Proyecto Fin de Carrera

Ingeniería de Telecomunicación

Extracción y análisis de información de Twitter utilizando técnicas de Big Data: Aplicación de monitorización de servicios web

Autor: Alejandro Guerra Cabrera

Tutor: Antonio J. Estepa Alonso

Departamento de Ingeniería Telemática

Escuela Técnica Superior de Ingeniería

Universidad de Sevilla

Sevilla, 2016

Proyecto Fin de Carrera
Ingeniería de Telecomunicación

Extracción y análisis de información de Twitter utilizando técnicas de Big Data: Aplicación de monitorización de servicios web

Autor:
Alejandro Guerra Cabrera

Tutor:
Antonio J. Estepa Alonso
Profesor titular

Departamento de Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2016

Extracción y análisis de información de Twitter utilizando técnicas de Big Data: Aplicación de monitorización de servicios web

Autor: Alejandro Guerra Cabrera

Tutor: Antonio J. Estepa Alonso

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2016

El Secretario del Tribunal

A mis padres

A mi hermana

A mis seres queridos

Agradecimientos

En primer lugar quiero agradecer especialmente a mis padres la educación, los valores y las oportunidades que me han dado en la vida, siempre desde el esfuerzo, el sacrificio y el amor incondicional. A ellos les debo todo y me faltarían vidas para agradecérselo. Espero al menos haberme convertido en la persona de provecho que ellos esperaban.

Gracias a mi hermana por apoyarme siempre en todo y sobre todo por ser ella. Sé que me admira como buena hermana pequeña, pero para mí ella también es un orgullo y digna de la mayor admiración.

Gracias a Lorena por su compresión y por el ánimo y empuje transmitidos que me han ayudado a terminar la carrera.

Por último, pero no menos importante, gracias a mi familia, mis amigos de la infancia, del instituto, de la universidad, del trabajo,... todas esas personas especiales que van dejando huella en ti, haciéndote mejor tanto en lo personal como en lo profesional.

Alejandro Guerra

Sevilla, 29 de septiembre de 2016

Resumen

Este proyecto persigue el desarrollo de un software capaz de, por un lado, extraer y almacenar en una base de datos *NoSQL* información de *Twitter*, concretamente la información que publican cuentas cuyo fin es informar del estado de servicios web; y por otro lado, analizar esa información empleando técnicas del *Big Data* para extraer estadísticas de funcionamiento de dichos servicios.

Puesto que el objetivo de este proyecto no es el análisis semántico de los tuits, también se creará un software capaz de monitorizar un servicio web y publicar información de su estado en *Twitter* de una forma controlada.

Abstract

This project is focused on developing a software application that is able to extract and store in a *NoSQL* database information from *Twitter*, in particular, information published by *Twitter* accounts that aim to inform about the status of some web services. Also, the application should be able to analyze this information using *Big Data* techniques in order to extract running stats about that services.

As this project does not pretend to perform semantic analysis of *tweets*, it will be developed a new software focused on monitoring web services and publishing tweets about their status in a controlled way.

Índice

Agradecimientos ix

Resumen xi

Abstract xiii

Índice xv

Índice de Tablas xvii

Índice de Figuras xix

1 Introducción 1
1.1 Influencia de Twitter en el mundo 3
1.2 Big Data 4
1.2.1 Las cinco V's del *Big Data* 4
1.2.2 Tecnologías del *Big Data* 5
1.3 Motivación 10
1.4 Objetivos 11

2 Arquitectura 13
2.1 Especificación de requisitos 14
2.1.1 Requisitos funcionales 14
2.1.2 Requisitos no funcionales 14
2.2 Tecnologías del proyecto 14
2.2.1 Base de datos 15
2.2.2 Análisis de los datos 17
2.2.3 Lenguaje de programación 18
2.2.4 Otras tecnologías 19
2.2.5 Resumen de tecnologías y versiones utilizadas 20

3 Implementación 21
3.1 pfc-mon 23
3.1.1 Spring Batch 24
3.1.2 Implementación del *ItemReader* y el *ItemWriter* 26
3.1.3 Configuración del *job* 29
3.1.4 Perfiles, compilación y construcción del módulo con *Maven* 31
3.1.5 Ejecución 33
3.2 pfc-web 35
3.2.1 Spring 36
3.2.2 Liwenx 39
3.2.3 Modelo de datos 44
3.2.4 Estructura en capas 50
3.2.5 Spring Security 76
3.2.6 Despliegue 78
3.3 pfc-analysis 79
3.3.1 Spark 79
3.3.2 Implementación del analizador 80
3.3.3 Ejecución 87

4 Resultados **89**

5 Conclusiones **95**

5.1 Posibles líneas futuras de investigación *96*

Referencias **97**

Índice de Tablas

Tabla 2-1. Tecnologías y versiones utilizadas 20

Tabla 4-1. Estadísticas de los servicios monitorizados 90

Índice de Figuras

Figura 1-1. Evolución de usuarios globales de Internet 2

Figura 1-2. Distribución de base de datos según el teorema CAP 7

Figura 1-3. Diagrama de Venn del principio SVC 9

Figura 2-1. Diagrama de funcionamiento de la aplicación 13

Figura 2-2. Diagrama de funcionalidades de Datastax Enterprise 17

Figura 2-3. Diagrama de funcionalidades de Apache Spark 18

Figura 3-1. Diagrama de funcionamiento de Spring Batch 24

Figura 3-2. Diagrama de módulos de *Spring* 36

Figura 3-3. Diagrama en forma de entidad-relación del modelo de datos 49

Figura 3-4. Esquema de la estructura de la aplicación en capas 50

Figura 3-5. Página inicial de la aplicación 72

Figura 3-6. Twitter OAuth. Paso 1 73

Figura 3-7. Twitter OAuth. Paso 2 73

Figura 3-8. Twitter OAuth. Paso 3 74

Figura 3-9. Página principal de usuario 74

Figura 3-10. Pantalla para añadir servicio a monitorizar 75

Figura 3-11. Página de detalle de servicio monitorizado 75

Figura 3-12. Interfaz web de Spark 87

Figura 4-1. Resultados de los servicios monitorizados 90

Figura 4-2. Estadísticas de la Web del UIT 91

Figura 4-3. Estadísticas de la web de la ETSI 91

Figura 4-4. Estadísticas de la web TenGolf 92

Figura 4-5. Estadísticas de la web del W3C 92

Figura 4-6. Estadísticas de la web del IEEE 93

Figura 4-7. Estadísticas de la *API* de *Spotify* 93

Figura 4-8. Estadísticas de la web Nabumbu 94

1 INTRODUCCIÓN

La informática y las comunicaciones han evolucionado de forma muy rápida en muy poco tiempo. Desde la aparición de los primeros ordenadores personales y su uso común en los hogares hasta hoy han pasado poco más de 30 años.

Aunque la primera computadora data de 1938, el Z1 de K. Zuse, no fue hasta 1947 cuando apareció la primera computadora de propósito general, el ENIAC fabricado por la Universidad de Pensilvania, una máquina de Turing completa que funcionaba con válvulas. En 1953 IBM fabrica su primera computadora a escala industrial, el IBM 650, que procesaba datos mediante tarjetas perforadas. Es a finales de los 70, tras avances como el transistor, los circuitos integrados y el desarrollo de los lenguajes de programación, cuando comienzan a aparecer los primeros ordenadores personales (PC) de la mano de empresas como Apple, Commodore o Atari. Su uso, inicialmente en los hogares como meras herramientas de entretenimiento, se extiende al ámbito profesional en los años 80 con el desarrollo de nuevos softwares, como editores de texto, hojas de cálculo,...

Después de la aparición del PC, el siguiente gran hito en el mundo de la informática y las comunicaciones ha sido la aparición de Internet, concretamente en 1969 cuando se crea el primer enlace entre las universidades de UCLA y Stanford por medio de una línea telefónica conmutada, creando así la primera red interconectada, ARPANET.

En los años 80 se desarrollan e implantan los nuevos protocolos de comunicaciones TCP/IP y en 1989 se integra el modelo de protocolos OSI. Durante esta década surgen nuevas redes en EEUU como NSFNET, propiedad de la NSF (*National Science Foundation*) y cuyo propósito era interconectar todas las universidades norteamericanas y así poder compartir datos y resultados de investigaciones. Puesto que ARPANET deja de funcionar en 1990, es de hecho NSFNET la primera red que hace de columna vertebral o *backbone* de Internet [1].

A principios de los 90 comienzan a producirse las primeras conexiones de otros países a NSFNET, España por ejemplo en julio de 1990. Y en 1995, con la cesión por parte de la NSF de sus redes a distintas empresas proveedoras de servicios de conexión, Internet deja de estar en manos de un país (EEUU) para dar paso al modelo descentralizado de la actualidad.

A finales de 1990 se lanza primer cliente el web, *WorldWideWeb* desarrollado por Tim Berners-Lee [2]. También surge el primer servidor Web. Es de hecho este servicio, el servicio Web, el que ha sido más popular en Internet ya que permite la publicación de documentos en la red con enlaces entre ellos. Se trata por tanto del nacimiento de la *World Wide Web* (WWW). Hoy en día Tim Berners-Lee es director del *Word Wide Web Consortium* (W3C).

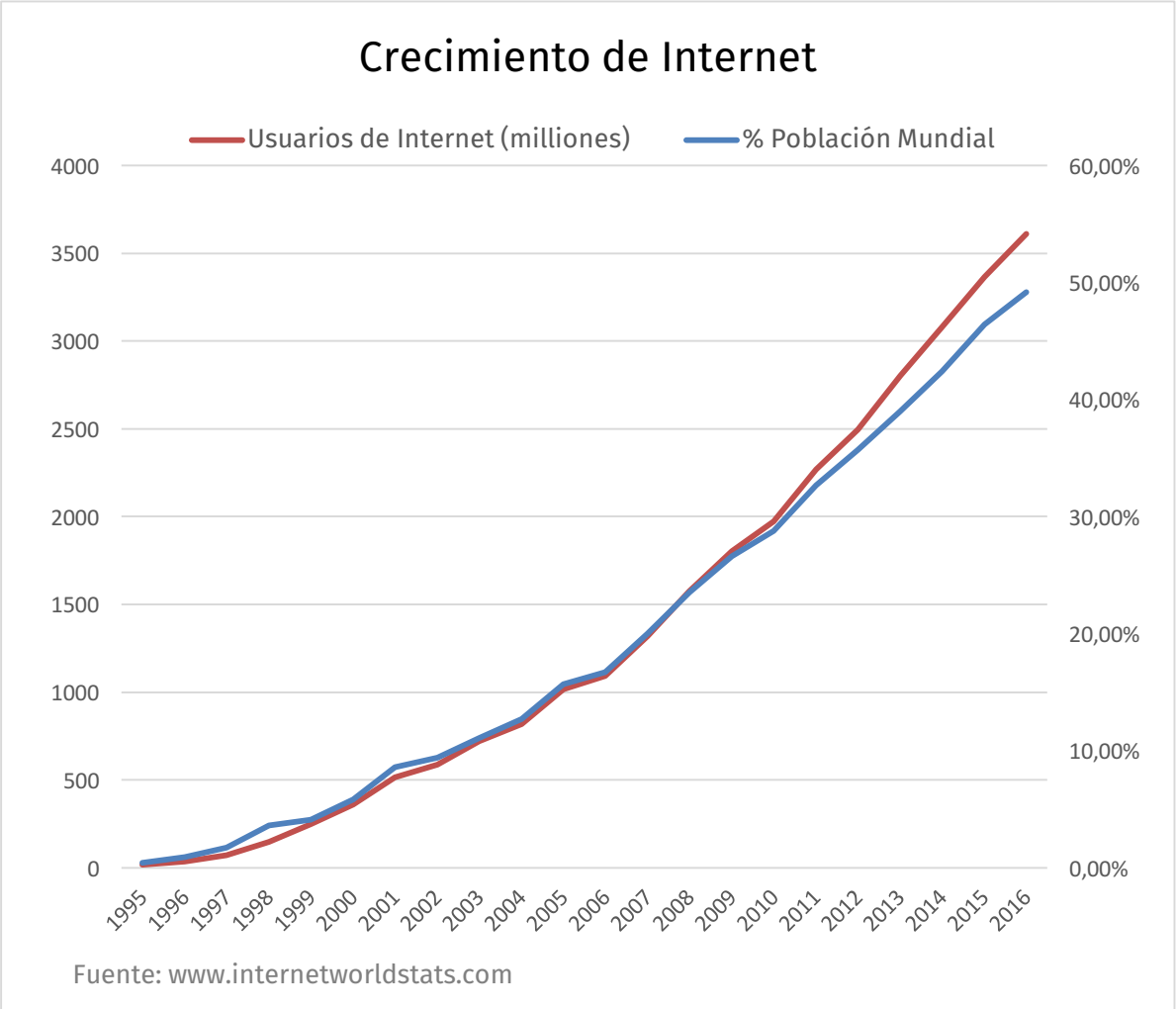

Figura 1-1. Evolución de usuarios globales de Internet

La Figura 1-1 muestra la evolución de los usuarios de internet desde 1995 hasta hoy [3]. Ha pasado de 16 millones de usuarios en diciembre de 1995 a 3.611 millones en junio de 2016, que representan casi el 50% de la población mundial.

Uno de los factores que más ha potenciado su crecimiento en los últimos 15 años ha sido el desarrollo de las conexiones y el incremento de su velocidad. Los últimos pasos han ido encaminados hacia la eliminación de cables con conexiones tipo inalámbricas como el estándar 802.11 (WiFi) y las redes de telefonía móvil. Estas últimas han evolucionado desde los primeros

SMS y conexiones WAP con máximos de 9600bps a los 100Mbps de máximo teórico en las actuales redes LTE (4G).

Además de la eliminación de los cables, otro de los hitos que ha favorecido este crecimiento es el desarrollos de las baterías, que ha dado lugar a la aparición de distintos dispositivos portátiles: ordenadores, teléfonos móviles, videoconsolas,...

Según los datos de la GSMA, en agosto de 2016 el número de dispositivos móviles conectados a nivel mundial ascendía a 7.800 millones, más que personas en el mundo [4]. En 2014, un 84% de los españoles poseía al menos un Smartphone según un estudio de Ametic y Accenture [5].

Como se puede observar, desde su aparición, el uso de Internet se ha ido extendiendo hasta llegar a ser de uso diario y convirtiéndose en una de las principales fuentes de información, comunicación y entretenimiento en la sociedad actual. De hecho ha cambiado incluso la forma en la que nos relacionamos las personas.

Para darnos cuenta de esto último basta nombrar servicios que hace unos años no existían y que a día de hoy los utilizamos con bastante asiduidad en nuestro día a día. Algunos ejemplos son: Facebook, con 1650 mill. de usuarios activos en abril 2016 [6]; Whatsapp, con 1000 mill. de usuarios activos en febrero 2016 [7]; Google, con más de 3500 mill. de búsquedas diarias [8]; Youtube, con más de 1000 mill. de usuarios mensuales que reproducen más de 6000 mill. de horas de vídeo al mes [9]; o Twitter con 313 mill. de usuarios activos en junio de 2016 [10] y al que analizaremos más en profundidad en el siguiente apartado.

1.1 Influencia de Twitter en el mundo

Twitter nace el 21 de marzo de 2006 de la mano de Jack Dorsey, Noah Glass, Biz Stone y Evan Williams en San Francisco, California. Fue lanzada en julio de ese mismo año y se trata de una red de *microblogging* donde los usuarios pueden enviar mensajes con una longitud máxima de 140 caracteres, *tweets*, que se muestran en su página principal. Los usuarios pueden suscribirse a los tweets de otros usuarios, *seguirlos*, convirtiéndose en *seguidores* de dichos usuarios.

Nueve meses más tarde de su lanzamiento, en 2007, Twitter coloca pantallas en el *South by Southwest* celebrado anualmente en Austin, Texas. Los asistentes al evento podían enviar sus mensajes durante las conferencias y estos aparecían en dichas pantallas. La plataforma aumenta considerablemente su popularidad durante este evento [11].

En el accidente del US Airways en el río Hudson de Nueva York en enero de 2009 diferentes tuits enviados por testigos y pasajeros fueron acompañados por enlaces con fotos de lo ocurrido, generando información antes que los medios tradicionales. El más famoso fue la foto Janis Krum tan sólo 10 minutos después del aterrizaje forzoso con el mensaje *Hay un avión en el Hudson* [12]. Esto puso en evidencia el potencial de Twitter como fuente de noticias.

Otro de los aspectos importantes de Twitter es el poder de movilización que, junto con otras redes, ha demostrado. Irán tras las elecciones presidenciales de 2009, las primaveras árabes de 2011 en Túnez, Egipto, Libia, Siria, Yemen, Argelia, Jordania, Omán,..., el 15-M en España, la

Revolución de los Paraguas de 2014 en Hong Kong, Guatemala en 2015, Venezuela y Brasil en 2016, en todas estas protestas Twitter ha servido como vía de comunicación para cerrar quedadas masivas y denunciar hechos a nivel mundial.

Durante estos 10 primeros años de vida, Twitter ha revolucionado el mundo de la comunicación. Según Riyaad Minty, que estuvo a cargo de las redes sociales en Al Jazeera, Twitter aporta un valor extra a la información, la confianza. Afirma que lejos de pensar que lo que cuelgue un particular tiene menos credibilidad, la gente en Twitter pincha un enlace, el que sea, porque se lo pone la gente en la que cree, sea un amigo, una institución afín, una personalidad que les gusta... Se fían de ellos y de su filtro y eso imprime un valor a la información que no tienen los medios convencionales [13].

En la actualidad su uso es bastante extendido en multitud de situaciones, desde eventos en directo de toda índole: deportivos, políticos, musicales, ... ; pasando por el seguimiento de programas de TV, intercambio de opinión sobre noticias de actualidad, humor, o simplemente charlas entre amigos. Todo ello generan 500 millones de tuits diarios.

1.2 Big Data

Big Data es un término que hace referencia al almacenamiento de grandes cantidades de datos y a los procedimientos usados para administrarlos y procesarlos en un tiempo razonable, cuando el software y las técnicas tradicionales son insuficientes.

Para que un conjunto de datos sea considerado dentro del ámbito del *Big Data* ha de cumplir una o más características propias del mismo.

1.2.1 Las cinco V's del *Big Data*

Las características del *Big Data* o cinco V's son las siguientes [14]:

- **Volumen:** grandes volúmenes de datos requieren un tipo de almacenamiento y un procesamiento distinto a los tradicionales. En el mundo se generan alrededor de 2,5 EBs[1] de datos cada día. Por hacer una comparativa, la Biblioteca del Congreso de Estados Unidos alberga hoy día 300 TBs de datos, casi un millón de veces menos. Fuentes de datos típicas que generan gran cantidad de datos son las transacciones online, experimentos e investigaciones científicas, sensores o distintas redes sociales como Facebook o Twitter.
- **Velocidad:** en entornos *Big Data* la información puede llegar a grandes velocidades y enormes conjuntos de datos pueden ser acumulados en breves periodos de tiempo. Procesar esta cantidad de datos en un tiempo razonable es uno de los objetivos del *Big Data*. Por poner algunos ejemplos, cada minuto se generan 350.000 tweets, se suben más de 300 horas de vídeo a Youtube o se envían 171 millones de emails.

[1] 1 exabyte (EB) = 2^{60} bytes (B)

- **Variedad:** se refiere a los tipos de formatos que deben ser soportado por las soluciones *Big Data*. Los datos pueden ser estructurados (formatos tabulares), no estructurados (imágenes, vídeos, ...), o semi-estructurados (emails, JSON, XML,...).
- **Veracidad:** hace alusión a la calidad o fidelidad de los datos. En este sentido, el procesamiento de los datos debe tener en cuenta los datos no relevantes o inválidos y eliminar ese *ruido*.
- **Valor:** se define en términos de utilidad de los datos para los consumidores de los mismos. Esta característica está relacionada con la veracidad ya que a mayor fidelidad de los datos, más valor poseen. Por otro lado, también está relacionada con el tiempo en que los datos tardan en procesarse. Mientras más se tarda en convertir los datos en información útil, menor es su valor.

1.2.2 Tecnologías del *Big Data*

Existen distintas tecnologías asociadas al *Big Data*. Éstas las podemos agrupar en dos grandes grupos: almacenamiento y análisis.

1.2.2.1 Almacenamiento

En este grupo de tecnologías tenemos principalmente las bases de datos *NoSQL*, cuyo término viene de *Not Only SQL*, y se refiere a que estos sistemas no cumplen con el esquema entidad-relación, es decir, al contrario que las bases de datos relacionales, no imponen una estructura de datos en forma de tablas y relaciones entre ellas sino que proveen un esquema mucho más flexible [14].

Las bases de datos NoSQL se pueden clasificar en cuatro grandes grupos:

- **Almacenamiento clave-valor**: los datos se almacenan en formato de mapas o diccionario de datos, donde se accede al dato a partir de una clave única. Los datos son aislados e independientes entre sí. Al no tener una estructura de datos clara no se requiere un formateo de los datos especialmente estricto. Ejemplos de este tipo de base de datos son *DynamoDB* o *Redis*.
- **Almacenamiento orientado a documento:** este caso es muy parecido al anterior salvo en los valores que guardan. Si en el almacenamiento clave-valor no se requería una estructura concreta de los datos, aquí se guardan datos semi-estructurados, llamados documentos y pueden estar formateados por ejemplo en *XML*, *JSON*, *Binary JSON*,... Ejemplos de este tipo de base de datos son *MongoDB o CouchDB*.
- **Almacenamiento orientado a grafos:** se basan en la teoría de grafos, donde se establece que la información son los nodos y las relaciones entre la información son las aristas que unen a los nodos. Los nodos pueden contener objetos, variables y atributos diferentes de uno a otro y las operaciones de *join* se sustituyen por recorridos a través del grafo. Ejemplo de este tipo de base de datos son *Neo4j, GraphDB* o *Infinite Graph*.

- **Almacenamiento orientado a columnas:** funcionan parecido a las bases de datos relacionales pero almacenando columnas de datos en lugar de registros, dando lugar a las familias de columnas. Cada fila, identificada por una clave única, consiste en múltiples familias de columnas y pueden tener a su vez distintos grupos de columnas, permitiendo flexibilidad en el formato de los datos. Ejemplos de este tipo de base de datos son *BigTable*, *HBase* o *Cassandra*.

Las bases de datos *NoSQL* están generalmente pensadas para trabajar como *clusters de nodos*. Cada nodo está interconectado con los demás funcionando como un todo. Debido a que los volúmenes de datos son grandes, normalmente los nodos contienen sólo una parte de los datos. Típicamente, el rango de claves se divide en un número de partes y cada nodo almacena sólo los registros de datos correspondiente a su rango de claves. Esto se llama *sharding*.

Con el objetivo de ser tolerante a fallos, los nodos suelen implementar también la *replicación* de los datos. En este caso, cada nodo, además de almacenar los registros correspondiente a su rango de claves, almacena de forma redundante los registros correspondientes a otros nodos. En el caso de caída de algún nodo, no habrá pérdida de datos y el sistema podrá seguir funcionando sin interrupción del servicio.

Cuando hablamos de *cluster de nodos* hablamos de sistemas distribuidos. Es interesante en este caso hablar del **teorema CAP o de Brewer** [15], que dice que en sistemas de almacenamiento distribuidos es imposible garantizar a la vez consistencia, disponibilidad y tolerancia a particiones.

- **Consistencia (C)**: al realizar una consulta o inserción siempre se tiene que recibir la misma información, con independencia del nodo que procese la petición.
- **Disponibilidad (A)**: que todos los clientes puedan leer y escribir, aunque se haya caído uno de los nodos.
- **Tolerancia a particiones (P)**: los sistemas distribuidos pueden estar divididos en particiones (generalmente de forma geográfica). Esta condición implica que el sistema tiene que seguir funcionando aunque existan fallos o caídas parciales que dividan el sistema.

En la Figura 1-2 se pueden apreciar cómo se reparten las distintas bases de datos según dicho teorema.

- **AP**: garantizan disponibilidad y tolerancia a particiones, pero no la consistencia, al menos de forma total. Algunas de ellas consiguen una consistencia parcial a través de la replicación y la verificación.
- **CP**: garantizan consistencia y tolerancia a particiones. Para lograr la consistencia y replicar los datos a través de los nodos, sacrifican la disponibilidad.

- **CA**: garantizan consistencia y disponibilidad, pero tienen problemas con la tolerancia a particiones. Este problema lo suelen gestionar replicando los datos.

Hay que tener en cuenta que la clasificación de las distintas bases de datos no es definitiva ya que algunas permiten ajustar su comportamiento mediante configuración.

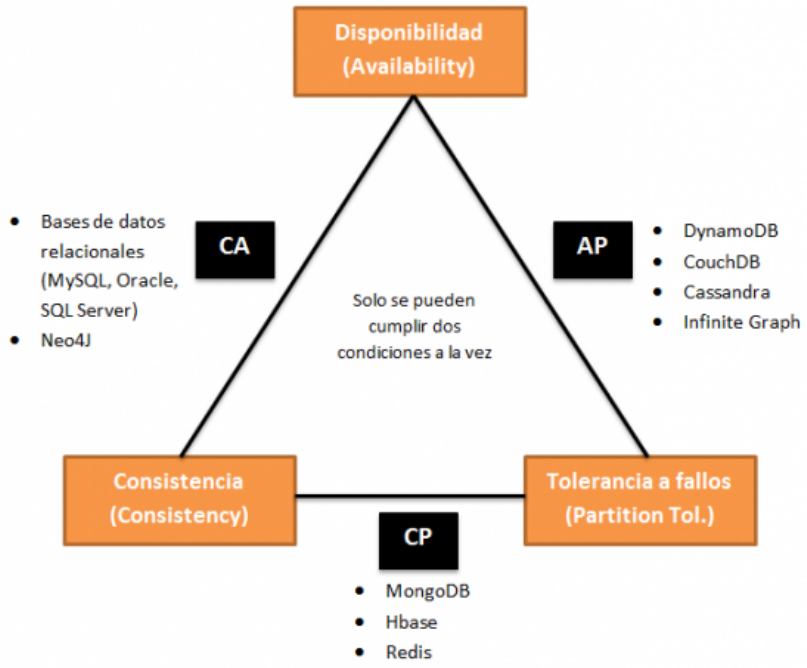

Figura 1-2. Distribución de base de datos según el teorema CAP

1.2.2.2 Análisis

El procesamiento y análisis de volúmenes de datos del tipo *Big Data* requiere soluciones distintas a las tradicionales. Estas soluciones permiten hacer análisis de distintos tipos: cuantitativos, cualitativos, minería de datos, análisis estadísticos (tests A/B, correlación, regresión), *machine learning*, análisis semánticos y visualizar los datos de distintas maneras (mapas de calor, series de datos, grafos, mapeo espacial,...)

Dependiendo de cómo se realice el procesamiento de los datos distinguimos procesamiento de datos por lotes y procesamiento en tiempo real.

1.2.2.2.1 Procesamiento por lotes

Los datos son procesados en lotes posteriormente a su generación, obtenidos de algún sistema de almacenamiento. El tiempo de respuesta puede variar desde minutos a horas. Este tipo de procesamiento se centra en las características *velocidad* y *variedad* de las cinco V's del *Big Data*. La mayoría de procesamientos de datos *Big Data* son de este tipo: es fácil de configurar y de bajo coste comparado con el procesamiento de datos en tiempo real.

Aquí entran en juego técnicas como el procesamiento de datos en paralelo, donde las tareas son divididas en subtareas y se ejecutan en paralelo por distintos procesadores o nodos de análisis.

Una de las técnicas de análisis más extendidas es *MapReduce*, desarrollada por *Google* en su algoritmo para calcular el *PageRank* y posteriormente implementada como software libre por Apache dando lugar a *Hadoop* [16].

El concepto *MapReduce* se basa en separar las tareas de procesado en dos tipos de operaciones *Map* y *Reduce*. Estas operaciones están definidas partiendo de datos estructurados en tuplas (clave, valor). La función *Map* se encarga de mapear los datos, asignándole a cada valor una clave, y la función *Reduce*, toma la salida de la operación *Map*, y se encarga de procesar los conjuntos de valores que tienen una misma clave.

El nodo *master*, coordinador de la tarea, se encarga en cada paso de dividir las tareas en tareas de menor entidad y de enviarlas a los nodos *workers*, que a su vez pueden subdividirlas. Finalmente los *workers* procesan los datos y devuelven la respuesta al nodo *master*. Éste recopila los datos y pasa al siguiente paso, volviendo a dividir la tarea principal en subtareas y enviándosela a los nodos *workers* nuevamente.

1.2.2.2.2 Procesamiento en tiempo real

En este tipo de procesos, los datos son procesados en memoria en cuanto son capturados antes de ser guardados en algún sistema de almacenamiento. Los tiempos de procesamiento suelen ser menores a un minuto, normalmente por debajo del segundo. Este tipo de procesamiento de datos se centra principalmente en la característica *velocidad* de las cinco V's del *Big Data*.

También se le conoce como procesamiento de eventos, cuando los datos llegan a intervalos, o de flujo, cuando éstos llegan continuamente. Normalmente los paquetes de datos son de pequeño tamaño pero su naturaleza continua genera grandes volúmenes.

Relacionado con el procesamiento en tiempo real existe un principio llamado **principio SVC** [14], que afirma que los sistemas de procesamiento de datos distribuidos sólo pueden ser diseñados para soportar dos de los siguientes requisitos:

- **Velocidad (S):** se refiere a cómo de rápido son procesados los datos una vez generados.
- **Consistencia (C):** se refiere a la precisión y solidez de los resultados. Son precisos si se ajustan al valor correcto mientras que son sólidos si los resultados son cercanos unos a otros. Un sistema más consistente hará uso de más datos disponibles.
- **Volumen (V)**: se refiere a la cantidad de datos que pueden ser procesados. Procesar grandes volúmenes de datos asegurando consistencia y velocidad no es posible.

La Figura 1-3 muestra un diagrama de Venn que resume el principio SVC. Si se requiere velocidad (S) y consistencia (C), no es posible procesar grandes volúmenes de datos ya que resultaría en una velocidad de procesamiento baja, en ese caso sería un sistema CV. Por último, si se busca velocidad (S) y procesar un alto volumen de datos (V) normalmente esto implica muestrear los datos, lo que afectaría a la consistencia (C).

