se puede obtener un gran valor, pero para ello se necesita procesarlos y saber cuál es la información que queremos obtener. Para entender la cantidad de información que se genera y el crecimiento que está teniendo en estos últimos años, se puede ver la figura 2.1.

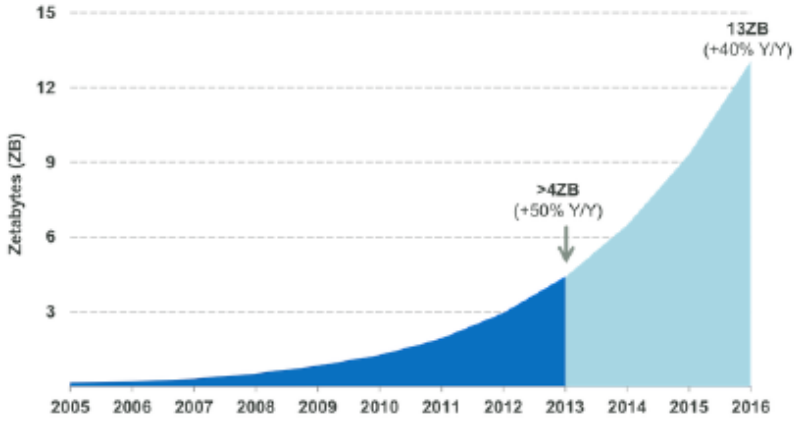

Fig. 2.1: Crecimiento de los datos en los últimos 10 años

En los años 80, la cantidad de datos generados ocupaba *0,02 exabytes*, en estos últimos años, en los que la información *per-capita* se esta doblando cada 40 meses y se prevén que se generen *13 zettabytes* de datos en el año 2016.
Teniendo esta gran cantidad de datos, cabe esperar poder obtener una gran cantidad de información a partir de ellos. Para esto, es necesario el uso de *Big Data* y las técnicas que ofrece.

El término *Big Data* ha sido descrito en 2001 por el grupo de investigación *META Group* como el "*[...] crecimiento constante de datos que supera la capacidad del software habitual para ser capturados, administrados y procesados en un tiempo razonable.*" [7]. Después de que este termino apareciera estuvo apartado de la investigación durante unos años, hasta que en 2008 un grupo de académicos describió el término *big-data computing* en uno de sus *papers* [8].

Hoy en día, tal y como vimos antes, el *Big Data* es fundamental debido al nivel de datos que se manejan. Pero para esto es necesario, también, actualizar los sistemas que analizan y almacenan los datos, ya que los actuales se quedaban anticuados debido a las limitaciones en sus prestaciones. Y la solución tampoco pasa por ampliar las prestaciones, ya que como pudimos ver anteriormente, el crecimiento de la cantidad de datos es exponencial, por lo que llegará un punto en el que no se dispongan de suficiente recursos *hardware*. Por tanto, el nuevo reto de los científicos e investigadores era encontrar un nuevo enfoque que permita analizar y almacenar esta cantidad de datos.

2.1.1 Nuevo Paradigma

Como ya se ha comentado, el *Big Data* trae consigo una grandiosa cantidad de datos, por lo que no es sencillo almacenarlos y gestionarlos en una base de datos común. Se necesita un *software* de calidad que permita que se ejecuten concurrentemente las operaciones y permita distribuirse sobre cientos, o incluso miles, de servidores. Esto permite que los datos se traten con una rapidez mucho mayor y obtener resultados en un tiempo razonable.

El *Big Data* se distancia del clásico, y obsoleto, paradigma del almacenamiento y manipulación, y se basa en cinco principios básicos, conocidos como *las 5Vs del Big Data* [9] [10]. A continuación se hará una descripción sobre esto.

1. *Volumen*

   Se refiere a la gran cantidad de datos generados cada segundo. Solo hay que pensar en todos los *e-mails*, mensajes de *Twitter*, vídeos de *Youtube*, imágenes

de *Instagram*, sensores externos, etc.. Se estan produciendo e intercambiando datos cada segundo [11]. Por tanto, es la cantidad la que nos marca el valor y el potencial de los datos y si realmente se puede considerar *Big Data* o no. Habitualmente se combinan los datos que se poseen con otras fuentes de datos, con el fin de aumentar el volumen de datos a analizar. Este es un punto importante para aquellos que quieren obtener valor de sus datos, y no dejar que mueran almacenados.

2. *Velocidad*

   El término *velocidad* puede referirse, tanto a la generación de datos, como al tiempo de análisis y procesado de los datos para obtener la información demandada. Inicialmente, las compañías analizaban sus datos usando procesos *batch*, es decir, procesos que se ejecutaban en un segundo plano y que, habitualmente, tardaban un tiempo elevado, ya que iban procesando todos los datos. Esto ha cambiado, y ahora las compañias están procesando sus datos en tiempo real. Es decir, según van obteniendo sus datos, los analizan y se quedan con lo realmente valioso que obtienen de ellos. Esto permite, también, que se puedan visualizar los resultados según se obtienen y procesan los datos, con un tiempo de respuesta muy corto. Las nuevas tecnologías son las que se encargan de encontrar soluciones óptimas entre el tiempo de generación de los datos, y el tiempo de procesado, e intentar rebajar este tiempo al menor posible, haciendo posible el análisis en tiempo real. Sin alguna duda, la mejor práctica para aumentar la velocidad del sistema es evitar el almacenamiento de los datos, ya que son demasiados y demasiado rápidos y esto hace que los sistemas de almacenamiento crezcan a un ritmo desmesurado. Por tanto, lo óptimo es procesarlos y guardar solamente aquella información valiosa que se obtiene de ellos.

3. *Variedad*

   Este término se refiere a los diferentes tipos de datos que se obtienen. En el pasado, el *Big Data* se centraba en datos estructurados, como *XML*, que permitían guardarlos en una base de datos relacional. Hoy en día, más del 80% de los datos no son estructurados [10], por lo que almacenarlos en bases de datos relacionales puede causar ciertos problemas. Con la tecnología del *Big Data* se obtiene información de los mensajes, imágenes, sensores, etc.. y todos ellos se almacenan, por lo que la consistencia de los datos en una base de datos relacional sería inviable. Por esta razón, se crearon bases de datos no

relacionales, conocidas como *NoSQL* [12], como por ejemplo *Cassandra*, de la que hablaremos más adelante.

4. *Veracidad*
   La veracidad de los datos es algo muy importante, ya que podemos estar analizando datos que nada tienen que ver con nuestro problema, y que, por tanto, nos van a estropear el análisis. Hay que tener en cuenta siempre la incertidumbre de los grandes volúmenes de datos, y esto se debe a la inconsistencia y a la completitud de los datos, lo que lleva al desafío de mantener los datos constantemente organizados para poder obtener lo máximo de información y que esta sea fiable.

5. *Valor*
   El valor representa el valor de negocio que tienen los datos y la información que podemos extraer de ellos. Cada vez más, las empresas están apostando por la obtención y recolección de un gran volumen de datos relacionados con su *target* los cuales le pueden aportar mucha información de valor de negocio.

2.1.2 Apache Spark

Para el análisis del *Big Data* hay diversos entornos de trabajo que facilitan las tareas de análisis y procesado, como pueden ser *Hadoop*, *Storm* o *Apache Spark*. En este caso, y debido a la popularidad, se ha usado *Apache Spark* [13], por lo que en esta sección se describirá dicho entorno de trabajo. Este *framework* ha sido estudiado y analizado por diversos grupos de investigación, en el caso que nos ocupa, el grupo de investigación en el que se ha realizado este proyecto ha analizado el rendimiento de ejecución en *MareNostrum*, obteniendo datos de ejecución realmente asombrosos tal y como muestra el articulo [14].

Gran parte del contenido de esta sección pertenece al libro *Introducción a Apache Spark* [1] en el cual he colaborado en la escritura de ciertos capítulos.

Spark es una plataforma diseñada para mejorar la velocidad y rendimiento de las aplicaciones *Big Data*. El término inglés *spark* suele usarse para designar cualquier partícula incandescente y suele traducirse por *chispa*. En un sentido metafórico, Spark pretende ser esa chispa de velocidad que permite a las aplicaciones que se ejecutan

[1] Escrito por Mario Macías, Jordi Torres, Rubèn Tous y Mauro Gómez. `http://www.sparkbarcelona.es/`

desde Internet ser más eficientes. De hecho, *Spark* fue diseñado para hacer más rápida la respuesta a consultas interactivas y algoritmos iterativos a partir de la repartición de los procesos a través de la memoria de distintas máquinas.

Spark nació en los laboratorios de investigación de la Universidad de Berkeley como un proyecto de investigación de código abierto en 2009. Sin embargo, empezó a coger renombre a partir del 2013, cuando pasó a ser un proyecto de la *Apache Software Fundation*, y desde entonces lleva una carrera meteórica.

El proyecto *Apache Spark* provee diferentes componentes que definen su ecosistema. En su corazón encontramos el *Spark Core*, el motor responsable para la planificación, distribución y monitorización de aplicaciones ejecutadas en un clúster. Sobre este elemento central se asientan otros componentes especializados de más alto nivel, diseñados para ser fácilmente integrados, como las librerías de aprendizaje automático, por poner un ejemplo.

Los módulos que rodean el núcleo de *Spark* están creciendo con el paso del tiempo para dar soporte a todas las necesidades que van apareciendo. Sin embargo, en estos momentos, los módulos centrales del sistema son los siguientes:

- **Spark Core Engine** es el motor de ejecución de *Spark* sobre el que todos los demás ejecutan sus algoritmos. Soporta persistencia de datos en memoria principal, con tal de proporcionar altas velocidades de ejecución.

- **Spark SQL** da soporte a consultas interactivas para explorar conjuntos de datos mediante un subconjunto del *Structured Query Language* (SQL), ampliamente utilizado en las clásicas bases de datos relacionales.

- **Spark Streaming** permite a las aplicaciones la posibilidad de procesar y analizar los datos en tiempo real. Este módulo permite el procesado de datos generados en tiempo real y en grandes cantidades, como por ejemplo el análisis en tiempo real de *tweets*[2]. El hecho de que *Spark* proporcione en un mismo entorno de trabajo los dos modos de procesado (por lotes o *batch* y *streaming*) es uno de sus puntos fuertes frente a otras alternativas especializadas en uno u otro (por ejemplo Hadoop para el modo *batch* o Storm para el modo *streaming*). La integración de ambos paradigmas convierte a *Spark* en una implementación de lo que a nivel teórico se conoce como arquitectura *lambda*. La manera en que *Spark* incorpora el procesado de flujos de datos sin la necesidad de rehacer

[2]mensajes de texto plano de corta longitud, con un máximo de 140 caracteres, intercambiados en la red social Twitter

todos los demás componentes de la plataforma consiste en capturar pequeños paquetes de datos del flujo de entrada cada cierto intervalo de tiempo y presentarlos como un *RDD*. De este modo, el flujo de entrada toma la forma de una serie continua de *RDDs*, lo que se denomina un *DStream* (del inglés *Discretized Stream*), el concepto esencial y sobre el que gira todo *Spark Streaming*.

- **MLlib** es la librería de aprendizaje automático de *Spark*; una librería escalable que proporciona una amplia gama de algoritmos de alta calidad. MLlib ha experimentado un rápido crecimiento gracias a su comunidad de código abierto de más de 140 colaboradores y sin duda es el componente de todo el ecosistema que más interés suscita y que más rápido evolucionará incorporando muchos algoritmos que en estos momentos aún no incluye. MLlib está diseñado para invocar algoritmos de aprendizaje automático sobre conjuntos de datos numéricos representados en RDDs, y de esta manera permite interaccionar perfectamente con los otros componentes de Apache Spark. Es importante hacer notar que MLlib solo proporciona algoritmos que puedan ser paralelizados en un entorno paralelo o distribuido. Por esta razón, de momento, alguno de los algoritmos clásicos de aprendizaje automático no se encuentran incluidos en la librería MLlib.

- **GraphX** es un motor para el análisis de grafos, construido sobre el núcleo de *Spark* y que permite a los usuarios crear, transformar y obtener conclusiones sobre datos estructurados.

- **SparkR** es un paquete que integra *Spark* con el lenguaje estadístico R, el cual está ganando popularidad desde el estallido del *Big Data*.

En la siguiente figura se puede ver cómo está repartido el entorno de *Spark*.

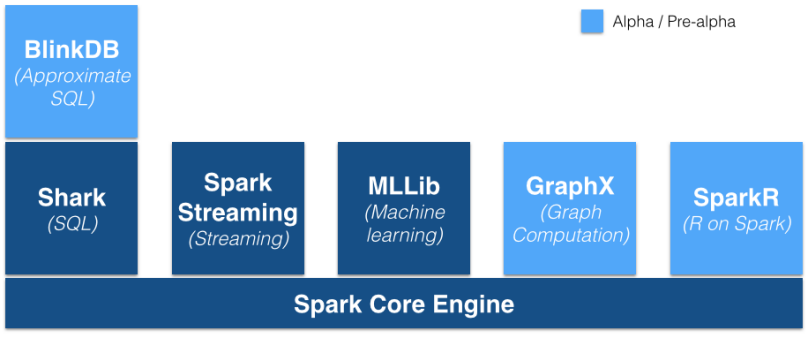

Fig. 2.2: Entorno de Spark

En este proyecto se trabajará sobre el módulo de *KMeans*, más adelante se describirá en detalle el uso que se ha hecho.

2.1.2.1 KMeans

El agrupamiento o *clustering* es un problema de aprendizaje no supervisado que permite agrupar una serie de subconjuntos de entidades relacionados con alguna base de conocimiento. El agrupamiento se utiliza a menudo para el análisis exploratorio previo de los datos.

El módulo MLlib ofrece una serie de métodos para usar *K-Means*, un algoritmo de agrupamiento muy usado que nos ofrece la posibilidad de agrupar una serie de datos en un número predefinido de grupos. La implementación que ofrece MLlib incluye una variante del algoritmo tradicional, llamada *K-Means||*, que ofrece paralelización en la ejecución.

El algoritmo *KMeans* [15] es uno de los algoritmos más simples para realizar agrupamientos. Existen otros algoritmos que realizan esto con mayor precisión, como puede ser *DBSCAN*, *Expectation Minimization*, pero *KMeans* permite ejecutarlo de manera distribuida, y con un coste computacional bajo, muy fácilmente. *KMeans* trata de encontrar un agrupamiento minimizando el error cuadrático *WCSS* 2.1:

$$WCSS = \sum_{i=1}^{k} \sum_{x \in S_i} (\parallel \chi - \mu_i \parallel)^2 \tag{2.1}$$

donde los datos se denotan como $(\chi_1, \chi_2, ..., \chi_n)$ y que serán particionados en los conjuntos $(S_1, S_2, ..., S_k)$, y por último, μ_i, que representa la media de los puntos del conjunto S_i.

El algoritmo *KMeans* se ejecuta en 6 simples pasos [15]:

1. Seleccionar *k centroides*. En el algoritmo original, estos puntos son seleccionados aleatoriamente. Sin embargo, para optimizar el resultado, es conveniente calcularlos mediante el método *Arthur and Vassilvitskii* [16] el cual permitirá reducir el error cuadrático.

2. Calcular la distancia *Euclídea* media entre los puntos de los datos, y los *centroides* de cada grupo.

3. Asignar cada punto de los datos al *centroide* con la distancia *Euclídea* más pequeña.

4. Recalcular los nuevos *centroides* a partir de la media de los puntos de cada uno de los grupos.

5. Recalcular la distancia *Euclídea* entre los puntos y los nuevos *centroides*.

6. Si ninguno de los puntos se modifica de su grupo, o se han alcanzado el máximo de iteraciones, se finaliza. En otro caso, se vuelve al paso 3.

2.2 Cassandra

Apache Cassandra es una base de datos *NoSQL* [17] que ofrece un alto rendimiento. Las bases de datos relaciones tradicionales como *Oracle* o *Microsoft SQL Server* han sido los almacenes de datos primarios para aplicaciones de datos, hasta la década de los 80. Desde que comenzó el *Big Data*, fueron necesarias bases de datos que pudieran almacenar una gran cantidad de datos y a una gran velocidad. Fue ese el motivo por el cual se ha desarrollado esta base de datos, fundamental en este proyecto.

La arquitectura de Cassandra es muy flexible y permite una gran escalabilidad, potencia y ofrece una disponibilidad continua de los datos. *Cassandra* ha sido creado desde el principio pensando que el *hardware* que ejecuta la base de datos puede, y va, a fallar. Es por esto por lo que Cassandra gestiona los datos de tal forma que se puedan consultar a pesar de que uno de los nodos falle. En lugar de mantener un obsoleto diseño *maestro-esclavo*, *Cassandra* implementa una arquitectura *peer-to-peer* distribuido, que hace que la configuración y el funcionamiento sea mucho más sencillo y ligero que con otras arquitecturas. Esto hace que los nodos se comuniquen entre sí, de modo que se intercambian información acerca de cómo está el resto del sistema, y puedan obtener datos unos de los otros. *Cassandra* está pensado para una arquitectura completamente escalable y permite manejar *petabytes* de información y miles de *queries* concurrentes por segundo. Por tanto, gracias a la arquitectura *peer-to-peer* y al factor de replicación, es posible que algunos nodos fallen y *Cassandra* siga estando disponible. El factor de replicas (*replication factor*) es el número de nodos que deberán almacenar un mismo contenido, y se especifica al nivel de *keyspace*. Existen dos posibles estrategias de replicación [18]:

1. *Simple Replication Strategy*
   Los datos se almacenan en el nodo en el que ha sido insertado, y en los siguiente nodos, siguiendo un sentido horario. En la siguiente imagen se puede apreciar cómo seria el reparto.

