UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

# DESARROLLO DE UNA HERRAMIENTA PARA APOYAR LA PARTICIPACIÓN CIUDADANA EN LAS SMART CITY

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

JUAN PABLO ARANCIBIA DONAIRE

PROFESOR GUÍA:
NELSON ANTRANIG BALOIAN TATARYAN

MIEMBROS DE LA COMISIÓN:
JOHAN FABRY
AIDAN HOGAN

SANTIAGO DE CHILE
2015

# RESUMEN

Las Ciudades Inteligentes o Smart City requieren un conjunto de herramientas tecnológicas para mejorar la calidad de vida de sus habitantes y en donde las fuentes de información más complejas son las personas. A través de las redes sociales, las personas han empezado a intercambiar ideas y a generar debate sobre los acontecimientos urbanos, creando contenido que se puede aprovechar al momento de tomar decisiones. Es en este contexto en donde nace la oportunidad que se desarrolló en esta memoria, y consiste en entregar valor a la información obtenida en un contexto geo urbano a través de múltiples formas de visualización, relacionándolas entre sí.

Se desarrolló una aplicación web que permite crear eventos, asociarlos a temáticas urbanas y a un lugar geográfico. Los eventos se muestran en una línea de tiempo, un mapa geográfico y un mapa conceptual. Además se crearon funcionalidades que participan en el proceso de decisión, tales como comentarios y votación. También se implementó un módulo de estadística sobre los eventos y temas más relevantes de la aplicación.

La aplicación se probó en un condominio, para apoyar el proceso de selección de proyectos internos que afectan a sus vecinos. Los resultados obtenidos reflejaron un aporte de la aplicación en la comprensión, discusión y selección de los proyectos planteados por la directiva.

Si bien los objetivos establecidos en la memoria se cumplieron, esto no significa el fin del proyecto, sino todo lo contrario, el inicio del desarrollo de un ecosistema tecnológico, completo e integral con el fin último de mejorar la calidad de vida de las personas.

# AGRADECIMIENTOS

*“A mis profesores, Nelson Baloian y Gustavo Zurita, quienes me guiaron con sus consejos.*

*A mis amigos, que me acompañaron en este largo proceso.*

*A mi hermano, mi madre y mi padre, que sin su cariño y apoyo no hubiese salido adelante.*

*Y de forma muy especial a mi tío Pepe, siempre estarás en mi corazón.*

*Muchas Gracias”*

TABLA DE CONTENIDO

# INDICE DE ILUSTRACIONES

## INDICE DE TABLAS

# 1 INTRODUCCIÓN

## 1.1 Contexto

El crecimiento de la población y los problemas que genera: aumento del tráfico, degradación de los barrios, segregación, privación socio-económica e inequidades en salud y educación, representan un desafío importante para la planificación y gestión de las ciudades en un futuro no muy lejano. Gestionar los recursos limitados de manera eficiente será trascendental para mantener la calidad de vida de las personas.

De igual forma como ha crecido la densidad poblacional, las manifestaciones sociales en nuestro país han aumentado, un claro ejemplo fue la marcha de los pingüinos el 2008 o la más reciente marcha de los enfermos. Esta libertad de expresión y el uso de la misma reflejan la necesidad de las persona de demostrar su descontento u opinión ante la actual sociedad y/o gobierno.

Por otro lado, las redes sociales y su masiva penetración entre los jóvenes, adultos y adultos mayores están sirviendo para mantener el contacto con otros, conocer personas, compartir información, o simplemente comentar sobre los acontecimientos del día. De esta forma las marchas y las redes sociales han sido una herramienta para expresar las opiniones personales y grupales.

En Chile, la penetración de redes sociales tales como Facebook y Twitter, han cambiado la forma de consumir información por parte de los usuarios de Internet. Ahora gastan más tiempo visitando sitios de redes sociales que hace 10 años atrás.

¿Qué sucedería si las redes sociales, además de mostrar la información que generan los usuarios también muestran indicadores o estadística de la misma? ¿Generaría valor para los usuarios mostrar dicha información en diferentes modos, tal que sirva en la toma de decisión de la persona? ¿Y si la información se puede contextualizar en diversos planos (temporal, físico, relacional, etc.) a la vez, cómo ayudaría esto a una mejor toma de decisión? Son algunas de las respuestas que se pretenden responder al final de este trabajo.

## 1.2 Motivación

Generar una herramienta que contribuya a la generación de una Smart City representa un gran desafío, ya que consiste en construir una aplicación en un contexto que en la actualidad no se tiene total conocimiento y menos el equipamiento para ser una verdadera Smart City en nuestro país.

La motivación principal es entregar una aplicación a los ciudadanos para que puedan expresar sus opiniones, comentarios, interés, etc. en un contexto urbano, por ende dinámico, progresivo y continuo.

Dicha herramienta debe ser transversal a los temas del momento y ser capaz de adaptarse a los nuevos escenarios. Es por ello que resulta interesante proponer un conjunto de opciones para generar nuevos temas dentro de la misma y la forma en la cual estos son visualizados por los usuarios. Puede ser desde un simple mapa con los puntos hasta complejas visualizaciones de realidad aumentada en dispositivos móviles. También es interesante crear ontologías/the saurus personalizadas de acuerdo a las preferencias de las personas dentro de la aplicación. Y luego ver cómo dichas redes de conocimientos se relacionan entre sí. Otra área de exploración resulta el Data Minning de los datos recopilados por la herramienta, dado que está orientada a un uso intensivo y multipropósito. Otros temas que se asocian fácilmente con la problemática son las de Big Data y Sistemas Distribuidos.

La herramienta resulta ser el principio de un conjunto de desarrollos orientados a proveer un ambiente global a las ciudades inteligentes.

## 1.3 Visualización de la información en las redes Sociales

Cada red social privilegia una forma de mostrar la información: por un lado Twitter muestra los tweets en forma de microblogging en un espacio temporal (con su línea de tiempo). Existe la posibilidad de asociar una ubicación determinada al mensaje pero no es posible visualizarlo en conjunto con otros mensajes en la proximidad física. Esto limita el plano geográfico de los mensajes. Otro aspecto que Twitter no considera es la estadística subyacente de los temas tratados. Si bien están los Trendic Topics, no entrega a los usuarios el dato concreto de frecuencia ni mucho menos detalles del resto

de los temas tratados. Facebook por otro lado, es similar a Twitter en cuanto a una asociación temporal de la información, y tiene la ventaja de permitir un amplio uso de recursos multimedia de cada usuario.

Las redes sociales que privilegian el plano físico son por ejemplo Waze y Foursquare, en donde se potencia la visualización de la información a través de un mapa como elemento principal.

De las redes sociales investigadas, ninguna privilegia una visualización orientada a la relación entre temas, a través de mapas mentales, conceptuales u otros. Algunos ejemplos de este tipo de aplicación son Google Wonder Wheels[1] o Immersion Media[2] del MIT pero que no están diseñadas para soportar usuarios y generar asociaciones.

La propuesta de la aplicación a desarrollar pretende rescatar ventajas de cada red social, tales como la asociación físico-temporal de la información, agregando la asociación temática. La característica principal será la visualización de la información desde distintos planos (geográfico, temporal o relacional) y también en conjunto. Además proveer información consolidada y estadística sobre los temas tratados, contingentes e históricos.

## 1.4 Objetivo principal

El objetivo de la memoria consiste en generar una aplicación para el usuario de Internet donde pueda intercambiar sus comentarios, preferencias e ideas en un contexto geo-urbano y personalizado, a través de una aplicación web y/o móvil, con el objetivo de generar mayor participación social.

Un caso en concreto resulta ser la toma de decisiones en una comunidad o recinto habitacional que cubren un área geográfica extensa, por ejemplo un condominio de parcelas de agrado. La aplicación debe apoyar dicho proceso selectivo, proveyendo un conjunto de herramientas necesarias para presentar un tema de discusión, generar debate y converger a una postura común.

[1] http://www.googlewonderwheel.com
[2] https://immersion.media.mit.edu

Si bien no se espera que las resultados del uso de la aplicación en esta comunidad tengan un carácter resolutivo, si es deseable que sean influyentes en el proceso final de la toma de decisión por parte de las personas que deban hacerlo.

## 1.5 Objetivos específicos

- Diseñar un framework para manejar múltiples temáticas urbanas, desde su creación, desarrollo y adaptabilidad. Definir el ciclo de vida y las herramientas útiles en cada etapa, tanto de visualización como de personalización.
- Diseñar e implementar el modulo maestro de la aplicación, que se encargará de las operaciones básicas (crear, leer, actualizar y borrar) de los eventos y su posterior sincronización eficiente de los datos según parámetros como tiempo, espacio, tópicos, etc.
- Crear una aplicación que interactúe con otras redes sociales populares como Facebook, Twitter o Foursquare.
- Desarrollar distintas formas de visualización de la información ingresada por los usuarios.
- Crear un módulo de estadística sobre los temas ingresados por los usuarios.

## 1.6 Glosario técnico

| Término | Descripción |
|---|---|
| **AngularJS** | Librería en JavaScript que sigue el patrón MVC |
| **Apache Cassandra** | Base de datos no relacional distribuida |
| **ArborJS** | Librería en JavaScript usada para dibujar grafos |
| **Asynchronous Javascript And XML (AJAX)** | Técnica de desarrollo para crear aplicaciones interactivas |
| **Bootstrap** | Framework HTML, CSS y JS para crear aplicaciones responsivas |
| **Concept Map** | Mapa conceptual |
| **Expression Lenguaje (EL)** | Lenguaje de programación utilizado para agregar expresiones en las páginas web |
| **Google Maps** | Servicio de Google que provee geolocalización y mapas geográficos del mundo |
| **Hibernate** | Herramienta de mapeo ORM en Java |
| **Hibernate Search** | Proyecto de Jboss para implementar búsquedas eficientes sobre objetos ORM |
| **HighCharts** | Librería en JavaScript para generar gráficos |
| **Inversion de Control (IoC)** | Método de programación en donde el flujo de ejecución de un programa se invierte |
| **Java Enterprise Edition (Java EE)** | Plataforma de programación en java para desarrollar software que corren en un servidor de aplicación |
| **Java Persistence API (JPA)** | Api de persistencia desarrollada para la plataforma Java EE |
| **Java Server Pages (JSP)** | Tecnología para crear páginas web dinámicas basadas en HTML |
| **Java Server Pages Standard Tag Libs (JSTL)** | Componentes Java EE que proveen utilidades para construir páginas web |
| **Java Standard Edition (Java SE)** | Colección de APIs en Java para desarrollar aplicaciones, generalmente de escritorio |
| **JavaScript Object Notation (JSON)** | Formato ligero para intercambiar datos |

| | |
|---|---|
| **Jquery** | Biblioteca de JavaScript que interactúa con los documentos HTML |
| **Mapper** | Clase que transforma objetos |
| **Microblogging** | Servicio que permite a los usuarios enviar y publicar mensajes breves |
| **Plain Old Java Object (POJO)** | Hace referencia a clases Java simples que no dependen de un framework |
| **Programación Orientada a Aspectos (AOP)** | Paradigma de programación que tiene como objetivo la modularización y correcta separación de responsabilidades en una aplicación |
| **Servidor de Objetos Acopados (COS)** | Servidor encargado de sincronizar mensaje entre clientes |
| **Software como Servicio (SaaS)** | Modelo de distribución de software donde la lógica y los datos se almacenan en una empresa TI |
| **Spring Data** | Proyecto de Spring para trabajar con la capa de persistencia |
| **Spring Framework** | Proyecto de Spring que provee inyección de dependencias, administración de transacciones, aplicaciones web, acceso de datos, mensajes, testeo entre otras |
| **Spring MVC** | Proyecto de Spring para crear aplicaciones web |
| **Spring Security** | Proyecto de Spring enfocado en autenticación y control de acceso de la aplicación |
| **Spring Social** | Proyecto de Spring que provee conexión transparente a redes sociales |
| **Timeline** | Tipo de visualización de la información a través de una línea de tiempo |
| **Value Object (VO)** | Objeto simplificado de otra entidad |

**Tabla 1: Glosario técnico**

# 2 MARCO TEÓRICO

## 2.1 Smart City

En los últimos años, el rápido aumento demográfico en el mundo ha sido un tema de interés general. Desde el año 1960 a la fecha, la población mundial ha crecido de 2.500 millones a 6.000 millones de personas. Según la teoría de Thomas Maltus (1766-1834), en el año 2600 existirán 630.000 millones de personas.

Ante la inevitable sobrepoblación del planeta, surgen iniciativas que tratan de abordar dichos desafíos: temas medio ambientales y sustentabilidad, transporte, altos costos de administración, participación ciudadana, restricciones energéticas, realzar la herencia cultural, etc. Así es como nace el concepto de Smart City.

Según IBM "*Una ciudad es un sistema de sistemas. Un trabajo dinámico en progreso, con progreso como lema. Un trípode (infraestructura, operaciones y gente) que se basa en un fuerte apoyo para y entre sus pilares para convertirse en una ciudad inteligente para todos*" [1]

De acuerdo al MIT "*… las ciudades son sistemas de sistemas, y que ha emergido la oportunidad para introducir un sistema nervioso digital, inteligencia responsable, y optimización en cada nivel de integración – desde los dispositivos individuales y dispositivos de los edificios a los sistemas de las regiones y países*" [2]

Ambas corrientes tienen como principal definición de una ciudad como un sistema de sistemas interconectados por TICs.

Algunos de los temas importantes en las Smart City son: servicios a los ciudadanos, participación ciudadana, gobierno abierto transparente y confiable, y la conciencia, eficiencia y administración de la sustentabilidad.

Un aspecto técnico a considerar en las Smart City es el gran volumen de información recolectada de todos los sistemas y sus interacciones. Según Gartner, Big Data es *"un conjunto de datos de gran volumen, gran velocidad y procedente de una gran variedad de fuentes de información que demandan formas innovadoras y efectivas de procesamiento de información"* [3]. Producto de esto, se hace necesario tratar de clasificar los datos obtenidos para su mejor análisis y procesamiento. Un modelo que

trata de clasificar los datos generados en una Smart City es el propuesto por Ajit Jaokar [4] que los divide en tres categorías principales: “duros”, “blandos” y “compuestos”. En la Ilustración 1 se presenta el modelo descrito anteriormente:

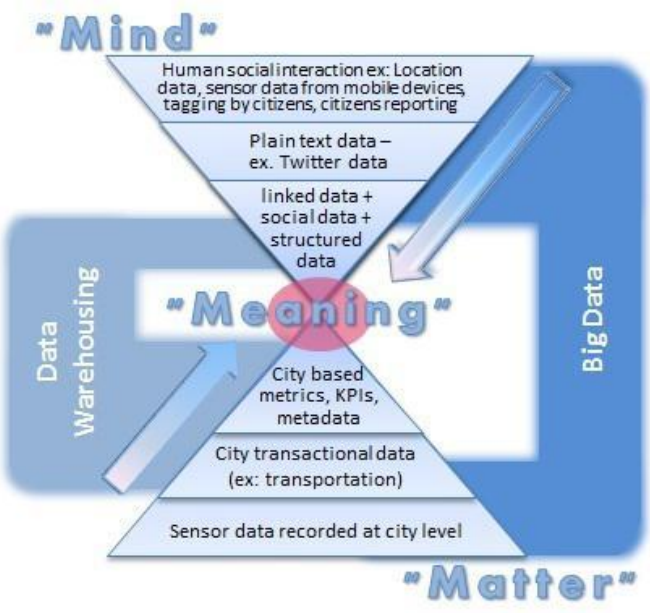



**Ilustración 1: Modelo de datos en las Smart City.**[3]

En la parte inferior esta toda la información proveniente directamente del mundo físico, pasando por sensores que la identifican, acciones que la generan y finalmente indicadores que las utilizan. En la parte superior corresponde a la información proveniente de las mentes, de la interacción humana con la sociedad, su posterior materialización (con un texto, imagen, etc.) y finalmente su asociación con otros datos. Dicha información es menos estructurada y requiere un procesamiento distinto al del triángulo inferior.

Es en la unión de ambos triángulos en donde se generan los datos compuestos de una sociedad y donde se pueden obtener información valiosa tanto para las personas como instituciones gubernamentales.

[3] http://www.opengardensblog.futuretext.com/archives/2012/08/big-data-for-smart-cities-for-hackers-data-scientits-and-citizens.html

## 2.2 GeoSmartCity y Geo-colaboración

Un apoyo considerable para los desafíos que proponen las Smart City son las TICs, y en particular los sistemas geográficos y dispositivos de localización. Estos han tenido un aumento en su uso producto de la masificación de los celular/tablets con sistema GPS y un sin número de aplicación con las cuales dichos dispositivos se pueden asociar y en distintos contextos (redes sociales, Google Maps, sistemas GIS, etc.). La asociación espacial en conjunto con la Smart City generó el área de las geoSmartCity [5]

Algunas de las preguntas que se pueden resolver con la arista espacial en las Smart City son las asociadas a la ubicación propia y del resto, asociación temática a un lugar, estadística por lugar, flujo de datos por ubicación, contenido histórico/espacial de un lugar, etc.

Según [6], la geo-colaboración "*es un modelo de tareas colaborativas desarrolladas para un grupo de personas que envuelve la contextualización, construcción e intercambio de geo-referencias basadas en una interfaz humano-computador que muestra el mapa de la zona física en el fondo, donde las tareas pueden ser desarrolladas y/o espacialmente contextualizadas por el uso de dispositivos móviles o desde computadores*". Es decir, corresponde una forma de intercambiar ideas, opiniones, comentarios o sugerencias entre personas en un contexto dentro de la ciudad con foco principal en la locación.

Si a la asociación GeoSmartCity se le agrega una nueva componente colaborativa, obtenemos un sistema multipropósito para iniciativas urbanas y participación ciudadana.

## 2.3 Teoría ecológica de Bronfenbrenner

Urie Bronfenbrenner (1917-2005) fue un psicólogo estadounidense que creó la teoría ecológica del desarrollo y el cambio de conducta del individuo. Esta teoría define el desarrollo como un cambio perdurable en el modo en que la persona percibe el ambiente que lo rodea (ambiente ecológico) y en el modo en que se relaciona con él. Bronfenbrenner denominó cuatro sistemas: microsistema, mesosistema, exosistema y macrosistema. En Ilustración 2 se muestra la relación entre ellos.

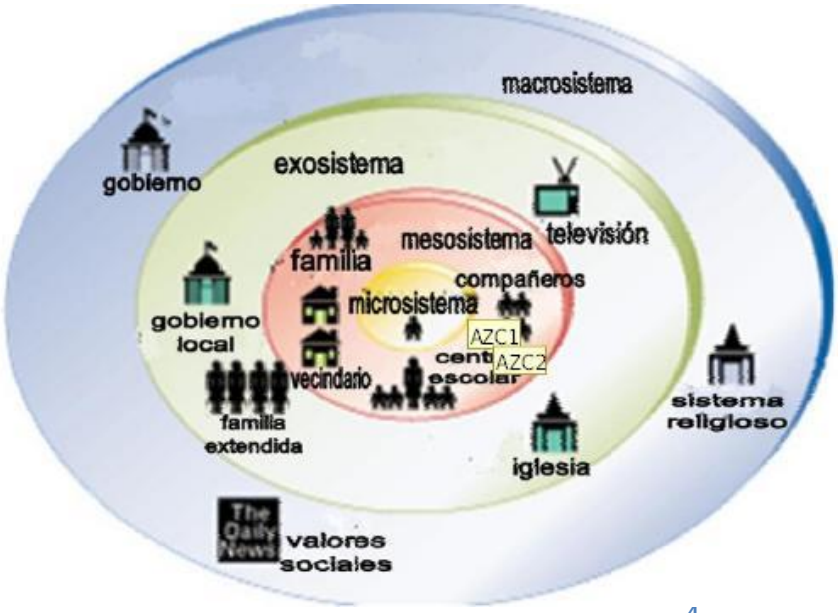



**Ilustración 2: Diagrama de la teoría ecológica**[4]

El microsistema corresponde al nivel más cercano al individuo, generalmente su familia y amigos. El mesosistema comprende la relación entre dos o más entornos en los que la persona participa activamente, por ejemplo universidad y vida familiar. El exosistema es el contexto más amplio que no consideran a la persona como un sujeto activo. Y finalmente el macrosistema lo configura la cultura y leyes en las que se desenvuelve la persona y toda la sociedad.

Según Villalba, *"Bronfenbrenner argumenta que la capacidad de formación de un sistema depende de la existencia de las interconexiones sociales entre ese sistema y otros. Es decir, cada nivel del modelo dependen unos de otros, y por lo tanto, requieren una participación conjunta de los diferentes contextos y su respectiva comunicación"* [7].

Dada la complejidad de sistemas en una Smart City, surge la necesidad de crear un conjunto de herramientas capaces de comunicar el contexto privado y cotidiano de las personas, ya sea de un modo horizontal entre personas, como también vertical entre personas e instituciones. Traspasar información desde el microsistema al exosistema. Disminuyendo así la brecha existente entre ambos sistemas.