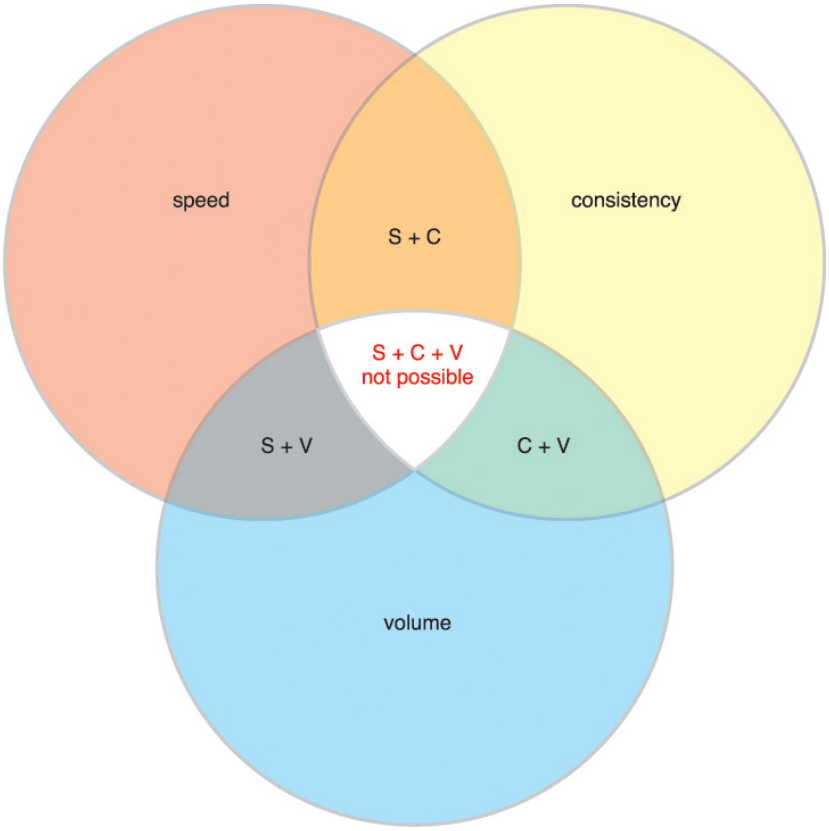

Figura 1-3. Diagrama de Venn del principio SVC

Otra clasificación que se puede hacer dentro de este tipo de sistemas es la siguiente:

- **Procesamiento de flujo de eventos (ESP):** existe un flujo de eventos continuo, generalmente de una única fuente y ordenados en el tiempo, y son procesados continuamente.
- **Procesamiento de eventos complejos (CEP):** existe una serie de eventos provenientes de distintas fuentes y llegando a intervalos irregulares. Los eventos son analizados buscando patrones e iniciando acciones. Aquí se aplican algoritmos basados en reglas y técnicas estadísticas. Normalmente, este tipo de procesamiento implica subprocesos ESP.

MapReduce no suele ser aconsejable para procesamiento de datos en tiempo real por diversas razones. Una de ellas es que no puede procesar datos de manera incremental, sino que requiere que todos los datos estén disponibles antes de la ejecución de las tareas. De hecho, generalmente la tarea *Reduce* necesita que todas las tareas *Map* hayan terminado antes de poder comenzar. Al procesar datos en tiempo real, los datos suelen estar incompletos y llegando continuamente.

Aun así es posible encontrar escenarios y configurar soluciones *MapReduce* para este tipo de procesado. Normalmente no son situaciones puramente en tiempo real (*hard-realtime*) sino cercanas al mismo (*near-realtime).*

Para solventar estos problemas surgen proyectos *Apache* como *Spark* [17], *Storm* [18] y *Tez* [19], que proveen soluciones reales de procesamiento en tiempo real.

Apache Storm fue desarrollado por *Backtype* y posteriormente adquirido por *Twitter*. Su estructura es similar al de *MapReduce* con la diferencia de que los datos son procesados en tiempo real de forma individual en lugar de por lotes. Es adecuado para procesamientos CEP, al igual que *Apache Tez*, que corre sobre *Hadoop YARN* [20].

Spark fue desarrollado en 2009, en la Universidad de California, Berkeley, y posteriormente donado a la fundación Apache en 2013. *Spark* nace en respuesta a las limitaciones del modelo de procesamiento *MapReduce* e introduce el concepto de *Resilient Distributed Dataset* (RDD), un multi-conjunto de datos distribuidos sobre un *cluster* de nodos y que es mantenido de manera tolerante a fallos.

RDD permite tratar todos los datos a procesar como un único conjunto de datos, facilitando el acceso cíclico a los mismos en las distintas operaciones. La baja latencia de estas operaciones hacen que la velocidad de procesamiento de *Spark* sea del orden de 100 veces mayor que *Hadoop* cuando se ejecuta en memoria y 10 veces más rápida si los datos tienen que pasar a disco durante su procesamiento. Es adecuado para procesamientos ESP.

1.3 Motivación

Como se ha podido observar, cada vez es mayor el volumen de datos generados día tras día. A su vez, las técnicas de *Big Data* nos permiten procesarlos y analizarlos en un tiempo razonable, extrayendo información que nos puede ser útil para distintos propósitos.

Por otro lado, como hemos visto, Twitter se ha convertido en los últimos años en una herramienta de comunicación donde cada día se genera mucha información de diversa índole. Dicha información procesada puede darnos muchas pistas sobre lo que está pasando en el mundo. Biz Stone dice en su libro *Cosas que me contó un pajarito* [21], que durante las primaveras árabes prácticamente podían predecir cuál sería la siguiente revolución según la evolución del volumen de tuits de una zona determinada.

Por otro lado, no sólo las personas tuitean, sino también robots (*bots*) que tras ciertos patrones de comportamiento escondidos tras sus líneas de código, son capaces de publicar información en Twitter. Es común ver *bots* recopilando y publicando tuits de noticias de un determinado tema, incluso interactuando con los usuarios respondiendo con frases predeterminadas, o simplemente informando sobre el estado de un servicio online.

Por la facilidad a la hora de procesar dichos mensajes, nos vamos a centrar en estos últimos, los *bots* que publican información de estado de servicios online.

1.4 Objetivos

El objetivo de este proyecto es la implementación de un software que, dadas determinadas cuentas que publican información sobre el estado de servicios online, sea capaz de recopilar dichos tuits, procesarlos con técnicas de *Big Data* y extraer estadísticas de funcionamiento del servicio.

Con el fin de generar un entorno controlado, también se implementarán *bots* capaces de monitorizar un servicio web regularmente, lanzando un tuit con información acerca de cada muestra tomada.

2 ARQUITECTURA

Dados los objetivos del proyecto, se ha diseñado un esquema de funcionamiento de la aplicación como el de la Figura 2-1.

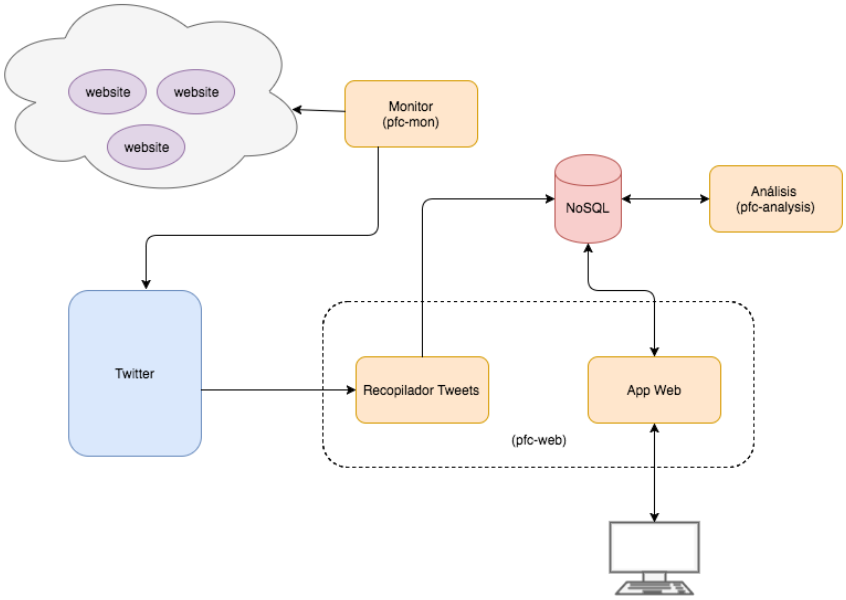

Figura 2-1. Diagrama de funcionamiento de la aplicación

Se parte de una serie de servicios o portales web que quieren ser monitorizados. El módulo *Monitor* se encarga de realizar peticiones a sus respectivas URL de forma periódica y de publicar un tuit por cada una de las muestras tomadas en distintas cuentas de *Twitter*, una por cada servicio. Estas cuentas son típicamente cuentas de estado de cada uno de dichos servicios. El módulo *Monitor* se encarga pues de simular una serie de *bots* facilitando un entorno controlado como punto de partida del proyecto.

Desde el punto de vista del usuario, el núcleo del proyecto es la propia aplicación web, módulo *App Web*. Los usuarios podrán crearse su propia cuenta y elegir qué cuentas de *Twitter* desean monitorizar. Partiendo de estas cuentas, el módulo *Recopilador de Tweets* se encarga de recopilar todos los tuits que se van publicando en las mismas. Éstos se almacenarán en una base de datos *NoSQL* que a su vez sirve de soporte de almacenamiento para todo el proyecto.

El módulo *Análisis* se encargará de procesar de forma periódica los tuits extraídos mediante técnicas de *Big Data*. El resultado de este análisis quedará almacenado en la base de datos y listo para que la *App Web* pueda mostrarlo a sus usuarios.

2.1 Especificación de requisitos

Con el objetivo de sentar las de bases y especificaciones del proyecto, a continuación se detallan los requisitos funcionales y no funcionales que el sistema debe cumplir.

2.1.1 Requisitos funcionales

1. El módulo *Monitor* será capaz de monitorizar cualquier servicio que funcionen sobre el protocolo HTTP.
2. El módulo *Monitor* publicará un tuit por cada muestra obtenida con la siguiente información: disponibilidad, latencia y código de estado HTTP recibido.
3. El módulo *Monitor* publicará los tuits de cada servicio monitorizado en distintas cuentas de *Twitter*, una para cada servicio.
4. Los usuarios podrán acceder a la aplicación mediante un navegador web.
5. Los usuarios podrán crearse una cuenta en la aplicación y acceder a la misma utilizando su propia cuenta de *Twitter*.
6. Los usuarios podrán añadir diversas cuentas de *Twitter* para monitorizar.
7. Para cada una de las cuentas monitorizadas, la aplicación mostrará los tuits recopilados.
8. Además, para cada una de las cuentas monitorizadas, la aplicación mostrará estadísticas de funcionamiento del servicio, típicamente disponibilidad y latencia, permitiendo visualizar estos parámetros por año, mes, semana, día y minuto.
9. Por último, los usuarios dispondrán de un panel de control donde se muestren datos globales de funcionamiento de cada servicio.

2.1.2 Requisitos no funcionales

1. El sistema debe capaz de almacenar grandes volúmenes de información y ser escalable en cuanto a almacenamiento.
2. El sistema debe ser capaz de procesar grandes volúmenes de información y ser escalable en cuanto a procesamiento de datos.
3. El sistema debe ser capaz de funcionar en cualquier plataforma *Unix* (*Linux, MacOS,...)*

2.2 Tecnologías del proyecto

Los requisitos del proyectos establecen una serie de restricciones que nos harán decantarnos por unas tecnologías en lugar de otras.

2.2.1 Base de datos

Dada la naturaleza del proyecto, los datos involucrados y el requisito no funcional 1, es imperativo el uso de una base de datos *NoSQL*. Dentro de este tipo de base de datos, podemos descartar las bases de datos orientadas a grafos por la propia naturaleza de los datos. Las bases de datos tipo clave-valor, como *Redis*, se suelen usar con estructuras de datos muy simples en aplicaciones como cacheo de datos, contadores en tiempo real (i.e. bolsa), análisis en tiempo real y suelen ser un apoyo a otros tipos de bases de datos [22].

Dicho esto, nos quedan las bases de datos orientadas a documentos, como *MongoDB*, y las orientadas a columnas, como *Cassandra*. Veamos algunas de sus características.

2.2.1.1 MongoDB

Está escrita en C++, es multiplataforma y se distribuye con licencia de código abierto. Es una base de datos orientada a documentos, principalmente JSON. Empezó a desarrollarse en 2007 por la compañía *10gen*, ahora *MongoDB Inc.*, y es utilizada por empresas como *MTV Network*, *Craiglist* o *Foursquare*.

Dentro de sus características podemos destacar:

- Replicación maestro/esclavo.
- Permite el *sharding.*
- Las consultas a base de datos son expresiones *javascript*.
- Capaz de ejecutar funciones *javascript* en el servidor.
- Dispone de diversos motores de almacenamiento con diferentes características de rendimiento.
- *Framework* que permite consultas de agregación similares a las *GROUP BY* de *SQL*.
- Indexación
- Validación de documentos
- Búsqueda de texto integradas

De entre los principales inconvenientes de *MongoDB* podemos nombrar los siguientes:

- Problemas de consistencia ya que las lecturas estrictamente consistente ven versiones anteriores de documentos [23]
- Bloqueo a nivel de documento en cada operación de escritura.
- Presenta mermas de rendimiento cuando los datos superan los 100GB [24].

Es adecuada cuando se necesitan consultas dinámicas; si se prefieren definir índices en lugar de funciones *map/reduce*; si se requiere un buen rendimiento en bases de datos grandes, principalmente en lectura.

2.2.1.2 Cassandra

Fue desarrollada inicialmente en *Facebook* por Avinash Lakshman y Prashant Malik en 2007. En 2009 se transforma en proyecto Apache. Es una base de datos orientada a columnas escrita en *Java*. En la actualidad es utilizada por empresas como *Twitter*, *Facebook*, *Apple* o *Netfix*. Sus principales características son:

- Modelo descentralizado donde todos los nodos son considerados como iguales, no hay nodos maestros.
- Soporta *sharding* y *replicación* y funciona muy bien en replicaciones entre distintos *datacenters*.
- Es altamente escalable ya que el rendimiento en lectura y escritura aumenta linealmente conforme se van añadiendo nodos.
- Las escrituras son mucho más rápidas que las lecturas y no existe el bloqueo a nivel de fila. Cada columna es independiente.
- Proporciona CQL3, un lenguaje similar a SQL, con ciertas limitaciones, que facilita las consultas. Las consultas pueden ser por clave, por rango de clave o por índices secundarios.
- Consistencia configurable en las lecturas y escrituras, que junto con los ajustes de replicación, permiten mover el sistema por las distintas aristas del gráfico del teorema CAP (ver Figura 1-2) haciendo que el sistema distribuido pueda ser CA, AP o CP según el caso.
- Soporta *MapReduce* ya que integrada *Apache Hadoop*.

Entre sus desventajas se pueden destacar las siguientes:

- No permite realizar consultas con *JOINs* o agregaciones.
- Los índices secundarios tiene sus limitaciones

Su uso es adecuado cuando se tienen enormes cantidades de datos que no caben en un único servidor o cuando la velocidad en escritura es una necesidad, ya que es una de las más rápidas del mercado en escritura. Un ejemplo típico de uso es en analíticas web, donde se contabilizan visitas por hora, por navegador, por IP, … o datos provenientes de un conjunto de sensores.

2.2.1.3 Solución elegida: Datastax Enterprise

En vista de lo expuesto en los apartados anteriores, parece que *Cassandra* puede ser una muy buena solución para este proyecto ya que sus características cubren sus necesidades y es adecuada para los objetivos del proyecto. Además, ya incorpora herramientas de análisis de datos como *Hadoop*. Sin embargo, existe una solución *Cassandra* muy interesante: *Datastax Enterprise* [25].

Esta solución toma como base *Cassandra* e incorpora sobre ésta una serie de capas que le dotan de una serie de capacidades extras, como se puede ver en la Figura 2-2. Entre ellas podemos destacar que es capaz de soportar los cuatro tipos de bases de datos *NoSQL*, integra el motor de indexación y búsquedas *Apache Solr* [25] y diversas herramientas para el análisis de los datos como *Hadoop* o *Spark*.

Por todo ello, el sistema de base de datos elegido es *Datastax Enterprise*. La versión elegida es la última estable a la hora de comenzar el proyecto, la 4.8.6, que a su vez incluye:

- Apache Cassandra 2.1
- Apache Hadoop 1.0.4
- Apache Spark 1.4

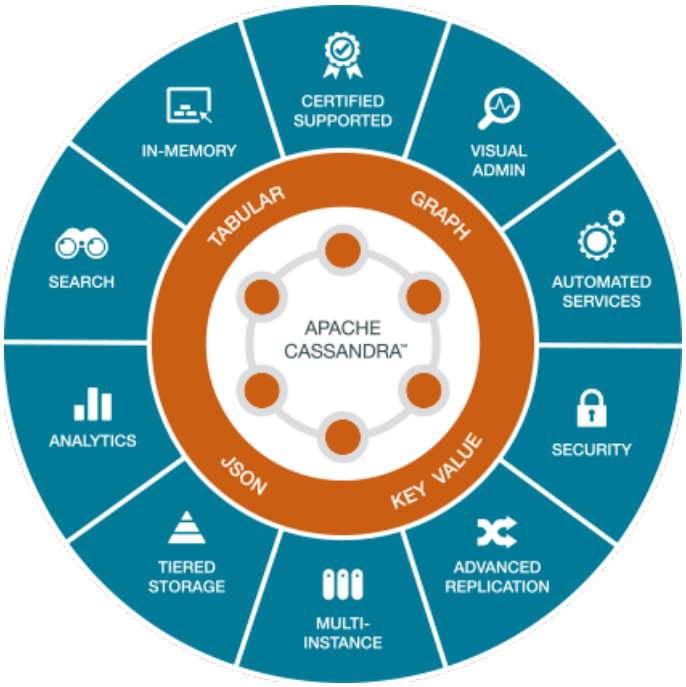

Figura 2-2. Diagrama de funcionalidades de Datastax Enterprise

2.2.2 Análisis de los datos

Dada la tipología del proyecto, el tipo de análisis que se va a realizar es un procesamiento por lotes. Los *tuits* almacenados por el módulo *Recopilador de Tweets* serán procesados para

analizar la media de disponibilidad y latencia de cada servicio en cada intervalo de tiempo y los resultados serán de nuevo almacenados en la base de datos, quedando disponibles para ser consultados por los usuarios mediante la aplicación web.

Para este análisis nos bastaría cualquier tecnología que implemente *MapReduce*, la más popular es *Hadoop*, que además está integrado en *Cassandra*. Sin embargo, al utilizar *Datastax Enterprise*, nos encontramos que también nos ofrece *Spark*, que aunque lo hemos visto dentro del procesamiento de datos en tiempo real, realmente se trata de un motor de procesamiento de datos de ámbito general con distintas librerías para procesamiento de datos en tiempo real (*streaming)*, *machine learning*, procesamiento de grafos o de datos almacenados en bases de datos SQL (ver Figura 2-3).

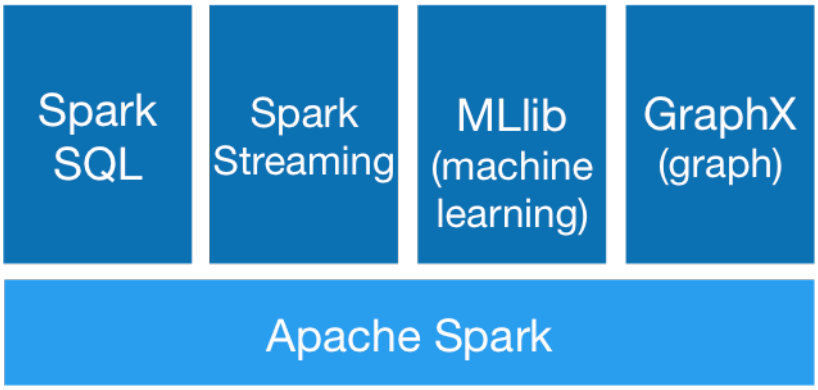

Figura 2-3. Diagrama de funcionalidades de Apache Spark

Teniendo en cuenta que *Spark* es capaz de procesar datos hasta 100 veces más rápido que *Hadoop*, éste se sitúa como el mejor candidato para realizar nuestros análisis. Lo utilizaremos en la versión proporcionada por *Datastax Enterprise*, la 1.4.

2.2.3 Lenguaje de programación

A la hora de elegir un lenguaje de programación existe mucha controversia. Es fácil encontrar fanáticos de uno u otro lenguaje que afirman que el suyo es el Santo Grial de todos los lenguajes. La realidad es bien distinta. Cada lenguaje nace bajo unas circunstancias y para cubrir unas necesidades concretas. Aunque vayan evolucionando y aun siendo de propósito general, cada uno tiene unas características propias y puede resultar más adecuado que otros en según qué casos. Lo más sensato es ver un lenguaje de programación como un medio y nunca como un fin en sí mismo.

Llegados a este punto, las únicas restricciones que tenemos es a la hora de realizar el análisis de los datos, ya que *Spark* nos permite escribir las tareas en *Java, Scala,* o *Python*. En cualquier caso, esto no implica que no podamos construir los otros módulos en otros lenguajes como *PHP*, *C#*, *Ruby*, *C++*... *Datastax Enterprise* proporciona *drivers* de conexión para todos ellos, todos

pueden correr en sistemas *Unix* y son ampliamente utilizados para desarrollar aplicaciones web, por lo que en principio cumplirían todos los requisitos.

Teniendo en cuenta el conocimiento de otros *frameworks* y librerías que nos facilitarán la implementación del resto de módulos, y que veremos a continuación, lo más adecuado es usar *Java*. Por otro lado, buscando el máximo reaprovechamiento del código, lo utilizaremos en todos los módulos del proyecto. La versión a utilizar será la última estable, la 1.8.

2.2.4 Otras tecnologías

En este apartado se detallan el resto de tecnologías a utilizar.

- **Apache Maven**: software de gestión de proyectos que permite gestionar las distintas dependencias, construcción, *reporting* y documentación del proyecto desde un fichero *XML* [26].
- **Apache Tomcat:** es el contenedor de aplicaciones Java más popular [28].
- **Apache HTTP Server:** se trata del servidor web más extendido. Será utilizado únicamente para procesar las peticiones de elementos estáticos de la aplicación web, pasando las peticiones dinámicas a *Tomcat*. Esta configuración es típica en aplicaciones web *Java* [29].
- **Spring:** se trata de un *framework* para el desarrollo de aplicaciones facilitando el desarrollo de las mismas. Entre las funcionalidades que proporciona podemos destacar la inversión del control o inyección de dependencias, programación orientada a aspectos, modelo vista-controlador, acceso a datos, acceso remoto RPC, autenticación y autorización y procesamiento por lotes. Estas dos últimas funcionalidades las proporcionan los módulos **spring-security** y **spring-batch** y se utilizarán en el módulo *Monitor* y la *App Web* respectivamente [30].
- **Freemarker:** se trata de un motor de plantillas basado en *Java*. Aunque es de propósito general, es muy utilizado para generar código *HTML* en aplicaciones web.
- **Liwenx:** se trata de un motor ligero de portales web desarrollado por el autor de este proyecto en 2010 y publicado bajo licencia software libre. Utiliza *Spring y Freemarker,* y permite configurar las distintas páginas de una aplicación web y sus componentes fácilmente a través de ficheros *XML* [31].
- **Apache HTTP Client:** se trata de un cliente *HTTP* que permite consultar aplicaciones publicadas bajo dicho protocolo y procesarlas. Se utilizará en el módulo *Monitor.*
- **Twitter4j:** librería no oficial de *Twitter* que permite interactuar con sus servicios web de forma sencilla.
- **Eclipse IDE:** se utilizará como entorno de desarrollo. Es uno de los más populares para desarrollar aplicaciones *Java*.

2.2.5 Resumen de tecnologías y versiones utilizadas

Componente	Versión
Datastax Enterprise	4.8.6
Apache Cassandra	2.1.13.1218
Apache Spark	1.4.1.1
Java SE Development Kit	1.8.0_51
Apache Maven	3.2.5
Apache Tomcat	7.0.61
Apache HTTP Server	2.4.18
Spring Framework	4.3.0
Spring Batch	3.0.7
Spring Security	4.1.0
Freemarker	2.3.20
Liwenx	0.0.25
Apache HTTP Client	4.3.2
Twitter4j	4.0.4
Eclipse IDE	4.4.2

Tabla 2-1. Tecnologías y versiones utilizadas

3 IMPLEMENTACIÓN

Como hemos visto en la sección anterior el proyecto se divide en cuatro partes: *Monitor, Recopilador de Tweets*, *App Web* y *Análisis*. En esta sección veremos los detalles de implementación de cada una de ellas. Antes de entrar a verlos en profundidad es necesario aclarar por qué se ha divido la implementación en tres partes y no cuatro.

El *Recopilador de Tweets* se encarga de extraer los tuits de cada una de las cuentas monitorizadas por los usuarios: lee de la base de datos las cuentas a monitorizar de forma periódica y actualiza con los nuevos tuits si los hubiera. Sin embargo, aparte de este *trigger* periódico, existe otro momento para actualizar las cuentas, cuando un usuario añade una nueva cuenta a monitorizar. Para implementar una buena experiencia de usuario es deseable que se empiecen a recopilar tuits en ese mismo instante y no esperar a la siguiente actualización periódica. Esto hace que la relación entre *Recopilador de Tweets* y la *App Web* sea muy estrecha y por ello se ha decidido implementar ambos en el mismo módulo.

Tal y como se comentó en la sección anterior, se ha utilizado *Apache Maven* como gestor del proyecto. Se ha implementado un único proyecto *Maven* con tres módulos:

- pfc-mon
- pfc-web
- pfc-analysis

En la Figura 2-1 se pueden identificar estos tres módulos fácilmente. Se puede observar cómo *pfc-web* incluye tanto el *Recopilador de Tweets* como la *App Web*.

La forma que tiene *Maven* de describir un proyecto es mediante un fichero *XML* llamado *Project Object Model*, el fichero *pom.xml*, que se incluye tanto en el proyecto principal como en cada uno de los módulos.

Del fichero *pom.xml* del proyecto principal podemos destacar las siguientes partes:

Nombre del proyecto (artefacto), grupo y versión

```
    <groupId>me.alexguerra</groupId>
    <artifactId>pfc</artifactId>
    <version>0.0.1-SNAPSHOT</version>
```

Módulos de los que está compuesto el proyecto

```
<modules>
  <module>pfc-web</module>
  <module>pfc-mon</module>
  <module>pfc-analysis</module>
</modules>
```

Sistema de control de versiones

En este caso se utiliza el sistema de control de versiones distribuido *git* [32] con *Bitbucket* como repositorio remoto.

```
<scm>
  <connection>scm:git:git@bitbucket.org:xelagc/pfc.git</connection>
  <developerConnection>scm:git:git@bitbucket.org:xelagc/pfc.git</developerConnection>
  <tag>HEAD</tag>
</scm>
```

Especificación de la versión del compilador *Java*

Se hace a través del plugin *maven-compiler-plugin*

```
<plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-compiler-plugin</artifactId>
   <version>2.0.2</version>
   <configuration>
        <source>1.8</source>
        <target>1.8</target>
   </configuration>
</plugin>
```

Propiedades del proyecto

Aquí se definen propiedades que pueden ser utilizadas en otros *pom.xml* de forma que se puede centralizar en un único sitio la gestión de las mismas. En este caso, se han centralizado las versiones de los componentes más importantes del proyecto.

```
<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <cassandra-driver.version>3.0.1</cassandra-driver.version>
  <liwenx.version>0.0.25</liwenx.version>
  <twitter4j.version>4.0.4</twitter4j.version>
  <spring-batch.version>3.0.7.RELEASE</spring-batch.version>
  <spring-security.version>4.1.0.RELEASE</spring-security.version>
  <spring.version>4.3.0.RELEASE</spring.version>
  <httpclient.version>4.3.2</httpclient.version>
  <spark.version>1.4.1</spark.version>
  <dse.version>4.8.6</dse.version>
</properties>
```

3.1 pfc-mon

Como se ha comentado anteriormente este módulo se encarga de monitorizar distintos servicios web y publicar tuits informando del estado de los mismos en sus correspondientes cuentas de *Twitter*. En términos de requisitos funcionales es el encargado de cumplir los números 1, 2 y 3.

Hace uso de *Spring Batch*, de *Apache HTTP Client* y de *Twitter4j* para conectarse a la API de *Twitter*, como se puede observar en las dependencias declaradas en su fichero *pom.xml*. Obsérvese también cómo las versiones hacen referencia a las propiedades declaradas en el proyecto padre mediante *${nombre-propiedad}*.

```
<dependencies>
      <dependency>
            <groupId>org.springframework.batch</groupId>
            <artifactId>spring-batch-core</artifactId>
            <version>${spring-batch.version}</version>
      </dependency>
      <dependency>
            <groupId>org.apache.httpcomponents</groupId>
            <artifactId>httpclient</artifactId>
            <version>${httpclient.version}</version>
      </dependency>
      <dependency>
            <groupId>org.twitter4j</groupId>
            <artifactId>twitter4j-core</artifactId>
            <version>${twitter4j.version}</version>
      </dependency>
</dependencies>
```

Este módulo dará como resultado dos ficheros. Por un lado un fichero *.jar (Java ARchive)* con todo el código compilado sin incluir sus dependencias, y por otro lado un fichero *.rar (Resource Adapter Archive)*. Éste último empaqueta en formato comprimido *ZIP* tanto el *.jar* resultante del módulo como todas sus dependencias.

