

1	Introducción .....	2
2	Motivación y objetivos.....	4
2.1	Motivación.....	4
2.2	Objetivos.....	4
3	Estado del arte .....	5
3.1	Servicios RESTful.....	5
3.2	REST en Glassfish .....	7
3.3	Seguridad en Glassfish.....	8
3.4	Autenticación .....	9
3.5	CORS .....	10
3.6	Ajax y jQuery.....	12
3.7	Datos de sesión en el cliente javascript .....	12
4	Especificación .....	12
4.1	Análisis de requisitos .....	12
4.2	Especificación del sistema .....	13
4.3	Planificación y estimación de costes .....	13
5	Desarrollo del proyecto.....	13
5.1	Análisis.....	13
5.1.1	Descripción de Sparrow .....	13
5.1.2	Casos de uso.....	16
5.1.3	Diagramas de clases .....	18
5.1.4	Base de datos .....	25
5.2	Diseño.....	26
5.3	Implementación .....	27
5.3.1	Acceso a la base de datos .....	27
5.3.2	Entidades.....	27
5.3.3	Data Acces Object (DAO) .....	30
5.3.4	Business Objects (BO) .....	32
5.3.5	Seguridad .....	32
5.3.6	Servicios .....	34
5.3.6.1	JSON.....	35
5.3.6.2	ServicioUsers .....	36
5.3.6.3	ServicioChips.....	37
5.3.6.4	Otras funciones del servicio .....	38
5.3.6.5	Definiendo la seguridad .....	38

5.3.6.6	CORS .....	39
5.3.7	Cliente Java .....	39
5.3.7.1	Seguridad y Login .....	39
5.3.7.2	Casos de uso .....	40
5.3.7.3	Entidades .....	44
5.3.7.4	FiltroHeader .....	44
5.3.7.5	Diseño de la web .....	44
5.3.8	Cliente HTML .....	48
5.3.8.1	Ajax .....	49
5.3.8.2	datos.js .....	50
5.3.8.3	Volviendo de Ajax .....	52
5.3.8.4	Seguridad y Login .....	52
5.3.8.5	Casos de uso .....	53
5.3.8.6	Diseño de la web .....	55
6	Pruebas y resultados .....	62
6.1	Descripción de experimentos .....	62
6.2	Resultados y discusión .....	63
6.2.1	Pruebas del servicio RESTful .....	63
6.2.1.1	Acceso a zona pública .....	63
6.2.1.2	Acceso a zona privada .....	63
6.2.1.3	Cliente Java .....	65
6.2.1.4	Cliente HTML .....	69
6.3	Evaluación presupuestaria .....	74
7	Conclusiones y trabajo futuro .....	74
8	Referencias .....	75

# Servicios RESTFUL en J2EE

## 1 INTRODUCCIÓN

---

Java EE proporciona un modelo de aplicaciones por capas que divide la lógica en componentes distribuidos que pueden estar funcionando en diferentes máquinas. Así tenemos una parte de cliente ejecutándose en la máquina del usuario, páginas web y EJB's en el servidor Java EE y bases de datos en el servidor de estas (ilustración 1).

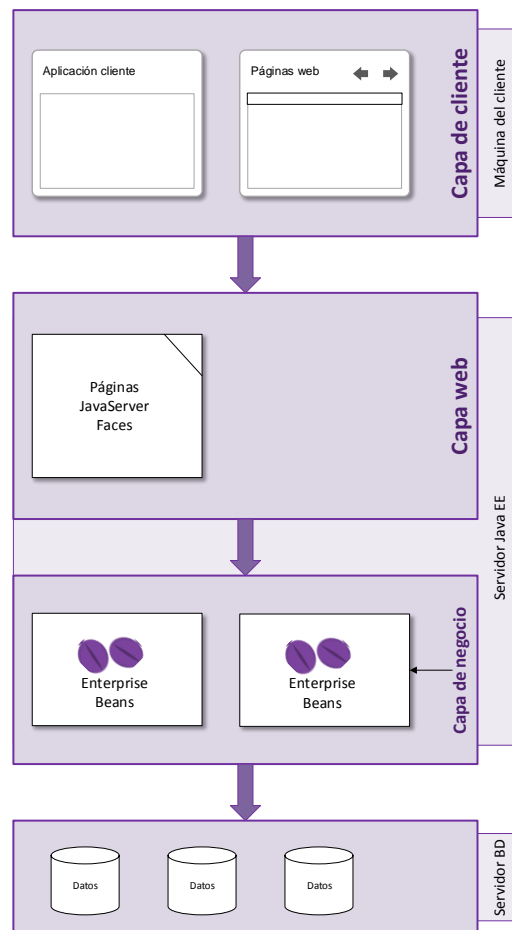


Ilustración 1

Los componentes distribuidos son los EJB. De esta forma el programador solo tiene que centrarse en la lógica de su programa dejando de lado toda la parte de control de la aplicación empresarial. En este caso se van a utilizar EJB de entidad que se encargan de mapear la información de la base de datos mediante objetos (ORM) de esta forma se puede acceder a los datos mediante clases Java sin tener que programar directamente la base de datos. También vamos a utilizar EJB de sesión que serán una fachada para los servicios proporcionados por nuestra aplicación y también son utilizados en algunos casos para guardar información del cliente.

Una aplicación J2EE típica estaría dentro de un proyecto EAR en el que se agruparán las entidades de la base de datos, los DAO para acceder a ellas y los BO que proporcionan la funcionalidad de esta junto al código del cliente que está compuesto por servlets y la vista que será un conjunto de páginas web en formato html o jsp (ilustración 2).

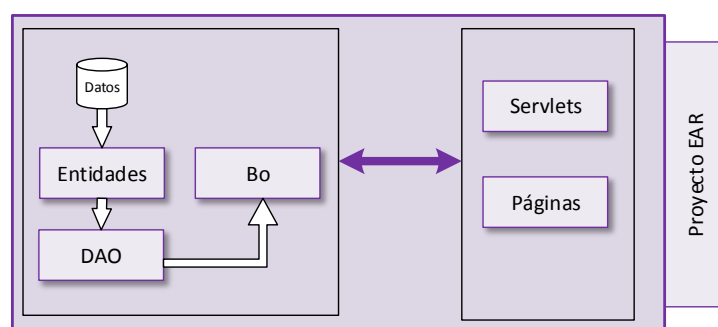


Ilustración 2

Como podemos ver es posible escribir todos los clientes que se quiera pero estos deben estar dentro de nuestra aplicación por lo que se restringe el acceso a las funcionalidades de la aplicación por parte de terceros y también por parte de otros sistemas ya que el cliente será obligatoriamente una página web o un programa Java puro.

En este proyecto se va a encontrar una forma de crear clientes diferentes que estén funcionando fuera de nuestra aplicación empresarial. Esto se hará convirtiendo la aplicación en un servicio RESTful.

Un servicio REST se puede explicar de forma sencilla como aquel que responde a llamadas http efectuadas mediante diferentes URI's devolviendo la información solicitada. Esto esencialmente es lo mismo que harán los BO dentro de nuestra aplicación empresarial al ser inyectados en el cliente.

Estos servicios se podrían comparar con el funcionamiento cliente-servidor de una página web estática en la que el cliente le manda una URL al servidor y este le responde con los datos que componen la página solicitada sin tener en cuenta la información de sesión del cliente. En un servicio REST hay más libertad ya que estas peticiones se pueden hacer enviando datos extra de diferentes formas como por ejemplo paso de parámetros, envío de datos en formato XML, JSON... y este nos puede responder también mediante diferentes formatos como texto plano, HTML, XML o JSON por ejemplo.

## 2 MOTIVACIÓN Y OBJETIVOS

---

### 2.1 MOTIVACIÓN

Se va a partir de una aplicación existente, Sparrow, que es una pequeña red social similar a Twitter escrita en J2EE y por lo tanto con un cliente que forma parte de la aplicación empresarial.

Teniendo ya escrito el código de entidades y lógica se va a adaptar para que se convierta en un servicio REST y así poder escribir clientes externos que puedan consumir sus recursos como otros servicios existentes en la actualidad.

Al convertir la aplicación será posible utilizarla de diferentes formas y en diferentes sistemas además de poder abrir esta a otros desarrolladores y llegando a una cantidad mayor de usuarios.

### 2.2 OBJETIVOS

El primer paso será la conversión a servicio creando una nueva aplicación web. Para el intercambio de datos se va a utilizar el formato JSON ya que se probará con un cliente en javascript y la lectura de datos en JSON es más fácil y rápida que en otros.

Sparrow es una aplicación que permite el acceso a usuarios invitados a ciertas secciones pero que requiere que el usuario esté registrado para poder utilizarla a fondo por lo que hay que establecer un sistema de seguridad que pueda controlar el acceso de los clientes.

En el momento en el que ya esté el servicio en funcionamiento se puede proceder a escribir la parte de cliente. Como objetivo se ha puesto comprobar la compatibilidad de este servicio con diferentes

tipos de cliente por lo que se va a escribir un cliente con Java que será una aplicación web externa a nuestro servicio y otro cliente que funcionará con HTML y javascript.

## 3 ESTADO DEL ARTE

---

### 3.1 SERVICIOS RESTFUL

¿Que es REST? El termino apareció en la disertación de Roy Fielding en el año 2000 (Sandoval, 2009) y viene de Representational State Transfer. REST no es una arquitectura en sí, es un conjunto de reglas que salen del *null space* que representa la viabilidad de cada tecnología y estilo de programación sin límites. Partiendo de esto las reglas que rigen un sistema REST son:

- Es un sistema cliente-servidor.
- Es sin estado. Las llamadas al servicio son independientes entre sí.
- Uniformemente accesible. Cada recurso tiene una dirección única.
- Va por capas y es escalable.
- Provee código si se le pide. Esto es optativo. Las aplicaciones se pueden extender en tiempo de ejecución permitiendo que se descargue su código. No es nuestro caso.

Entonces teniendo en cuenta estas reglas veremos que nuestro servicio es cliente servidor ya que se van a escribir dos clientes que consumirán el servicio que está en el servidor. Las llamadas van a ser sin estado, ya que aunque el cliente sí tendrá en cuenta los datos del usuario el servicio no los necesita al recibir una llamada, si esta es correcta él responderá proporcionando los datos solicitados. Sí se utilizarán los datos de autenticación pero estos se le pasan al servidor no al servicio. Cada recurso tiene su propia dirección y es independiente del resto no existe ninguna interacción entre ellos. Va por capas ya que partimos de una aplicación empresarial y esta ya viene organizada por capas. Finalmente como la última regla es optativa, no se comparte el código.

Para acceder a la información se van a utilizar URI's (Uniform Resource Identifier). No se especifica que las URI tengan que ser enlaces pero como el servicio funciona sobre la web estos terminan siéndolo. Las llamadas se van a hacer por medio del protocolo http por lo que se nos permite utilizar los mensajes GET, POST, PUT, DELETE Para realizar consultas, modificar y añadir datos y borrar respectivamente dándonos capacidades CRUD. Así una llamada a un servicio REST sería tan simple como escribir una URI:

**`http://jsonplaceholder.typicode.com/users`**

Esto enviaría una petición GET al servicio sin parametros. El servicio alojado en jsonplaceholder.typicode.com responde a la petición GET en la dirección /users y como está programado para responder en esta nos devolverá los datos que se han solicitado:

```
[
  {
    "id": 1,
    "name": "Leanne Graham",
    "username": "Bret",
    "email": "Sincere@april.biz",
    "address": {
      "street": "Kulas Light",
      "suite": "Apt. 556",
      "city": "Gwenborough",
      "zipcode": "92998-3874",
      "geo": {
        "lat": "-37.3159",
        "lng": "81.1496"
      }
    },
    "phone": "1-770-736-8031 x56442",
    "website": "hildegard.org",
    "company": {
      ...
    }
  }
]
```

Como se puede observar la respuesta obtenida es en formato JSON. Esta sería una llamada simple mediante GET. También sería posible enviar parámetros en la URI o incluso información amplia en formatos como XML o JSON. Por ejemplo:

**<http://jsonplaceholder.typicode.com/users/10>**

Esta llamada refinaría la búsqueda indicando que se quiere ver el usuario con el id 10.

Para hacer llamadas al servicio previamente se deben conocer las URI que se pueden utilizar y sus parámetros de entrada y salida.

```
{
  "id": 10,
  "name": "Clementina DuBuque",
  "username": "Moriah.Stanton",
  "email": "Rey.Padberg@karina.biz",
  "address": {
    "street": "Kattie Turnpike",
    "suite": "Suite 198",
    "city": "Lebsackbury",
    "zipcode": "31428-2261",
    "geo": {
      "lat": "-38.23
    ...
  }
}
```

### 3.2 REST EN GLASSFISH

Se han barajado las posibilidades de utilizar los servidores Apache o Glassfish para la creación de la aplicación. Finalmente se ha escogido Glassfish ya que es el servidor utilizado durante el curso y habría que adaptar el código de Sparrow para utilizar Apache ya que la inyección de las entidades es diferente y requiere el uso de módulos externos para poder utilizar servicios REST.

Con este servidor se podrán publicar las aplicaciones JavaEE, instalar bases de datos, controlar la seguridad del servicio mediante roles de usuario y crear servicios REST.

Como vemos en (Gulabani, 2013) Java tiene el framework JAX-RS 2.0 que nos ayuda a escribir la parte de servidor de un servicio REST la implementación de este framework que se utilizará es Jersey 2.0. JAX-RS también establece sus reglas para la creación de servicios las cuales son muy parecidas a las propuestas originalmente.

- Todo tiene un identificador asignado.
- Las cosas van unidas entre sí.
- Se utilizarán una serie de métodos comunes.
- Se pueden utilizar diferentes tipos de representación.
- Las comunicaciones serán sin estado.

En esta misma publicación se nos indica que el tipo de proyecto con el que se tiene que empezar es del tipo Dynamic Web Project puesto que se necesitará conectividad a internet y las capacidades de configuración que proporciona el archivo web.xml que será esencial a la hora de añadir Jersey a la aplicación ya que la librería se tendrá que mapear como un servlet (Jendrock, Cervera-Navarro, Evans, Haase, & Markito, The Java EE 7 Tutorial: Volume 1, Fifth Edition, 2014) también hay que especificar aquí la URI principal de la cual colgarán las que responden a las peticiones que se realicen así como la configuración de seguridad del servidor.

Dentro de este proyecto irán los recursos que contienen la lógica del programa o las respuestas a las llamadas que realice el usuario. Esto incluirá las entidades para acceder a la base de datos, los DAO y los BO que son los que utilizará el código del servicio REST. Para ello se utilizarán las anotaciones facilitadas por Jersey que entre otras cosas especifican el path o URI de acceso, el tipo de llamada que se acepta (GET, POST...), el tipo de datos que se van a consumir o a producir, los parámetros de entrada y las opciones de seguridad.

Puesto que uno de los clientes que van a consumir el servicio estará escrito en javascript tiene sentido escoger como formato para intercambio de datos JSON ya que se podrá utilizar fácilmente mediante jquery (Libby, 2015) cargándolo con Ajax. Aunque existen gran variedad de librerías como Moxy o Jackson (Gulabani, 2013) que se pueden encargar de hacer la conversión o Marshalling entre clases y entidades Java y su versión JSON se ha optado por utilizar las funciones nativas que proporciona Jersey ya que son muy potentes sin tener que añadir mas librerías por lo que una vez mas la opción Glassfish/Jersey ha sido la apropiada. En (Heffelfinger, 2014) se puede ver que el API JAXB se encarga de hacer la conversión de clase a XML para que luego Jersey lo recoja y lo convierta en JSON. Esto se hace mediante anotaciones específicas para clases y entidades.

### 3.3 SEGURIDAD EN GLASSFISH

Es necesario implementar algún tipo de seguridad para proteger la parte de la página que es solo para usuarios registrados. Sparrow da acceso a usuarios invitados al index que tiene el Login y links a las otras dos partes públicas que son el formulario de alta de usuario y recuperar password. El resto de secciones requieren autenticación ya que acceden a recursos privados del servicio.

Puesto que se va a utilizar Jersey la primera opción es ver con que opciones de seguridad cuenta, así que se ha consultado la documentación oficial (Oracle Corporation, 2015). En esta se puede ver que es muy fácil definir que llamadas van a requerir seguridad, esto se puede hacer mediante anotaciones o a través del archivo web.xml siendo estas las únicas opciones que se ofrecen para la parte de servidor dejando el control de la seguridad a este. Para la parte de cliente ofrece la posibilidad de utilizar OAuth.

A la hora de programar servicios REST se puede escoger entre varios tipos de seguridad. Como vemos en (Mehta, 2014) hay que tener en cuenta que la seguridad tiene dos partes: autenticación y autorización. Como pone en dicha publicación la autenticación es el proceso mediante el cual un usuario demuestra al sistema que es él mismo y la autorización es el proceso que comprueba que un usuario determinado tiene permiso para ejecutar una operación.

Para poder cumplir con estas partes tenemos un amplio abanico de posibilidades como pueden ser SAML, OAuth, OpenId y tokens de acceso. SAML no tiene compatibilidad con REST, OAuth ya hemos visto que solo se puede utilizar en la parte de cliente por lo que de momento no sirve ya que aún estamos en la parte de servidor, OpenId funciona encima de OAuth 2.0 y tiene compatibilidad con servicios REST así que podría ser una solución y finalmente el uso de tokens de acceso que es una posibilidad muy extendida pero que requiere la programación del sistema de tokens entero.

Teniendo claros los posibles sistemas hay que ver las posibilidades que puede ofrecer el servidor en sí para ahorrar código y no tener que añadir librerías de terceros a la aplicación. Así en (Kalali, 2010) podemos observar que JavaEE ofrece parte de lo que es un sistema de seguridad propio basado en roles.

Este sistema cuenta con usuarios que se pueden agrupar y que cuentan con roles. Los roles especifican que acciones puede hacer o no un usuario. Por ejemplo el rol administrador tendría acceso a todas las acciones ofrecidas por el servicio, el rol registrado daría permisos para una parte de las acciones (quitando las de administración del sistema) y el rol invitado solo permitiría entrar en la página principal, darse de alta en el sistema y recuperar la clave. También tenemos los términos principal (la identidad que se va a identificar) y credential que sería la información necesaria para autenticar, por ejemplo una clave de entrada.

Todos estos términos se puede decir que son partes utilizadas por el Security Realm al cual han definido como el canal de acceso del servidor de la aplicación a un sistema de almacenamiento que contiene los datos de autenticación, que serían los que se acaban de mencionar. En el archivo web.xml se podrá configurar el realm y se definirá el nivel de seguridad de cada una de las acciones que permite realizar nuestro servicio.

Sabiendo esto ahora hay que ver que opciones nos ofrece Glassfish ya que JavaEE nos proporciona las herramientas para definir la seguridad pero en el fondo es el servidor el que se va a encargar de aplicarla. En este mismo manual tenemos que el servidor permite la creación de Security Realms de



diferentes tipos como: File Realm, JDBC Realm, LDAP Realm, Certificate Realm y Custom Realm. Por lo que habrá que elegir el que mejor se adapte a nuestro sistema.

File Realm es el tipo de seguridad más básico. Se reduce a un archivo de texto plano que contiene usuarios, claves y grupos. Solo es recomendado para desarrollo por lo que se descarta. JDBC Realm utilizará una base de datos para mantener estos datos, en un principio parece una buena opción puesto que la aplicación cuenta ya con una base de datos. LDAP Realm guarda la información en una estructura en árbol, como la estructura de organización de una empresa, un ejemplo de este sistema es Microsoft Active Directory, es una solución potente pero requiere el uso de un servidor externo lo cual complicaría el sistema. El Certificate Realm utiliza un tipo de credenciales diferente al visto, aquí el usuario necesita un certificado digital para acceder a los recursos por medio del protocolo https, esta solución sería la más segura pero también la más costosa en tiempo y dinero por lo que finalmente lo mejor es decantarse por JDBC Realm.

Para utilizar este tipo de realm hay que adaptar la base de datos original. A la hora de configurarlo se puede escoger el tipo de cifrado de clave se ha elegido SHA-256 así que ahora es necesario guardar las claves en este formato, también se guardará la clave en formato texto plano. También será necesaria una tabla nueva para definir los roles de cada usuario. Debido a estos cambios el código original a la hora de dar de alta a un usuario nuevo ya no valdrá por lo que se cambia también.

El acceso a la base de datos se hace mediante un pool de conexiones gestionado por el servidor (Kou, 2009). De esta forma se evitan problemas de acceso a la base de datos ya que la aplicación tendrá múltiples usuarios que requerirán acceso a esta. De esta forma se mantienen abiertas un grupo de conexiones distribuidas en hilos así se van utilizando conforme queden libres para realizar transacciones y cuando estas han terminado se pueden repartir entre los nuevos usuarios que estaban esperando.

### 3.4 AUTENTICACIÓN

Esta parte será gestionada por JavaEE. En el archivo web.xml se puede escoger el método de autenticación teniendo como posibilidades: basic, form, client-cert y digest. El método client-cert queda descartado ya que no se van a utilizar certificados de seguridad. Actualmente cuando se escriba en el navegador la ruta de una petición segura, el navegador sacará una ventana emergente que pide el nombre de usuario y la clave, si estos son correctos el servicio nos devolverá los datos solicitados (ilustración 3).

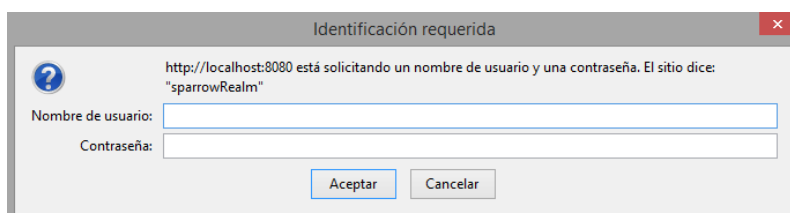


Ilustración 3

Esto se debe a que por defecto el realm utiliza el método basic. Si escogiéramos el método form se cargaría un formulario que debe programar el desarrollador, este necesita como mínimo dentro de una etiqueta form dos campos de texto (usuario y clave) y un botón de envío, estos campos tendrán unos nombres proporcionados por Java. Finalmente el método digest es como el basic solo que tiene seguridad añadida a la hora de enviar las credenciales.

Entonces, en un principio la opción más viable es form ya que podemos hacer el formulario nosotros mismos y queda integrado dentro de nuestro cliente si este se carga desde el navegador. Así el proceso es fácil, se pone este formulario, el usuario se autentica y saltamos a la página principal de la aplicación la cual hace llamadas al servicio, como el usuario ya está autenticado debemos obtener respuesta. Pues no funciona así. Si se hiciera esto se crearía un bucle infinito en el que la aplicación mandaría al usuario una y otra vez al formulario de acceso sin poder salir de ahí. Ello se debe a que basic y form están hechos para interactuar con un usuario y aquí lo que tenemos es un programa (cliente) interactuando con el servidor por lo que no puede rellenar él esos campos ya que es el cliente el que va a hacer las llamadas al servicio REST.

¿Cuál es la solución entonces? En (Knutson, 2012) podemos observar que la autenticación basic lo que realmente hace es obtener por medio de cabeceras http estos datos por lo que solo hay que añadir una cabecera nueva que contiene el nombre y la clave separados por dos puntos ":" y codificados en Base64. Ahora el cliente sin importar en que lenguaje se programe puede enviar esta información en la cabecera y así consumir datos de respuestas que requieran autenticación. La cabecera a enviar es, siendo la parte en negrita la que se codificará en Base64.