[4] http://karindhz.blogspot.com/2011/09/taller-medio-ambiente-y-aprendizaje-el.html

## 2.4 Participación social de los jóvenes en Chile

La Sociedad Civil, según el PNUD(2004), corresponde a la forma de asociación autónoma del mercado y el Estado, que tienen como objetivo reivindicar derechos, expresar opiniones, influir en las decisiones que afectan a la comunidad y controlar a sus autoridades. En este sentido, los jóvenes son el principal recurso de las sociedades para producir cambios sociales y promover la innovación tecnológica.

La participación política de los jóvenes en Chile es baja. Según la 7ma encuesta nacional de la juventud [8] solo 1% declaró pertenecer a un partido político y el 10% demostró interés en votar en las elecciones municipales del 2012. Esto contrasta con la participación en internet, donde el 19% de los jóvenes entre 15 y 29 años participa en campañas por internet y el 17% participa en una comunidad virtual.

La influencia de las tecnológicas en la vida de los jóvenes va en aumento y son la herramienta para expresar sus opiniones. Ante la pregunta sobre qué acciones realizaría para dar a conocer su opinión en caso de que una ley sea mala o injusta, el 48% dijo que buscaría en algún grupo de internet que comparta su opinión. El 61% mencionó que las redes sociales son una mejor herramienta que el voto para dar a conocer sus demandas, el 69% estuvo de acuerdo con que sin las redes sociales las manifestaciones serían mucho menos masivas que en la actualidad y el 41% afirmó que las redes sociales le permiten incidir en forma más directa en la toma de decisiones del estado. [8]

Con respecto al uso de la tecnología, el 48% declaró que usa internet todos los días. Las actividades que realizan con más frecuencia son visitar Facebook y Twitter, seguida por chatear, enviar/recibir emails y buscar información en general.

## 2.5 Redes sociales en Chile

Una red social se puede definir como un conjunto de elementos conectados entre sí de acuerdo a alguna característica en común (parentesco, intereses, amistad, etc.). De acuerdo a un estudio realizado por Brandtzaeg y Heim, las principales razones para el uso de las redes sociales son: buscar nuevas relaciones y gente nueva, mantener el contacto con amigos y conocidos y compartir experiencias y comentarlas [9]

En nuestro país, y a nivel mundial, dos de las principales redes sociales son Facebook y Twitter. Según un estudio del 2011, la compañía de Mark Zuckerberg alcanzó los 6.7 millones de usuarios, mientras que la red social del pájaro celeste obtuvo 1.2 millones de usuarios. Considerar que el crecimiento estimado es entre un 10% y un 16% para este periodo [10]

Entonces cabe preguntar ¿Por qué dichas redes sociales tienen tanta popularidad en Chile y el resto del mundo? En el caso de Twitter, un estudio reveló que sus principales usos son: hablar sobre eventos contingentes o sobre lo que la gente está haciendo, responder a otros usuarios con mensajes cortos, compartir información útil y reportar noticias [11]. Por otro lado, otro estudio concluyó que Facebook es usado para: contactar amigos y conocidos, conocer gente virtualmente y compartir fotos o videos [12]. Dentro de las razones principales para sus usos son la necesidad de pertenencia y auto presentación.

Cabe destacar el aumento del uso de Foursquare y Waze en el contexto urbano físico. La primera está enfocada en generar una red de lugares de interés donde los usuarios pueden registrarse y comentarlos. Según un estudio realizado por IAB [13] Chile posee cerca de 6000 usuarios de Foursquare. La segunda red social está orientada a las personas que utilizan la red vial para trasladarse, informa el estado de las calles y permite generar rutas para ir de un lugar a otro. En el año 2013, los usuarios registrados en esta aplicación en Chile eran aproximadamente 1.9 millones.

## 2.6 Teorías del aprendizaje y múltiples vistas

Existen diversas formas en la que las personas adquieren conocimiento, ya sea en el hogar, el colegio, universidad, entre otros. Sin embargo, el aprendizaje es una actividad diaria y cotidiana, por ende de contexto variable. Existen algunas teorías que pueden ayudar a la mejor comprensión del modo en que las personas aprenden.

Algunas de estas mencionadas en [14] por Zurita, et al., son: aprendizaje fluido, aprendizaje situado, aprendizaje rizomático y aprendizaje continuo. El aprendizaje fluido se refiere a las actividades marcadas por continuas experiencias a través de distintos contextos, y apoyado por tecnología móvil y ubicua. El aprendizaje situado corresponde a un modelo de aprendizaje que toma lugar en el mismo contexto que es aplicado. El aprendizaje rizomático se refiere a la interconexión entre las ideas y la exploración desde distintos puntos de vista. Finalmente el aprendizaje continuo corresponde al proceso continuo, voluntario y auto motivado de adquirir conocimiento en el ámbito personal y profesional y no tiene asociado espacio ni tiempo.

De las cuatro teorías mencionadas anteriormente, los tópicos que se desprenden de ellos según [14] son: Contexto, Continuidad, Movilidad y No Delimitado. Donde lo no delimitado corresponde al aprendizaje continuo y el contexto al rizomático y situado. En la siguiente figura se muestra cada una de sus relaciones.

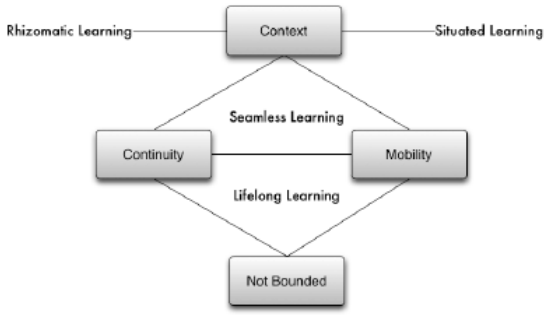



**Ilustración 3: Conceptos de las teorías del aprendizaje**

También en [14] se propone un conjunto de visualizaciones que integran los conceptos determinados anteriormente. En resumen son cuatro vistas:

Vista de calendario: para especificar el contexto temporal en que el usuario registra un elemento y como este se relaciona con otros dentro del mismo día, mes o año.

Vista de mapa: para especificar el contexto geográfico de diferentes elementos, representado por un marcador sobre el mapa.

Línea de tiempo: para agrupar las notas en un contexto temporal en la que fueron agregadas los elementos y también provee información contextual de los mismos.

Mapa Conceptual: para mostrar una extensa vista de cómo la información se ha ido relacionando, a través de temas principales o conceptos.

En la Ilustración 4 se muestra un ejemplo del prototipo propuesto por Zurita

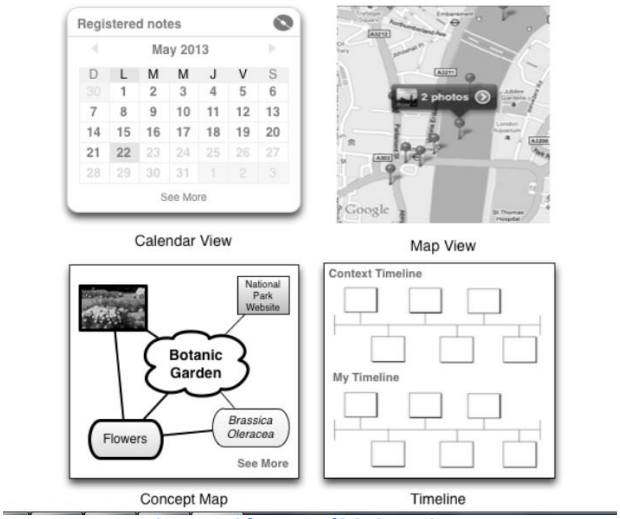



Ilustración 4: Múltiples vista

Este resumen da un contexto teórico y general del medio ambiente propuesto para la aplicación a desarrollar durante esta memoria. En el siguiente capítulo se explicará algunos aspectos técnicos relevantes para la implementación.

## 2.7 Fundamentos técnicos

En el desarrollo de aplicaciones Java destaca un framework muy utilizado denominado Spring. Su uso va desde simples aplicaciones Java SE hasta aplicaciones web sobre plataforma Java EE. Un aspecto importante de Spring es su gran cantidad de módulos independientes que proveen un ambiente apropiado para resolver las problemáticas propias de una aplicación web social, por ejemplo: seguridad, conexión a redes sociales, ORM, conexión a base de datos no relacional, búsquedas optimizadas, patrón MVC, entre otras.

Dentro del contexto de las redes sociales, la persistencia de los datos es una problemática relevante dado el volumen de información que se genera en ella. Apache Cassandra ofrece escalabilidad y alta disponibilidad sin degradar su desempeño.

### 2.7.1 Spring Framework

Spring es un framework es una plataforma que provee una extensa infraestructura para desarrollar aplicaciones en Java. Spring es modular, es decir, permite la utilización selectiva de sus funcionalidades. Los módulos principales son:

- Core: provee las características fundamentales del framework (IoC e Inyección de Dependencias). También provee un estándar para la utilización de los objetos dentro del framework. Soporta EL, utilizados por los JSP.
- AOP e Instrumentación: provee una implementación de AOP para programar métodos interceptores y pointcuts.
- Mensajes: provee funcionalidades para programación orientada a mensajes.
- Acceso e Integración de datos: provee una capa de acceso a los datos, utilizando JDBC, ORM, OXM, JMS y módulos de transacción.
- Web: provee funcionalidades básicas para soportar interacciones entre cliente y servidor a través de la web, tales como Servlet, patrón MVC, etc.
- Test: provee un ambiente de test unitarios y de integración a la aplicación.

En la Ilustración 5 se muestra la relación entre los distintos componentes del framework

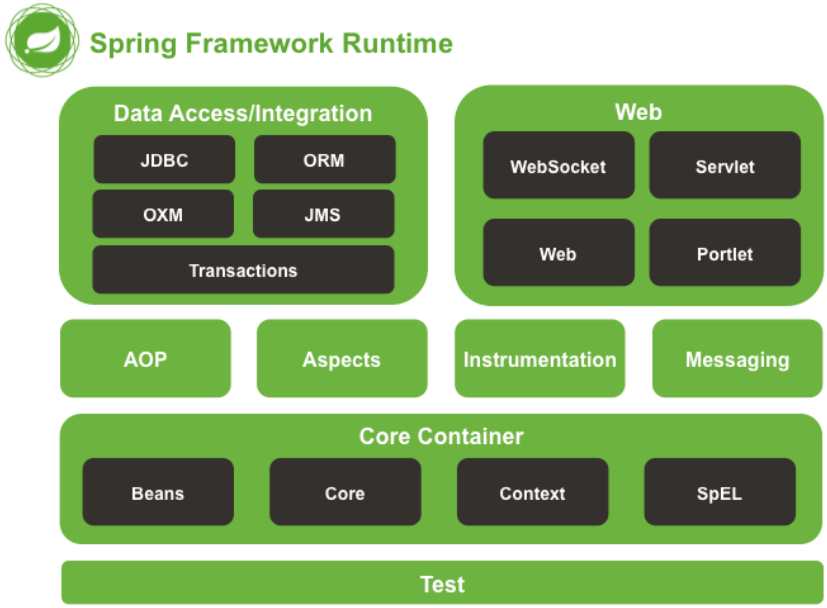



Ilustración 5: Spring Framework[5]

### 2.7.2 Apache Cassandra

Apache Cassandra es un administrador de base de datos distribuida NoSQL de código abierto que soporta escalabilidad y alta disponibilidad. Posee estrategias de replicación que otorgan un alto desempeño. Una de las principales ventajas es su manejo de índices sobre columnas, con registro de actualizaciones, desnormalización y vistas materializadas. Cassandra es utilizada por empresas como eBay, CERN, Instragram y GitHub.

La estructura sobre la cual se guardan los datos en esta base de datos difiere levemente con las bases de datos SQL. Una diferencia relevante son las columnas, ya que estas pueden generarse en dos niveles y cada nivel guardar datos, llave y valor. Esto permite personalizar una fila de acuerdo a un conjunto de columnas, o buscar sobre las llaves de las columnas. A continuación se presenta una analogía entre objetos en el modelo relacional y uno no relacional.

[5] http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/

| Relational Model | Cassandra Model |
|---|---|
| Database | Keyspace |
| Table | Column Family (CF) |
| Primary key | Row key |
| Column name | Column name/key |
| Column value | Column value |

Ilustración 6: Analogía entre modelo relacional y no relacional[6]

La representación de una *Column Familiy (Table para la versión 2.0)* tiene diversas configuraciones de acuerdo al subconjunto de columnas que se consideren. Un ejemplo es la presentada en la Ilustración 7, en donde destaca su llave primaria, las llaves por columnas y los valores de las columnas. Considerar que tanto las llaves de las columnas como los valores de las columnas pueden almacenar datos primitivos de Cassandra (Date, Varchar, Integer, Double, etc).

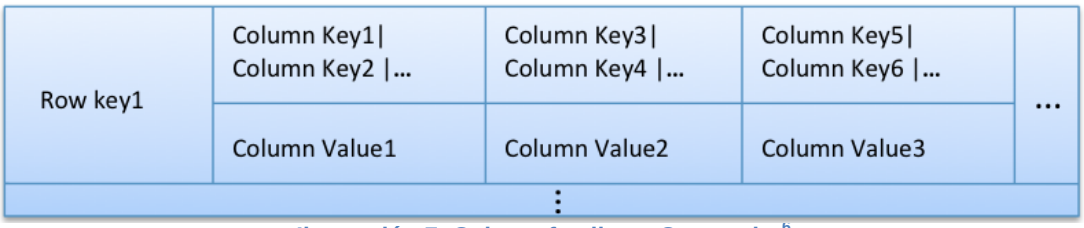



Ilustración 7: Column family en Cassandra[6]

La gran ventaja de definir llaves en las columnas se observan en sus posteriores consultas y búsquedas por rangos, ya que internamente Cassandra maneja dichas estructuras como mapas ordenados sobre las filas y también sobre las columnas. Tal como se puede observar en la Ilustración 8:

$$SortedMap < RowKey, SortedMap < ColumnKey, ColumnValue >>$$

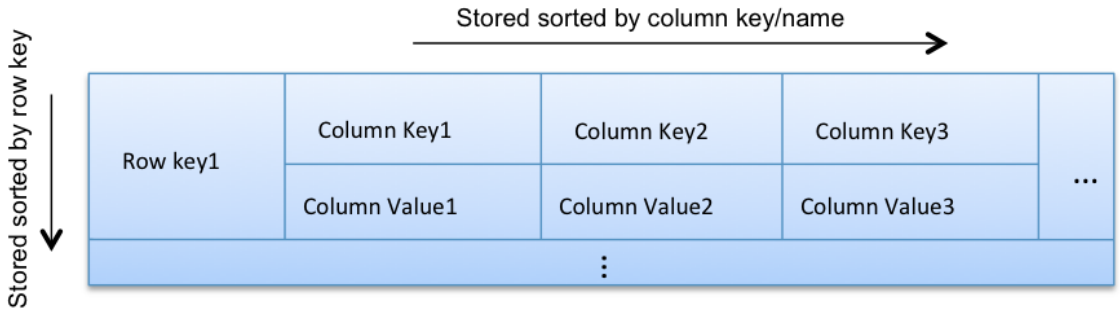



Ilustración 8: Mapa ordenado en las Column Family[6]

[6] http://www.ebaytechblog.com/2012/07/16/cassandra-data-modeling-best-practices-part-1/#.VHdBxIuG8k3

### 2.7.3 OAuth1 y OAuth2

OAuth es un protocolo de seguridad que permite a los usuarios de una aplicación (proveedor) autorizar el acceso de los datos de una aplicación tercera (consumidor) sin compartir la contraseña de la primera. OAuth 1 fue lanzado en el año 2007 y OAuth 2 en el año 2010. Aplicaciones que utilizan este protocolo son Facebook y Twitter para sus servicios de autentificación. A continuación se explica brevemente el flujo entre el consumidor y el proveedor en ambos casos.

**OAuth 1(Twitter)**

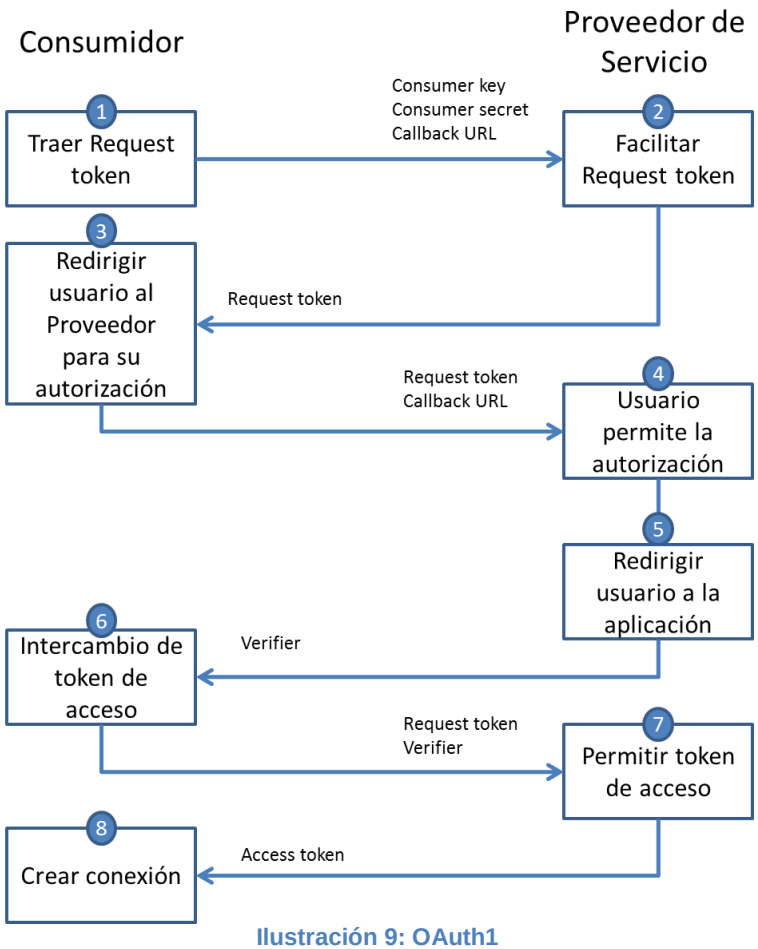



Ilustración 9: OAuth1

1. El flujo comienza con la aplicación preguntando por el Request Token. El propósito del request token es obtener la aprobación del usuario y solo puede ser obtenido con un token de acceso.
2. El proveedor de servicios entrega un request token al consumidor

3. La aplicación re direcciona al usuario a la página de autorización del proveedor de servicio, pasando el Request token como parámetro. También se entrega la URL de callback.
4. El proveedor de servicio lanza la consulta de autorización de la aplicación consumidora y el usuario lo acepta.
5. El proveedor de servicios re dirige al usuario a la página de la URL entregada como parámetro. En este punto el request token está autorizado. Entrega el verificador
6. La aplicación intercambia el request token autorizado para obtener un token de acceso. Le entrega el verificador como parámetro.
7. El proveedor de servicio entrega el token de acceso.
8. El consumidor recibe el token, lo asocia a la cuenta local del usuario y procede a establecer la conexión entre el usuario de la aplicación y el usuario del proveedor de servicio

**OAuth 2(Facebook)**

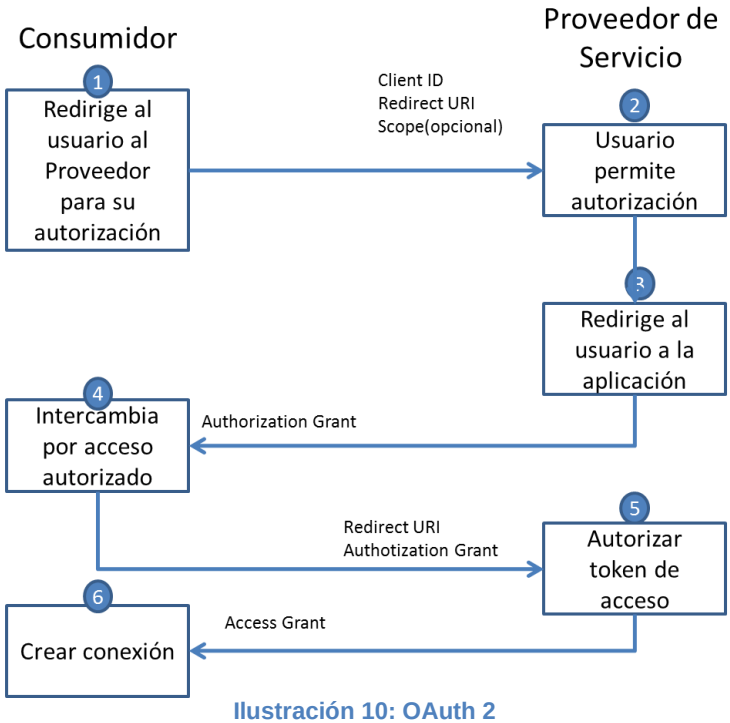



**Ilustración 10: OAuth 2**

1. El flujo comienza con la aplicación re direccionando al usuario a la página del proveedor. Ahí se le despliega una ventana solicitando al usuario que autorice el acceso y actualización de los datos por parte del consumidor.
2. El usuario autoriza el acceso a la aplicación

3. El proveedor de servicio redirige al usuario a la página del consumidor, a través de la URI entregada como parámetro.
4. La aplicación intercambia el código de autorización para obtener un acceso autorizado.
5. El proveedor de servicio entrega el acceso autorizado a la aplicación. La autorización incluye un token de acceso y un de actualización.
6. La aplicación utiliza el acceso autorizado para establecer una conexión entre la cuenta local de usuario y la cuenta del proveedor de servicio.