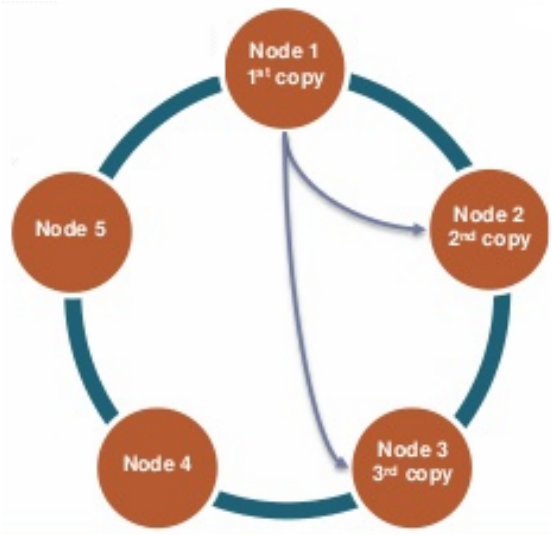

Fig. 2.3: Estrategia de replicación simple (RF = 3)

2. *Network Topology Strategy*
   Los datos se almacenan en el nodo en el que han sido insertados, y se replican en los nodos del mismo *datacenter*, pero en diferentes *racks*, siempre que sea posible.

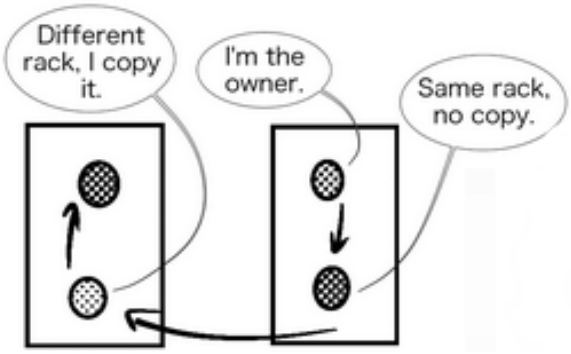

Fig. 2.4: Estrategia de replicación en red (RF = 1)

Otro de los puntos claves de Cassandra es el nivel de consistencia. Este parámetro define el nivel de consistencia de los datos en todas sus replicas. *Cassandra* extiende este concepto hasta el punto *consistencia eventual* al poder definir *consistencia sintonizable*, es decir, que cada operación pueda elegir el nivel de consistencia de los

datos solicitados. Esto, sobre todo, es a nivel de lectura, ya que puede haber algunas *queries* cuyo nivel de consistencia no deba ser tan alto como el de algunas otras.

2.3 *Visión por Computador*

En este sistema se han empleado técnicas de reconocimiento de imágenes, con el fin de obtener información de las imágenes que los usuarios suben a *Instagram*. En esta sección se hará una pequeña introducción a las diferentes técnicas que existen hoy en día, sin entrar en gran detalle ya que no entra en el alcance del del proyecto.

Hoy en día, la visión por computador está avanzando a pasos agigantados. Cada vez son más las aplicaciones que hacen uso de estas técnicas y por ello es necesario que los sistemas de reconocimiento sean más fiables y robustos. En las últimas décadas, las técnicas que se han empleado y las que más han avanzado, son las siguientes:

- *Bag of Words*[19]
  En la visión por computador, este modelo se aplica para la clasificación de imágenes mediante el tratamiento de las características de la imagen como palabras. Para ello se crean una serie de vectores que contienen ciertas características de las imágenes, y que se utilizarán para describir a cada una de las imágenes. Las características que construyen cada uno de los vectores suelen ser patrones visuales que construyen cada una de las imágenes, y la aparición de estas características son únicas para cada una de las categorías. Por ejemplo, una imagen de una manzana tendrá una serie de curvas y pequeñas parcelas circulares irregulares, las cual nos permitirá reconocerla [20].
  Además de todo esto, este algoritmo permite realizar un análisis visual de alta precisión con un coste computacional relativamente bajo.

- *Part-Based Model*
  Part-Based model de refiere a una amplia clase de algoritmos usados para el reconocimiento de imágenes. Estos modelos hacen uso de diversas partes de una imagen para comprobar sí, y donde, existe un objeto de interés. Las relativas localizaciones entre las diferentes partes de una imagen son las usadas para un preciso reconocimiento.
  Estos modelos se basan en la idea original de *Fischler* y *Elschlager* [21] del uso de posiciones relativas.

- *Redes Neuronales*
  Recientemente, métodos de *deep learning* como *Redes Neuronales Recurrentes*

(RNNs)[3] o *Redes Neuronales Convolucionales* (CNNs)[4] se han estado usando para el reconocimiento de imágenes y generar pequeños descriptores de texto. Cuando se usan para el reconocimiento de imágenes, estos métodos se componen de múltiples capas compuestas por pequeñas colecciones de *neuronas*, que analizan pequeñas partes de las imágenes.

El análisis de los datos de redes sociales es un tema complejo, debido a las estructuras que se forman entre ellos. Para conseguir optimizar los cálculos, así como estudiar el comportamiento de ciertos componentes de la estructura, se ha realizado una de las teorías más destacas de este ámbito, la conocida *Teoría de Gráfos*. En la siguiente sección se describirán, brevemente, ciertos puntos claves de dicha teoría.

2.4 Teoría de Grafos

Una de las más importantes teorías que se aplica en este campo, es la conocida *Teoría de Grafos* [22]. Esta teoría estudia las propiedades de los grafos y su comportamiento y nació en el S. XVIII con el problema de *los puentes de Könisberg*. Esta teoría ha servido de inspiración para la creación del concepto de red social. Este concepto sustituye los nodos de los grafos por actores y verifica la importancia de cada actor dentro de la red. Estos cambios permiten cuantificar y obtener relaciones complejas dadas en el grafo, como por ejemplo obtener relaciones de poder entre diferentes actores al identificar vínculos (aristas), dirección e intensidades.
Como se intuye, todo esto está muy relacionado con la concepción que se le puede dar a una red social, como puede ser *Instagram*. En ella, y pensándolo en forma de grafo, los nodos sería cada uno de los usuarios y las aristas las relaciones entre cada uno de ellos. Esto provoca que se generen diferentes multigrafos completos [23], dependiendo de las conexiones entre los usuarios, y se formen islas entre ellos, tal y como se puede observar en la figura 2.5.

[3]Del Inglés, Recurrent Neural Networks
[4]Del Inglés, Convolutional Neuronal Networks

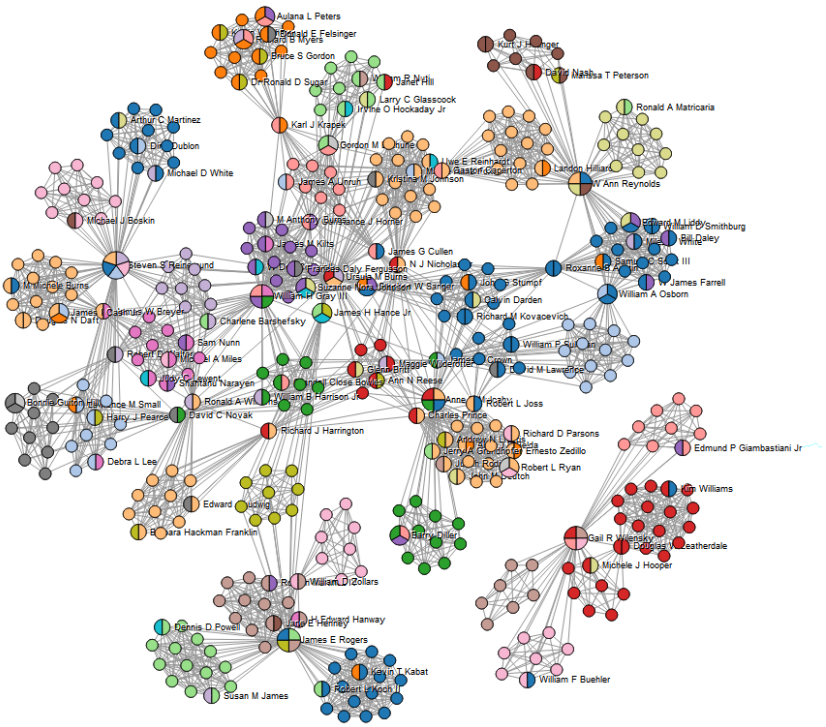

Fig. 2.5: Representación de una red social

Como se ha mencionado anteriormente, a partir de esta teoría se han elaborado un sinfín de algoritmos y métricas que permiten obtener información acerca de los grafos que se generan en las redes sociales.
Algunas de las métricas más destacadas, son:

1. Betweenness Centrality [24]
   Indica la frecuencia en la que un nodo, usuario en este caso, actúa como puente a lo largo del camino más corto entre otros dos nodos. Esto, desde el punto de vista de redes sociales, nos permite cuantificar el control de un usuario en la comunicación existente con otros usuarios de la misma red. La idea del algoritmo es que si se elijen dos nodos al azar, y luego también al azar uno de los posible caminos más cortos entre ellos, entonces los nodos con más intermediación serán los que aparezcan con mayor probabilidad dentro del camino.

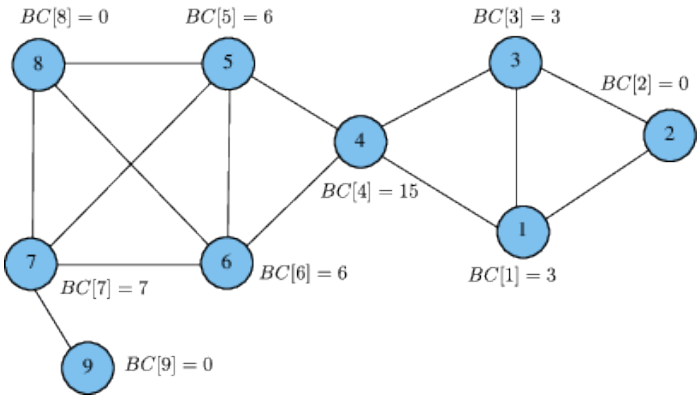

Fig. 2.6: Representación del Betweenness Centrality en un grafo

2. Degree Centrality [25]

   Esta métrica permite calcular el peso de cada nodo, o usuario, dentro de la red a la que pertenecen. Con esto podemos saber cuales son los usuarios más influyentes dentro de un entorno, y los más populares. Con esta métrica, además, podemos calcular la centralidad del grado tanto de entrada, como de salida. Es decir, el peso del usuario según el número de contactos que tiene, o según el número de usuarios que lo tienen entre sus contactos. En esta métrica también influyen, por supuesto, los contactos de tus contactos. Por tanto, si tus contactos también tienen un grado importante en el grafo, esto hará que aumente más el tuyo.

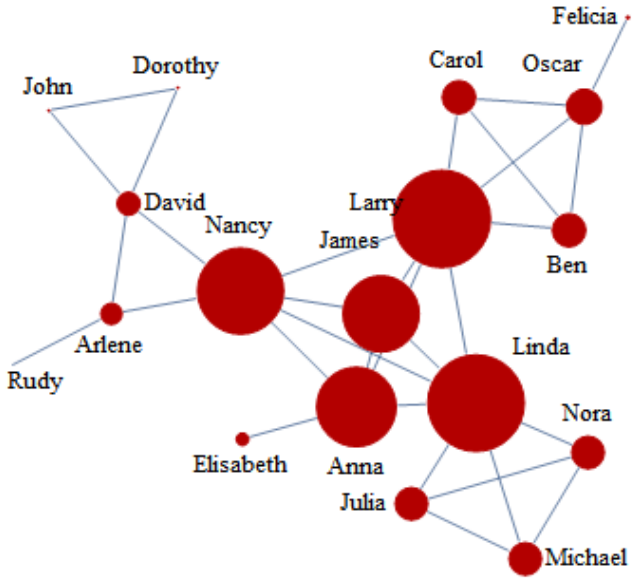

Fig. 2.7: Representación del Degree Centrality en un grafo

3. Closeness Centrality

   Una medida fundamental de la centralidad de los nodos es *Closeness Centrality* [24]. Esta métrica se calcula mediante la suma de las distancias al resto de nodos del grafo, por lo que indica cuanto de cerca está un nodo del grafo del resto. En la imagen 2.8 se puede comprobar.

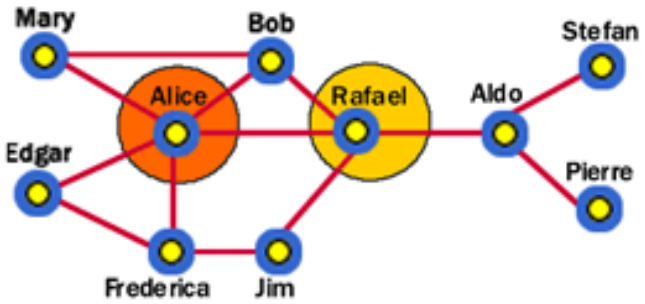

Fig. 2.8: Representación del Closeness Centrality en un grafo

Capítulo 3

Estado del Arte

El análisis de las redes sociales (*Social Network Analysis, SNA*) está en pleno crecimiento [26]. Desde hace poco tiempo muchos investigadores y empresas se han percatado de la cantidad de información que se estaba perdiendo en las redes sociales. Esta información proviene tanto de los datos que los usuarios suben, ya sean imágenes, vídeos, audio, o texto, como de los metadatos que se generan en cada una de las publicaciones. Estos metadatos hacen referencia a la hora de subida, desde dónde se ha realizado, los amigos a los que ha interesado, etc.. Y aun que pueda parecer una obviedad, hasta hace poco tiempo, estos datos se estaban dejando pasar como si no fuesen realmente relevantes [27].

3.1 Sistemas existentes

Existen ciertos sistemas de análisis de redes sociales que proporcionan información a los usuarios a partir de diferentes métricas.
Sistemas como *SumAll*[1], *Twitonomy*[2] o *Brandchats*[3] permiten obtener información a partir de los metadatos y el contenido que los usuarios suben a las redes sociales. En todos estos casos, los sistemas se centran en el análisis de la red social *Twitter*, y para ello hacen uso de algoritmos de análisis semántico, geolocalización o sentimiento así como diferentes técnicas de visualización de datos. Estos sistemas han permitido analizar muchas campañas en *Twitter* con mucho éxito, llegando a analizar millones de datos en campañas como las creadas en la final de la *Champions League*.

[1] `https://sumall.com/`
[2] `https://www.twitonomy.com/`
[3] `http://www.brandchats.com/`

A pesar de esto, ninguno de estos sistemas se centra en el análisis de imágenes de las obtenidas en *Instagram*. Tal y como se comentó anteriormente, esta red social es una de las que más ha crecido en los últimos años y en la que se está generando mayor tráfico de información. Por tanto, estamos ante una gran oportunidad ya que como se pudo observar, son muchas las empresas interesadas en el análisis para posteriores campañas de marketing.

3.2 Aplicaciones prácticas

El análisis de las redes sociales se está usando de manera muy extensa y en muchos aspectos diferentes de aplicaciones y disciplinas en los últimos años. La gran parte de las aplicaciones de análisis incluyen predicción de datos y minería, modelado de la propagación de los datos, análisis del comportamiento de los usuarios, interacción basada en la geolocalización, sistemas de recomendación, etc...[28].
En el sector privado, las empresas realizan el análisis de redes sociales con el fin de realizar campañas u obtener *feedback* de los usuarios en las redes.
El análisis de las redes sociales también se empieza a usar en el ámbito de la inteligencia de estado, espionajes, y el cumplimiento de la ley. Estas técnicas permiten que los analistas puedan descubrir organizaciones clandestinas, criminales o abusos. La Agencia Nacional de Seguridad de EEUU (*National Security Agency, NSA*) utiliza sus programas de vigilancia electrónica para generar los datos que necesitan con el fin de mejorar sus análisis, como puede ser el terrorista. La NSA ha estado llevando a cabo análisis de redes sociales en registros detallados de llamadas (CDR), también conocidos como metadatos, poco después de los ataques del 11 de septiembre [29].

Como se puede observar, el análisis de redes sociales tiene un amplio espectro que cubrir. Hoy en día son pocos los analistas que se centran en la red social *Instagram*, y menos aún, los que obtienen información acerca de las fotos de los usuarios. Es por eso por lo que todavía queda mucho camino por andar en este ámbito, y la principal razón de este proyecto.

Capítulo 4

Especificación

En este capítulo se describirán detalladamente los requisitos del sistema, tanto funcionales como no funcionales, y se hará una aproximación a los diferentes *workflows* que se han definido.
Antes de comenzar con esto, es necesario definir algunos conceptos clave para comprender el funcionamiento del sistema.

- **Evento**
  El concepto de Evento se entiende como un hecho destacable que ocurre en las redes sociales y del cual se quiere obtener información. Los eventos tienen una fecha tanto de inicio como de fin, y estarán compuestos por diversos canales.

- **Canal**
  Los canales son los que permiten definir un Evento. Cada canal se centrará en obtener datos de un hecho concreto en *Instagram* y se pueden definir tanto a partir de *hashtags* como de *geolocalizaciones*.

- **Subscripción**
  Las subscripciones se refieren a aquellos canales que están en un evento todavía activo y reciben notificaciones de *Instagram*. Cuando se crea un canal, se edita, o se elimina, el sistema gestiona la subscripción de ese canal en la red social, para recibir los datos en tiempo real.

4.1 Requisitos Funcionales

Los requisitos funcionales definen la naturaleza del funcionamiento del sistema, expresando así cómo interacciona dicho sistema con su entorno y cuales van a ser sus estados y funcionamientos. A continuación se lista y describen cada uno de los requisitos que se han definido para este proyecto:

1. **Alta Usuario**
   El sistema debe permitir la creación de nuevos usuarios, solicitando una serie de datos básicos y contemplando que tanto el *email* como el *nickname* no se puedan repetir en el sistema.

2. **Login**
   Se debe permitir a los usuarios registrados acceder al sistema mediante un formulario donde se soliciten las credenciales introducidas en el alta.