El fichero *.rar* incluye todo lo necesario para desplegar el módulo en cualquier máquina y ejecutarlo mediante el comando *java*, como veremos más adelante. La generación de este fichero la realiza el plugin *maven-rar-plugin*, incluido en el *pom.xml*.

```
<plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-rar-plugin</artifactId>
   <version>2.2</version>
   <executions>
        <execution>
           <id>build-rar</id>
           <goals>
                <goal>rar</goal>
           </goals>
           <configuration>
                <rarSourceDirectory>src/main/script/</rarSourceDirectory>
           </configuration>
        </execution>
   </executions>
</plugin>
```

La estructura de directorios del módulo es la siguiente:

- *src*: contiene todo el código fuente del módulo: ficheros *.xml*, *.java*, *.properties*
- *src/main/java*: contiene todos los ficheros fuentes escritos en *Java*
- *src/main/resources*: contiene otros ficheros fuentes que no son *Java*, y por tanto no serán compilados. Formarán parte del resultado final del módulo. Típicamente son ficheros *.xml* y ficheros *.properties*
- *src/main/script*: se trata de un directorio añadido cuyo contenido se incluirá en fichero resultante final del módulo. En este caso contiene un script *Bash* para facilitar la ejecución.
- *src/main/config*: mediante perfiles *Maven* es posible generar distintas configuraciones de artefactos, por ejemplo, una para las versiones que se despliegan en entornos de desarrollo y otra para las versiones de entornos de producción. Aquí, en un subdirectorio por perfil, se incluyen los ficheros de configuración que dependan del entorno final donde se despliegan. Concretamente en nuestro caso son dos ficheros, *log4j.properties*, para *logging*; y *pfc-mon.properties*, que contiene las claves para el acceso a la API de *Twitter*. En principio se podría querer un nivel de *logging* distinto para producción y desarrollo y unas claves *API* distintas.
- *target*: contiene los ficheros resultantes de compilar el código fuente y construir el proyecto. Aquí se generarán entre otros los ficheros *.jar* y *.rar* anteriormente comentados.

3.1.1 Spring Batch

Spring Batch es un *framework* para facilitar el procesamiento tradicional por lotes. Proporciona un marco de trabajo base ofreciendo funcionalidades como gestión de trabajos (*job)*, tareas o pasos *(step)*, gestión de transacciones e implementaciones de escenarios de procesamientos de datos comunes como ficheros de texto plano o ficheros *XML*.

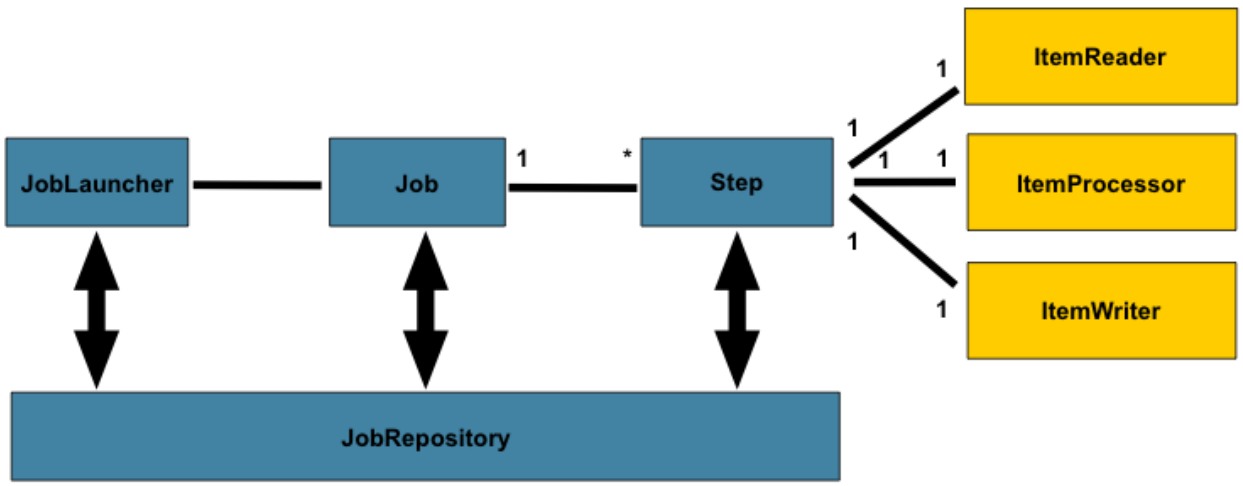

Figura 3-1. Diagrama de funcionamiento de Spring Batch

El esquema de funcionamiento es el mostrado en la Figura 3-1. Una trabajo de procesamiento se denomina *job* y está compuesto por uno o más pasos *(step)*. Cada *step* se divide en tres fases, ejecutadas por cada uno de los tres elementos que lo componen: *ItemReader*, encargado de leer los datos a procesar; *ItemProcessor*, encargado de transformarlos; e *ItemWriter*, encargado de escribir los datos procesados en algún sistema de almacenamiento.

Cuando se lanza una ejecución de un *job* por un *JobLauncher*, es el *JobRepository* el que se encarga de gestionar y controlar la ejecución de los trabajos y sus distintos pasos.

Hay que mencionar que existen otro tipo de *step* más simples, los *Tasklets*, para pasos que no siguen el patrón lectura-procesado-escritura de cada elemento de información.

Los elementos de color azul son proporcionados por el *framework* mientras que los elementos amarillos son los que hay que implementar y están más estrechamente relacionados con la tarea que se quiere realizar. La implementación de estos componentes es muy sencilla ya que basta con desarrollar clases que implementen cada una de las siguientes interfaces definidas.

ItemReader

```
public interface ItemReader<T> {

	T read() throws Exception, UnexpectedInputException, ParseException;
}
```

ItemProcessor

```
public interface ItemProcessor<I, O>{

	O process(I item) throws Exception;
}
```

ItemWriter

```
public interface ItemWriter<T> {

	void write(List<? extends T> items) throws Exception;
}
```

En este módulo vamos a utilizar *Spring Batch* de la siguiente forma: tendremos un único *job* con un único *step*. Éste no tendrá *ItemProcessor* sino únicamente *ItemReader* e *ItemWriter*. El primero, cada vez que el *framework* se lo pida, se encargará de tomar una muestra (petición HTTP) de uno de los servicios web a monitorizar y de devolver el resultado de la misma (fecha, estado HTTP, si se pudo conectar con el servicio o no y tiempo de proceso de la petición). El *framework* irá pasando las muestras al *ItemWriter* que se encargará de publicar un tuit por cada una de ellas.

3.1.2 Implementación del *ItemReader* y el *ItemWriter*

Antes de entrar en detalles del *ItemReader* y el *ItemWriter*, conviene mostrar detalles de implementación de las clases que van a representar los datos.

HttpSample

Contiene los datos resultantes de una petición HTTP a un servicio web. Almacena información del servicio web concreto mediante un objeto de tipo *HttpServiceConfig*.

```
public class HttpSample implements Serializable{
[…]
        private HttpServiceConfig service;
        private String url;
        private Calendar date;
        private boolean connected;
        // Http status
        private int status;
        // in ms
        private long processingTime;
[…]
```

HttpServiceConfig

Almacena los datos correspondientes a un servicio web y a la cuenta de *Twitter* donde se van a publicar sus tuits

```
public class HttpServiceConfig implements Serializable{
[…]
        private String url;
        private String twitterAccessToken;
        private String twitterTokenSecret;
[…]
```

HttpMonitorSampler

Es la clase que implementa a *ItemReader<HttpSample>*. Contiene una cola de servicios a procesar que son inyectados inicialmente una vez creado el objeto (*bean)* mediante el método *setServices()*. También contiene una segunda cola donde son insertados los servicios cuya primera muestra ha dado error. Podría ser un error puntual y se intenta una segunda vez antes de devolver una muestra con “servicio no disponible”.

Cada vez que el *framework* llama al método *read()* para obtener un elemento se llama al método *collectSample()*. Éste obtiene el siguiente servicio en la cola principal, y una vez se agote, si hay servicios a monitorizar en la cola de reintentos, se esperan dos minutos antes de iniciar las muestras por segunda vez.

Por último, en el método *readSample()* se puede observar cómo se hace uso de la librería *Apache HTTP Client* para hacer una petición HTTP mediante método *GET* [33].

```
public class HttpMonitorSampler implements ItemReader<HttpSample> {

	private static final String ACCESS_TOKEN_SEPARATOR = "##";

	private BlockingQueue<HttpServiceConfig> serviceQueue = new LinkedBlockingQueue<>();
	private BlockingQueue<HttpServiceConfig> retryQueue = new LinkedBlockingQueue<>();

	private boolean retryStageStarted = false;

	public HttpSample read() throws Exception, UnexpectedInputException, ParseException,
NonTransientResourceException {

		HttpSample sample = collectSample();

		return sample;
	}

	private HttpSample collectSample() throws InterruptedException{
		HttpServiceConfig service = serviceQueue.poll();

		HttpSample sample = null;
		if (service != null) {
			sample = readSample(service);
			if (!sample.isConnected()) {
				retryQueue.offer(service);
				sample = collectSample(); // Next sample
			}
		} else {
			if(!retryStageStarted){
				retryStageStarted = true;
				Thread.sleep(120000);
			}
			service = retryQueue.poll();
			if (service != null) {
				sample = readSample(service);
			}
		}

		return sample;
	}

	private HttpSample readSample(HttpServiceConfig service) {

		RequestConfig reqConfig =
RequestConfig.custom().setSocketTimeout(30000).setConnectTimeout(30000).build();
		HttpClient httpClient =
HttpClientBuilder.create().setDefaultRequestConfig(reqConfig).build();
		HttpGet method = new HttpGet(service.getUrl());
		HttpContext context = new BasicHttpContext();

		HttpSample sample = new HttpSample();
		sample.setService(service);
		sample.setDate(Calendar.getInstance());
		try {
			long tStart = System.currentTimeMillis();
			HttpResponse response = httpClient.execute(method, context);
			long tEnd = System.currentTimeMillis();
			sample.setConnected(true);
			sample.setProcessingTime(tEnd - tStart);
			sample.setStatus(response.getStatusLine().getStatusCode());

		} catch (IOException e) {
			// not connected
			sample.setConnected(false);
		}

		return sample;
	}
```

```
        public void setServices(Map<String, String> services) {
                for (Entry<String, String> e : services.entrySet()) {
                        HttpServiceConfig s = new HttpServiceConfig();
                        s.setUrl(e.getKey());
                        String accessToken = e.getValue();
                        String[] accessTokenParts = accessToken.split(ACCESS_TOKEN_SEPARATOR);
                        s.setTwitterAccessToken(accessTokenParts[0]);
                        s.setTwitterTokenSecret(accessTokenParts[1]);
                        serviceQueue.add(s);
                }
        }
}
```

HTTPMonitorTweeter

Es la clase que implementa a *ItemWriter<HTTPSample>*. El *framework* llamará al método *write()* pasándole las muestras obtenidas por *HTTPMonitorSampler*. Éste *bean* se encagará de publicar un tuit por cada muestra haciendo uso de *Twitter4j*.

Se puede observar cómo se hace uso de los *tokens* de acceso de los distintos servicios a través del objeto *HttpServiceConfig* de cada muestra. A su vez, esta clase contiene unas claves de acceso a la API de *Twitter*, mediante las cuales se conectará al mismo con su cuenta de aplicación. Para publicar los tuits en nombre de las cuentas de los distintos servicios (cuentas de usuario de *Twitter*), utilizarán los *tokens*, que representan la autorización de dichas cuentas de usuarios a nuestra aplicación para publicar tuits.

```
public class HttpMonitorTweeter implements ItemWriter<HttpSample> {


        private static final String serviceConnectedTweetFormat = "At this moment our service
is {0}, returning HTTP code {1} in {2}ms. [{3,time}]";
        private static final String serviceNotConnectedTweetFormat = "At this moment our
service is unavailable. Unable to connect. [{0,time}]";

        private final Log log = LogFactory.getLog(getClass());

        private String twitterKey;
        private String twitterSecret;

        private Twitter getTwitterService(String accessToken, String tokenSecret) {
                AccessToken at = new AccessToken(accessToken, tokenSecret);
                return buildTwitterFactory().getInstance(at);
        }

        private TwitterFactory buildTwitterFactory() {
                ConfigurationBuilder cb = new ConfigurationBuilder();
                cb.setOAuthConsumerKey(twitterKey);
                cb.setOAuthConsumerSecret(twitterSecret);
                cb.setDebugEnabled(true);
                return new TwitterFactory(cb.build());
        }
```

```
	@Override
	public void write(List<? extends HttpSample> samples) throws Exception {
		for (HttpSample s : samples) {
			try {
				HttpServiceConfig c = s.getService();
				Twitter twitter = getTwitterService(c.getTwitterAccessToken(),
c.getTwitterTokenSecret());
				String status;
				if (s.isConnected()) {
					int statusCode = s.getStatus();
					String availabilityText;
					if (statusCode < 400) {
						availabilityText = "available";
					} else {
						availabilityText = "unavailable";
					}
					status =
MessageFormat.format(serviceConnectedTweetFormat, availabilityText, statusCode,
s.getProcessingTime(), s.getDate().getTime());
				} else {
					status =
MessageFormat.format(serviceNotConnectedTweetFormat, s.getDate().getTime());
				}

				twitter.updateStatus(status);
			} catch (Exception e) {
				log.error(e.getMessage(), e);
			}
		}
	}

	/**
	 * @param twitterKey the twitterKey to set
	 */
	public void setTwitterKey(String twitterKey) {
		this.twitterKey = twitterKey;
	}

	/**
	 * @param twitterSecret the twitterSecret to set
	 */
	public void setTwitterSecret(String twitterSecret) {
		this.twitterSecret = twitterSecret;
	}
}
```

3.1.3 Configuración del *job*

La configuración de *Spring Batch* se hace a través de ficheros *XML*. El fichero principal del módulo y al cuál haremos referencia en cada ejecución del mismo es *monitorJob.xml* situado en *src/main/resources*.

Aunque en este fichero se utilicen funcionalidades de *Spring*, que veremos en más profundidad en el apartado 3.2.1.Spring, lo único que tenemos que saber de momento es que las etiquetas *<bean/>* permiten instanciar un objeto de la clase especificada y con las propiedades especificadas en dicha etiqueta.

Se puede observar cómo se instancia un objeto de tipo *HTTPMonitorSampler*, con los servicos a monitorizar, donde las claves del mapa representan las urls y el valor el *token* de acceso a sus respectivas cuentas de *Twitter*.

Por otro lado, *HTTPMonitorTweeter* es instanciado proporcionándole las claves de acceso de la aplicación a la API de *Twitter*. Estas claves no son proporcionadas directamente sino que, mediante el *bean PropertyPlaceholderConfigurer*, configurado en primer lugar, se leen del fichero *pfc-mon.properties*.

Es interesante observar cómo se configuran el gestor de transacciones, el *jobRepository* y el *jobLauncher*, mediante implementaciones ya proporcionadas por *Spring Batch*.

Por último, nuestro *job* es definido con un único *step, step1*, donde se hacen referencia a nuestro *ItemReader* y nuestro *ItemWriter*. La propiedad *commit-interval* se pone a 1 para que cada muestra obtenida se pase directamente al *ItemWriter*, ya que nos interesa que los tuits sean publicados lo más inmediatamente posible a la obtención de la muestra.

```
<beans:beans xmlns="http://www.springframework.org/schema/batch"
     xmlns:beans="http://www.springframework.org/schema/beans"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans.xsd
           http://www.springframework.org/schema/batch
           http://www.springframework.org/schema/batch/spring-batch-2.2.xsd">


   <beans:bean id="liwenx-properties"
       class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
       <beans:property name="ignoreUnresolvablePlaceholders" value="true" />
       <beans:property name="locations">
          <beans:list>
              <beans:value>pfc-mon.properties</beans:value>
          </beans:list>
          </beans:property>
       </beans:bean>


   <beans:bean name="monitorSampler" class="me.alexguerra.pfc.mon.HttpMonitorSampler">
       <beans:property name="services">
          <beans:map>
              <beans:entry key="http://www.ten-golf.com/es/" value="760830411421474816-
Xc1vlpBpxEjbE1cflXBMuDoKqSRmMCc##hEl1AyfhnUeKQPrhJvvBtljS2HqguUSoY66xOmgWTLi8V"/>
              <beans:entry key="http://www.etsi.us.es" value="761123576644644864-
EQhcLYT3YBvz5Jnu22RGV1Svc2G2HnK##uTbpfg930pVile5cTBlMO5BwkjLkPXYP9Ght71kgiD6Cd"/>
              <beans:entry key="https://api.spotify.com/v1/artists/51Blml2LZPmy7TTiAg47vQ"
value="761129981556600833-
t34y9rj4Lp9c2scgkLnV0FKPvohHseh##V23LYJqAZr5XdeLt1RxULcIGiYy8vO22a1npQEsVQLivP"/>
              <beans:entry key="http://www.ieee.org/index.html" value="761131721534218240-
XdBEMzYjrOpwpo8YSTkMZ4Ow2ZTuxaC##8xSi8ifAlwqLSSmIhDjW70J3TF9pAoGAa3E8nOIamHEJH"/>
              <beans:entry key="http://www.itu.int/es/Pages/default.aspx"
value="761137732039180288-
P3YAvR4wcxD15duXy1n7HVzCo6S0p2e##FtEzw8K2VImLKh6vaid4U5q0yA4trYe33phH6swgJaV16"/>
              <beans:entry key="https://www.w3.org/" value="761137071587221504-
4AUvIK44hnLMoz1AqqxPlj52Jl5Dmca##NEybnr2eEEzMlI2JLLvz5C2UL77EWpwkFRvIE9htFW3Wq"/>
```

```
                <beans:entry key="https://nabumbu.com/es" value="76113886022719488l-
CMXuwFDYuDjjVmGbGuHsU9I4jHIYFpQ##MTYCeffe4lVVVuMI7isfmaTYakjy4N1eHL2tHmAi1thC7"/>
            </beans:map>
        </beans:property>
    </beans:bean>

    <beans:bean name="monitorTweeter" class="me.alexguerra.pfc.mon.HttpMonitorTweeter">
        <beans:property name="twitterKey" value="${twitter.consumerKey}"/>
        <beans:property name="twitterSecret" value="${twitter.consumerSecret}"/>
    </beans:bean>

    <!-- Spring batch -->
    <beans:bean name="transactionManager"
        class="org.springframework.batch.support.transaction.ResourcelessTransactionManager"
/>

    <beans:bean id="jobRepository"
       class="org.springframework.batch.core.repository.support.MapJobRepositoryFactoryBean">
        <beans:property name="transactionManager" ref="transactionManager" />
    </beans:bean>

    <beans:bean id="jobLauncher"
        class="org.springframework.batch.core.launch.support.SimpleJobLauncher">
        <beans:property name="jobRepository" ref="jobRepository" />
    </beans:bean>

    <job id="monitorJob">
        <step id="step1">
            <tasklet>
                <chunk reader="monitorSampler" writer="monitorTweeter"
                       commit-interval="1"/>
            </tasklet>
        </step>
    </job>

</beans:beans>
```

3.1.4 Perfiles, compilación y construcción del módulo con *Maven*

Una de las características interesantes de *Maven* es que permite configurar distintos perfiles de construcción, lo que permite generar versiones con distinta configuración para diferentes entornos.

Los perfiles se definen en el fichero *pom.xml* y permiten configurar ciertos valores para determinadas propiedades que luego pueden ser usadas a conveniencia. En este caso, lo que varía de un perfil a otro es la propiedad *profile.config.dir*. Se puede observar cómo hay dos perfiles: *local* y *production*, y establecen el valor de la propiedad en *src/main/config/local* y *src/main/config/production* respectivamente. El perfil por defecto es *local*.

```
<profiles>
   <profile>
       <id>local</id>
       <activation>
          <activeByDefault>true</activeByDefault>
       </activation>
       <properties>
          <profile.config.dir>src/main/config/local/</profile.config.dir>
       </properties>
   </profile>
   <profile>
       <id>production</id>
       <activation>
          <activeByDefault>false</activeByDefault>
       </activation>
       <properties>
          <profile.config.dir>src/main/config/production/</profile.config.dir>
       </properties>
   </profile>
</profiles>
```

En la configuración de los recursos *no-Java* dentro del *POM*, se establece que los ficheros *log4j.properties y pfc-mon.properties* se tomen del directorio configurado en dicha propiedad. De esta forma, según el perfil con el que se construya el proyecto tomará unos ficheros u otros.

```
<resources>
   <resource>
       <directory>src/main/resources</directory>
       <filtering>false</filtering>
   </resource>
   <resource>
       <directory>${profile.config.dir}</directory>
       <includes>
          <include>pfc-mon.properties</include>
          <include>log4j.properties</include>
       </includes>
       <filtering>false</filtering>
   </resource>
</resources>
```

A la hora de construir el proyecto con *Maven* hay que tener en cuenta los distintos *build lifecycles* o ciclos de vida de construcción. Existen tres ciclos de vida principales: ***default***, que se encarga de la construcción y despliegue del proyecto en repositorios *Maven*; ***clean***, encargado de limpiar el proyecto; y ***site*** que se encarga de generar el sitio web de documentación del proyecto.

Cada *lifecycle* está dividido en fases, y éstas a su vez en *plugin goals* u objetivos de plugins. Por ejemplo, el *default lifecycle* contiene las siguientes fases entre otras [34]:

- *validate*: donde se valida que el proyecto es correcto y toda la información necesaria está disponible.

- *compile*: se compila todo el código fuente del proyecto.
- *package*: se empaqueta el código compilado en su correspondiente formato: jar, war, rar, ear, ...
- *install*: se instala el paquete en el repositorio *Maven* local, permitiendo su uso en otros proyectos locales.
- *deploy*: se instala el paquete en el repositorio *Maven* remoto para compartirlo con otros desarrolladores y proyectos.

Para compilar y construir el proyecto, basta con invocar el comando *mvn* seguido de las fases que queremos ejecutar. Todas las fases previas de las que dependen las fases especificadas serán ejecutadas. Por ejemplo, especificando la fase *install* también se ejecutarían *validate, compile* y *package*.

Las fases que nos interesan ejecutar son *clean* e *install*.

```
bash$ mvn clean install
```

Si no se especifica ningún perfil, el perfil por defecto será utilizado. Para especificar otro perfil basta indicarlo con el parámetro P.

```
bash$ mvn –Pproduction clean install
```

3.1.5 Ejecución

Un proyecto *Spring Batch* puede ejecutarse de diversas maneras. La que nos interesa en este caso es la ejecución por línea de comandos. Para ello basta ejecutar la clase *org.springframework.batch.core.launch.support.CommandLineJobRunner* con el comando *java* pasándole como argumentos el fichero *XML* con la configuración y el nombre del *job* a ejecutar. Al comando *java* hay que especificarle en la opción *classpath* todos los ficheros *.jar* que componen el sistema.

Tal y como se mencionó anteriormente, se ha incluido en el *.rar* resultante un script *Bash*, *run.sh*, que facilita la ejecución del módulo. Este script obtiene un listado de los ficheros *.jar* del directorio de ejecución y genera el comando *java* al completo. Se ha diseñado para recibir los mismos argumentos por la línea de comandos que *CommandLineJobRunner*. Por último, también implementa un sistema de ficheros de bloqueo que impide que se ejecuten por error dos ejecuciones del mismo *job* simultáneamente.

La ejecución del módulo *Monitor* a través del script se realizaría de la siguiente forma:

```
bash$ ./run.sh monitorJob.xml monitorJob
```

run.sh

```
#/bin/bash
##
## Script para ejecutar el monitor de servicios
## Autor: Alejandro Guerra Cabrera <alex.guerra.cabrera@gmail.com>
##

# Cambiar por la ruta donde estén los .jar. No acaba en /
# BASE_PATH=`pwd`
BASE_PATH=`pwd`

# Cambiar por la ruta hasta el comando java que se desea usar
#JAVA_BIN=`which java`
JAVA_BIN=`which java`
JAVA_OPTS="-server -Xmx128m"

export LANG="en_GB.UTF-8";
LOCK_FILE_PATH="/tmp/"
LOG_FILE="/tmp/pfc-mon.log"
# Tiempo máximo en segundos de ejecución del proceso.
# Si pasado este tiempo se lanza una ejecución, se aborta el proceso anterior y se lanza
la nueva ejecución.
PROCESS_TIMEOUT=$((12*60*60));
if [ $# -ge 2 ]
then
        PROCESS_NAME=$2'_'$$;
        echo "[`date +"%d/%m/%Y %H:%M:%S"`] Starting job "$PROCESS_NAME >> $LOG_FILE

        LOCK_FILE_NAME='importerJob_'$1'.lock';
        MUST_START_PROCESS=1;
        if [ -a ${LOCK_FILE_PATH}${LOCK_FILE_NAME} ]
        then
                now=`date +%s`;
                process_start=`stat -c %Y ${LOCK_FILE_PATH}${LOCK_FILE_NAME}`;
                process_spent=$(($now-$process_start));
                if [ $process_spent -lt $PROCESS_TIMEOUT ]
                then
                        echo "A instance is already running. Lock file exists in
${LOCK_FILE_PATH}${LOCK_FILE_NAME}.";
                        echo "[`date +"%d/%m/%Y %H:%M:%S"`] A instance of job $2 is
already running. Lock file exists in ${LOCK_FILE_PATH}${LOCK_FILE_NAME}" >> $LOG_FILE;
                        MUST_START_PROCESS=0;
                else
                        old_process_pid=`cat ${LOCK_FILE_PATH}${LOCK_FILE_NAME}`;
                        echo "A instance is already running, but it takes
${process_spent} s. Process timeout is setted up to ${PROCESS_TIMEOUT} s. Process will be
aborted.";
                        echo "[`date +"%d/%m/%Y %H:%M:%S"`] A instance of job $2 is
already running, but it takes ${process_spent} s. Process timeout is setted up to
${PROCESS_TIMEOUT} s. Process will be aborted." >> $LOG_FILE;
                        kill -9 ${old_process_pid};
                        sleep 2;
                       rm -f ${LOCK_FILE_PATH}${LOCK_FILE_NAME}
                fi
        fi
        if [ $MUST_START_PROCESS = 1 ]
        then
                touch ${LOCK_FILE_PATH}${LOCK_FILE_NAME}
                list=`ls ${BASE_PATH}/*.jar`
                command=${JAVA_BIN}' '${JAVA_OPTS}' -classpath '
               class='org.springframework.batch.core.launch.support.CommandLineJobRunner'
                first='0'
                for i in $list;
                do
```

```
                        if [ $first == '0' ];
                          then
                                  first='1';
                          else
                                  command=$command':';

                        fi
                        command=$command$i;
                done;
                command=$command' '$class' '$1' '$2;
                #echo $command;

                echo "[`date +"%d/%m/%Y %H:%M:%S"`] Initializing main process
"$PROCESS_NAME >> $LOG_FILE
                t1=$(date +%s%N | cut -c -13)
                ${command} 2>&1 &
                pid=$!;
                echo ${pid} > ${LOCK_FILE_PATH}${LOCK_FILE_NAME};
                wait ${pid};
                error_code=$?
                t2=$(date +%s%N | cut -c -13)
                if [ ${error_code} -eq  0 ]
                then
                        echo "[`date +"%d/%m/%Y %H:%M:%S"`] Job "$PROCESS_NAME" was
completed successfully. It took "$(($t2 - $t1))" ms." >> $LOG_FILE
                else
                    echo "[`date +"%d/%m/%Y %H:%M:%S"`] An error occurs in job
"$PROCESS_NAME" execution. Job took "$(($t2 - $t1))" ms." >> $LOG_FILE
                fi
                rm -f ${LOCK_FILE_PATH}${LOCK_FILE_NAME}
        fi
        echo "[`date +"%d/%m/%Y %H:%M:%S"`] Closing job "$PROCESS_NAME >> $LOG_FILE
else
        echo 'Usage: run.sh jobPath jobName [jobParameters]'
        echo 'Example: run.sh monitorJob.xml monitorJob'
fi
```

3.2 pfc-web

Este módulo implementa la aplicación web a través de la cual los usuarios pueden interactuar añadiendo cuentas de *Twitter* para monitorizar y visualizando los resultados obtenidos tras el análisis del módulo *pfc-analysis*. Además, como ya se ha mencionado anteriormente, implementa el *Recopilador de Tweets*. En términos de requisitos funcionales es el encargado de cumplir los requisitos del 4 al 9, con la salvedad del número 8, que únicamente es encargado de mostrar los datos, no de analizarlos.

Dado que se trata de una aplicación web *Java* este módulo se ha configurado en el *POM* para que genere un fichero *.war (Web ARchive)*, que es como se empaquetan las aplicaciones web en *Java*.

```
    <parent>
        <groupId>me.alexguerra</groupId>
        <artifactId>pfc</artifactId>
        <version>0.0.1-SNAPSHOT</version>
    </parent>
    <artifactId>pfc-web</artifactId>
    <packaging>war</packaging>
```

La estructura de directorios del módulo es la siguiente:

- *src*: contiene todo el código fuente del módulo: ficheros *.xml*, *.java*, *.properties*
- *src/main/java*: contiene todos los ficheros fuentes escritos en *Java*
- *src/main/resources*: contiene otros ficheros fuentes que no son *Java*, y por tanto no serán compilados. Formarán parte del resultado final del módulo. Típicamente son ficheros *.xml* y ficheros *.properties*
- *src/main/config*: al igual que en el módulo anterior se usa para incluir un directorio por perfil *Maven* con ficheros que contienen configuración que puede variar en distintos entornos de despligue.
- *src/main/sql: contiene ficheros que definen el modelo de datos.*
- *src/main/webapp:* este directorio se traducirá al construir el proyecto en el directorio raíz de la aplicación web. De aquí colgarán distintas carpetas con ficheros de hojas de estilo *CSS,* imágenes, javascript y por supuesto el directorio *WEB-INF*, que contiene los recursos de la aplicación que no son de dominio público.
- *target:* en este directorio se almacenan los ficheros generados al construir la aplicación. El más relevante en este caso es el fichero *.war* que podrá ser desplegado en el contenedor de aplicaciones *Tomcat*.

3.2.1 Spring

Spring es un *framework* modular que permite construir aplicaciones *Java* ofreciendo soluciones a los problemas más comunes a los que se enfrentan los desarrolladores. Para ello dispone de multitud de módulos, que se pueden incluir de forma individual en la aplicación. La Figura 3-2 muestra los módulos más importantes y cómo se agrupan dentro del *framework*.