## 2.8 Servidor de objetos acoplados

Como parte de su memoria de título, Diego Aguirre, desarrolló un framework de objetos acoplados [15]. Dicho framework utiliza algunas ventajas de HTML5, y otras librerías, para sincronizar objetos de la interfaz de los clientes de tal forma que si alguno de estos cambia, el resto de los clientes vea el mismo cambio. Alguna de las características principales son: arquitectura replicada y orientada a mensajes, sincronización dinámica, sincronización parcial.

Este framework funciona de forma trasparente para las aplicaciones que la utilizan y lo hacen con un cliente liviano y simple escrito en JavaScript.

Dado su simplicidad y bajo costo de integración, se decidió considerar esta herramienta para proveer interacciones en tiempo real entre los clientes.

# 3 ESPECIFICACIÓN DEL PROBLEMA

## 3.1 Descripción del problema

El contexto urbano en los próximos años exigirá un conjunto de herramientas para manipular la información que provendrá de distintas fuentes: censores, cámaras, sistemas integrados, y en particular, las personas como generadora de información. Resulta necesario entregar una herramienta que entregue valor a la información generada en un contexto urbano, mediante múltiples visualizaciones con el fin último de ayudar al proceso de toma de decisión en todos sus niveles de acción.

Un caso particular de interés corresponde a la elección de proyectos en un condominio de parcelas en la comuna de Talagante. Durante el primer trimestre del 2015, los residentes deberán elegir los proyectos a desarrollar por la directiva durante ese mismo año. En las reuniones anteriores entre la directiva y los habitantes se obtuvieron 18 proyectos en carpeta que deberán ser discutidos y seleccionados de acuerdo a las condiciones y presupuesto que se dispongan. Los proyectos se distribuyen por todo el condominio, y cubren diversas necesidades de las personas que habitan en él. Destacan proyectos relacionados con la seguridad, infraestructura, recreación y deporte.

## 3.2 Requerimientos de la aplicación

Luego de tener los objetivos determinados para la aplicación, se levantaron los requerimientos principales:

- Aplicación web: La aplicación debe ser web, de esta forma se podrá acceder tanto por dispositivos móviles como por notebook o pcs que posean un navegador de internet.
- Registro de usuarios: La aplicación debe tener usuarios propios. Se debe proveer una interfaz para registrarse e ingresar. Los residentes del condominio podrán registrarse completando el formulario. Además la directiva poseerá un usuario especial encargado de precargar los datos necesarios al sistema.
- Ingreso de eventos urbanos: La aplicación debe facilitar el registro de nuevos eventos en la aplicación. Debe contener al menos un título y una

descripción. El usuario de la directiva ingresa los 18 proyectos en carpeta con su respectiva información. Además se debe permitir ingresar otros proyectos a cualquier usuario del sistema.

- Asociación geográfica del evento: La aplicación debe permitir asociar un evento a una ubicación geográfica. La interfaz de asociación es mediante un mapa.
- Asociación temática del evento: La aplicación debe permitir asociar un evento a tema en particular, generado previamente o creado por el usuario. Los temas relacionados en los distintos proyectos son cargados al sistema por el usuario de la directiva del condominio. Además se debe proveer un mecanismo para ingresar nuevos temas no incluidos por la directiva y que pueden ser asociados en cualquier momento por los usuarios a un determinado proyecto.
- Visualización de los eventos en un mapa geográfico: La aplicación debe mostrar los eventos asociados geográficamente en un mapa y permitir la interacción con los mismos.
- Visualización de los eventos en un mapa conceptual: La aplicación debe mostrar los eventos asociados temáticamente en un mapa conceptual y permitir la interacción con los mismos.
- Estadística asociada a los temas: Debe mostrar información estadística relevante de los eventos y temas. Se deberá mostrar las temas más asociados, los proyectos más comentados, la distribución de los votos por proyecto y los usuarios que más proyectos han ingresado al sistema.
- Conexión con otras redes sociales como Facebook y Twitter: La aplicación proveerá una funcionalidad para que el usuario ingrese a la aplicación mediante su cuenta de Facebook y/o Twitter.

## 3.3 Consideraciones del diseño de la aplicación

Con el fin de entregar una experiencia completa al usuario de la aplicación, se decidió crear un nombre, un logo y una nomenclatura para los distintos elementos desplegados.

El nombre para la aplicación es Feedback, que viene de retroalimentación. El sistema (personas + ciudad + gobierno) requiere de una constante retroalimentación para adaptarse a los cambios y así entregar una mejor calidad de vida a sus habitantes. Ese es el espíritu detrás de Feedback

Los elementos de Feedback son:

- Feed: corresponde al evento reportado por un usuario. Posee un título, una descripción, comentarios, tags y votos.
- Tag: corresponde a un tema en particular, por ejemplo: ciudad, contaminación, accidente, etc. Permite relacionar feeds. Un usuario puede seguir un conjunto de tags para obtener información de los feeds asociados a ellos.
- Comentario: corresponde a una declaración por parte de un usuario a un determinado Feed.
- Voto: corresponde a una herramienta para entregar validez a un feed.

# 4 DISEÑO DE FEEDBACK

## 4.1 Descripción

Feedback fue diseñado en cuatro etapas consecutivas. En primer lugar se evaluó el hardware necesario para soportar los requerimientos asegurando desempeño, escalabilidad y disponibilidad. Luego se creó el diseño de las componentes principales y su interacción dentro de la aplicación. Posteriormente se diseñó el modelo de datos SQL y NoSQL que persistirá la información. Finalmente se crearon mockups de las interfaces de usuario.

## 4.2 Diseño de la arquitectura física

La arquitectura física sigue una arquitectura cliente servidor. El cliente es un navegador de internet, ya sea desde un dispositivo móvil o notebook/pc. Este navegador debe soportar la especificación HTML5, el que provee librerías para geolocalización y manejo de imágenes con canvas.

En el lado del servidor, Feedback posee un servidor web/aplicación, Pivotal tc Server, el cual es un servidor Apache Tomcat 7.0 optimizado para Spring. Junto con este servidor también existe el Servidor de Objetos Acopados (COS) que se encarga de sincronizar los mensajes generados en cada cliente. La sincronización de mensajes es trasparente para Feedback, es decir no existe comunicación directa entre esta última y COS. Para asegurar la consistencia de los datos, primero se envían los cambios a Feedback y luego de persistir la información, el cliente envía el mensaje al resto de los clientes por medio del COS.

La capa de datos se separó en dos bases de datos, una relacional que contiene la información general de la aplicación y una segunda base de datos utilizada para manejar aquellas estructuras que demanden una alta escalabilidad. Para ello se utilizó Postgresql 9.3 y Apache Cassandra 2.0 respectivamente. Si bien ambas son bases de datos estructuradas, en Postgresql las tablas están normalizadas y las consultas se construyen a partir del cruce de tablas, en cambio en Cassandra, las tablas son parcialmente desnormalizadas, es decir, las tablas contienen la información agregada según lo requieran y no es necesario hacer cruces con otras tablas (Cassandra no soporta JOINS ni Subquerys).

En la Ilustración 11 se muestra la relación entre cada una de las componentes de la arquitectura física.

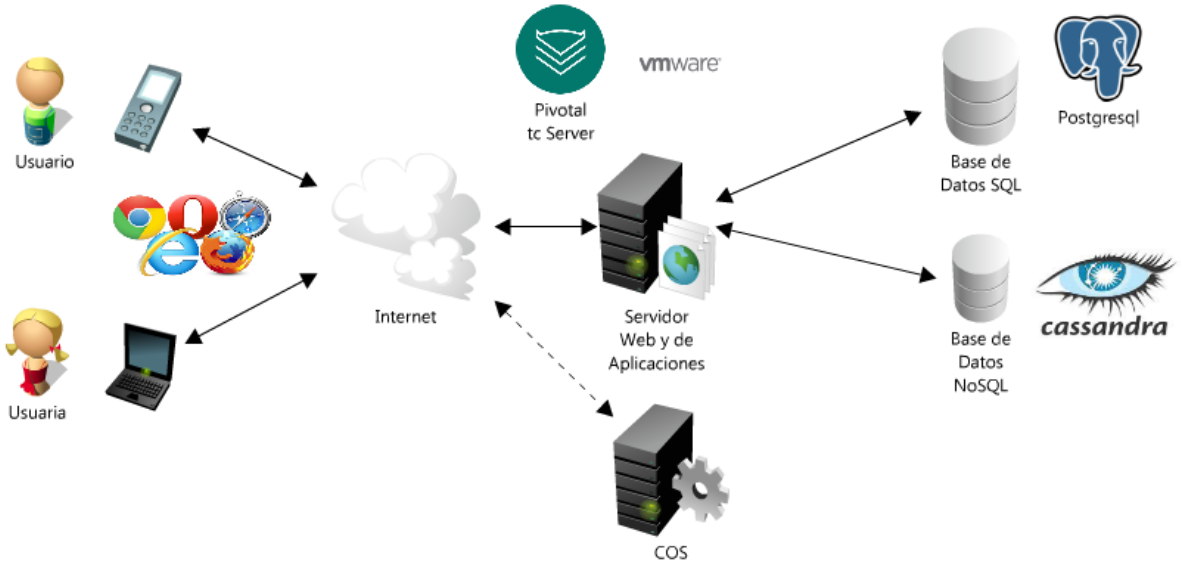



Ilustración 11: Arquitectura física

## 4.3 Diseño de la arquitectura lógica

La arquitectura lógica está separada en cuatro capas funcionales y una capa de datos con dos niveles de profundidad.

Las capas funcionales son:

- View (Vista): corresponde a las vistas de la aplicación disponibles para los usuarios. Usa tecnología JSP para generar el HTML que se despliega en el navegador. Además se incorporan lenguajes propios del cliente, tales como JavaScript y Css. Los formularios desplegados en las vistas se comunican directamente con los controladores.
- Controller (Controlador): corresponde al conjunto de métodos encargados de recibir los request del cliente y enviar los responses con la información solicitada. También se encarga de dejar disponible los recursos consumidos a través de AJAX. Se comunican con las vistas y la capa de servicios.
- Service (Servicio): corresponde a las funcionalidades específicas de aquellos objetos comunes dentro de la aplicación. Se comunican con los controladores y los repositorios. Poseen una interfaz que define los métodos que implementa.

- Repository (Repositorio): corresponde a la capa más cercana a la persistencia de los datos. Existe un repositorio por cada objeto mapeado en la base de datos. Incluyen métodos para realizar el CRUD, además se implementan funcionalidades de búsqueda especializados. Se comunica con la capa de servicio.

Los objetos, POJOs, que se intercambian a través de las distintas capas tienen dos niveles de profundidad dependiendo del contexto de su uso:

- Entity (entidad): corresponde al mapeo directo de las entidades de la base de datos a la aplicación. Solo están disponible en la capa de servicios y repositorios.
- Model (modelo): corresponde a los objetos que se utilizan para entregar información a las vistas. En general son simplificaciones de las entidades. Además se crean estructuras de datos específicas que resuelven problemáticas en las vistas, por ejemplo formularios, grafo, etc.

Los encargados de transformar las entidades en modelos, y viceversa, son los Mappers, instanciados en la capa de servicios.

En la Ilustración 12 se muestra las distintas capas de la arquitectura lógica.

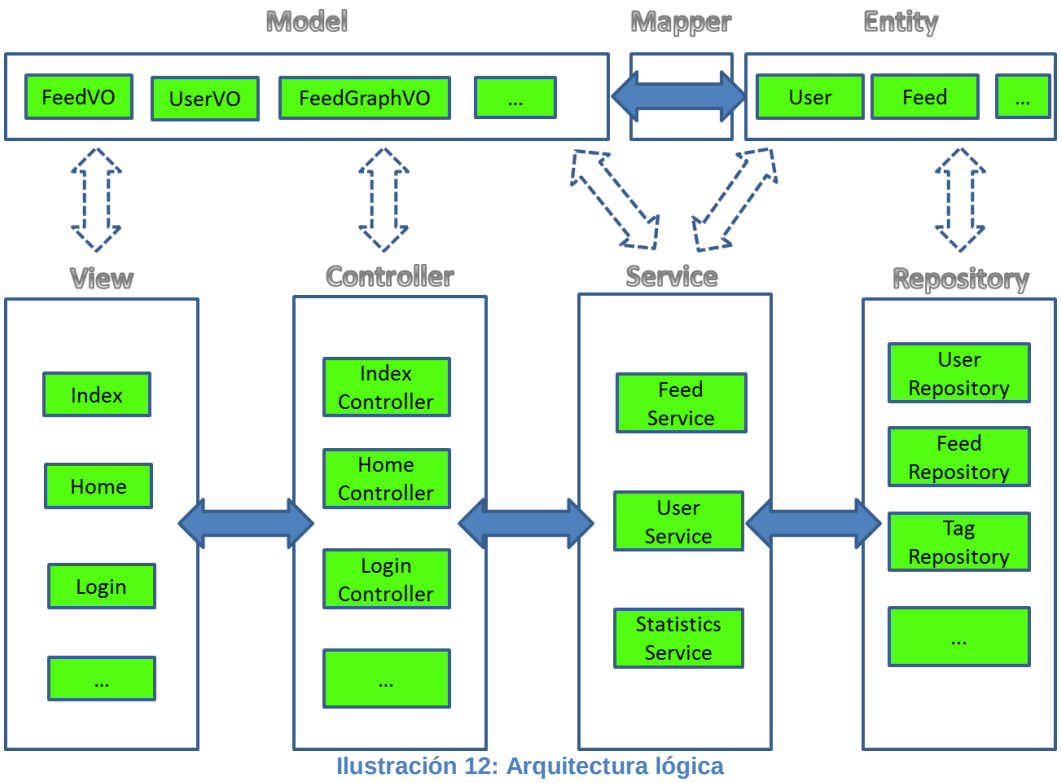



Ilustración 12: Arquitectura lógica

## 4.4 Modelo de datos

La capa de persistencia fue diseña con dos modelos, uno SQL y otro NoSQL. El primero tiene como propósito guardar la información de los distintos objetos creados dentro de la aplicación y sus respectivas asociaciones. La segunda tiene como propósito guardar datos que requieran una mayor escalabilidad, por ende tienen una estructura diferente al modelo relacional y corresponde a estructuras con datos desnormalizados.

Dos elementos importantes que exigen esta funcionalidad son los feeds y los tags. Dado que son los elementos principales dentro de Feedback, ellos pueden ser creados y asociados libremente por los usuarios, por lo tanto se debe asegurar escalabilidad y desempeño en su uso.

Dado que el contexto de uso es muy similar a una red social, el diseño del software debe considerar escalabilidad. Si bien las bases de datos SQL poseen un alto desempeño y confiabilidad, no están diseñadas para soportar BigData.

Facebook debe mantener millones de personas con su respectiva información y para ello ha desarrollado una base de datos dedicada y parte de ese resultado es Apache Cassandra. Caso análogo con Ebay o Github, donde los volúmenes de información son parte de su realidad.

Si bien no se espera tener un millón de usuarios en sus inicios, Feedback está diseñado para llegar en algún momento a esa cantidad de personas. Se entiende también que el escalamiento es un proceso iterativo por lo que se deberá seguir desarrollando para responder de la mejor forma ante esa situación.

### 4.4.1 Modelo SQL

Las entidades más relevantes en Feedback son:

- *feed*: corresponde al elemento que contiene la información que desea publicar un usuario. Tiene título, descripción, ubicación, fecha de creación, entre otros.
- *region*: corresponde a las regiones de Chile. Tiene nombre y número de región

- *comuna*: corresponde a las comunas de Chile. Tiene nombre y región a la cual corresponde.
- *location_table*: corresponde a una ubicación geografica. Posee latitud, longitud, dirección, comuna, entre otros.
- *origin*: corresponde al origen de los datos del feed. Este puede ser desde Feedback, Facebook, Twitter, u otra red social.
- *visibility*: corresponde al nivel de visibilidad de un objeto. Puede ser privado, público o restringido.
- *tag*: corresponde a una etiqueta que será asociada a los feeds y también pueden ser seguidos por los usuarios. Posee nombre, visibilidad, usuario, entre otras.
- *feed*_tag: corresponde a la asociación entre un feed y un tag. Tiene un feed, un tag, un usuario, fecha de creación, entre otras.
- *rating*: corresponde al voto de un usuario a un feed. Posee puntaje, un usuario, un feed y fecha.
- *comment*: corresponde a un comentario realizado por un usuario sobre un feed. Posee un feed, un usuario y el comentario propiamente tal.
- *user_table*: corresponde a un usuario de la aplicación. Tiene nombre, nombre de usuario, contraseña, email, sexo, entre otros.

Para ver más detalle de las entidades y sus relaciones ver el anexo A2.

### 4.4.2 Modelo NoSQL

El modelo NoSQL posee dos objetos que representan a un feed (con sus datos agregados) y al conjunto de tags que sigue un usuario.

*Feed* representa a la información que se guarda en el modelo SQL pero de forma agregada. Posee los datos propios del feed, además el conjunto de comentarios y tags, entre otros campos. En la Tabla 2 se muestra un esquema del modelo.

| ***feed*** | data | | | | comments | | | | tags | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| feed_id | title | description | user_id | location | id | id | id | … | id | id | id | … |

**Tabla 2: Modelo NoSQL - feed**

El conjunto de tags que sigue un usuario está diseñada en el objeto *user_tag_session* y por cada usuario posee los identificadores de los tags que el usuario sigue. En la Tabla 3 se muestra dicho modelo.

| ***user_tag_session*** | tags | | | |
|---|---|---|---|---|
| user_id | id | id | id | … |

**Tabla 3: Modelo NoSQL - user_tag_session**

## 4.5 Interfaz

Las interfaces de usuario principales son: registro, login, interacción con los feeds, agregar feeds y estadística de feeds.

La interacción del usuario con el mapa, el mapa conceptual y los feeds es a través de una vista integrada. En el costado izquierdo se muestra la lista de feeds, con su título, descripción, comentarios y tags asociados. Además se entrega información del usuario creador y fecha. También se proveen funcionalidades para comentar, agregar tags y votar. En el costado derecho parte superior se muestra el mapa, donde se muestran marcadores por cada feed con geolocalización. En el costado izquierdo parte inferior se encuentra el mapa conceptual con los tags de la aplicación y los feeds asociados a dichos tags. Destacar que existe interacción entre las tres componentes, es decir, al seleccionar un feed desde alguna de las componentes, este debe seleccionarse, cuando sea posible, en el resto de las componentes.

Otro aspecto del diseño fue la creación del logo de Feedback que rescata la esencia de su nombre, un intercambio continuo de ideas.

A continuación se presentan alguna de las interfaces del usuario:

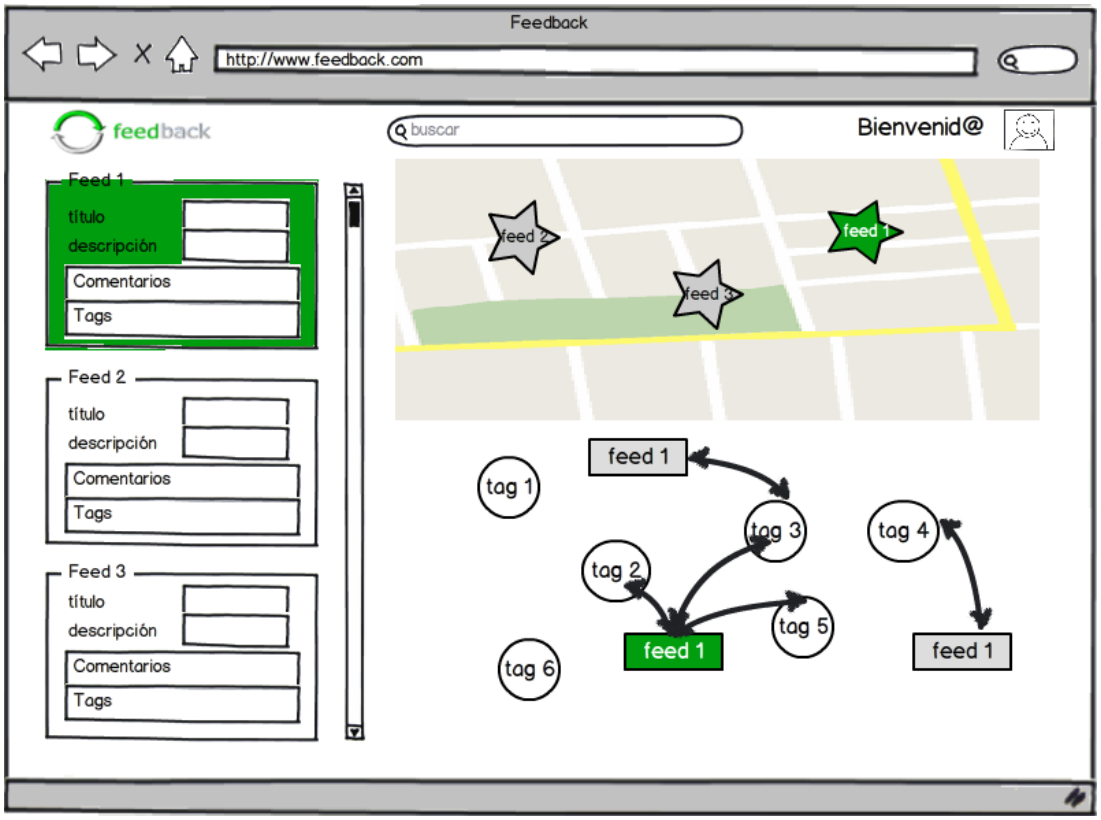



Ilustración 13: Interacción entre feeds, mapa y mapa conceptual


Ilustración 14: Logo de Feedback

# 5 IMPLEMENTACIÓN

## 5.1 ORM y Repositorios

El mapeo desde las bases de datos, relacional y no relacional, hacia la aplicación y viceversa se realizó con Spring Data JPA y Spring Data Cassandra respectivamente.

