Servicios RESTFUL en J2EE

# 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, envió de datos en formato XML, JSON… y este nos puede responder también mediante diferentes formatos como texto plano, HTML, XML o JSON por ejemplo.

# Motivación y objetivos

## 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.

## 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.

# Estado del arte

## 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**](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**](http://jsonplaceholder.typicode.com/users/10)

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

{

"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

…

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

## 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.

## 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.

## 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).

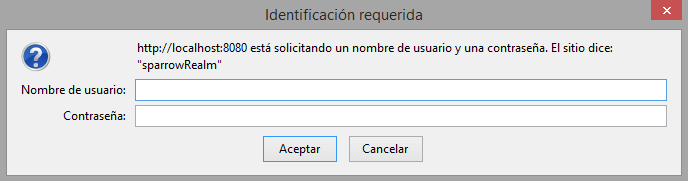


Ilustración 3

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 (HttpUrlConnection 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.

## 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):

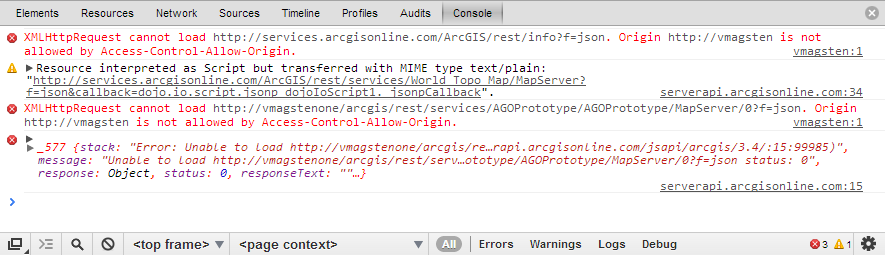


Ilustración 4

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.

## Ajax y jQuery

El envio 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.

## 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).



Ilustración 6

# Especificación

## 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.

## 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.

## Planificación y estimación de costes

//TODO pues todo

# Desarrollo del proyecto

## Análisis

### 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).



Ilustración 7



Ilustración 8

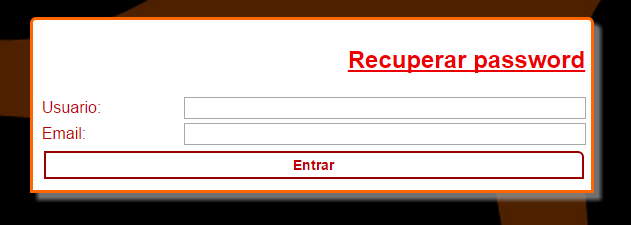


Ilustración 9

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).

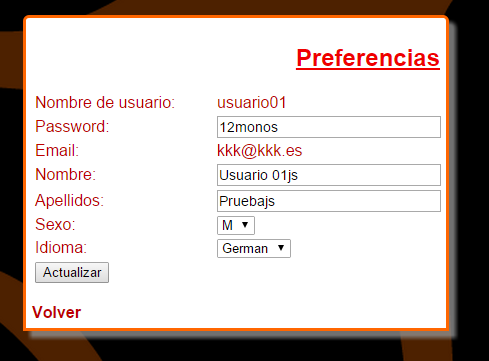


Ilustración 11

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.

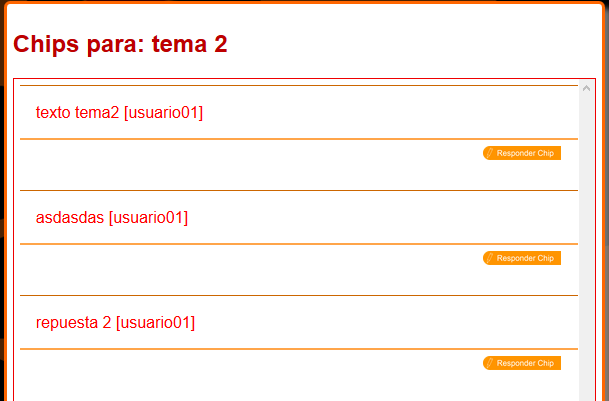


Ilustración 12

### 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 | **Actores:** Usuario invitado |
| **Prerrequisitos:** Ninguno |
| **Descripción:** 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 | **Actores:** Usuario invitado |
| **Prerrequisitos:** Ninguno |
| **Descripción:** 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 | **Actores:** Usuario invitado |
| **Prerrequisitos:** Ninguno |
| **Descripción:** 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 | **Actores:** Usuario registrado |
| **Prerrequisitos:** Tener credenciales de acceso y enviarlas en la cabecera de la petición |
| **Descripción:** 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 | **Actores:** Usuario registrado |
| **Prerrequisitos:** Tener credenciales de acceso y enviarlas en la cabecera de la petición |
| **Descripción:** 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 | **Actores:** Usuario registrado |
| **Prerrequisitos:** Tener credenciales de acceso y enviarlas en la cabecera de la petición |
| **Descripción:** 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 | **Actores:** Usuario registrado |
| **Prerrequisitos:** Tener credenciales de acceso y enviarlas en la cabecera de la petición y el tag del tema |
| **Descripción:** 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 | **Actores:** Usuario registrado |
| **Prerrequisitos:** Tener credenciales de acceso y enviarlas en la cabecera de la petición y el nombre de usuario |
| **Descripción:** 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 | **Actores:** Usuario registrado |
| **Prerrequisitos:** Tener credenciales de acceso y enviarlas en la cabecera de la petición y el thread al que se responde. |
| **Descripción:** El usuario responde a un tema. Los datos se envían en la clase Chips. |

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

|  |  |
| --- | --- |
| Id | Caso de uso – No seguir a usuario |
| C-2 | **Actores:** Usuario registrado |
| **Prerrequisitos:** Tener credenciales de acceso y enviarlas en la cabecera de la petición y el usuario. |
| **Descripción:** 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 | **Actores:** Usuario registrado |
| **Prerrequisitos:** Tener credenciales de acceso y enviarlas en la cabecera de la petición |
| **Descripción:** 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 | **Actores:** Usuario registrado |
| **Prerrequisitos:** Tener credenciales de acceso y enviarlas en la cabecera de la petición |
| **Descripción:** Se recibe una lista de objetos MiniUser desde el servicio que mostrará los seguidores del usuario. Desde aquí es posible seguirles a ellos. |

### Diagramas de clases

**SERVICIO**

**Entidades**











**Servicio**

Estas clases también se utilizarán en los clientes.