3. **Logout**
   El usuario debe poder cerrar la sesión del sistema en caso de que la tenga abierta.

4. **Nuevo Evento**
   Los usuarios con sesión iniciada deben poder crear eventos en el sistema, con el fin de iniciar una nueva campaña de recolección de datos en *Instagram*.

5. **Editar Evento**
   El sistema debe permitir editar los eventos definidos por el usuario cuya sesión está iniciada.

6. **Eliminar Evento**
   El sistema debe permitir editar los eventos definidos por el usuario cuya sesión está iniciada.

7. **Nuevo Canal**
   Los usuarios del sistema deben poder crear canales asociados a alguno de sus eventos. Estos canales deben definirse sobre un *hashtag* o una *geolocalización*. Una vez se crea un canal, el sistema gestiona una nueva subscripción.

8. **Editar Canal**
   Al igual que con los Eventos, los usuarios podrán modificar los canales que tengan asociados a cada uno de sus eventos. Cuando se produce una modificación, el sistema también modifica la subscripción del canal en *Instagram*.

9. **Eliminar Canal**
   El sistema permite que el usuario con sesión iniciada elimine los canales asociados a su cuenta, y así eliminar las subscripciones de *Instagram*.

10. **Calcular rendimiento del Evento**
    El sistema, a partir de los canales, va almacenando información sobre las actualizaciones que se producen en *Instagram*. Esto permite que los usuarios puedan ver diferentes gráficas y comprobar el rendimiento que están teniendo sus canales y si realmente se está produciendo tráfico en dichas subscripciones.

11. **Calcular mapa de Voronoi del Evento**
    Al igual que en el anterior, al usuario le interesa conocer dónde se produce el tráfico de sus canales. Es por eso por lo que se ha generado un mapa de *Voronoi* sobre los datos *geoposicionados* con el cual el usuario podrá hacerse una idea de dónde tienen más influencia los canales definidos.

4.2 Requisitos no Funcionales

Los requisitos no funcionales definen una serie de restricciones sobre el espacio de soluciones de un problema. Es decir, presentan criterios que permiten valorar el proyecto a partir de sus características de funcionamiento. A continuación se definen los requisitos no funcionales que se han tenido en cuenta en este proyecto:

1. **Escalabilidad**
   El sistema que se ha desarrollado debe ser totalmente escalable, es decir, que se puedan manejar una cantidad de usuarios mucho mayor a la actual, sin perder calidad o eficiencia en el sistema. Esto, por consiguiente, trae asociado un aumento de los recursos *hardware* al sistema.

2. **Eficiencia**
   Debido al contenido con el que trata el sistema, es importante que el tiempo de reacción sea bastante bajo de cara a realizar los cálculos y mostrarle datos al usuario.

3. **Tiempo Real**
   En este tipo de sistemas es importante que los datos que se muestran al usuario puedan ir cambiando con el paso del tiempo, ya que son datos que se actualizar cada pocos segundos. El usuario tiene que poder observar estos cambios en tiempo real.

4. **Alta disponibilidad**
   Este sistema tiene que estar disponible para procesar los datos que recibe de

Instagram con los canales definidos por los usuarios. Por tanto, aun que no haya sesiones abiertas por los usuarios, es importante que el sistema se mantenga estable 24/7.

4.3 Workflow

En esta sección se presentaran diversos diagramas en los que se podrá apreciar el flujo que sigue el sistema en algunos de los escenarios más destacables. Se mostrará el flujo del sistema tanto para los casos de interacción del usuario con el sistema, como para los casos en los que se gestionan y se reciben actualizaciones de las subscripciones en *Instagram*.

1. **Visualizar evento**
   En esta acción, el usuario solicita visualizar la información de un evento que tiene creado. En esta vista el sistema le muestra al usuario las últimas imágenes recibidas, el rendimiento que tiene el evento a partir de sus canales y el mapa de *Voronoi* con la distribución de los datos recibidos. Para esta acción, el flujo de ejecución del sistema será similar al que se puede apreciar en la imagen 4.1.

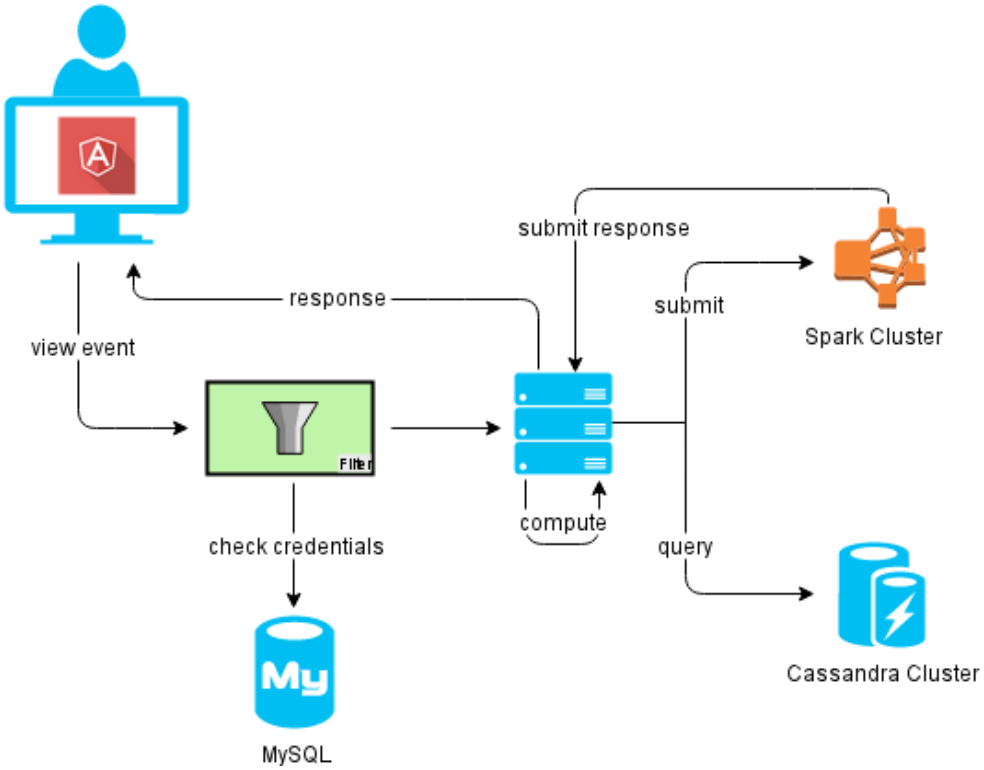

Fig. 4.1: Flujo de ejecución para visualizar un evento

2. **Actualización Instagram**

   En esta acción, el sistema recibe una nueva actualización de algún canal subscrito a *Instagram*. Una vez recibida, se lee el mensaje -*JSON* [1]- y se tiene que hacer una petición a la *API*[2] *REST*[3] de *Instagram* con el identificador correspondiente. A continuación, en la imagen 4.2, se puede ver el flujo que sigue el sistema.

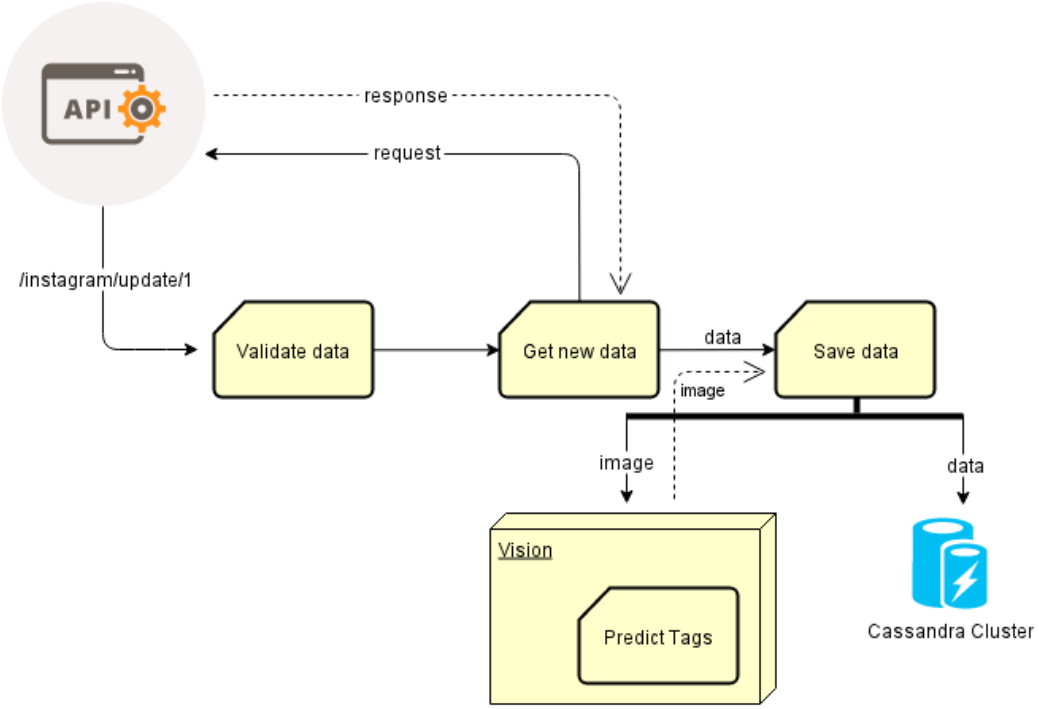

Fig. 4.2: Flujo de ejecución para una actualización de Instagram

4.4 Arquitectura

La arquitectura diseñada para este sistema se ha ido formando con el paso del tiempo, con el fin de adaptarla lo mejor posible a todos los módulos. Esto hace que la arquitectura sea modular y re-adaptable en cualquier momento. A continuación se muestra la imagen 4.3 en la que se puede ver claramente el diseño, y seguidamente una explicación de todas las capas.

Como se puede ver en la imagen 4.3, el sistema se divide en tres capas:

[1] Javascript Object Notation [30]
[2] Application Programming Interface
[3] Representational Estate Transfer [31]

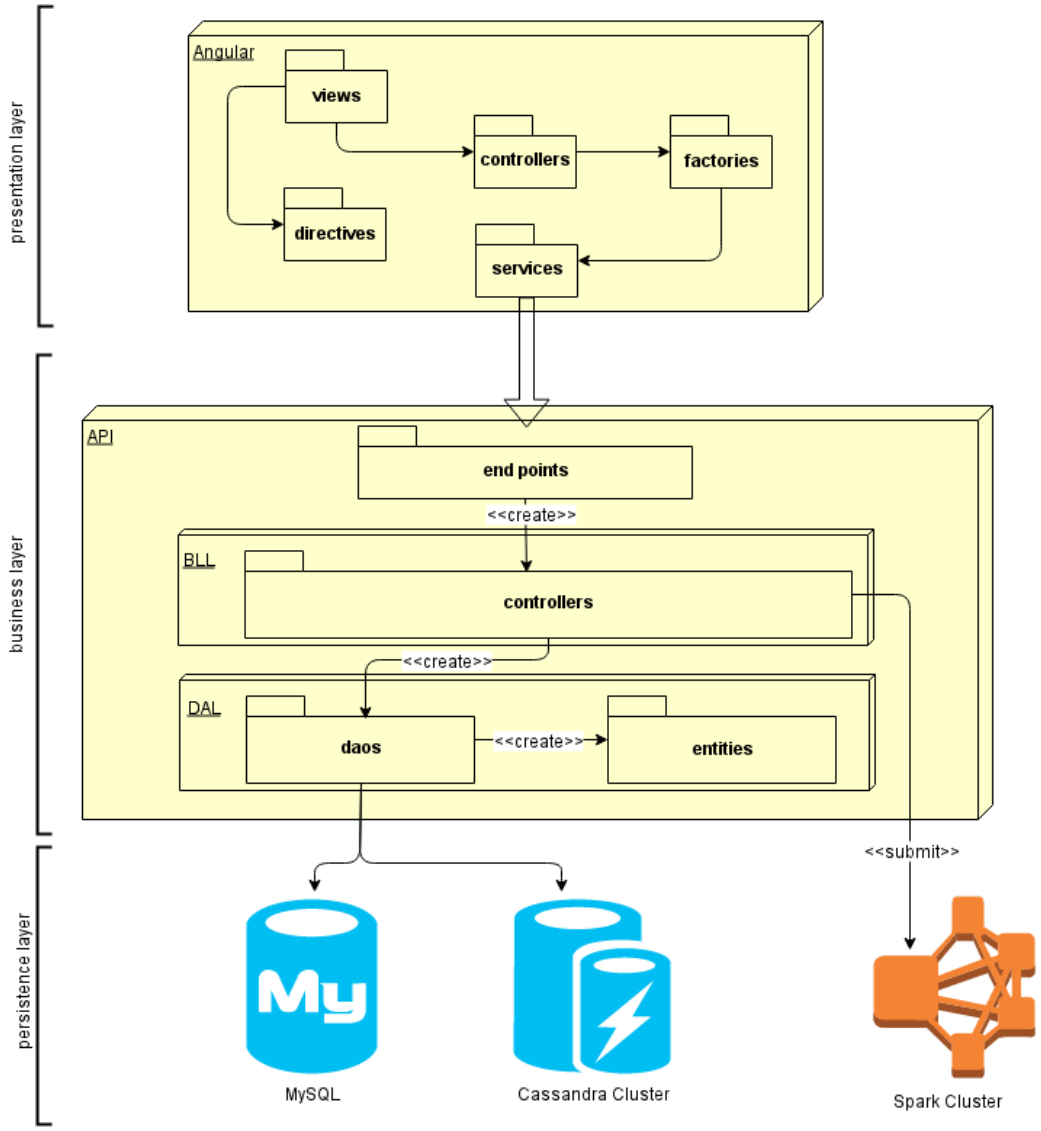

Fig. 4.3: Arquitectura del sistema

- **Capa de presentación**

  Esta capa engloba todo lo referente a la visualización en la parte del cliente. Es la capa encargada de gestionar las peticiones que el cliente realiza en la aplicación web, y solicitar los datos necesarios a la capa de negocio. En esta capa se reciben los datos y se formatean para mostrárselos al usuario mediante gráficas u otros modos. Como se observa en la imagen 4.3, esta capa está dividida en diversos paquetes con el fin de agrupar y modularizar lo máximo posible la aplicación.

- **Capa de negocio**

  Esta capa es la encargada de gestionar la lógica de negocio. Dispone una *API*

REST con la que permite que se comunique la capa superior. A partir de cada petición *HTTP* que recibe, la gestiona y lleva a cabo toda la lógica para devolver una respuesta a la petición. Esta es la capa principal de la aplicación, desde donde se guardan y obtienen los datos de *Cassandra* y desde donde se ejecutan los *scripts* de *Apache Spark* en el clúster. Esta capa, a su vez, se divide en tres subcapas:

1. *Endpoints* En una arquitectura *REST*, los *Endpoints* son las *url* disponibles para que se hagan las peticiones. Esta capa es la que recibe las peticiones *HTTP* y genera la lógica de negocio pertinente.
2. *BLL* Es la capa encargada de crear la lógica de negocio para la aplicación. Esta capa expone las acciones a la capa superior, y se encargar de comunicarse con la capa inferior para obtener datos y procesarlos.
3. *DAL* Esta capa está dividida en dos módulos. El primero, los *DAOs*[4], son los que se encargan de comunicarse con la capa inferior y superior, y realizar las transacciones a las bases de datos. Por otro lado, el módulo de *Entities* -las entidades del sistema- son las que permiten hace un *ORM*[5], y convertir los datos almacenados en las bases de datos en objetos de nuestra aplicación.

- **Capa de persistencia o datos**
  Por último, esta capa engloba todo lo referente al almacenamiento de datos. Tanto la base de datos *MySQL*, utilizada para guardar los datos de la aplicación, como el clúster de *Cassandra*, usado para almacenar los datos de *Instagram*, son accesibles desde la capa superior de los *DAOs* tanto para guardar como para obtener datos. En esta capa se ha añadido también el clúster de *Apache Spark* ya que, a pesar de no almacenar datos, sí que consulta datos de la base de datos, y nos proporcionará unos datos en cada llamada.

A continuación se muestra una imagen con el diseño de la arquitectura *hardware* del sistema. Como se puede ver en la imagen 4.4, se han creado varios módulos que gestionan las diferentes partes del sistema. Esto permite que el sistema sea ampliamente escalable y pueda gestionar mucho mejor más carga.