Los objetos mapeados con Spring Data JPA fueron 16, los cuales resultaron ser POJOs con anotaciones propias de JPA en la declaración de sus atributos. Para más detalle ver el diagrama de clases de las entidades que está en el anexo A3.

En la Ilustración 15 se muestra el objeto Feed que representa a la tabla con su mismo nombre y caracteriza al elemento publicado por un usuario en la aplicación. Se omiten los getters y setters del objeto.

Lo primero que se debe destacar es la notación @*Entity* e indica que esa clase es una entidad persistida en la base de datos. Con la notación @*Table* se indica el nombre de la tabla a la que se hace referencia. La notación @*Id* representa la llave privada y @*Column* al nombre de la columna dentro de la tabla.

Los tipos de datos de la base de datos son mapeados generalmente por un tipo de datos análogo en java:

Java ↔ Postgresql

- String ↔ text
- Integer ↔ integer
- Date ↔ timestamp

Para el tipo Date se utilizó la notación @*Temporal* para extender el tipo primitivo de Java al tipo *timestamp* de Postgresql.

Otro aspecto es el mapeo de objetos complejos, que son aquellos en donde sus atributos son otros objetos de la aplicación. Tal es el caso de *User* en *Feed*. Para ello se utilizó la notación @*JoinColumn* que establece una relación en *Feed* del objeto *User*, es decir, la tabla feed posee una columna llamada *user_id,* que es llave foránea y hace referencia a la tabla usuario. Además se establece que es una relación 1-1 con la notación @*OneToOne,* línea 70 y 71 de la Ilustración 15.

```
40  @Entity
41  @Table(name="feed")
42  @Indexed
43  public class Feed implements Serializable{
44
45          private static final long serialVersionUID = 5546407254241348577L;
46
47          @Id
48          @GeneratedValue(strategy=GenerationType.AUTO)
49          @Column(name="feed_id")
50          private Integer id;
51
52          @Column
53          @Field(index=Index.YES, store=Store.NO)
54          @Analyzer(definition = "customanalyzer")
55          private String title;
56
57          @Column
58          @Field(index=Index.YES, store=Store.NO)
59          @Analyzer(definition = "customanalyzer")
60          private String description;
61
62          @Temporal(TemporalType.TIMESTAMP)
63          @Column(name="created_date")
64          private Date createdDate;
65
66          @JoinColumn(name="location_id")
67          @OneToOne(cascade=CascadeType.ALL)
68          private Location location;
69
70          @JoinColumn(name="user_id")
71          @OneToOne
72          private User user;
73
74          @JoinColumn(name="origin")
75          @ManyToOne
76          private Origin origin;
77
78          @JoinColumn(name="visibility")
79          @ManyToOne
80          private Visibility visibility;
81
82          @OneToMany(mappedBy = "feed")
83          private List<Note> notes;
84
85          @OneToMany(mappedBy = "feed")
86          private List<Media> medias;
87
88          @OneToMany(mappedBy = "feed")
89          private List<Comment> comments;
```

**Ilustración 15: Objeto Feed**

Aquellos objetos declarados en *Feed* pero que su referencia no está ahí, sino en los objetos atributos, se utilizó mappedBy. Un ejemplo de esto es la lista de *Comments* de la línea 88 de la Ilustración 15.

En el caso de los objetos complejos, se decidió utilizar la estrategia Lazy para obtención de la información, es decir, los objetos de la aplicación que son atributos no se traen completamente por defecto (no se realiza el join con la tabla respectiva) y solo son cargados cuando se necesiten. Esto permite mayor eficiencia a la hora de consultarlos. Si no se hubiese considerado esta estrategia, el resultado de la búsqueda

de todos los feeds hubiese tenido un pobre desempeño ya que por cada feed se cargarían todos los comentarios, tags, votos, etc.

Los objetos mapeados con Spring Data Cassandra utilizan un estándar similar a JPA pero con sus propias notaciones. Un ejemplo de esto es *UserTagSession* que representa a la asociación de un tag a un usuario. La Ilustración 16 muestra dicho objeto. Se omiten los getters y setters del objeto.

```
13  @Table(value="user_tag_session")
14  public class UserTagSession implements Serializable {
15
16          private static final long serialVersionUID = 1L;
17
18          @PrimaryKeyColumn(name = "user_id", ordinal = 0, type = PrimaryKeyType.PARTITIONED)
19          private Integer userId;
20
21          @PrimaryKeyColumn(name = "tag_id", ordinal = 1, type = PrimaryKeyType.CLUSTERED)
22          private Integer tagId;
23
24          @Column(value="date")
25          private Date date;
```

**Ilustración 16: Objeto UserTagSession**

La notación *@Table* indica que el objeto está mapeado con la *Column Family* denominada user_tag_session. Dado que Cassandra permite llaves por fila y por columnas se utilizó la notación *@PrimaryKeyColumn* con el atributo *PrimaryKeyType* para identificar si es la primera (Partitioned) o la segunda (Clustered), y que corresponden al identificador del usuario y al identificador del tag respectivamente. *@Column* es similar al de JPA.

Una característica de Cassandra son las *"wide row"* o filas extensas. La forma de observarlo es que cada vez que un usuario sigue un determinado tag, esta asociación se guarda de forma independiente del resto (es una columna más en la fila que tiene como llave el identificador del usuario) y cuando se consultan los tags de un usuario, retorna el conjunto de identificadores de los tags. Se realizan comparaciones sobre el conjunto de usuarios y no sobre el conjunto de las asociaciones de la aplicación. Suponiendo que hay n usuarios y m tags, el número de comparaciones, en el peor caso, en el primer conjunto son n y en el segundo conjunto son n x m. Por lo tanto, sin duda es una mejora en su desempeño.

Los repositorios son los encargados de interactuar directamente con la base de datos e implementan las funcionalidades CRUD. Gracias a Spring Data los repositorios son sencillos y no requieren demasiado desarrollo. La forma de construirlos es mediante la extensión de la clase *CrudRepositoy*. Las operaciones Create, Update y Delete son provistas por *CrudRepository*. Si bien también se proveen operaciones de lectura, se crean otras adicionales que son utilizadas por la aplicación. Un ejemplo de los objetos mapeados a la base de datos Postgresql es *FeedRepository* que se muestra en la Ilustración 17.

```
12  public interface FeedRepository extends CrudRepository<Feed, Integer>,FeedRepositoryCustom {
13          Feed findById(Integer id);
14          List<Feed> findByTitleContaining(String title);
15          List<Feed> findByTitleContainingIgnoreCase(String title);
16          List<Feed> findByDescriptionContaining(String description);
17          List<Feed> findByDescriptionContainingIgnoreCase(String description);
18          List<Feed> findByCreatedDateBetween(Date start, Date end);
19          List<Feed> findByLocationAddress(String name);
20          List<Feed> findByLocationAddressContaining(String name);
21          List<Feed> findByLocationAddressContainingIgnoreCase(String name);
22          List<Feed> findByLocationComunaName(String name);
23          List<Feed> findByLocationComunaNameIgnoreCase(String name);
24          List<Feed> findByLocationComunaRegionName(String name);
25          List<Feed> findByLocationComunaRegionNameIgnoreCase(String name);
26          List<Feed> findByUserUserName(String username);
27          List<Feed> findByUserUserNameIgnoreCase(String username);
28          List<Feed> findByUserId(Integer id);
```

**Ilustración 17: FeedRepository**

Primero notar que es una interfaz y no una clase. Spring Data se encarga de transformar la declaración del método en una query y retornar el objeto esperado. Para ello, el nombre del método debe seguir un patrón determinado. Comienza con findBy y posteriormente se anidan restricciones sobre los atributos que se están buscando. Por ejemplo el siguiente método:

```
List<Feed> findByTitleContainingIgnoreCase(String title);
```

Busca los feeds que tengan atributo title igual al String title entregado, ignorando las mayúsculas y minúsculas.

En el caso que el atributo sea un objeto complejo se procede de forma análoga, colocando el nombre del atributo y luego el nombre de los atributos del objeto anterior. Tal como se hace en el siguiente método que busca los feeds que tienen el atributo username del objeto *User* igual al String username entregado.

```
List<Feed> findByUserUserName(String username);
```

Para aquellas búsquedas más personalizadas se utiliza la notación *@Query* que permite la utilización del lenguaje JPQL para realizar la query. A continuación se muestra un ejemplo que retorna la lista de enteros correspondientes a los identificadores de los tags que tienen como identificador del usuario igual al Integer id que se le entrega.

```
@Query("select distinct ft.tag.id from FeedTag ft where ft.user.id=:id")
List<Integer> findByUserId(@Param("id") Integer id);
```

Los repositorios de los objetos mapeados en la base de datos NoSQL, utilizan la API que provee Cassandra para Java y que se incluye en Spring Data Cassandra. El objeto *CassandraOperations* implementa las funcionalidades CRUD. Además ofrece un conjunto de métodos para la utilización del lenguaje nativo de Cassandra, CQL. En la Ilustración 18 se muestra el repositorio *UserTagSessionRepository* encargado de manejar el objeto *UserTagSession* explicado anteriormente. La línea 18 se importa *CassandraOperations* y en la 24 se utiliza.

```
14  @Component
15  public class UserTagSessionRepository {
16          @Autowired
17          @Qualifier(value="cassandraTemplate")
18          public CassandraOperations cassandraOperations;
19
20          public List<Integer> getTagsIdsByUserId(Integer id){
21                  if(id==null)
22                          return null;
23                  String query="select tag_id from user_tag_session where user_id="+id;
24                  List<UserTagSession> list=this.cassandraOperations.select(query, UserTagSession.class);
25                  List<Integer> result=new ArrayList<Integer>();
26                  for(UserTagSession s:list)
27                          result.add(s.getTagId());
28                  return result;
29          }
```

**Ilustración 18: UserTagRepository**

Por último, existen casos más complejos de lectura de objetos, tales como búsqueda por texto y estadística, en donde es necesario implementar los métodos que consultan a la base de datos. En la sección 5.6.2 y 5.6.7 se explicarán en mayor detalle.

## 5.2 Servicios

La capa de servicios se implementó utilizando el patrón de diseño Adapter. Para ello se definió una interfaz por cada servicio, con la lista de métodos que serán consumidos por la capa de controladores y/o capas futuras como web services. El diagrama de clases de los servicios se puede ver en el anexo A4.

En total se crearon tres servicios:

- UserService: servicio encargado de proveer funcionalidades de usuario en la aplicación tales como crear o buscar.
- FeedService: servicio encargado de proveer funcionalidades propias de los feeds tales como crear feed, buscar por texto, asociar a tags, etc.
- StatistitcsService: servicio encargado de proveer funcionalidades propias de la estadística tales como feeds más votados, tags más asociados, etc.

Una de las características principales de los servicios es que utilizan los repositorios definidos anteriormente para entregar/recibir los objetos requeridos por la capa de controladores. En la línea 52 de la Ilustración 19 se importa el *CommentRepository* y en la línea 144 de la Ilustración 20 se utiliza dicho repositorio para guardar el comentario en la base de datos.

```
45
46  @Component
47  public class FeedService implements FeedServiceRemote {
48
49          @Autowired
50          FeedRepository feedRepo;
51          @Autowired
52          LocationRepository locationRepo;
53          @Autowired
54          UserRepository userRepo;
55          @Autowired
56          OriginRepository originRepo;
57          @Autowired
58          VisibilityRepository visibilityRepo;
59          @Autowired
60          ComunaRepository comunaRepo;
61          @Autowired
62          CommentRepository commentRepo;
63          @Autowired
64          RatingRepository ratingRepo;
65          @Autowired
66          FeedTagRepository feedTagRepo;
67          @Autowired
68          TagRepository tagRepo;
69          @Autowired
70          UserTagSessionRepository userTagRepo;
71
```

**Ilustración 19: FeedRepository (A)**

```
130         @Override
131         public Integer commentFeed(CommentVO commentVO){
132                 if(commentVO==null || commentVO.getFeed()==null || commentVO.getUser()==null)
133                         return null;
134                 User u=userRepo.findByUserName(commentVO.getUser());
135                 Feed f=feedRepo.findOne(commentVO.getFeed());
136                 if(u==null || f==null)
137                         return null;
138                 Comment c=new Comment();
139                 c.setComment(commentVO.getComment());
140                 c.setUser(u);
141                 c.setFeed(f);
142                 c.setCreatedDate(new Date());
143                 c.setLevel(commentVO.getLevel());
144                 commentRepo.save(c);
145                 return c.getId();
146         }
```

**Ilustración 20: FeedRepository (B)**

El método *commentFeed* recibe un comentario realizado en la aplicación por un usuario a un feed. Luego de chequear que los datos entregados son consistentes, se procede a crear el nuevo objeto y a guardarlo. Finalmente se retorna el identificador correspondiente.

## 5.3 VOs y Mappers

El primer nivel de profundidad de los objetos son las entidades mapeadas directamente a la base de datos, tal como se explicó en el capítulo 5.1. El segundo nivel corresponde a los VOs. Los VOs corresponden a simplificaciones o adaptaciones de las entidades para resolver algún requerimiento especial de la aplicación. Los Mappers son los objetos que se encargan de transformar las entidades en VOs y viceversa. Esta estrategia permite generar distintos VOs a partir de una entidad. El diagrama de clases de los VOs se encuentra en el anexo A5.

En la Ilustración 21 se muestra la simplificación del objeto *Location*. Se observa que los atributos comuna y región son String y no objetos de la aplicación como en la entidad. Esta estrategia permite un intercambio de información más personalizado a los requerimientos de la aplicación y no se restringe al modelo de datos.

```java
public class LocationVO implements Serializable {

	private static final long serialVersionUID = 5718031640849145228L;

	private Integer id;

	private BigDecimal lat;

	private BigDecimal lng;

	private String address;

	private String comuna;

	private String region;

	private Date createdDate;
```

Ilustración 21: LocationVO

Los Mappers extienden de una interfaz que define distintos niveles de objeto: Data, Basic y Summary. Para efectos de la aplicación solo se consideró Basic y corresponde al segundo nivel de profundidad. Los otros niveles quedan propuestos para un futuro desarrollo en donde se requieran. Además, dicha interfaz se define como una interfaz genérica, de esta forma es utilizable para transformar cualquier par de objetos. A continuación se muestra la implementación de un Mapper.

```java
public class LocationMapper implements Mapper<Location, LocationVO> {

	@Override
	public LocationVO getSummary(Location entity) {
		// TODO Auto-generated method stub
		return null;
	}

	@Override
	public LocationVO getBasic(Location entity) {
		if(entity==null)
			return null;
		LocationVO vo=new LocationVO();
		vo.setId(entity.getId());
		vo.setAddress(entity.getAddress());
		vo.setLat(entity.getLat());
		vo.setLng(entity.getLng());
		if(entity.getComuna()!=null){
			vo.setComuna(entity.getComuna().getName());
			vo.setRegion(entity.getComuna().getRegion().getName());
		}
		return vo;
	}
```

Ilustración 22: LocationMapper

La clase *LocationMapper* se encarga de transformar la entidad *Location* en *LocationVO*. El método getBasic verifica correctitud de la entidad entregada, crea el nuevo objeto VO, lo completa y finalmente lo retorna. Notar que de las entidades comuna y región solo rescata sus nombres.

## 5.4 Controladores

Para construir la aplicación web se utilizó Spring Web MVC. Las componentes principales son:

- Modelo: corresponde a los objetos que se envían/reciben hacia y desde el cliente. En la aplicación son los VOs explicados anteriormente.
- Vista: corresponde a los JSP que renderizan las vistas al cliente. Se explicarán en mayor detalle en el capítulo 5.5
- Controlador: corresponden a los métodos que exponen los recursos para ser accedidos por el cliente a través de Internet.

Los controladores disponibilizan un recurso a través de una URL. El recurso retornado puede ser de dos tipos: una vista JSP que se transforma en un HTML o un objeto JSON. Este último está diseñado para ser utilizado mediante llamadas AJAX desde el cliente. Esta es la capa encargada de utilizar los servicios, por lo que importan las interfaces según lo necesiten.

En la Ilustración 23 se muestra un ejemplo de controlador. Con la notación *@Controller* Spring reconoce la clase *HomeController* como una clase controlador. La línea 31 importa la interfaz *FeedServiceRemote*, la cual define los métodos de ese servicio. Con la notación *@RequestMapping* se define la URL de acceso y el método del mismo, en este caso será en */welcome* con el método GET. En las líneas 37 y 38 se solicitan objetos a la capa de servicio. En la línea 43 se define el nombre de la vista (JSP) que retornará y le carga tanto los objetos anteriormente solicitados como algunos que representen formularios, como por ejemplo *FeedVO* de la línea 44.

```
26
27  @Controller
28  public class HomeController {
29
30          @Autowired
31          FeedServiceRemote feedService;
32
33          @RequestMapping(value = { "/welcome**" }, method = RequestMethod.GET)
34          public ModelAndView home(String var) {
35                  Authentication auth = SecurityContextHolder.getContext().getAuthentication();
36              String username = auth.getName();
37                  List<TagVO> listTags=feedService.getAllTags();
38                  List<Integer> tagsFollow=feedService.getFollowingTags(username);
39                  ConfigTagVO configTag= new ConfigTagVO();
40                  if(tagsFollow!=null && tagsFollow.size()>0)
41                          configTag.setTagsIds(tagsFollow);
42                  ModelAndView model =  new ModelAndView();
43                  model.setViewName("home");
44                  model.addObject("feed", new FeedVO());
45                  model.addObject("comment", new CommentVO());
46                  model.addObject("rating", new RatingVO());
47                  model.addObject("listTags", listTags);
48                  model.addObject("configTag", configTag);
49                  return model;
50          }
```

**Ilustración 23: HomeController – home**

Los métodos que responden objetos JSON se ejemplifica en la Ilustración 24.El método publishFeedAJAX se encarga de recibir un *FeedVO* e invocar *FeedService* para que lo persista. La notación *@ResponseBody* le dice a Spring que la respuesta es un objeto *FeedVO* pero en formato JSON. Notar que el objeto feed del tipo *FeedVO* que recibe como parámetro, ya viene cargado con los datos desde el cliente por lo que solo falta verificar correctitud de los datos y llamar a la capa de servicios para que lo persista. Finalmente se retorna el Feed ya persistido.

```
105         @RequestMapping(value = "/ajax/publish_feed", method = RequestMethod.POST)
106         public @ResponseBody FeedVO publishFeedAJAX(@ModelAttribute("feed") FeedVO feed,
107                         BindingResult result) {
108                 if(feed.getTitle()==null || feed.getUser()==null ||
109                                 feed.getTitle().isEmpty() || feed.getUser().isEmpty())
110                         return null;
111                 Integer var=feedService.createFeed(feed);
112                 if(var==null)
113                         return null;
114                 else
115                         return feedService.findFeedById(var);
116         }
```

**Ilustración 24: HomeController – publishFeed**

## 5.5 Vistas

Las vistas son JSP que se cargan con los objetos entregados por los controladores. Se utilizaron distintos Tag Libs que permitieron facilitar el manejo de formularios, restricciones de seguridad, formatos de fecha y funcionalidades comunes de JSP. El uso de EL lo provee Spring Framework. Aquí además se importan los archivos que contienen los estilos de la página y las distintas librerías JavaScript que se ejecutan en el cliente. En el diseño se consideró la construcción de templates de JSP que se reutilizan frecuentemente en las distintas páginas, tal es el caso de header, footer y panel izquierdo. Los estilos ocupados son del template de Bootstrap denominado Unify y fue adaptado a las necesidades de la aplicación.