Authorization: Basic **usuario:clave**

Las cabeceras http que encontramos serán:

```
accept = application/json
authorization = Basic dXN1YXJpbzAxOjEybW9ub3M=
user-agent = Jersey/2.10.4 (URLConnection 1.8.0_40)
host = localhost:8080
connection = keep-alive
```

Aquí se puede observar la cabecera que se ha añadido y que el agente no es un navegador, es Jersey por lo que se demuestra que el servicio está siendo consumido por un programa.

En muchas de las llamadas que se hacen al servicio será necesario conocer el nombre usuario pero como ya sabemos REST no guarda la información de sesión. Se podría enviar el nombre de usuario en cada petición de este tipo pero esta práctica sería acusada por la seguridad ya que tendríamos parte de la autenticación viajando continuamente. Para eso Jersey nos proporciona la anotación:

@Context SecurityContext

Que nos devuelve los datos de autenticación y por lo tanto el nombre de usuario.

### 3.5 CORS

Con todo lo anterior ya es posible programar el cliente Java sin problemas. Ahora es cuando se empieza a escribir el cliente en HTML y javascript. El principal problema que va a traer este cliente se llama CORS (Cross Origin Resource Sharing). La explicación la podemos encontrar en (Hossain, 2014), CORS permite que un cliente web pueda hacer llamadas http a servidores de diferentes orígenes. Es una tecnología que se aplica tanto en cliente como en servidor.

En la parte de servidor se configura que peticiones CORS se pueden realizar y en la parte de cliente las que se podrán recibir. Una explicación más sencilla es que CORS es hacer llamadas http desde un sitio a otro. Por lo general es fácil de hacer en cualquier lenguaje menos en javascript ya que el navegador web no lo permite por motivos de seguridad.

A la hora de acceder a los datos de nuestro servicio desde una página web por medio de javascript nos encontraremos con un desagradable mensaje en la consola del navegador (ilustración 4):

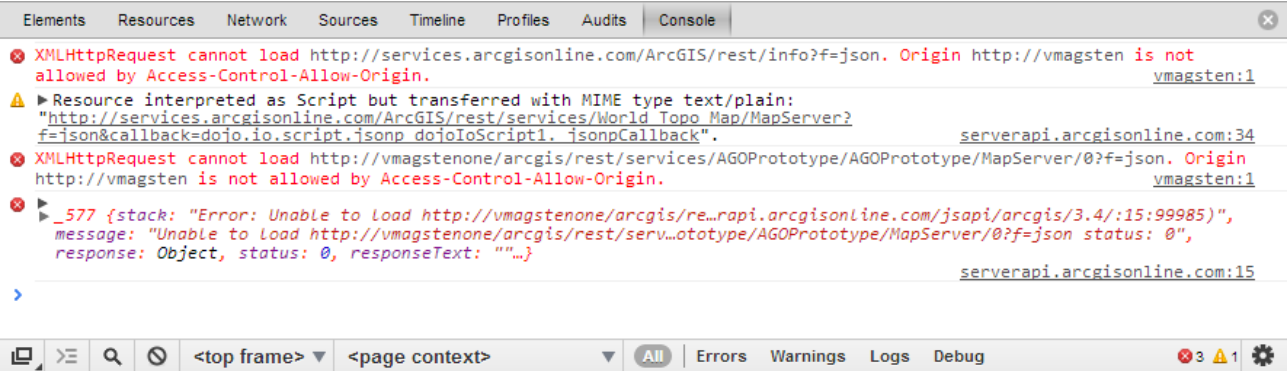


Ilustración 4

Para solucionar este problema tenemos que hacer cambios en el servidor. Este debe indicar al navegador que permite utilizar CORS (ilustración 5).

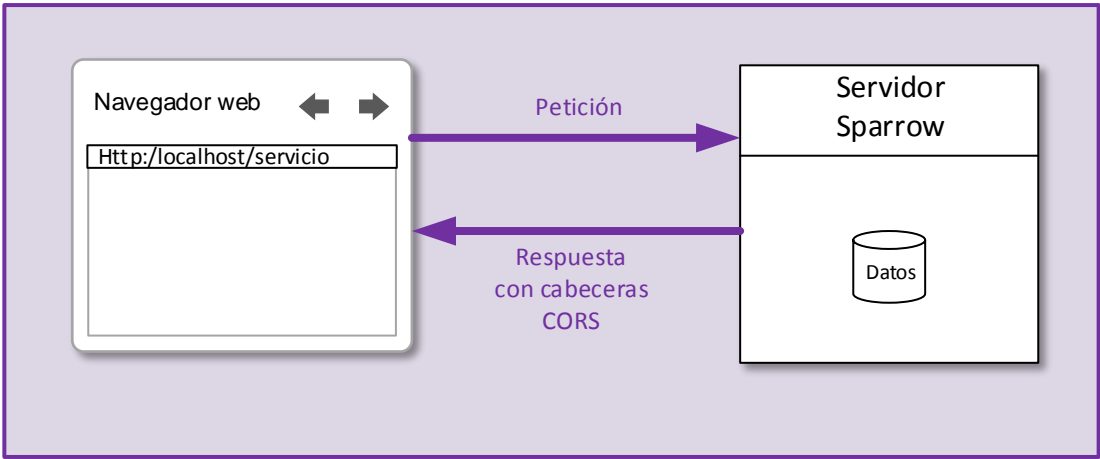


Ilustración 5

Las cabeceras que tendrá que enviar el servidor serán (Hossain, 2014):

Access-Control-Allow-Origin	*
Access-Control-Allow-Methods	GET, POST
Access-Control-Allow-Headers	X-Requested-With,Authorization, Content-Type, X-Codingpedia

Ahora el problema es como enviar cabeceras en cada llamada que se haga al servicio. Lo ideal sería utilizar un filtro y añadirlas siempre pero como estamos trabajando en REST no funcionaría correctamente por lo que se ha buscado otra solución. En (Oracle Corporation, 2015) se ha encontrado esta y son los filtros propios de Jersey. Hemos añadido un filtro que en cada llamada

que se hace se van a devolver las cabeceras haciendo que estas sean correctas para el navegador y sin influir en el funcionamiento del cliente Java ya que no las va a leer.

### 3.6 AJAX Y JQUERY

El envío y recepción de datos al servicio se hará mediante llamadas GET y POST. Para ciertas operaciones se pasarán parámetros por la URI, en otros casos hay que enviar clases completas en JSON y los datos se reciben en JSON.

Aunque es posible hacer estas operaciones mediante javascript puro se ha optado por utilizar jQuery ya que asegura compatibilidad con diferentes navegadores y hace que esta tarea sea más fácil. En (Libby, 2015) se ha comprobado que las funciones para hacer este tipo de llamadas son similares a las que hay para hacerlas mediante Ajax por lo que se han adoptado estas últimas para aprovechar sus capacidades.

Puesto que los datos se guardan en JSON se han creado clases iguales que las que podemos encontrar en el servicio.

### 3.7 DATOS DE SESIÓN EN EL CLIENTE JAVASCRIPT

Una página web no puede guardar los datos del usuario entre páginas. Para ello se van a utilizar cookies. Para facilitar su uso se ha añadido la librería js.cookies que facilitará las cosas.

Se va a guardar el nombre de usuario y la clave en Base64 para poder enviarla en las cabeceras cuando se haga una llamada a una operación que requiera autenticación, no es un sistema seguro ya que ahora sí que viaja toda la información de autenticación pero es la única posibilidad si no se utiliza un sistema de seguridad más avanzado como OAuth. También se guarda el nombre de usuario ya que es necesario para ciertas llamadas al servicio. Así las cookies quedarán (ilustración 6).

Name	sparrowData	Name	sparrowUsr
Value	dXN1YXJpbzAxOjEyW9ub3M=	Value	usuario01
Host	localhost	Host	localhost
Path	/	Path	/
Expires	Fri, 21 Aug 2015 10:19:28 GMT	Expires	Fri, 21 Aug 2015 10:19:28 GMT
Secure	No	Secure	No
HttpOnly	No	HttpOnly	No

Ilustración 6

## 4 ESPECIFICACIÓN

### 4.1 ANÁLISIS DE REQUISITOS

El usuario final de este proyecto va a obtener un servicio web que le dará acceso al sistema de Sparrow por medio de REST. También se escribirán dos clientes para probar la compatibilidad del sistema en diferentes lenguajes.

Para ello se adapta el código de la aplicación original. Lo primero será hacer los cambios necesarios en la base de datos para que pueda ser utilizada por el realm y modificar las entidades para que

puedan utilizar las nuevas tablas y puedan ser convertidas a JSON. Se utilizarán los DAO y los BO originales.

Puesto que se van a enviar datos muy específicos en JSON se han creado nuevas clases que son una versión reducida de las entidades.

Casi todos los cambios van a caer entonces en la parte de seguridad configurando el servidor de forma apropiada y configurando el archivo web.xml y por supuesto escribiendo toda la parte correspondiente al servicio en sí.

El servicio debe facilitar todas las operaciones necesarias para poder escribir un cliente en cualquier lenguaje por lo que se tienen en cuenta las limitaciones de los distintos lenguajes siendo el más limitado javascript.

## 4.2 ESPECIFICACIÓN DEL SISTEMA

El desarrollo del servicio y del cliente Java se hará con el entorno de desarrollo Eclipse Luna 2 puesto que permite el desarrollo de este tipo de aplicaciones y la creación de entidades desde la base de datos, al servicio se le añadirá la librería Jersey 2.0. Para el cliente javascript se contará Netbeans 8.0.2 ya que posee mejores opciones de edición para archivos HTML y javascript. A este cliente se le añadirán las librerías js.cookie y jQuery.

Como base de datos tendremos MySQL y para su gestión las herramientas phpMyAdmin y mysqlWorkbench.

El servidor para conectar con la base de datos, publicar las aplicaciones y controlar la seguridad ya se ha dicho que será Glassfish 4.

Para hacer pruebas serán necesarios los navegadores Chrome y Firefox ya que cuentan con plugins para análisis de cabeceras y cookies.

## 4.3 PLANIFICACIÓN Y ESTIMACIÓN DE COSTES

//TODO pues todo

# 5 DESARROLLO DEL PROYECTO

---

## 5.1 ANÁLISIS

### 5.1.1 Descripción de Sparrow

Sparrow es una pequeña red social en la que se pueden crear temas y responderlos con mensajes cortos de 500 caracteres de máximo. También es posible buscar usuarios y seguirlos o dejar de hacerlo.

La página principal (ilustración 7) incluye el formulario de Login y da acceso al formulario de inscripción (ilustración 8) o al de recuperación de clave (ilustración 9).



## Sparrow

Nombre:

Clave:

[Crear cuenta](#)

[Recuperar password](#)

Ilustración 7

## Registrarse como nuevo usuario

Nombre:

Apellidos:

Sexo:

Idioma:

Email:

Username:

Password:

Repite password:

Ilustración 8

## Recuperar password

Usuario:

Email:

Ilustración 9

Una vez el usuario se ha autenticado accede a lo que sería la aplicación en sí. Desde aquí se pueden crear nuevos temas, ver los que se han escrito, ver usuarios seguidos y seguidores, buscar usuarios y acceder a la configuración de la cuenta (ilustración 10).



Ilustración 10

La aplicación necesita algunas páginas más para dar todas las funcionalidades. El menú preferencias debe permitir editar los datos de usuario exceptuando su nombre y email (ilustración 11).

The screenshot shows the 'Preferencias' (Preferences) form. The title 'Preferencias' is in red and underlined. The form contains several fields for user data: 'Nombre de usuario:' (usuario01), 'Password:' (12monos), 'Email:' (kkk@kkk.es), 'Nombre:' (Usuario 01js), 'Apellidos:' (Pruebajs), 'Sexo:' (M with a dropdown arrow), and 'Idioma:' (German with a dropdown arrow). There is an 'Actualizar' (Update) button and a 'Volver' (Back) link at the bottom left.

Ilustración 11

Al pulsar sobre un tema se accederá a la lista de chips de este (ilustración 12) y desde aquí se podrán añadir nuevos chips a estos. También hay páginas para búsqueda de usuarios y crear tema.

Chips para: tema 2

texto tema2 [usuario01]

Responder Chip

asdasdas [usuario01]

Responder Chip

repuesta 2 [usuario01]

Responder Chip

Ilustración 12

### 5.1.2 Casos de uso

Los casos de uso van a ser esencialmente los mismos que tenía la aplicación original mas los añadidos para los nuevos tipos de clientes.

#### Cliente

Id	Caso de uso – Acceso a zona privada
A-1	<b>Actores:</b> Usuario invitado
	<b>Prerrequisitos:</b> Ninguno
	<b>Descripción:</b> El usuario suministra su nombre y clave. Estos datos se guardan junto a su versión codificada en Base64.

Id	Caso de uso – Registro de usuario
A-2	<b>Actores:</b> Usuario invitado
	<b>Prerrequisitos:</b> Ninguno
	<b>Descripción:</b> El usuario invitado debe registrarse para acceder a la zona privada. Se suministra la siguiente información: Nombre de usuario único, clave, email, nombre y apellidos, sexo (V,M), idioma (de,en,es). Si la información es correcta se mandará en formato JSON al servicio utilizando la clase User. Si no lo es, mostrar mensaje de error.

Id	Caso de uso – Recuperar clave
A-3	<b>Actores:</b> Usuario invitado
	<b>Prerrequisitos:</b> Ninguno
	<b>Descripción:</b> El usuario ha olvidado su clave y necesita recuperarla. La aplicación le preguntará su nombre de usuario y email. Estos datos se enviarán en formato JSON al servicio dentro de la clase Password y el servicio devolverá un objeto JSON de tipo Password con la clave o un mensaje de error en el campo de email.



Id		Caso de uso – Gestión de datos del usuario
A-4	<b>Actores:</b> Usuario registrado	
	<b>Prerrequisitos:</b> Tener credenciales de acceso y enviarlas en la cabecera de la petición	
	<b>Descripción:</b> El usuario puede editar parte de los datos de su cuenta. Para ello se recibirá en JSON en un objeto tipo Users desde el servicio los datos del usuario para mostrarlos. El nombre de usuario y su email no se pueden editar. Estos datos se tomarán y se enviarán al servicio en JSON con un objeto tipo USERS. La clave debe estar codificada en SHA-256.	

Id		Caso de uso – Buscar usuario
A-5	<b>Actores:</b> Usuario registrado	
	<b>Prerrequisitos:</b> Tener credenciales de acceso y enviarlas en la cabecera de la petición	
	<b>Descripción:</b> El usuario puede buscar a otros usuarios del sistema introduciendo sus apellidos. Se enviarán los apellidos al servicio como parámetro en la URL. Si hay usuarios que corresponden con esos apellidos se mostraran en una lista que se recibe en JSON con la clase Users. Si la lista está vacía es que no hay coincidencias.	

Id		Caso de uso – Mostrar temas de discusión
B-1	<b>Actores:</b> Usuario registrado	
	<b>Prerrequisitos:</b> Tener credenciales de acceso y enviarlas en la cabecera de la petición	
	<b>Descripción:</b> Al entrar en la aplicación se muestra una lista de temas (chips con thread null) que se recibirán del servicio en JSON con la clase Topics. Desde aquí el usuario podrá acceder al caso: Mostrar chips por tag.	

Id		Caso de uso – Mostrar chips por tag
B-2	<b>Actores:</b> Usuario registrado	
	<b>Prerrequisitos:</b> Tener credenciales de acceso y enviarlas en la cabecera de la petición y el tag del tema	
	<b>Descripción:</b> Mostrar una lista con los chips de respuesta del tema que se reciben con la clase Chips, se mandará el tema por URL. Desde aquí se llega al caso: Contestar chip.	

Id		Caso de uso – Crear tema
B-3	<b>Actores:</b> Usuario registrado	
	<b>Prerrequisitos:</b> Tener credenciales de acceso y enviarlas en la cabecera de la petición y el nombre de usuario	
	<b>Descripción:</b> El usuario crea un tema nuevo facilitando el texto y el tag de este. Se enviarán los datos con un objeto de la clase Topics.	

Id		Caso de uso – Contestar un chip
B-4	<b>Actores:</b> Usuario registrado	
	<b>Prerrequisitos:</b> Tener credenciales de acceso y enviarlas en la cabecera de la petición y el thread al que se responde.	
	<b>Descripción:</b> El usuario responde a un tema. Los datos se envían en la clase Chips.	

Id	Caso de uso – Seguir usuario
C-1	<b>Actores:</b> Usuario registrado
	<b>Prerrequisitos:</b> Tener credenciales de acceso y enviarlas en la cabecera de la petición y el usuario a seguir.
	<b>Descripción:</b> El usuario añade el usuario a la lista enviando un objeto Follows.

Id	Caso de uso – No seguir a usuario
C-2	<b>Actores:</b> Usuario registrado
	<b>Prerrequisitos:</b> Tener credenciales de acceso y enviarlas en la cabecera de la petición y el usuario.
	<b>Descripción:</b> El usuario quita a otro usuario de la lista. El usuario se indica al servicio mediante la clase Follows.

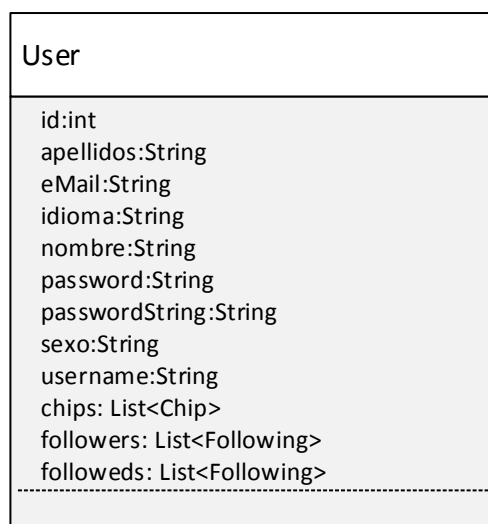
Id	Caso de uso – Usuarios seguidos
C-3	<b>Actores:</b> Usuario registrado
	<b>Prerrequisitos:</b> Tener credenciales de acceso y enviarlas en la cabecera de la petición
	<b>Descripción:</b> Se recibe una lista de objetos MiniUser desde el servicio que mostrará los usuarios seguidos. Desde aquí se da la opción de dejar de seguir a un usuario.

Id	Caso de uso – Seguidores
C-4	<b>Actores:</b> Usuario registrado
	<b>Prerrequisitos:</b> Tener credenciales de acceso y enviarlas en la cabecera de la petición
	<b>Descripción:</b> Se recibe una lista de objetos MiniUser desde el servicio que mostrará los seguidores del usuario. Desde aquí es posible seguirles a ellos.

### 5.1.3 Diagramas de clases

#### SERVICIO

#### Entidades



UsersGroup
id:int groupid:String username:String

Chip
id:int tag:String text:String chip:Chip chips:List<Chip> userBean:User

Following
id:FollowingPK since:Timestamp followed:User follower:User

FollowingPK
user:int followed:int
equals(Object:other):boolean hashCode():int

## Entidades de Servicio

Estas clases también se utilizarán en los clientes.

Chips
texto:String autor:String id:int
Chips(String texto,String autor,int id)

Follows
miUsuario:String idSeguido:String

MiniUser
username:String id:String
MiniUser(String username, String id)

Password
email:String nombre:String

Topics
id:int tag:String text:String user:String

Dao

<<Interfaz>> Dao<K,E>
persist(Entity:E) remove(Entity:E) findById(K:id):E

JpaDao<K,E>
entityClass:Class<E> em:EntityManager
JpaDao(EntityManager:em) findById(K:id):E persist(Entity:E) remove(Entity:E)

## UserDao

```
Userdao(EntityManager:E)
getAllUsers():List<User>
findUserByUsername(String:username):User
findByApellidos(String:apellidos):User
update(User:usr)
```

## UserGroupDao

```
UserGroupDao(EntityManager:em)
setGroup(UsersGroup:grupo)
```

## ChipDao

```
ChipDao(EntityManager:em)
getAllChips():List<Chip>
getAllThemes():List<Chip>
getByTag(String:tag):List<Chip>
```

## FollowDao

```
FollowDao(EntityManager:em)
update(Following:follow)
buscar(User:seguido, User:miUsuario):Following
```

## BO

<<Interfaz>>

## UserBoRemote

```
listaUsuarios():List<User>
buscarUsuario(String:username):User
validaUsuario(String:nombre,String:pass):boolean
addUser(User:nuevo)
recuperaPassword(String:nombre, String:email):String
editUser(User:editado)
buscaApellidos(String:apellidos):List<User>
buscarUsuarioID(String:id):User
```

UserBo
em:EntityManager udao:UserDao
init() finaliza() listaUsuarios():List<User> buscaUsuario(String:username):User buscaUsuariold(String:id):User validaUsuario(String:nombre, String:pass):boolean addUser(User:nuevo) recuperaPassword(String:nombre, String:email):String editUser(User:editado) buscaApellidos(String:apellidos):List<User>

<<Interfaz>>
UsersGroupBoRemote
ponEnGrupo(String usuario)

UsersGroupBo
em:EntityManager uGDao:UserGroupDao
init() finaliza() ponEnGrupo(String usuario)

<<Interfaz>>
ChipBoRemote
listaChips():List<Chip> addChip(Chip:nuevo) listaTemas():List<Chip> listaPorTag(String:tag):List<Chip> damePorld(String:id):Chip

ChipBo
em:EntityManager cDao:ChipDao
init() finaliza() listaChips():List<Chip> addChip(Chip:nuevo) listaTemas():List<Chip> listaPorTag(String:tag):List<Chip> damePorId(String:id):Chip

<<Interfaz>> FollowBoRemote
seguir(Following:follow) noSeguir(Following:follow)

FollowBo
em:EntityManager fDao:FollowDao uDao:UserDao
init() finaliza() seguir(Following:follow) noSeguir(Following:follow)

## Datos de usuario

<<Interfaz>> SimpleUserRemote
getUsername():String setUsername(String:username) getId():int setId(int:id)

SimpleUser
username:String id:int
getUsername():String setUsername(String:username) getId():int setId(int:id)

<<Interfaz>>

UsersBeanRemote

setUser(User:usuario)  
getUser():User  
setLogged(boolean:logged)  
getLogged():boolean

UsersBean

name:String  
password:String  
logged:boolean  
user:User  
-----  
setUser(User:usuario)  
getUser():User  
setLogged(boolean:logged)  
getLogged():boolean

## Servicio

ServicioUsers

context:UriInfo  
userBo:UserBoRemote  
userG:UserGroupBoRemote  
followBo:FollowBoRemote  
USERBEAN\_ATTR:String  
-----  
dameListaUsers(SecurityContext:sc, HttpServletResponse: response):List<User>  
dameSeguidos(String:usuario):ArrayList<Mini User>  
dameSeguidores(String:usuario):ArrayList<Mini User>  
addUser(User:usuario)  
editUser(User:usuario)  
sigueUsuario(Follows:seguir)  
noSeguirUsuario(Follows:seguir)  
resetPassword(Password:pass):Password  
buscaUsuario(String:apellido):List<User>  
generaClave(String:password\_str):String  
dameUsuario(String:username):User

ServicioChips

context:UriInfo  
chipBo:ChipBoRemote  
userBo:UserBoRemote  
-----  
dameTemas(SecurityContext:sc):ArrayList<Topics>  
dameTemasPorTag(String:tag):ArrayList<Chips>  
ponTema(SecurityContext:sc, Topics:tema)  
respondeChip(SecurityContext:sc, Chips:chip)



Headers
doFilter(ServletRequest:request, ServletResponse:response, FilterChain:chain)

FiltroAjax
filter(ContainerRequestContext:requestContext, ContainerResponseContext:responseContext)

## CLIENTES

El cliente java es una aplicación web que utiliza las entidades de servicio y servlets, el cliente web también utiliza las mismas clases por lo que no hay más clases que añadir.