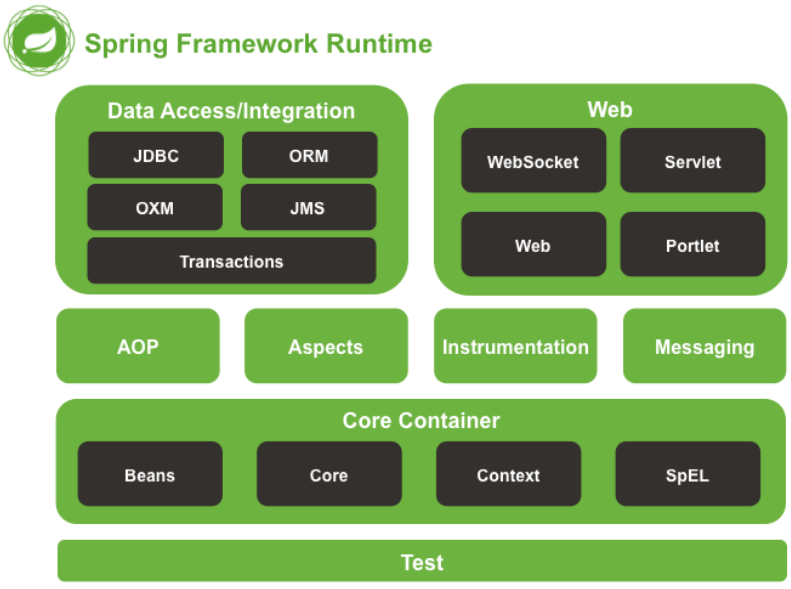

Figura 3-2. Diagrama de módulos de *Spring*

La base de este *framework* es la *Inversión del Control*, o *IoC* por su nombre en inglés, también conocida como inyección de dependencias o *Dependency Injection*. En sentido amplio se puede decir que una aplicación *Java* consiste en un conjunto de objetos, o instancias de clases, que colaboran entre sí ofreciendo una serie de funcionalidades. Estos objetos tienen cierta dependencia unos de otros.

Spring, a través de su módulo *Spring Beans,* permite definir estos objetos y sus dependencias de forma sencilla a través de *XML* y anotaciones *Java.* Si observamos el siguiente *XML* extraído del proyecto, se puede notar cómo las etiquetas *<bean/>* definen la instancia de una clase especificada con el atributo *class*, es decir, un objeto o b*ean*. Éste es configurado posteriormente asignando valores a sus propiedades a través de los métodos *setter*.

Por ejemplo, podemos deducir que el *bean messageSource* tiene los métodos *setBasenames(), setCacheSeconds()* y *setDefaultEncoding()*. El primero recibe un conjunto de cadenas de texto o *Set<String>*, el segundo un número entero y el tercero una cadena de texto o *String*. El *bean systemMessageSource* tiene una dependencia con el anterior, que se inyecta a través del método *setMessageSource()*. En este caso, en lugar de especificar el atributo *value* en la etiqueta *property*, se usa el atributo *ref*.

```
<bean id="messageSource"
   class="org.springframework.context.support.ReloadableResourceBundleMessageSource">
        <property name="basenames">
           <set>
                <value>classpath:/net/sf/liwenx/message</value>
                <value>classpath:/me/alexguerra/pfc/message</value>
           </set>
        </property>
        <property name="cacheSeconds" value="-1" />
        <property name="defaultEncoding" value="UTF-8"/>
</bean>

<bean id="systemMessageSource" class="net.sf.liwenx.util.DefaultLocaleMessageSource">
   <property name="messageSource" ref="messageSource" />
</bean>
```

Otra forma de especificar la dependencia entre *beans* es mediante la anotación *@Autowired* colocada en los métodos *setters*. El siguiente ejemplo muestra un *setter* de un *bean* que depende del *bean AuthenticationManager*.

```
@Autowired
public void setAuthManager(AuthenticationManager authManager) {
       this.authManager = authManager;
}
```

Otro de los módulos importantes a la hora de construir una aplicación web es *Spring WebMVC*, que proporciona la base para construir *servlets* que sigan el patrón *modelo-vista-controlador*. Si inspeccionamos el fichero *web.xml* [35], que describe entre otras cosas cómo las URLs son mapeadas a los distintos *servlets,* podemos observar lo siguiente.

```
<context-param>
          <param-name>contextConfigLocation</param-name>
          <param-value>/WEB-INF/liwenx-datasources.xml, /WEB-INF/liwenx-services.xml,
/WEB-INF/liwenx-security.xml</param-value>
</context-param>

<listener>
   <listener-class>
        org.springframework.security.web.session.HttpSessionEventPublisher
   </listener-class>
</listener>

<listener>
   <listener-class>
        org.springframework.web.context.ContextLoaderListener
   </listener-class>
</listener>

<servlet>
   <servlet-name>liwenx</servlet-name>
   <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
   <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
   <servlet-name>liwenx</servlet-name>
   <url-pattern>/liwenx/*</url-pattern>
</servlet-mapping>
```

Por un lado se definen dos *listeners* de eventos: *ContextLoaderListener*, que carga los ficheros especificados en el parámetro de contexto *contextConfigLocation*; y por otro lado *HttpSessionEventPublisher*, encargado de mantener la sesión del usuario.

Por otro lado se define un *servlet* ya proporcionado por *Spring* a través de la clase *org.springframework.web.servlet.DispatcherServlet*. Este *servlet* procesará todas las peticiones que empiezen por la URL *nombre_app/liwenx/* tal y como indica la etiqueta *servlet-mapping*. Además, puesto que se llama *liwenx* buscará un fichero llamado *liwenx-servlet.xml* y cargará todos los *beans* y configuración especificados en el mismo.

Si echamos un vistazo a dicho fichero observamos que se define un *bean* llamado *urlMapper* cuya labor es mapear las URLs a los distintos controladores definidos. En este caso todas las peticiones van al mismo, que está implementado por *Liwenx*.

```
<bean name="liwenx-controller" class="net.sf.liwenx.LiwenxController">
   <property name="liwenx" ref="liwenx" />
</bean>

<bean id="urlMapper"
        class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
   <property name="mappings">
        <map>
           <entry key="*" value-ref="liwenx-controller" />
        </map>
   </property>
</bean>
```

Antes de pasar a detallar *Liwenx* es interesante notar que ya tenemos un *servlet* que es capaz de mapear URLs a controladores, control de sesión de usuario y aún no hemos escrito una línea de código *Java*. Todas estas funcionalidades, comunes en el desarrollo de aplicaciones web, son ofrecidas por *Spring WebMVC*.

3.2.2 Liwenx

Esta librería la implementé en 2010 con el objetivo de ser un motor de portales web ligero que permitiera a los desarrolladores generar sitios web de manera sencilla, preocupándose únicamente de implementar la lógica relativa a los problemas que pretende resolver y escondiendo los detalles de implementación comunes a todas las aplicaciones web. Las páginas son generadas separando los datos de su presentación. Los primeros son generados en formato *XML*, que luego es transformado mediante un motor de plantillas, en este caso *Freemarker,* para ser presentados al usuario en el formato adecuado, *HTML/CSS/javascript, XML, ...*

Utiliza *Spring MVC* y, por tanto, su configuración parte del fichero *liwenx-servlet.xml*, como hemos visto en el apartado anterior.

```
<bean id="router" class="me.alexguerra.pfc.liwenx.router.PFCRouter">
   <property name="messageSource" ref="systemMessageSource" />
   <property name="defaultPage" value="home" />
   <property name="debug" value="${debug}" />
   <property name="defaultLocale" value="es" />
</bean>

<bean id="configLoader" class="net.sf.liwenx.config.ConfigLoader">
   <property name="configResources" value="WEB-INF/config/page-template-config.xml" />
   <property name="messageSource" ref="systemMessageSource" />
</bean>

<bean id="freemarkerView" class="net.sf.liwenx.impl.FreemarkerView">
   <property name="systemMessageSource" ref="systemMessageSource" />
   <property name="encoding" value="UTF-8" />
   <property name="templatesDir" value="WEB-INF/freemarker/web" />
   <property name="mainTemplate" value="main.ftl" />
   <property name="globals" ref="freemarkerGlobals" />
   <property name="userMessageSource" ref="translationsMessageSource" />
   <property name="debug" value="${debug}" />
</bean>

<bean id="liwenx" class="net.sf.liwenx.impl.LiwenxImpl">
   <property name="configLoader" ref="configLoader" />
   <property name="encoding" value="UTF-8" />
   <property name="messageSource" ref="systemMessageSource" />
   <property name="router" ref="router" />
   <property name="view" ref="freemarkerView" />
</bean>

<bean name="liwenx-controller" class="net.sf.liwenx.LiwenxController">
   <property name="liwenx" ref="liwenx" />
</bean>
```

El *bean liwenx-controller* procesa todas las peticiones que le pasa el *servlet* y se las pasas al *bean liwenx*, implementado por la clase *LiwenxImpl*. Éste depende del *configLoader*, que cargará

la configuración de las distintas páginas de la aplicación del fichero *page-template-config.xml* situado en *WEB-INF/config*.

Otra de las dependencias es *freemarkerView*, *bean* que transformará las páginas en formato *XML* a *HTML* con las plantillas definidas en el directorio *WEB-INF/freemarker/web*. Éste último también toma como dependencia *freemarkerGlobals*, un mapa con valores dependientes del entorno de despliegue, como las URLs base de la aplicación web. Es por ello por lo que vienen del fichero de configuración *liwenx.properties*, situado en *src/main/config* y que podría ser distinto para cada perfil *Maven*.

```
<bean id="liwenx-properties"
   class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
   <property name="ignoreUnresolvablePlaceholders" value="true" />
   <property name="locations">
        <list>
           <value>/WEB-INF/liwenx.properties</value>
        </list>
   </property>
</bean>

<util:map id="freemarkerGlobals" key-type="java.lang.String"
   value-type="java.lang.Object">
   <entry key="base_resources" value="${bases.resources}" />
   <entry key="base_pages" value="${bases.pages}" />
   <entry key="base_server" value="${bases.server}" />
   <entry key="base_version" value="${bases.version}" />
   <entry key="debug" value="${debug}"/>
</util:map>
```

Otra de las dependencias de *freemarkerView* es *translationsMessageSource*, un *bean* que permite a las plantillas *Freemarker* obtener las traducciones correspondientes a cada idioma. Estos ficheros se sitúan en el directorio *WEB-INF/translations* y en su nombre indican el idioma de los textos que contienen añadiendo como sufijo el código del idioma en formato *BCP 47* definido por *IEEE* [36]. Por ejemplo, *translations_es.properties*, *translations_en*.properties, *translations_fr.properties*. De esta forma, para traducir la vista a un idioma basta con poco más que añadir un nuevo fichero con las traducciones correspondientes a este directorio.

```
<bean id="translationsMessageSource"
   class="org.springframework.context.support.ReloadableResourceBundleMessageSource">
   <property name="basenames">
        <set>
           <value>/WEB-INF/translations/translations</value>
        </set>
   </property>
   <property name="defaultEncoding" value="UTF-8" />
   <property name="cacheSeconds" value="0" />
</bean>
```

Siguiendo con el *bean Liwenx*, otra de sus dependencias es el *bean router*, implementado en este proyecto por la clase *PFCRouter*. Éste es el encargado de, dada una petición, calcular la

página que se debe ejecutar, el idioma y el tipo de vista entre otras cosas. Esto lo devuelve mediante un objeto de tipo *Locator*.

```
public class PFCRouter implements Router{

   public static final String PATH_LIST = "path-list";
   public static final String LOCAL_JSON = "local-requested-json";
   public static final String SESSION_REAL_IP = "session-real-ip";
   public static final String SESSION_REAL_IP_NULL = "_NULL_";

   private final Log log = LogFactory.getLog(getClass());
   private UnifiedLocaleMessageSource messageSource;
   private String defaultPage = "index";
   private String defaultLocale;
   private boolean debug;

   @Override
   public Locator route(LiwenxRequest request) {
        String path = request.getPathInfo();
        if(log.isTraceEnabled()){
           log.trace(messageSource.getMessage("router.path", path));
        }
        String page;
        if(path!=null && !path.equals("/")){
           String[] pathArr=path.substring(1).split("/");
           page = pathArr[0];

           if (pathArr.length > 1) {
                List<String> pathList = new ArrayList<String>();
                for (int i = 1; i < pathArr.length; i++) {
                   pathList.add(pathArr[i]);
                }
                request.setLocalValue(PATH_LIST, pathList);
           }
        }else{
           page=defaultPage;
        }

        request.setLocalValue(LOCAL_JSON, page.endsWith(".json"));

        Device dev;
        if(debug){
           String d = request.getRequestValue("d");
           if(d!=null && d.trim().equals("xml")){
                dev = Device.XML;
           }else{
                dev = Device.HTML;
           }
        }else{
           dev = Device.HTML;
        }

        // Locale
        Locale l = LocaleUtils.toLocale(defaultLocale);

        discoverIp(request);

        return new Locator(page, dev, l);
   }

[…]
```

En esta implementación es muy básica ya que lo que hace es mapear directamente el siguiente elemento de URL tras el nombre del *servlet* a la página con el mismo nombre. Por ejemplo, la URL *http://server:8080/pfc/liwenx/sign-up* se corresponde con la página llamada *sign-up*.

Como ya hemos visto en la configuración de *Liwenx* las páginas se definen en el fichero *page-template-config.xml* situado en *WEB-INF/config*. El formato de este *XML* se muestra a continuación.

```
<?xml version="1.0" encoding="UTF-8"?>
<liwenx:config xmlns:liwenx="http://liwenx.sourceforge.net">
   <!-- Templates -->
   <liwenx:template name="generic" source="WEB-INF/config/templates/generic.xml">
   </liwenx:template>

   <liwenx:template name="inner-page" source="WEB-INF/config/templates/inner-page.xml">
       <liwenx:post-router name="innerPagePostRouter" />
   </liwenx:template>

   <!-- Pages -->
   <liwenx:page name="home" source="WEB-INF/config/pages/home.xml" template="generic"
private="false">
       <liwenx:post-router name="homePostRouter" />
   </liwenx:page>

   <liwenx:page name="sign-up" source="WEB-INF/config/pages/sign-up.xml" template="generic"
private="false">
       <liwenx:post-router name="signUpPostRouter" />
   </liwenx:page>

   <liwenx:page name="user-home" source="WEB-INF/config/pages/user-home.xml"
template="inner-page" private="true" >
   </liwenx:page>

   <liwenx:page name="service" source="WEB-INF/config/pages/service.xml" template="inner-
page" private="false" >
   </liwenx:page>

   <liwenx:page name="twitter-auth-callback" source="twitterAuthCallback" single-
component="true" private="false"/>

   <!-- Json requests -->
   <liwenx:page name="discover-twitter-auth-url" source="discoverTwitterAuthUrl" single-
component="true" private="false" />

   <liwenx:page name="is-logged-user.json" source="isLoggedUser" single-component="true"
private="false" />

   <liwenx:page name="sign-up.do" source="signUp" single-component="true" private="false" >
       <liwenx:post-router name="signUpPostRouter" />
   </liwenx:page>

   <liwenx:page name="is-email-registered.json" source="isEmailRegistered" single-
component="true" private="false"/>

   <liwenx:page name="save-account.do" source="saveAccount" single-component="true"
private="false"/>

   <liwenx:page name="read-tweets.json" source="readTweets" single-component="true"
private="false"/>

   <liwenx:page name="read-account.json" source="readAccount" single-component="true"
private="false"/>

</liwenx:config>
```

Se pueden definir páginas basadas en un fichero *XML* como *home* o directamente asociadas a un *bean*, como la página *sign-up.do*, que procesa el registro. Las páginas pueden ser públicas o privadas, como *user-home* a la que sólo podemos acceder a través de una redirección interna.

También existe la figura de los *post-routers*, que son *beans* que se ejecutan justo antes de ejecutar la página y que pueden decidir que se ejecute otra página en lugar de la propuesta por el *router*.

En este caso tenemos dos *post-routers*: *innerPagePostRouter*, que se ejecuta en todas las páginas donde se requiera un usuario identificado. De no ser así redirige a la página *home* en lugar de ejecutar la página solicitada; y *homePostRouter*, que se añade a la página *home* (home pública) y en caso de haber usuario identificado, redirige a la página *user-home* (home de usuario). De esta forma, bajo la misma URL se pueden ejecutar una u otra página dependiendo de si se requiere identificación.

Por último, comentar que las páginas *XML* pueden basarse en otra plantilla o *template XML*. Las plantillas son páginas *XML* que tienen puntos de extensión, definidos por la etiqueta *liwenx:extension-point*, donde se inserta el *XML* de las páginas. A modo de ejemplo a continuación se presenta la plantilla *inner-page*.

```
<?xml version="1.0" encoding="UTF-8"?>
<page xmlns:liwenx="http://liwenx.sourceforge.net">
   <head>
        <liwenx:component name="loggedUserInfo"/>
        <liwenx:component name="readAccounts"/>
   </head>
   <body>
        <liwenx:extension-point name="body"/>
   </body>
   <footer>
   </footer>

</page>
```

Tanto las páginas *XML* como las plantillas pueden incluir código *XML* dinámico llamando a los componentes de página mediante la etiqueta *liwenx:component*. Estos *beans* están definidos en el fichero *liwenx-component.xml*, situado en la carpeta *WEB-INF*. La página *service*, por ejemplo, que muestra los datos de un servicio monitorizado por un usuario, llama a los *beans readAccount* y *readMonitorizedAccountStats* para leer los detalles de la cuenta de servicio monitorizada y las estadísticas del mismo respectivamente.

```
<?xml version="1.0" encoding="UTF-8"?>
<page xmlns:liwenx="http://liwenx.sourceforge.net">
   <liwenx:extension-point name="body">
        <service>
           <liwenx:component name="readAccount"/>
           <liwenx:component name="readMonitorizedAccountStats"/>
        </service>
   </liwenx:extension-point>

</page>
```

Si echamos un vistazo al componente *readAccounts*, definido en la plantilla *inner-page* y que lee todas las cuentas monitorizadas por un usuario, podemos observar cómo éstos deben implementar la interfaz *PageComponent* y por ello únicamente tienen que implementar el método *process()*, que recibe una petición a través del objeto *LiwenxRequest* y devuelve una respuesta mediante el objeto *LiwenxResponse*.

```
public class ReadAccounts implements PageComponent {

   private MonitorizedAccountManager monitorizedAccountManager;

   @Override
   public LiwenxResponse process(LiwenxRequest request) throws Exception {

        Element root = new Element("monitorized_accounts");
        SortedSet<MonitorizedAccount> accounts = monitorizedAccountManager.readAll();
        if (!CollectionUtils.isEmpty(accounts)) {
           for (MonitorizedAccount acc : accounts) {
                Element accElem = new Element("account");
                XmlUtil.addAttribute(accElem, "id", acc.getAccountId());
                String screenName = acc.getScreenName();
                if (screenName != null) {
                   XmlUtil.addAttribute(accElem, "name", screenName);
                }
                root.appendChild(accElem);
           }
        }

        return LiwenxResponseFactory.xmlResponse(root);
   }

   @Autowired
   public void setMonitorizedAccountManager(MonitorizedAccountManager
monitorizedAccountManager) {
        this.monitorizedAccountManager = monitorizedAccountManager;
   }
}
```

Aquí se observa como se leen las cuentas monitorizadas a través del *bean MonitorizedAccountManager*, que veremos en 3.2.4.2.Capa de lógica de negocio. Éste internamente sabe quién es el usuario identificado mediante la sesión Java y devuelve sus cuentas monitorizadas tras leerlas de la base de datos. Luego son iteradas y su información expuesta en formato *XML*, que se utiliza para construir la respuesta final. Este *XML* sustituirá en la página *XML* a la etiqueta *liwenx:component* que provocó la ejecución del mismo.

3.2.3 Modelo de datos

Antes de detallar el modelo de datos empleado conviene aclarar una serie de conceptos propios de *Cassandra* y, más concretamente, de la implementación *Datastax Enterprise*.

El concepto *tabla* en bases de datos relacionales equivale a *columnas de familias* en *Cassandra*. Éstas tienen una clave primaria que está compuesta de dos partes:

- *Partition key*: es la parte de la clave primaria que se utiliza para decidir en qué nodo se ubica cada fila. Cada nodo se reparte un rango de claves posibles.

- *Clustering columns*: son las columnas que junto con la *partition key* hace única a una fila. No es obligatorio que una clave primaria contenga *clustering columns*.

Datastax Enterprise proporciona un lenguaje similar al *SQL* pero con ciertas limitaciones, se trata de *CQL* [37]. Es interesante ver el comando *CREATE TABLE*, que es mediante el que se define el modelo de datos en la base de datos.

```
CREATE TABLE IF NOT EXISTS pfc.service_info_by_day(
       user_id bigint,
       year int,
       month int,
       day int,
       availability double,
       delay double,
       PRIMARY KEY(user_id, year, month, day)
);
```

Con el comando anterior se define una tabla con la columna *user_id* como *partition key* y las columnas *year, month* y *day* como *clustering columns.* Las otras dos columnas son simplemente columnas de datos.

Si quisiéramos tener una *partition key* compuesta simplemente habría que agrupar las columnas que la forman con paréntesis. Por ejemplo, si quisiéramos definir las columnas *user_id* y *year* como *partition key* el comando quedaría de la siguiente forma.

```
CREATE TABLE IF NOT EXISTS pfc.service_info_by_day(
       user_id bigint,
       year int,
       month int,
       day int,
       availability double,
       delay double,
       PRIMARY KEY((user_id, year), month, day)
);
```

En lo referente a consultas, *Cassandra sólo permite* consultar las filas mediante clave primaria, rango de claves o índices secundarios.

En cuanto a clave primaria puede ser únicamente mediante su *partition key, partition key* + algunas *clustering columns*, o mediante la clave primaria al completo. Si se utilizan sólo algunas *clustering columns*, es necesario que sea en el orden en que son definidas al crear la tabla. Por ejemplo, en la tabla anterior podríamos consultar sólo por *user_id*, por *user_id* + *year*, *user_id* + *year* + *month*, pero nunca por *user_id* + *month* o únicamente por *month*. En cuanto a consultas por rango de claves ocurre algo parecido: el rango sólo puede estar en la última *clustering column* por la que se filtra.

Por último, comentar que las consultas por índices secundarios están bastante limitadas ya que por razones de rendimiento sólo es aconsejable cuando los valores distintos de la columna que tiene el índice secundario no son muchos ni tampoco muy pocos, como por ejemplo dos.

Al definir un modelo de datos en una base de datos *NoSQL* hay que hacerlo desde una perspectiva muy distinta a cuando se hace para una base de datos relacional. En este segundo caso se realiza buscando cumplir las *Formas Normales* que nos garanticen la no redundancia de datos y la integridad de los mismos, así como minimizar problemas de actualización de las tablas [37].

Esto es posible entre otras cosas porque las bases de datos relacionales permiten hacer *JOINs* entre tablas. En la mayoría de las bases de datos *NoSQL*, y concretamente en *Cassandra*, esto no es posible. Esto hace que se deba definir el modelo de datos pensando en las consultas que se van a realizar en lugar de en los conceptos que representan los datos y las relaciones entre ellos.

Dicho de otra forma, todos los datos devueltos en una consulta deben estar contenidos en la misma familia de columnas. Esto es así incluso a riesgo de duplicar datos, en cuyo caso, la consistencia entre los mismos deberemos mantener por nuestra cuenta. En las bases de datos relacionales duplicar datos se ve como un problema, en *Cassandra* es una práctica muy común [38].

A continuación se muestra el modelo de datos definido mediante *CQL*.

```
CREATE KEYSPACE pfc with replication = {'class' : 'SimpleStrategy', 'replication_factor' : 1};

USE pfc;

CREATE TABLE IF NOT EXISTS pfc.tweet(
        user_id bigint,
        id bigint,
        created_date timestamp,
        is_rt boolean,
        text text,
        geo_longitude double,
        geo_latitude double,
        place_id text,
        place_fullname text,
        retweeted_user_id bigint,
        retweeted_twitter_id bigint,
        retweeted_created_date timestamp,
        PRIMARY KEY(user_id, id)
)
WITH CLUSTERING ORDER BY (id DESC);

CREATE TABLE IF NOT EXISTS pfc.twitter_user(
        user_id bigint,
        screen_name text,
        name text,
        PRIMARY KEY(user_id)
);
```

```
CREATE TABLE IF NOT EXISTS pfc.twitter_account(
        user_id bigint,
        screen_name text,
        name text,
        last_update timestamp,
        last_tweet_id bigint,
        first_read_completed float,
        PRIMARY KEY(user_id)
);

CREATE TABLE IF NOT EXISTS pfc.twitter_account_counter(
        user_id bigint,
        tweets counter,
        PRIMARY KEY(user_id)
);

CREATE TABLE IF NOT EXISTS pfc.user_twitter_account(
        user_id bigint,
        account_id bigint,
        account_order int,
        PRIMARY KEY(user_id,account_id)
);

CREATE TABLE IF NOT EXISTS pfc.user(
        user_id bigint,
        screen_name text,
        twitter_access_token text,
        twitter_token_secret text,
        email text,
        register_date timestamp,
        register_ip text,
        PRIMARY KEY(user_id)
);

CREATE TABLE IF NOT EXISTS pfc.user_by_email(
        email text,
        user_id bigint,
        PRIMARY KEY(email)
);


CREATE TABLE IF NOT EXISTS pfc.service_info_lifetime(
        user_id bigint,
        availability double,
        delay double,
        PRIMARY KEY(user_id)
);

CREATE TABLE IF NOT EXISTS pfc.service_info_by_year(
        user_id bigint,
        year int,
        availability double,
        delay double,
        PRIMARY KEY(user_id, year)
);

CREATE TABLE IF NOT EXISTS pfc.service_info_by_month(
        user_id bigint,
        year int,
        month int,
        availability double,
        delay double,
        PRIMARY KEY(user_id, year, month)
);
```

```
CREATE TABLE IF NOT EXISTS pfc.service_info_by_week(
        user_id bigint,
        year int,
        week int,
        availability double,
        delay double,
        PRIMARY KEY(user_id, year, week)
);

CREATE TABLE IF NOT EXISTS pfc.service_info_by_day(
        user_id bigint,
        year int,
        month int,
        day int,
        availability double,
        delay double,
        PRIMARY KEY(user_id, year, month, day)
);

CREATE TABLE IF NOT EXISTS pfc.service_info_by_hour(
        user_id bigint,
        year int,
        month int,
        day int,
        hour int,
        availability double,
        delay double,
        PRIMARY KEY(user_id, year, month, day, hour)
);

CREATE TABLE IF NOT EXISTS pfc.service_info_by_minute(
        user_id bigint,
        year int,
        month int,
        day int,
        hour int,
        minute int,
        availability double,
        delay double,
        PRIMARY KEY(user_id, year, month, day, hour, minute)
);
```

La Figura 3-3 muestra el modelo de datos en forma de diagrama entidad-relación. Hay que tener en cuenta que las relaciones no existen realmente en la base de datos. Las columnas marcadas con *K* forman parte de la *partition key* mientras que las marcadas con *C* son *clustering columns.*

Entrando más en detalle en el modelo, la familia de columnas *user* representa un usuario de nuestra aplicación. Además, se dispone de *user_by_email* para dar soporte a las consultas de usuarios por email. En este caso, en lugar de duplicar información de usuario, se ha decidido que esa tabla devuelva el *user_id* y realizar una segunda consulta a la familia de columnas *user*. El uso de esa consulta doble es bastante ocasional, únicamente en el *login*, y se puede asumir.

La familia de columnas *twitter_account* representa las distintas cuentas de servicios monitorizadas por los usuarios y están relacionadas con estos por *user_twitter_account*. La familia *twitter_account_counter* representa el contador de tuits importados para una cuenta de *Twitter* monitorizada. Los contadores son un tipo de datos específicos de *Cassandra* que permiten operaciones incrementales de actualización.

Figura 3-3. Diagrama en forma de entidad-relación del modelo de datos

La familia *tweet* representa los tuits almacenados para una cuenta. Se puede observar que está relacionada consigo misma cuando los tuits representan un *retuit*. Por otro lado, también se ayuda de la familia *twitter_user* para almacenar la información de los usuarios que hicieron el *retuit*. Mirando la clave primaria de *tweet* puede parecer redundante la columna *user_id* ya que un tuit está identificado únicamente por su *id*. Se ha añadido *user_id* como *partition key* para

forzar que todos los tuits de un mismo usuario estén en el mismo nodo *Cassandra* y así mejorar el rendimiento de sus consultas.

Por último, las familias *service_info_lifetime* y *service_info_by_xxx*, almacenarán los datos procesados de cada servicio monitorizado consolidados en los distintos intervalos de tiempo, es decir, las medias de disponibilidad y latencia globales, por año, por mes, por semana, por día, por hora y por minuto.

3.2.4 Estructura en capas

Se ha decidido dividir la aplicación en tres partes o capas principales. A cada capa se le confía una misión simple lo que permite escalar la arquitectura de una manera más sencilla. Además, a la hora de afrontar modificaciones en la aplicación será más cómodo y fácil ya que se trabajará sobre las capas requeridas en lugar de tener que estar revisando código mezclado. Por último, cuando se trabaja en equipos de desarrollo es posible que los distintos desarrolles se puedan abstraer del resto de capas repartiendo mejor las distintas tareas a realizar.

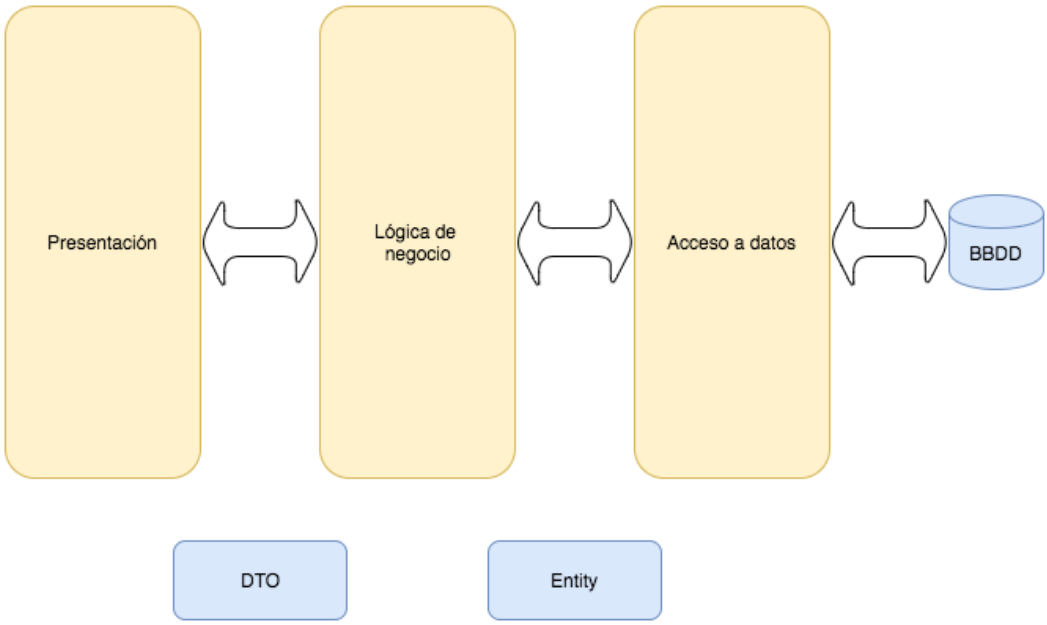

Figura 3-4. Esquema de la estructura de la aplicación en capas

En la Figura 3-4 se puede observar un esquema de la estructura de la aplicación en sus correspondientes capas:

- **Capa de acceso a datos**: se encarga de acceder a la base de datos en operaciones de lectura y escritura. Los *beans* de acceso a datos se denominan *DAOs (Data Access Object)* y los objetos que representan cada registro de las distintas tablas o columnas de familia se denominan entidades. Estas entidades viajan en la comunicación entre esta capa y la capa de lógica de negocio.
- **Capa de lógica de negocio**: se encarga de implementar las distintas funcionalidades propiamente dichas de la aplicación. Es aquí donde se establecen las distintas reglas que deben cumplirse. Los *beans* que dan soporte a esta capa los hemos denominado

managers y trabajan con las entidades que ofrece la capa de acceso a datos transformándolos en otros objetos independientes de la base de datos que se denominan *DTOs (Data Transfer Object)*. Esta capa intercambia estos objectos con la capa de presentación.