En la Ilustración 25 se muestra la forma de importar los tag libs desde la JSP. En este caso se cargan *security tags, jstl, form* y *fmt*. Su uso se puede ver en la Ilustración 26, donde se observa el tag *<sec:authorize>* que recibe como atributo access un booleano. En este caso pregunta al contexto si el usuario está autenticado. En el caso que no lo esté, se redirige a la URL raíz de la aplicación. También se puede observar cómo se agregan el topbar definido en otro JSP, con el tag *<jsp:include>.*

```
<%@taglib prefix="sec"
        uri="http://www.springframework.org/security/tags"%>
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt"%>
<%@page session="true"%>
<head>
<title>Feedback</title>
```

**Ilustración 25: Import de Tag Libs**

```
<body data-spy="scroll" data-target=".tab-v2">
        <sec:authorize access="!isAuthenticated()">
                <c:redirect url="/" />
        </sec:authorize>
        <!-- End Small Modal -->
        <div class="wrapper">
                <!--=== Header ===-->
                <div class="header">
                        <!-- Topbar -->
                        <jsp:include page="topbar.jsp"></jsp:include>
                        <!-- End Topbar -->
```

Ilustración 26: Tag Security

Los formularios se completan con los objetos entregados por los controladores, tal como lo muestra la Ilustración 27. Este formulario se creó para que un usuario comente un determinado feed. El tag *<form:form>* tiene como atributo modelAttribute el objeto comment, el cual será el que se completará y enviará a la URL especificada en el atributo action. El tag *<form:input>* recibe como atributo path y corresponde al nombre del atributo del objeto del formulario, en este caso *<form:input path="user">* hace referencia al atributo user de comment.

```
<form:form
        action='${pageContext.request.contextPath}/ajax/comment_feed'
        id="comment-form" modelAttribute="comment"
        class="sky-form sky-form-panel comment-form">
        <div class="hidden">
                <form:input path="user"
                        value="${pageContext.request.userPrincipal.name}" />
                <form:input path="level" value="0" />
                <form:input path="feed" value="{{feed.id}}" />
        </div>
        <div class="text-center">
                <label class="textarea"> <form:textarea row="2"
                                id="comment" name="comment" placeholder="Coméntalo!"
                                path="comment" cssClass="rounded comment" />
                </label>
        </div>
        <div class="text-center">
                <button type=submit class="btn-u btn-u-xs rounded">Enviar</button>
        </div>
</form:form>
```

Ilustración 27: Tag Form

## 5.6 Funcionalidades relevantes

### 5.6.1 Redes sociales

La conexión con las redes sociales más utilizadas, Facebook y Twitter, se realizó con el proyecto Spring Social. Spring Social permite establecer conexiones con proveedores de servicios SaaS tales como Facebook y Twitter por medio de sus usuarios.

Las componentes de software son:

- Un framework que maneja las funcionalidades principales de la autorización y el flujo de conexión con los proveedores de servicio.
- Un controlador que maneja el intercambio de información de los protocolos de autenticación, Oauth1 y Oauth2, entre la aplicación, el proveedor de servicios y el usuario.
- Un controlador de ingreso que permite al usuario autenticarse en la aplicación utilizando las cuentas de las redes sociales.
- Extensibilidad del framework para agregar otras redes sociales.

La integración de Spring Social con Feedback comenzó con la configuración de la aplicación. En primer lugar entregar las credenciales de cada red social, posteriormente definir el data source sobre el cual el framework asocia una cuenta de Feedback con la de las redes sociales. Y finalmente los distintos controladores web para comunicarse con el proveedor de servicio. En la Ilustración 28 se muestra un extracto del archivo.

Luego de la configuración, se procedió a compatibilizar la seguridad de la aplicación y la base de datos con el nuevo modelo para un usuario que se conecta desde su red social favorita.

```
<context:property-placeholder location="classpath:META-INF/application.properties" />

<facebook:config app-id="${facebook.appKey}" app-secret="${facebook.appSecret}"/>
<twitter:config app-id="${twitter.appKey}" app-secret="${twitter.appSecret}"/>
<linkedin:config app-id="${linkedin.appKey}" app-secret="${linkedin.appSecret}"/>

<social:jdbc-connection-repository data-source-ref="dataSource"/>
<bean id="userIdSource" class="org.springframework.social.security.AuthenticationNameUserIdSource" />

<bean id="connectController" class="org.springframework.social.connect.web.ConnectController" autowire="constructor"/>

<bean id="psc" class="org.springframework.social.connect.web.ProviderSignInController" autowire="constructor" />
<bean id="signInAdapter" class="cl.uchile.dcc.feedback.social.SimpleSignInAdapter" autowire="constructor">
        <constructor-arg ref="requestCache"/>
</bean>

<bean id="requestCache" class="org.springframework.security.web.savedrequest.HttpSessionRequestCache"/>

<bean id="disconnectController" class="org.springframework.social.facebook.web.DisconnectController"
        c:_0-ref="usersConnectionRepository" c:_1="${facebook.appSecret}" />
```

Ilustración 28: Configuración Spring Social

A continuación se muestran algunas capturas del proceso de autenticación utilizando las cuentas de Facebook y Twitter.

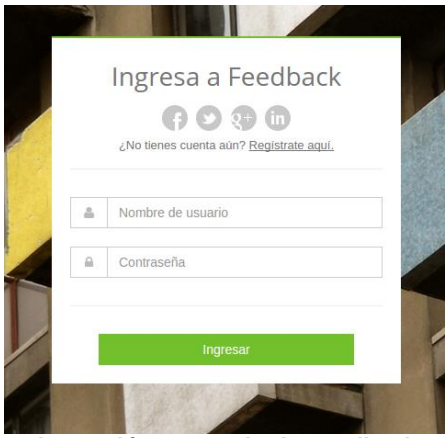



Ilustración 29: Login de Feedback


Ilustración 30: Autorización de Feedback en Facebook


Ilustración 31: Autorización de Feedback en Twitter

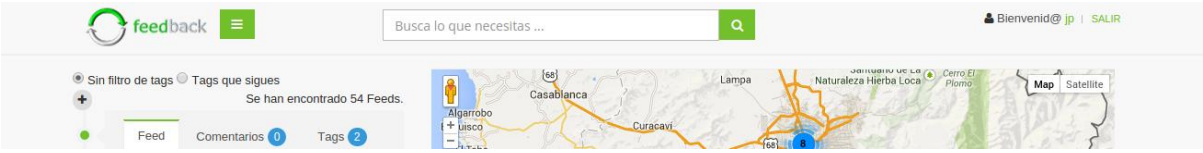



Ilustración 32: Re direccionamiento a Feedback

### 5.6.2 Búsquedas en la aplicación

Una de las complejidades en los sistemas con altos volúmenes de información es la búsqueda por texto. El enfoque simple pero costoso es hacer búsquedas utilizando el comparador LIKE a nivel de base de datos con el carácter % para buscar patrones o subconjuntos de caracteres. Esto es muy costoso en tiempo y desempeño por lo que se descartó.

Un segundo enfoque proviene de la utilización de herramientas especializadas. Una de ellas es Hibernate Search, proyecto que combina la potencia de Hibernate, implementación del estándar JPA, con Apache Lucene, motor de búsquedas textuales basada en índices inversos.

Las características principales de Hibernate Search son:

- Proveen una compatibilización entre los índices de Lucene y los objetos mapeados en la base de datos.
- Sincronización y optimización de acceso a los índices con respecto a los cambios en la base de datos.
- Búsquedas de objetos utilizando una API unificada.

Dado que *Feed* es uno de los objetos principales dentro de la aplicación, se decidió utilizar la funcionalidad de búsqueda sobre el título y la descripción. También sobre los comentarios asociados a los feeds.

La configuración es simple, solo se agrega la ruta de los índices en la unidad de persistencia. La integración con la aplicación es similar a la configuración de los objetos mapeados en la base de datos. En primer lugar se ocupa la notación *@Indexed* para que Hibernate Search cree índices sobre ese objeto. Luego, sobre los atributos que posteriormente se realizarán las búsquedas se anotan con *@Field* y *@Analizer*. Este último agrega meta datos a dicho atributo. El Analizer utilizado consiste en un Tokenizer (separador de palabras) y dos filtros (LowerCase y SnowBallPorter en español). En la Ilustración 33 se muestra un ejemplo con la clase *Feed*.

```
@AnalyzerDef(name = "customanalyzer",
tokenizer = @org.hibernate.search.annotations.TokenizerDef(factory = StandardTokenizerFactory.class),
filters = {
        @org.hibernate.search.annotations.TokenFilterDef(factory = LowerCaseFilterFactory.class),
        @org.hibernate.search.annotations.TokenFilterDef(factory = SnowballPorterFilterFactory.class, params = {
    @Parameter(name = "language", value = "Spanish")
  })
})
@Entity
@Table(name="feed")
@Indexed
public class Feed implements Serializable{

        private static final long serialVersionUID = 5546407254241348577L;

        @Id
        @GeneratedValue(strategy=GenerationType.AUTO)
        @Column(name="feed_id")
        private Integer id;

        @Column
        @Field(index=Index.YES, store=Store.NO)
        @Analyzer(definition = "customanalyzer")
        private String title;

        @Column
        @Field(index=Index.YES, store=Store.NO)
        @Analyzer(definition = "customanalyzer")
        private String description;
```

**Ilustración 33: Feed en Hibernate Search**

En Ilustración 34 se muestra la búsqueda de la palabra “talaga”, la cual encontró dos feeds que la contenían, ya sea en el título, descripción o comentarios. Gracias al tokenizer, Talagante fue matcheado como una respuesta válida para esta consulta.

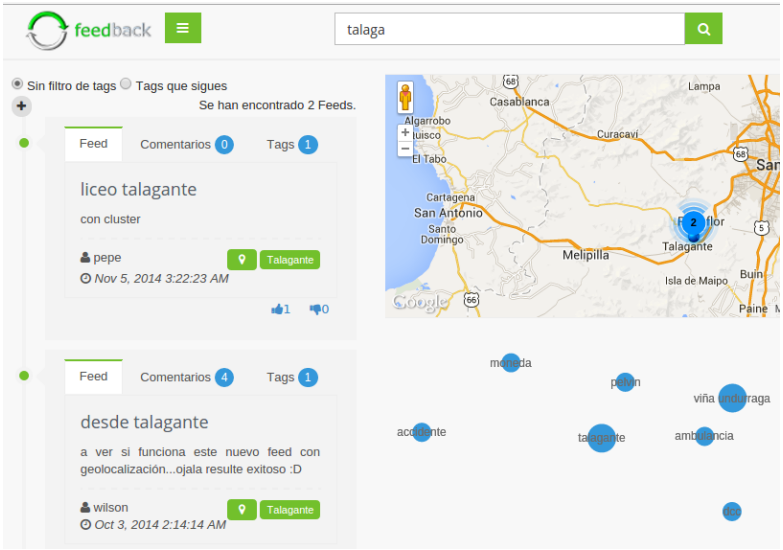



Ilustración 34: Búsqueda en Feedback

### 5.6.3 Sincronización de mensajes

El servidor de objetos acopados (COS) utilizado para sincronizar los mensajes entre los clientes, tiene dos estrategias de sincronización: por mensaje y por estado. En el primero los mensajes representan métodos del objeto acoplado. Con el segundo, lo que se sincroniza es su estado, es decir, los valores de sus atributos en ese momento. Si bien la segunda opción está mejor diseñada para ser escalable, tiene limitaciones ya que sobrecarga el COS con información propia de cada aplicación.

Se rediseñó el COS para soportar sincronización por mensajes escalables. El principal cambio consiste en modificar la forma de interacción del COS con el cliente. La nueva propuesta define al COS como un streaming de mensajes solo para aquellos mensajes creados después de la conexión del usuario con la aplicación.

Situación anterior:

1. Cliente se conecta a la aplicación
2. Acopla los objetos.
3. Se reciben todos los mensajes asociados a la sesión
4. Se actualiza el objeto
5. Se hacen cambios a través de la vista y se envían como mensajes
6. Los otros clientes reciben los mensajes y se actualizan.

Situación propuesta:

1. Cliente se conecta a la aplicación
2. La aplicación le entrega el estado de los objetos acoplados
3. Se acoplan los objetos
4. No se reciben mensajes al principio
5. Se hacen cambios a través de la vista y se envían como mensajes
6. Los otros clientes reciben los mensajes y se actualizan.

Los puntos 1 al 4 corresponden a las acciones realizadas por la aplicación luego de que el usuario entra a ella. El 5 y 6 son acciones que se generan a partir de la interacción del usuario con la aplicación.

La mejora está en no recibir los mensajes iniciales asociados a la sesión, ya que la aplicación se encarga de entregarle el último estado de los objetos. Y solo se reciben/envían mensajes después de la primera conexión. La estrategia es una mezcla entre sincronización por estado y por mensaje, la primera la recibe desde Feedback y la segunda desde COS.

Los cambios se localizaron principalmente en dos partes: la clase cliente, escrita en JavaScript para agregar la nueva opción y modificar el servidor, escrito en java, para que reciba dicho parámetro. En la Ilustración 35 se muestra el nuevo parámetro initMessage con valor false. Por defecto es true, para dejar compatible aplicaciones que ocupan COS. El servidor por su parte lo recibe y si no está definido o es true envía los mensajes iniciales, caso contrario no los envía.

```
Synchronizer.prototype.couple = function() {
    this.adapter.coupleObject("synchronizer", this, {
        messageType : "EVENT",
        explicitMapping : [ "syncMessage" ],
        initMessage : false
    });
};
```

```
@Override
public synchronized void couple(String objectId, ServerAdapter adapter,
        List<Message> messages,Boolean initMessage) {
    if (this.stateManager != null && (initMessage==null || initMessage)) {
        adapter.initializeObjectData(objectId, stateManager);
    }
```

**Ilustración 35: Cambio del COS - Clase cliente y Servidor**

### 5.6.4 Patrón MVC y AngularJS

Para sincronizar los mensajes se utilizó el servidor de objetos acoplados. Luego de recibir los mensajes fue necesario mostrar los cambios de los objetos en la interfaz del usuario, esto es en el mapa, la lista de feeds y en el mapa conceptual. La primera solución fue manipular los elementos del DOM solo con jquery o la librería estándar de JavaScript. Sin embargo, este enfoque dificulta el desarrollo ya que toda la sincronización entre objetos JavaScript y vistas se hace programáticamente. El segundo enfoque consistió en utilizar un framework MVC para JavaScript. Algunas opciones fueron Backbones, Ember y Angular. Se eligió Angular principalmente por su amplia documentación existente en la red.

AngularJS es un framework MVC desarrollado por Google y tiene como principal objetivo de diseño la articulación flexible entre la presentación, datos y componentes lógicas. Extiende el conjunto de tags disponibles en HTML para ligar algún modelo con objetos del DOM (una lista por ejemplo) o capturar eventos del DOM (evento clic en la lista).

El framework se utilizó con distintos objetos, y en particular con la lista feeds. Luego que la aplicación le entrega al cliente la lista de feeds en formato JSON, estos se guardan en el modelo de Angujar. A su vez este modelo se asoció a una lista HTML que se renderiza con los datos entregados.

En la Ilustración 36, línea 6 se muestra la declaración del objeto feeds como parte del modelo de Angular con *$scope*. Luego en la Ilustración 37 se modifica dicho modelo. Finalmente en la Ilustración 38 se muestra como se asocia la lista con el modelo, con el tag *ng-repeat*. Notar el uso de {{}} para acceder a los atributos del modelo.

```
1  var app = angular.module("myApp", []);
2
3  app.controller("homeController", function($scope) {
4          $scope.show_panel = false;
5          $scope.allFeeds = undefined;
6          $scope.feeds = undefined;
```

Ilustración 36: Modelo en Angular

```
a.$apply(function(){a.feeds=feeds;});
a.$apply(function(){a.allFeeds=feeds;});
```

Ilustración 37: Cambio en el modelo Angular

```
<li ng-repeat="feed in feeds"><i
        class="cbp_tmicon rounded-x hidden-xs"></i>
        <div class="cbp_tmlabel">
                <div class="text-justify">
                        <div class="tab-v2" id="feed-{{feed.id}}">
                                <ul class="nav nav-tabs">
                                        <li class="active"><a href="#home-{{feed.id}}"
                                                data-toggle="tab">Feed</a></li>
                                        <li><a href="#profile-{{feed.id}}" data-toggle="tab">Comentarios
                                                <span class="badge rounded-2x badge-blue">{{feed.comments.length}}</span>
```

Ilustración 38: Vista asociada al modelo Angular

### 5.6.5 Mapa y Google Maps

Uno de los requerimientos de la aplicación era la visualización del contenido de acuerdo a su ubicación geográfica. Una de las mejores librerías para este propósito es Google Maps. Posee una extensa API en JavaScript en su versión 3, y entre otras cosas, permite obtener la latitud y longitud de un punto en el mapa. Dicha ubicación se guarda, en el caso que el usuario lo determine, junto con la información del feed para posteriormente utilizarla y mostrar un marcador donde corresponde. Dado el objetivo de este documento, no se entrará en detalle del conjunto de funcionalidades utilizadas. Sin embargo se ejemplificará la inicialización del mapa y la creación de marcadores.

En la Ilustración 39 se define el contenedor que tendrá el mapa de Google Maps, posteriormente en la Ilustración 40 se inicializa el mapa en el contenedor definido anteriormente con algunas opciones por defecto.

Un consideración a la hora de utilizar los marcadores de Google Maps fue que no soportan meta data de forma transparente, por lo que se decidió crear un objeto JSON, *Mark*, que contenía el marcador de Google en su atributo marker y meta data como el identificador del feed en el atributo feed. Este objeto es extensible para agregar más meta data en el futuro. En la Ilustración 41 se muestra la creación de dicho objeto.

```
<!-- Begin Content -->
<div class="col-md-8">
        <div id="map" class="map"></div>
```

Ilustración 39: Contenedor del mapa

```
var mapOptions = {
        zoom : 14,
        center : santiago
};
map = new google.maps.Map(document.getElementById('map'), mapOptions);
```

Ilustración 40: Inicialización del mapa

```
function addMarker(data){
        var ll=new google.maps.LatLng(data.location.lat, data.location.lng);
        var newMark={marker:new google.maps.Marker({
                                                    animation : google.maps.Animation.DROP,
                                                    position : ll
                                                }),
                                feed:data.id};
        markers.unshift(newMark);
        return newMark;
}
```

**Ilustración 41: Objeto Mark**

### 5.6.6 Mapa Conceptual y ArborJS

Otro requerimiento importante de la aplicación era la visualización de la información a través de mapas conceptuales. La estructura subyacente a un mapa conceptual puede ser modelada por medio de un grafo. En particular corresponden a grafos no dirigidos que pueden contener ciclos. El grafo se diseñó con dos tipos de nodo: nodo interno y nodo terminal. El nodo interno corresponde a los tags de la aplicación y el nodo terminal corresponde a un feed. Las aristas se crean cuando un usuario asocia un feed a un determinado tag, es decir se crea una relación entre el nodo terminal y el nodo interno respectivamente. Un tag puede estar asociado a varios feeds y un feed puede estar asociado a varios tags. No existe relación directa entre nodos internos, solo a través de algún feed, es decir, a través de los nodos terminales. Mientras más relaciones existan de un feed con dos tags, más próximos estarán ambos nodos internos.

Para soportar la estructura descrita anteriormente se definió el objeto *FeedGraphVO*, que contiene una lista de nodos (*NodeVO*) y una lista de aristas (*EdgeVO*). Para más detalle en el anexo A5 está el diagrama de clases con los tres objetos mencionados anteriormente.

La forma de diferenciar a un nodo interno de un nodo terminal, es su atributo feed <0 para el primer caso y >0 para el segundo. Por su parte las aristas tienen un nombre, desde donde nacen las aristas y que corresponde al nombre del tag, y también poseen una lista de String, hacia donde llega la arista y que corresponde a los títulos de los feeds concatenados con sus identificadores. De esta forma la definición del grafo puede ser interpretada y extendida para ser dibujada por diferentes librerías JavaScript.

La librería utilizada fue ArborJS que permite visualizar grafos con funcionalidades de HTML5 y jquery. En la capa de servicios se creó un método para retornar la estructura antes mencionada, que luego se deja disponible en la capa de controladores para ser consumida a través de AJAX desde el cliente. Una vez que cliente tiene la estructura en JSON, se procede a inicializar el grafo. En la Ilustración 42, línea 253 se reconoce si es un nodo interno o nodo terminal y se agrega a la lista de nodos. Posteriormente se procede a agregar las aristas en la línea 267 y finalmente se crean los objetos que se encargan de la visualización e interacción en el grafo, líneas 276 y 277.

```
251  function loadGraph(graph){
252          var n={};
253          for(var i=0;i<graph.nodes.length;i++){
254                  if(graph.nodes[i].feed==-1)
255                          n[graph.nodes[i].name]={color:CLR.branch, shape:"dot", alpha:1,size:graph.nodes[i].nFeeds};
256                  else
257                          n[graph.nodes[i].name]={color:CLR.feed, alpha:0,link:'/'+graph.nodes[i].feed};
258                  tagsArray.push(graph.nodes[i].name);
259          }
260          var e={};
261          for(var j=0;j<graph.edges.length;j++){
262                  var ed=graph.edges[j];
263                  var sub={};
264                  for(var k=0;k<ed.edges.length;k++){
265                          sub[ed.edges[k]]={length:10};
266                  }
267                  e[ed.name]=sub;
268          }
269          var theUI = {
270                          nodes:n,
271                      edges:e
272                    }
273
274                sys = arbor.ParticleSystem({stiffness:5000, repulsion:10000,gravity:false,friction:0.9, dt:0.015})
275                //sys.parameters()
276                sys.renderer = Renderer("#sitemap")
277                sys.graft(theUI)
278                $(".Fullscreen").addClass("hidden");
279  }
```

**Ilustración 42: Inicialización del grafo**

### 5.6.7 Estadística y HighCharts

Se diseñaron tres tipos de estadística de acuerdo a los objetos más relevantes de la aplicación: feed, tag y usuario. En particular se muestran los feeds más votados y los más asociados, los tags más asociados y los usuarios que más feeds han publicado. La estructura básica utilizada fue *StatisticsDataVO* y representa a un dato con su valor estadístico (frecuencia, porcentaje, etc,). También se definió una serie como el conjunto de datos estadísticos asociados a un elemento en particular, por

ejemplo para los feeds más votados, cada feed es una serie y los datos al interior de la serie son su cantidad de votos positivos y votos negativos, es decir cada uno de ellos es un *StatisticsDataVO*. Finalmente se definieron tres clases que agrupan a los objetos anteriormente descritos y representan a los tres tipos de estadística: *StatisticsFeedsVO*, *StatisticsTagsVO* y *StatisticsUserVO*. Para más detalle ver el diagrama de clases del anexo A5

En el lado del cliente se utilizó la librería HighCharts para dibujar los gráficos apropiados: barra y torta. Una funcionalidad destacable de esta librería, es la opción de descargar los gráficos en distintos formatos: png, jpg, svg, entre otros.