### 5.1.4 Base de datos

El diseño de la base de datos modificada quedaría como se muestra en la ilustración 13. Se puede observar que los datos necesarios para el realm estarán en la tabla users (password, password\_string y username) y en la tabla users\_groups.

En esta última tabla se indica el username y se le asigna como groupid USERS que es el grupo asignado a los usuarios registrados en la aplicación.

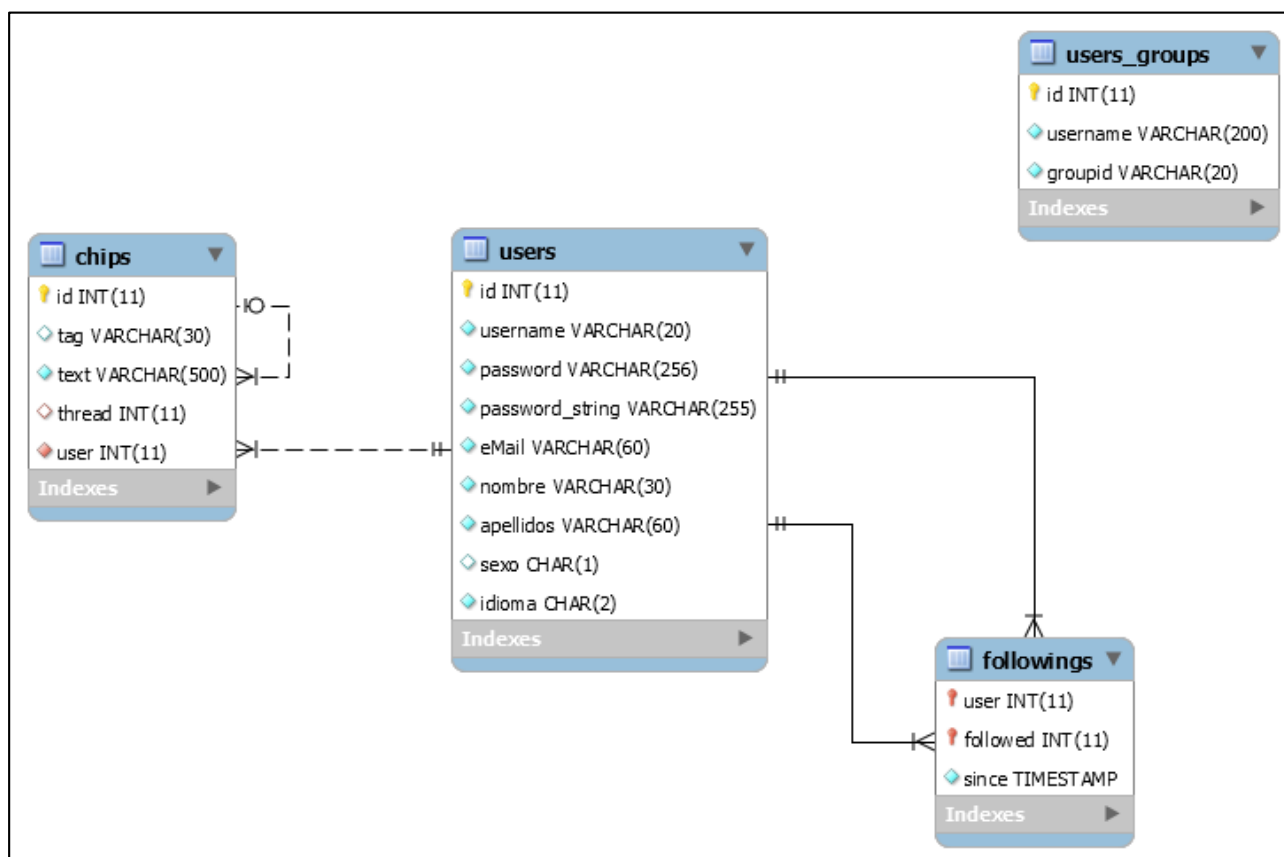


Ilustración 13

## 5.2 DISEÑO

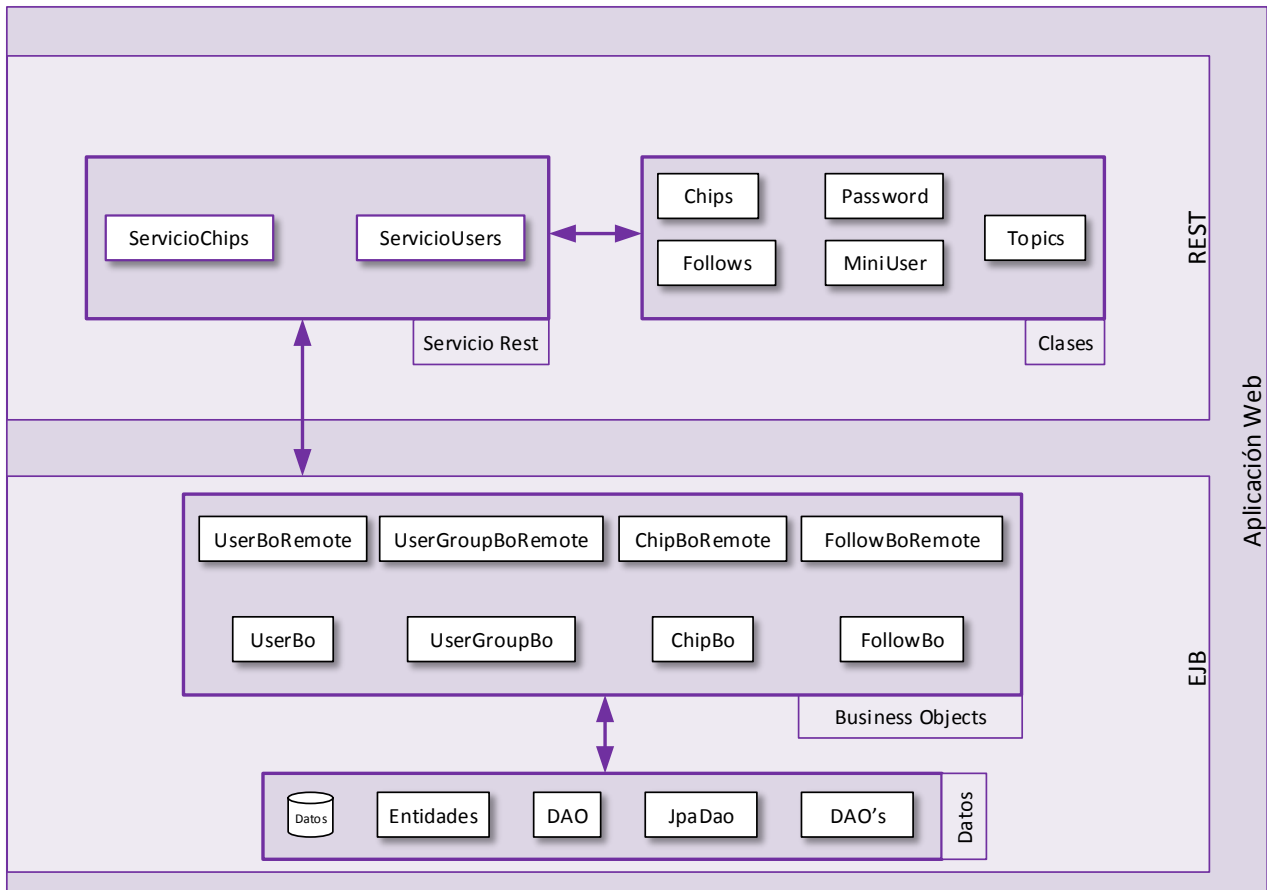


Ilustración 14

En la ilustración 14 se puede observar el diseño de la aplicación y como este funciona por capas. La que se podría llamar la capa inferior es la que conectará con la base de datos. Esto se hará a través de un pool de conexiones mediante el servidor. La base de datos nos proporciona las entidades que se controlarán mediante los DAO's.

Por encima están los Business Objects que serán los que proporcionarán las funciones de acceso a la base de datos y la lógica necesarias para estas. Estas dos partes componen lo que sería la parte EJB de la aplicación.

La que sería una capa superior es la encargada del servicio RESTful en sí. Contiene las clases necesarias para los clientes y el acceso a los métodos que podrán acceder estos. El servicio se divide en dos partes la que se encarga de funciones relacionadas con datos de usuario y la que proporciona acceso a los Chips. Para hacer esto se comunicará con los BO y estos a su vez con la capa encargada de gestionar los datos de la aplicación.

## 5.3 IMPLEMENTACIÓN

### 5.3.1 Acceso a la base de datos

Para guardar toda la información se utilizará una base de datos MySQL con el diseño expuesto en 5.1.4. Esta base de datos se cargará en un servidor MySQL y para acceder a ella se hará mediante un pool de conexiones a través del servidor Glassfish. Así la conexión a la base de datos ya no es directa como vemos en la ilustración 15.

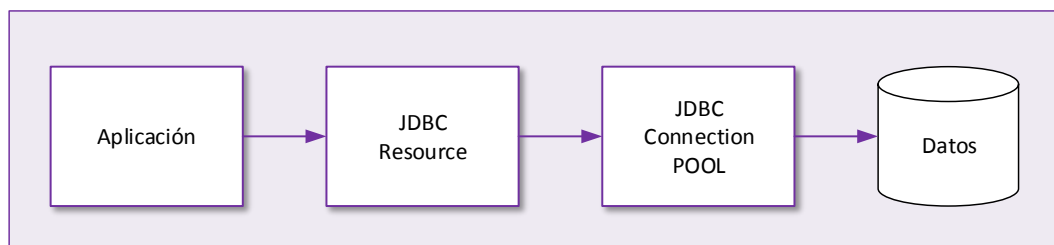


Ilustración 15

El primer paso es instalar el controlador de MySQL en el servidor para lo que se descargará la librería necesaria, en este caso se ha utilizado la versión 5.1.34 de la librería MySQL connector para java, para copiarla en la carpeta lib del servidor. Al iniciar este ya estará funcionando.

A continuación hay que crear un pool de conexiones al que llamaremos sparrow cuya configuración es la de la siguiente tabla:

Pool name	<b>sparrow</b>
Resource Type	javax.sql.DataSource
Driver	mysql
Url y url	jdbc:mysql://localhost:3306/sparrow
password	12monos
databaseName	sparrow
serverName	localhost
user	root
portNumber	3306

Para terminar la configuración del servidor se configurará un recurso JDBC nuevo llamado jdbc/sparrowpool y que hará referencia al pool sparrow.

### 5.3.2 Entidades

Las entidades son clases que harán referencia a la base de datos evitando la programación directa de esta en muchos casos. Como leemos en (Heffelfinger, 2014) estas entidades funcionan mediante JPA que se introdujo en la versión 5 de JavaEE y se encargarán de la persistencia de los datos.

A simple vista son como las clases normales pero con una serie de anotaciones que las identifican, siendo la principal @Entity. A continuación se especifica la tabla a la que hace referencia mediante @Table. Las columnas serán atributos de la clase y con anotaciones como @OneToMany es posible establecer las relaciones con otras clases. Estas clases deben implementar al interfaz Serializable y tendrán un constructor vacío para poder moverlas ya que estamos utilizando un sistema distribuido.

Para crear las entidades se ha utilizado un módulo de creación que incluye el IDE Eclipse que se encuentra dentro de las JPA Tools. Antes de poder utilizarlo hay que especificar una conexión a la

base de datos. Esta utilizará la misma información que se ha utilizado para configurar el pool de conexiones como se puede observar en la ilustración 16.

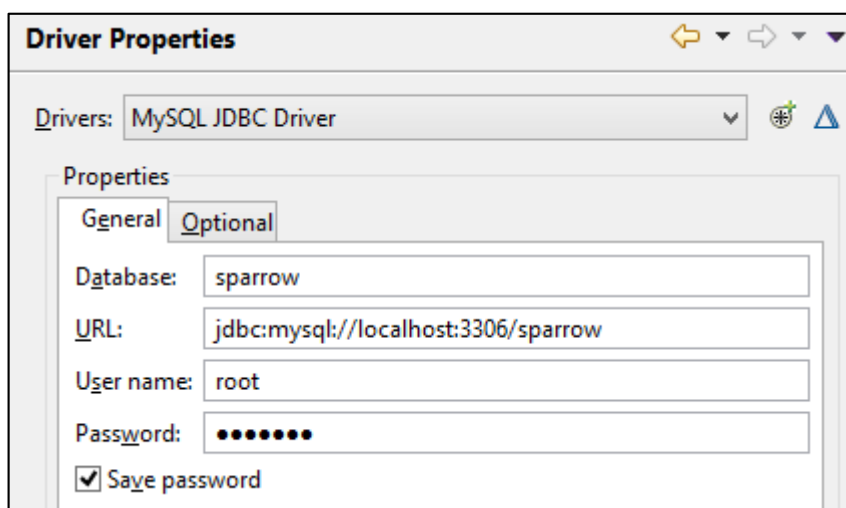


Ilustración 16

El esquema de la base de datos con la que vamos a trabajar es el expresado en la ilustración 17 o en la ilustración 13.

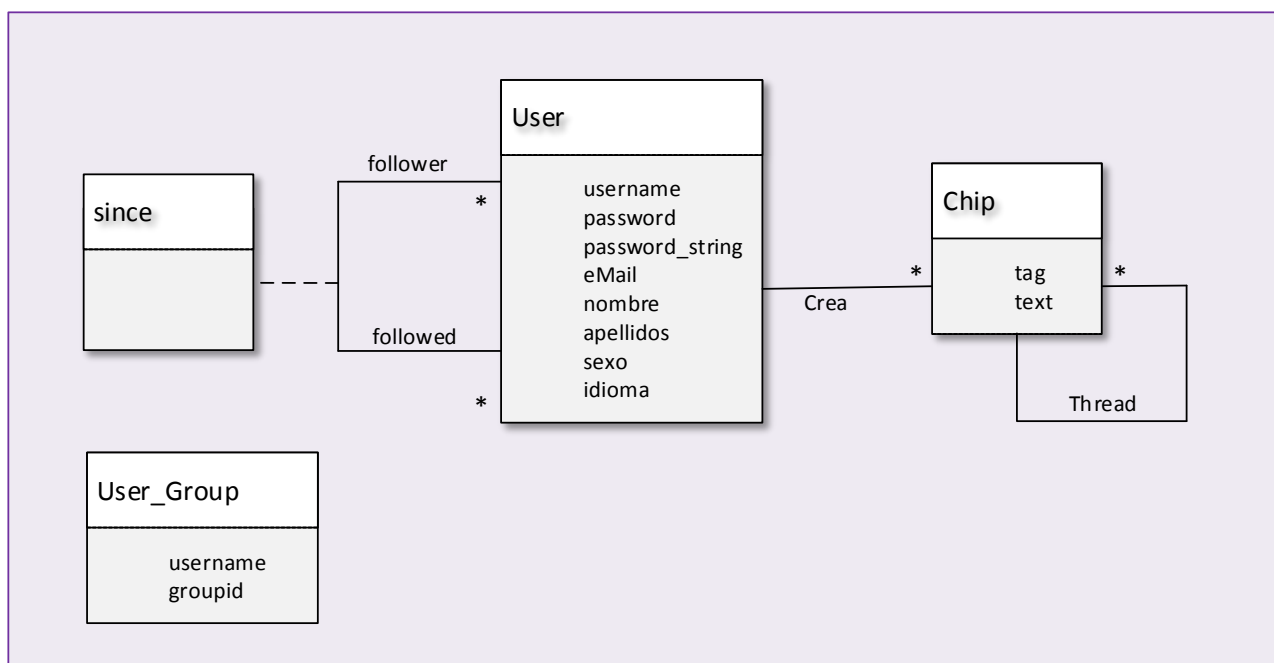


Ilustración 17

Viendo este esquema podemos saber que un usuario puede tener usuarios follower que son los que le siguen y usuario followed. El usuario puede crear chips que tienen un tag que sería el título y un texto.

Los chips pueden tener threads que serían respuestas. Así si un chip tiene un thread null es que es un tema y desde este colgarán otros chips respuesta que tendrán un thread que es el tema principal.

En la tabla User\_Group se indica a que grupo pertenece cada User, esta información se utilizará para el realm de seguridad. Todos los usuarios pertenecen al grupo USER por defecto.

Una vez configurada la conexión a la base de datos en Eclipse ya es posible utilizar JPATools para obtener las entidades desde la base de datos. Al aplicarlo a la conexión establecida se obtendrá por defecto la configuración de la ilustración 18.

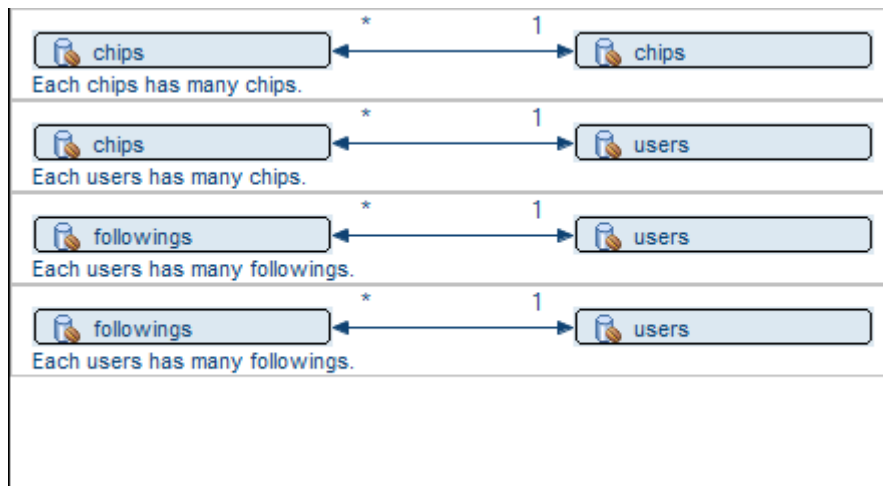


Ilustración 18

En un principio se podría utilizar con los valores por defecto que ofrece la herramienta pero se van a cambiar algunos para mejorar la legibilidad del código ya que el nombre de las variables afectará a los nombres de los getters/setters generados y los actuales no dejan claro a que se refieren así que se procede a cambiar las dos últimas relaciones de acuerdo con las ilustraciones 19 y 20.

Table associations

Diagrama de asociaciones JPATools con configuración personalizada. Muestra cuatro relaciones entre entidades:

- chips (1) a chips (\*): Each chips has many chips.
- chips (1) a users (\*): Each users has many chips.
- followings (1) a users (\*): Each users has many followings.
- followings (1) a users (\*): Each users has many followings.

☒ Generate this association

Cardinality: many-to-one

Table join: followings.followed=users.id

☒ Generate a reference to users in followings

Property: followed

Cascade:

☒ Generate a reference to a collection of followings in users

Property: followers

Cascade:

Ilustración 19

Table associations

chips	*	1	chips	Each chips has many chips.
chips	*	1	users	Each users has many chips.
followings	*	1	users	Each users has many followings.
followings	*	1	users	Each users has many followings.

☒ Generate this association

Cardinality: many-to-one

Table join: followings.user=users.id

☒ Generate a reference to users in followings

Property: follower

Cascade:

☒ Generate a reference to a collection of followings in users

Property: followeds

Cascade:

Ilustración 20

Con dicha configuración ya se pueden obtener las entidades descritas en 5.1.3. y el archivo persistence.xml que se encarga de enlazar las entidades creadas con el pool de conexiones. Además de las entidades básicas se añadirá FollowingPK que se encarga de la relación followed de User.

### 5.3.3 Data Acces Object (DAO)

Se va a utilizar el patrón DAO que nos permite separar la lógica de negocio, los Bussines Objects, de la capa de persistencia de forma que si se modifica la base de datos solo hay que cambiar las entidades de acuerdo con ella para seguir utilizando la aplicación.

El DAO va a proporcionar un API para dar acceso a la base de datos mediante operaciones CRUD (Create, Read, Update y Delete) además de las operaciones de búsqueda necesarias para el sistema.

En nuestro caso se va a partir de un interface Dao que se implementará mediante JpaDao a partir del cual se crearán subclases específicas para las diferentes entidades. En la ilustración 21 se puede observar el detalle de que Dao es una clase parametrizada por lo que las subclases tendrán que indicar los valores que requiere. De esta forma podremos utilizar cualquier tipo de entidad en el DAO.

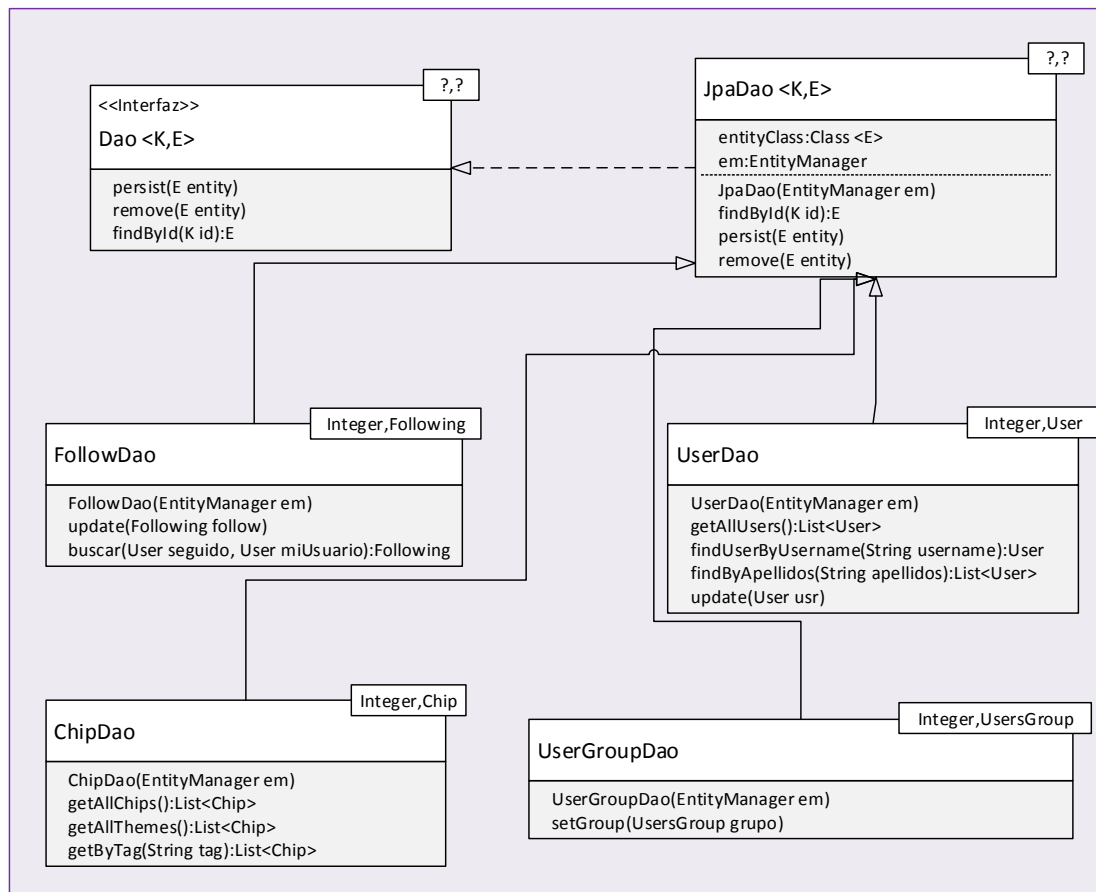


Ilustración 21

Para poder hacer búsquedas específicas como se observa en los DAO de las entidades habrá que editar estas y añadir namedQueries en formato JPQL. Así tenemos que para la entidad encargada de la tabla de usuarios las líneas a añadir serán:

```

@NamedQueries({
    @NamedQuery(name="User.findAll", query="SELECT u FROM User u"),
    @NamedQuery(name="User.findByUsername", query="SELECT e FROM User e
WHERE e.username LIKE :username"),
    @NamedQuery(name="User.findByApellidos", query="SELECT e FROM User e
WHERE e.apellidos LIKE :apellidos")
})

```

En el caso de la entidad Following:

```

@NamedQueries({
    @NamedQuery(name="Following.findAll", query="SELECT f FROM Following
f"),
    @NamedQuery(name="Following.noFollow", query="SELECT f FROM Following f
WHERE f.followed.id LIKE :seguido AND f.follower.id LIKE :seguidor")
})

```

Y para los chips:

```
@NamedQueries({
    @NamedQuery(name="Chip.findAll", query="SELECT c FROM Chip c"),
    @NamedQuery(name="Chip.findThemes", query="SELECT c FROM Chip c WHERE
c.chip=NULL"),
    @NamedQuery(name="Chip.findByTag", query="SELECT c FROM Chip c WHERE
c.tag LIKE :nombreTag")
})
```

El resto de entidades solo tienen el query por defecto que devuelve todas las entradas.