Para este sistema se han utilizado 5 servidores, donde 4 de los cuales han sido destinados a crear un clúster tanto de *Apache Spark* como de *Apache Cassandra*. Este

[4]Del inglés, *Data Access Object*
[5]Del inglés, *Object Relational Mapping*

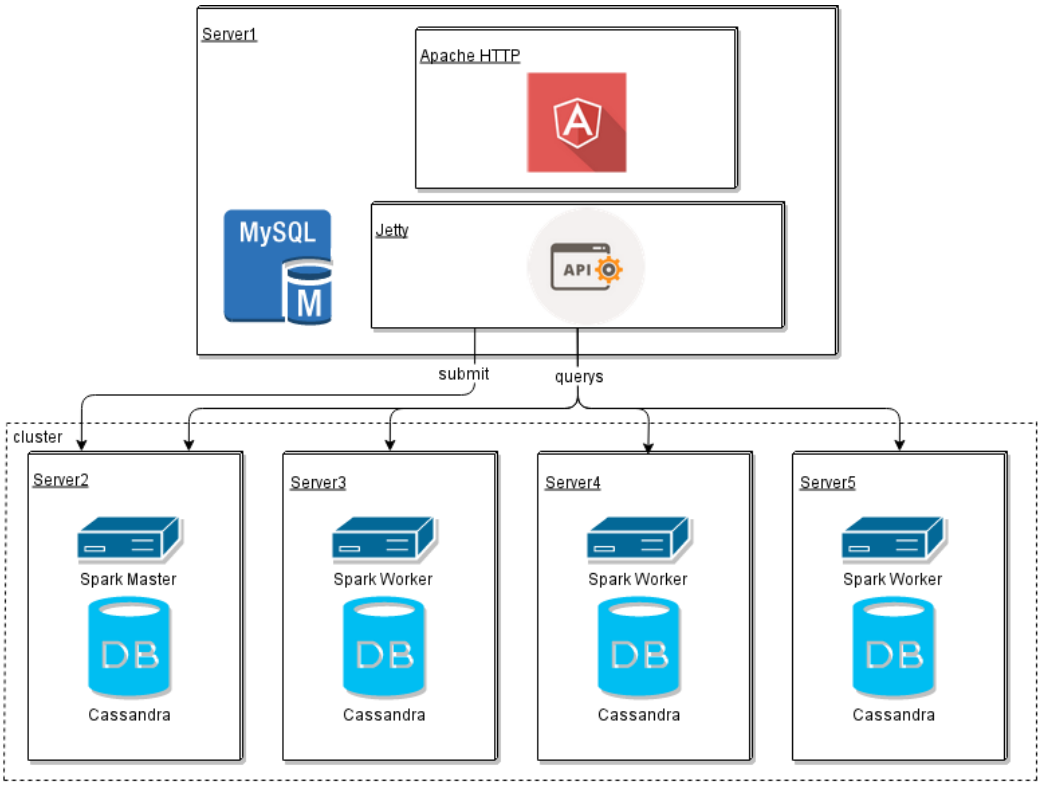

Fig. 4.4: Arquitectura del sistema

módulo, por el momento, no soporta una gran carga pero se ha configurado con vistas a que se amplíe en cualquier momento y no haga falta reconfigurarlo, sino solamente configurar los nuevos equipos.

En cuanto al *Server1*, es el encargado de ejecutar tanto el servidor *Jetty* ,que gestionará el *API REST*, como el servidor *Apache*, que se encargará de ejecutar la capa de presentación del sistema, así como una instancia de la base de datos *MySQL*. Todo esto puede ser exportado a nuevos equipos, ya que se ha diseñado con ese fin.

Por falta de recursos, en este momento se ha montado de esta forma, pero se prevé una ampliación de los recursos *hardware*.

Capítulo 5

Diseño

Este capítulo describirá la solución propuesta al problema descrito en la sección anterior. Se mostrarán diversos diagramas en los que se podrán ver las soluciones tomada para los diferentes problemas planteados, así como las optimizaciones y cambios que se han realizado con propósito de mejorar el sistema.

5.1 Frameworks

Para desarrollar este sistema, se han tomado ciertas decisiones previas a la realización del diseño. Una de ellas ha sido los *frameworks* o librerías que se iban a utilizar, así como los diferentes lenguajes de programación.
A continuación se describirán los lenguajes y *frameworks* asociados a cada una de las capas vistas en la imagen 4.3, y el motivo de su elección.

- **Capa de presentación**
  Tal y como se describió en la sección 4.4, esta capa es la encargada de gestionar los datos y mostrárselos al usuario en tiempo real. Para llevar a cabo esto, se ha decidido usar el *framework AngularJS*[1]. La elección de este entorno de trabajo fue debido a varios motivos:

  1. *Productividad* Este *framework*, desarrollado por Google, trae consigo un gran aumento de la productividad debido a la arquitectura interna que define. La arquitectura *MVC*[2] permite, a pesar de tener una elevada curva de aprendizaje, elevar la productividad y mantener una estructura limpia y clara.

[1]Framework MVVM Javascript `https://angularjs.org/`
[2]En inglés, Model-View-Controller `https://developer.chrome.com/apps/app_frameworks`

2. *Eficiencia* El lenguaje *Javascript*, junto con *Typescript* permiten una gran eficiencia y rapidez en el navegador del cliente. *AngularJS* está basado en el lenguaje *Typescript* lo que hace que todo el núcleo del sistema haya sido optimizado a grandes niveles.
3. *Open Source* Poder usar un proyecto *Open Source*[3] del nivel de *AngularJS* es una gran alternativa frente a muchos otros. La popularidad de este *framework* está haciendo que la comunidad esté dedicando un gran esfuerzo a mantenerlo y mejorarlo día a día. Por tanto, se ha escogido también con el fin de aumentar y fomentar el uso de este tipo de proyectos.

A mayores de esto, y con el propósito de crear una serie de gráficas para mostrarle al usuario, se han usado librerías como *HighCharts*[4] y *Chart.js*[5], que permiten crear un gran número de gráficos dinámicos y de manera sencilla. También se ha hecho uso de la librería *Bootstrap*[6] con el fin de obtener un resultado visual agradable y de gran usabilidad para el usuario final. Por último, y como es de esperar, el lenguaje de programación usado ha sido *Javascript*, junto con otros de modelado web como *HTML*, *CSS*, etc..

- **Capa de negocio**

  Como vimos en la sección 4.4, ésta es la capa encargada de toda la lógica de negocio del sistema. Esta capa recibe las peticiones de la capa de presentación, y ejecuta la lógica correspondiente para devolver una respuesta. Para llevar esto a cabo, se ha usado el lenguaje *Java* y se ha creado una *API REST* con la que se exponen los métodos disponibles del sistema. Para realizar esto se ha hecho uso del *framework Jersey*[7], el cual permite realizar una *API REST* de una manera muy sencilla y muy robusta. Uno de los motivos de esta decisión ha sido el desacoplo que ofrece con respecto a la capa superior. Esta *API* permite que la capa de negocio se pueda ejecutar y escalar sin problema y no sea dependiente de la capa superior. Además, permite que otros clientes puedan ejecutar y obtener datos de nuestro sistema sin tener que usar la aplicación Web, por tanto también nos abre otro pequeño mercado.

  A mayores de este *framework*, utilizado para el módulo de los *endpoints*, también se han usado otros para los módulos inferiores. En cuanto al módulo de *DAOs*,

[3] `https://opensource.org/`
[4] `http://www.highcharts.com/`
[5] `http://www.chartjs.org/`
[6] `http://getbootstrap.com/`
[7] `https://jersey.java.net/`

se ha usado el *framework Hibernate*[8], para el acceso a la base de datos *MySQL*. Este *framework* permite gestionar las transacciones y generar *queries* para obtener y almacenar datos en la base de datos.

- **Capa de datos**
  En esta capa no se han utilizado *frameworks*. Para crear esta capa, se ha instalado tanto *MySQL* como dos clústers de *Cassandra* y *Apache Spark*, en los que se entrará en más detalle en secciones posteriores.

Una vez vistos los *frameworks* elegidos para las diferentes capas del sistema, entraremos en detalle en cada una de ellas. En las siguientes secciones se explicarán las diferentes capas ya entrando en el detalle de su arquitectura y mediante diversos diagramas, se podrá entender el funcionamiento y dichas decisiones.

5.2 Capa de presentación

Tal y como se ha comentado anteriormente, esta es la capa encargada de procesar los eventos del usuario en el navegador y mostrarle los datos del sistema. Para ello, se ha decidido hacer uso del *framework AngularJS* junto con los lenguajes del *stack* web. Este *framework*, tal y como se ha visto en secciones anteriores, impone una arquitectura *MVVC*. Esta arquitectura, representada en la imagen 5.1, deriva del patrón de diseño *Presentation Model* expuesto por Martin Fowler en 2004 [32].

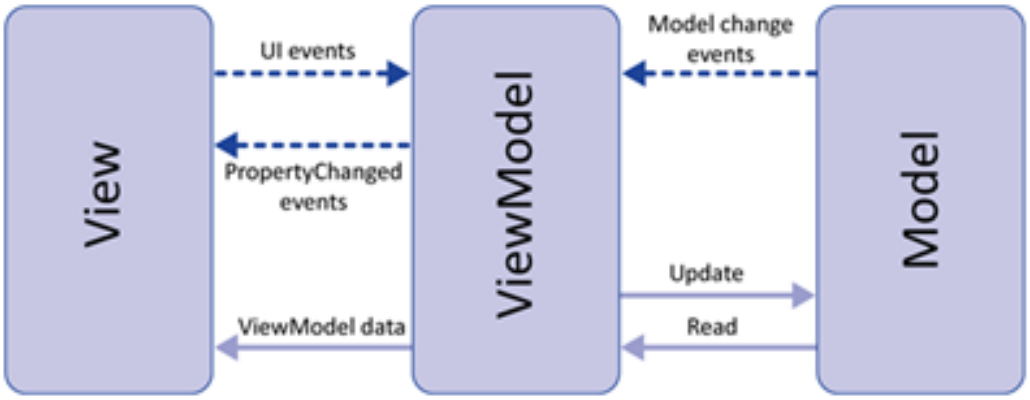

Fig. 5.1: Arquitectura AngularJS

Este diseño permite una gran separación de los componentes, ya que toda la parte visual - *view* - está completamente separada de los datos de la aplicación. Para

[8] `http://hibernate.org/`

comprender un poco mejor el funcionamiento de este diseño, se detallarán las partes implicadas:

- **View**
  Igual que en otros diseños, como *MVC* o *MVP*, esta capa es la encargada de la interfaz de usuario. En ella solamente se alberga código de maquetación y el necesario para responder a los eventos web.

- **ViewModel**
  Esta capa es una abstracción de la vista en la cual se exponen métodos y eventos. Es la encargada de hacerle llegar los datos de cada *Model*, y gestionar los eventos y los cambios de cada uno de ellos. El *ViewModel* se puede describir como un estado de los datos en el *Model*.

- **Model**
  El modelo es la representación de los datos de la aplicación. En este caso, esta capa será la que proporcione los métodos *privados* para acceder a la lógica de negocio del *backend*.

Dada esta imposición por parte del *framework* utilizado, la arquitectura diseñada para el sistema es de unas características muy similares, pero aprovechando la flexibilidad de *Angular JS* se han realizado algunas mejoras. En la imagen 5.2 se puede ver el resultado final del diseño creado para el sistema.

En este diseño, se ha seguido el patrón impuesto por *AngularJS* pero con ciertas ampliaciones, ya que tanto la capa *View* como *Model* se han ampliado para desacoplar ciertos comportamientos. En cuanto a la capa de presentación, se han usado *directives*[9], las cuales permiten crear código *HTML* y ser usado en diferentes páginas, evitando así la repetición. Estas directivas se inyectan en las diferentes vistas, dependiendo de si serán usadas o no, y se introduce una llamada en el código, que será luego compilado por el *core Angular JS*. En la imagen 5.3 se puede ver un fragmento de código donde se utiliza una directiva, que luego representará un *datetimepicker*[10].

En cuanto a la capa de *Models*, se ha optado por una división en la cual se usan dos elementos fundamentales en *Angular JS*, las *factories*[11] y los *services*[12]. Las *factorias* nos ayudarán a representar los objetos del sistema. Será donde se definan las operaciones que cada uno de ellos puede realizar, y estarán expuestas a los *controllers*

[9] `https://docs.angularjs.org/guide/directive`
[10] Elemento HTML con el que seleccionar fecha, y hora en un calendario.
[11] `https://docs.angularjs.org/guide/providers`
[12] `https://docs.angularjs.org/guide/services`

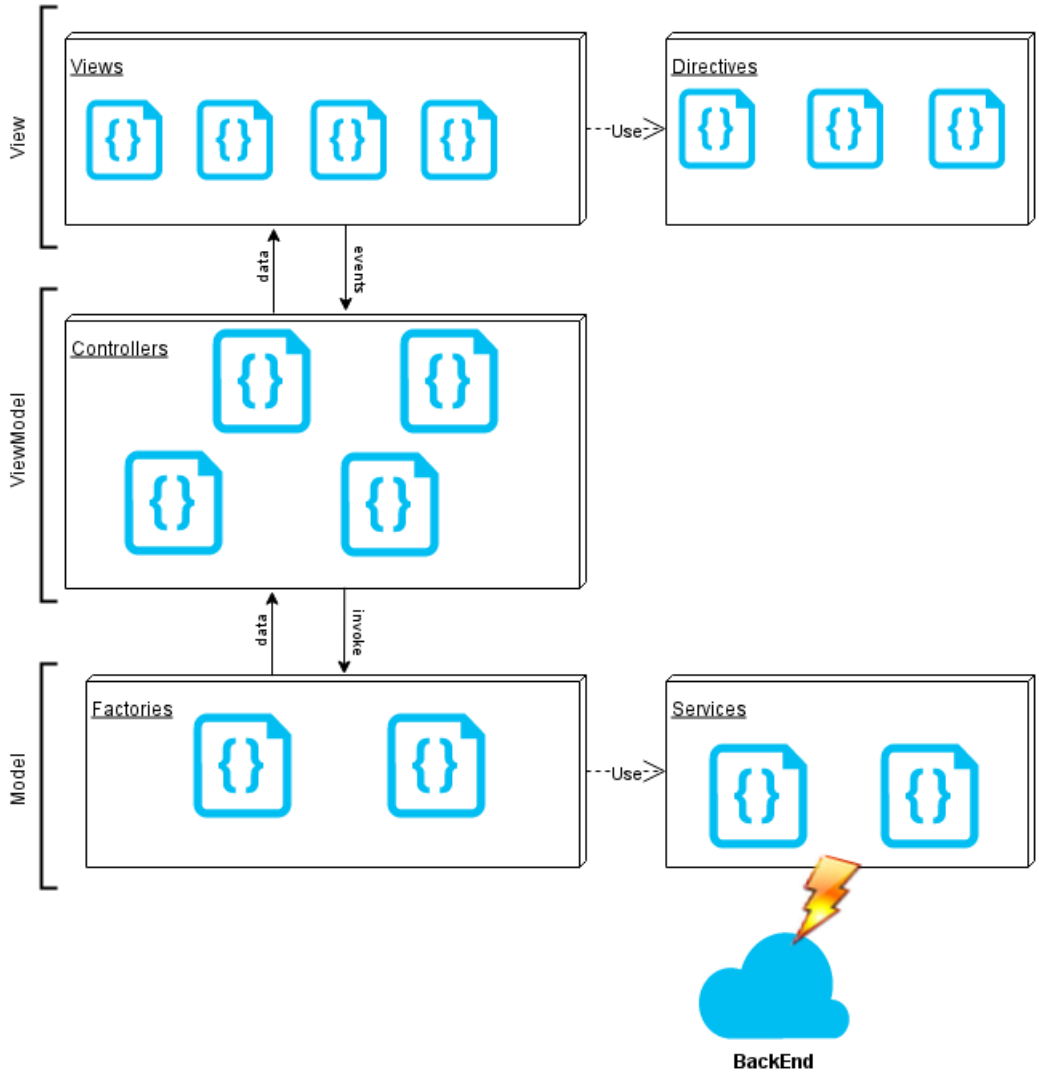

Fig. 5.2: Arquitectura final Angular JS

```
<ul class="dropdown-menu" role="menu" aria-labelledby="dLabel">
    <datetimepicker data-ng-model="selectedEvent.formated_start_date"
                    data-datetimepicker-config="{ dropdownSelector: '#dropdown-startdate' }"
                    data-on-set-time="convertTime(newDate, 'start_date')"></datetimepicker>
</ul>
```

Fig. 5.3: Ejemplo de uso de directivas

definidos en la capa superior. A su vez, estas son las encargadas de realizar las llamadas a los métodos pertinentes del *service* asociado. Los *services* son los elementos encargados de realizar las peticiones *HTTP* a la *API REST* expuesta en el *backend*. Por tanto, serán los que definan cada uno de los métodos y los parámetros para cada una de las *urls*.

Como se puede esperar, cada *factoría* tiene su *servicio* asociado, ya que están representando al *endpoint* de un objeto del sistema.

5.3 Capa de negocio

La capa de negocio, o de lógica, es la encargada de realizar todas las operaciones que define el sistema. Para ello, se ha dispuesto de una *API REST*, tal y como se ha comentado en la sección 4.4 del capítulo 4, mediante la que se recibirán las peticiones *HTTP* y se procesarán según la lógica pertinente. Para realizar esto, se ha decidido implementar una *API REST* con la cual cualquier cliente pueda realizar peticiones a ella, y consultar sus eventos y canales. Para ello, se ha hecho uso de *Jersey*[13] una librería de *Java* que permite crear *APIs* muy robustas y con gran facilidad. Los *endpoints* que se han definido se han tratado de englobar dependiendo de la funcionalidad dentro del sistema. Para ello se han creado los siguientes *endpoints* en la *API*:

- */channel* Es el encargado de gestionar toda la lógica que concierne a los canales.
- */event* Recibe todas las peticiones de la lógica que implica a los eventos.
- */instagram* Este *endpoint* se encarga de la gestión de las subscripciones de *Instagram*, así como de la gestión del *OAuth*[14].
- */spark* Se encarga de gestionar los *submits* de *Apache Spark* así como de recibir los datos una vez han terminado las ejecuciones en el clúster.
- */user* Gestiona el alta y baja de usuarios, así como el *login* en la aplicación, y los permisos para acceder a los diferentes módulos.

La imagen que se muestra a continuación expone, de manera gráfica, los *endpoints* creados y los métodos en cada uno de ellos.

Como se puede observar, en la imagen 5.4 se muestran los diferentes métodos *HTTP* y las acciones que conllevan en el sistema. A mayores de esto, también se pueden observar diferentes tipos de parámetros, *param* y *?param*, dependiendo si son parámetros de url[15] o parámetros de consulta[16].