- **Capa de presentación**: es la capa que se le presenta al usuario, le comunica información y la captura. En este proyecto, esta capa estará compuesta tanto por el código *HTML/CSS/javascript* como por los *PageComponent* de *Liwenx*.

3.2.4.1 Capa de acceso a datos

La implementación de esta capa se realiza en el paquete *Java me.alexguerra.pfc.data.cassandra*. Puesto que el sistema de base de datos utilizado es *Datastax Enterprise*, el acceso a los datos se realizará a través de los *drivers* proporcionados por la plataforma. Concretamente, en el *pom.xml* de este módulo podemos encontrar las siguientes dependencias:

```
<dependency>
        <groupId>com.datastax.cassandra</groupId>
        <artifactId>cassandra-driver-core</artifactId>
        <version>${cassandra-driver.version}</version>
</dependency>
<dependency>
        <groupId>com.datastax.cassandra</groupId>
        <artifactId>cassandra-driver-mapping</artifactId>
        <version>${cassandra-driver.version}</version>
</dependency>
```

Se incluyen en el proyecto tanto las funcionalidades comunes del *driver*, el *core*, como un funcionalidad conocida como *object mapper* a través del módulo *mapping*. Esta última permite mapear las familias de columnas directamente como entidades a través de anotaciones *Java*. Esto facilita las operaciones de guardado y las consultas simples. Se puede observar en el siguiente extracto de código cómo se mapea la entidad *ServiceInfoByDayEntity*.

```
@Table(keyspace = "pfc", name = "service_info_by_day", readConsistency = "ONE",
writeConsistency = "ONE")
public class ServiceInfoByDayEntity implements Serializable {

   @PartitionKey
   @Column(name = "user_id")
   private long userId;

   @ClusteringColumn(0)
   @Column(name = "year")
   private int year;

   @ClusteringColumn(1)
   @Column(name = "month")
   private int month;

   @ClusteringColumn(2)
   @Column(name = "day")
   private int day;

   @Column(name = "availability")
   private double availability;

   @Column(name = "delay")
   private Double delay;
```

La anotación *@Table* indica el esquema y la familia de columnas que representa esta clase, así como la consistencia en lectura y en escritura. Además, cada una de las propiedades contiene anotaciones *@Column* que la relaciona con su correspondiente columna, y también existen anotaciones para indicar la *partition key* y las *clustering columns*.

Para facilitar la creación de los distintos *DAO* del proyecto se ha abstraído la implementación mediante genéricos de *Java, creado* la interfaz *GenericDAO<T, ID>,* que especifica las operaciones de guardado, borrado y acceso por clave primaria, donde *T* es la entidad e *ID* el objeto que representa su clave primaria; y una implementación base, *AbstractDAO<T, ID>*.

GenericDAO

```
public interface GenericDAO<T,ID> {

   T findById(ID id);

   void save(T t);

   void save(Collection<T> ts);

   void delete(ID id);
}
```

AbstractDAO

```
public abstract class AbstractDAO<T, ID> implements GenericDAO<T, ID>, InitializingBean {

   private final Log log = LogFactory.getLog(getClass());

   private ClusterManager clusterManager;
   private MappingManager mappingManager;
   private Mapper<T> mapper;

   protected ClusterManager getClusterManager() {
        return clusterManager;
   }

   public void afterPropertiesSet() throws Exception {
        mappingManager = new MappingManager(clusterManager.getSession());
   }

   protected Select buildSelect(){
        return QueryBuilder.select().all().from(getMapper().getTableMetadata());
   }

   protected Mapper<T> getMapper() {
        if (mapper == null) {
           Type superClass = getClass().getGenericSuperclass();
           Type t = ((ParameterizedType) superClass).getActualTypeArguments()[0];
           try {
                @SuppressWarnings("unchecked")
                Class<T> clazz = (Class<T>) Class.forName(t.getTypeName());
                mapper = mappingManager.mapper(clazz);
           } catch (ClassNotFoundException e) {
                log.fatal(e.getMessage(), e);
           }
        }

        return mapper;
   }
```

```
    protected Session getSession(){
        return clusterManager.getSession();
    }

    public T findById(ID id) {
        return getMapper().get(getPrimaryKeyObjects(id));
    }

    protected Object[] getPrimaryKeyObjects(ID id){
        Object[] objs;
        if (id instanceof ComplexPrimaryKey) {
           objs = ((ComplexPrimaryKey) id).getPrimaryKeyObjects();
        } else {
           objs = new Object[]{id};
        }

        return objs;
    }

    public void save(T t) {
        getMapper().save(t);
    }

    public void save(Collection<T> ts) {
        BatchStatement batch = new BatchStatement();
        for(T t:ts){
           batch.add(getMapper().saveQuery(t));
        }
        getSession().execute(batch);
    }

    public void delete(ID id) {
        getMapper().delete(getPrimaryKeyObjects(id));
    }

    @Autowired
    public void setClusterManager(ClusterManager clusterManager) {
        this.clusterManager = clusterManager;
    }

}
```

Es interesante la construcción del objeto *Mapper* mediante genéricos en el método *getMapper()*. Las operaciones de guardado y borrado quedan delegadas en este objeto. También resulta interesante el uso del *QueryBuilder* del driver para crear consultas en el método *buildSelect()*. En ningún momento se escriben consultas *CQL* de forma directa.

Por último, el método *findById()* hace uso de la clave primaria de la entidad para realizar la consulta. Para poder definir *DAOs* de entidades con clave primaria compleja, *type ID*, se ha creado la interfaz *ComplexPrimaryKey que únicamente exige saber los nombres de la clave primaria en el orden correcto.*

```
public interface ComplexPrimaryKey extends Serializable{

   Object[] getPrimaryKeyObjects();
}
```

El hecho de haber construido esta implementación genérica hace que la creación de un *DAO* concreto se reduzca a heredar de *AbstractDAO* y poco más, salvo que se quiera hacer alguna consulta que no sea por clave primaria completa.

```
@Repository
public class UserByEmailDAOImpl extends AbstractDAO<UserByEmailEntity, String> implements
UserByEmailDAO{

}
```

En cualquier caso, si se desea implementar algún método para consultas específicas, los métodos de utilidad proporcionados por *AbstractDAO* lo hacen muy sencillo. En el siguiente extracto se puede observar la implementación del método *readByUser()* de *UserTwitterAccountDAOImpl*, que permite leer las cuentas de *Twitter* monitorizadas por un usuario a partir de su id.

```
@Repository
public class UserTwitterAccountDAOImpl extends AbstractDAO<UserTwitterAccountEntity,
UserTwitterAccountPk> implements UserTwitterAccountDAO {

   @Override
   public List<UserTwitterAccountEntity> readByUser(long userId) {

        Mapper<UserTwitterAccountEntity> mapper = getMapper();
        Where select = buildSelect().where(eq("user_id", userId));

        Result<UserTwitterAccountEntity> entities =
mapper.map(getSession().execute(select));

        return entities.all();
   }

}
```

Para finalizar con esta capa, comentar la anotación *@Repository* que llevan todos los *DAO* en sus implementaciones. Se trata de un marcado que permite instanciar todos los *DAO* mediante una única línea de configuración de *Spring* en el fichero *liwenx-services.xml* en lugar de instanciarlos uno a uno con sus etiquetas *<bean/>*.

```
<!-- DAOs -->
<context:component-scan base-package="me.alexguerra.pfc.data.cassandra.impl" />
```

Con esta línea, *Spring* busca en el paquete indicado todos los *beans* anotados con *@Repository, @Component* o *@Service,* y los instancia, resolviendo también sus dependencias, como la del *ClusterManager* que contiene la configuración de acceso a la base de datos.

3.2.4.2 Capa de lógica de negocio

Esta capa utiliza la capa anterior a través de los *DAOs* e intercambia con ella información a través de los objetos entidades. A su vez, esta capa ofrece una serie de servicios o *managers* a la

capa superior y realiza el intercambio de información a través de los *DTOs*. Estos servicios ya contemplan el control de permisos, control de errores y validación de los datos de entrada. Es por ello por lo que realmente se puede considerar esta capa como la propia *API* de la aplicación. Sobre ella se pueden construir tanto servicios web de distinto tipo (REST, SOAP, binarios,...), una aplicación web, como es el caso de este proyecto o cualquier cosa que se nos ocurra. Aquí reside la lógica fundamental de la misma.

3.2.4.2.1 DTOs

Los *DTOs* de la aplicación se encuentran en el paquete *me.alexguerra.pfc.data.dto* y suelen ser simples mapeos de las entidades eliminando cualquier vinculación con la base de datos. Éstos deben ser libres de residir fuera de esta capa.

Sin embargo, existen algunos casos en los que conviene que difieran de las entidades. El primer caso es cuando conviene que una propiedad sea de un tipo de datos distinto al que se guarda en base de datos. Por ejemplo, las propiedades que toman un valor de entre unos predefinidos se suelen expresar en los *DTO* como *enum* y en las entidades como un número entero. También cuando la base de datos nos obliga a usar objetos *Date* en lugar de *Calendar*, que son más manejables.

Otro ejemplo sería cuando la entidad guarda una referencia a otra entidad. En la entidad se suele guardar únicamente la clave primaria de la otra entidad y en el *DTO* puede que nos convenga tener el objeto completo. Por supuesto, esto requeriría de una segunda consulta para obtener esos datos.

A continuación se puede observar como la entidad *TweetEntity* guarda la clave primaria del usuario propietario del tuit, *userId*, mientras que en el *DTO Tweet* se guarda una propiedad de tipo *TwitterUser* con todos los datos del usuario, *user*. También se puede observar la propiedad *createdDate* que se representa mediante un objeto tipo *Date* en la entidad y *Calendar* en el *DTO*.

TweetEntity

```
@Table(keyspace = "pfc", name = "tweet", readConsistency = "ONE", writeConsistency =
"ONE")
public class TweetEntity implements Serializable {

   private static final long serialVersionUID = -1813158825262638702L;

   @PartitionKey
   @Column(name = "user_id")
   private long userId;

   @ClusteringColumn
   @Column(name = "id")
   private long id;

   @Column(name = "created_date")
   private Date createdDate;

   […]
```

Tweet

```
public class Tweet implements Serializable {

   private static final long serialVersionUID = -4821981860449666846L;

   private TwitterUser user;

   private long id;

   private Calendar createdDate

   […]
```

Un segundo caso de diferencias entre entidad y *DTO* es cuando varias propiedades de la entidad son representadas por un única propiedad en el *DTO*. Es el caso del *token* de acceso de *Twitter*, que consta de dos partes almacenadas en dos columnas de la entidad *UserEntity*, *twitterAccessToken* y *twitterTokenSecret*, y de única propiedad de tipo *AccessToken*, que contiene las dos partes, en el *DTO*.

UserEntity

```
@Table(keyspace = "pfc", name = "user", readConsistency = "ONE", writeConsistency =
"ONE")
public class UserEntity implements Serializable{

   private static final long serialVersionUID = 2241534209925810178L;

   @PartitionKey
   @Column(name = "user_id")
   private long id;

   @Column(name = "screen_name")
   private String screenName;

   @Column(name = "twitter_access_token")
   private String twitterAccessToken;

   @Column(name = "twitter_token_secret")
   private String twitterTokenSecret;

   @Column(name = "email")
   private String email;

...[…]
```

User

```
public class User implements Serializable, UserDetails {

   private static final long serialVersionUID = 6461873700148828272L;

   private long id;

   private String screenName;

   private AccessToken twitterAccessToken;

   private String email;

   […]
```

Esto también puede ocurrir no sólo con las propiedades sino con las propias entidades. Por ejemplo, el *DTO MonitorizedAccount* representa a la vez a las entidades correspondientes a las familias de columnas *user_twitter_account* y *twitter_account* (ver Figura 3-3) y, en un caso más extremo, el *DTO ServiceInfo*, que permite representar a las 7 entidades correspondientes a las familias de columnas *service_info_xxx*, ya que contiene todas las propiedades posibles de sus claves primarias permitiendo valores nulos. Obsérvese que éstas son objetos *Integer* en lugar de tipos primarios *int*.

MonitorizedAccount

```
public class MonitorizedAccount implements Serializable, Comparable<MonitorizedAccount> {

   private static final long serialVersionUID = 7989493835731437040L;

   private long accountId;

   private int order;

   private String screenName;

   private String name;

   private Calendar lastUpdate;

   private float firstReadCompleted;

   private long tweets;
```

ServiceInfo

```
public class ServiceInfo implements Serializable{

   private static final long serialVersionUID = -3556417998732221370L;

   private long userId;

   private Integer year;

   private Integer month;

   private Integer week;

   private Integer day;

   private Integer hour;

   private Integer minute;

   private double availability;

   private Double delay;
```

Un tercer caso de diferencias entre entidad y *DTO* es cuando queremos esconder una propiedad. El valor existe en la base de datos y por tanto en la entidad, pero no queremos hacerlo público a través del *DTO*. Este caso no se ha dado en este proyecto.

En definitiva, los *DTOs* representan los datos de la aplicación más allá de la capa de lógica de negocio, y aunque como hemos visto se pueden mapear directamente con las entidades, tenemos cierta flexibilidad para esconder detalles de implementación o adaptarlos para trabajar con ellos de otra forma.

Para transformar las entidades a *DTOs* se han utilizado una serie de *beans* que implementan la interfaz de la librería *guava com.google.common.base.Function<F, T>*, que define un método para transformar objetos tipo *F* en tipo *T*. Todos estos *beans* se encuentran en el paquete *me.alexguerra.pfc.data.dto.functions* y se les denominan *functions*.

```
public interface Function<F, T> {
   T apply(F input);
   […]
}
```

3.2.4.2.2 Servicios

Las interfaces de los distintos servicios o *managers* implementados se encuentran en el paquete principal de la aplicación, *me.alexguerra.pfc*, mientras que sus implementaciones se encuentran en el paquete *me.alexguerra.pfc.impl*. Estos servicios son *AuthenticationManager, UserManager, MonitorizedAccountManager, TweetManager, ServiceInfoManager* y por último *TweetCollector*.

AuthenticationManager

Se encarga de ofrecer información acerca del usuario identificado en el sistema si lo hubiera. También ofrece un método para refrescar la información del usuario identificado, útil cuando éste se modifica por algún otro servicio.

```
public interface AuthenticationManager {

   /**
    * It returns the logged user.
    *
    * @return logged user or null if there's no logged user.
    */
   User getLoggedUser();

   /**
    * It updates current logged user.
    */
   void updateLoggedUser();
}
```

UserManager

Se encarga de ofrecer métodos para la lectura y el guardado de usuarios de la aplicación.

```
public interface UserManager {

   /**
    * It reads an user by id
    * @param id - user id
    * @return user or null if there's no user with specified id
    */
   User read(long id);

   /**
    * It saves or create an user
    *
    * @param user - user to save
    * @return saved user
    * @throws ForbiddenOperationException - if user exists and user is not logged user
    */
   User save(User user) throws ForbiddenOperationException;

   /**
    * It reads an user by email
    *
    * @param email - user email
    * @return user or null if there's no user with specified email
    */
   User readByEmail(String email);

}
```

Se puede observar en la implementación cómo se hace uso de los distintos *DAOs*, *functions* y de otros *managers* como *AuthenticationManager* para obtener el usuario logado y hacer la comprobación de permisos. Obsérvese como la implementación de los dos métodos de lectura es muy sencilla debido a que se hacen uso de los *DAOs* y los *functions*. En cuanto al método *save()*, es interesante observar cómo al crear un usuario se mantiene la tabla *user_by_email* a través de su correspondiente *DAO*.

```
@Service("userManager")
public class UserManagerImpl extends AbstractManager implements UserManager,
UserDetailsService{

   private UserDAO userDAO;
   private UserByEmailDAO userByEmailDAO;
   private UserEntity2DTO userEntity2DTO;
   private AuthenticationManager authManager;

   @Override
   public User read(long id) {
        return userEntity2DTO.apply(userDAO.findById(id));
   }

   @Override
   public User readByEmail(String email) {
        User user;
        UserByEmailEntity e = userByEmailDAO.findById(email);
        if(e!=null){
           user = read(e.getId());
        }else{
           user=null;
        }

        return user;
   }
```

```
    @Override
    public User save(User user) throws ForbiddenOperationException {
        Preconditions.checkNotNull(user, getMessage(MSG_PRECOND_NOTNULL, User.class));

        User loggedUser = authManager.getLoggedUser();

        UserEntity e = userDAO.findById(user.getId());
        boolean isCreation = (e==null);
        if(isCreation){
           if(loggedUser != null){
               throw new
ForbiddenOperationException(getMessage("userManager.create.forbidden", loggedUser,
user));
           }
           e = new UserEntity();
           e.setId(user.getId());
           e.setRegisterDate(Calendar.getInstance().getTime());
           e.setRegisterIp(user.getRegisterIp());
        }else{
           if(loggedUser == null || !loggedUser.equals(user)){
               throw new
ForbiddenOperationException(getMessage("userManager.save.forbidden", loggedUser, user));
           }
        }

        e.setScreenName(user.getScreenName());
        e.setEmail(user.getEmail());
        AccessToken accessToken = user.getTwitterAccessToken();
        if(accessToken!=null){
           e.setTwitterAccessToken(accessToken.getToken());
           e.setTwitterTokenSecret(accessToken.getTokenSecret());
        }

        userDAO.save(e);
        if(isCreation){
           UserByEmailEntity ube = new UserByEmailEntity();
           ube.setEmail(e.getEmail());
           ube.setId(e.getId());
           userByEmailDAO.save(ube);
        }

        return userEntity2DTO.apply(e);

    }
[…]
```

Por último, conviene advertir la anotación *@Service*, que permite instanciar los *beans* de manera similar a como se comentó en el apartado 3.2.4.1.Capa de acceso a datos.

MonitorizedAccountManager

Se encarga de gestionar las cuentas monitorizadas por un usuario de la aplicación. Podemos observar métodos de lectura y guardado de las mismas.

```
public interface MonitorizedAccountManager {

   /**
    * It reads an twitter account monitorized by logged user
    *
    * @param accountId - account id
    * @return monitorized account or null if logged user is not monitorizing that account
    * @throws ForbiddenOperationException - if there's no logged user
    */
   MonitorizedAccount read(long accountId) throws ForbiddenOperationException;

   /**
    * It reads all accounts monitorized by logged user
    * @return set of accounts
    * @throws ForbiddenOperationException - if there's no logged user
    */
   SortedSet<MonitorizedAccount> readAll() throws ForbiddenOperationException;

   /**
    * It saves monitorized account
    *
    * @param account - account
    * @throws ForbiddenOperationException - if there's no logged user
    */
   void save(MonitorizedAccount account) throws ForbiddenOperationException;

}
```

```
@Override
public void save(MonitorizedAccount account) throws ForbiddenOperationException {
   Preconditions.checkNotNull(account, getMessage(MSG_PRECOND_NOTNULL,
MonitorizedAccount.class));

   User loggedUser = authManager.getLoggedUser();
   if (loggedUser != null) {
        UserTwitterAccountPk pk = new UserTwitterAccountPk();
        pk.setUserId(loggedUser.getId());
        pk.setAccountId(account.getAccountId());

        UserTwitterAccountEntity e = userTwitterAccountDAO.findById(pk);
        if (e == null) {
           e = new UserTwitterAccountEntity();
           e.setUserId(loggedUser.getId());
           e.setAccountId(account.getAccountId());
        }

        e.setOrder(account.getOrder());
        userTwitterAccountDAO.save(e);

        TwitterAccountEntity tae = twitterAccountDAO.findById(account.getAccountId());
        if (tae == null) {
           tae = new TwitterAccountEntity();
           tae.setId(account.getAccountId());
           tae.setScreenName(account.getScreenName());
           tae.setName(account.getName());
           tae.setFirstReadCompleted(0);
           twitterAccountDAO.save(tae);
           tweetCollector.queueTwitterAccount(tae);
        }
   } else {
        throw new
ForbiddenOperationException(getMessage("monitorizedAccount.save.forbidden",
account.getAccountId(), loggedUser));
   }

}
```

En la implementación de este servicio es interesante ver el método *save()*. Obsérvese cómo cuando un usuario añade una cuenta de *Twitter* para monitorizar, si no se tienen datos de la misma (objeto *TwitterAccountEntity)*, ésta se crea y se encola en el servicio *TweetCollector*. Esto permite que la recolección de tuits empiece lo más pronto posible, normalmente de manera inmediata, y no haya que espera a la siguiente recolección periódica, consiguiendo una mejor experiencia de usuario tal y como se comentó al inicio de este capítulo.

TweetManager

Permite leer los tuits de una cuenta de *Twitter* monitorizada.

```
public interface TweetManager {

   /**
    * It read tweets from a twitter account
    *
    * @param userId - user id
    * @param offsetId - max tweet id to read (exclusive). It can be null
    * @param pageSize - page size
    * @return
    */
   List<Tweet> readByUser(long userId, Long offsetId, int pageSize);

}
```

En la implementación de este servicio es interesante ver cómo se construye el *DTO Tweet*. Anteriormente comentábamos que éste contenía un objeto *TwitterUser* para completar más la información de un tuit respecto a la entidad, que contenía únicamente el id del usuario. Se comentó que esto implicaba una lectura opcional, pero para evitar una lectura tuit por tuit y sus consiguientes problemas de rendimiento, se puede observar cómo al transformarlos en la primera parte del método, se van almacenando en el *multimap usersToUpdate* los usuarios a leer y los tuits que habría que actualizar. Posteriormente, se leen de una vez todos los usuarios que se necesiten y se actualizan todos los objetos *Tweets* que lo requieran.

```
@Override
public List<Tweet> readByUser(long userId, Long offsetId, int pageSize) {

   List<Tweet> tweets = null;
   List<TweetEntity> tweetEntities = tweetDAO.readByUser(userId, offsetId, pageSize);
   if (!CollectionUtils.isEmpty(tweetEntities)) {
        Multimap<Long, Tweet> usersToUpdate = HashMultimap.create();
        tweets = new ArrayList<>(tweetEntities.size());
        for (TweetEntity e : tweetEntities) {
           Tweet t = tweetEntity2DTO.apply(e);
           tweets.add(t);
           // Users to refresh
           usersToUpdate.put(t.getUser().getId(), t);
           TwitterUser rtUser = t.getRetweetedUser();
           if(rtUser !=null){
               usersToUpdate.put(rtUser.getId(), t);
           }
        }

        // Refresh users
        List<TwitterUserEntity> users = twitterUserDAO.readByIds(new
ArrayList<>(usersToUpdate.keySet()));
```

```
            if(!CollectionUtils.isEmpty(users)){
               for(TwitterUserEntity e:users){
                    Collection<Tweet> ts = usersToUpdate.get(e.getId());
                    if(!CollectionUtils.isEmpty(ts)){
                       TwitterUser u = twitterUserEntity2DTO.apply(e);
                       for(Tweet t:ts){
                            TwitterUser tu = t.getUser();
                            TwitterUser rtu = t.getRetweetedUser();
                            if(tu.getId() == u.getId()){
                               t.setUser(u);
                            }else if(rtu!=null && rtu.getId() == u.getId()){
                               t.setRetweetedUser(u);
                            }
                       }
                    }
               }
            }
         }

         return tweets;
      }
```

ServiceInfoManager

Se encarga de ofrecer acceso a la información procesada de las cuentas de *Twitter* monitorizadas, es decir, a las estadísticas de los servicios. Como ya se comentó anteriormente, el *DTO ServiceInfo* es capaz de mapear las 7 entidades de estadísticas de servicio, ocultando esa complejidad y manteniendo el uso de este servicio simple.

```
      public interface ServiceInfoManager {

         /**
          * It reads service info in a specified range of time with a specified granularity.
          *
          * @param serviceId - twitter account id
          * @param granularity - time granularity
          * @param timeFilter - time filter. It can be null
          * @return
          */
         List<ServiceInfo> read(long serviceId, TimeGranularity granularity, TimeFilter
      filter);

         /**
          * It reads lifetime stats for a collection of services
          *
          * @param serviceIds - service ids
          * @return
          */
         Map<Long, ServiceInfo> readLifetimeByIds(List<Long> serviceIds);
      }
```

El método *readLifetimeByIds()* permite obtener las estadísticas globales de varios servicios a la vez. El método *read()* permite filtrar por servicio, granularidad temporal y por ventana de tiempo. A continuación se muestran cómo son estos objetos.

TimeFilter

Obsérvese cómo al ser las propiedades *Integer* en lugar de *int*, éstas pueden ser nulas y se puede filtrar únicamente por un subconjunto de ellas.

```
public class TimeFilter implements Serializable{

   private static final long serialVersionUID = 2292557478485005979L;

   public Integer year;

   public Integer month;

   public Integer week;

   public Integer day;

   public Integer hour;

   public Integer minute;
```

TimeGranularity

```
public enum TimeGranularity {
   LIFETIME(0), YEAR(1), MONTH(2), WEEK(3), DAY(4), HOUR(5), MINUTE(6);

   private int id;

   private TimeGranularity(int id) {
        this.id = id;
   }

   public int getId() {
        return id;
}

   public static TimeGranularity getById(int id){
        TimeGranularity tg = null;
        TimeGranularity[] values = values();
        for(int i=0;i<values.length && tg==null;i++){
           TimeGranularity t = values[i];
           if(t.getId()==id){
                tg = t;
           }
        }

        return tg;
   }
}
```

TweetCollector

Es uno de los servicios más importantes de la aplicación. Su interfaz es muy sencilla ya que cuenta únicamente con un método para encolar cuentas a monitorizar. Éste método es usado por el servicio *MonitorizedAccountManager* como ya hemos visto anteriormente. Sin embargo, la parte más importante reside en su implementación, ya que contiene varios hilos encargados de la recolección de los tuits de las cuentas monitorizadas.

```
public interface TweetCollector {

   /**
    * It add specified {@link User} to the queue
    *
    * @param user - user to add.
    */
   void queueTwitterAccount(TwitterAccountEntity e);
}
```

Conviene ver la implementación por partes. En primer lugar, veamos las propiedades que tiene este *bean*, los distintos *beans* de los que depende y la implementación del método *queueTwitterAccount()*.

```
public class TweetCollectorImpl extends AbstractManager implements TweetCollector,
InitializingBean, DisposableBean {

   private final Log log = LogFactory.getLog(getClass());

   private final BlockingQueue<TwitterAccountEntity> priorityQueue = new
ArrayBlockingQueue<>(100000, true);
   private final BlockingQueue<TwitterAccountEntity> normalQueue = new
ArrayBlockingQueue<>(100000, true);
   private final ConcurrentMap<Long, Integer> processingSummaries = new
ConcurrentHashMap<>(); // key account id -
                       // value thread id

   private TwitterAccountDAO twitterAccountDAO;
   private UserDAO userDAO;
   private TwitterHelper twitterHelper;
   private TweetDAO tweetDAO;
   private TwitterUserDAO twitterUserDAO;
   private TwitterAccountCounterDAO twitterAccountCounterDAO;

   private int accountUpdateLimit = 5; // in minutes
   private int twitterPageSize = 100;
   private int twitterMaxTweets = 3200;
   private ScheduledExecutorService executor;
   private ScheduledExecutorService executor2;
   private boolean stopping = false;
   private int threadPool = 10;
   private Calendar nextRateLimitWindow;

   @Override
   public void queueTwitterAccount(TwitterAccountEntity acc) {
       try {
          if (normalQueue.contains(acc)) {
              normalQueue.remove(acc);
          }
          priorityQueue.offer(acc, 1, TimeUnit.SECONDS);
       } catch (InterruptedException e) {
          log.error(getMessage("tweetCollector.queueWebUser.error", acc), e);
       }
   }
```

Podemos observar que se definen dos colas, una prioritaria y otra con prioridad normal. El método *queueTwitterAccount()* encola la cuenta a monitorizar en la cola prioritaria, eliminándola de la normal si ya estuviera ahí.

Por otro lado, también se definen los *DAOs* a usar y algunos parámetros de configuración como el tiempo de actualización, parámetros de paginación de *Twitter* y el número de hilos recolectores de tuits que van a correr en paralelo.