#### 5.3.4 Business Objects (BO)

Estas clases se van a encargar de proporcionar la lógica a la aplicación o reglase de negocio. Son Beans de sesión Stateless con interfaz remoto. Sus métodos nos darán acceso a operaciones complejas que requieren el uso de los DAO de forma que la aplicación en sí evitará el acceso directo a los DAO. Cada entidad cuenta con su propio BO el cual no tiene por qué limitarse a utilizar solo los DAO de su entidad.

Entre los métodos de UserBo encontramos recuperaPassword. Este recogerá el nombre de usuario y el email, si estos son correctos devolverá la clave en un String. Se ha aprovechado esto para devolver directamente el mensaje de error que mostrará la aplicación cliente.

En el caso de UsersGroupBo tenemos el método ponEnGrupo que añadirá el usuario al grupo USERS directamente cuando este se dé de alta en el sistema. Esta información será utilizada mas tarde en el control de seguridad que realiza el servidor mediante el realm.

#### 5.3.5 Seguridad

La seguridad está controlada por el servidor por lo que antes de continuar es necesario configurar el security realm para poder ir añadiendo al archivo web.xml las funciones del servicio que necesitan autenticación. Así que el primer paso es entrar en la consola de configuración de Glassfish y empezar a definir el realm. Este se llamará sparrowRealm y será del tipo JDBCRealm. La configuración de este será la siguiente:

<b>JAAS Context</b>	jdbcRealm
<b>JNDI</b>	Jdbc/sparrowpool
<b>User Table</b>	users
<b>User Name Column</b>	username
<b>Password Column</b>	password
<b>Group Table</b>	users_groups
<b>Group Table User Name Column</b>	username
<b>Group Name Column</b>	Groupid
<b>Password Encryption Algorithm</b>	none
<b>Encoding</b>	SHA-256
<b>Charset</b>	UTF-8

Para configurar correctamente estos valores se ha tenido en cuenta la documentación de (Kalali, 2010) ya que si estos no se introducen correctamente será imposible acceder al servicio.



JAAS Context es el contexto del realm ya que se está utilizando uno de tipo JDBC se pondrá jdbcRealm. En JNDI se especifica el pool de la base de datos que contiene las credenciales de los usuarios una vez introducido se deben indicar las columnas y las tablas que las contienen. Para los datos de usuario tenemos la tabla users que se especifica con User Table. A continuación se le dirán las tablas que contienen el nombre de usuario y la clave con User Name Column y Password Column respectivamente. Ahora viene la tabla que se encarga de la asignación de grupos a los usuarios que irá en Group Table. El nombre de usuario es Group Table User Name Column y su grupo es Group Name Column.

Finalmente hay que configurar como se guardará el password. En este caso no se utiliza encriptado, esta información la facilitamos a través de Password Encryption Algorithm. El texto del password se debe codificar en SHA-256 y lo definimos en el campo Encoding. Finalmente se introduce la codificación de caracteres que se va a utilizar en Charset, en nuestro caso será UTF-8.

Con esta información el realm estará activo en cuanto se reinicie el servidor. Para utilizar el realm en el servicio hay que configurarlo en el archivo web.xml siguiendo las indicaciones de (Heffelfinger, 2014). Lo primero será especificar el realm y el tipo de autenticación que se utilizará, como ya hemos visto esta será BASIC y el realm es sparrowRealm, las líneas que se tienen que añadir entonces serán:

```
<login-config>
  <auth-method>BASIC</auth-method>
  <realm-name>sparrowRealm</realm-name>
</login-config>
```

A continuación se deben configurar los roles de seguridad. En nuestra aplicación solo hay un rol que se llamará USERS.

```
<security-role>
  <role-name>USERS</role-name>
</security-role>
```

Según (Jendrock, Cervera-Navarro, Evans, Haase, & Markito, The Java EE 7 Tutorial, Volume 2, Fifth Edition, 2014) también hay que editar el archivo glassfish-web.xml con el siguiente contenido:

```
<security-role-mapping>
  <role-name>USERS</role-name>
  <group-name>USERS</group-name>
</security-role-mapping>
```

Volviendo a web.xml es la hora de especificar que URL's del servicio requieren autenticación, esto se hará mediante el siguiente código:

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>NOMBRE</web-resource-name>
    <url-pattern>URL </url-pattern>
    <http-method>MÉTODO HTTP</http-method>
  </web-resource-collection>
  <auth-constraint>
    <description>DESCRIPCIÓN </description>
    <role-name>ROL</role-name>
  </auth-constraint>
</security-constraint>
```

Cada función que lo necesite utilizará una entrada security-constraint. Se pondrá un nombre a cada restricción con web-resource-name. La URL a proteger se indica mediante url-pattern. Con http-method elegimos el método con el que se va a llamar a la función pudiendo ser GET, POST, PUT o DELETE. Se puede añadir una descripción con description y una de las partes más importantes es el rol para el que se ha creado esta restricción que será role-name.

De esta forma cuando se quiera acceder a una URL protegida el sistema pedirá autenticación, al recibirla comprobará que el nombre de usuario y su clave son correctos en la tabla users a continuación, con la tabla, users\_groups sabrá si el usuario tiene el rol exigido por la restricción.

### 5.3.6 Servicios

Como ya hemos visto el servicio va a ser que proporcione acceso a los recursos de la aplicación a los clientes que se conecten a ella. Utilizará los BO para ello pero eso no significa que de un acceso directo a estos en todos los casos ya que ofrece algunas operaciones nuevas.

El servicio debe ser configurado en el archivo web.xml para que la aplicación sepa como utilizarlo. Al enlazar la librería Jersey se va a añadir la configuración necesaria para un servlet y se indicarán las librerías necesarias para que este funcione. A continuación se agregará el mapeado de este que lo que va a indicar en este caso es la URL principal desde la que colgará el servicio.

```
<servlet-mapping>
  <servlet-name>JAX-RS Servlet</servlet-name>
  <url-pattern>/rest/*</url-pattern>
</servlet-mapping>
```

Como se puede observar esta información va contenida en url-pattern y se indica que la URL raíz de nuestro servicio va a ser /rest.

Cada método que quede como público para los servicios debe especificar su path mediante la anotación @path (estos path serán los que cuelgan de la dirección /rest), en la que se pueden incluir los parámetros de entrada que se pueden recibir por la URL. El tipo de acceso que va a permitir que puede ser @POST, @GET, @PUT o @DELETE. Si va a consumir datos y su formato @Consumes(formato) o si los va a devolver @Produces(formato).

El formato de los datos utilizados se puede especificar con una cadena de texto o mediante la clase MediaType.

“application/json”  
mediaType.APPLICATION\_JSON

La estructura de URL's completa de esta aplicación será la que se puede ver en la siguiente ilustración:

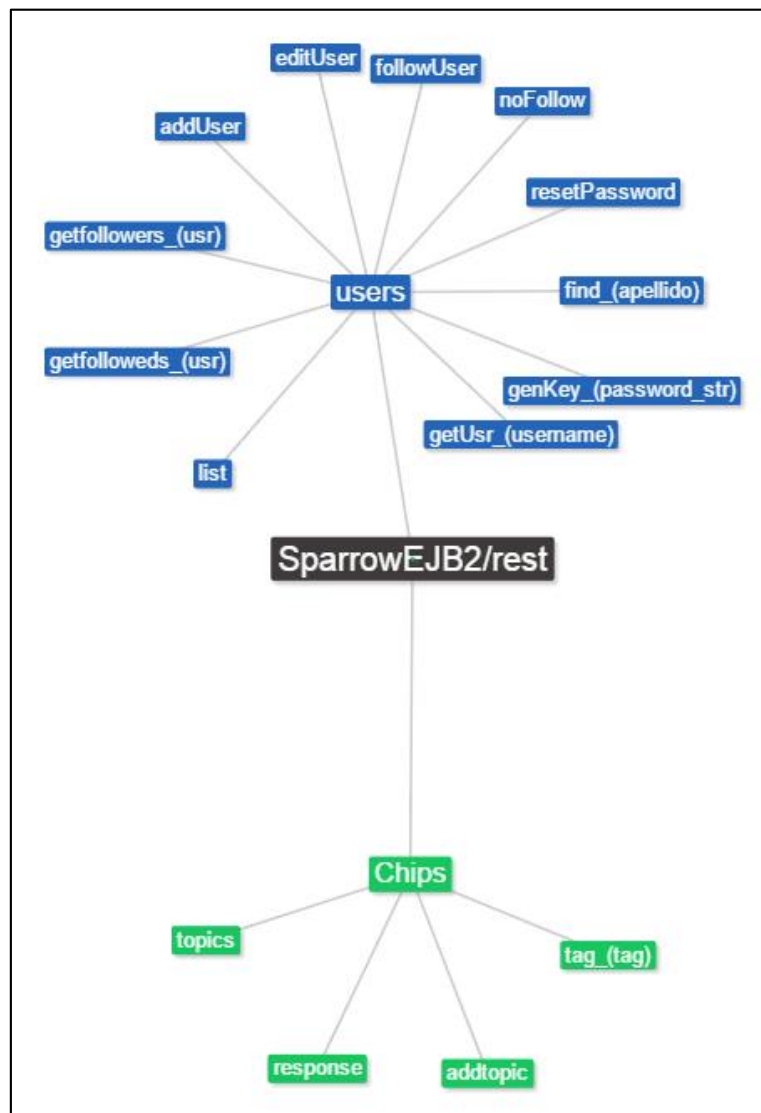


Ilustración 22

#### 5.3.6.1 JSON

Para poder hacer la conversión entre entidades en java y JSON se deben modificar estas. El primer paso será añadir la anotación `@XmlRootElement` que servirá para indicar que se puede convertir en XML. Esta anotación se añadirá a las clases `User`, `UsersGroup`, `Following` y `Chip`. La clase `FollowingPK` no lo necesita ya que se utiliza de forma interna y no accedemos a ella directamente.

Los métodos que utilicen algún tipo de relación con otra tabla llevarán la anotación `@XmlTransient`. En la clase `User` esta se aplicará a los métodos `getChips`, `getFollowers` y `getFolloweds`. En la clase `Chip` será para `getChips` solo. El resto de entidades no necesitarán esta anotación.

### 5.3.6.2 *ServicioUsers*

Este servicio se encargará de todas las operaciones relativas al control de usuarios y por lo tanto de las acciones necesarias para seguirlos. Responderá a llamadas dentro de la URL users.

La mejor forma de describir su funcionamiento es a través de los casos de uso, también se podrá ver el funcionamiento de los BO. Este servicio corresponde con los casos A2 hasta A5 y C1 a C4. El caso A1 es especial, cada cliente tiene que manejar a su modo como enviar los datos de autenticación al realizar llamadas al servicio.

#### **A2.** Registro de usuario

El cliente debe recoger toda la información relativa al usuario y enviarla al servicio en formato JSON a la dirección addUser. Se ha utilizado directamente la clase User de las entidades ya que acepta que solo se carguen los datos básicos sin especificar el resto.

Los datos recogidos se cargan en un objeto de tipo User y se guardan llamando a UserBo. Para asignar el usuario a un grupo se hace mediante UsersGroupBoRemote al que solo se le pasa el nombre de usuario ya que el BO le asignará por defecto el rol USERS.

#### **A3.** Recuperar clave

El usuario debe introducir su email y su nombre de usuario para que el sistema compruebe que son correctos y en ese caso devolver la clave guardada en password\_string ya que la columna password lleva la clave codificada.

Se enviará un objeto del tipo Password con esta información a la dirección resetPassword. Con el método recuperaPassword de UserBo se hará la comprobación y se devolverá en el campo email de un objeto Password la clave del usuario o un mensaje de error si los datos son incorrectos o este no existe. Que los datos se devuelvan así en lugar de en texto plano se debe a como se hacen las llamadas al servicio. La llamada utilizada en este caso es:

```
newPassword=target.request(MediaType.APPLICATION_JSON).post(Entity.entity(password,MediaType.APPLICATION_JSON),Password.class);
```

Al hacer la llamada se debe configurar con request o con response el tipo de datos que se van a utilizar pero solo se puede indicar uno de estos por lo que la información devuelta debe ser igual que la enviada.

#### **A4.** Gestión de datos del usuario

Para esta operación es necesario realizar dos llamadas. La primera será getUsr\_(username) a la que se le pasa por la URL el nombre del usuario, esta buscará el usuario mediante el BO de usuario y devolverá un objeto del tipo Users con la información básica del usuario.

La segunda llamada será a editUser que recogerá la información del usuario en un objeto User y la actualizará con userBo.

#### **A5.** Buscar usuario

En este caso se llamará a la dirección find\_(apellido) pasándole por URL el apellido del usuario que se quiere encontrar. Por medio de UserBo se recuperará una lista de objetos tipo User y se devolverá en JSON.

### **C1. Seguir usuario**

El path utilizado en este caso es followUser al que se le pasará un objeto del tipo Follows, esta clase identifica al usuario actual y al que se va a seguir pero con un formato un poco especial ya que el usuario actual se indica mediante el nombre de usuario y el usuario a seguir mediante su id. Esto se debe a como se devuelven las listas de usuarios seguidos y de seguidores.

Mediante esta información se buscarán los usuarios y se guardarán en un objeto User para luego pasárselos a otro del tipo Following que será utilizado por el BO de follow. En esta situación se tiene que indicar a los usuarios también el following para que sepan a quien sigue (el usuario actual) y quien el sigue (el usuario seguido), para esto se utilizará UserBo.

### **C2. No seguir a usuario**

Esta es la operación inversa a la anterior y se realiza mediante una llamada a noFollow. Igual que en el caso anterior se va a recibir un objeto del tipo Follows con el que se buscarán a los usuarios para instanciar un objeto Following que se pasará a FollowBo.

### **C3. Usuarios seguidos**

En la página principal de la aplicación hay una lista con los usuarios seguidos, para obtenerla se llamará a getFolloweds\_(usr) y se le pasará como parámetro el nombre del usuario actual. Este valor se utilizará para buscar el usuario mediante UserBo, la información sobre los usuarios seguidos está en la clase User que nos devolverá una lista de tipo Following con los usuarios.

Puesto que devuelve un tipo de datos que es una entidad y resulta pesada y puede dar problemas debido a las relaciones se va a convertir a una lista de tipo MiniUser que contiene la información requerida por la aplicación. Esta lista se devuelve entonces en JSON.

### **C4. Seguidores**

Este caso funciona exactamente igual que el anterior con la excepción de que ahora se preguntará al usuario los seguidores. Para recuperar esta lista el cliente debe utilizar el path getFollowers\_(usr) con el nombre de usuario actual como parámetro en la URL.

#### **5.3.6.3 ServicioChips**

Este otro servicio se encarga del manejo de chips. Funcionará desde la URL chips y como en el servicio anterior se va a analizar a partir de los casos de uso los cuales serán de B1 a B4.

### **B1. Mostrar temas de discusión**

En la página principal se mostrarán los temas de los que cuelgan los chips o respuestas. Los temas serán chips con thread null o lo que es lo mismo, que no responden a ningún otro chip. Por ello será necesario recuperar una lista que solo contenga ese tipo de chips.

Para recuperar la lista se hará una llamada en la dirección topics. Esta por medio de ChipBo recupera los chips con thread null, esta operación la realiza el DAO de Chips con una búsqueda específica definida en la entidad Chips.

```
@NamedQuery(name="Chip.findThemes",query="SELECT c FROM Chip c WHERE  
c.chip=NULL")
```

Como se puede comprobar en el código la condición para devolver el chip es que su chip sea NULL.

Se devolverá una lista de objetos del tipo Topics que es una versión mínima de la entidad Chip. Puesto que el BO devuelve una lista del tipo Chip será necesaria una conversión a este tipo de dato reducido.

## **B2. Mostrar chips por tag**

Si el usuario elige una entrada de la lista anterior obtendrá el tag del chip, que es el título del tema. Pasándole este tag por url a la dirección tag\_(tag) se hará una búsqueda de todos los chips relacionados con este. Esto se hace por medio del BO de chips que nos devolverá una lista de tipo Chip, una vez mas hay que hacer una conversión a un tipo de datos menos pesado por lo que en este caso se utilizará el tipo Chips.

## **B3. Crear tema**

Los usuarios de la aplicación tienen la posibilidad de crear nuevos temas. Para ello se realizará una llamada a la dirección addtopic que consume un objeto del tipo Topics en JSON.

El servicio recoge este objeto y lo convierte a otro del tipo Chip para añadirlo a la base de datos mediante el BO. Para especificar que el thread es null no se cargará el atributo chip con setChip.

## **B4. Contestar un chip**

En la lista de chips de respuesta es posible añadir una respuesta nueva. Esto se hará mediante una llamada al path response que recibe un objeto del tipo Chips en JSON.

El servicio se encargará de convertirlo a tipo Chip y esta vez sí que se debe especificar el Chip al que responde este. Para ello se obtiene el chip del que cuelga haciendo una búsqueda por id, el id de este se ha guardado previamente en el objeto Chips recibido.

Ahora ya se ha obtenido el chip del tema. Se guardan los datos recibidos en un Chip nuevo y le pasamos el chip del tema con setChip para que este ya sea una respuesta.

### **5.3.6.4 Otras funciones del servicio**

Los servicios cuentan con algunas funciones extra que no quedan reflejadas en los casos de uso se puede decir que son funciones de utilidad para los clientes.

En el servicio de usuarios se puede encontrar la dirección list que devolverá una lista de usuarios. Una utilidad muy importante que ofrece es en el path genkey al que se le pasará el password en texto plano por la url y devolverá también en texto el password codificado en SHA-256, es una función muy útil para el cliente a la hora de añadir usuarios nuevos al sistema.

### **5.3.6.5 Definiendo la seguridad**

Una vez están definidas todas las funciones del servicio hay que añadir a web.xml las que requieran autenticación. Teniendo en cuenta los casos de uso y las funciones utilidad las URL seguras son:

- getUsr (GET)
- find (GET)
- followUser (POST)
- noFollow (POST)
- getFolloweds (GET)

- getFollowers (GET)
- topics (GET)
- tag (GET)
- addtopic (POST)
- response (POST)
- list (GET)
- genkey (GET)

Añadiendo las entradas security-constraint necesarios con esta información ya tendremos un servicio seguro.

#### 5.3.6.6 CORS

Como ya vimos si el cliente que se va a conectar a nuestro servicio lo hace por medio de HTML y javascript nos encontraremos con el problema provocado por CORS. Para solucionarlo se ha creado un filtro Jersey. Este se llama FiltroAjax.

El filtro Jersey da acceso a ContainerRequestContext y a ContainerResponseContext que son clases que contienen información específica sobre Request y Response como la URI, cabeceras...

En nuestro caso vamos a recuperar las cabeceras relativas a Response mediante:

```
MultivaluedMap<String, Object> headers = responseContext.getHeaders();
```

Esta línea devuelve las cabeceras en un MultivaluedMap. Ahora hay que añadir las cabeceras que se vieron en 3.5 para que el navegador no de errores de CORS. Como es una lista añadir las cabeceras es muy fácil. Este filtro responde a todas las llamadas que se hagan al servicio devolviendo estas cabeceras.

Para que el filtro funcione hay que declararlo en el archivo web.xml en la sección JAX-RS.