[13] `https://jersey.java.net/`
[14] Protocolo que permite flujos simples de autorización entre diferentes sistemas.
[15] Por ejemplo, */event/5*
[16] Por ejemplo, */event?user_id = 5*

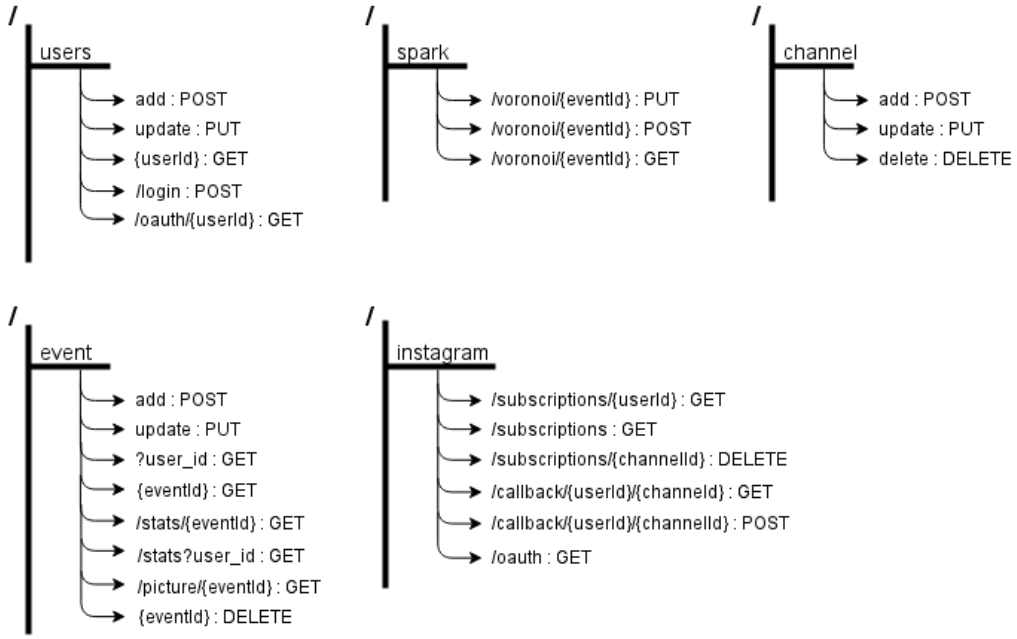

Fig. 5.4: Endpoints de la API REST

Una vez se ha dispuesto diseñado cómo será la *API REST* que expondrá el sistema, es necesario diseñar una arquitectura coherente para ejecutar la lógica de negocio. Para ello se ha creado una diseño con el cual se mantuvieran separadas las diferentes capas del sistema y fuera sencillo desarrollar nuevas funcionalidades, sin tener que modificar cosas ya existentes. El diseño final del sistema es el que se muestra en la imagen 5.5:

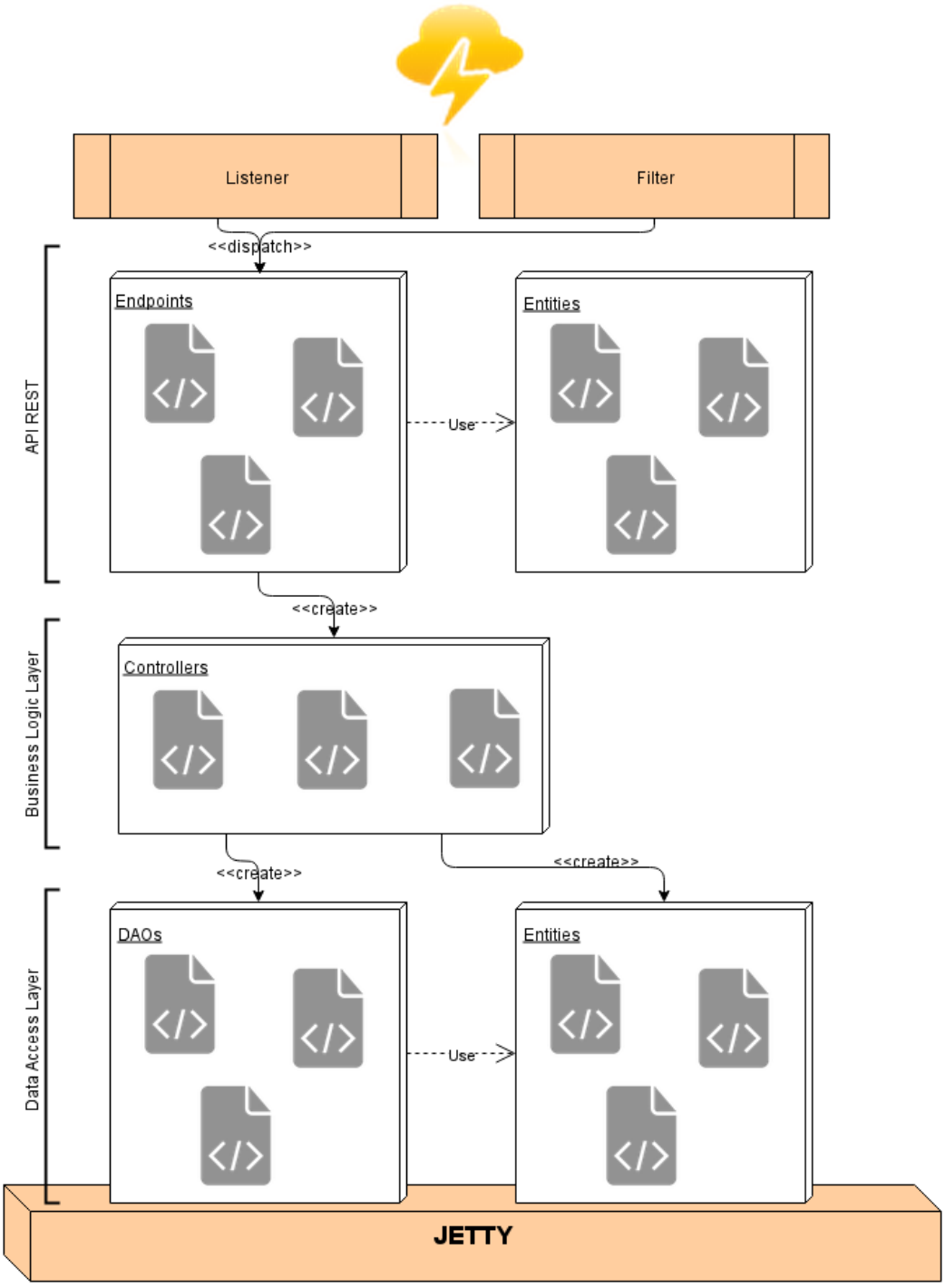

Fig. 5.5: Diseño del backend del sistema

Como se puede observar, el sistema se ejecuta sobre el contenedor *Jetty*[17], el cual proporciona un servidor web con soporte *HTTP/2*, *WebSocket*, etc... Antes de que se ejecuten las acciones del *API REST*, se ejecutan una serie de *listeners*, y *filtros* que

[17] `http://www.eclipse.org/jetty/`

inicializan la sesión y comprueban los permisos del usuario que realiza la petición. En caso de que se tengan permisos, la petición se pasa a la siguiente capa, ejecutando el *endpoint* correspondiente. La capa *DAL*[18] es la que se encarga de gestionar las transacciones en las llamadas a la base de datos.
Con el fin de poder trabajar con objetos, y no con los resultados de las consultas directamente, se han creado entidades que representan los datos almacenados con el fin de realizar un *ORM*[19].
En cuanto a las entidades creadas en la capa de la *API*, es debido a que los datos que se reciben en las peticiones son en formato *JSON*[20], y con el fin de tratarlos como objetos, se ha utilizado la librería *Jackson*[21] la cual convierte los datos recibidos en clases del sistema.

5.4 Capa de datos

Esta capa es la encargada de almacenar los datos de la aplicación. Para ello, y teniendo en cuenta los requisitos, se han creados dos sistemas de almacenamiento. Por una parte se deben guardar los datos del sistema, es decir, los datos de usuario y los datos de los eventos y canales del usuario. Por otra parte, el sistema debe almacenar los datos que *Instagram* retorna de las subscripciones. La solución a esto ha sido crear un sistema de almacenamiento con el *SGBD*[22] *MySQL*[23].
Este gestor permite crear bases de datos relacionales con un coste de procesador bajo y asequible para el sistema. Este será el sistema que almacene y gestione los datos básicos de la aplicación, con el siguiente esquema generado:

[18]Del Inglés, Data Access Object
[19]Del Inglés, Object-Relational Mapping
[20]JavaScript Object Notation
[21]`http://wiki.fasterxml.com/JacksonHome`
[22]Siglas de: Sistema de Gestión de Base de Datos
[23]`https://www.mysql.com/`

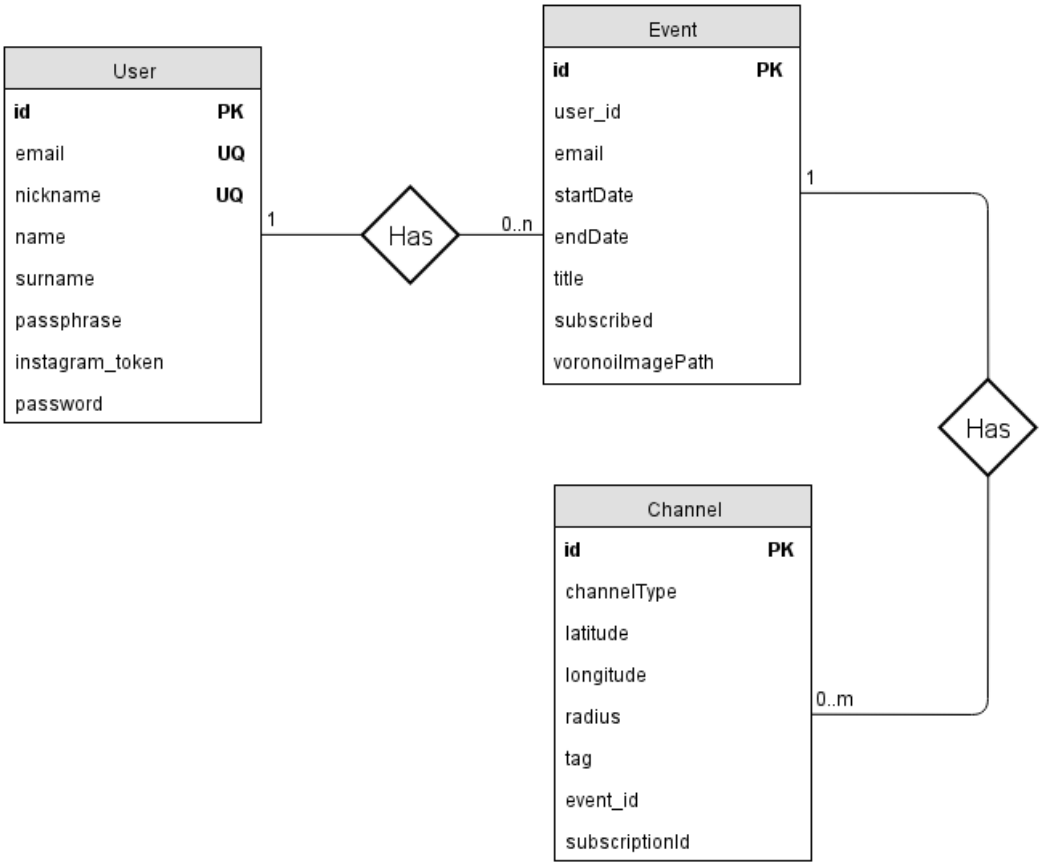

Fig. 5.6: Diseño del backend del sistema

Como se puede observar, se han creado tres tablas que serán las que almacenen los datos de los *User*, *Event* y *Channel*. Las relaciones entre ellas son bastante simples ya que cada *Event* pertenecen a un único *User* y un *User* puede tener múltiples definidos. En cuanto a la relación entre *Event* y *Channel*, es muy similar ya que cada canal pertenece a un único evento, y estos pueden tener múltiples canales definidos.

Por otra parte, el sistema debe almacenar los datos de las subscripciones de *Instagram* para posteriormente procesarlas. Para ello, se ha creado un clúster de *Cassandra*, tal y como se ha visto en la sección 4.4 del capítulo 4 con el fin de que gestione todo el procesado. El clúster consta de 4 equipos, y *Cassandra* se encargará de repartir los datos entre los diferentes nodos y gestionar las *queries* que se realicen. El modelado de los datos de *Cassandra* es el que se muestra en la imagen 5.7.

Cassandra es una base de datos *NoSQL* y esto hace que las relaciones no funcionen exactamente igual que las explicadas con *MySQL*. Para esto, ha sido necesario definir las siguientes tablas, y los tipos compuestos.

Cassandra ofrece una serie de tipos de datos básicos que se pueden usar en la definición

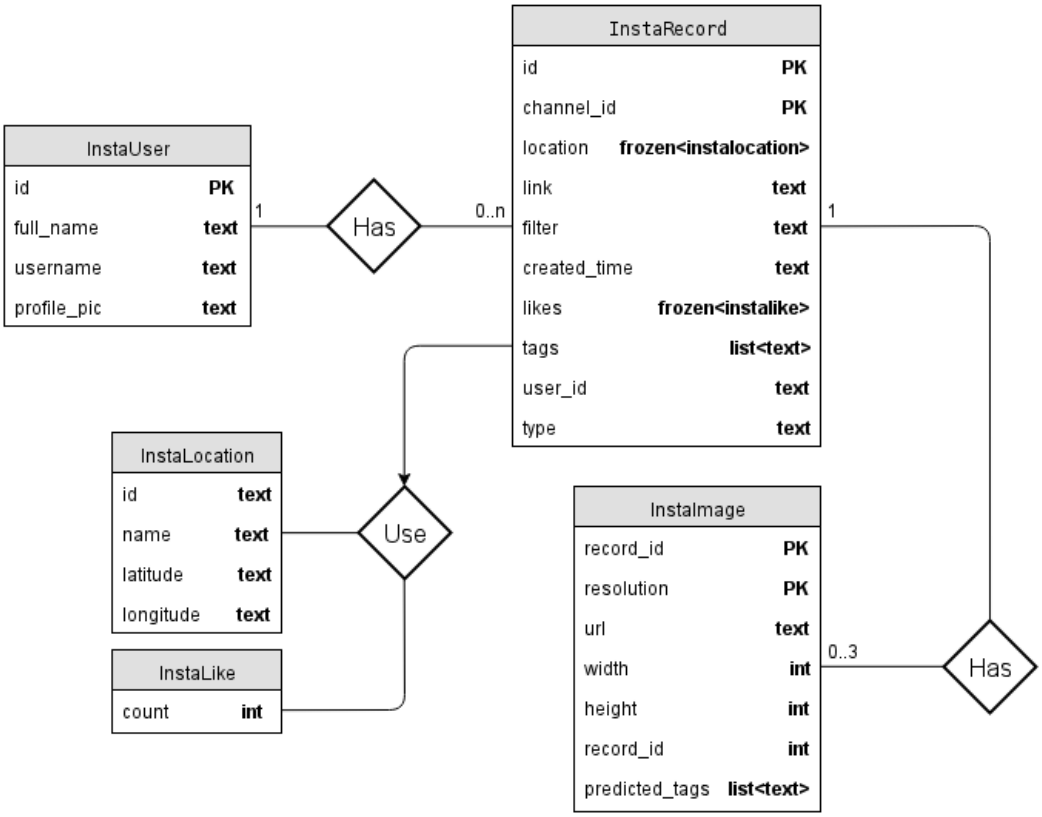

Fig. 5.7: Diseño del backend del sistema

de las columnas de una tabla, pero muchas veces es necesario crear tipos complejos con el fin de optimizar la tabla. En este caso, ha sido necesario definir un tipo *InstaLocation*, que será el que almacene los datos de la localización de un *InstaRecord*, así como el tipo *InstaLike*, que ahora mismo solo almacena un contador, pero que se puede ampliar para almacenar información acerca del usuario que hizo *like* a una imagen.

Como esta base de datos trabaja con datos en formato no estructurado, como puede ser *JSON*, una representación de la tabla *InstaRecord* en este formato podría ser la siguiente:

```
{
    "id"                : 1,
    "channel_id"        : 1,
    "location"          : {
        "id"         : 1,
        "name"       : "Barcelona",
        "latitude"   : "41.390205",
        "longitude"  : "2.154007"
    },
    "link"              : "www.example.com",
    "filter"            : "Earlybird",
    "created_time"      : "1453223271",
    "likes"             : {
        "count"      : 42
    },
    "tags"              : ["tag", "insta", "image"],
    "user_id"           : "12979420",
    "type"              : "image"
}
```

Listing 1: Ejemplo tabla en formato JSON

Como se puede ver, los tipos de dato que hemos definido son tipos compuestos los cuales albergan más información que un *string*, *int* u otro tipo ya definido por *Cassandra*.

Capítulo 6

Implementación

Esta sección muestra detalles concretos de la implementación llevada a cabo para este sistema. La implementación que se ha realizado es muy similar al análisis expuesto en la sección anterior, ya que no ha sido necesario modificar la arquitectura u otros temas notorios.
Con el fin de que sea más sencillo entender la implementación, este capítulo se dividirá en tres secciones diferentes, correspondientes a cada una de las capas de la aplicación. Cada sección mostrará detalles de implementación en dicha capa.

6.1 Capa de presentación

6.1.1 Librerías externas

En esta capa son importantes los detalles, por ello, se han utilizado diversas librerías que han ayudado a crear una interfaz de usuario completa y con todo nivel de detalle para que resulte lo más sencilla al usuario. Las librerías usadas han sido las siguientes:

- *jQuery*[1]
  Esta librería de *JavaScript* nos permite simplificar la manera de interactuar con el *DOM*[2], gestionar eventos de manera dinámica, uso de Ajax, etc.. y todo esto de una manera mucho más simple que mediante *JavaScript*.

- *Bootstrap*[3]
  Framework creado por *Twitter* que permite crear interfaces web mediante *CSS*[4] y *JavaScript* de una manera muy sencilla y con un gran potencial. Con esta

[1]Más información `https://jquery.com/`
[2]Del Inglés, Data Object Model
[3]Más información `http://getbootstrap.com/`
[4]Del Inglés, Cascading Style Sheets

librería también se ha creado la interfaz *responsive*, es decir, que adaptable a los diferentes tipos de pantalla.