A continuación se muestran los métodos que se ejecutan al inicio y al final del ciclo de vida de este *bean*: *afterPropertiesSet()* y *destroy()*, definidos por las interfaces de *Spring InitalizingBean* y *DisposableBean*. Al iniciarse el *bean* se generan dos *ScheduledExecutorServices* [40] que permiten la ejecución periódica del hilo *TweetCollectorConductor*, encargado de encolar las cuentas monitorizadas para su actualización; y una serie de hilos recolectores *TweetCollectorThread*, que serán los encargados de coger cuentas de las colas y actualizarlas con los últimos tuits. El método *destroy()* por su parte realiza la parada ordenada de todos los hilos en el momento de parar la aplicación.

```
@Override
public void afterPropertiesSet() throws Exception {
   if (threadPool != 0) {
        executor2 = Executors.newScheduledThreadPool(1);
        executor2.scheduleAtFixedRate(new TweetCollectorConductor(), 30, 120,
TimeUnit.SECONDS);
        executor = Executors.newScheduledThreadPool(threadPool);
        for (int i = 0; i < threadPool; i++) {
           executor.scheduleWithFixedDelay(new TweetCollectorThread(i), 20000, 500,
TimeUnit.MILLISECONDS);
        }
   }
}

@Override
public void destroy() throws Exception {
   if (threadPool != 0) {
        executor.shutdown();
        executor2.shutdown();
        stopping = true;
        while (!executor.awaitTermination(100, TimeUnit.MILLISECONDS)) {
           // do nothing
        }
        while (!executor2.awaitTermination(100, TimeUnit.MILLISECONDS)) {
           // do nothing
        }
   }

}
```

En el siguiente bloque de código se muestra el hilo *TweetCollectorConductor*. Se puede observar cómo se leen todas las cuentas monitorizadas y se encolan las cuentas desactualizadas, que dependerán del valor de la variable *accountUpdateLimit*. Las cuentas en las que no se ha completado la primera importación de tuits se meten en la cola prioritaria.

```
    private class TweetCollectorConductor implements Runnable {

       @Override
       public void run() {
            try {
               NavigableSet<TwitterAccountEntity> summs = twitterAccountDAO.readAll();
               if (!CollectionUtils.isEmpty(summs)) {
                    TwitterAccountEntity limit = buildLimit();
                    SortedSet<TwitterAccountEntity> outdated = summs.headSet(limit);
                    int queuedUsers = 0;
                    for (TwitterAccountEntity a : outdated) {
                       if (!processingSummaries.containsKey(a.getId()) &&
    !priorityQueue.contains(a) && !normalQueue.contains(a)) {
                            if (a.getFirstReadCompleted() != 1f) {
                               priorityQueue.offer(a, 10, TimeUnit.SECONDS);
                               queuedUsers++;
                            } else {
                               normalQueue.offer(a, 10, TimeUnit.SECONDS);
                               queuedUsers++;
                            }
                       }
                    }
                    if (log.isTraceEnabled()) {
                       log.trace(getMessage("tweetCollector.updatingQueue.task.
    collectedOutdatedSummaries.some", summs.size(), queuedUsers));
                    }
               } else {
                    if (log.isTraceEnabled()) {
                       log.trace(getMessage("tweetCollector.updatingQueue.task.
    collectedOutdatedSummaries.some", 0, 0));
                    }
               }
            } catch (Exception e) {
               log.error(getMessage("tweetCollector.updatingQueue.task.error"), e);
            }

       }
    }

    private TwitterAccountEntity buildLimit() {
       TwitterAccountEntity e = new TwitterAccountEntity();
       e.setFirstReadCompleted(1f);
       Calendar limit = Calendar.getInstance();
       limit.add(Calendar.MINUTE, -accountUpdateLimit);
       e.setLastUpdate(limit.getTime());
       return e;
    }
```

El hilo *TweetCollectorThread* es el encargado de tomar la siguiente cuenta a procesar de las colas, leer los tuits de *Twitter* y guardarlos en la base de datos. A continuación se puede ver el método principal del hilo y cómo se conecta con *Twitter* a través de la librería *Twitter4j* con los *tokens* de acceso de los usuarios de la aplicación. Estos se van rotando para evitar alcanzar el límite de conexiones dada por *Twitter* [41]. Mientras más usuarios tengamos en la aplicación, mayor será la capacidad de lectura de tuits. Se puede observar también cómo se van obteniendo los tuits paginados y cómo el comportamiento es distinto si se trata de la primera importación, en la que se avanza del tuit más reciente al más antiguo que permite leer *Twitter* y se guardan página a página; a si se trata de una importación de tuits posterior, en las que se avanza desde el tuit más antiguo importado, al más actual y se guardan todas las páginas de una vez.

```
private class TweetCollectorThread implements Runnable {

   private int id;
   private List<UserEntity> users;
   private Iterator<UserEntity> usersIt;
   private List<UserEntity> nextUsers;

   private TweetCollectorThread(int id) {
       this.id = id;
   }

   @Override
   public void run() {
       try {
          TwitterAccountEntity tae = getNextSummary();
          if(tae!=null){
              users = userDAO.readAll();
              if (!CollectionUtils.isEmpty(users)) {
                 usersIt = users.iterator(); // users iterator
                 nextUsers = new ArrayList<>(); // next round of users
                 UserEntity currentUser = usersIt.next();
                 Calendar now = Calendar.getInstance();
                 while (currentUser != null && tae != null && !stopping &&
(nextRateLimitWindow == null || now.after(nextRateLimitWindow))) {
                     if (log.isTraceEnabled()) {
                        log.trace(getMessage("tweetCollector.collector.nextSummary", id,
tae.getId()));
                     }

                     try {
                        processingSummaries.putIfAbsent(tae.getId(), id);
                        Twitter twitter = null;
                        twitter4j.User credentials = null;
                        do {
                            try {
                               twitter =
twitterHelper.getTwitterService(buildAccessToken(currentUser));
                               credentials = twitter.verifyCredentials();
                            } catch (TwitterException e) {
                               if (e.getStatusCode() == 401 || e.getStatusCode() == 404)
{
                                   currentUser = getNextUser();
                               }
                            }
                        } while (credentials == null && currentUser != null);

                        if (credentials != null) {
                            boolean firstTime = tae.getFirstReadCompleted() != 1;
                            Integer firstTimeTotalTweets = null;
                            if (firstTime) {
                               firstTimeTotalTweets =
Math.min(credentials.getStatusesCount(), twitterMaxTweets);
                            }
                            Long lastTweetId = tae.getLastTweetId();
                            int page = 1;
                            Paging p = new Paging();
                            p.setCount(twitterPageSize);
                            p.setPage(page);
                            if (lastTweetId != null) {
                               if (firstTime) {
                                   p.setMaxId(lastTweetId-1);
                               } else {
                                   p.setSinceId(lastTweetId);
                               }
                            }
```

```
                                List<TweetEntity> accumulatedTweets = new
ArrayList<TweetEntity>();
                                boolean end = false;
                                Set<TwitterUserEntity> savedUsers = new HashSet<>();
                                Set<TwitterUserEntity> usersToSave = new HashSet<>();
                                while (!end) {
                                   List<TweetEntity> tweets = readTweets(twitter, tae, p,
usersToSave);
                                   if (!CollectionUtils.isEmpty(tweets)) {
                                        if (firstTime) {
                                           saveTweets(tweets, firstTimeTotalTweets,
usersToSave, savedUsers, tae.getId());
                                           TweetEntity lastTweet = tweets.get(tweets.size() -
1);
                                           p.setMaxId(lastTweet.getId()-1);
                                           p.setPage(1);
                                        } else {
                                           page++;
                                           p.setPage(page);
                                           accumulatedTweets.addAll(tweets);
                                        }
                                   } else {
                                        if (firstTime) {
                                           forceFirstReadToCompleted(tae.getId());
                                        }
                                        end = true;
                                   }
                                }
                                if(!accumulatedTweets.isEmpty()){
                                   saveTweets(accumulatedTweets, firstTimeTotalTweets,
usersToSave, savedUsers, tae.getId());
                                }
                                nextUsers.add(currentUser);
                             }
                          } catch (TwitterException e) {
                             if (e.getStatusCode() == 429) { // Rate limit
                                RateLimitStatus limitStatus = e.getRateLimitStatus();
                                if (limitStatus != null) {
                                   int secondsUntilReset =
limitStatus.getSecondsUntilReset();
                                   Calendar nextLimit = Calendar.getInstance();
                                   nextLimit.add(Calendar.SECOND, secondsUntilReset);
                                   nextRateLimitWindow = nextLimit;
                                   log.info(getMessage("tweetCollector.collecting.rate-
limit", secondsUntilReset, priorityQueue.size(), normalQueue.size()));
                                } else {
                                   log.error(getMessage("tweetCollector.collecting.error",
tae), e);
                                }
                             } else {
                                log.error(getMessage("tweetCollector.collecting.error",
tae), e);
                             }
                          } catch (Exception e) {
                             log.error(getMessage("tweetCollector.collecting.error", tae),
e);
                          } finally {
                             processingSummaries.remove(tae.getId());
                          }
                          currentUser = getNextUser();
                          tae = getNextSummary();
                       }
                    }
                 }
              } catch (Exception e) {
                 log.error(getMessage("tweetCollector.collecting.task.error"), e);
              }
           }
```

Dentro de este hilo, es interesante echar un vistazo al método que guarda los tuits, *saveTweets()*. Se puede observar cómo se guardan los tuits, los usuarios involucrados en los mismos y cómo se actualiza la tabla *twitter_account* así como la que contiene el contador de tuits, *twitter_account_counter*.

```
    private void saveTweets(List<TweetEntity> tweets, Integer firstTimeTotalTweets,
Set<TwitterUserEntity> usersToSave, Set<TwitterUserEntity> savedUsers, long accountId){
        if (!tweets.isEmpty()) {

            tweetDAO.save(tweets);

            if(!CollectionUtils.isEmpty(usersToSave)){
                Set<TwitterUserEntity> effUsersToSave = new HashSet<>();
                effUsersToSave.addAll(usersToSave);
                effUsersToSave.removeAll(savedUsers);
                if(!effUsersToSave.isEmpty()){
                    twitterUserDAO.save(effUsersToSave);
                }
            }

            TwitterAccountEntity tae = twitterAccountDAO.findById(accountId);
            if(tae!=null){
                long prevTotalCount = 0;
                TwitterAccountCounterEntity counterEntity =
        twitterAccountCounterDAO.findById(accountId);
                if(counterEntity!=null){
                    prevTotalCount = counterEntity.getTweets();
                }
                long tweetsCount = prevTotalCount + tweets.size();
                if (firstTimeTotalTweets != null && tae.getFirstReadCompleted() != 1f) {
                    float completedRatio = Math.min(tweetsCount / (float)
firstTimeTotalTweets, 1);
                    tae.setFirstReadCompleted(completedRatio);
                    long lastTweetId = tweets.get(tweets.size()-1).getId();
                    tae.setLastTweetId(lastTweetId);
                } else {
                    long firstTweetId = tweets.get(0).getId();
                    tae.setLastTweetId(firstTweetId);
                }
                tae.setLastUpdate(Calendar.getInstance().getTime());
                twitterAccountDAO.save(tae);

                twitterAccountCounterDAO.incrementTweets(accountId, tweets.size());
            }

        }

    }
```

Por último, se muestran los métodos que calculan la siguiente cuenta a procesar *getNextSummary()* y el siguiente usuario del que utilizar su *token* para conectar con *Twitter*. Se puede observar cómo de entre las dos colas se le da prioridad a la cola prioritaria y cómo a la hora de elegir al siguiente usuario, si se llega al final de la lista, se vuelve a empezar por el principio.

```
    private TwitterAccountEntity getNextSummary() throws InterruptedException {
        TwitterAccountEntity u = priorityQueue.poll(100, TimeUnit.MILLISECONDS);
        if (u == null) {
           u = normalQueue.poll(100, TimeUnit.MILLISECONDS);
        }

        return u;

    }


    private UserEntity getNextUser() {
        UserEntity currentUser;
        if (usersIt.hasNext()) {
           currentUser = usersIt.next();
        } else if (!nextUsers.isEmpty()) {
           usersIt = nextUsers.iterator();
           nextUsers = new ArrayList<>();
           currentUser = usersIt.next();
        } else {
           currentUser = null;
        }

        return currentUser;
    }
```

3.2.4.3 Capa de presentación

Esta capa utiliza la capa de lógica de negocio y se encarga de proporcionar al usuario la interfaz de la aplicación, así como del intercambio de información con éste.

Esta capa está compuesta por varias partes. Por un lado se encuentran los componentes de página de *Liwenx*, *PageComponent*, que son los procesan los formularios y utilizan directamente la capa de negocio tanto para guardar información así como para proporcionar las partes dinámicas de las páginas; y por otro lado se encuentra todo el código a ejecutar en la parte de cliente y que construyen la interfaz propiamente dicha, esto es *HTML/CSS/javascript*. Dentro de las librerías utilizadas en la parte cliente podemos destacar *jQuery* [42] y *Bootstrap* [43]. Como ya vimos en el apartado 3.2.2.Liwenx, este código es generado a través de plantillas *Freemarker*.

Todos los componentes de página de la aplicación están definidos en el fichero *liwenx-components.xml* en el directorio *src/main/webapp/WEB-INF*.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-4.3.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-4.3.xsd">

    <context:annotation-config />
    <!-- Post routers -->
    <bean id="homePostRouter" class="me.alexguerra.pfc.liwenx.router.HomePostRouter">
        <property name="baseServer" value="${bases.server}"/>
        <property name="basePages" value="${bases.pages}"/>
    </bean>

    <bean id="innerPagePostRouter"
class="me.alexguerra.pfc.liwenx.router.InnerPagePostRouter"/>
```

```
    <bean id="signUpPostRouter" class="me.alexguerra.pfc.liwenx.router.SignUpPostRouter"/>

    <!-- Page Components -->
    <bean id="discoverTwitterAuthUrl"
class="me.alexguerra.pfc.liwenx.component.user.DiscoverTwitterAuthUrl">
        <property name="returnAddress" value="${bases.server}${bases.pages}/twitter-auth-
callback"/>
    </bean>

    <bean id="twitterAuthCallback"
class="me.alexguerra.pfc.liwenx.component.user.TwitterAuthCallback"/>
    <bean id="signUp" class="me.alexguerra.pfc.liwenx.component.user.SignUp"/>

    <bean id="isEmailRegistered"
class="me.alexguerra.pfc.liwenx.component.user.IsEmailRegistered"/>

    <bean id="isLoggedUser" class="me.alexguerra.pfc.liwenx.component.user.isLoggedUser"/>

    <bean id="loggedUserInfo"
class="me.alexguerra.pfc.liwenx.component.user.LoggedUserInfo"/>

    <bean id="readAccounts"
class="me.alexguerra.pfc.liwenx.component.account.ReadAccounts"/>

    <bean id="saveAccount"
class="me.alexguerra.pfc.liwenx.component.account.SaveAccount"/>

    <bean id="readTweets" class="me.alexguerra.pfc.liwenx.component.tweet.ReadTweets"/>

    <bean id="readAccount" class="me.alexguerra.pfc.liwenx.component.tweet.ReadAccount"/>

    <bean id="readAllMonitorizedAccountLifetimeStats"
class="me.alexguerra.pfc.liwenx.component.stats.ReadAllMonitorizedAccountLifetimeStats"/>

    <bean id="readMonitorizedAccountStats"
class="me.alexguerra.pfc.liwenx.component.stats.ReadMonitorizedAccountStats"/>

</beans>
```

Estos suelen ser bastante simples ya que únicamente llaman a los distintos servicios de la capa de negocio y generan *XML* a través de la libería *XOM* [44]. Sin embargo, es interesante ver cómo se resuelve la autenticación a través de *Twitter* mediante *OAuth* [45].

Figura 3-5. Página inicial de la aplicación

Se parte de la pantalla inicial de la aplicación mostrada en la Figura 3-5. Cuando el usuario pulsa en el botón acceder, mediante una petición *AJAX* se llama al componente de página *discoverTwitterAuthURL*, que se encarga de llamar a la *API* de *Twitter* para obtener un *request token*, que representa una solicitud para obtener el *token* de acceso a *Twitter*; y calcular una *URL* de *Twitter* a la que redirigir al usuario para que se identifique en dicha plataforma. En esta petición también se le indica a *Twitter* la página de nuestra aplicación que ha de llamar cuando el usuario haya terminado de identificarse en *Twitter*, es la *oauth_callback* (ver Figura 3-6 y Figura 3-7).

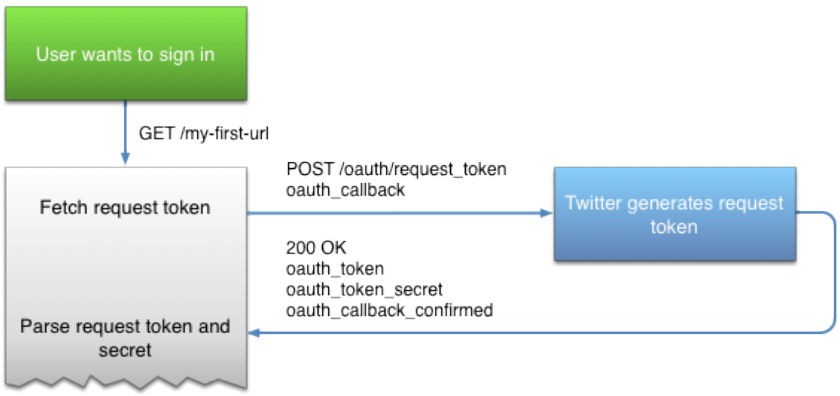

Figura 3-6. Twitter OAuth. Paso 1

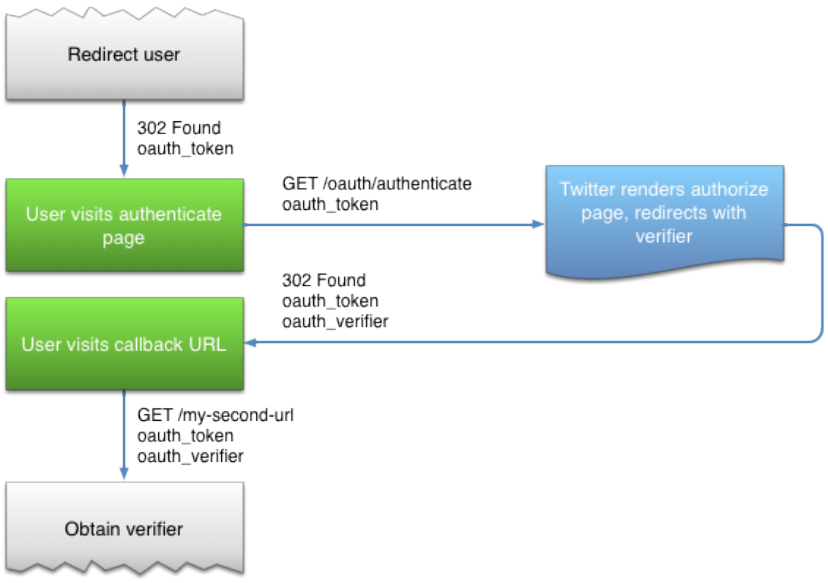

Figura 3-7. Twitter OAuth. Paso 2

Una vez el usuario ha vuelto a la aplicación a la página de *oauth_callback*, el componente *twitterAuthCallback* es el encargado de, a partir de los datos proporcionados por Twitter, *request token* y *verifier*, hacer una petición a *Twitter* para obtener el *token* de acceso y almacenarlo en sesión. Posteriormente, la interfaz redirige al usuario a la *URL* de entrada del sistema de identificación de la aplicación basado en *Spring Security* (ver Figura 3-8). Aunque lo veremos en

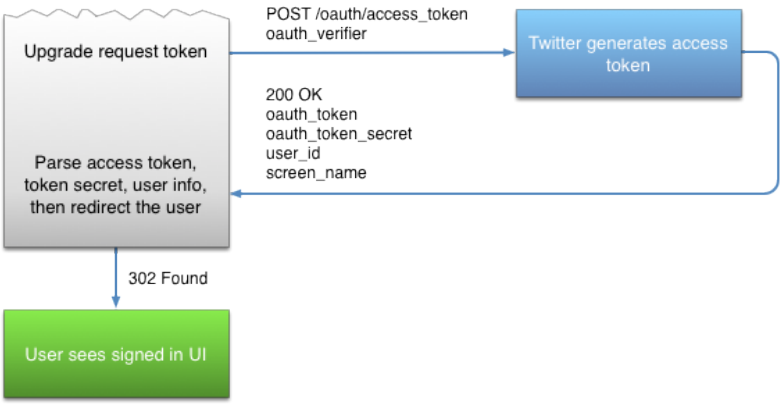

Figura 3-8. Twitter OAuth. Paso 3

profundidad en el siguiente apartado, básicamente lo que se hace es ver si existe algún usuario registrado en la aplicación con esa cuenta de *Twitter*. Si es así, se procede al *login* y se redirige a su *home* de usuario. En caso contrario se redirige a la página de registro, donde tendrá que proporcional información adicional, como su email, para crear su cuenta de usuario en la aplicación.

Por último, es interesante ver algunas páginas de la aplicación. En la Figura 3-9 podemos ver la *home* de usuario, es la página principal del mismo. Se puede observar cómo hay un panel de control de todos los servicios monitorizados donde se muestran las estadísticas globales de disponibilidad y latencia de los mismos.

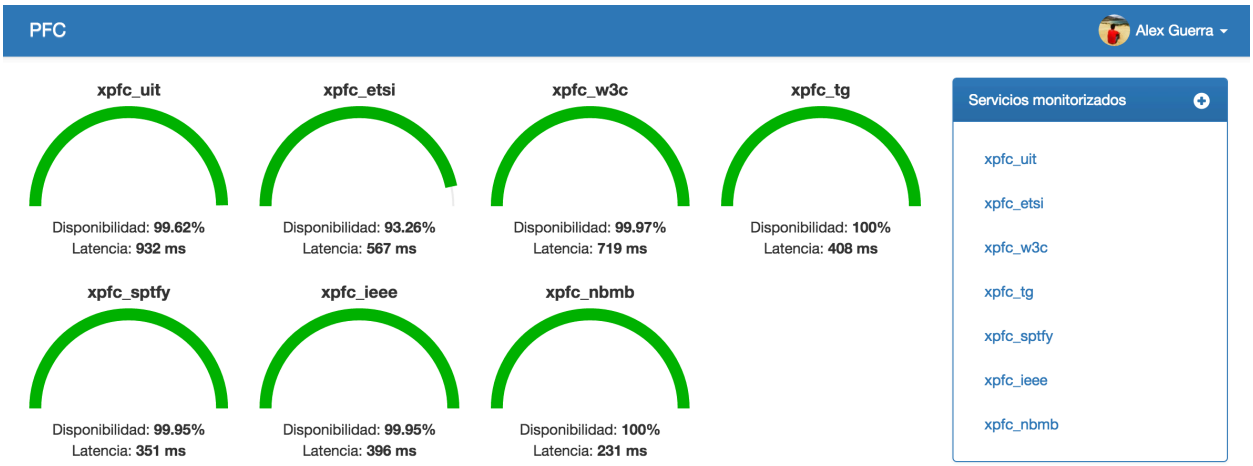

Figura 3-9. Página principal de usuario

En el menú derecho se muestra un listado con todos los servicios monitorizados y existe un botón para añadir nuevos servicios. Al pulsar en él se muestra la pantalla de la Figura 3-10.

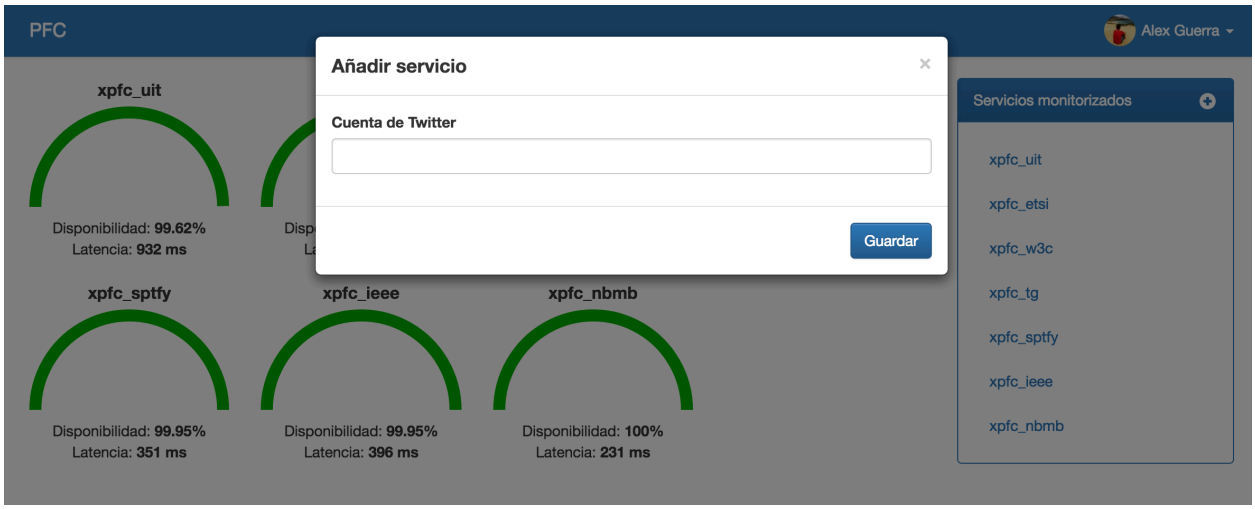

Figura 3-10. Pantalla para añadir servicio a monitorizar

Al pulsar sobre los distintos servicios accedemos a la página de detalle de los mismos, que se puede ver en la Figura 3-11. Ésta muestra una gráfica de latencia y disponibilidad y los tuits recopilados de la cuenta de *Twitter* del servicio, que se cargan por *AJAX* haciendo *scroll* infinito.

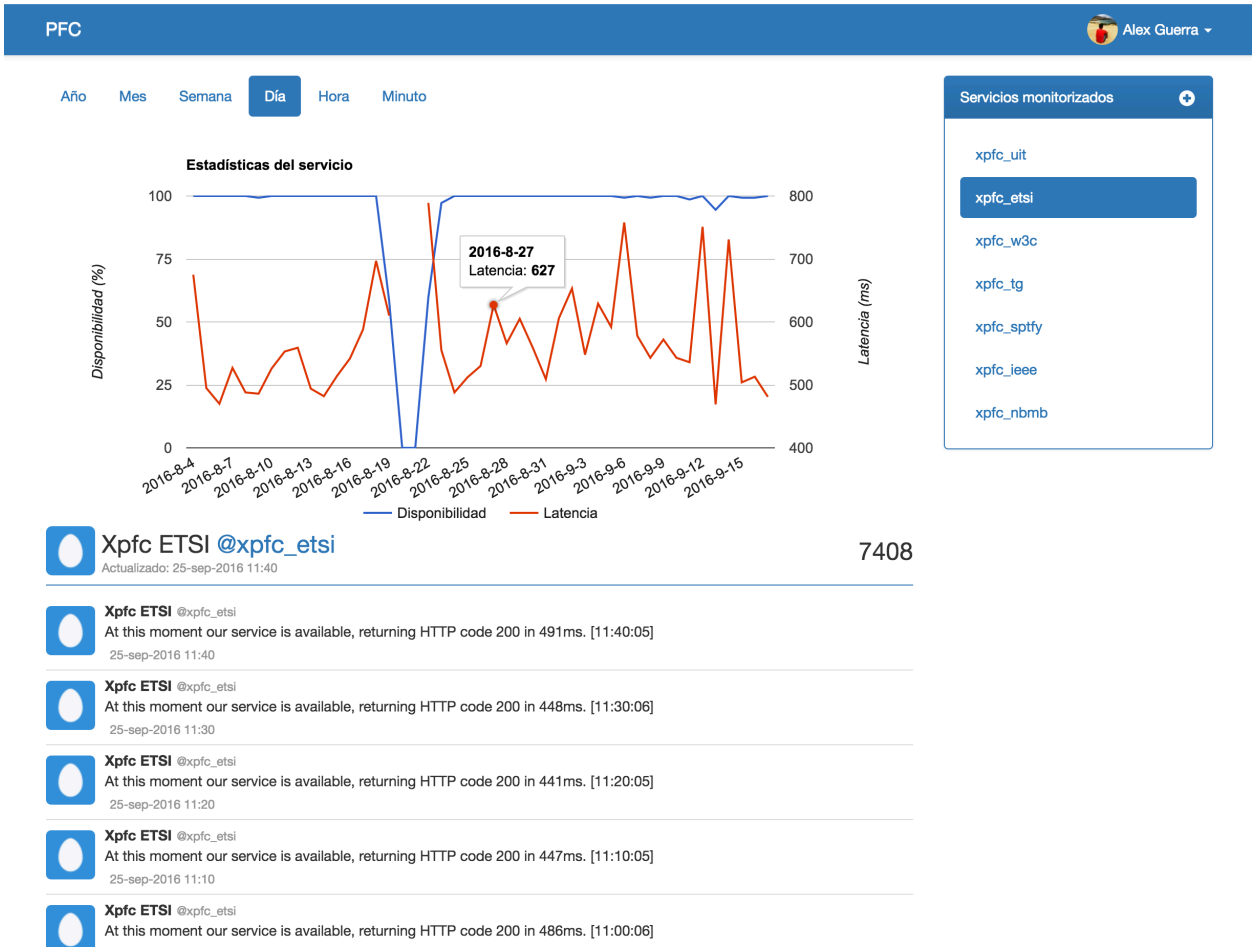

Figura 3-11. Página de detalle de servicio monitorizado

En la gráfica podemos ajustar la granularidad temporal por año, mes, día y minuto. Al pulsar sobre los puntos de la gráfica nos adentramos en ese intervalo de tiempo haciendo más fina la granularidad. Por ejemplo, si nos entramos en granularidad por días, al pulsar en un día concreto accederemos a una nueva gráfica donde se muestran los resultados de ese día concreto por el siguiente nivel de granularidad, en este caso por horas. Si volvemos a pulsar en un punto, veríamos las estadísticas de esa hora por minutos.

3.2.5 Spring Security

Se trata de un módulo de *Spring* que provee soluciones de autenticación y autorización para aplicaciones *Java*. Proporciona implementaciones para los sistemas de autenticación más comunes como *HTTP Basic, HTTP Digest, HTTP X.509, LDAP,* autenticación basada en formulario, *OpenID*, ... También permite personalizar y construir una solución propia [46].

En este proyecto se ha implementado la autenticación a través de *Twitter* a partir de la solución de *Spring Security* basada en formulario. Todos los *beans* de este módulo están definidos en el fichero *liwenx-security.xml* del directorio *src/main/webapp/WEB-INF*.

Conviene destacar el *bean* que funciona como punto de acceso a *Spring Security*. Se trata de un filtro (*Filter)* de *javax.servlet* [47]. Si vemos la declaración del *bean* a continuación, se puede observar cómo éste actúa sobre la *URL /liwenx/tw-login.do*, el punto de acceso al login una vez se tiene el *token* de acceso de *Twitter*. Además, se pueden observar la dependencia del *bean twitterTargetUrlResolver*, que será encargado de redirigir a la página de usuario en caso de login correcto o la página de registro en caso de que el usuario aún no tenga cuenta en la aplicación.