```
<init-param>  
  <param-name>jersey.config.server.provider.classnames</param-name>  
  <param-value>es.uv.bd.sparrow.service.FiltroAjax</param-value>  
</init-param>
```

#### 5.3.7 Cliente Java

Para comprobar que el servicio es compatible se tiene que probar con varios clientes, uno de estos se escribirá en Java. Es una aplicación web normal que utiliza Servlets y páginas jsp obteniendo los datos necesarios mediante llamadas al servicio web.

##### 5.3.7.1 Seguridad y Login

El primer caso de uso se dejaba para el cliente puesto que la seguridad en el servicio depende del servidor. Cada vez que se hace una llamada a una función del servicio que necesita autenticación por parte del usuario, el cliente debe enviar sus datos en la cabecera http, el servidor comprueba que son correctos si es así dará acceso a los recursos del servicio, si no, redirigirá a una página que muestra un mensaje de error.

Para ello el Servlet Login recoge las credenciales del usuario de la página index.jsp, concatena el nombre de usuario y el password separados por el carácter dos puntos ":" y codifica esta cadena en Base64. Estos datos se guardan en el Bean de sesión UserBean:

UserBean
nombre:String pass:String b64:String

Este será llamado por el resto de servlets para obtener los datos del usuario. Las llamadas al servicio se hacen mediante la clase Java-RS WebTarget. Un ejemplo de llamada sería:

```
UserBean userBean = (UserBean)
request.getSession().getAttribute(USERBEAN_ATTR);

ClientConfig clientConfig=new ClientConfig();

clientConfig.register(Headers.class);

Client client=ClientBuilder.newClient(clientConfig);

WebTarget targetTopics =
client.target("http://localhost:8080/SparrowEJB2/rest/chips/topics");

ArrayList<Topics> listaTemas =
targetTopics.request(MediaType.APPLICATION_JSON).header("Authorization", "Basic "+userBean.getB64()).get(new GenericType<ArrayList<Topics>>(){});
```

Lo primero que se hará es recuperar el Bean que contiene los datos de usuario. Ahora se configura y se crea el cliente que se pasará al WebTarget con la URL a la que se quiere llamar.

La última línea es la llamada en sí. Podemos ver que en este caso hacemos una petición a topics por lo que en esta línea se cargan datos en una lista de Topics. Como se puede ver se añade la cabecera de seguridad Authorization que lleva como datos Basic y el usuario y la clave en Base64. Entonces, esta es la utilidad del Bean que se acaba de crear y este es el sistema que hay para llamar al servicio. Se puede observar que hay código que indica que se envía JSON, en este caso no era necesario. También al final de la línea está la configuración de cómo se van a recibir los datos, se ha hecho una petición GET que quiere recibir una lista.

Otro detalle es que aquí ya ha entrado Jersey. Se están utilizando directamente clases Java para la recepción de datos, solo se ha tenido que especificar que los datos deben viajar en JSON y Jersey se encarga de hacer las conversiones necesarias.

#### 5.3.7.2 Casos de uso

Puesto que ya estamos en el cliente se van a aplicar los casos de uso de la aplicación original directamente ya que se está buscando el mismo funcionamiento.

##### A1. Acceso a zona privada



En la página index.jsp tenemos un formulario en el que introducir el nombre de usuario y la clave. Como hemos visto estos datos los recoge un Servlet y los guarda en un Bean de sesión para ser utilizados más tarde. El acceso a la zona privada en sí es posible. Se podría intentar cargar la página mainPage.jsp pero no tendría datos que mostrar.

Al cargar MainMenu este Servlet hará llamadas al servicio web para recuperar los temas, los seguidores y los usuarios seguidos por lo que se accede a zonas privadas del servicio y se requiere la autenticación. Si los datos enviados en la cabecera son correctos, Glassfish “nos dejará pasar” al servicio y este devolverá los datos que el servlet enviará a la página jsp. Este proceso se puede observar más rápidamente en el siguiente diagrama:

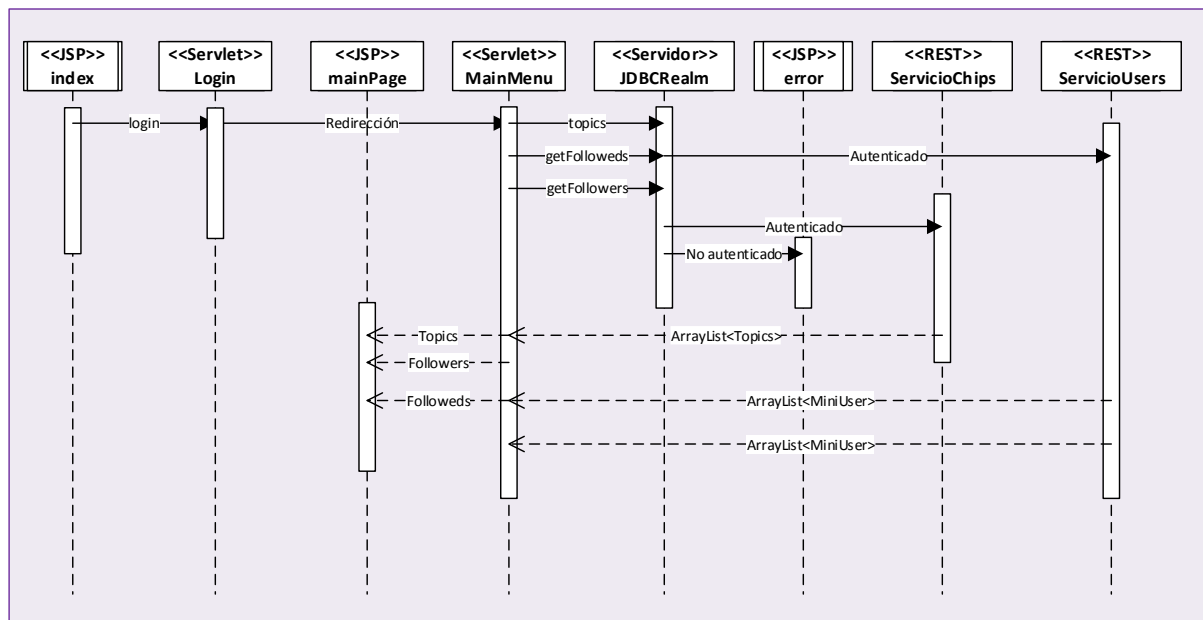


Ilustración 23

Puesto que es Glassfish el encargado de manejar la seguridad al hacer Login, si los datos no son correctos será el propio servidor el que informe de que se ha producido un error al llamar al servicio con datos incorrectos. Esta comprobación no la hace el cliente.

## A2. Registro de usuario

En este caso de uso no se necesitan funciones de la zona privada del servicio por lo que las llamadas son más cortas al no tener que incluir cabeceras de autenticación. El servlet RegistraUsuario recogerá los datos de un formulario en registerUser.jsp y los enviará al servicio. El usuario debe introducir la contraseña dos veces y por medio de javascript se compara si las dos coinciden mostrando un mensaje. Si las dos coinciden se activará el botón de envío.

Hay que tener en cuenta que la clave debe enviarse tanto en texto plano como codificada en SHA-256, puesto que se está utilizando Java es posible hacer esta conversión en el propio cliente ahorrando llamadas al servicio y evitando posibles problemas de seguridad al mandar los datos. Al terminar la operación se volverá a la página index.jsp.

## A3. Recuperar clave

El usuario tiene que introducir en un formulario de la página resetPassword.jsp su email y su username. Estos datos se envían al servicio en un objeto Password y este devolverá un texto plano

con el password o con un mensaje de error. El servlet RecuperaPassword recibe los datos y redirecciona a getPassword que muestra el texto recibido directamente ya que es el servicio el que se encarga de dar el mensaje de error si los datos enviados no eran correctos.

#### A4. Gestión de datos del usuario

En esta ocasión van a ser necesarias dos llamadas a zonas protegidas del servicio. En doGet el servlet pedirá los datos de usuario al servicio y cargará un formulario con estos. En doPost recibirá la información de este formulario y la enviará otra vez al servicio para que este la actualice. Se puede observar este proceso en la ilustración 24.

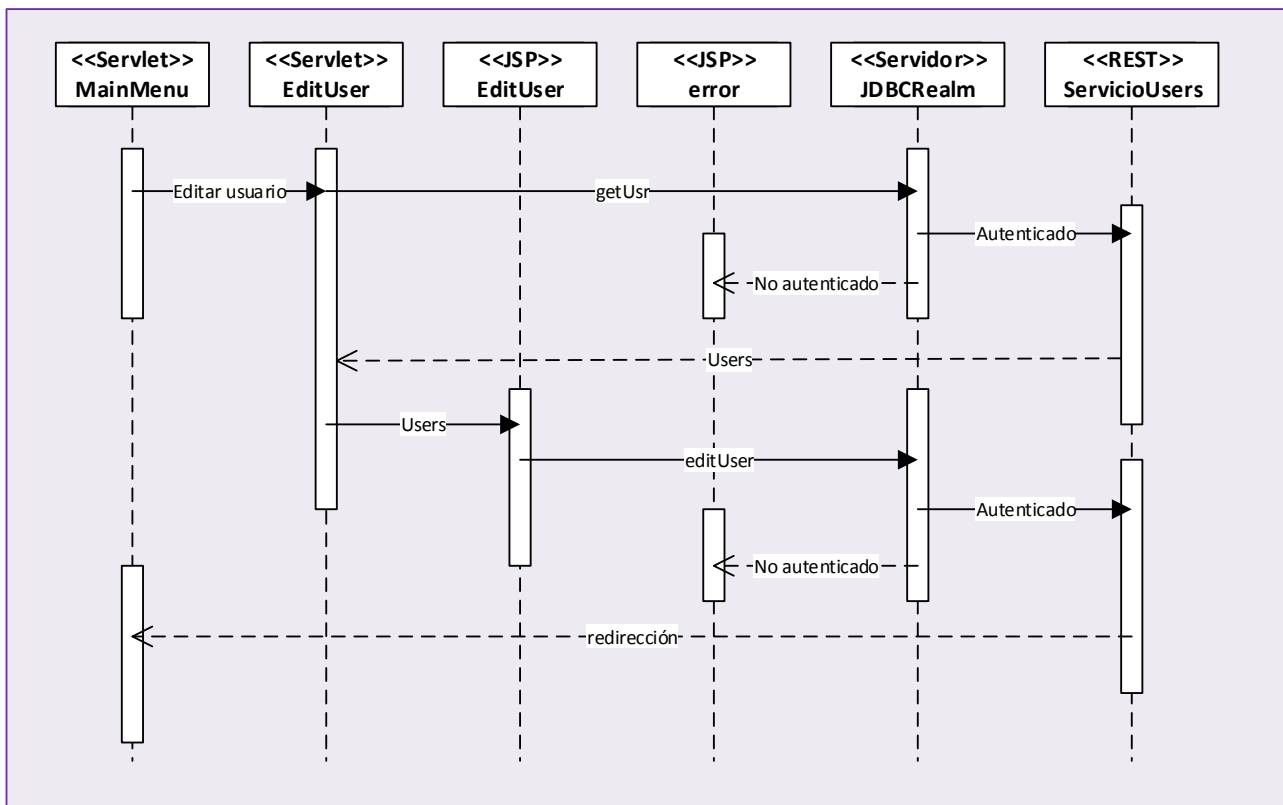


Ilustración 24

#### A5. Buscar usuario

Para buscar un usuario el Servlet FindUser envía al usuario a un formulario de búsqueda (searchUsr.jsp). Accediendo a la parte con autenticación del servicio enviará los apellidos recogidos del formulario y recibirá una lista de Users que mostrará en searchUsrOK desde la cual se podrán añadir como seguidos por medio de un enlace que llama a FollowUser pasándole el id del usuario.

#### B1. Mostrar temas de discusión

Este caso ya se ha visto en A1 pero hay que puntualizar que a la hora de representar los temas en la lista de mainPage.jsp se mostrará el nombre del tag, siendo este texto un enlace a ViewByTag al cual se le pasa como parámetro el tag del tema. También se mostrará el autor del tag.

#### B2. Mostrar chips por tag

Al pulsar sobre el enlace de un tema se llama al servlet ViewByTag que lleva el parámetro tag para saber que tema buscar. Una vez más se hace una llamada a la parte privada del servicio que devolverá una lista de Chips que recoge el servlet y mostrará en la página viendoPorTag.jsp.

Esta página incluye un botón para crear nuevas respuestas que invocará a ChipResponse enviándole por el parámetro chipActual el id del chip.

### **B3. Crear tema**

Para este caso de uso se utilizará el servlet AddTopic. Este mandará al usuario a la página nuevoChip.jsp de la que recogerá el tag y el texto. Con esta información se crea un objeto Tema que se envía a la parte privada del servicio.

### **B4. Contestar chip**

El servlet ChipResponse va a obtener por medio de la URL el parámetro chipActual que es el chip para el que se va a escribir la respuesta. En la página responderChip se escribirá el texto de la respuesta. El servlet al recibir estos datos cargará un objeto de tipo Chips que contiene el texto, el autor y el chip del tema o thread del chip. Este objeto se mandará a la zona privada del servicio.

### **C1. Seguir usuario**

En la página principal y en la página de búsqueda de usuario hay botones para poder seguir a un usuario concreto. Para ello el servlet FollowUser recibe como parámetro el id de este. Como ya se vio en los casos de uso para el servicio se cargará un objeto Follows con el username del usuario actual y el id del que se va a seguir, esta es la razón por la que esta clase utiliza este tipo de formato.

Finalmente, para completar la operación se llama a la parte protegida del servicio y se envía.

### **C2. No seguir a usuario**

La lista de usuarios seguidos cuenta con botones para dejar de seguir a un usuario. Estos hacen una llamada al servlet NoFollow pasándole el id del usuario. Este lo recoge y lo carga en un objeto Follows junto al nombre del usuario actual y lo envía a la zona privada del servicio.

### **C3. Usuarios seguidos**

En mainPage hay una lista con los usuarios seguidos. Para obtenerla, el servlet MainPage hace una llamada a la función protegida getFolloweds que devuelve una lista de la clase MiniUser. Esta se envía a la página la cual los añade en una lista indicando el nombre del usuario y poniendo al lado un botón con un link para dejar de seguir al usuario (caso C2).

### **C4. Seguidores**

Otra lista que podemos encontrar en mainPage es la de seguidores. Para obtenerla el proceso es idéntico al del caso anterior con la diferencia que esta vez se llama a la función getFollowers del servicio el cual devolverá otra lista de MiniUser. En la página se mostrará el nombre de usuario junto a un botón para poder seguirlo (caso C1).

#### 5.3.7.3 Entidades

Para poder comunicarse con el servicio REST será necesario que el cliente cuente con las mismas entidades que este por lo que se han añadido las siguientes clases:

- Chips
- Follows
- MiniUser
- Password
- Topics
- Users

#### 5.3.7.4 FiltroHeader

Este es un filtro que se encarga de recoger las cabeceras de Request y Response y las saca por consola.

#### 5.3.7.5 Diseño de la web

El cliente debe ser lo mas parecido posible a la aplicación original, puesto que esta fue programada también como un aplicación web con páginas en formato jsp la adaptación ha sido directa y ha podido compartir con esta las imágenes y los archivos CSS.

La página index.jsp es:

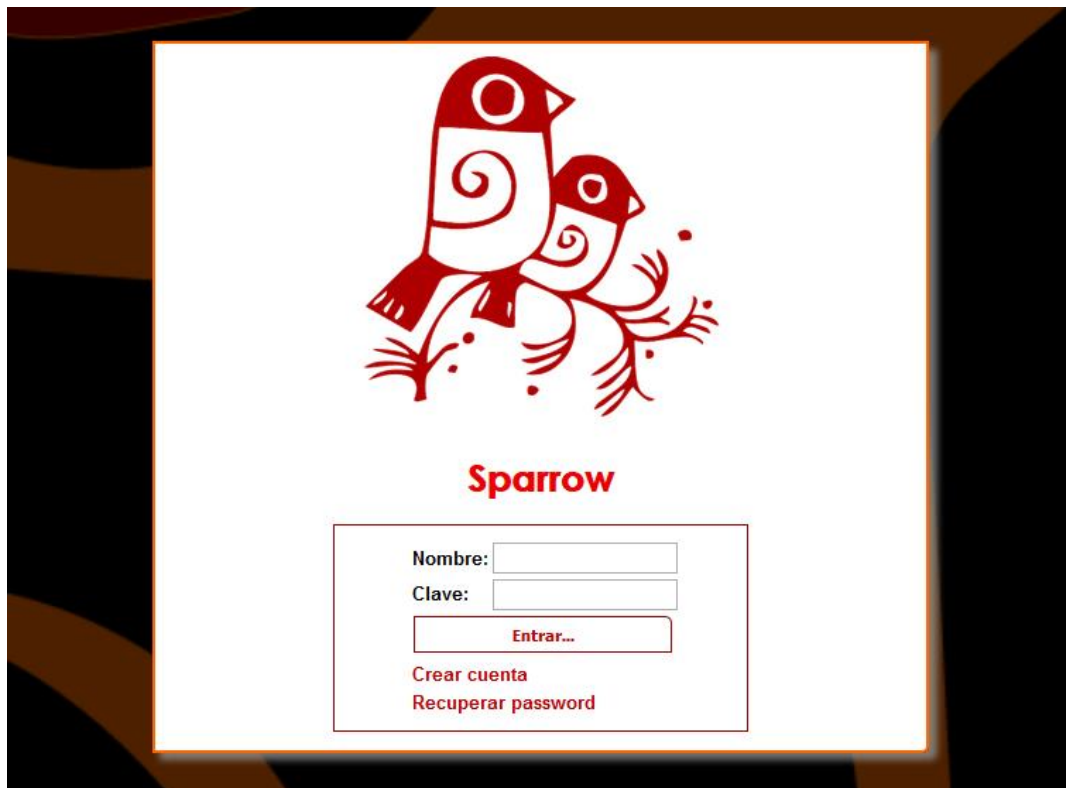


Ilustración 25

En la siguiente ilustración se puede apreciar la página registerUser.jsp.

**Registrarse como nuevo usuario**

Nombre:

Apellidos:

Sexo:

Idioma:

Email:

Username:

Password:

Repite password:

Ilustración 26

Una vez se ha registrado el usuario se redirige a:

**Registro completado**

Te has registrado correctamente

[Volver](#)

Ilustración 27

La página para recuperar la clave es resetPassword.jsp.

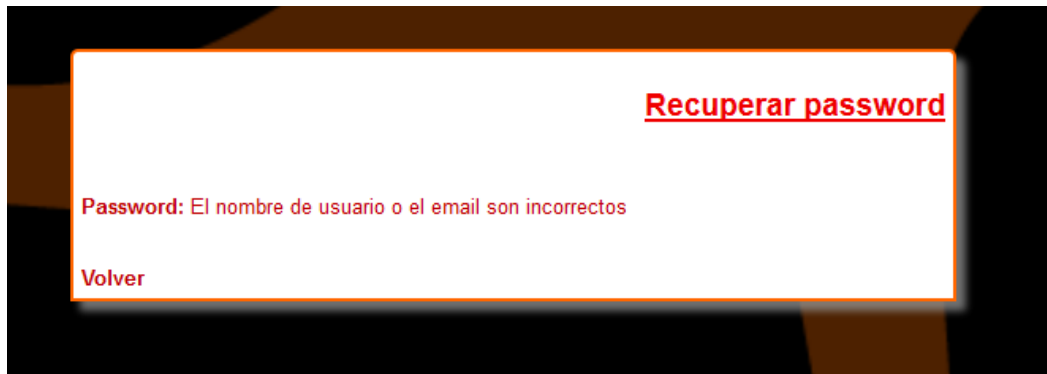
**Recuperar password**

Usuario:

Email:

Ilustración 28

La cual redirige a getPassword.jsp:



**Recuperar password**

**Password:** El nombre de usuario o el email son incorrectos

**Volver**

Ilustración 29

En la ilustración 30 se puede observar la página principal de la aplicación, mainPage.jsp. En las listas de Seguidores y Siguiendo se pueden observar los botones para añadir o quitar usuarios.



**Sparrow** Bienvenido usuario01

Buscar usuarios Crear tema Preferencias

**Seguidores**

loloA +

**Siguiendo**

loloA +

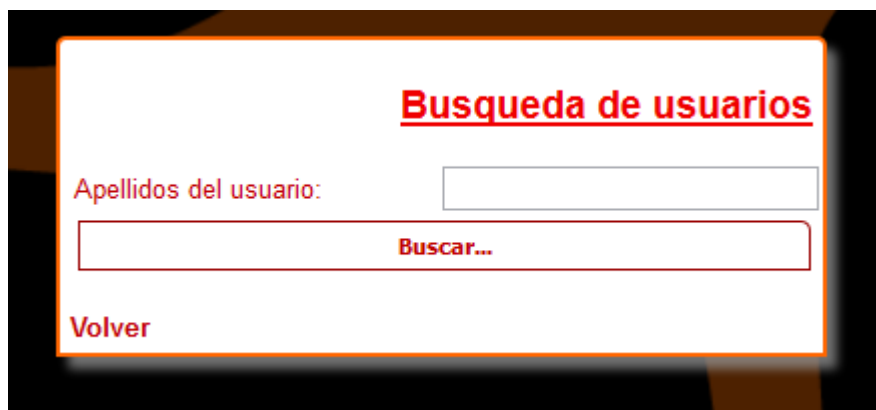
koko +

**Últimos chips**

aaaa	usuario01
tema 2	usuario01
prueba 3	usuario01
tema seguro	usuario01
tema grupos	grupos

Ilustración 30

En searchUsr.jsp la aplicación muestra la ventana de búsqueda de usuarios.



**Busqueda de usuarios**

**Apellidos del usuario:**

**Buscar...**

**Volver**

Ilustración 31

Si se encuentran usuarios con los apellidos buscados se mostrará la siguiente página (searchUsrOk.jsp) en la que se puede ver el nombre completo y el username de los usuarios encontrados, además del botón para añadirlo a la lista de seguidos.

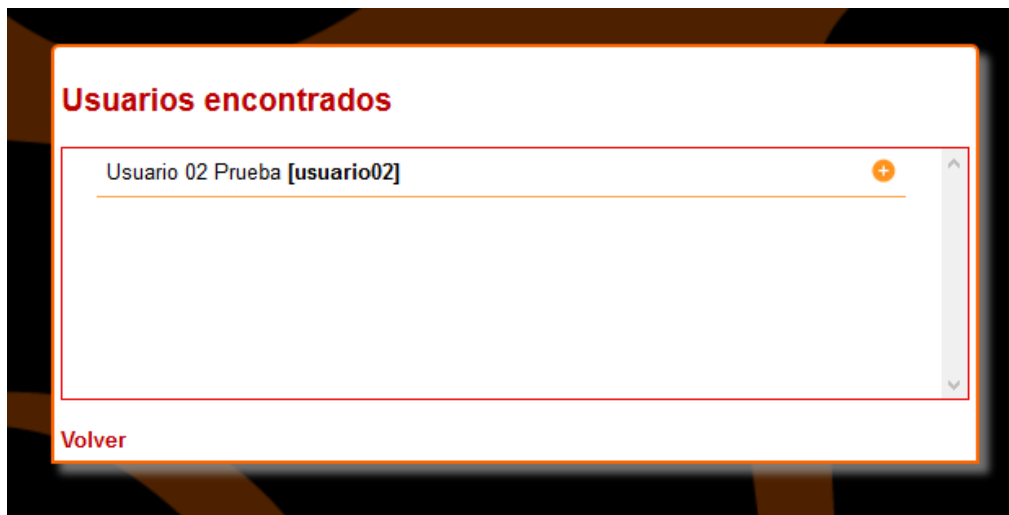


Ilustración 32

A la hora de crear un nuevo tema se cargará la página nuevoChip, esta se comparte con la opción de escribir respuestas y es el servlet el que se encarga de guardar la información como corresponda.

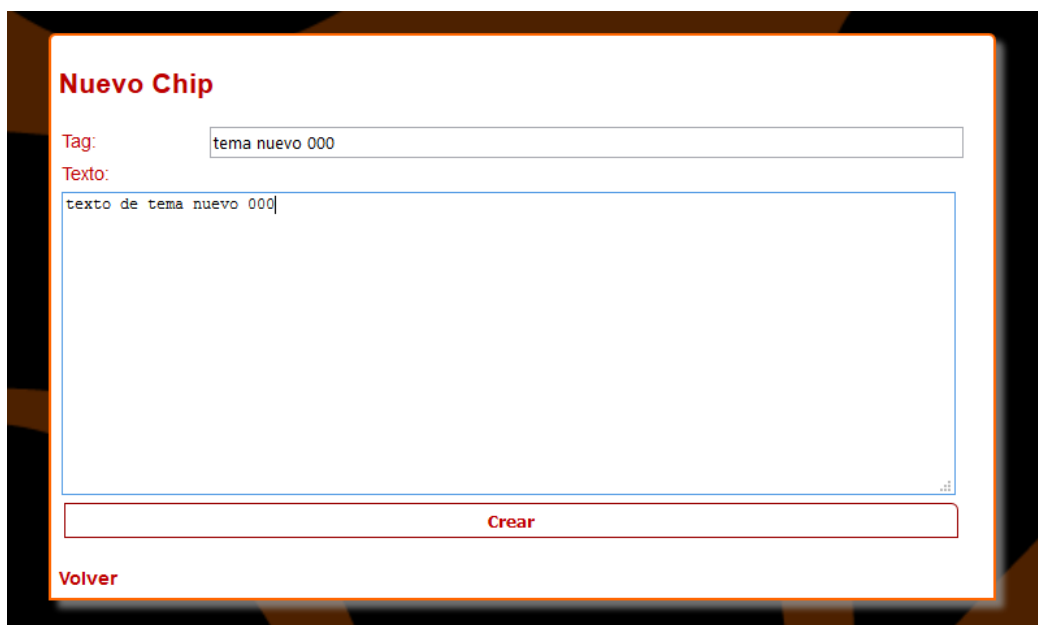


Ilustración 33

Esta es la ventana de edición de preferencias de la cuenta (editUsr.jsp). Los campos nombre de usuario y email no son editables.

**Preferencias**

Nombre de usuario: usuario01

Password: 12monos

Email: kkk@kkk.es

Nombre: Usuario 01js

Apellidos: Pruebajs

Sexo: M

Idioma: German

**Actualizar**

**Volver**

Ilustración 34

Finalmente, en la ilustración 35, tenemos la lista de chips respuesta a un tema.

**Chips para: aaaa**

aaaa [usuario01] **Responder Chip**

repuesta aaa [usuario01] **Responder Chip**

repuesta aaa2 [usuario01] **Responder Chip**

asd [usuario01] **Responder Chip**

repuesta bbb [usuario01] **Responder Chip**

**Volver**

Ilustración 35

### 5.3.8 Cliente HTML

Otro cliente que se utilizará para comprobar la compatibilidad del servicio REST es en formato HTML con javascript. Para la recepción de datos se utilizará AJAX por medio de jQuery. Este cliente presenta más dificultades ya que nos enfrentamos al problema que presenta CORS pero tras la inclusión de filtro que se vio en 5.3.6.6 ya podemos utilizar el servicio de forma normal.

En este cliente entonces se van a utilizar las librerías jQuery para el envío y recepción de datos en Ajax y para aprovechar sus capacidades gráficas y de acceso al DOM. También se va a incluir la librería js.cookie.js que servirá para controlar cookies.

Además de las páginas y el contenido javascript en ellas se ha escrito el archivo sesion.js que se encarga de controlar los datos de usuario durante la sesión, datos.js que se encarga del tráfico de



datos en Ajax además de contar con algunas utilidades extra y clases.js que contiene las clases necesarias para poder comunicarse con el servicio.

Como ya se ha dicho las clases están definidas en el archivo clases.js. A la hora de enviarlas habrá que convertirlas JSON. Para ello se utiliza la función javascript `JSON.stringify()` a la que se le pasará la clase que queremos convertir.

#### 5.3.8.1 Ajax

Mediante jQuery las llamadas Ajax para el envío y recepción de datos son fáciles de hacer no hay mas que hacer una llamada a `$.ajax` y cargar los datos necesarios ya que es un objeto. Por ejemplo:

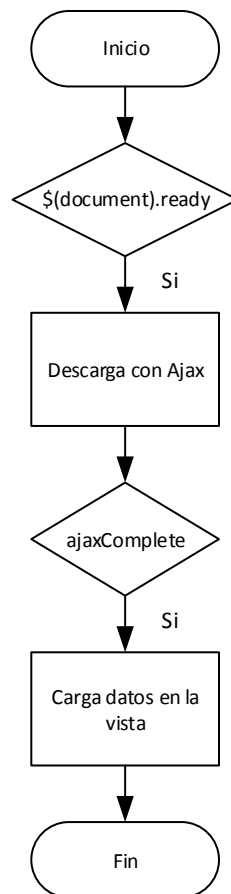
```
$.ajax({
  headers: {
    'Accept': 'application/json',
    'Content-Type': 'application/json'
  },
  'type': 'POST',
  'url': url,
  'data': data,
  'dataType': 'json',
  'complete': function(),
  'error': function () {
    alert("Error enviando datos");
  }
})
```

En el atributo Headers se van a especificar las cabeceras que se quieren mandar, en nuestro caso se van a añadir las correspondientes con el formato de datos y la cabecera de autorización si la llamada a realizar lo requiere. En type se indica el tipo de llamada http, nuestro cliente solo utilizará GET y POST. En url se pondrá la dirección del servicio a la que se quiere acceder. Hay que volver a especificar el tipo de datos mediante dataType.

El problema de Ajax o de una llamada http normal es que es asíncrona. Esto significa que no podemos hacer una función que devuelva los datos recibidos porque hasta que no se ha terminado la transferencia estos no existen. Cuando esta a terminado se ejecutará la función especificada en el atributo complete o success y esta será la encargada de mandar el resultado a donde sea necesario. En caso de que haya algún error en la operación se ejecutará la función especificada en error.

Otro detalle a tener en cuenta con Ajax es que no se puede hacer una redirección directa a otra página cuando se ejecutan success o complete porque los datos se perderán. Esto se ha solucionado mostrando los datos obtenidos en la misma página superponiendo un div al contenido original.

Para asegurarse de que se han recibido los datos el proceso será el siguiente:



Como se está utilizando jQuery se aprovecha el método `$(document).ready` que no se ejecutará hasta que se haya cargado todo el código HTML. En ese momento se llamará a las funciones que cargan o descargan datos mediante Ajax. Cuando todas las operaciones de transferencia han terminado jQuery dispara `ajaxComplete` por lo que es el momento de utilizar la información recogida y cargarla en las secciones de la página apropiadas.

#### 5.3.8.2 *datos.js*

Antes de entrar en el código del cliente sí es necesario hacer un repaso a los archivos que contienen la lógica del programa. Aunque este archivo no es una clase ya que corresponde más con programación estructurada puede ayudar un diagrama de clases para ver su contenido. Ya que este archivo se compone de un conjunto de funciones y procedimientos y sería parecido a una clase.

datos.js
<pre> var datos=null var clave=null var devuelto=null var seguidos=null var seguidores=null </pre> <hr/> <pre> getData(donde) getSeguidos(usr) getSeguidores(usr) setData(url,data,funcion) setDataAuth(url,data,funcion) setGetData(url,data,funcion) dameTemas() buscaUsuario(apellidos) actualizaUsuario(usuario) cargaUsuario(username) nuevoUsuario(usr) respondeChip(texto,autor,id) nuevoTema(tag,text,user) seguir(miUsuario,idSeguido) noSeguir(miUsuario,idSeguido) generaClave(clave) ventanaVolver() resetPass(nombre,email) verPass(res) dameParam(param) quitaEspacio(cadena) </pre>

Como ya se ha comentado en 5.3.8.1 las funciones de llamada no pueden devolver los datos así que tenemos una serie de variables globales iniciadas a null en las que se depositará la información recibida.

La función `getData` es general, servirá para hacer llamadas a la url introducida en la variable `donde`, el resultado de la operación se guarda en la variable `datos`, esta lleva incluidas las cabeceras de autorización. `setData` sería la función contraria, enviará la información contenida en `data` a la url especificada y ejecutará la función que le indiquemos. `setDataAuth` hace lo mismo que la función anterior pero se utilizará para llamadas a la parte privada del servicio. La última función general sería `setGetData` que puede enviar y recibir datos.

Las funciones `getSeguidores` y `getSeguidos` son similares a las anteriores. Se podría haber utilizado una de las anteriores pero se han separado porque así solo hay que pasarles el nombre de usuario para que llene las variables `seguidores` y `seguidos`.

Desde `dameTemas` hasta `noSeguir` todas estas funciones utilizarán a las anteriores con los valores apropiados para que cumplan su función.

El procedimiento `ventanaVolver` será el encargado de mostrar un div en el que se representará la respuesta a la acción requerida por el usuario. `verPass` tiene el mismo funcionamiento pero solo se utiliza para que el usuario vea su password.

Para obtener los parámetros que se pasan por URL se ha escrito la función `dameParam` que devolverá directamente el resultado.

Cuando se estaba probando el cliente se pudo comprobar que si se envían parámetros por URL que contengan espacios como por ejemplo:

`http://localhost:8080/SparrowEJB2/rest/chips/tag_tema numero 1`

Al recuperar el valor del parámetro tag este será: tema. Recortará todo lo que haya detrás del espacio. La solución entonces es cambiar los espacios por %20 de forma que ahora el enlace es:

`http://localhost:8080/SparrowEJB2/rest/chips/tag_tema%20numero%201`

Por lo que el resultado a la hora de leer el parámetro será: tema numero 1.

#### 5.3.8.3 Volviendo de Ajax

Al hacer una llamada al servicio por Ajax se ejecutará una función cuando esta haya terminado. Como ya se ha visto si se hace una redirección a otra página los datos se pierden por lo que la mejor solución es mostrarlos en la propia página mediante un div superpuesto al contenido original.

Para ello se ha escrito la función `ventanaVolver` que con una animación tapará el contenido de fondo con un div semitransparente que ocupa toda la ventana y otro div preparado para que el usuario vea el resultado de la operación.

Las páginas que necesiten esta utilidad necesitan declarar estos dos divs en la sección body.