- *Highcharts*[5]
  Con esta librería de *JavaScript* se han creado parte de los gráficos que se muestran en la web. Permite la creación de una gran cantidad de gráficos y todos ellos dinámicos en los que se permite una interacción con el usuario.

- *Charts.js*[6]
  Esta librería está basada en la anterior, pero amplia algunas de las gráficas. Nos ha ayudado a crear ciertos gráficos del sistema de una manera sencilla y elegante.

- *D3.js*[7]
  Esta es la librería de gráficos por excelencia. Permite crear todo tipo de gráficos mediante *HTML*, *SVG* y *CSS*. A pesar de que finalmente se ha decidido no mostrar ciertos gráficos, sí que se han realizado varios de ellos con esta potente librería de *JavaScript*.

- *Moment.js*[8]
  Uno de los grandes problemas de *JavaScript* hasta hace poco tiempo era la manipulación de fechas y datas. Esta librería contiene una gran cantidad de funciones que permiten gestionar las fechas de una manera simple y cómoda.

6.1.2 Diagrama de Componentes

El diagrama de componentes[33] permite mostrar cómo se ha dividido un sistema en componentes, y muestra las dependencias entre cada uno de ellos. Tal y como se muestra en la imagen 6.1, esta capa se ha dividido en diferentes componentes que cooperan entre sí para generar la vista que se mostrará al usuario.

[5]Más información `http://www.highcharts.com/`
[6]Más información `http://www.chartjs.org/`
[7]Más información `http://d3js.org/`
[8]Más información `http://momentjs.com/`

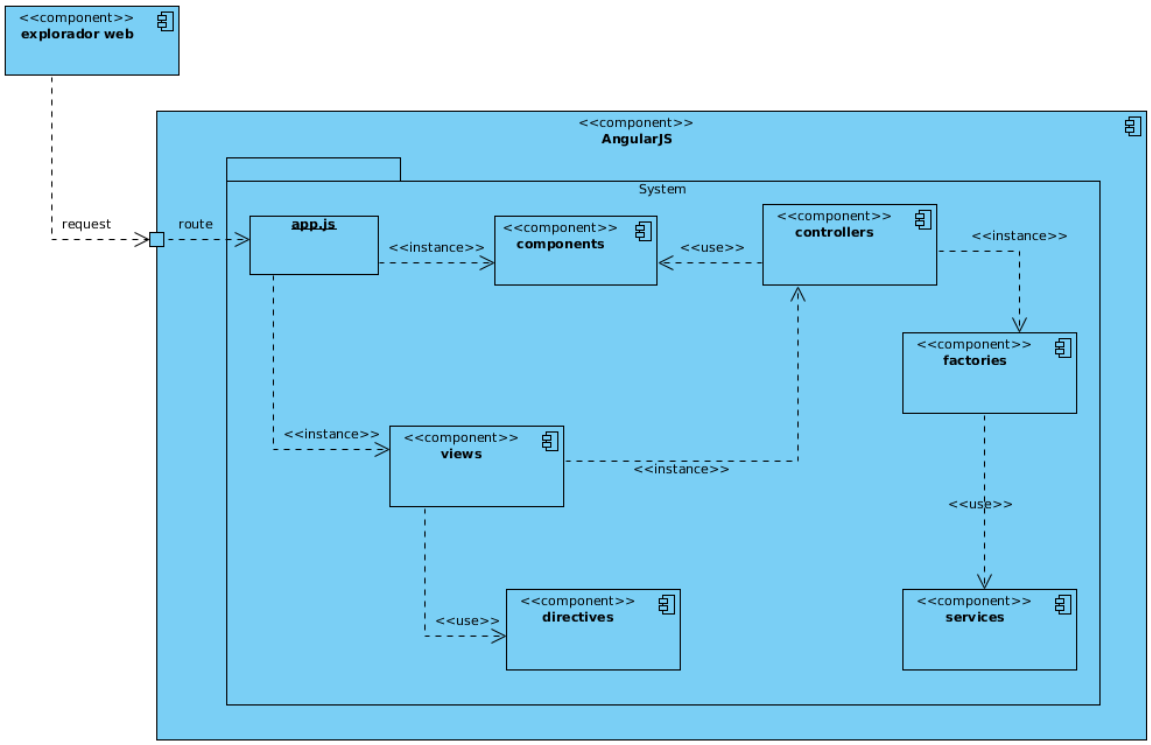

Fig. 6.1: Diagrama de Componentes del módulo de AngularJS

6.2 Capa de negocio

6.2.1 Librerías externas

Como se comentó en secciones previas, esta capa alberga toda la lógica del sistema y se encargar de gestionar las peticiones entrantes mediante la *API REST* y ofrecer una respuesta.

Para implementar este sistema, se ha hecho uso de diversas librerías que facilitaron su elaboración.

- *Jersey*[9]
  Esta librería, *Open Source*, ofrece una implementación de la *API JAX-RS*. *JAX-RS* es una abstracción que proporciona el lenguaje *Java* para crear servicios web. La librería *Jersey* ofrece una implementación y ampliación que permite crear, con mucha facilidad, un servicio web *REST*.

- *Jackson*[10]
  Esta librería se suele usar junto con la anterior, con el fin de facilitar el uso

[9] `https://jersey.java.net/`
[10] `http://wiki.fasterxml.com/JacksonHome`

de datos en formato *JSON* en *Java*. Con esta librería se pueden transformar objetos del sistema en cadenas en formato *JSON*, y viceversa. Como es de esperar, esto facilita mucho la gestión de una *API REST*, donde los datos, tanto que se envían como que se reciben, están en formato *JSON*.

- *Hibernate*[11]
  Hibernate es una herramienta de Mapeo objeto-relacional (ORM) para el lenguaje *JAVA*, que facilita el mapeo de atributos entre una base de datos relacional y el modelo de objetos de una aplicación, mediante la definición de ficheros *XML* o mediante anotaciones.

- *Datastax Cassandra*[12]
  Cliente de *Java* para Cassandra. Permite crear conexiones entre un servidor o clúster *Cassandra* y ejecutar instrucciones *CQL*[13]. Agiliza y facilita en gran manera el uso de *Cassandra* en este lenguaje.

- *Log4j*[14]
  Log4j es una librería adicional de *Java* que permite a nuestra aplicación mostrar mensajes de información de lo que está sucediendo en ella, lo que habitualmente se conoce como un *log*. Tiene la ventaja de ser mucho más dinámico y, por tanto, configurable que un simple *System.out.println()*.

6.2.2 Diagrama de Componentes

En este diagrama, se podrá ver cómo interactúan los diferentes componentes de esta capa. El sistema propone un punto de entrada a la *API* que será el encargado de enviar las peticiones a dicha capa. En cuanto a la capa de *BLL*[15], es la que se contiene toda la lógica y la encargada de comunicarse con los dos módulos de *Spark* y *Vision*. El componente de *Spark* se comunica con el sistema, nuevamente, mediante una petición *HTTP* para enviarle los datos de la ejecución. En cuanto al módulo de *Vision*, devuelve una respuesta directa a la llamada que se le hace.

[11] `http://hibernate.org/`
[12] `http://datastax.github.io/java-driver/2.1.5/`
[13] Cassandra Query Language
[14] `http://logging.apache.org/log4j/2.x/`
[15] Del Inglés, Business-Logic Layer

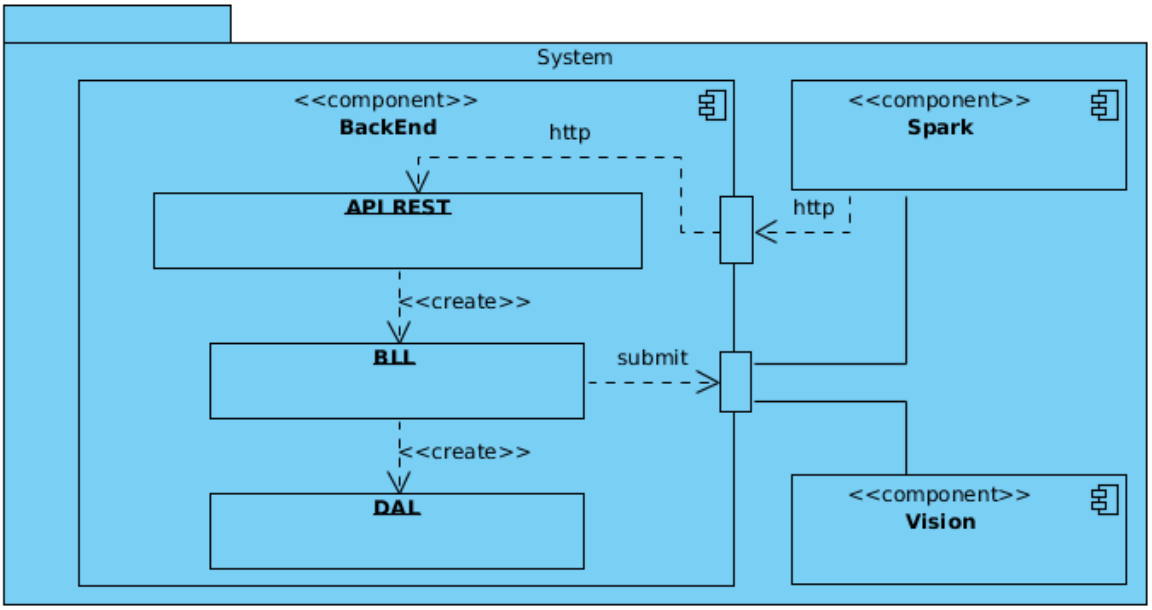

Fig. 6.2: Diagrama de Componentes de la capa de lógica

6.2.3 Diagrama de Clases

Los diagramas de clase son diagramas, de estructura estática, que muestran las clases del sistema y sus interrelaciones, incluyendo herencia, agregación, etc.. En la imagen 6.3 se muestra el diagrama una vez se ha implementado el sistema. En él se pueden ver todas las relaciones y el funcionamiento interno, así como la interacción con los módulos de *Vision* y *Spark*.

A pesar de que no se ha especificado, en el diagrama se puede ver una clara separación de los componente, tal y como se mencionó en la sección 5.3 del capítulo 5. Esta separación corresponde a las capas de los *end-points* -*Event*, *Channel*, ...-, la capa de la lógica de negocio -*EventController*, *ChannelController*, ...- y la capa de datos -*EventDAO*, *ChannelDAO*, ...-.

Otro punto importante a destacar es la clase *HibernateListener*. Esta clase se ejecuta en la inicialización de la aplicación y es el encargado de inicializar tanto la conexión al clúster de *Cassandra* como la librería de *Vision*. Estas dos clases se han implementado siguiendo el patrón de diseño *Singleton*[34], por lo que es necesario inicializarlas antes de que el resto puedan hacer uso de ellas.

En cuanto a la capa de datos, se ha implementado una clase abstracta de la que luego extenderán el resto. Esto se ha hecho con el fin de evitar duplicidades de código ya que muchas de las funciones de los *DAOs*[16] serán iguales en todos ellos.

[16]Del Inglés, Data Access Object

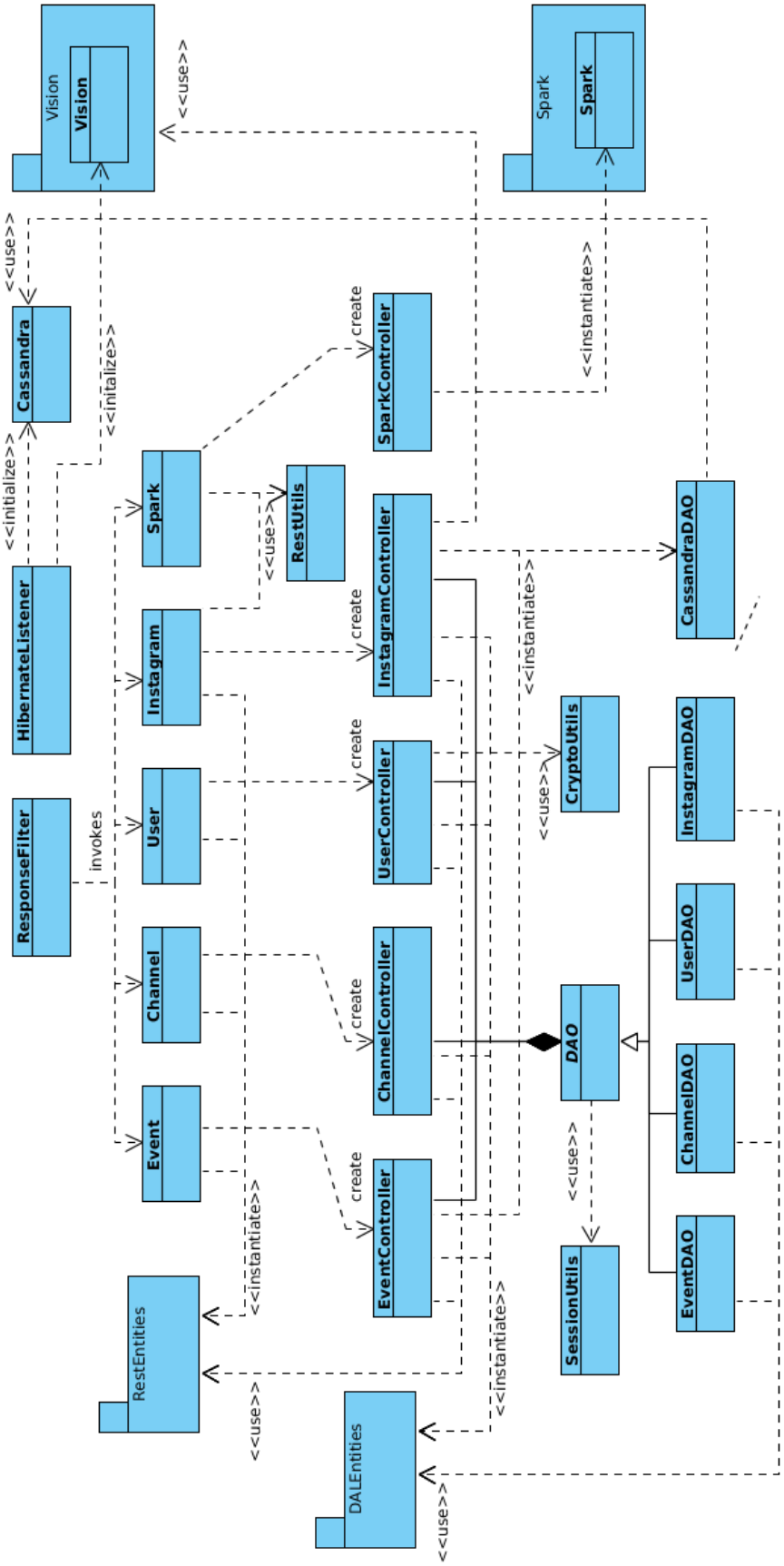

Fig. 6.3: Diagrama de Clases de la capa de lógica

6.2.4 Modulo de Visión

Como se ha comentado anteriormente, este sistema hace uso del módulo desarrollado por *Sana Imtiaz* para clasificar imágenes a partir de ciertos modelos ya entrenados. A pesar de estar fuera del alcance de este proyecto, sí es necesario saber cómo funciona para comprender qué es lo que realmente hace. Sin entrar en mucho detalle, se explicará qué hace y el proceso que sigue.

Este proyecto se ocupa del problema del reconocimiento automático de imágenes, de *Instagram* en nuestro caso, con el objetivo de resolver y clasificar las imágenes con la ayuda de mecanismos de aprendizaje supervisado. Para este propósito, es necesario obtener una colección de imágenes y realizar el reconocimiento visual de la categoría deseada, en este caso comida. Para ello primeramente se extraen las características locales del contenido de las imágenes -*SIFT descriptors*- y se filtran en vectores de características compactos -conocidos como *Bag of Words*, y explicados en la sección 2.3-. Un clasificador -*Super Vector Machine*- se entrena con la ayuda de los *hashtags* de *Instagram* y se extraen las características de las imágenes.

Los resultados del algoritmo desarrollado son resaltables, ya que predice con un 93% de acierto algunas de las categorías entrenadas, tal y como se puede ver en la imagen 6.4.

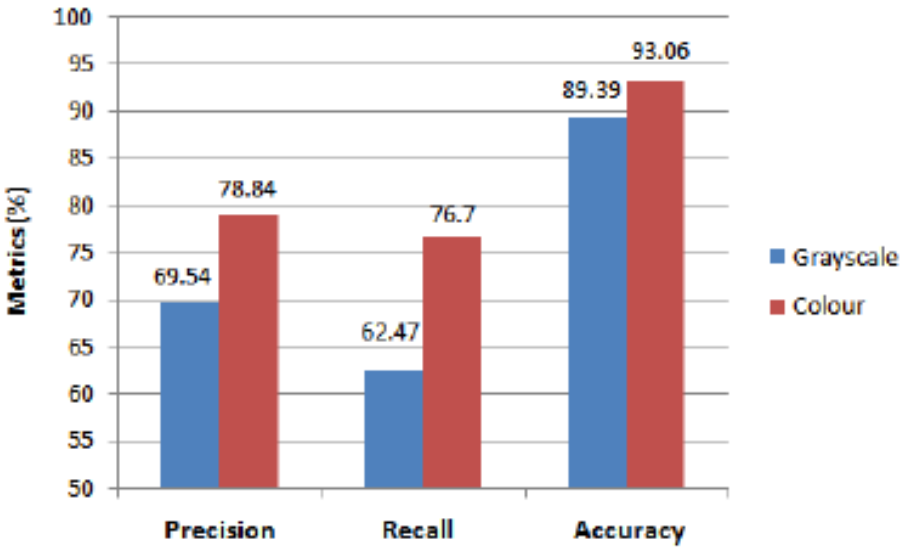

Fig. 6.4: Resultados ejecución

Para hacer uso de este módulo se ha implementado una clase, *Vision*, que se encarga de recibir una *url* de una imagen de *Instagram* y predecir el contenido con dicho módulo, tal y como muestra la imagen 6.3.