Para más detalle de las vistas del mapa geográfico, mapa conceptual y estadística revisar las capturas de la aplicación que se encuentran en el anexo A6

# 6 EVALUACIÓN DE FEEDBACK

## 6.1 Caso de prueba

Con el propósito de evaluar la aplicación en un contexto real, se decidió hacer una prueba para medir desempeño y usabilidad. Además recibir retroalimentación del funcionamiento en general para comprobar si se cumplen los objetivos planteados al principio.

La aplicación se puso a prueba en el condominio Santa Carolina, ubicado en Lucas Pacheco 0140, Talagante. Lo componen 32 casas distribuidas en 20 hectáreas de terreno. Dicho condominio experimentaba un proceso de elección de proyectos que abordarían el año 2015, tales como: mejoras en la luminaria, arreglo de caminos, instalación de caseta de seguridad y reglamento de uso del espacio público. El objetivo era apoyar el proceso de elección de los proyectos por parte de los vecinos. Al finalizar el periodo de prueba se les solicitó completar una encuesta de usabilidad de la aplicación.

Inicialmente se les solicitó a los administradores del condominio identificar los proyectos con su título, descripción y ubicación. Además identificaron los temas relevantes para la comunidad. A continuación se presenta un resumen de los resultados de la primera etapa.

Proyectos: Caseta de seguridad, Ronda de guardias privados, Pavimentar camino principal, Arreglar lomos de toro, Colocar señalética del tránsito, Cambiar focos de luminaria a luces led, Quincho, Sala de juego, Oficina de administración, Barreras de entrada, Registro de visitas, Construcción de enfermería, Instalación de buzón de correspondencia en la entrada, Basureros de reciclaje, Basurero común, Plaza en el sector las orquídeas, Instalación de máquinas de ejercicio al aire libre, Ensanchar canales de regadío, Proveer agua potable.

Temas: Seguridad, Camino, Iluminación, Recreación, Administración, Acceso, Accidente, Correspondencia, Basura, Deporte, Agua

## 6.2 Procedimiento

La segunda fase comenzó con la presentación de la aplicación y su funcionamiento a los habitantes del condominio. Posteriormente se definió el periodo de prueba desde el 24 al 30 de Noviembre de 2014. Se preparó un ambiente ad-hoc para el despliegue, instalando el software y datos necesarios para el funcionamiento. Se crearon los feeds y tags definidos en la fase anterior.

A las 10:00 horas del día lunes 24 de noviembre se habilitó Feedback para ser ocupado por los vecinos. Las edades de los usuarios registrados estaban entre los 18 y 47 años. Cada uno de los proyectos de la directiva fueron precargados a la aplicación, con sus respectivos tags. Durante una semana, los usuarios podían comentar sobre los feeds (proyectos), votar y visualizarlos en sus tres formas (línea de tiempo, mapa geográfico y mapa conceptual). Además podían crear nuevos feeds y asociarlo a los tags ya creados o tags nuevos. En el anexo A6 se muestran capturas de interacción del usuario con la aplicación. La prueba finalizó a las 22 horas del día domingo 30 de noviembre.

## 6.3 Resultados

A continuación se muestra una tabla resumen de los resultados obtenidos:

| Variable | Cantidad |
|---|---|
| Total de feeds | 23 |
| Feeds con geolocalización | 15 |
| Feeds con tags | 19 |
| Total de comentarios | 55 |
| Total de votos | 48 |
| Votos positivos | 31 |
| Votos negativos | 17 |
| Total de tags | 24 |
| Total de usuarios | 24 |
| Usuarios con cuenta en Facebook o Twitter | 20 |

**Tabla 4: Resultados del periodo de prueba**

Se logró la participación del 75% de los vecinos del condominio creándose cuentas en la aplicación. De los cuales un 84% las asoció con su cuenta de Facebook y/o Twitter. A parte de los feeds creados con los proyectos de la directiva, se crearon 8 más proveniente de los usuarios. Se registraron 55 comentarios, lo que arroja un promedio de 3 comentarios por feed. En total se recibieron 48 votos, de los cuales un 64% corresponde a votos positivos. Además de los 15 tags creados por la directiva, los usuarios crearon 9 más.

## 6.4 Encuesta de usabilidad

Al finalizar el periodo de prueba, se procedió a entregar la encuesta de usabilidad. Dicha encuesta se confeccionó en base a la utilizada en un programa de educación en España[7] con adaptaciones para Feedback. Para mayor detalle de la encuesta revisar el anexo A7. Los resultados más relevantes se presentan a continuación:

1. Estructura de la aplicación: La organización estructural, la consistencia de la estructura y la densidad estructural fueron bien evaluadas, predominando la opción de acuerdo y muy de acuerdo.
2. Operación de la aplicación: La mayoría de estos aspectos fueron bien evaluados, predominando la opción de acuerdo. Sin embargo Interactividad tuvo una mala evaluación, donde destacó la opción en desacuerdo.
3. Información al usuario: el sistema de ayuda y retroalimentación tuvieron una evaluación regular, predominando de acuerdo y en desacuerdo. La búsqueda de información tuvo muy buena evaluación, destacando la opción muy de acuerdo.
4. Apariencia: tuvo buena evaluación, concentrando los mayores votos la opción de acuerdo
5. Intuición: también tuvo una buena evaluación, predominando la opción de acuerdo.
6. Experiencia del usuario: la mayoría calificó estar de acuerdo con una experiencia agradable al utilizar la aplicación.

[7] http://www.tdx.cat/bitstream/handle/10803/6542/12ApendiceA.pdf;jsessionid=3E9507DC7C301043262B029A8D616D84.tdx1?sequence=11

Una de las preguntas finales de la encuesta  hace referencia a la utilidad de la herramienta para obtener información de los proyectos del condominio. La mayoría de los comentarios respondieron de forma positiva, destacando que se podían ver los proyectos físicamente donde corresponde.

Un aspecto a mejorar, mencionado en los comentarios negativos, hace referencia al manejo de la cuenta del usuario en la aplicación. Algunos reportaron la imposibilidad de recuperar la contraseña y edición del perfil.

# 7 CONCLUSIONES

Feedback nació con el propósito de fomentar el intercambio de información entre y para las personas. Y a su vez disminuir las brechas entre el plano colectivo y el institucional a cargo de tomar decisiones. Feedback es una retroalimentación constante que busca generar una mejor calidad de vida para los habitantes de la ciudad.

El trabajo realizado en esta memoria consistió en desarrollar una herramienta que apoye la participación ciudadana. En particular se implementó una aplicación web que permite ingresar eventos de la ciudad y asociarlos a un lugar geográfico y a temas urbanos. A su vez se pueden observar estadística relevante de los temas tratados.

En la fase de diseño de la aplicación se consideraron aspectos evolutivos del software, tales como escalabilidad, reutilización, desempeño y usabilidad. Con la ayuda del framework Java Spring, el desarrollo resultó ser un proceso armónico y amigable, ya que entrega herramientas útiles para resolver problemas recurrentes.

Tanto el objetivo principal como los secundarios de esta memoria se cumplieron. La aplicación resultó ser un apoyo para el condominio Santa Carolina al momento de elegir sus proyectos comunitarios. Les entregó herramientas de votación, discusión, y visualización de la información en distintos planos.

Sin embargo esto no significa el término del proyecto. Hay mucho que mejorar, desde aspectos de usabilidad de la aplicación reflejados en el capítulo de evaluación, como en nuevos objetivos que enriquezcan el contenido de esta. Además, con el fin de validar los objetivos en un contexto más amplio, resulta necesario disponibilizar y masificar la aplicación para su uso.

Proveer un ecosistema de TI para una Smart City es una labor titánica que involucra a todas las personas. Con iniciativas como esta y tantas otras, se aporta un grano de arena en mejorar la calidad de vida de las personas en la ciudad.

# 8 TRABAJO FUTURO

## 8.1 Manejo de gran volumen de tags en el mapa conceptual

Una funcionalidad importante en la aplicación es la creación libre de tags. Si bien se provee una lista finita de tags que se muestran en el mapa conceptual, es necesaria una optimización a la hora de mostrar los tags. Una opción es considerar los tags más asociados. Para ello, la aplicación deberá guardar estadística y actualizarse dinámicamente para colocar o sacar tags de la lista de los más asociados. Una herramienta útil para este propósito es Hadoop y Hive, que permiten procesar altos volúmenes de datos alojados en Cassandra mediante operaciones en bash.

Además de limitar la visualización de los tags, también se deben limitar la visualización de los feeds asociados al tag. Una opción para esto es generar un único nodo hoja en el caso que se exceda un límite de feeds asociados a dicho tag, y para ver el detalle se inspecciona este nuevo nodo hoja.

## 8.2 Filtro por fecha

Así como existe un filtro por texto, es deseable crear un filtro por fechas. El usuario ingresa fecha de inicio y término y arroja la lista de feeds que coinciden con la búsqueda. Además se deberá modificar tanto el mapa como el mapa conceptual. Para implementar esto se deberá considerar un histórico de las asociaciones de los feeds a los tags ya que es deseable mostrar las distintas asociaciones de un determinado tema a lo largo del tiempo.

## 8.3 Mejoras en la estrategia de búsqueda

La búsqueda implementada lo hace sobre el título, descripción y comentarios de un feed, además de los tags. Una mejora consiste en ir adaptando el Analizer de los objetos, por ejemplo agregando filtros y tokenizers. El tuning de los índices y meta data de las tablas de la base de datos dependerá del uso. Se propone hacer uso intensivo de la aplicación durante un mes, y luego ajustar los parámetros para optimizar la búsqueda.

## 8.4 Clientes Twitter y Facebook

Una de los desafíos más importantes para el futuro desarrollo de la aplicación es la de generar clientes de redes sociales que rescaten información y la publiquen en Feedback. A modo de prueba se realizaron streaming desde Twitter con algunos tags de la aplicación. Luego de una hora de recibir tweets localizados en la región metropolitana, el volumen de información resultó ser inmanejable y escapaba del propósito de esta memoria, alcanzando los 200 MB de memoria en disco. En dos días un disco de 1 TB se llena completamente.

Es por ello que es necesario crear una estrategia para filtrar los mensajes que se reciben, manipular los archivos donde se almacenan y programar un hilo de ejecución que se encargue de guardarlos en la base de datos.

La herramienta utilizada para el streaming fue Spring XD, el cual contiene la API de Twitter para desarrollo. En el caso de Facebook no se encontró una API para estos propósitos. Una forma de abordar la recuperación de mensajes desde Facebook es a través de los mismos usuarios, y requiere una estrategia de consultas sobre cada uno de los usuarios registrados en Feedback y que hayan asociado su cuenta de Facebook.

# 9 BIBLIOGRAFÍA


[1] IBM, «Smarter Cities. Overview,» [En línea]. Available: http://www.ibm.com/smarterplanet/us/en/smarter_cities/overview.

[2] MIT, «MIT City Science, Smart Cities: Vision,» [En línea]. Available: http://smartcities.media.mit.edu/frameset.html.

[3] Gartner Inc, «Hype Cycle for Big Data,» 2012. [En línea]. Available: http://www.gartner.com/id=2100215.

[4] A. Joakar, «Open Gardens,» [En línea]. Available: http://www.opengardensblog.futuretext.com/wp-content/uploads/2012/08/Big-Data-for-Smart-cities-How-do-we-go-from-Open-Data-to-Big-Data-for-Smart-cities.pdf.

[5] S. Daniel y M. A. Doran, «geoSmartCity: geomatics contribution to the Smart City,» de *ACM 1-58113-000-0/00/0010*, Quebec, 2013.

[6] G. Zurita y N. Baloian, «Using Geocollaboration and Microblogging to Support Learning: Identifying problems and Opportunities for Technological Business,» de *Springer-Verlag Berlin Heidelberg*, Berlin, 2013.

[7] C. Villalba, «Redes Sociales: Un concepto con importantes implicaciones en la intervención comunitaria.Intervencion Psicosocial.,» *Revista sobre igualdad y calidad de vida.,* vol. Vol. 2, pp. 8-9, 2003.

[8] INJUV, «7ma Encuesta Nacional de la Juventud,» 2012. [En línea]. Available: http://www.injuv.gob.cl/portal/wp-content/files_mf/septimaencuestanacionainjuvcorr2.pdf.

[9] P. B. Brandtzaeg y J. Heim, «Why People Use Social Networking Sites.,» *Springer, Heidelberg,* vol. LNCS vol. 5621, 2009.

[10] ComScore, «El crecimiento de las Redes Sociales en América Latina,» 2011.

[11] A. Java, X. Song, T. Finin y B. Tseng, «Why We Twitter: Understanding

Microblogging Usage and Communities. In: Zhang, H., Spiliopolou, M., Mobasher, B., et al.,» *Springer, Heidelberg,* vol. LNCS vol. 5439., pp. 118-138, 2007.

[12] A. N. Joinson, «'Looking At', 'Looking Up' or 'Keeping Up' With People? Motives and Uses of Facebook.,» de *Proceedings of the 26th SIGCHI Conference on Human Factors in Computing Systems*, Florence, Italy , 2008.

[13] IAB, «Interactive Advertising Bureau,» 2010. [En línea]. Available: http://www.iab.cl/usuarios-chilenos-en-foursquare/.

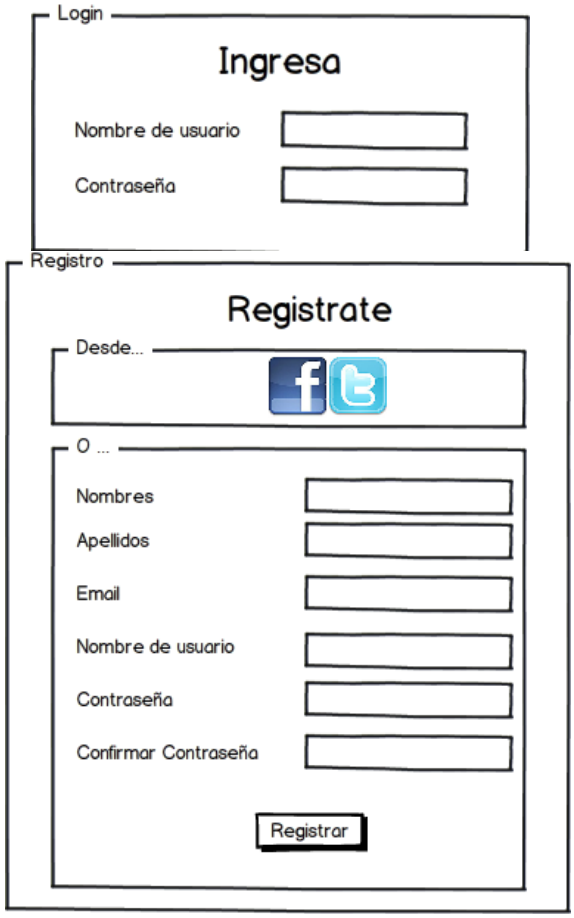
[14] G. Zurita, N. Baloian y F. Gutierrez, «Multiple Views for Supporting Lifelong, Highly Contextual and Ubiquitous Social Learning,» Santiago, 2013.

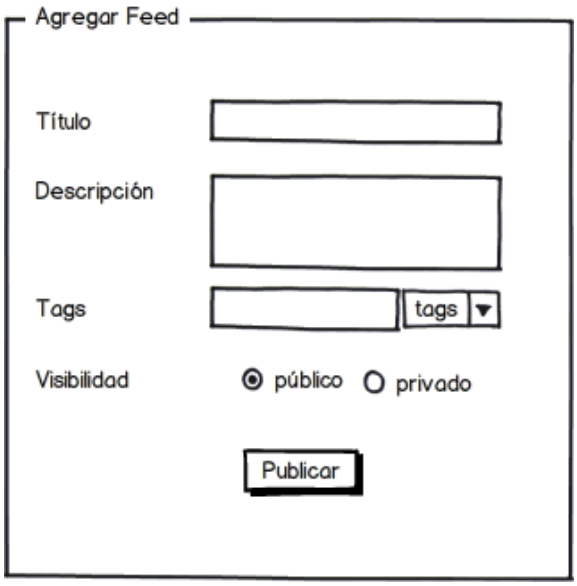
[15] D. Aguirre, «Desarrollo de una herramienta colaborativa para el levantamiento de procesos BPMN en dispositivos móviles,» Santiago, Chile, 2012.

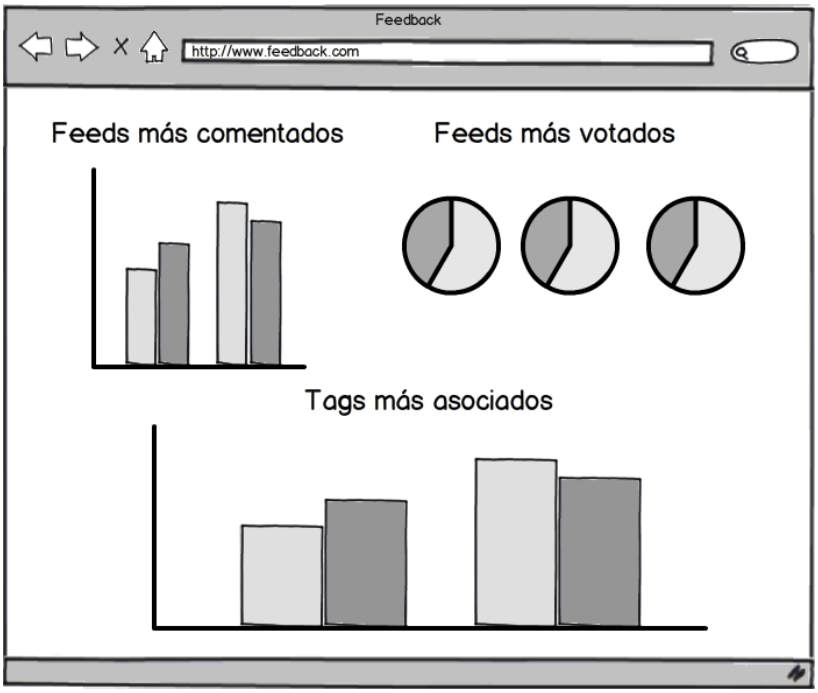
[16] P. Mechant, L. De Marez y L. Claeys, «Crowdsourcing for smart engagement apps in an urban context: an exporative study,» Guent, Belgium, 2011.