```
<div class="tapar" id="tapa"></div>
<div class="ventanaVolver" id="volver"></div>
```

En muchos casos desde esa propia función o al dispararse `ajaxComplete` se cargarán los datos dentro de estos divs por medio de jQuery.

#### 5.3.8.4 Seguridad y Login

El principal problema que se nos presenta con este cliente es que funciona con HTML y javascript puro sobre un navegador así que al contrario del cliente Java no podemos tener un Bean de sesión con los datos del usuario. Así que la posibilidad por la que se ha optado es el uso de cookies.

Para ello se van a utilizar las funciones del archivo `sesion.js`. Una vez más este archivo no contiene una clase pero un esquema puede ayudar:

sesion.js

```
codifica(cadena)
creaSesion(usr,pass)
dameUsr()
getAuth()
```

En la página `index.html` al pulsar el botón de envío se recogerá el nombre de usuario y la clave y se llama la función `creaSesion` pasándole estos datos. Esta función recoge los datos los concatena para que queden en el conocido formato `usuario:password` y mediante `codifica` lo convierte a Base64, esta operación se hace mediante la función javascript `window.btoa`. Con toda esta información se

van a crear dos cookies, sparrowData y sparrowUsr. Una que contiene el nombre de usuario y otra con la cadena que se ha convertido a Base64. El resultado es el siguiente:

Name	sparrowData	Name	sparrowUsr
Value	dXN1YXJpbzAxOjEybW9ub3M=	Value	usuario01
Host	localhost	Host	localhost
Path	/	Path	/
Expires	Thu, 03 Sep 2015 11:00:49 GMT	Expires	Thu, 03 Sep 2015 11:00:49 GMT
Secure	No	Secure	No
HttpOnly	No	HttpOnly	No

Ahora con las funciones dameUsr y getAuth se pueden obtener el nombre de usuario y los datos de autenticación respectivamente. Así cuando se haga una llamada al servicio que necesite la cabecera de autorización solo hay que llenar una cadena de esta manera:

```
var auth = 'Basic ' + getAuth();
```

Ahora solo es necesario añadir la cabecera authorization y darle el contenido de esa cadena. A la hora de hacer llamadas que requieran el nombre del usuario como por ejemplo getSeguidos solo hay que utilizar la función dameUsr.

#### 5.3.8.5 Casos de uso

##### A1. Acceso a la zona privada

El proceso de Login ya se ha visto en 5.3.8.3. Este se hace a través de la página index.html. Como en el servicio Java no es un Login en sí, solo es una forma de obtener los datos de autenticación del usuario para poder realizar llamadas al servicio REST con estos.

##### A2. Registro de usuario

Para añadir un usuario nuevo a la aplicación se utiliza la página registraUsuario. En esta hay que introducir la clave dos veces y se comparan las claves, si estas coinciden el botón de envío se activa. Al pulsar este botón se llamara a la función generaClave. Esta recoge la clave que ha escrito y llama a la función genKey del servicio para que se la devuelva codificada en SHA-256.

Una vez tenemos la clave codificada se recoge el resto de datos del formulario y los guarda en un objeto de tipo Usuario. Este se pasará a la función nuevoUsuario que lo envía mediante la función setData.

Al terminar la operación se llama a ventanaVolver que muestra un mensaje que el usuario ha sido registrado en la aplicación.

##### A3. Recuperar clave

Este proceso se hace en la página resetPass que recoge el nombre de usuario y su email y los envía a la función resetPass que instancia un objeto de tipo Password que recibe estos datos.

A continuación se enviará el objeto al servicio a la dirección `resetPassword` mediante `getSetData` ya que espera a que se devuelva un resultado. Este resultado se mostrará en pantalla con la función `verPass`.

#### **A5. Buscar usuario**

La búsqueda de usuarios se hace desde la página `buscaUsuario` que tiene un formulario que pide los apellidos de los usuarios a buscar. Se llamará a la función `buscaUsuario` dándole los apellidos, esta llamará a la dirección `find` con `getData`.

Al lanzarse `ajaxComplete` se recogerá la información depositada en la variable `datos` y se pondrá en una lista que va dentro de un `div` que se cargará con una animación.

#### **B1. Mostrar temas de discusión**

En la página principal de la aplicación, `mainPage`, habrá una lista con los últimos temas. Para obtener los datos en el momento en que se lanza la función `$(document).ready` se llama a `dameTemas` que hace una llamada a `topics` con `getData`.

Una vez se ha terminado la transferencia y se carga `ajaxComplete` se recogen los datos y se añaden a la lista que hay preparada para ello.

#### **B2. Mostrar chips por tag**

Al pulsar sobre un tema de la lista se accederá a `verPorTag.html`. Al cuando el documento esté cargado se hará una llamada `tag` pasándole el `tag` que se ha obtenido mediante `dameParam`. Al terminar de cargar los datos se ejecutará `ajaxComplete` se cargarán los temas recibidos en la variable global `datos` mostrando el texto del chip, el autor y un botón para poder responder a este.

#### **B3. Crear tema**

Para crear un tema nuevo se accederá a `crearTema`. En esta página se recoge el `tag` y el texto del tema en un formulario al pulsar sobre el botón de envío se llamará a la función `nuevoTema` que recoge esta información y el nombre de usuario que lo ha escrito.

La función `nuevoTema` va a instanciar un objeto de la clase `Tema` que contiene estos datos y lo enviará a la parte privada del servicio mediante `setDataAuth`. Al terminar la operación desplegará una `ventanaVolver` informando al usuario que el tema se ha creado.

#### **B4. Contestar chip**

Para contestar a un chip se seleccionará un tema de la lista que tenemos en la página principal. Este link redirigirá a un formulario similar al de B3. En este formulario al pulsar en `enviar` se llamará a la función `respondeChip` que recoge el texto, el usuario y el `tag` del tema que se recibe por medio de la URL llamando a la función `dameParam` para el parámetro `chipActual`.

Esta función guardará los datos en un objeto de tipo `Chip` y lo enviará a la URL `response`. Al terminar la operación también informará al usuario desplegando una `ventanaVolver` mediante `ventanaVolver`.

### **C1. Seguir usuario**

Para seguir a un usuario se cuenta con dos opciones. Con el botón correspondiente que hay en la lista de usuarios que devuelve la página de búsqueda o con el botón que hay en la lista de seguidores. El funcionamiento de los dos es el mismo, llamar a `seguirUsuario.html` pasándole el id del mismo por el parámetro `id`.

Esta página recogerá el id de usuario mediante `dameParam` y el usuario actual mediante `dameUsr` y llamará a la función `seguir` con estos datos. La función `seguir` recoge esta información y la carga en un objeto del tipo `Follows` para enviarlo a la dirección `followUser` haciendo uso de la función `setDataAuth`. Al terminar, esta función informará al usuario con `ventanaVolver`.

### **C2. No seguir a usuario**

El funcionamiento de este caso es similar al de C1. En la lista de usuarios seguidos se dispone de un botón que llamará a la página `noSeguirUsuario.html` con el parámetro `id` que contendrá el id del usuario en cuestión.

Igual que en el otro caso se recoge el parámetro con `dameParam` y el usuario con `dameUsr` para pasárselos a la función `noSeguir`. Esta instancia un objeto `Follows` con dicha información para enviarlo con `setDataAuth` a la dirección `noFollow`. Finalmente informará al usuario de que ha terminado la operación con `ventanaVolver`.

### **C3. Usuarios seguidos**

En `mainPage` hay un `div` que contiene la lista de usuarios seguidos en la que se imprimen sus nombres junto a un botón para dejar de seguirlos.

Para recuperar esta lista cuando se ha terminado de cargar el documento se hace una llamada a `getSeguidos` pasándole el nombre de usuario obtenido por `dameUsr`. Esta función llamará a `getFolloweds_` concatenando el nombre de usuario y guardando los datos recibidos en `seguidos`.

Otra vez en `mainPage` cuando se lanza `ajaxComplete` se utilizan la variable `seguidos` para llenar la lista.

### **C4. Seguidores**

El funcionamiento de este caso es el mismo que el de C3 solo que en vez de utilizarse la función `getSeguidos` se llamará a `getSeguidores` que utilizará el path `getFollowers` y guardará los datos en `seguidores`.

#### **5.3.8.6 *Diseño de la web***

Se busca una vez mas que el cliente sea lo más parecido a la versión original del programa. Puesto que esta versión se programa en HTML5 se pueden aprovechar los diseños iniciales y los archivos de estilo. En esta versión del cliente se van a aprovechar las capacidades de jQuery además hay que tener en cuenta que no se puede hacer una redirección al recibir los datos de las peticiones al servicio. La página de entrada a la aplicación será `index.html`:

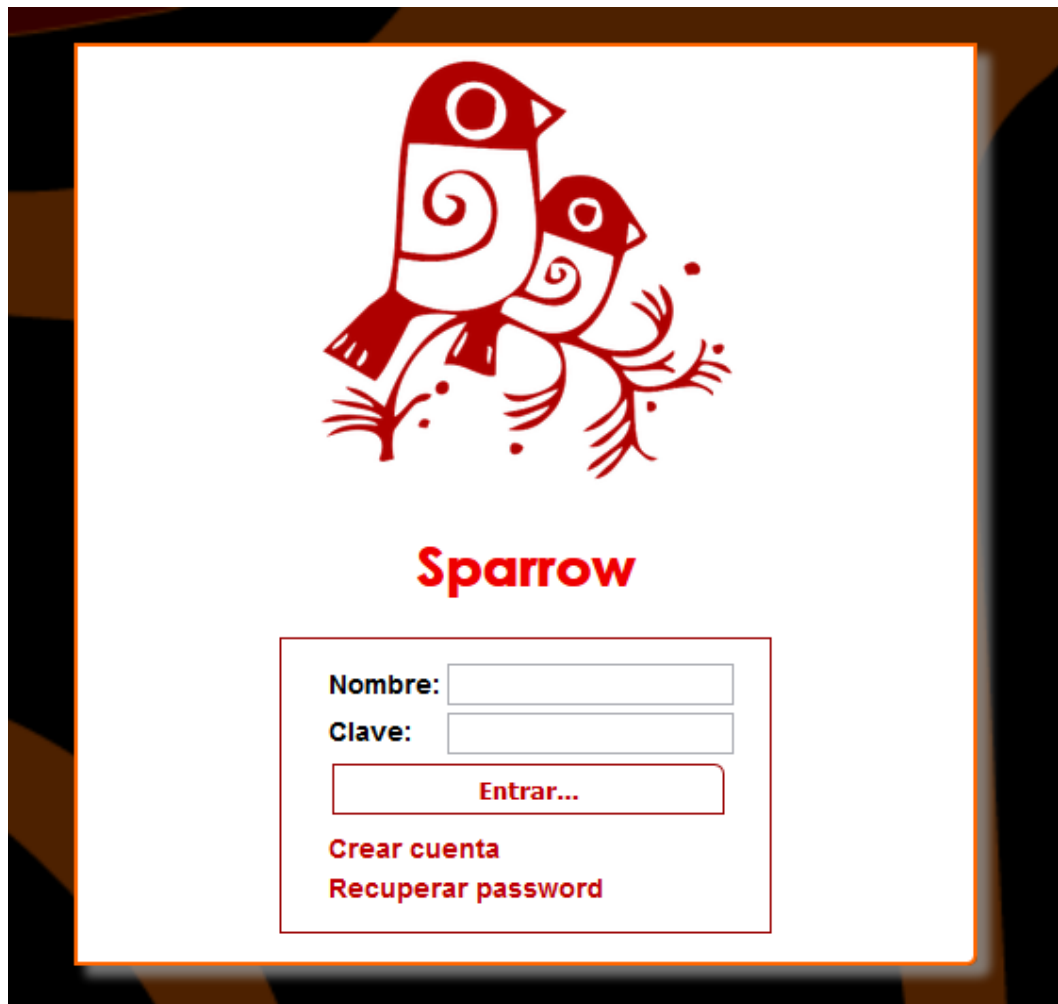


Ilustración 36

En la ilustración 37 se puede ver la página en la que un usuario se puede registrar. En esta se puede apreciar el detalle de la comprobación de claves siendo iguales en este caso.

Ilustración 37 muestra la interfaz de registro de un nuevo usuario. El título principal es 'Registrarse como nuevo usuario' en rojo y subrayado. El formulario de registro está encerrado en un recuadro con una borde roja y contiene los siguientes elementos:

- Etiquetas y campos de entrada:
  - Nombre: Usuario
  - Apellidos: 500
  - Sexo: V (con una flecha hacia abajo)
  - Idioma: Spanish (con una flecha hacia abajo)
  - Email: usuario500@email.com
  - Username: usuario500
  - Password: Representado por siete puntos negros.
  - Repita password: Representado por siete puntos negros.
- Un recuadro verde que aparece debajo del campo de repetición de contraseña con el texto 'Coincide'.
- Un botón rectangular con el texto 'Entrar'.

Ilustración 37



Al enviar los datos la aplicación responde con ventanaVolver. Se puede ver la capa transparente que cubre el formulario y la ventana de respuesta en la siguiente captura:

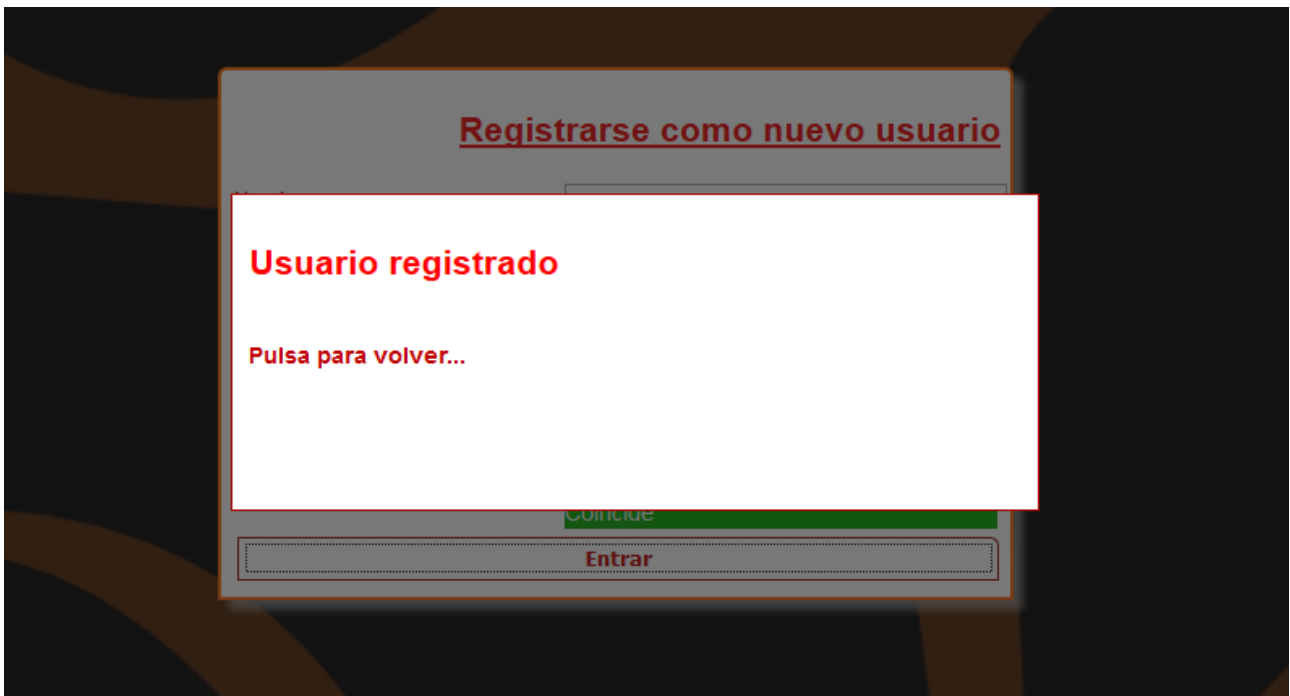


Ilustración 38

Esta es la página resetPass.html en la que se puede recuperar el password.

A screenshot of a web application interface for password reset. The title "Recuperar password" is in red. Below it, there are two input fields: "Usuario:" with the value "usuario01" and "Email:" with the value "email@error.es". Below these fields is a red button labeled "Entrar". The background is dark with some abstract shapes.

Ilustración 39

En la ilustración 40 se puede observar la respuesta del programa en la misma página.

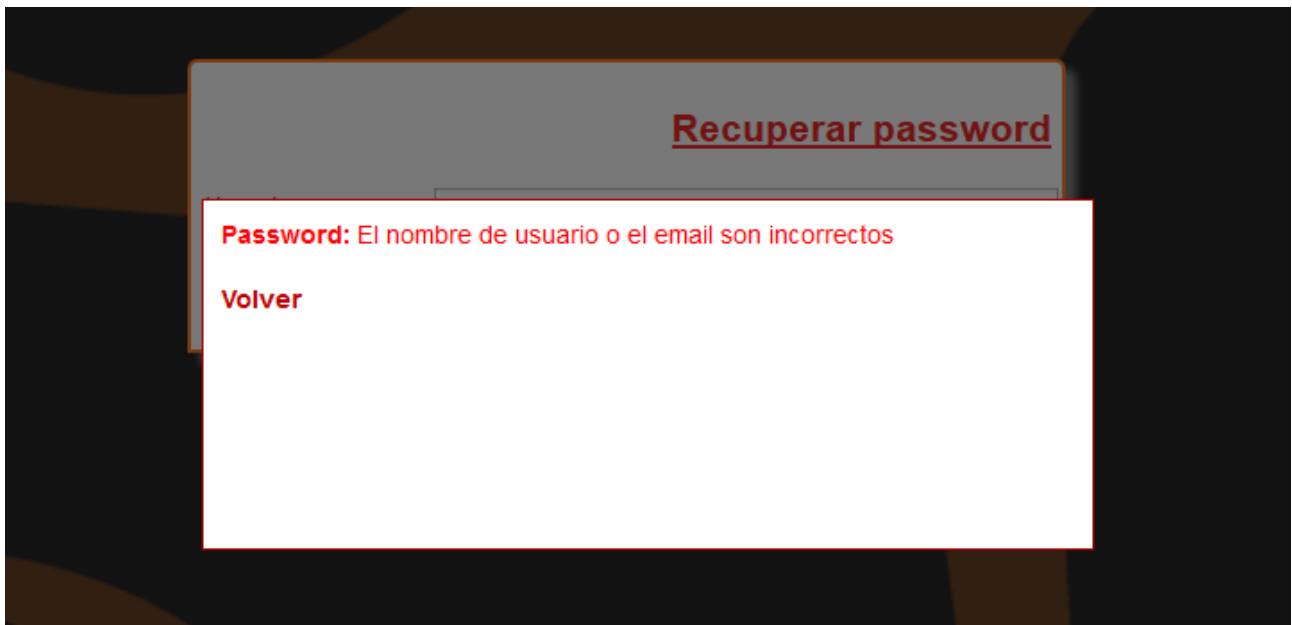


Ilustración 40

Una vez se han introducido los datos de autenticación ya se puede acceder a la página principal. Se puede observar que al compartir los estilos CSS el diseño es el mismo.



Ilustración 41

Esta será la página buscaUsuario.html que se encarga de las búsquedas.

**Busqueda de usuarios**

Apellidos del usuario:

**Volver**

Ilustración 42

Una vez mas la respuesta del cliente se verá en la misma página.

**Usuarios encontrados**

Usuario 02 Prueba [usuario02]	<input type="button" value="+"/>	<input type="button" value="↑"/>
-------------------------------	----------------------------------	----------------------------------

**Volver**

Ilustración 43

A la hora de crear un tema nuevo se llamará a crearTema.html que mostrará el formulario para ello.

**Nuevo Chip**

Tag:

Texto:

**Volver**

Ilustración 44

En esta respuesta se puede apreciar mejor aun la capa transparente desplegada por ventanaVolver.



Ilustración 45

Al pulsar sobre el nombre de un tema se podrán ver los chips a través de verPortag.html.

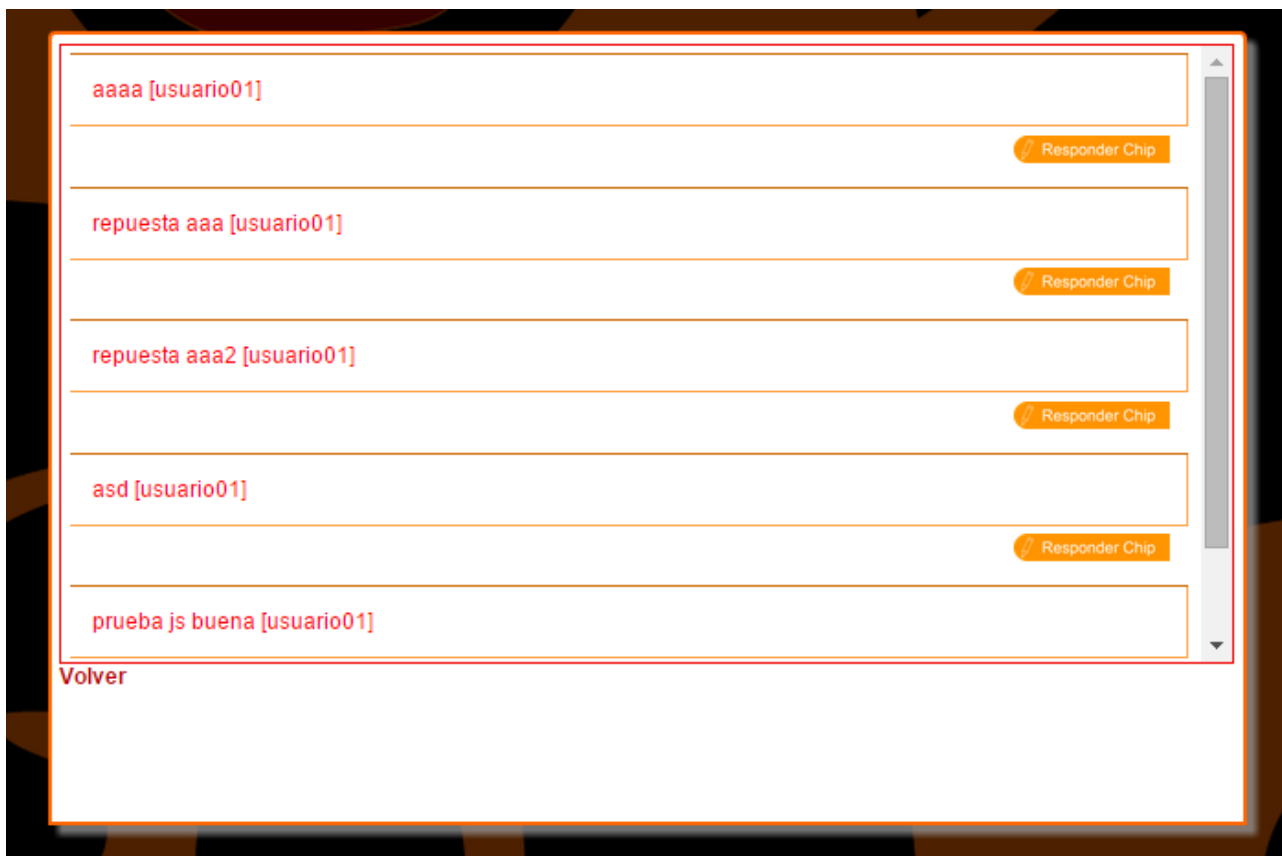


Ilustración 46

Desde esta misma página se pueden enviar respuestas al tema como se puede observar en la ilustración 47.

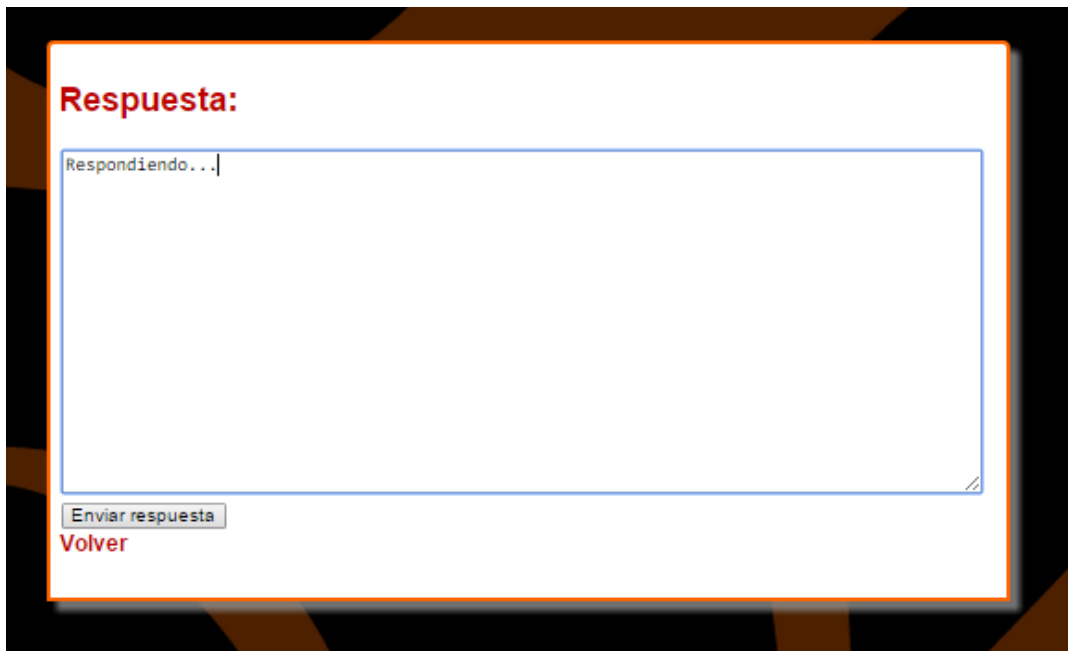


Ilustración 47

Obteniendo una ventana de respuesta.

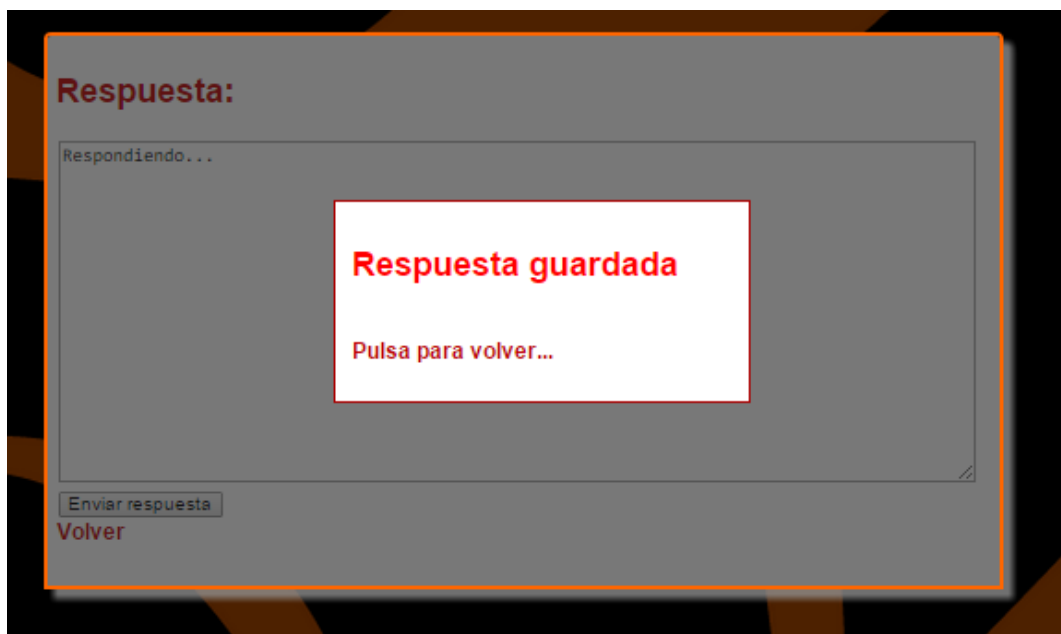


Ilustración 48

El usuario podrá editar la configuración de su cuenta mediante la página `gestionaDatos.html`

**Preferencias**

Nombre de usuario: usuario01

Password: 12monos

Email: kkk@kkk.es

Nombre: Usuario 01js

Apellidos: Pruebajs

Sexo: M ▼

Idioma: German ▼

Actualizar

Volver

Ilustración 49

## 6 PRUEBAS Y RESULTADOS

---

### 6.1 DESCRIPCIÓN DE EXPERIMENTOS

El principal objetivo de este proyecto es la realización de un servicio RESTful a partir de una aplicación JavaEE. Una vez este funcione hay que comprobar su compatibilidad con diferentes clientes y la seguridad del sistema.

Teniendo en cuenta estos dos objetivos la primera prueba a realizar es la comprobación de que el servicio REST funciona. Para ello se accederá a una dirección pública de este para observar los resultados. A continuación se va a repetir la misma prueba pero con una dirección privada para comprobar que el servidor pide los datos de autenticación. Esta prueba además incluirá un acceso a la base de datos para probar que la conexión a esta es correcta y otra para comprobar la respuesta del servicio al enviar datos en JSON. También se hará un análisis de las cabeceras http.

Una vez se ha hecho la comprobación de la parte de servidor se procederá a probar el cliente Java. El primer paso será acceder a las partes públicas del sistema y comprobar su funcionamiento contrastando los datos introducidos con los obtenidos al consultar la base de datos. La mejor opción será crear un usuario nuevo para ver si el sistema está codificando el password correctamente.

El paso final para comprobar este cliente será entrar en la zona privada y analizar las cabeceras http para comprobar que las credenciales se están enviando correctamente. Una vez más habrá que introducir datos mediante alguna sección y contrastarlos con una consulta a la base de datos. En este caso se harán cambios en la configuración de la cuenta, se creará un tema y se responderá un tema ya creado con anterioridad.

En el cliente HTML el primer paso será el mismo que en la versión Java. A continuación se accederá a la zona privada y se analizarán las cabeceras para comprobar los datos de autenticación y las cabeceras CORS. También hay que consultar las cookies para ver si estas se han grabado y su contenido es el esperado. Una vez mas se harán cambios en alguna sección siguiendo los mismos pasos que en el cliente Java.

Para la realización de los experimentos se utilizarán los entornos de programación Eclipse y Netbeans. El navegador Firefox con el plugin Live HTTP Headers para consultar las cabeceras HTTP.

El navegador Google Chrome con el plugin Web Developer para ver las cookies y para hacer consultas mySQL la aplicación phpMyAdmin.

## 6.2 RESULTADOS Y DISCUSIÓN

### 6.2.1 Pruebas del servicio RESTful

#### 6.2.1.1 Acceso a zona pública

Para esta prueba se ha programado el método prueba que está en el path /users/prueba y devuelve un texto plano. El resultado es el siguiente:

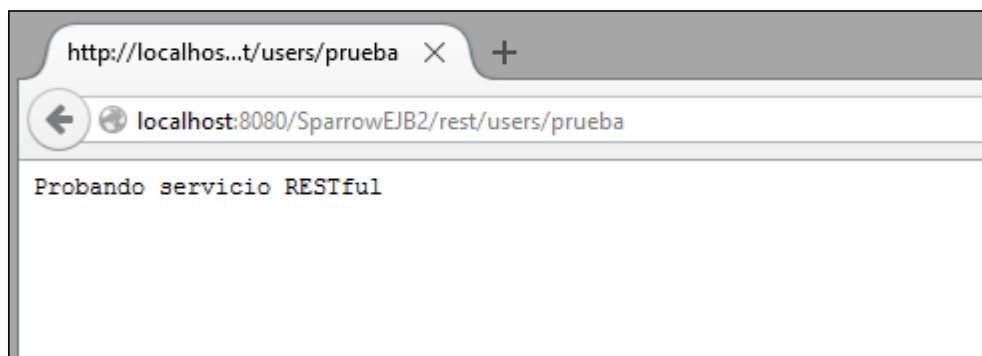


Ilustración 50

El resultado es el esperado. El servicio parte del path /rest, el método está en ServicioUsers que corresponde con la ruta /users y el método está en prueba. Al invocarlo se obtiene el texto esperado.

#### 6.2.1.2 Acceso a zona privada

Se va a acceder a la ruta /users/list que accede a la base de datos y devuelve una lista con todos los usuarios en formato JSON. De esta forma se puede comprobar la seguridad, el acceso a la base de datos y la respuesta en JSON proporcionada por Jersey. Al cargar la ruta el servidor nos responde:

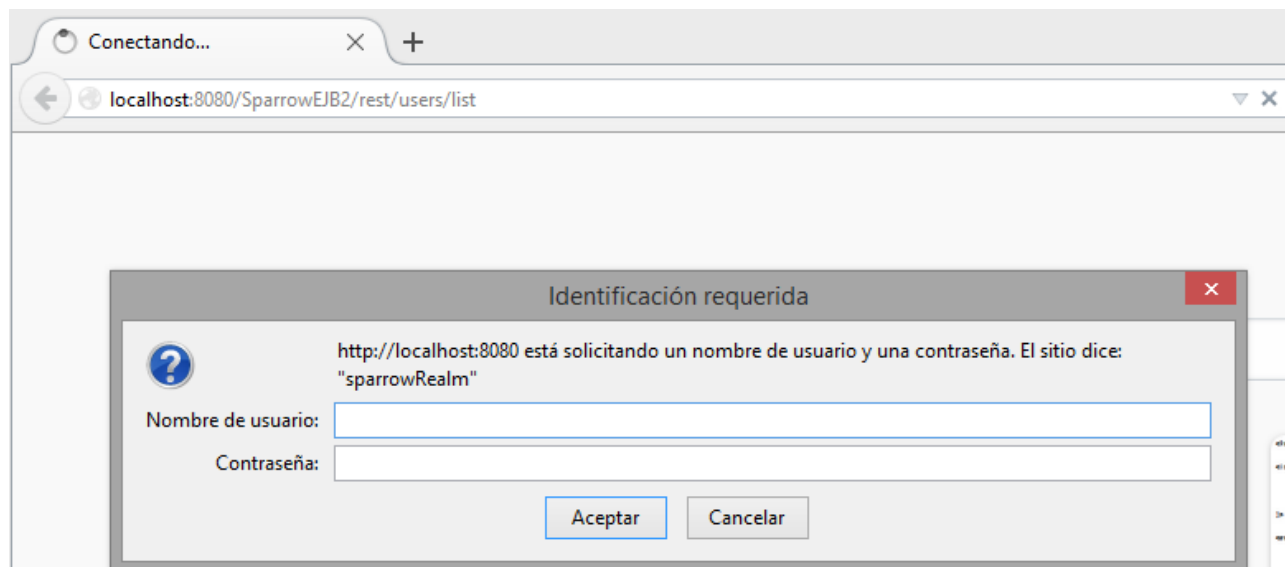


Ilustración 51

Como se esperaba el servidor ha respondido mediante autenticación BASIC por lo que el navegador despliega una ventana para introducir las credenciales.

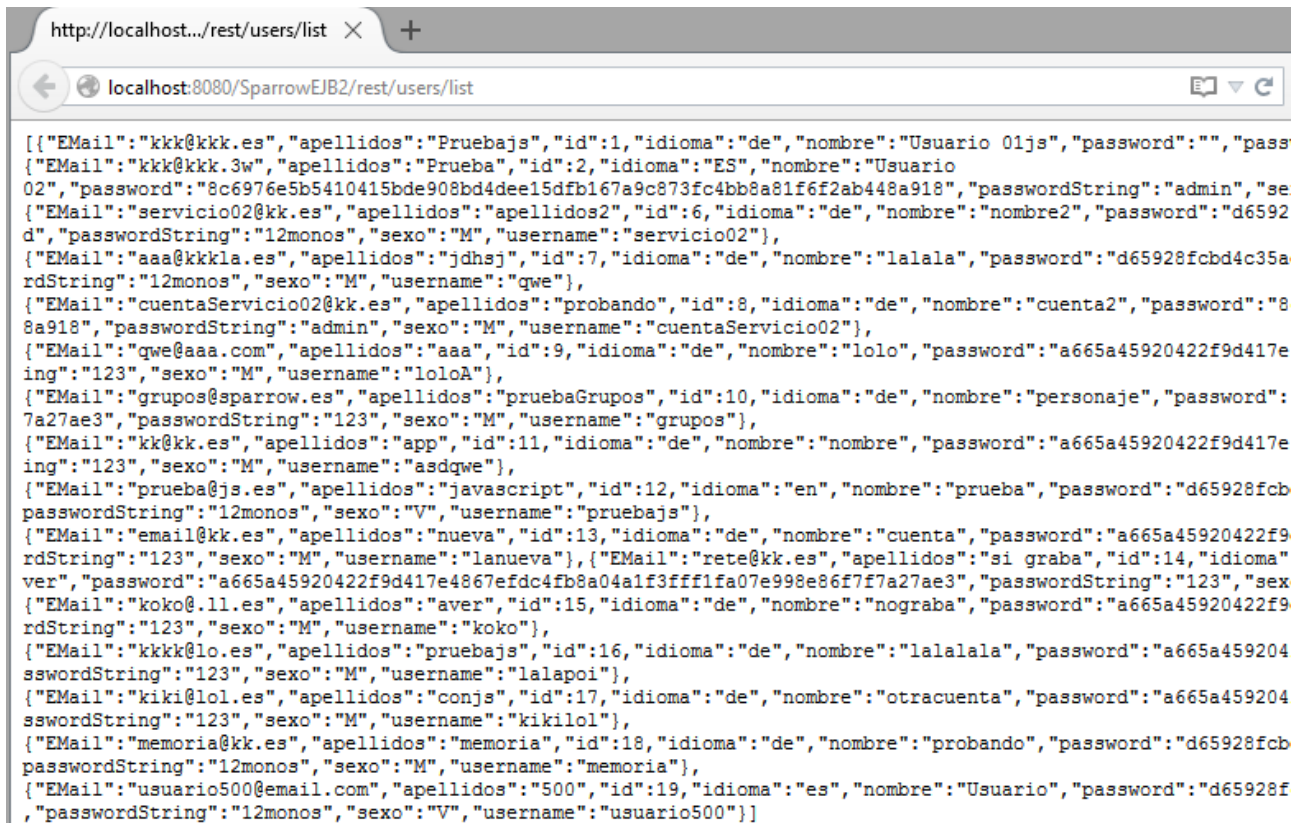


Ilustración 52

Una vez se han introducido las credenciales se muestra en pantalla una lista con los usuarios en formato JSON. A continuación se van a inspeccionar las cabeceras.

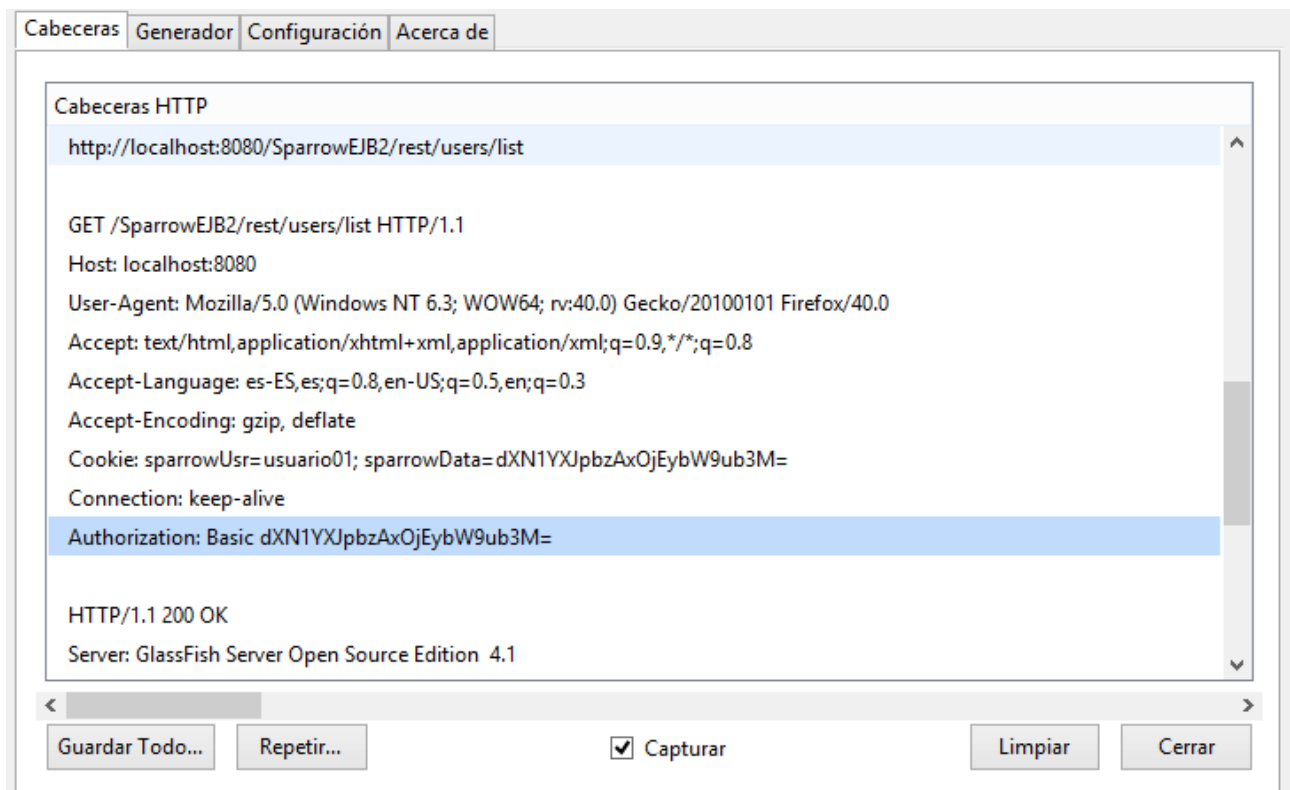


Ilustración 53



En la ilustración 53 se puede ver marcada la cabecera de autorización. Esta será la que el cliente tendrá que enviar cuando quiera acceder a la parte privada.

### 6.2.1.3 Cliente Java

#### 1. Dar de alta a un usuario

En la ilustración 54 hay una captura con los datos introducidos para la prueba.

**Registrarse como nuevo usuario**

Nombre: Experimento

Apellidos: Cliente Java

Sexo: V

Idioma: English

Email: experimento@java.es

Username: eJava

Password: 12monos

Repite password: 12monos

OK

Entrar

Ilustración 54

Al consultar la base de datos el resultado contiene los datos que se han introducido en el formulario.

20	eJava	d65928fcbd4c35addb58c9f2c7ce882341b1e3b57edabf4401...	12monos	experimento@java.es	Experimento	Cliente Java	V	en
----	-------	---	---------	---------------------	-------------	--------------	---	----

La clave codificada en SHA-256 tiene el siguiente valor:

d65928fcbd4c35addb58c9f2c7ce882341b1e3b57edabf4401c79feb635d957d

Mediante una aplicación externa se va a codificar el mismo contenido para comprobar si coincide.

#### Data

12monos

#### SHA-256 hash

d65928fcbd4c35addb58c9f2c7ce882341b1e3b57edabf4401c79feb635d957d

Ilustración 55

En la ilustración 55 podemos comprobar que el password codificado en SHA-256 coincide con el que ha calculado el cliente por lo tanto es correcto.

Cuando se crea un usuario nuevo se le debe asignar un rol. Esto se puede encontrar en la tabla `users_groups` en la que debe salir el nombre de usuario y el rol asignado que será `USERS`.

24	eJava	USERS
----	-------	-------

Se puede comprobar que el usuario ha sido guardado correctamente también en esta tabla.

## 2. Acceso a la zona privada

Para poder analizar las cabeceras se ha utilizado la clase `Headers` que implementa un filtro que imprime en consola todas las cabeceras puesto que es el cliente el que está enviando los datos al servicio estas no se pueden ver en el navegador. El resultado ha sido:

```
Información: =====
Información: RESPONSE
Información: Server
Información: X-Powered-By
Información: =====
Información: EJB REQUEST
Información: accept = application/json
Información: authorization = Basic ZUphdmE6MTJtb25vcw==
Información: user-agent = Jersey/2.10.4 (HttpURLConnection 1.8.0_40)
Información: host = localhost:8080
Información: connection = keep-alive
Información: =====
Información: EJB RESPONSE
Información: Server
Información: X-Powered-By
```

Se puede comprobar que la cabecera de autorización ha sido enviada por lo que en el navegador se obtiene la siguiente respuesta:



Ilustración 56

Por lo que el acceso ha sido correcto y el servicio responde devolviendo los datos de seguidores, seguidos y los temas. Estas son tres llamadas que se hacen a la parte privada del servicio.

### 3. Edición de datos del usuario

Al acceder a las preferencias de la aplicación se podrán editar los datos de usuario.

**Preferencias**

Nombre de usuario: eJava

Password: 12monos

Email: experimento@java.es

Nombre: Experimento

Apellidos: Cliente Java

Sexo: V ▼

Idioma: English ▼

**Actualizar**

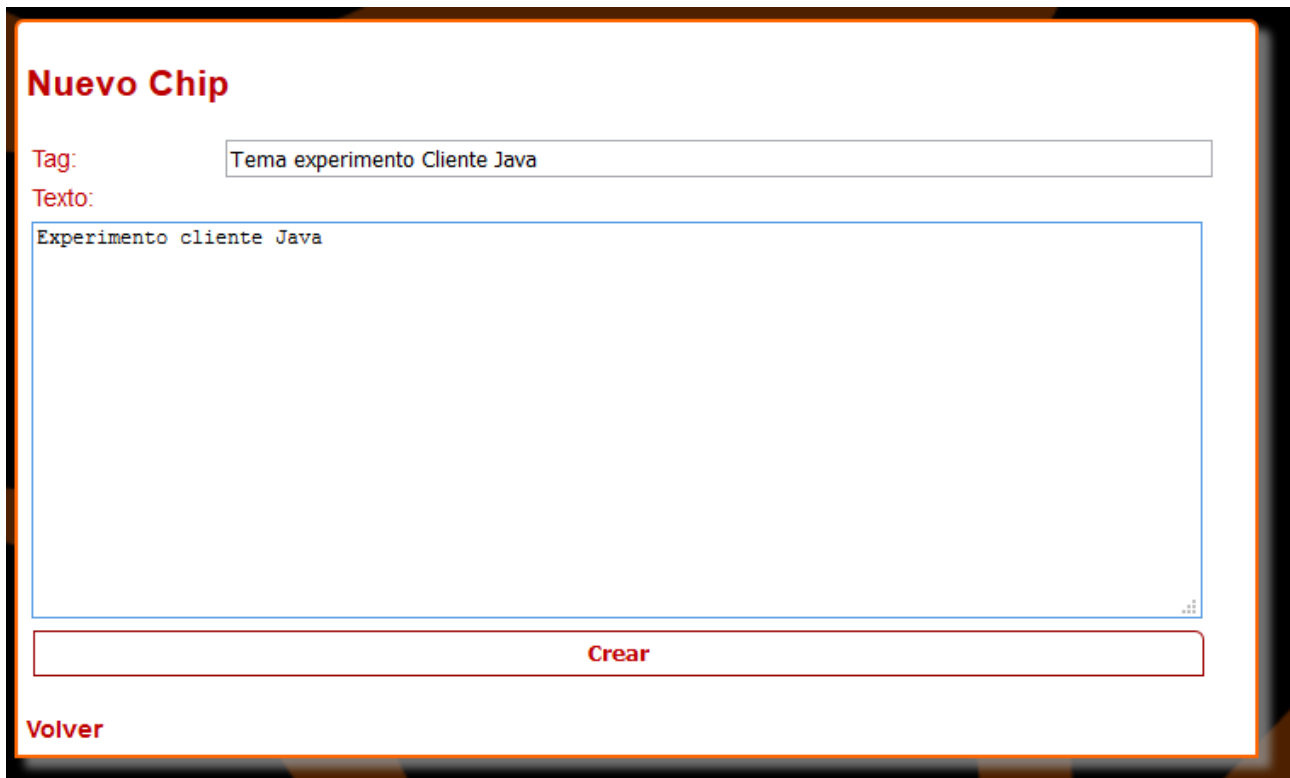
**Volver**

Ilustración 57

En este caso se van a modificar el nombre, el sexo y el idioma cambiándolos a Experimento00, M y Spanish respectivamente. Al consultar la base de datos se comprueba en la ilustración 58 que los datos han cambiado correctamente:

#### 4. Creación de un tema

Se va a proceder a crear un tema nuevo con el siguiente contenido:



**Nuevo Chip**

Tag:

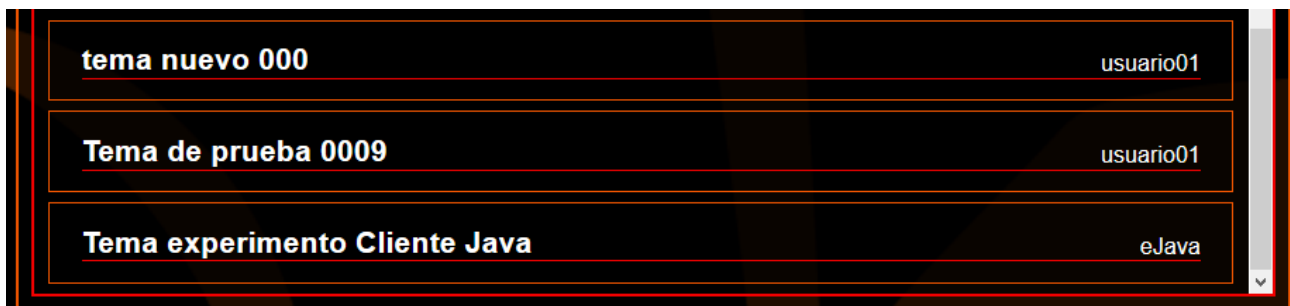
Texto:

[Crear](#)

[Volver](#)

Ilustración 58

La forma más rápida de comprobar que el tema se ha creado es consultando la página principal ya que hay una lista de temas.



<u>tema nuevo 000</u>	<u>usuario01</u>
<u>Tema de prueba 0009</u>	<u>usuario01</u>
<u>Tema experimento Cliente Java</u>	<u>eJava</u>

Ilustración 59

Por lo que el tema debería estar también en la base de datos.

23	Tema experimento Cliente Java	Experimento cliente Java	NULL	20
----	-------------------------------	--------------------------	------	----

Se puede comprobar que los datos son correctos. Y que como el chip es un tema su thread es NULL.

#### 5. Responder tema

Se va a responder al tema aaaa que tiene el id 1 con el siguiente texto:

**Respuesta:**

Experimento cliente Java

**Enviar respuesta**

**Volver**

Ilustración 60

Al volver a la lista de chips de respuesta del tema es visible al final de lista el chip nuevo:

Respondiendo... [usuario01]

Responder Chip

Experimento cliente Java [eJava]

Responder Chip

Ilustración 61

En la base de datos podemos encontrar el nuevo chip con thread 1 creado por el usuario 20, que es el que acabamos de dar de alta en el sistema.

24	aaaa	Experimento cliente Java	1	20
----	------	--------------------------	---	----

Por lo que esta comprobación también ha sido correcta.

#### 6.2.1.4 Cliente HTML

##### 1. Dar de alta a un usuario

Los datos que se van a introducir para esta prueba serán:

## Registrarse como nuevo usuario

Nombre:   
 Apellidos:   
 Sexo:  ▼  
 Idioma:  ▼  
 Email:   
 Username:   
 Password:   
 Repite password:   
Coincide

**Entrar**

*Ilustración 62*

El resultado obtenido al consultar en la base de datos es el correcto. Esta vez no hace falta comprobar el password ya que se ha introducido el mismo que en el usuario anterior y se puede observar que coinciden. Lo cual demuestra también que la función utilidad genKey también funciona.

21	eHTML	d65928fcbd4c35addb58c9f2c7ce882341b1e3b57edabf4401...	12monos	html@experimento.es	Cliente HTML	Experimento	M	de
----	-------	---	---------	---------------------	--------------	-------------	---	----

25 eHTML	USERS
----------	-------

También se comprueba que el usuario tiene asignado el rol que le corresponde.

## 2. Acceso a la zona privada

Como en la prueba anterior se va a utilizar el usuario que se acaba de crear.



Ilustración 63

En la ilustración 63 queda patente que se ha podido recuperar toda la información que necesita acceso a la zona privada sin obtener ningún mensaje de error por parte del servidor.

A continuación se van a analizar las cabeceras en busca de la información CORS.

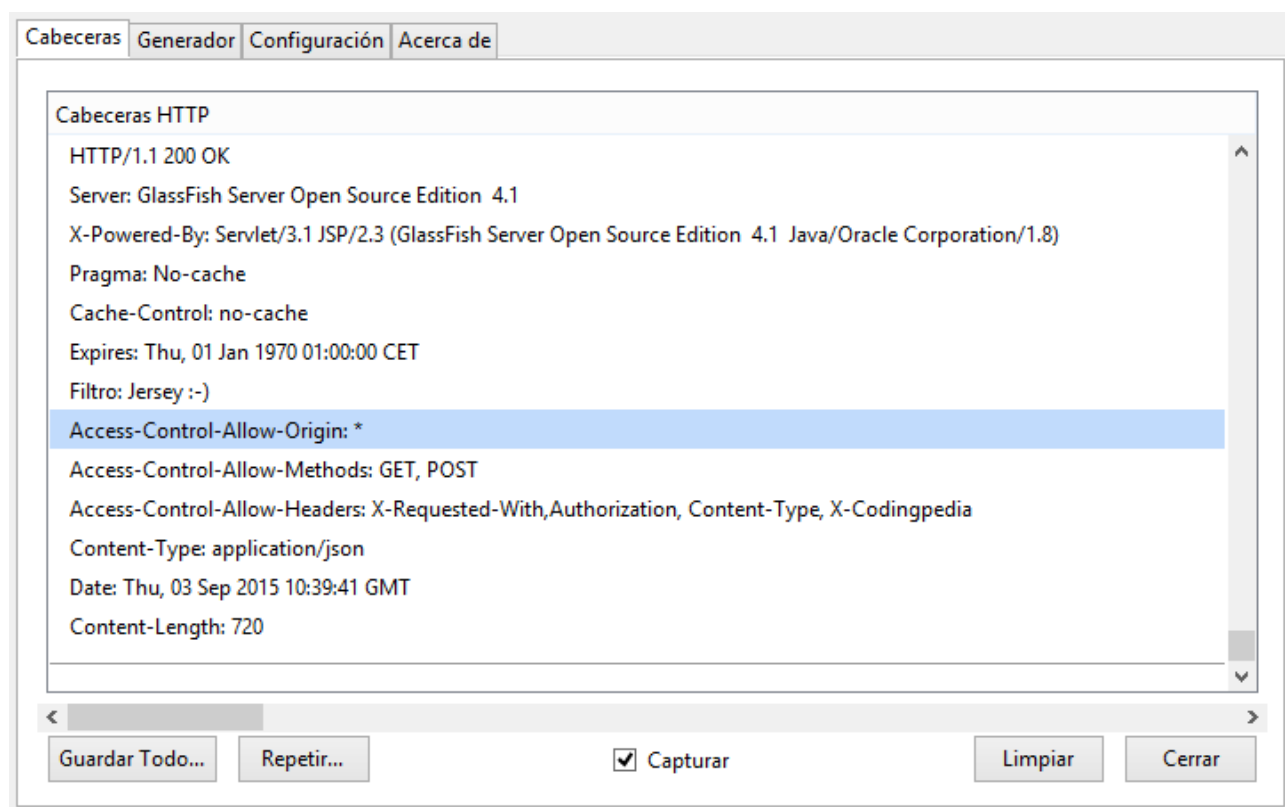


Ilustración 64

El servidor ha respondido con un 200 OK y a partir de la línea azul nos encontramos con las cabeceras devueltas por el servicio para solucionar el problema producido por CORS. En la línea anterior se especifica que estas cabeceras han sido enviadas por el filtro Jersey.

```

=====
EJB REQUEST
host = localhost:8080
user-agent = Mozilla/5.0 (Windows NT 6.3; WOW64; rv:40.0) Gecko/20100101 Firefox/40.0
accept = application/json, text/javascript, */*; q=0.01
accept-language = es-ES,es;q=0.8,en-US;q=0.5,en;q=0.3
accept-encoding = gzip, deflate
content-type = application/json
authorization = Basic ZUhUTUw6MTJtb25vcw==
referer = http://localhost:8383/sparrowClienteHTML/mainPage.html
origin = http://localhost:8383
connection = keep-alive
=====

```

Ilustración 65

En la ilustración 65 se comprueba mediante el filtro que se ha programado que la cabecera de autorización se envía correctamente.

Name	sparrowData	Name	sparrowUsr
Value	ZUhUTUw6MTJtb25vcw==	Value	eHTML
Host	localhost	Host	localhost
Path	/	Path	/
Expires	Fri, 04 Sep 2015 10:46:37 GMT	Expires	Fri, 04 Sep 2015 10:46:37 GMT
Secure	No	Secure	No
HttpOnly	No	HttpOnly	No

Ilustración 66

En la ilustración 66 tenemos las dos cookies que se han escrito para guardar los datos del usuario actual. En sparrowUsr se ha guardado correctamente el nombre del usuario. En sparrowData debería estar el nombre de usuario y la clave separados por dos puntos y codificado en Base64. Vamos a comprobar que la codificación es correcta mediante una aplicación externa.

eHTML: 12monos

> ENCODE <
UTF-8 ▼ (You may also select output charset.)

ZUhUTUw6MTJtb25vcw==

Ilustración 67



El resultado es el esperado. Era evidente porque si los datos no se hubieran codificado bien no se habría podido entrar en la zona privada.

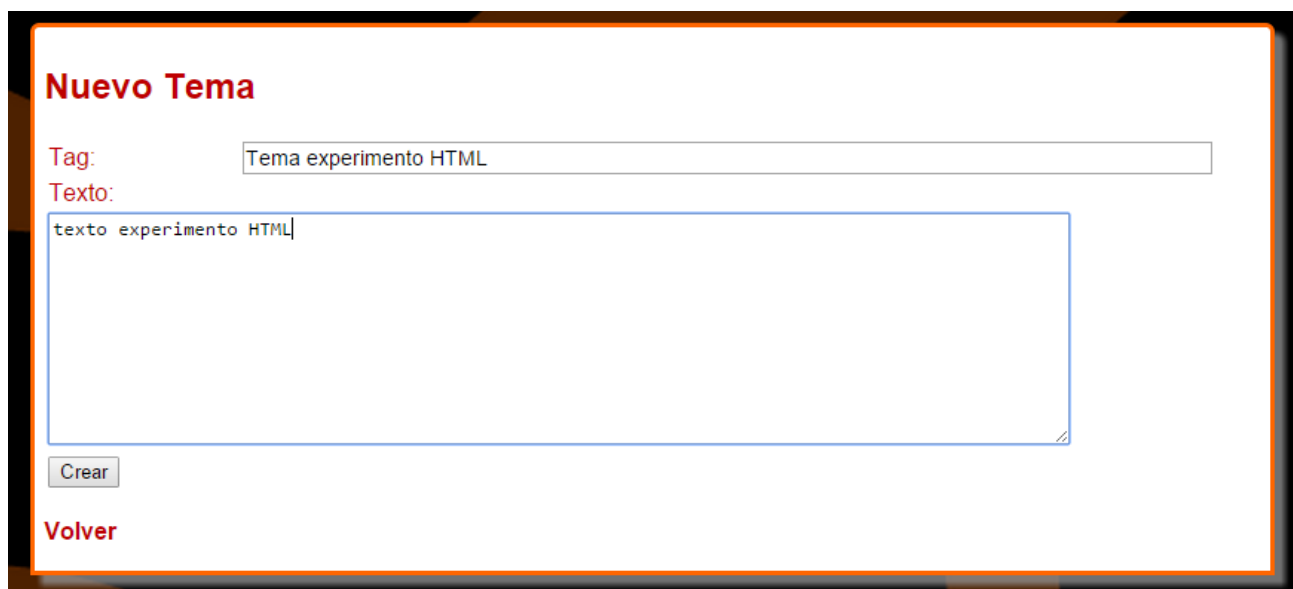
### 3. Edición de datos del usuario

Esta vez se cambiará el apellido por Experimento00, el sexo a V y el lenguaje a Spanish. Veamos si los resultados se actualizan en la base de datos.

//TODO!!! Al editar los datos no se graba la clave sha256. Hay que cambiar generaclave

### 4. Creación de un tema

Se va a crear un nuevo tema con el siguiente texto:



**Nuevo Tema**

Tag:

Texto:

[Volver](#)

Ilustración 68

En la página principal podemos encontrar el nuevo tema añadido a la lista.



Tema de prueba 0009	usuario01
Tema experimento Cliente Java	eJava
Tema experimento HTML	eHTML

Ilustración 69

El cual también se puede encontrar en la base de datos con su thread a NULL.

25	Tema experimento HTML	texto experimento HTML	NULL	21
----	-----------------------	------------------------	------	----

### 5. Responder tema

Para esta prueba se ha escogido el tema tema 2 que tiene un id 2. La respuesta será:

**Respuesta:**

Respuesta experimento HTML

Enviar respuesta
Volver

Ilustración 70

Al consultar las respuestas del tema se puede ver que este se ha añadido:

texto tema2 [usuario01]
 Responder Chip

asdasdas [usuario01]
 Responder Chip

repuesta 2 [usuario01]
 Responder Chip

Respuesta experimento HTML [eHTML]
 Responder Chip

Volver

Ilustración 71

Y en la base de datos encontramos una respuesta un tanto curiosa. El thread del chip debería ser 2 pero ha salido 7. Esto se debe a que hemos pulsado el botón de responder chip del chip respuesta 2 del usuario01. Entonces ¿Por qué aparece en esta lista si el id del thread es diferente? Ello se debe a que los temas no se buscan por id, se buscan por tag y el tag de estos dos chips es tema 2.

26	tema 2	Respuesta experimento HTML	7	21
----	--------	----------------------------	---	----

## 6.3 EVALUACIÓN PRESUPUESTARIA

//TODO

## 7 CONCLUSIONES Y TRABAJO FUTURO

//TODO

## 8 REFERENCIAS

---

- Gulabani, S. (2013). *Developing RESTful Web Services with Jersey 2.0*. Packt Publishing.
- Heffelfinger, D. R. (2014). *Java EE 7 with GlassFish 4 Application Server*. Packt Publishing.
- Hossain, M. (2014). *CORS in Action: Creating and consuming cross-origin APIs*. Manning Publications.
- Jendrock, E., Cervera-Navarro, R., Evans, I., Haase, K., & Markito, W. (2014). *The Java EE 7 Tutorial, Volume 2, Fifth Edition*. Addison-Wesley Professional.
- Jendrock, E., Cervera-Navarro, R., Evans, I., Haase, K., & Markito, W. (2014). *The Java EE 7 Tutorial: Volume 1, Fifth Edition*. Addison-Wesley Professional.
- Kalali, M. (2010). *GlassFish Security*. Packt Publishing.
- Knutson, M. (2012). *Java EE 6 Cookbook for Securing, Tuning, and Extending Enterprise Applications*. Packt Publishing.
- Kou, X. (2009). *GlassFish Administration*. Packt Publishing.
- Libby, A. (2015). *Mastering jQuery*. Packt Publishing.
- Mehta, B. (2014). *RESTful Java Patterns and Best Practices*. Packt Publishing.
- Oracle Corporation. (17 de 08 de 2015). *Jersey RESTful Web Services in Java*. Obtenido de [jersey.java.net](http://jersey.java.net)
- Sandoval, J. (2009). *RESTful Java Web Services*. Packt Publishing.