6.3 Capa de datos

Como ya se ha visto en anteriores secciones, la capa de datos es la encargada del almacenamiento de los datos, y de la ejecución de los algoritmos de *Apache Spark*. Para ello, se ha contemplado el uso de dos gestores de bases de datos distintos, ya que era necesario almacenar datos para diferentes propósitos.

6.3.1 Clúster Cassandra

Como *Cassandra* era el gestor que iba a tener más peso e iba a gestionar muchos más datos, se ha decidido implementar un clúster con cuatro máquinas. El clúster consigue que *Cassandra* pueda almacenar más información y la disponibilidad sea mucho mayor, ya que se puede acceder a cualquiera de ellos, y no se colapsan las peticiones del servidor.

```
admuser@spark-cluster-1:~$ nodetool status catwalk
Datacenter: DC1
===============
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address        Load       Tokens       Owns (effective)  Host ID                               Rack
UN  147.83.42.227  380.46 MB  256          50.0%             2bc5c311-11e8-451b-8498-92c6bce89a4b  RAC1
DN  147.83.42.145  360.99 MB  256          50.3%             77e59342-e0c0-4a63-8bd8-58c97233bbaf  RAC1
UN  147.83.42.180  323.61 MB  256          48.8%             03f3efb0-e37e-4d6a-8488-002cc457a7bf  RAC1
UN  147.83.30.202  534.8 MB   256          50.8%             638e1c8d-8a04-4cbf-925b-16d03c7a0c05  RAC1
```

Fig. 6.5: Estado del clúster de Cassandra

Como se puede ver en la imagen 6.5, el clúster permite tener una carga de datos más repartida entre los diferentes servidores, sin sobrecargar excesivamente cada uno de ellos.

6.3.2 Clúster Spark

Por último, esta capa también engloba a *Apache Spark*. Este es el encargado de recibir peticiones, ejecutar los *scripts* en máquinas externas, y devolver un resultado. Para ello, y después de ciertas pruebas y ver que no rendía lo suficiente en una sola máquina, se ha creado un clúster con 4 máquinas. En este clúster *Spark* se encarga de recibir las peticiones que realiza la capa de negocio y una vez ha terminado la ejecución informar, mediante una petición *HTTP*, a la *API REST* del resultado.

En un clúster *Spark* se crean diferentes componentes, que luego son los encargados de colaborar entre sí para ejecutar la petición. El principal es el *Spark Master*, que es el encargado de recibir la petición y ejecutar el *script* en el resto de servidores. Una vez se tiene el *master*, es necesario crear una serie de *Spark Workers* que serán los

encargados de ejecutar el código en los diferentes servidores del clúster. En la imagen 6.6 se hace una aproximación a la arquitectura del clúster que se ha creado.

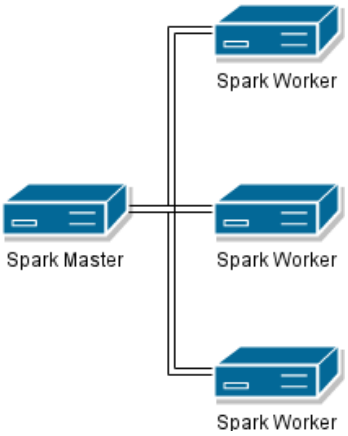

Fig. 6.6: Arquitectura del clúster de Spark

Para ejecutar los *scripts* disponibles en el clúster, es necesario hacer una petición al *Spark Master*, y una vez iniciada, este se encargará internamente de gestionarla.

6.3.2.1 Voronoi script

Hasta el momento, el único *script* que se ejecuta en este clúster es el que crea un mapa de *Voronoi* a partir de las localizaciones de los datos recibidos de *Instagram* para un evento en concreto.

Lo que hace este *script* es, a partir de los identificadores de los canales que componen un evento, obtiene los datos almacenados en *Cassandra* y genera un *array* con los campos *latitude* y *longitude*, que serán los que nos permitan calcular el algoritmo *K-Means*, explicado anteriormente. Una vez hemos ejecutado este algoritmo, podemos predecir a qué grupo pertenece cada dato recibido de *Instagram*, según las coordenadas, y así tenerlos agrupado en el número de grupos que nos interesa, en este caso 5, por proximidad.
En este momento, tenemos una estructura tipo *map* con el identificador del grupo, y los registros asociados a cada uno de ellos. Como estos registros tienen los campos *latitude* y *longitude* podemos situarlos en un mapa a partir de una serie de formulas matemáticas.

Para poder situar las coordenadas en el mapa hemos usado la proyección cilíndrica equidistante[35]. Esta simple proyección de mapas nos permite situar nuestras coordenadas en un mapa a partir de la proporcionalidad entre la longitud y el eje x y la

latitud y el eje *y*. Si queremos representar un punto de coordenadas en un mapa, a partir de esta proyección, debemos aplicar los siguiente cálculos:

$$pixel\_x = image\_width \times \frac{longitude}{360} + \frac{image\_width}{2} \tag{6.1}$$

$$pixel\_y = -image\_height \times \frac{latitude}{180} + \frac{image\_height}{2} \tag{6.2}$$

Como se puede ver, la posición del *pixel_x* se calcula a partir de dividir la coordenada de la longitud entre 360 - la esfera terrestre está dividida en 360 meridianos - y multiplicarla por el ancho de la imagen para escalarla. Por último se le suma la mitad del ancho, ya que el punto *(0,0)* de una imagen se sitúa en la esquina izquierda.
En cuanto al cálculo del *pixel_y*, este se calcula de una manera similar al anterior. Cabe resaltar el signo negativo del resultado ya que, en una imagen, la coordenada *y* va de arriba hacia abajo. En este caso, se divide la latitud entre 180, que son los paralelos en los que se divide la esfera terrestre.
Una vez tenemos estos cálculos realizados, ya es posible situar las coordenadas de los datos obtenidos de *Instagram* en el mapa, obteniendo un resultado similar al de la imagen 6.7.

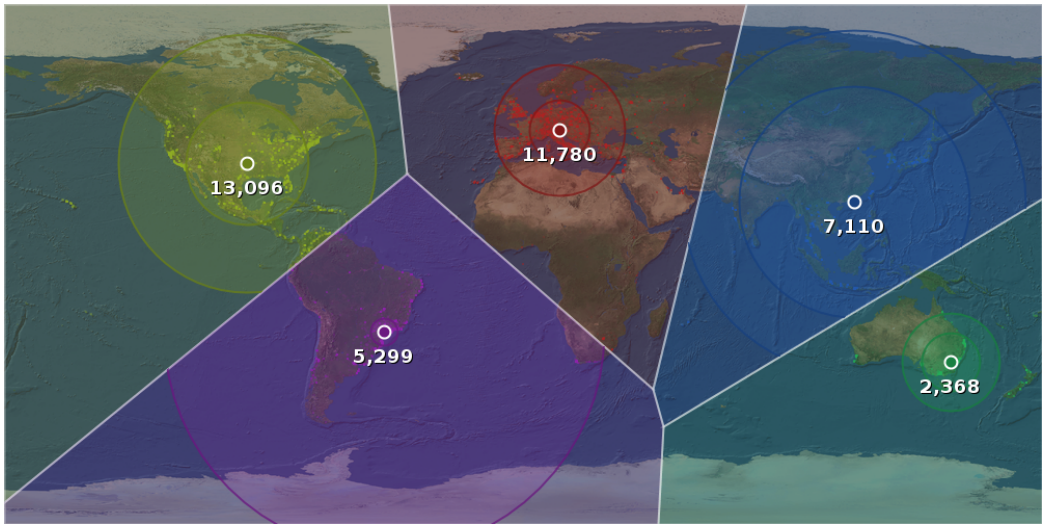

Fig. 6.7: Ejemplo mapa de Voronoi

Con el fin de mostrar una mayor información, y poder ver observar la concentración en el mapa, se han incluido dos círculos alrededor de cada *centroide*.

- El primero marca el *percentil* 50, es decir, dentro de este círculo se engloban el 50% de datos de dicho clúster.

- El segundo marca el *percentil* 90, es decir, contiene el 90% de los datos del clúster.

Como se puede ver, se han escogido 5 clúster. Esto se debe a que con este número, después de realizar varias pruebas, es con el que mejor se reparten los datos, creando siempre un mapa similar al que se muestra en la imagen - en cuanto a la disposición de los grupos. En caso de ampliar el número, los grupos quedan demasiado pequeños y juntos, y no permite apreciar el reparto. De todos modos, este número será editable por el usuario en futuras versiones.

Capítulo 7

Resultados

En este capítulo se mostrarán los diferentes resultados obtenidos una vez el sistema se ha creado y ha sido probado. Se comentarán tanto los problemas que se han observado en temas de rendimiento, o con la *API* de *Instagram*, así como diferentes ejemplos del resultado que obtendrá un usuario en la web.

7.1 Restricciones

Tanto el *hardware* como *Instagram* han mostrado diversas restricciones a la hora de ejecutar el sistema. A continuación de comentan los diferentes problemas que se han encontrado.

7.1.1 Rendimiento

El sistema se ha desarrollado teniendo en cuenta la gran cantidad de datos que se gestionarán, por ello se ha realizado un diseño modular con el cual se pueda escalar sin dificultades. Por tanto, si el número de usuarios crece o el actual sistema *hardware* no soporta todo el trabajo que se está generando, es posible distribuirlo en diferentes máquinas.

En estos momentos el sistema, la capa de negocio, se está ejecutando en una máquina con *1 Gb* de *RAM* y un procesador de *2,3 MHz*. Tras diversas pruebas de rendimiento, el sistema está funcionando correctamente en este entorno. En estos momentos se están gestionando *10* subscripciones de *Instagram*, las cuales generaron alrededor de *800Mb* de datos. Esto nos permite ver que, a pesar de no poseer un *hardware* demasiado potente, el sistema es capaz de gestionar una gran cantidad de datos, por lo que si se ampliasen las prestaciones más adelante, el sistema podría

llegar a gestionar enormes cantidades de datos así como una carga de usuarios y subscripciones considerable.

A pesar de esto, también se han encontrado limitaciones ya que el módulo de visión consume una gran cantidad de memoria *RAM*. En uno de los casos de prueba, se estaban gestionando *10* subscripciones de *Instagram* y en *3* de los canales se había activado la predicción de *tags*. Pasados unos ciertos niveles de peticiones por parte de *Instagram* y aumentar las imágenes que se estaban analizando, el sistema no tenía memoria suficiente y, tanto el servidor *Jetty*, como el módulo de visión, dejaban de ejecutarse. Esto se debe a los recursos en el entorno de pruebas, pero en caso de que estos se ampliasen, el sistema se comportará de forma correcta, sin presentar ninguno de estos tipos de problemas.

En cuanto al rendimiento del clúster tanto de *Cassandra* como de *Apache Spark*, se han realizado pruebas con el fin de comprobar la cantidad de *jobs* que es capaz de ejecutar el clúster de *Apache Spark*, así como la cantidad de datos que soporta *Cassandra* antes de volverse inaccesible. Se ha concluido en que el clúster de *Apache Spark* permite la ejecución de *3 jobs* y *300.000* registros paralelamente, con un tiempo de respuesta de, aproximadamente, *1.5* minutos. En cuanto al clúster de *Cassandra*, lo máximo que se han podido almacenar antes de empezar a encontrar comportamientos extraños y errores de ejecución, han sido *1.5Gb*. Se ha llegado hasta los *1.9Gb* y con este nivel de datos *Cassandra* ya no respondía correctamente a las *queries*

7.1.2 Instagram API

En cuanto a la *API* que ofrece *Instagram*, también se han encontrado diversos limites. El primero, y el que más se genera, es el límite de *5000* peticiones por hora y por usuario. El sistema usa el *OAuth* de *Instagram* con el fin de obtener las credenciales necesarias para ejecutar las peticiones relacionadas con sus eventos, esto hace que las peticiones que se realizan a *Instagram* no recaigan sobre una sola cuenta, con lo que el límite saltaría mucho antes. A pesar de esto, ciertas subscripciones generan mucho tráfico y causan que se exceda el límite en poco tiempo, por lo que hasta que el límite no se reinicia, se están perdiendo datos de dicha subscripción. Ante este problema no se ha encontrado ninguna solución ya que es un límite que impone *Instagram* para todos los usuarios. La mejor solución encontrada ha sido la de realizar la autenticación y que cada usuario gestione sus peticiones a partir de sus credenciales, y así repartir la carga en cada uno de los usuarios.

Otros de los problemas encontrados ha sido con las subscripciones de geolocalización. Como se puede ver en la web, el sistema limita el área de una localización a

un radio máximo de *5000* metros. Esta restricción también viene dada por *Instagram*, ya que no permite realizar subscripciones por localizaciones con un radio mayor de estos metros. Ante esta restricción no se ha conseguido ninguna solución, ya que es una restricción de la *API* de *Instagram*, por lo que no se puede realizar de ninguna otra manera.

7.2 Resultados visuales

A pesar de encontrar ciertos inconvenientes mencionados anteriormente, el sistema funciona correctamente en dicha arquitectura *hardware* con un nivel medio de datos. En esta sección se mostrarán diferentes imágenes en las que se puede observar el resultado conseguido una vez finalizado el sistema.

En esta primera imagen se puede observar el resumen de los eventos que verán los usuarios una vez hayan entrado en el sistema. En esta pantalla, los usuarios pueden observar el porcentaje de realización de cada uno de los eventos, así como eliminarlos o acceder a los detalles de cada uno de ellos.

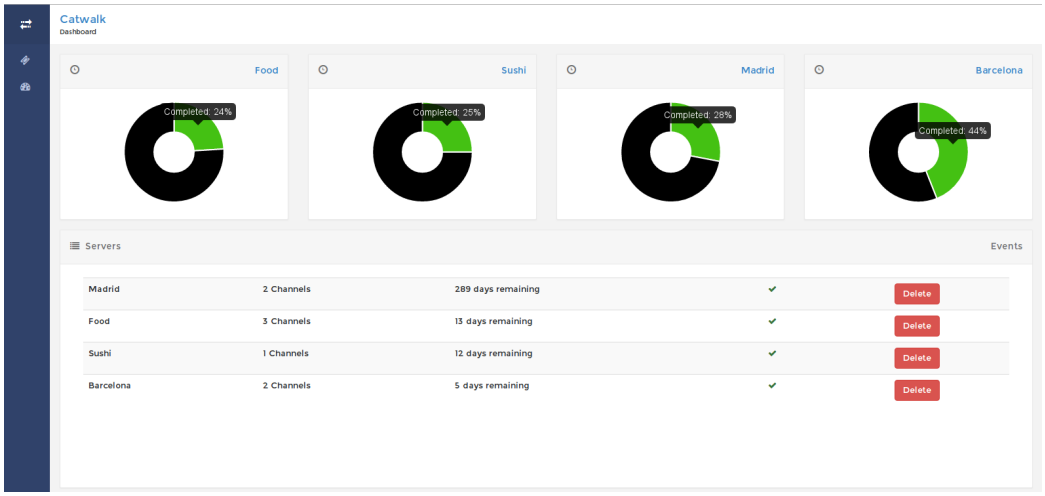

Fig. 7.1: Resumen del Dashboard

Tal y como muestra la imagen 7.1, los cuatro eventos creados se están ejecutando en este momento y ya han avanzado más del 20% cada uno de ellos.

Si el usuario decide ver los detalles de uno de ellos, se le mostrarán los detalles básicos del evento -nombre, inicio, fin..- así como la posibilidad de crear nuevos canales. En esa pantalla también se pueden observar tanto el diagrama de rendimiento de los canales, como el mapa de *Voronoi*, del que ya se ha hablado con anterioridad.

Además, en esta pantalla se mostrarán las 8 últimas imágenes recibidas de las subscripciones, para que el usuario se pueda hacer una idea de lo que se está recibiendo de cada uno de los canales. A continuación se muestran diferentes imágenes que resumen esta pantalla. En cada una de ellas se muestran los diferentes detalles de dicha pantalla.

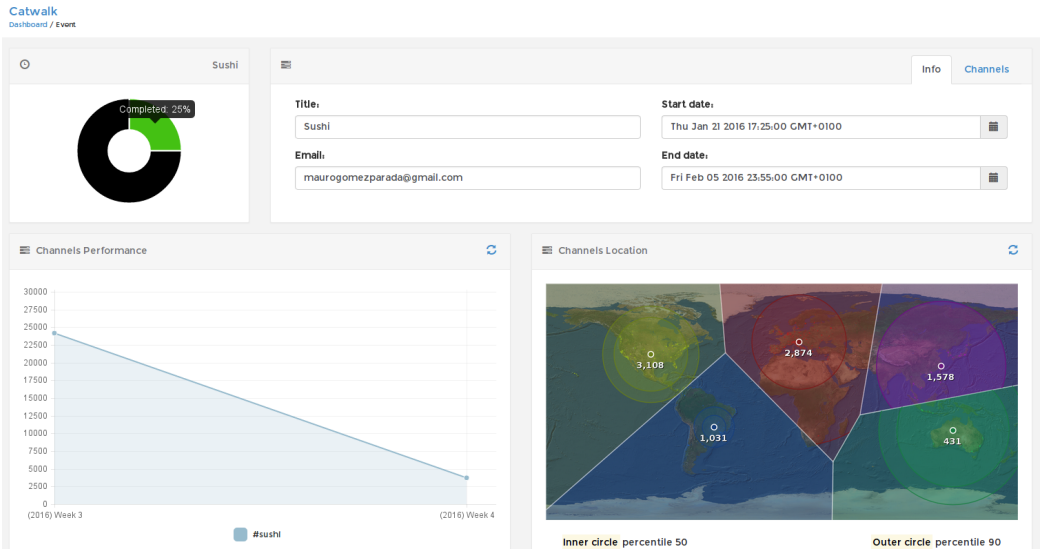

Fig. 7.2: Resumen del evento

La imagen 7.2 muestra el resumen de los datos del evento, en los que se permite modificar cualquiera de ellos, así como un gráfico en donde se puede ver el rendimiento de los canales - un único canal en este caso - como el mapa de *Voronoi* con las localizaciones de los datos recibidos.