```
<!-- Authentication processing filters -->
<!-- Twitter filter -->
<beans:bean id="twitterAuthenticationProcessingFilter"
   class="org.springframework.security.twitter.TwitterAuthenticationFilter">
   <beans:property name="authenticationManager" ref="authenticationManager" />
   <beans:property name="twitterHelper" ref="twitterHelper"/>
   <beans:property name="authenticationFailureHandler" ref="authenticationFailureHandler"
/>
   <beans:property name="authenticationSuccessHandler" ref="twitterTargetUrlResolver"/>
   <beans:property name="filterProcessesUrl" value="/liwenx/tw-login.do" />
   <beans:property name="authenticationDetailsSource" ref="authenticationDetailsSource"/>
</beans:bean>

<beans:bean id="twitterTargetUrlResolver"
   class="org.springframework.security.twitter.TwitterTargetUrlResolver">
   <beans:property name="defaultUrl" value="${bases.pages}/?logged=true"/>
   <beans:property name="registrationFormUrl" value="${bases.pages}/sign-up"/>
   <beans:property name="redirectStrategy" ref="redirectStrategy"/>
</beans:bean>
```

El filtro de autenticación utiliza el *bean authenticationManager*, distinto al de nuestra aplicación. Es un *bean* propio de *Spring Security* que contiene una serie de *authentication providers*. En este caso únicamente tenemos uno para *Twitter* y es el que se encarga de ver si existe algún usuario en nuestra aplicación registrado con la cuenta de *Twitter* que se ha identificado, en cuyo caso

completa el proceso de autenticación devolviendo un *token* de autenticación de tipo *TwitterAuthenticationToken*.

```
<authentication-manager alias="authenticationManager">
   <authentication-provider ref="twitterAuthenticationProvider"/>
</authentication-manager>

<!-- Authentication provider -->
<beans:bean id="twitterAuthenticationProvider"
   class="org.springframework.security.twitter.TwitterAuthenticationProvider">
</beans:bean>
```

TwitterAuthenticationProvider

```
public class TwitterAuthenticationProvider implements AuthenticationProvider {

   private UserManager userManager;
   private UnifiedLocaleMessageSource messageSource;

   public Authentication authenticate(Authentication authentication) throws
AuthenticationException {

       TwitterAuthenticationToken twitterAuthentication = (TwitterAuthenticationToken)
authentication;

       if (twitterAuthentication.getUid() == null) {
          throw new
BadCredentialsException(messageSource.getMessage("twitterAuthentication.nonLoggedUser"));
       }

       // Successful authentication
       // I look for user in DB.
       final long uid = twitterAuthentication.getUid();
       User user = userManager.read(uid);
       final Collection<? extends GrantedAuthority> authorities;
       if (user != null) {
          if (!user.isEnabled()) {
              throw new
DisabledException(messageSource.getMessage("twitterAuthentication.disabledUser",
user.getId()));
          } else if (!user.isAccountNonExpired()) {
              throw new
AccountExpiredException(messageSource.getMessage("twitterAuthentication.accountExpiredUse
r",user.getId()));
          } else if (!user.isAccountNonLocked()) {
              throw new
LockedException(messageSource.getMessage("twitterAuthentication.lockedUser",
user.getId()));
          }
          authorities = user.getAuthorities();
       } else {
          authorities = null;
       }

       AccessToken accessToken = twitterAuthentication.getAccessToken();
       TwitterAuthenticationToken succeedToken = new TwitterAuthenticationToken(user,
uid, accessToken, authorities);
       succeedToken.setDetails(authentication.getDetails());

       return succeedToken;
   }
```

Por último, comentar que entre otros *beans* también se define la *URL* para hacer *logout* y la página de destino tras realizar esta operación. Esto se especifica con la etiqueta *logout* dentro de la etiqueta *http*. El *bean customLogoutSuccessHandler* se encarga de la redirección posterior.

```
<beans:bean id="customLogoutSuccessHandler"
class="org.springframework.security.web.authentication.logout.SimpleUrlLogoutSuccessHandl
er">
   <beans:property name="defaultTargetUrl" value="${bases.pages}/?logout=true"/>
   <beans:property name="redirectStrategy" ref="redirectStrategy"/>
</beans:bean>

<http auto-config="false" entry-point-ref="authenticationProcessingFilterEntryPoint">
   <logout logout-url="/liwenx/logout.do"
           success-handler-ref="customLogoutSuccessHandler"/>
   <custom-filter after="FORM_LOGIN_FILTER" ref="twitterAuthenticationProcessingFilter"/>
   <csrf disabled="true"/>
</http>
```

3.2.6 Despliegue

Como ya se ha comentado anteriormente, este módulo genera un fichero *.war*. Este fichero será desplegado en el servidor de aplicaciones *Tomcat* configurado a tal efecto. Basta copiar el *.war* en el directorio *webapps* de *Tomcat* para que éste despliegue la aplicación.

Por último, comentar que se ha configurado un servidor *Apache HTTP*. El servidor HTTP corre en el puerto 80 mientras que *Tomcat* lo hace en el puerto 8080. En el fichero de configuración se puede observar cómo los estáticos se sirven directamente desde *Apache HTTP* mientras que las peticiones dinámicas son pasadas a *Tomcat* a través del comando *ProxyPass*.

```
<VirtualHost *:80>
   ServerAdmin webmaster@localhost
   ServerName pfc.alexguerra.me

   LogLevel warn
   ErrorLog /var/log/apache2/moodyo_error.log
   CustomLog /var/log/apache2/pfc_access.log combined


   Alias /js /opt/tomcat/webapps/pfc/js
   <Directory /opt/tomcat/webapps/pfc/js>
        Options FollowSymLinks
        AllowOverride None
        Require all granted
   </Directory>

   Alias /css /opt/tomcat/webapps/pfc/css
   <Directory /opt/tomcat/webapps/pfc/css>
        Options FollowSymLinks
        AllowOverride None
        Require all granted
   </Directory>

   Alias /images /opt/tomcat/webapps/pfc/images
   <Directory /opt/tomcat/webapps/pfc/images>
        Options FollowSymLinks
        AllowOverride None
        Require all granted
   </Directory>

   ProxyPass /js !
   ProxyPass /css !
   ProxyPass /images !
   ProxyPass /pfc/ http://localhost:8080/pfc/
   ProxyPassReverse /pfc/ http://localhost:8080/pfc/

</VirtualHost>
```

3.3 pfc-analysis

Este módulo *Maven* se encarga de procesar los tuits de cada una de las cuentas monitorizadas, calcular sus estadísticas de servicio (disponibilidad y latencia), y almacenarlas en la base de datos. Implementa la parte de análisis del requisito funcional 8.

El *POM* de este módulo es muy simple. Podemos destacar dos partes: la definición del módulo y las dependencias del mismo, que son básicamente la librerías de *Spark* proporcionadas por *Datastax Enterprise*.

```
<parent>
   <groupId>me.alexguerra</groupId>
   <artifactId>pfc</artifactId>
   <version>0.0.1-SNAPSHOT</version>
</parent>
<artifactId>pfc-analysis</artifactId>
```

```
<dependencies>
   <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>3.8.1</version>
        <scope>test</scope>
   </dependency>
   <dependency>
        <groupId>org.apache.spark</groupId>
        <artifactId>spark-core_2.10</artifactId>
        <version>${spark.version}</version>
   </dependency>
   <dependency>
        <groupId>com.datastax.spark</groupId>
        <artifactId>spark-cassandra-connector-java_2.10</artifactId>
        <version>${spark.version}</version>
   </dependency>
   <dependency>
        <groupId>com.datastax</groupId>
        <artifactId>dse-spark</artifactId>
        <version>${dse.version}</version>
        <scope>system</scope>
        <systemPath>/opt/dse/lib/dse-spark-${dse.version}.jar</systemPath>
   </dependency>
</dependencies>
```

3.3.1 Spark

A grandes rasgos se puede decir que una aplicación *Spark* consiste en un programa conductor, *driver program*, que ejecuta una función *main* y ésta ejecuta operaciones en paralelo en un clúster de nodos *Spark*.

Spark permite crear un conjunto de datos distribuidos o *RDD (Resilient Distributed Dataset)*, que consiste en una colección de elementos particionados a través de los nodos del clúster que puede ser operados en paralelo. Es posible utilizar los *RDD* para persistir datos en memoria permitiendo la reutilización de los datos de forma eficiente en las operaciones en paralelo.

Otra funcionalidad importante que ofrece *Spark* son las variables compartidas que pueden ser usadas en paralelo. Por defecto, cuando *Spark* ejecuta una función en paralelo como un conjunto de tareas en diferentes nodos, se envía una copia a las tareas de cada variable utilizada en cada función. A veces una variable necesita ser compartida a través de las tareas o enter las tareas y el *driver program*. *Spark* proporciona dos tipos de variables: las *broadcast variables*, que pueden ser usadas para cachear un valor en memoria en todos los nodos; y los acumuladores, que son variables a las que únicamente se les puede incrementar, como sumas o contadores. Esto es importante ya que en las tareas *Spark* se utiliza la programación funcional, y por lo tanto, no debe almacenar estado alguno como variables de instancia. Para solventar esto se utilizan este tipo de variables.

Por último, comentar que *Spark* y concretamente la versión proporcionada por *Datastax Enterprise*, permite crear *RDDs* fácilmente desde ficheros de texto plano, sistema de ficheros *Hadoop*, familias de columnas *Casssandra,...* y también permite guardar los *RDDs* en dichos sistemas.

3.3.2 Implementación del analizador

Antes de pasar a ver la implementación de la tarea principal, vamos a echar un vistazo a los distintos objetos que vamos a utilizar.

SampleKey

Contendrá la clave de una muestra. Esta clave puede ser de de distinta granularidad temporal y de ahí los métodos *isXXXKey()*. También es interesante observar los métodos *equal()* y *hashCode()*, que están implementado de forma que una clave es igual a otra si tiene los mismos valores y es de la misma granularidad.

```
public class SampleKey implements Serializable {

   private static final long serialVersionUID = 25259679484735978L;

   public Long userId;
   public Integer year;
   public Integer month;
   public Integer week;
   public Integer day;
   public Integer hour;
   public Integer minute;

   public boolean isMinuteKey() {
       return userId != null && year != null && month != null && week != null && day !=
null && hour != null && minute != null;
   }

   public boolean isHourKey() {
       return userId != null && year != null && month != null && week != null && day !=
null && hour != null && minute == null;
   }

   public boolean isDayKey() {
       return userId != null && year != null && month != null && week != null && day !=
null && hour == null && minute == null;
   }
```

```
    public boolean isWeekKey() {
        return userId != null && year != null && month == null && week != null && day ==
null && hour == null && minute == null;
    }

    public boolean isMonthKey() {
        return userId != null && year != null && month != null && week == null && day ==
null && hour == null && minute == null;
    }

    public boolean isYearKey() {
        return userId != null && year != null && month == null && week == null && day ==
null && hour == null && minute == null;
    }

    public boolean isLifetimeKey() {
        return userId != null && year == null && month == null && week == null && day ==
null && hour == null && minute == null;
    }

    @Override
    public boolean equals(Object obj) {
        boolean equals;
        if (obj instanceof SampleKey) {
            SampleKey o = (SampleKey) obj;
            if (isMinuteKey() && o.isMinuteKey()) {
                equals = userId.equals(o.getUserId()) && year.equals(o.getYear()) &&
month.equals(o.getMonth()) && week.equals(o.getWeek())
&& day.equals(o.getDay()) && hour.equals(o.getHour()) && minute.equals(o.getMinute());
            } else if (isHourKey() && o.isHourKey()) {
                equals = userId.equals(o.getUserId()) && year.equals(o.getYear()) &&
month.equals(o.getMonth()) && week.equals(o.getWeek())
&& day.equals(o.getDay()) && hour.equals(o.getHour());
            } else if (isDayKey() && o.isDayKey()) {
                equals = userId.equals(o.getUserId()) && year.equals(o.getYear()) &&
month.equals(o.getMonth()) && week.equals(o.getWeek())
&& day.equals(o.getDay());
            } else if (isWeekKey() && o.isWeekKey()) {
                equals = userId.equals(o.getUserId()) && year.equals(o.getYear()) &&
week.equals(o.getWeek());
            } else if (isMonthKey() && o.isMonthKey()) {
                equals = userId.equals(o.getUserId()) && year.equals(o.getYear()) &&
month.equals(o.getMonth());
            } else if (isYearKey() && o.isYearKey()) {
                equals = userId.equals(o.getUserId()) && year.equals(o.getYear());
            } else if (isLifetimeKey() && o.isLifetimeKey()) {
                equals = userId.equals(o.getUserId());
            } else {
                equals = false;
            }
        } else {
            equals = false;
        }
        return equals;
    }

    @Override
    public int hashCode() {
        int hash;
        if (isMinuteKey()) {
            hash = Objects.hashCode(userId, year, month, week, day, hour, minute);
        } else if (isHourKey()) {
            hash = Objects.hashCode(userId, year, month, week, day, hour);
        } else if (isDayKey()) {
            hash = Objects.hashCode(userId, year, month, week, day);
        } else if (isWeekKey()) {
            hash = Objects.hashCode(userId, year, week);
        } else if (isMonthKey()) {
```

```
            hash = Objects.hashCode(userId, year, month);
        } else if (isYearKey()) {
            hash = Objects.hashCode(userId, year);
        } else if (isLifetimeKey()) {
            hash = Objects.hashCode(userId);
        } else {
            hash = 0;
        }

        return hash;
    }
```

Sample

Contendrá los valores de la muestra y su clave asociadada. Obsérvese como la propieda *delay* es

```
public class Sample implements Serializable{

   private static final long serialVersionUID = 5200790329557492161L;

   private SampleKey key = new SampleKey();

   private double available;

   private Double delay;
```

de tipo *Double* y no *double* para permitir valores nulos en el caso de no disponibilidad.

AggregatedSample

Se utilizará para hacer la media de un conjunto de muestras. Se van acumulando el total de disponibilidad y latencia y el contador de las muestras.

```
public class AggregatedSample implements Serializable {

        private static final long serialVersionUID = 7889412119119244971L;

        private double availabilitySum;

        private long availabilityCount;

        private double delaySum;

        private long delayCount;
```

Una vez conocemos los objetos que se van a utilizar, empecemos a ver la clase *PFCAnalyzer*, que es donde se define el procesado de los tuits. En primer lugar, veamos el punto de inicio, el método *main()*, y el principio del método *start()*, donde se leen los datos de la tabla *tweet* utilizando el método *select()* de *CassandraUtils.javaFunctions*. Por último, obsérvese cómo se filtran los tuits obtenidos de la base de datos para quedarnos únicamente con los que no son *retuits*, ya que estos podrían dar información de otra cuenta. La especificación del criterio del filtro se hace a través de *lambdas expressions* introducidas en *Java 8* [48].

```
public class PFCAnalyzer implements Serializable {

   private static final long serialVersionUID = -2643168843018075l0L;

   public static void main(String[] args) {
        PFCAnalyzer analyzer = new PFCAnalyzer();
        analyzer.start();
   }

   private JavaSparkContext createSparkContext() {
        SparkConf conf = new SparkConf();
        conf.setAppName("PFC Analyzer");
        return new JavaSparkContext(conf);
   }

   private void start() {
        JavaSparkContext sc = createSparkContext();
        CassandraJavaRDD<Tweet> tweets =
CassandraJavaUtil.javaFunctions(sc).cassandraTable("pfc", "tweet", mapRowTo(Tweet.class))
.select(column("user_id").as("userId"), column("id"),
column("created_date").as("createdDate"), column("is_rt").as("retweet"), column("text"));

        tweets.filter((t) -> !t.isRetweet());
```

Continuamos con el método *start()* viendo cómo se transforma el *RDD* de objetos *Tweet* a objetos *Sample* mediante la función *flatMap* de *Spark* [49]. Se puede observar cómo de cada *Tweet* salen 7 muestras, una para cada granularidad temporal que queremos calcular, es decir, una muestra tendrá una clave global, únicamente con el *id* de la cuenta de *Twitter*, otra será una clave de *año*, otra *año-mes*, *año-semana*, *año-mes-día*, *año-mes-día-hora* y *año-mes-día-hora-minuto*. Obsérvese cómo para optimizar el rendimiento se comparte el patrón de expresiones regulares *Pattern* a través de la variable compartida *Broadcast p*.

Por último, mediante la función *keyBy* se se convierte el conjunto de objetos *Sample* en conjunto de pares *SampleKey, Sample*, extrayendo la clave de cada *Sample*.

```
        // Shared pattern
        final Broadcast<Pattern> p = sc.broadcast(Pattern.compile("our service is
(?<availability>\\w+)((?:.*)returning HTTP code (?<httpcode>\\d+) in (?<delay>[0-
9.]+)ms)?"));
        JavaRDD<Sample> samples = tweets.flatMap(new FlatMapFunction<Tweet, Sample>() {

           @Override
           public Iterable<Sample> call(Tweet t) throws Exception {
                String text = t.getText();
                Matcher m = p.value().matcher(text);
                List<Sample> samples = new ArrayList<>();
                if (m.find()) {
                   String availabilityStr = m.group("availability");
                   if (availabilityStr != null) {
                        double availability;
                        Double delay = null;
                        if (availabilityStr.equals("available")) {
                           availability = 1d;
                           String delayStr = m.group("delay");
                           if (delayStr != null) {
                                delay = new Double(delayStr.replaceAll("[.]", ""));
                           }
                        } else {
```

```
                            availability = 0d;
                        }

                        DateTime date = t.getCreatedDate();
                        long userId = t.getUserId();
                        // LifeTime
                        samples.add(createSample(createSampleKey(userId), availability,
    delay));
                        // Year
                        int year = date.getYear();
                        samples.add(createSample(createSampleKey(userId, year),
    availability, delay));
                        // Month
                        int month = date.getMonthOfYear();
                        samples.add(createSample(createSampleKey(userId, year, month),
    availability, delay));
                        // Week
                        int week = date.getWeekOfWeekyear();
                        samples.add(createSample(createSampleKeyWeek(userId, year, week),
    availability, delay));
                        // Day
                        int day = date.getDayOfMonth();
                        samples.add(createSample(createSampleKey(userId, year, month, week,
    day), availability, delay));
                        // Hour
                        int hour = date.getHourOfDay();
                        samples.add(createSample(createSampleKey(userId, year, month, week,
    day, hour), availability, delay));
                        // Minute
                        int minute = date.getMinuteOfHour();
                        samples.add(createSample(createSampleKey(userId, year, month, week,
    day, hour, minute), availability, delay));
                    }

                }
                return samples;
            }
        });

        JavaPairRDD<SampleKey, Sample> samplesByKey = samples.keyBy((s) -> s.getKey());
```

Una vez tenemos los pares *SampleKey, Sample*, a través de la función *aggregateByKey* se reducen todos los objetos *Sample* con la misma clave convirtiéndolos y agrupándolos en objetos de tipo *AggregatedSample*, que como vimos anteriormente contienen la suma y el número de valores de las propiedades *availability* y *delay* para posteriormente hacer la media. Obsérvense tres parámetros de la función *aggregateByKey*:

- Un primer parámetro que representa el *valor cero* de *AggregatedSample*, en este caso un objeto de ese tipo vacío.
- Un segundo parámetro que se trata de una función que describe cómo combinar un objeto de tipo *AggregatedSample* y otro de tipo *Sample* (de la misma clave) en un tercer objeto *AggregatedSample*.
- Un tercer parámetro que se trata de una función que describe cómo combinar dos objetos A*ggregatedSample* de la misma clave resultando un único objeto de ese tipo.

El *valor cero* se usa en la primera llamada para una clave a la función especificada como segundo parámetro, es decir, cuando todavía sólo tenemos un objeto *Sample* para esa clave se

llama a esa función con el *valor cero* como primer parámetro y la primera muestra como segundo parámetro. Posteriormente se usa la tercera función para combinar dos objetos *AggregatedSample*.

Hay que notar que puesto que una clave es distinta para dos cuentas distintas, no es necesario iterar sobre las cuentas. Una vez tenemos los pares clave,valor, reducimos por clave y se procesan todos los tuits de todas las cuentas a la vez.

```
        JavaPairRDD<SampleKey, AggregatedSample> aggregatedSamples =
samplesByKey.aggregateByKey(new AggregatedSample(),
        new Function2<AggregatedSample, Sample, AggregatedSample>() {

           @Override
           public AggregatedSample call(AggregatedSample v1, Sample v2) throws Exception {

               v1.setAvailabilitySum(v1.getAvailabilitySum() + v2.getAvailable());
               v1.setAvailabilityCount(v1.getAvailabilityCount() + 1);

               Double delay = v2.getDelay();
               if (delay != null) {
                  v1.setDelaySum(v1.getDelaySum() + delay);
                  v1.setDelayCount(v1.getDelayCount() + 1);
               }

               return v1;
           }

        }, new Function2<AggregatedSample, AggregatedSample, AggregatedSample>() {

           @Override
           public AggregatedSample call(AggregatedSample v1, AggregatedSample v2) throws
Exception {
               v1.setAvailabilitySum(v1.getAvailabilitySum() + v2.getAvailabilitySum());
               v1.setAvailabilityCount(v1.getAvailabilityCount() +
v2.getAvailabilityCount());
               v1.setDelaySum(v1.getDelaySum() + v2.getDelaySum());
               v1.setDelayCount(v1.getDelayCount() + v2.getDelayCount());

               return v1;
           }
        });
```

Teniendo ya un sólo objeto *AggregatedSample* por clave, es hora de calcular los valores medios. Para ello se usa de nuevo la función *map*, que nos permitirá convertir los pares *SampleKey*, *AggregatedSample* en objetos *Sample* donde sus valores son los valores medios, no una muestra concreta.

Obsérvese la llamada al método *persist()* al final. Esto se hace para que el *driver program* recopile todos los objetos del *RDD*. Se especifica que se mantenga en memoria y, si no cabe, que se guarde a disco.

```
        JavaRDD<Sample> processedSamples = aggregatedSamples.map(new
Function<Tuple2<SampleKey, AggregatedSample>, Sample>() {

            public Sample call(scala.Tuple2<SampleKey, AggregatedSample> v1) throws
Exception {
                Sample s = new Sample();
                s.setKey(v1._1());
                AggregatedSample as = v1._2();
                s.setAvailable(as.getAvailabilitySum() / as.getAvailabilityCount());
                long delayCount = as.getDelayCount();
                if (delayCount > 0) {
                   s.setDelay(as.getDelaySum() / as.getDelayCount());
                }

                return s;
            };
        }).persist(StorageLevel.MEMORY_AND_DISK());
```

Llegados a este punto ya tenemos todos los cálculos hechos por cuenta de *Twitter* y dentro de cada una, por granularidad temporal. Lo único que nos queda es almacenarlos en sus correspondientes tablas. Para ello, basta filtrar en cada caso por tipo de clave y luego guardar en *Cassandra* a través del método *saveToCassandra()* de *CassandraUtils.javaFunctions()*.

```
        // Store results
        // Lifetime
        JavaRDD<Sample> lifetimeSamples = processedSamples.filter((f) ->
f.getKey().isLifetimeKey());
        javaFunctions(lifetimeSamples).writerBuilder("pfc", "service_info_lifetime",
mapToRow(Sample.class, Pair.of("userId", "user_id"), Pair.of("available",
"availability"), Pair.of("key.delay", "delay"))).saveToCassandra();

        // Year
        JavaRDD<Sample> yearSamples = processedSamples.filter((f) ->
f.getKey().isYearKey());
        javaFunctions(yearSamples).writerBuilder("pfc", "service_info_by_year",
mapToRow(Sample.class, Pair.of("userId", "user_id"), Pair.of("year", "year"),
Pair.of("available", "availability"), Pair.of("key.delay", "delay"))).saveToCassandra();

        // Month
        JavaRDD<Sample> monthSamples = processedSamples.filter((f) ->
f.getKey().isMonthKey());
        javaFunctions(monthSamples).writerBuilder("pfc", "service_info_by_month",
mapToRow(Sample.class, Pair.of("userId", "user_id"), Pair.of("year", "year"),
Pair.of("month", "month"), Pair.of("available", "availability"), Pair.of("key.delay",
"delay"))).saveToCassandra();

        // Week
        JavaRDD<Sample> weekSamples = processedSamples.filter((f) ->
f.getKey().isWeekKey());
        javaFunctions(weekSamples).writerBuilder("pfc", "service_info_by_week",
mapToRow(Sample.class, Pair.of("userId", "user_id"), Pair.of("year", "year"),
Pair.of("week", "week"), Pair.of("available", "availability"), Pair.of("key.delay",
"delay"))).saveToCassandra();

        // Day
        JavaRDD<Sample> daySamples = processedSamples.filter((f) ->
f.getKey().isDayKey());
        javaFunctions(daySamples).writerBuilder("pfc", "service_info_by_day",
mapToRow(Sample.class, Pair.of("userId", "user_id"), Pair.of("year", "year"),
Pair.of("month", "month"), Pair.of("day", "day"), Pair.of("available", "availability"),
Pair.of("key.delay", "delay"))).saveToCassandra();
```

```
        // Hour
        JavaRDD<Sample> hourSamples = processedSamples.filter((f) ->
f.getKey().isHourKey());
        javaFunctions(hourSamples).writerBuilder("pfc", "service_info_by_hour",
mapToRow(Sample.class, Pair.of("userId", "user_id"), Pair.of("year", "year"),
Pair.of("month", "month"), Pair.of("day", "day"), Pair.of("hour", "hour"),
Pair.of("available", "availability"), Pair.of("key.delay", "delay"))).saveToCassandra();

        // Minute
        JavaRDD<Sample> minuteSamples = processedSamples.filter((f) ->
f.getKey().isMinuteKey());
        javaFunctions(minuteSamples).writerBuilder("pfc", "service_info_by_minute",
mapToRow(Sample.class, Pair.of("userId", "user_id"), Pair.of("year", "year"),
Pair.of("month", "month"), Pair.of("day", "day"), Pair.of("hour", "hour"),
Pair.of("minute", "minute"), Pair.of("available", "availability"), Pair.of("key.delay",
"delay"))).saveToCassandra();

        sc.stop();
    }
```

3.3.3 Ejecución

Para poder lanzar nuestra aplicación *Spark*, hay tener compilado el módulo *pfc-analysis* como un único fichero *.jar*. Esto lo conseguimos mediante los comandos siguientes de *Maven*.