# 10 ANEXO

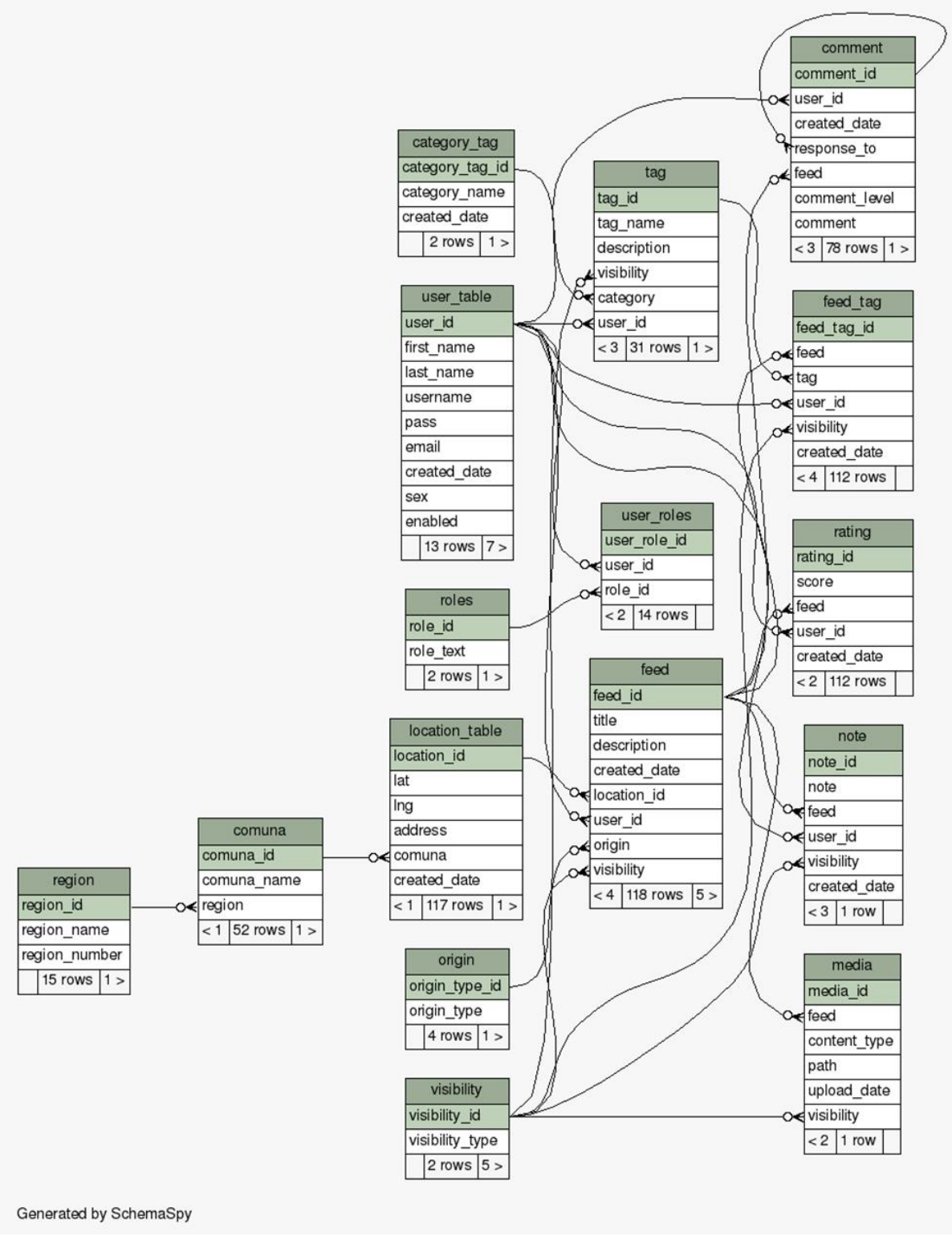
## A1 Mockups de Feedback



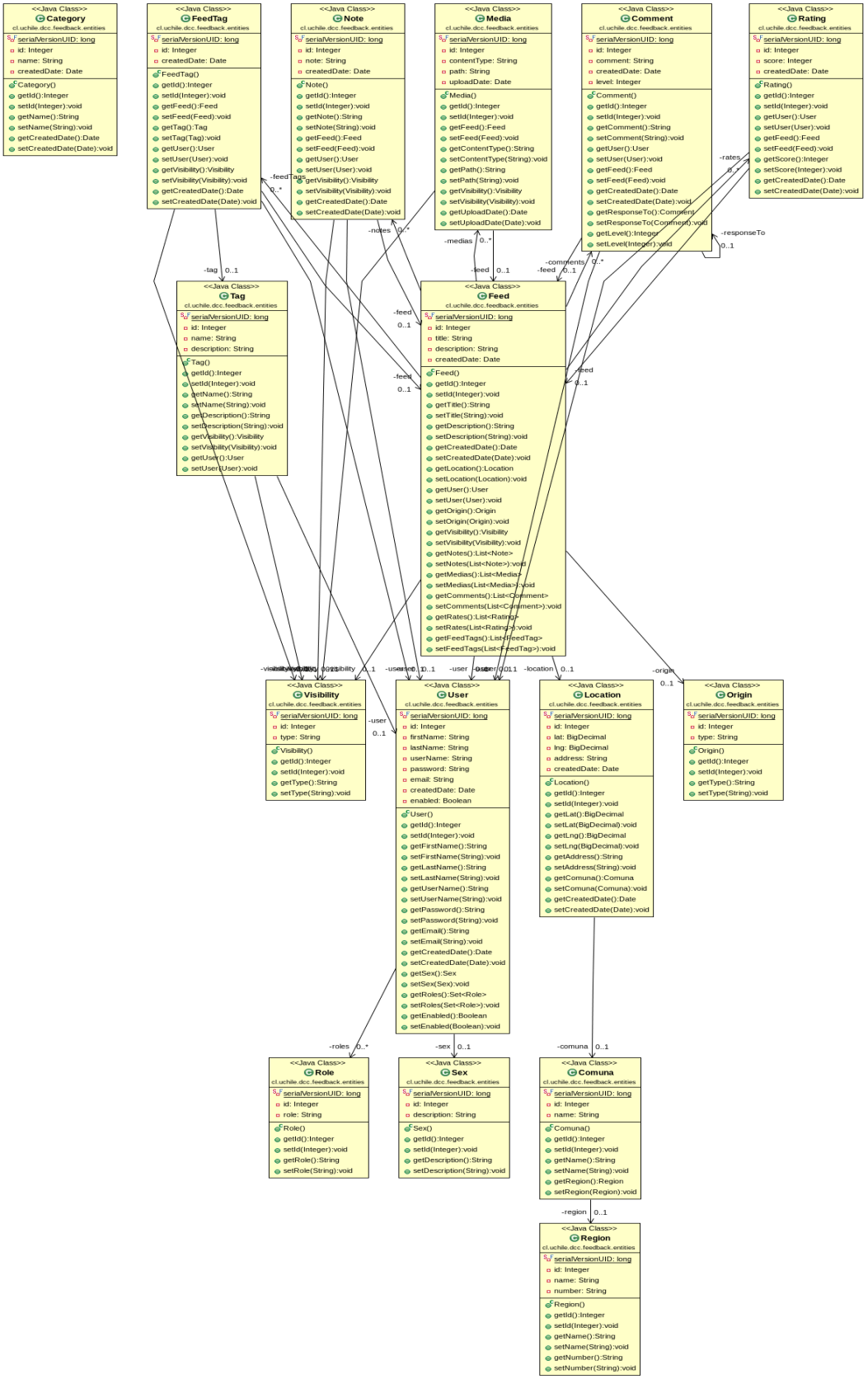



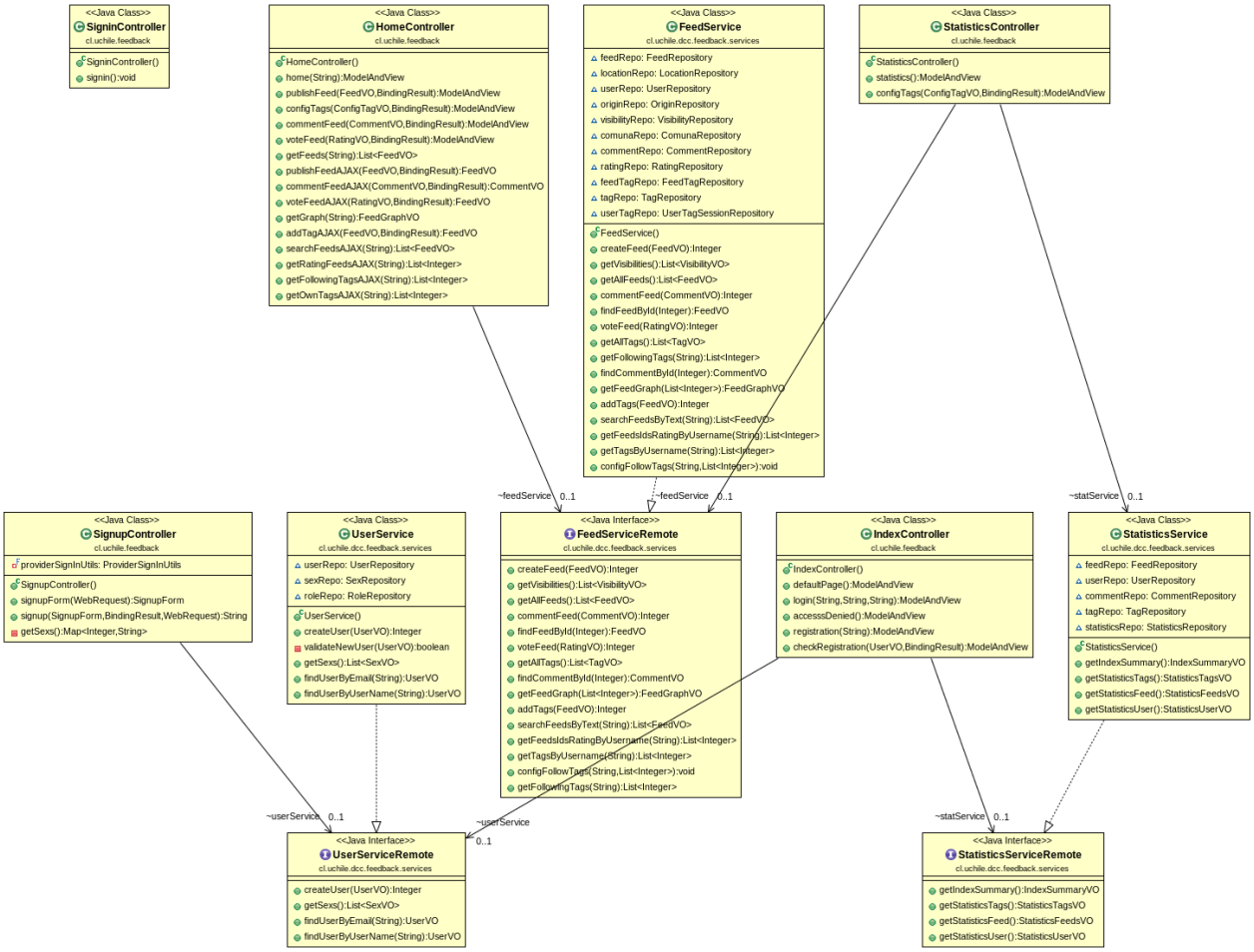
## A2 Modelo Entidad Relación



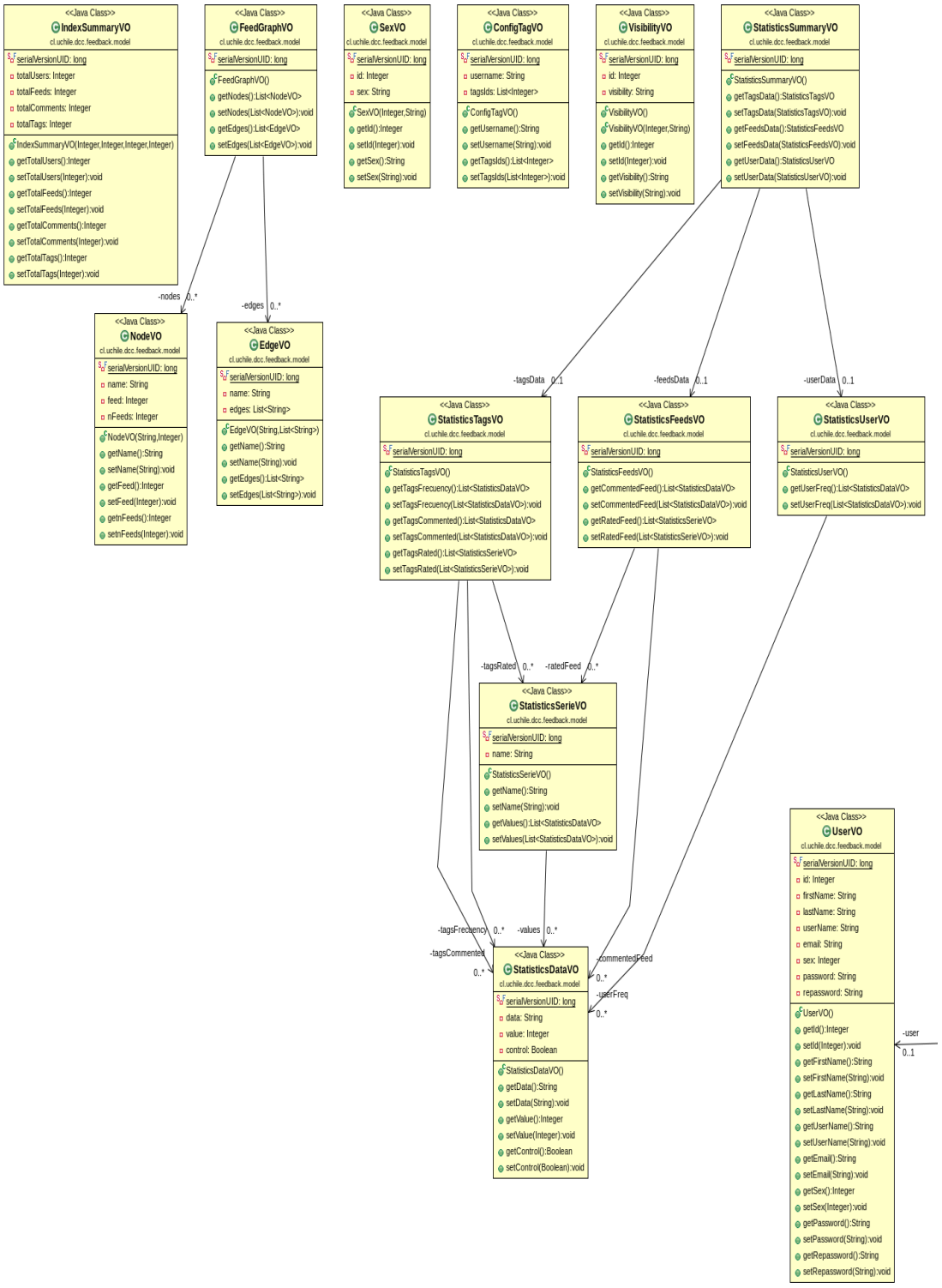
# A3 Diagrama de clases de las entidades

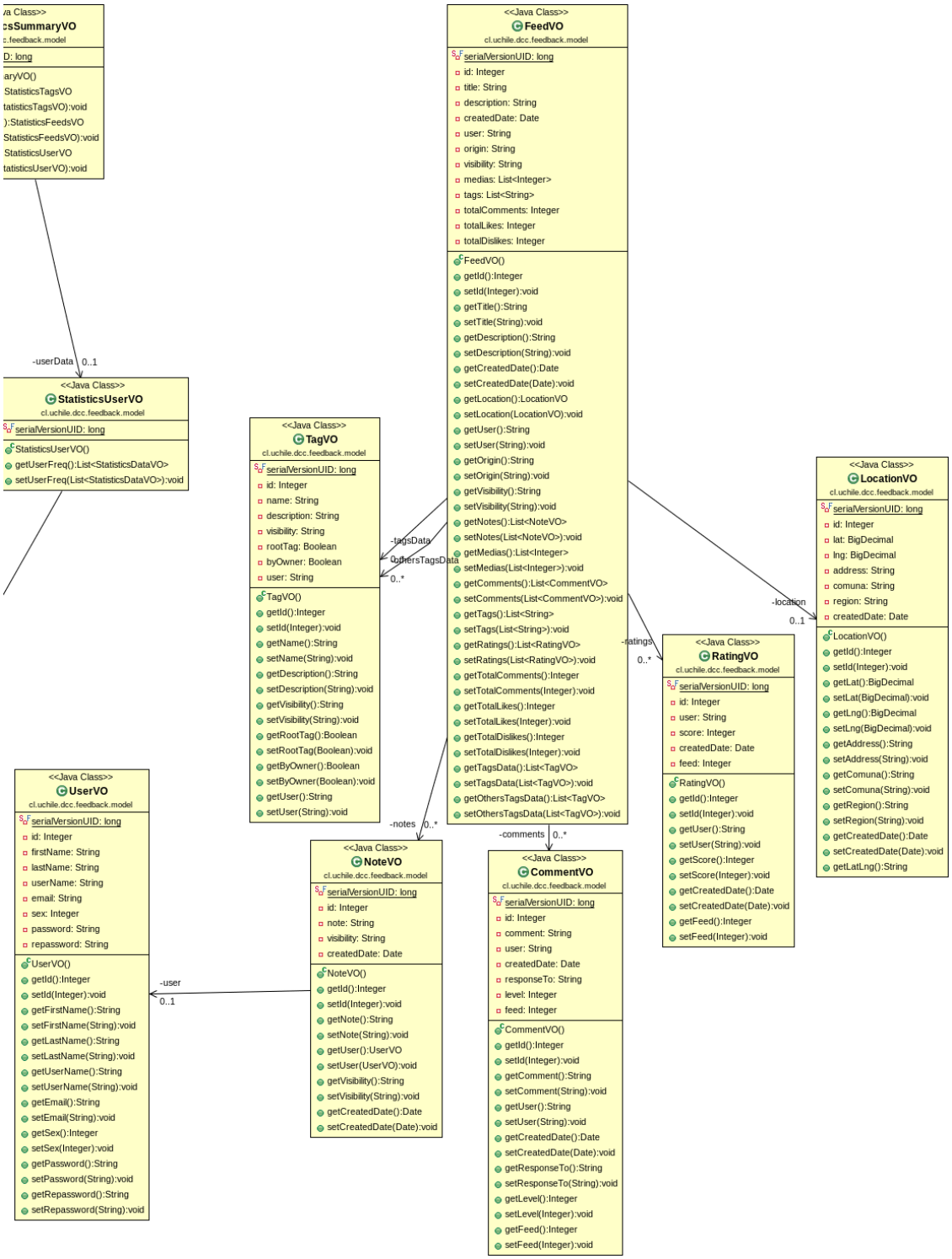


# A4 Diagrama de clases de servicios y controladores



# A5 Diagrama de clases de objetos VOs



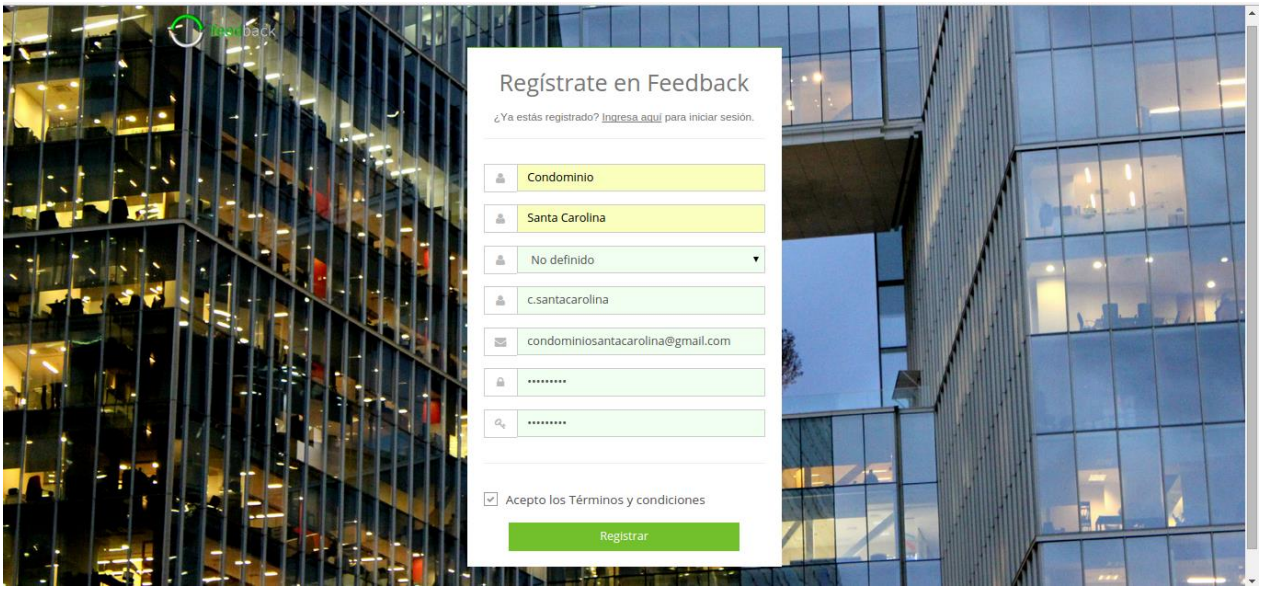

-userData
0..1
-tagsData
-othersTagsData
0..*
-location
0..1
-ratings
0..*
-notes
0..*
-comments
0..*
-user
0..1
<<Java Class>>
StatisticsUserVO
cl.uchile.dcc.feedback.model
serialVersionUID: long
StatisticsUserVO()
getUserFreq():List<StatisticsDataVO>
setUserFreq(List<StatisticsDataVO>):void
<<Java Class>>
TagVO
cl.uchile.dcc.feedback.model
serialVersionUID: long
id: Integer
name: String
description: String
visibility: String
rootTag: Boolean
byOwner: Boolean
user: String
TagVO()
getId():Integer
setId(Integer):void
getName():String
setName(String):void
getDescription():String
setDescription(String):void
getVisibility():String
setVisibility(String):void
getRootTag():Boolean
setRootTag(Boolean):void
getByOwner():Boolean
setByOwner(Boolean):void
getUser():String
setUser(String):void
<<Java Class>>
FeedVO
cl.uchile.dcc.feedback.model
serialVersionUID: long
id: Integer
title: String
description: String
createdDate: Date
user: String
origin: String
visibility: String
medias: List<Integer>
tags: List<String>
totalComments: Integer
totalLikes: Integer
totalDislikes: Integer
FeedVO()
getId():Integer
setId(Integer):void
getTitle():String
setTitle(String):void
getDescription():String
setDescription(String):void
getCreatedDate():Date
setCreatedDate(Date):void
getLocation():LocationVO
setLocation(LocationVO):void
getUser():String
setUser(String):void
getOrigin():String
setOrigin(String):void
getVisibility():String
setVisibility(String):void
getNotes():List<NoteVO>
setNotes(List<NoteVO>):void
getMedias():List<Integer>
setMedias(List<Integer>):void
getComments():List<CommentVO>
setComments(List<CommentVO>):void
getTags():List<String>
setTags(List<String>):void
getRatings():List<RatingVO>
setRatings(List<RatingVO>):void
getTotalComments():Integer
setTotalComments(Integer):void
getTotalLikes():Integer
setTotalLikes(Integer):void
getTotalDislikes():Integer
setTotalDislikes(Integer):void
getTagsData():List<TagVO>
setTagsData(List<TagVO>):void
getOthersTagsData():List<TagVO>
setOthersTagsData(List<TagVO>):void
<<Java Class>>
LocationVO
cl.uchile.dcc.feedback.model
serialVersionUID: long
id: Integer
lat: BigDecimal
lng: BigDecimal
address: String
comuna: String
region: String
createdDate: Date
LocationVO()
getId():Integer
setId(Integer):void
getLat():BigDecimal
setLat(BigDecimal):void
getLng():BigDecimal
setLng(BigDecimal):void
getAddress():String
setAddress(String):void
getComuna():String
setComuna(String):void
getRegion():String
setRegion(String):void
getCreatedDate():Date
setCreatedDate(Date):void
getLatLng():String
<<Java Class>>
RatingVO
cl.uchile.dcc.feedback.model
serialVersionUID: long
id: Integer
user: String
score: Integer
createdDate: Date
feed: Integer
RatingVO()
getId():Integer
setId(Integer):void
getUser():String
setUser(String):void
getScore():Integer
setScore(Integer):void
getCreatedDate():Date
setCreatedDate(Date):void
getFeed():Integer
setFeed(Integer):void
<<Java Class>>
UserVO
cl.uchile.dcc.feedback.model
serialVersionUID: long
id: Integer
firstName: String
lastName: String
userName: String
email: String
sex: Integer
password: String
repassword: String
UserVO()
getId():Integer
setId(Integer):void
getFirstName():String
setFirstName(String):void
getLastName():String
setLastName(String):void
getUserName():String
setUserName(String):void
getEmail():String
setEmail(String):void
getSex():Integer
setSex(Integer):void
getPassword():String
setPassword(String):void
getRepassword():String
setRepassword(String):void
<<Java Class>>
NoteVO
cl.uchile.dcc.feedback.model
serialVersionUID: long
id: Integer
note: String
visibility: String
createdDate: Date
NoteVO()
getId():Integer
setId(Integer):void
getNote():String
setNote(String):void
getUser():UserVO
setUser(UserVO):void
getVisibility():String
setVisibility(String):void
getCreatedDate():Date
setCreatedDate(Date):void
<<Java Class>>
CommentVO
cl.uchile.dcc.feedback.model
serialVersionUID: long
id: Integer
comment: String
user: String
createdDate: Date
responseTo: String
level: Integer
feed: Integer
CommentVO()
getId():Integer
setId(Integer):void
getComment():String
setComment(String):void
getUser():String
setUser(String):void
getCreatedDate():Date
setCreatedDate(Date):void
getResponseTo():String
setResponseTo(String):void
getLevel():Integer
setLevel(Integer):void
getFeed():Integer
setFeed(Integer):void

## A6 Implementación de interfaces

A continuación se muestran algunas capturas de Feedback en base al caso de uso mencionado en el capítulo 6.