Fig. 7.3: Últimas 8 imágenes del evento

En cuanto a la imagen 7.3, muestra las últimas 8 imágenes recibidas de los canales subscritos. Estas imágenes, al igual que el mapa de *Voronoi* se le permite al usuario ampliarlas para que pueda visualizarlas con un mayor detalle.

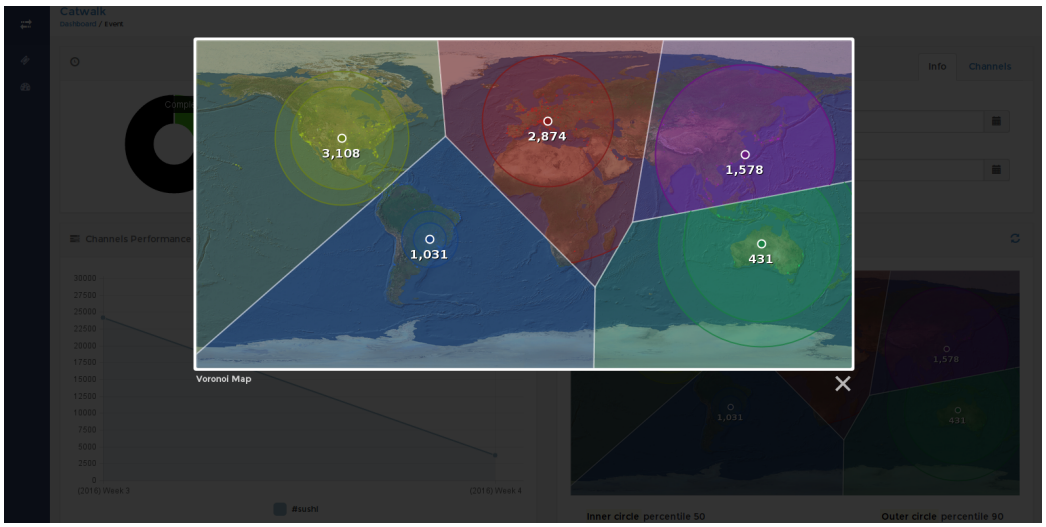

Fig. 7.4: Visualización del mapa de *Voronoi* a mayor escala

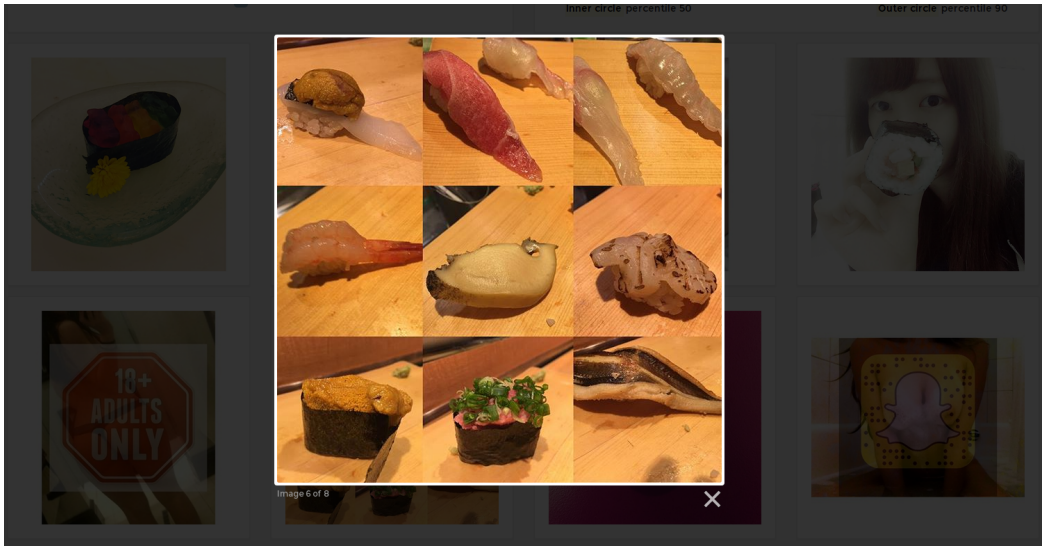

Fig. 7.5: Visualización de las imágenes recibidas como album

Como se puede observar, en las imágenes 7.4 y 7.5 se muestra la visualización que se obtendría al ampliar cualquiera que las imágenes. Esto permite que se puedan ver en mayor detalle, y, sobre todo, poder observar mejor el detalle del mapa de *Voronoi*.

Por último, en estas últimas imágenes se muestran los formularios y la visualización que tienen los usuarios de los canales creados en uno de sus eventos, tanto de *hashtags* como de *geolocalización*.

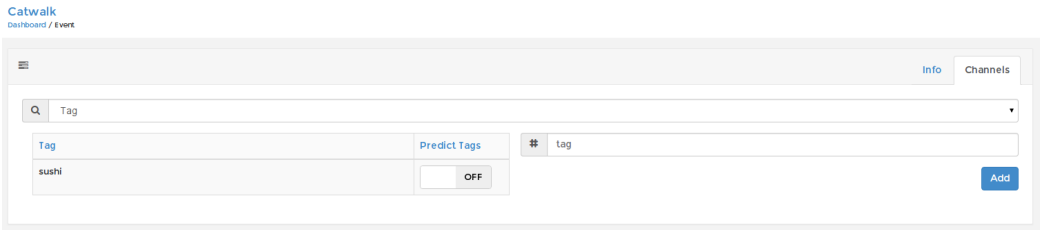

Fig. 7.6: Resumen de canales de *hashtags* y formulario

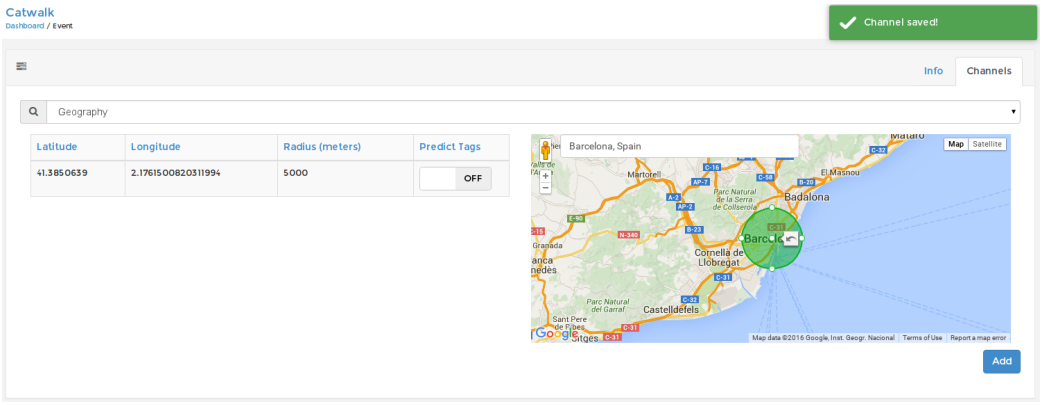

Fig. 7.7: Resumen de canales de *geolocalización* y formulario

Como se puede ver en las imágenes 7.6 y 7.7, estas secciones muestran tanto el formulario como un resúmen de los canales creados. En estas secciones los usuarios pueden activar la predicción de *tags* para las imágenes que se reciban.

Tal y como se pudo ver en las anteriores imágenes, este ha sido el resultado final de la elaboración del proyecto. Toda esta interfaz se ha diseñado intentando mantener la simplicidad para los usuarios finales, y con la idea de que pueda ser usada en cualquiera de los dispositivos.

Capítulo 8

Conclusiones y futuro trabajo

En este último capítulo se presentan las conclusiones finales del proyecto, así como las personales, y las líneas del futuro trabajo del mismo.

8.1 Conclusiones generales del proyecto

Haciendo una visión general del proyecto desarrollado, se puede ver como tanto la tecnología aplicada como el sistema creado, puede ser de gran ayuda tanto a grandes empresas como a pequeños emprendedores.

Este proyecto ha nacido a partir de una serie de ideas, que con el tiempo fueron madurando y cambiando hasta llegar a este punto. Una vez terminado este proyecto podemos obtener *feedback* de diversos usuarios y comprobar si realmente cumple la función que hasta ahora se le ha encomendado. A pesar de estar todavía inmaduro y necesitar mejoras y más funcionalidades, lo que sí se puede afirmar es que el sistema es funcional y posee ya el potencial de aportar valor a muchas empresas que necesiten obtener información y lanzar campañas de *marketing* en *Instagram*.

Cabe destacar, también, el importante papel que las tecnologías *open source* como *Cassandra*, *Apache Spark* o *Angular* han tenido en este proyecto, ya que han permitido construir un sistema novedoso, escalable y de alto rendimiento, ofreciendo flexibilidad en cuanto al dominio de los datos y una gran capacidad de gestión de grandes volúmenes de datos, que si no hubiera sido por estos *frameworks*, habría sido casi imposible realizar. Otro de los importantes puntos de este proyecto es la escalabilidad que permite el sistema. Tanto los módulos desarrollados para *Apache Spark* como para *Cassandra*, han sido diseñados pensando en la escalabilidad y esto se ha conseguido. En estos momentos, tal y como se comentó anteriormente, estos dos

módulos están ejecutandose sobre un clúster de cuatro máquinas, pero esto se podría ampliar en cualquier momento, haciendo que el sistema pueda soportar cargas mucho mayores. Esto también permite que la cantidad de imágenes que se puedan procesar sea mucho mayor gracias al aumento de prestaciones *hardware* y evitando tener que modificar en código creado.

Como resultado de este proyecto, tenemos un sistema funcional basado en tecnologías *Big Data* en tiempo real y que funciona sobre cualquier parte del mundo. A mayores, el sistema es generalizable y extensible a otras redes sociales, como puede ser *Twitter*. Dentro del marco del proyecto se han planteado diferentes escenarios que verifican que el sistema ha sido implementado siguiendo patrones de diseño *software* y permiten una gran reusabilidad de los diferentes módulos, así como la posibilidad de una rápida y sencilla ampliación de funcionalidades en caso de ser necesario.

En resumen, este proyecto se finaliza cumpliendo las expectativas marcadas al inicio, ofreciendo una solución *software* de calidad y funcional a través de diferentes tecnologías *Big Data* en tiempo real y con una gran posibilidad de monetización.

8.1.1 Conclusiones personales

A nivel personal, realizar este proyecto en el *Barcelona Supercomputing Center* me ha permitido aprender muchas cosas que difícilmente podría en otros ámbitos. También me ha permitido trabajar con gente experta en todas estas tecnologías, y de los que he podido aprender y mejorar mis conocimientos en estos ámbitos.

En este proyecto he podido conocer el mundo de la investigación, que hasta ahora no había tenido la oportunidad, y que ha resultado ser más que interesante. El hecho de utilizar tecnologías punteras para resolver ciertos problemas y poder investigar en cómo funcionan realmente y no solo en cómo utilizarlas, ha sido una gran motivación. También, la posibilidad de colaborar en la publicación de un libro sobre *Apache Spark*[1] me ha ayudado a ampliar mucho más mis conocimientos sobre este *framework* así como en el *Big Data* en general.

[1]Introducción a Apache Spark, `http://www.sparkbarcelona.es/`

8.2 Trabajo futuro

La realización de este proyecto ha abierto nuevos frentes, así como ideas de mejora que hasta el momento no habían aflorado, y que no se han podido llevar a cabo por motivos de tiempo o por el alcance de este proyecto.

A continuación se detallan los principales puntos de mejora y próximos desarrollos del sistema

- Hasta el momento nos hemos centrado en desarrollar todo el *stack* de *software* necesario para crear el sistema, sin poder dedicar mucho tiempo a lo realmente importante, que es predecir información de *Instagram* a partir tanto de las imágenes como de lo que la gente escribe. Este es un punto importante y que puede permitirnos destacar frente a otras soluciones similares que pueda haber en el mercado.

- Ligado al punto anterior, es necesario que el usuario pueda visualizar más información acerca de los datos obtenidos con sus eventos y canales. Para esto es necesario desarrollar nuevas ideas sobre qué datos son relevantes para realizar campañas de marketing, y realizar dichos gráficos.

- Hasta el momento no nos hemos preocupado demasiado por el rendimiento del sistema, pero es parte fundamental para un sistema de estas características. Debido a la falta de recursos, ahora mismo el sistema se está ejecutando sobre un clúster limitado, que será necesario ampliar para futuras versiones. En estos momentos hay muchos problemas tanto a la hora de leer los datos de *Cassandra* como en realizar acciones en *Spark* o ejecutar el algoritmo de predicción en imágenes, y todo es debido a las limitaciones del *hardware*, por lo que sería fundamentar probar a ejecutar el sistema sobre un *hardware* con las prestaciones suficientes y comprobar el rendimiento.

Estos son algunos de los puntos más importantes para un futuro cercano, y de cara a poder publicarlo y obtener *feedback* real.

Como nuevas funcionalidades, y teniendo en cuenta la filosofía del proyecto, se debería implementar un sistema de anuncios personalizado, en el que el usuario pudiera crear campañas de anuncios en *Instagram* a partir de los datos obtenidos en sus canales y focalizándolo a ciertos usuarios. Un ejemplo podría ser crear un evento para todos aquellos usuarios de la zona de Barcelona, y lanzarle una campaña de

anuncios a aquellos usuarios que adjunten imágenes de cervezas o vino, mediante el reconocimiento previo de las imágenes.

Referencias

[1] Craig Smith. Instagram user statistics, 2015.

[2] Jason Mander. Daily time spent on social networks rises to 1.72 hours, 2015.

[3] Maeve Duggan, Nicole B. Ellison, Cliff Lampe, Amanda Lenhart, and Mary Madden. Frequency of social media use, 2015.

[4] Màrius Carol. Instagram supera a twitter, 2016.

[5] Adam Dobrin. A review of properties and variations of vornoi diagrams. *Whitman College*, 2005.

[6] Gil Press. Internet of things by the numbers: Market estimates and forecasts, 2014.

[7] Doug Laney. 3d data management: Controlling data volume, velocity and variety. *META Group*, 949, 2001.

[8] Randal E. Bryant, Randy H. Katz, and Edward D. Lazowska. Big-data computing: Creating revolutionary breakthroughs in commerce, science, and society. *Computing Community Consortium*, 2008.

[9] Elorie Knilans. The 5 v's of big data, 2014.

[10] Bernard Marr. Why only one of the 5 vs of big data really matters, 2015.

[11] Facebook. Facebook reports third quarter 2015 results, 2015.

[12] Christof Strauch. Nosql databases.

[13] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. *University of California, Berkeley*, 2008.

[14] Rubén Tous, Anastasios Gounaris, Carlos Tripiana, Jordi Torres, Sergi Girona, Eduard Ayguadé, Jesús Labarta, Yolanda Becerra, David Carrera, and Mateo Valero. Spark deployment and performance evaluation on the marenostrum supercomputer. In *2015 IEEE International Conference on Big Data, Big Data 2015, Santa Clara, CA, USA, October 29 - November 1, 2015*, pages 299–306, 2015.

[15] Tapas Kanungo, David M. Mount, Nathan S. Netanyahu, Christine D. Piatko, Ruth Silverman, and Angela Y. Wu. An efficient k-means clustering algorithm: Analysis and implementation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24, 2002.

[16] David Arthur and Sergei Vassilvitskii. K-means++: The advantages of careful seeding. *ACM-SIAM symposium on Discrete algorithms*, pages 1027–1035, 2007.

[17] Datastax Corporation. Introduction to apache cassandra, 2013.

[18] Alberto Diego Prieto Löfkrantz. Do you know cassandra?, 2013.

[19] Chih-Fong Tsai. Bag-of-words representation in image annotation: A review. *ACM-SIAM symposium on Discrete algorithms*, 2012:19, 2012.

[20] Rob Fergus. Bag-of-words models, 2012.

[21] M.A. Fischler and R.A. Elschlager. The representation and matching of pictorial structures. pages 67–92, 1973.

[22] Norman L. Biggs, E. Keith Lloyd, and Robin J. Wilson. Graph theory 1736-1936. 1976.

[23] V. K. Balakrishnan. Graph theory. 1997.

[24] L.C. Freeman, S.P. Borgatti, and D.R. White. Centrality in valued graphs: a measure of betweenness based on network flow. pages 141–154, 1991.

[25] Jimeng Sun and Jie Tang. A survey of models and algorithms for social influence analysis. *Charu C. Aggarwal*, 949:177–214, 2011.

[26] Rubén Tous, Jordi Torres, and Eduard Ayguadé. Multimedia big data computing for in-depth event analysis. In *2015 IEEE International Conference on Multimedia Big Data, BigMM 2015, Beijing, China, April 20-22, 2015*, pages 144–147, 2015.

[27] John Scott. Social network analysis. 2013.

[28] J Golbeck. Analyzing the social web. *Morgan Kaufmann*, 2013.

[29] Alexander Dryer. How the nsa does social network analysis, 2013.

[30] D. Crockford. The application/json media type for javascript object notation (json), 2006.

[31] Roy Thomas Fielding. Representational state transfer (rest). 2000.

[32] Martin Fowler. Presentation model, 2004.

[33] MSDN. Diagramas de componentes de uml.

[34] OODesign. Singleton pattern.

[35] John P. Snyder. Flattening the earth: Two thousand years of map projections. 1993.