```
bash$ mvn clean install
```

Una vez tenemos la aplicación empaquetada, lo siguiente es enviar la tarea al nodo *Spark* maestro. Ello lo hacemos con el comando *spark-submit* de *Datastax Enterprise*. Suponiendo que éste estuviera instalado en */opt/dse*, y que nos encontramos en el directorio raíz del módulo, el comando sería el siguiente:

```
bash$ /opt/dse/bin/dse spark-submit --class me.alexguerra.pfc.analysis.PFCAnalyzer --master
spark://127.0.0.1:7077 target/pfc-analysis-0.0.1-SNAPSHOT.jar
```

Obsérvese cómo se le indica la clase principal de nuestra aplicación a través del parámetro *--class* y el nodo maestro a través del parámetro *--master*.

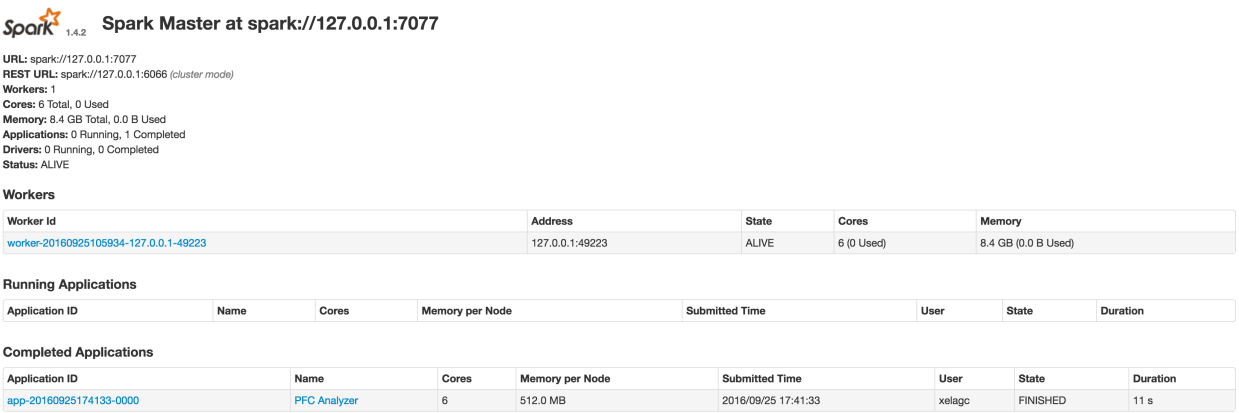

Figura 3-12. Interfaz web de Spark

Si entramos en la interfaz web del nodo maestro *http://127.0.0.1:7077* podemos ver información acerca de nuestra ejecución (ver Figura 3-12).

4 RESULTADOS

Para poder probar que el desarrollo realizado en este proyecto es correcto, desde el día 4 de agosto hasta el día 25 de septiembre de 2016 se han estado monitorizando 7 servicios webs. El módulo *pfc*-mon ha estado publicado tuits de información de servicio cada 10 minutos en 7 cuentas de *Twitter* que se han creado en este proyecto a tal efecto. Estos son:

- *http://www.itu.int/es/Pages/default.aspx* -> Web de la Unión Internacional de Telecomunicaciones, *UIT*. La cuenta de *Twitter* es *@xpfc_uit*
- *http://www.etsi.us.es* -> Web de la Escuela Técnica Superior de Ingenieros de Sevilla. La cuenta de *Twitter* es *@xpfc_etsi*
- *https://www.w3.org* -> Web de la W3C. La cuenta de *Twitter* es *@xpfc_w3c*
- *http://www.ten-golf.com/es* -> Portal Web especializado en golf con sede en Sevilla. La cuenta de *Twitter* es *@xpfc_tg*
- *https://api.spotify.com/v1/artists/51Blml2LZPmy7TTiAg47vQ* -> *API REST* de *Spotify*. Concretamente se ha monitorizado accediendo al perfil del grupo *U2*. La cuenta de *Twitter* es *@xpfc_sptfy*
- *http://www.ieee.org/index.html* -> Web del Instituto de Ingeniería Eléctrica y Electrónica, *IEEE*. La cuenta de *Twitter* es *@xpfc_ieee*
- *https://nabumbu.com/es* -> Plataforma online de comercio electrónico de juguetes con aspectos sociales de la que soy co-fundador. La cuenta de *Twitter* es *@xpfc_nbmb*

En el momento en que se escriben estas líneas cada cuenta ha publicado aproximadamente 7.448 tuits, siendo un total de 52.134 tuits entre todas las cuentas. Teniendo en cuenta que, como hemos visto en el apartado anterior, cada tuit genera 7 muestras, una para cada granularidad, el módulo *pfc-analysis* tiene que procesar 364.938 muestras. Si observamos la Figura 3-12, podemos observar que el procesamiento de todas estas muestras le ha llevado a *Spark* tan sólo 11 segundos. Estas pruebas se han realizado en un único nodo *Datastax Enterprise* sobre un *MacBook Pro Mid 2015* con un procesador *Intel Core i7 2,5GHz*, 16GB de *RAM* DDR3 1600MHz y disco duro SSD de 250GB.

Los resultados globales obtenidos en cuanto a disponibilidad y latencia de cada servicio son los que muestra el panel de control de la aplicación web en la Figura 4-1. Éstos también se recogen en la Tabla 4-1.

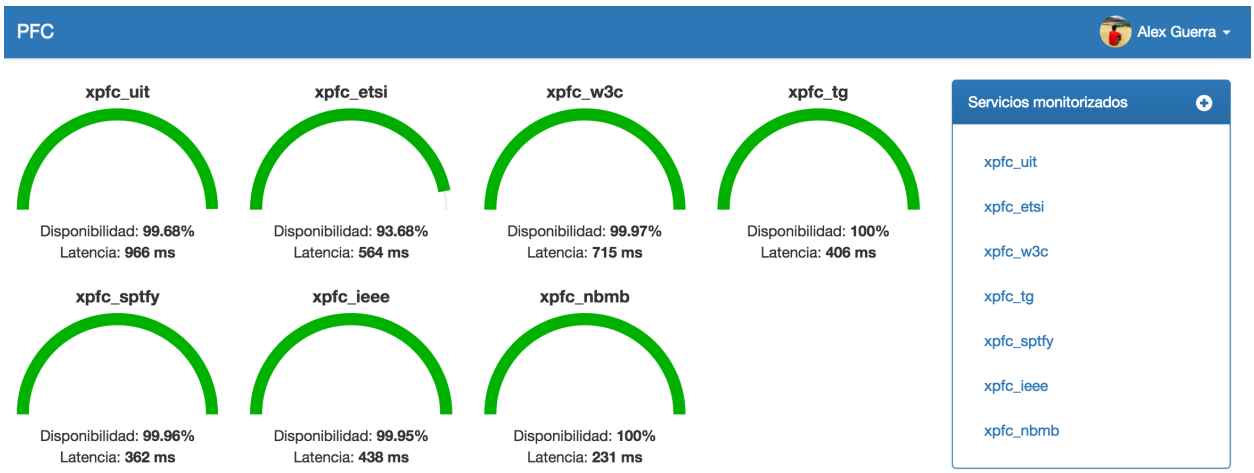

Figura 4-1. Resultados de los servicios monitorizados

Servicio	Disponibilidad	Latencia
Web UIT	99,68%	966 ms
Web ETSI	93,68%	564 ms
Web W3C	99,97%	715 ms
Web TenGolf	100%	406 ms
API Spotify	99,96%	362 ms
Web IEEE	99,95%	438 ms
Web Nabumbu	100%	231 ms

Tabla 4-1. Estadísticas de los servicios monitorizados

Observando los servicios monitorizados se puede observar cómo las webs *Nabumbu* y *TenGolf* han estado disponibles el 100% del tiempo mientras que, por el contrario, la web de la *ETSI* ha tenido la menor disponibilidad con diferencia, con un 93,68%.

En lo que se refiere a latencia, nuevamente *Nabumbu* vuelve a estar a la cabeza con 231ms de media, seguida de la *API* de *Spotify*, *TenGolf* y la web del *IEEE*, con 362ms, 406ms y 438ms respectivamente. El servicio más lento en responder ha sido la web del *UIT* con 966ms.

A continuación se muestran las estadísticas con un nivel de granularidad diaria de todos los servicios monitorizados.

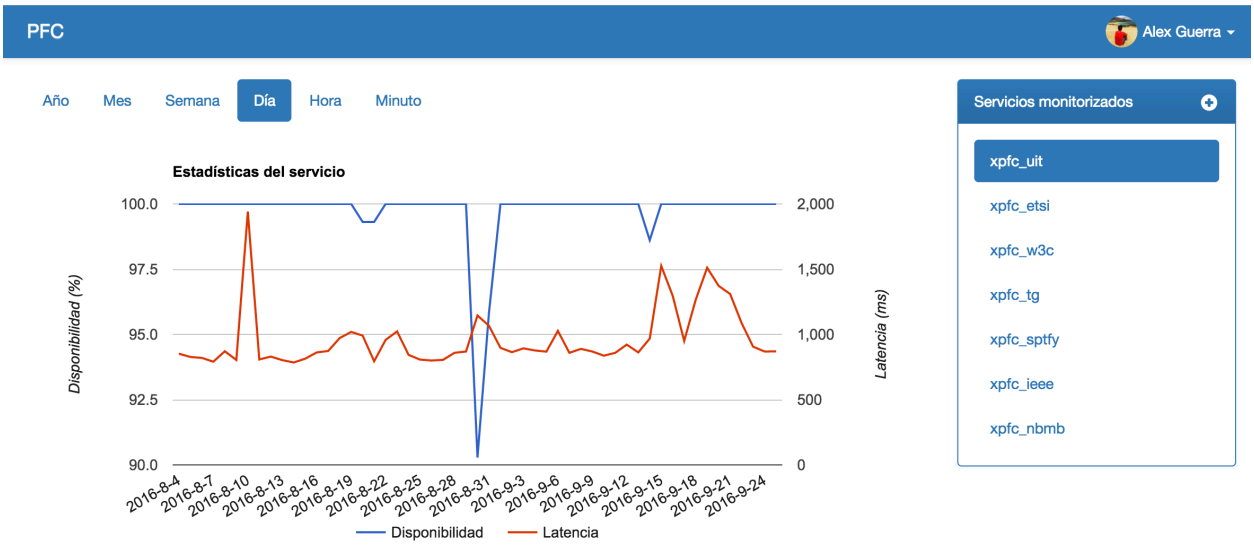

Figura 4-2. Estadísticas de la Web del UIT

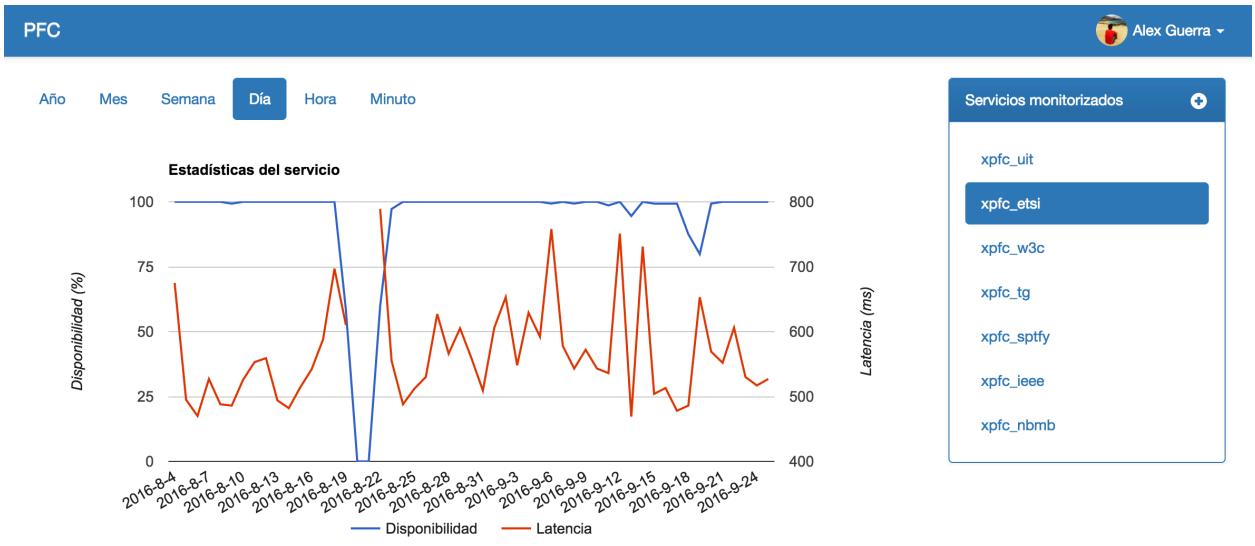

Figura 4-3. Estadísticas de la web de la ETSI

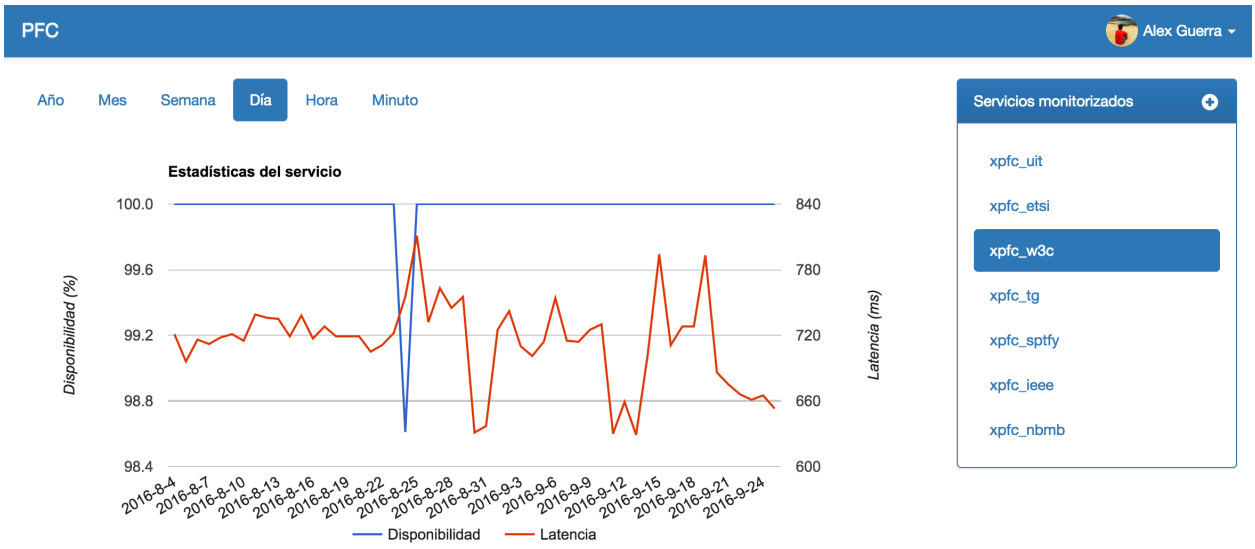

Figura 4-5. Estadísticas de la web del W3C

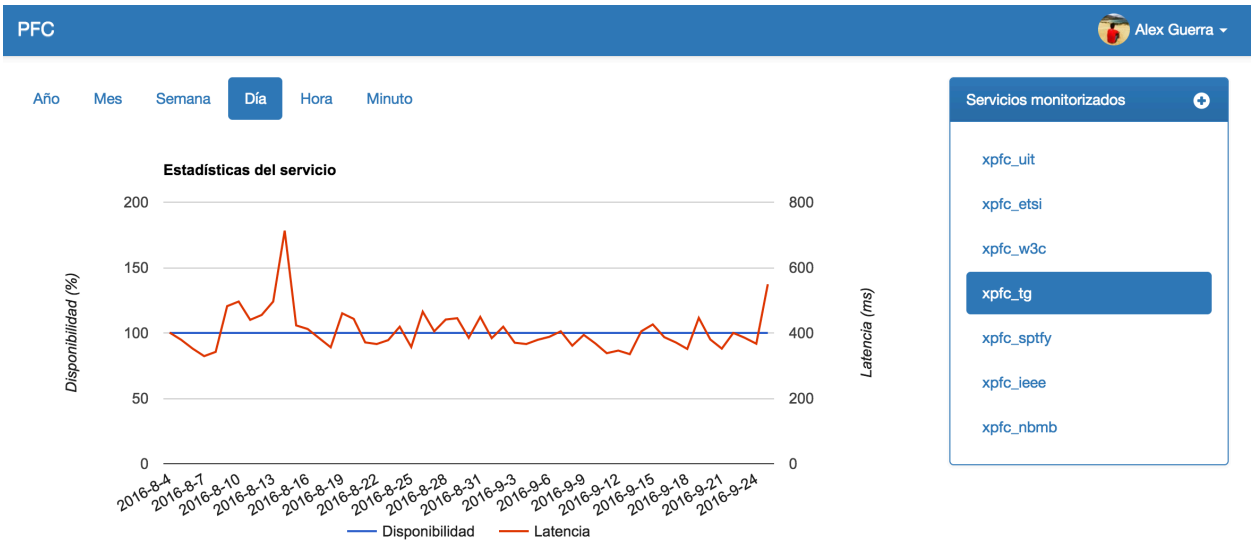

Figura 4-4. Estadísticas de la web TenGolf

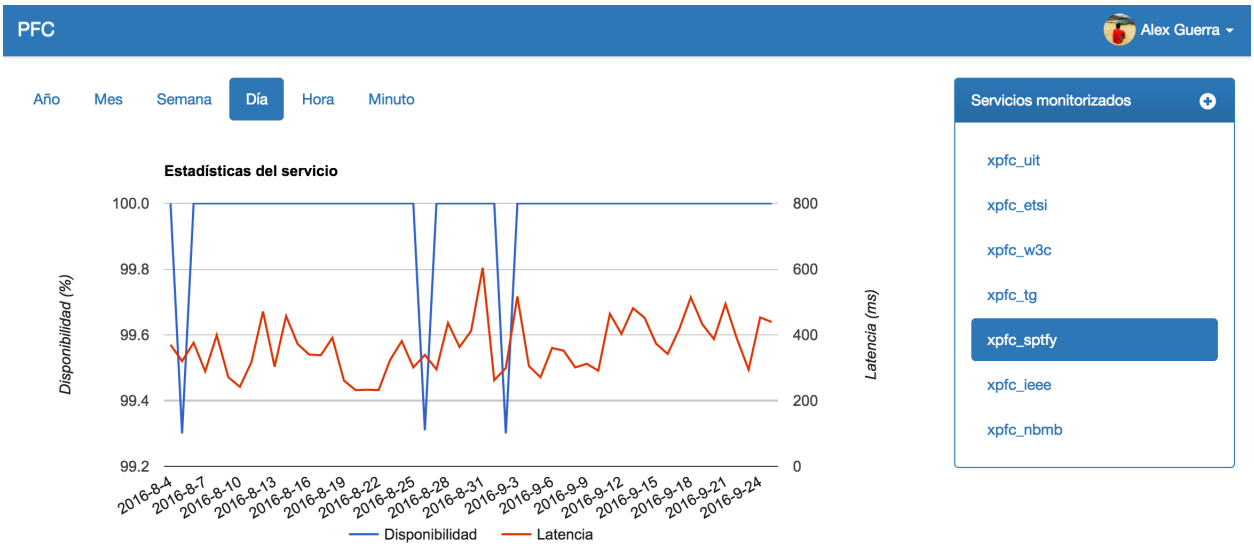

Figura 4-7. Estadísticas de la *API* de *Spotify*

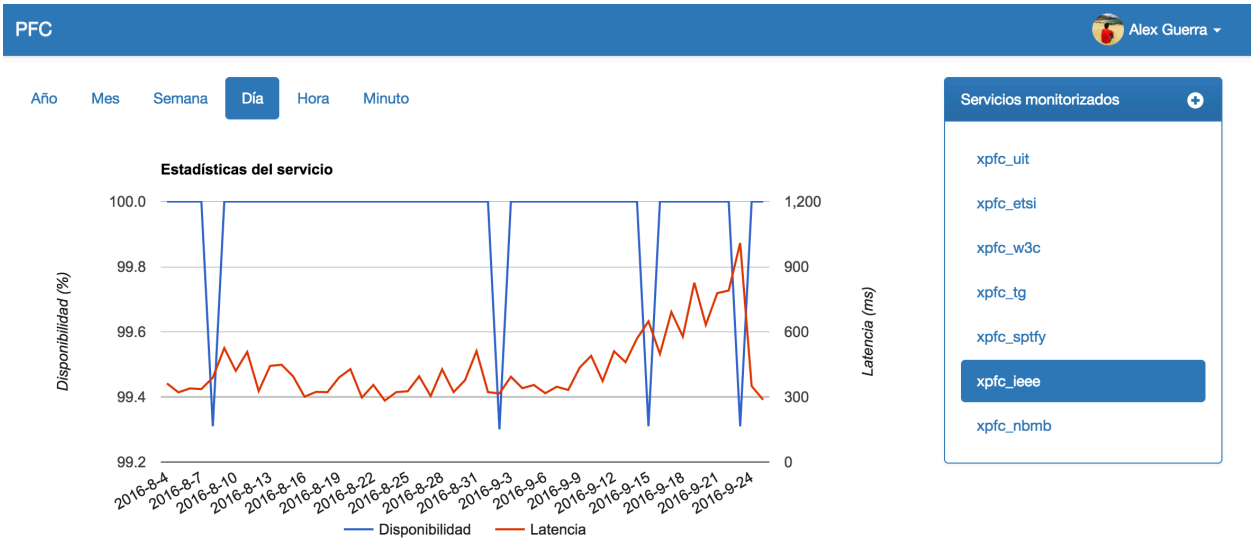

Figura 4-6. Estadísticas de la web del IEEE

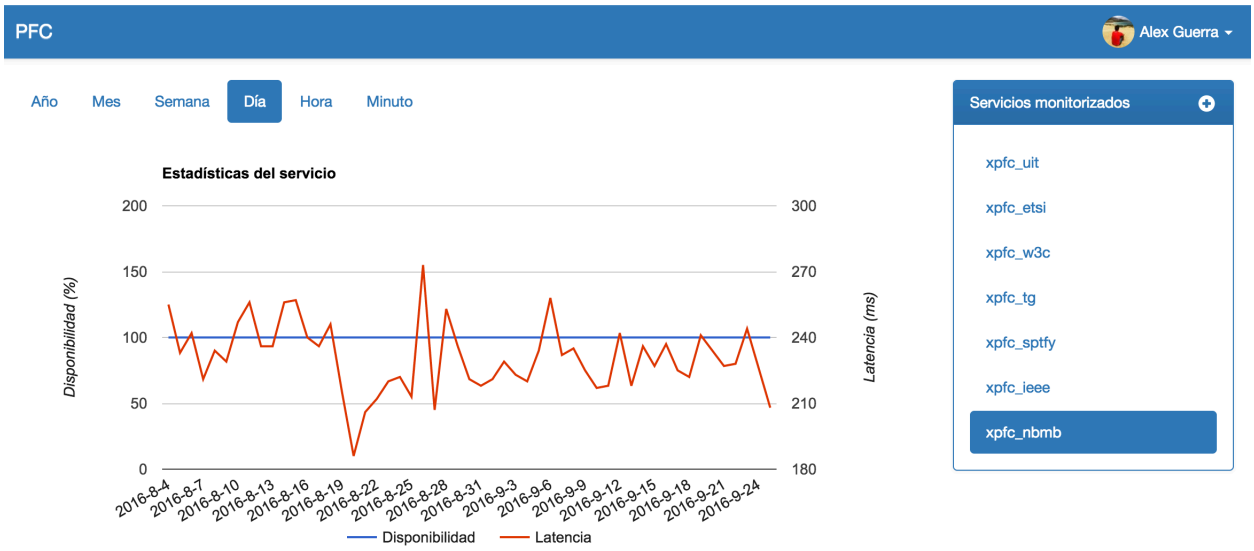

Figura 4-8. Estadísticas de la web Nabumbu

5 CONCLUSIONES

En este proyecto se ha abordado el problema de la extracción y análisis de información partiendo de una fuente de datos que cumple las reglas del *Big Data*, *Twitter*. Concretamente se ha procesado información de estado de un servicio web que éste iba publicando a través de su cuenta de *Twitter*. Existen numerosas cuentas de *Twitter* que hacen esto último. Algunos ejemplos son: *@SpotifyStatus*, *@StatusLinode*, *@wa_status*, *@herokustatus*, *@githubstatus*, *@stripestatus*, *@DOStatus* o *@StatusMinecraft*.

Sin embargo, puesto que no es objetivo de este proyecto el análisis semántico de los tuits, se ha desarrollado un módulo que permite la monitorización de un servicio *HTTP* y la publicación del estado del mismo a través de un tuit. Estos tuits tienen una estructura uniforme y la extracción de la información es bastante más sencilla.

A pesar de esto, de entre los 350.000 tuits que se generan cada minuto, debe haber mucha información que se pueda procesar de esta manera. Por ello, no hay que tomar el objetivo de este proyecto como un fin en sí mismo sino como un ejemplo de aplicación de las técnicas del *Big Data* a *Twitter*.

Una de las características de este proyecto es que se han utilizado multitud de tecnologías y librerías de terceros: *Liwenx, Spring, Apache HTTP Client, Freemarker, jQuery, Bootstrap, Twitter4j,...* Se ha desarrollado un monitor de servicios web, una aplicación web y una aplicación de análisis de los datos. Todos estos módulos podrían haberse desarrollado desde cero, sin embargo, al haber utilizado estas librerías se ha avanzado mucho más rápido con el consiguiente ahorro en tiempos y, en consecuencia, en costes.

Estas librerías aportan soluciones desarrolladas y flexibles a los problemas comunes que cualquier desarrollador se puede encontrar en cada situación. Además, al ser todas *software* libre, podemos modificar su código si lo necesitáramos. Dicho esto, se puede decir que ha sido muy positivo apostar por ellas y desde aquí animo al lector a publicar como *software* libre sus desarrollos que puedan resultar útiles a la comunidad, como ya hice en su momento con *Liwenx*.

Otro aspecto a destacar está relacionado con la base de datos *NoSQL*, *Cassandra*. Como hemos visto durante el proyecto, hay que cambiar la forma de pensar a la hora de definir el modelo de datos respecto a las bases de datos relacionales. Si bien en éstas nos fijamos únicamente en los datos y las relaciones entre los mismos, en las bases de datos *NoSQL* se parte de las consultas que se quieren hacer, ya que no es posible hacer *JOINs* entre tablas ni *agregaciones*. Esto nos lleva en muchas ocasiones a duplicar datos o a duplicar consultas, eligiendo entre duplicidad y rendimiento según el caso. Aunque la duplicación de datos en las bases de datos relacionales es

un error, en *NoSQL* es una práctica muy común y no hay que considerarlo como tal. Además, puesto que *Cassandra* es considerada una de las bases de datos más rápidas escribiendo, la actualización de familia de columnas no debería traer problemas de rendimiento.

Respecto a las técnicas de *Big Data,* se ha utilizado *Spark* para procesar los datos en modo de procesamiento por lotes. Ha sido muy instructiva la introducción a la programación funcional que implica describir la rutina de procesamiento de los datos. Resulta muy interesante escribir código siguiendo esta filosofía en lugar de la aproximación tradicional.

Más interesante aún resulta la velocidad con la que se procesan los datos y sobre todo el potencial de escalado que permite *Spark* y sus *RDDs.* Basta añadir nodos para multiplicar esta capacidad, lo cual resulta bastante sencillo.

5.1 Posibles líneas futuras de investigación

Se puede ampliar este proyecto a través de diversas líneas de investigación. La primera de ellas consiste en el estudio de un algoritmo semántico que sea capaz de analizar cualquier tipo de tuit y sacar información útil de ellos. Al añadir una cuenta de *Twitter* a la monitorización, se especificaría qué datos se quieren recopilar. De esta forma, se estaría eliminando el monitor y el entorno controlado que éste proporciona, abriendo la posibilidad de analizar cualquier información contenida en *Twitter*.

Otra línea de investigación estaría relacionada con el rendimiento y el volumen de datos a procesar. Se analizaría cómo el aumento del volumen de datos influye en el rendimiento tanto de *Cassandra* como de *Spark,* y cómo el aumento de los nodos del clúster permite incrementar éste, y si el aumento realmente es lineal, tal y como afirman los desarrolladores de estas tecnologías.

Por último, una tercera línea de investigación estaría relacionada con el rendimiento de las distintas técnicas de análisis, comparando *Hadoop* con *Spark* en distintas configuraciones de clúster y con distintos volúmenes de datos.

REFERENCIAS

[1] National Science Foundation, «The Launch of NSFNET» URL: https://www.nsf.gov/about/history/nsf0050/internet/launch.htm.

[2] W3C, «The WorldWideWeb browser» URL: https://www.w3.org/People/Berners-Lee/WorldWideWeb.html.

[3] Internet World Stats, «Internet Growth Statistics» URL: http://www.internetworldstats.com/emarketing.htm.

[4] GSMA Intelligence, «Global Data» URL: https://www.gsmaintelligence.com/.

[5] Ametic; Accenture, «Anáilsis de los hábitos y expectativas del nuevo consumidor digital en España» URL: http://www.asinte.org/upload/userfiles/NdP_Somos_digitales.pdf.

[6] CNBC, «Facebook shatters Wall Street estimates, proposes new share structure» URL: http://www.cnbc.com/2016/04/27/facebook-reports-first-quarter-earnings.html.

[7] WhatsApp, «WhatsApp Blog» URL: https://blog.whatsapp.com/616/Mil-millones.

[8] Internet Live Stats, «Google Search Statistics» URL: http://www.internetlivestats.com/google-search-statistics/.

[9] Social Bakers, «YouTube statistics directory» URL: https://www.socialbakers.com/statistics/youtube/.

[10] Twitter, «About Company» URL: https://about.twitter.com/company.

[11] E. Williams, «What is the process involved in launching a start-up at SXSW?» URL: https://www.quora.com/What-is-the-process-involved-in-launching-a-start-up-at-SXSW/answer/Evan-Williams.

[12] J. Krums, «There's a plane in the Hudson. I'm on the ferry going to pick up the people. Crazy.» URL: https://twitter.com/jkrums/status/1121915133?lang=es.

[13] El Huffington Post, «10 años de Twitter: de cómo un sofisticado sms es hoy una herramienta de cambio mundial» URL: http://www.huffingtonpost.es/2016/03/20/twitter-movilizacion-social_n_9496078.html.

[14] T. Erl, W. Khattak y P. Buhler, Big Data Fundamentals: Concepts, Drivers & Techniques, Prentice Hall, 2016.

[15] R. Fernández, «NoSQL: clasificación de las bases de datos según el teorema CAP» URL: http://www.genbetadev.com/bases-de-datos/nosql-clasificacion-de-las-bases-de-datos-segun-el-teorema-cap.

[16] «Apache Hadoop» URL: http://hadoop.apache.org/.

[17] «Apache Spark» URL: http://spark.apache.org/.

[18] «Apache Storm» URL: http://storm.apache.org/.

[19] «Apache Tez» URL: http://tez.apache.org/.

[20] «Apache Hadoop YARN» URL: http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.

[21] B. Stone, Cosas que me contó un pajarito, Gestión 2000, 2015.

[22] K. Kovacs, «Cassandra vs MongoDB vs CouchDB vs Redis vs Riak vs HBase vs Couchbase vs OrientDB vs Aerospike vs Neo4j vs Hypertable vs ElasticSearch vs Accumulo vs VoltDB vs Scalaris vs RethinkDB comparison» URL: https://kkovacs.eu/cassandra-vs-mongodb-vs-couchdb-vs-redis?.

[23] K. Kingsbury, «Jepsen: MongoDB stale reads» URL: https://aphyr.com/posts/322-call-me-maybe-mongodb-stale-reads.

[24] E. Bentley, «MongoDB mocked after posting “100GB Scaling Checklist”» URL: https://jaxenter.com/mongodb-mocked-after-posting-100gb-scaling-checklist-106822.html.

[25] «Datastax Enterprise» URL: http://www.datastax.com/products/datastax-enterprise.

[26] «Apache Solr» URL: http://lucene.apache.org/solr/.

[27] «Apache Maven» URL: https://maven.apache.org/.

[28] «Apache Tomcat» URL: http://tomcat.apache.org/.

[29] «Apache HTTP Server» URL: http://httpd.apache.org/.

[30] «Spring Framework» URL: https://spring.io/.

[31] A. Guerra, «Liwenx - Light Website Engine by Xela» URL: https://github.com/xelagc/liwenx.

[32] «Git» URL: https://git-scm.com/.

[33] W3C, «HTTP/1.1: Method Definitions» URL: https://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html.

[34] Apache Maven, «Introduction to the Build Lifecycle» URL: https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html.

[35] «JSR-000154 JavaTM Servlet 2.4 Specification» URL: https://jcp.org/aboutJava/communityprocess/final/jsr154/index.html.

[36] IEEE, «BCP 47» URL: https://tools.ietf.org/html/bcp47.

[37] Datastax, «CQL Reference» URL: http://docs.datastax.com/en/cql/3.1/cql/cql_reference/cqlReferenceTOC.html.

[38] E. C. Foster y S. V. Godbole, Database Systems A Pragmatic Approach.

[39] Datastax, «Data Model and CQL: Data Modeling Introduction» URL: https://academy.datastax.com/courses/ds201-cassandra-core-concepts/data-model-and-cql-data-modeling-introduction.

[40] Oracle, «ScheduledExecutorService (Java Platform SE 8)» URL: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ScheduledExecutorService.html.

[41] Twitter, «API Rate Limits» URL: https://dev.twitter.com/rest/public/rate-limiting.

[42] «jQuery» URL: http://jquery.com/.

[43] «Bootstrap» URL: http://getbootstrap.com/.

[44] «XOM» URL: http://www.xom.nu/.

[45] Twitter Developers, «Implementing Sign In with Twitter» URL: https://dev.twitter.com/web/sign-in/implementing.

[46] «Spring Security Reference» URL: http://docs.spring.io/spring-security/site/docs/4.1.3.RELEASE/reference/htmlsingle/.

[47] Oracle, «javax.servlet.Filter» URL: http://docs.oracle.com/javaee/6/api/javax/servlet/Filter.html.

[48] Oracle, «Lambda Expressions» URL:
https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html.

[49] «Spark Programming Guide - Transformations» URL:
http://spark.apache.org/docs/1.4.1/programming-guide.html#transformations.