En primer lugar los usuarios del condominio se crearon una cuenta y procedieron a ingresar a la aplicación:

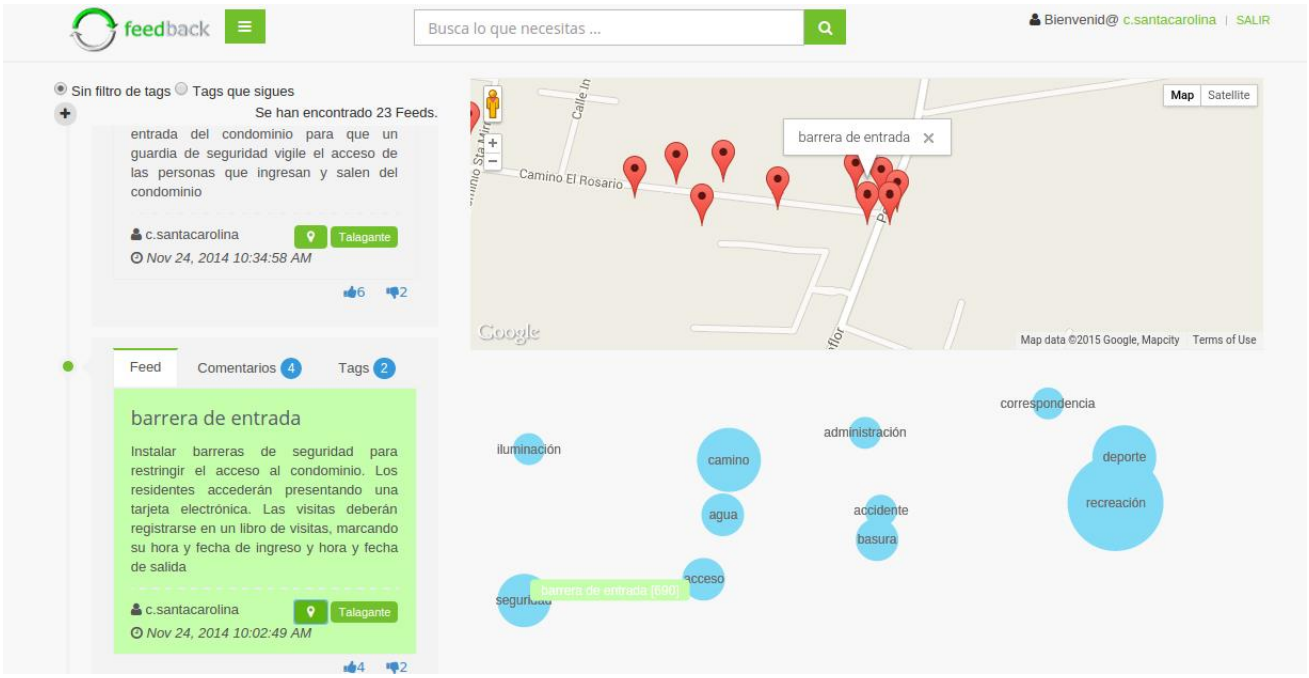
### Registro de usuarios




### Página de acceso a la aplicación

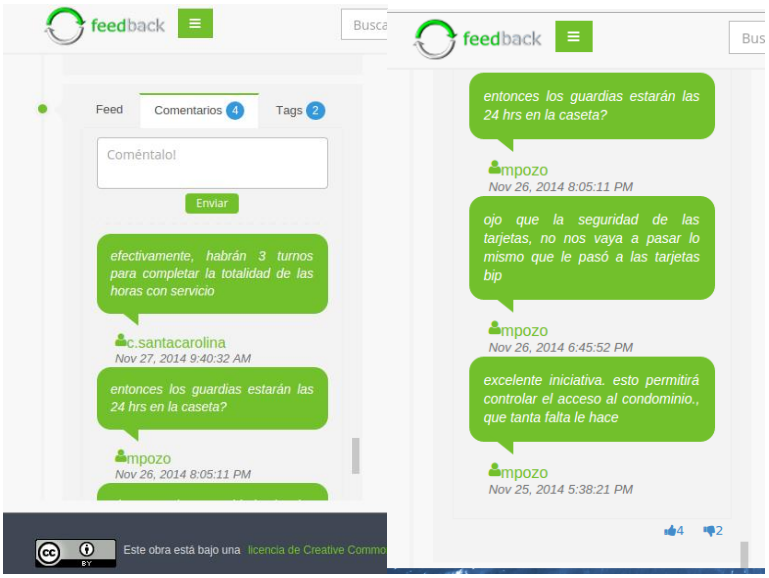
Luego ingresan al inicio de la aplicación donde se muestra en el costado izquierdo una lista de feeds, con su respectiva información, comentarios, tags y votos. En el costado derecho se muestra el mapa geográfico y el mapa conceptual de los feeds mostrados en la lista.

### Página de inicio

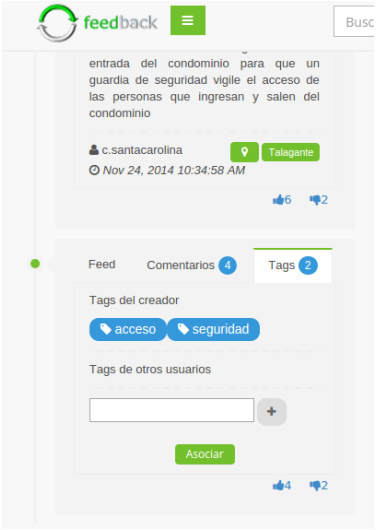



En el caso de prueba se ingresaron proyectos del condominio, que se ejemplifica con el proyecto "*barrera de entrada*", su descripción, entre otros. Notar que fue asociado a los temas "*seguridad*" y "*acceso*". También los usuarios hicieron comentarios del feed.

## Comentarios del feed “barrera de entrada”




## Tags del feed “barrera de entrada”

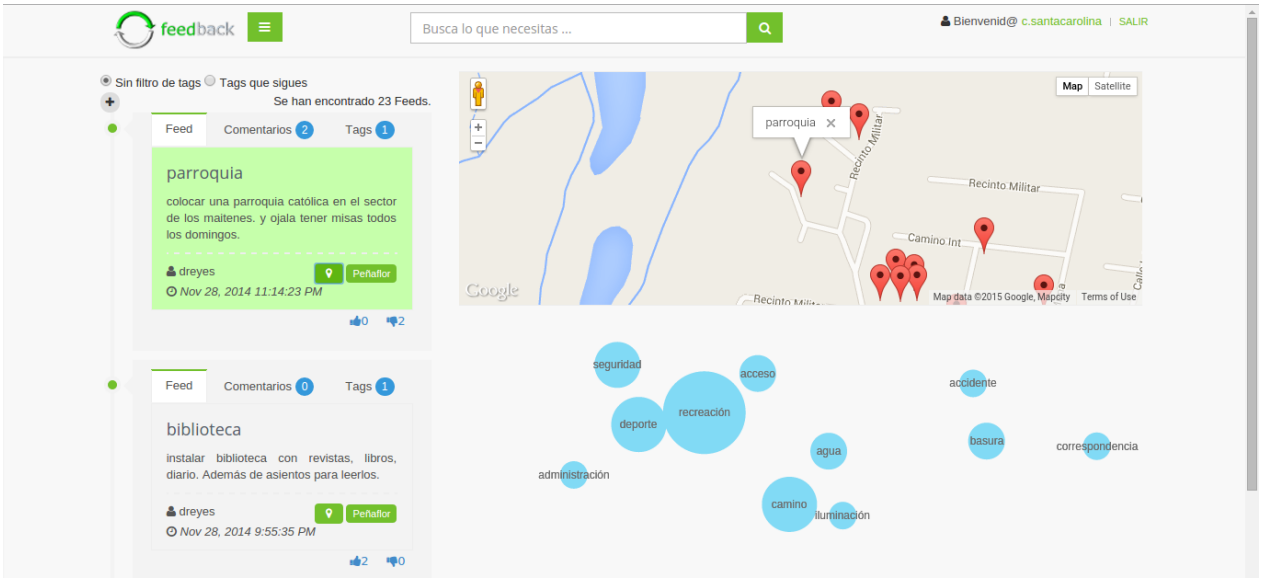



Así como la directiva ingresó la mayoría de los feeds al sistema, también los usuarios podían ingresar iniciativas personales a desarrollar en el 2015. Tan solo debían completar el formulario del feed, indicar la ubicación geográfica y los tags respectivos.

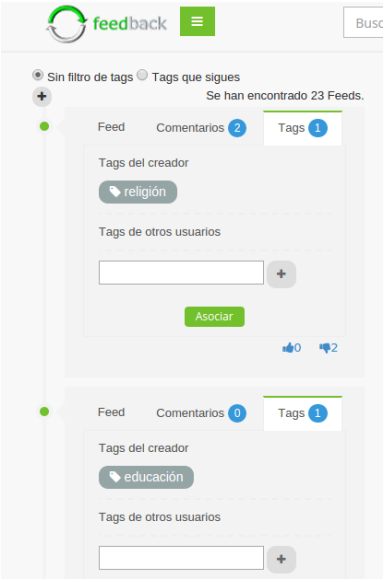
## Formulario de registro de nuevo feed


Un ejemplo de esta situación, corresponde a los proyectos registrados por el usuario "*dreyes*" con nombres "*parroquia*" y "*biblioteca*".

## Feeds creados por un usuario de la aplicación



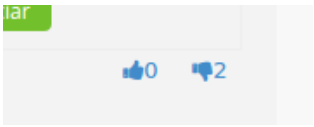
### Tags creados por los usuarios



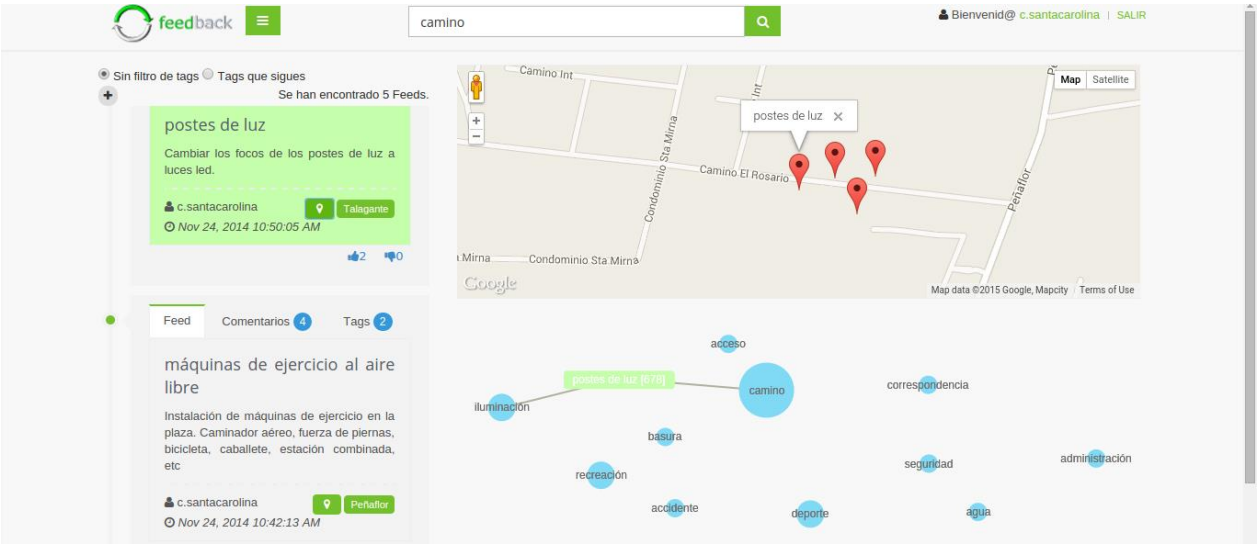

Notar que los tags creados por los administradores del condominio son de color azul y los registrados por los usuarios son grises. Otra funcionalidad importante es la asociación de un feed posterior a su creación por parte de cualquier usuario de la aplicación.

También se provee un sistema de votación por cada feed registrado en el sistema y búsqueda por texto libre.

### Funcionalidad de votación



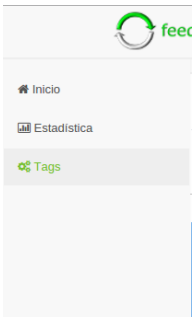
## Búsqueda en feedback




Tanto la lista de feeds, como el mapa geográfico y el mapa conceptual se adaptan a los resultados de la búsqueda.

En el caso de uso, los administradores del condominio utilizaron la funcionalidad de seguimiento de tags, de esta forma podían filtrar solo los proyectos ingresados por ellos. A continuación se presenta algunas capturas de la interfaz.

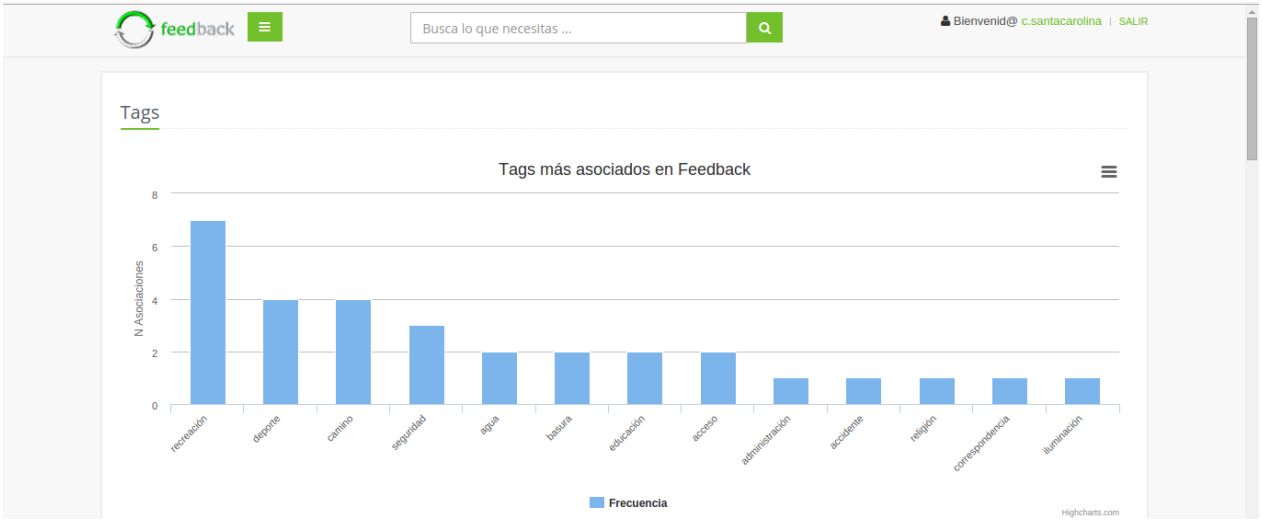
## Seguimiento de tags





Finalmente se presentaron los resultados en una vista dedicada a la estadística. Se presentan los tags más asociados, los feeds más comentados, los tags más votados y los usuarios que más feeds ingresaron.

## Estadística






feedback
Busca lo que necesitas ...
Bienvenid@ c.santacarolina | SALIR
Highcharts.com
Highcharts.com
Highcharts.com
Usuarios
Usuarios con más feeds
c.santacarolina
18
dreyes
2
fpolloni
1
warancibia
1
mpozo
1
Este obra está bajo una licencia de Creative Commons Reconocimiento 4.0 Internacional . Términos de uso
feedback

## A7 Encuesta de usabilidad

Nombre:

Usuario:

Fecha:

# ENCUESTA DE USABILIDAD DE FEEDBACK

Marque con una X las casillas de la izquierda con los números según corresponda:

1:Muy en desacuerdo 2: En desacuerdo 3: De acuerdo 4: Muy de acuerdo

| | | | | | |
|---|---|---|---|---|---|
| 1 | Estructura de la aplicación | | | | |
| 1.1 | Organización estructural: La distribución de los elementos estructurales de la aplicación (barras de desplazamientos, zonas de selección, botones, etc.) es buena | 1 | 2 | 3 | 4 |
| 1.2 | Densidad estructural: La cantidad de elementos estructurales que se utilizan en la aplicación es excesiva | | | | |
| 1.3 | Consistencia de la estructura: La distribución de los elementos estructurales se mantiene a lo largo de la aplicación | | | | |
| 2 | Operación de la aplicación | | | | |
| 2.1 | Navegabilidad: El recorrido que se hace por el contenido de la aplicación es fácil. | | | | |
| 2.2 | Interactividad: La relación mutua entre el usuario y la aplicación es buena | | | | |
| 2.3 | Accesibilidad: Las acciones que solicita la aplicación son fáciles de ejecutar | | | | |
| 2.4 | Sistema de indicación: Se identifican fácilmente las figuras, tablas, los hipertextos, las zonas activas y el tipo de acción que se debe ejecutar | | | | |
| 2.5 | Fiabilidad del sistema: Hay demasiados errores durante la operación | | | | |
| 2.6 | Consistencia de la operación: La ejecución de tareas(navegar por la aplicación, hacer clic en botones, seleccionar opciones, etc.) sigue un estándar a lo largo de la aplicación | | | | |
| 2.7 | Desempeño del sistema: La velocidad de funcionamiento de la aplicación, considerando el tipo de tarea que se exige, es buena | | | | |

| | | | | | |
|---|---|---|---|---|---|
| 3 | Información al usuario | | | | |
| 3.1 | Sistema de ayuda: Las dudas del usuario se resuelven fácilmente | | | | |
| 3.2 | Retroalimentación: La aplicación mantiene al usuario informado sobre las tareas en ejecución | | | | |
| 3.3 | Búsqueda de información: Los datos que busca el usuario son fáciles de encontrar | | | | |
| 4 | Apariencia: La presentación del contenido (tipo y tamaño de la fuente, uso de color, disposición de los elementos según significado, etc.) es buena | | | | |
| 5 | Intuición: Los procedimientos de navegación por la aplicación o ejecución de tareas asignadas se aprende de forma prácticamente inmediata | | | | |
| 6 | Experiencia del usuario: La interacción con la aplicación resulto ser agradable | | | | |
| | | | | | |
| ¿Ayudó la aplicación a mostrar la información de forma ordenada y completa, facilitando la toma de decisión? | | | | | |
| | | | | | |
| Aspectos Positivos de la aplicación | | | | | |
| | | | | | |
| Aspectos Negativos de la aplicación | | | | | |
| | | | | | |