Treball Fi de Carrera

ENGINYERIA TÈCNICA EN

INFORMÀTICA DE SISTEMES

Facultat de Matemàtiques

Universitat de Barcelona

APLICACIÓ WEB PER GESTIONAR CONTINGUTS MULTIMÈDIA MITJANÇANT SPRING I GWT

Anira Escrichs Martínez

Director: Eloi Puertas i Prats

Realitzat a: Departament de Matemàtica

Aplicada i Anàlisi. UB

Barcelona, 15 de gener de 2013

AL MEU PARE

Licencia seleccionada

Reconocimiento-NoComercial-CompartirIgual 3.0 Unported

Índice

1- Introducción....5

1.1-Descripción del proyecto.... 6

2-Análisis 7

2.1-Objetivos del sistema....7

2.2-Visión general de la arquitectura.... 8

2.3-Actores.... 8

2.4-Casos de uso textuales.... 9

2.4.1- Usuario invitado.... 9

2.4.2- Usuario común.... 10

2.4.3- Usuario administrador.... 10

3-Arquitectura.... 12

3.1-Modelo-Vista-Controlador.... 12

3.1.1- Capas....12

3.2-AJAX.... 13

3.3-REST - Representational State Transfer 15

4-Implementación.... 17

4.1-Entorno de desarrollo.... 17

4.2-Vista....21

4.2.1- Framework GWT....21

4.2.2- SmartGWT.... 21

4.2.3- Prototipo Interfaz Usuario 22

4.2.4-Implementación....24

4.3-Controlador o lógica/modelo de negocio....27

4.3.1- Spring MVC....27

4.3.2- Implementación de los casos de uso más importantes.... 34

4.4-Modelo....39

4.4.1- Alfresco.... 39

4.4.2- Web scripts.... 40

4.4.3- Site Alfresco.... 41

4.4.4- Invocación de los casos de uso más importantes.... 43

5-Anexos.......... 47
6-Conclusiones.......... 53
7-Fuentes.......... 54

Índice de ilustraciones

Ilustración 1: Caso de uso - Registro....9
Ilustración 2: Caso de uso - Login.... 10
Ilustración 3: Caso de uso - Download.... 10
Ilustración 4: Caso de uso - Upload....10
Ilustración 5: Caso de uso - Delete Fichero....10
Ilustración 6: Caso de uso - Delete User.... 11
Ilustración 7: Patrón MVC....13
Ilustración 8: AJAX – Motor AJAX....14
Ilustración 9: AJAX – XMLHttpRequest / HttpResponse.... 15
Ilustración 10: Ejemlo de pom.xml.... 18
Ilustración 11: Visualización final de las capas Vista y Controlador 19
Ilustración 12: Registro de un nuevo socio.... 23
Ilustración 13: Introducir credenciales.... 23
Ilustración 14: DataSource de documentos.... 25
Ilustración 15: DataSource de Usuarios.... 26
Ilustración 16: web.xml.... 28
Ilustración 17: applicationContext.xml.... 29
Ilustración 18: Spring @annotations....30
Ilustración 19: Component y derivados.... 31
Ilustración 20: JAX-RS @annotations.... 32
Ilustración 21: Ejemplo Spring y Jersey....33
Ilustración 22: Llamada REST - Registro....34
Ilustración 23: Llamada REST - Login.... 35
Ilustración 24: Invocación web script desde smartGWT - Download....36
Ilustración 25: Llamada REST - Upload....37
Ilustración 26: Llamada REST - Delete User.... 38
Ilustración 27: Web Script MVC Alfresco.... 40
Ilustración 28: Alfresco - Site UB.... 41
Ilustración 29: Alfresco - Contenido carpeta Documentos.... 42
Ilustración 30: Cliente - Contenido carpeta Documentos desde el GRID de smartGWT....42
Ilustración 31: Implementación Web Script Alfresco - Registro....43

Ilustración 32: Implementación Web Script Alfresco - Login.. 44
Ilustración 33: Implementación Web Script Alfresco - Upload..45
Ilustración 34: Implementación Web Script Alfresco - Delete User.. 46
Ilustración 35: Pantalla Inicial.. 47
Ilustración 36: Formulario de Registro y validaciones..48
Ilustración 37: Entrada usuario 48
Ilustración 38: Usuario descarga contenido 49
Ilustración 39: Usuario sube contenido..49
Ilustración 40: Login Admin 50
Ilustración 41: Admin Eliminar Usuarios... 50
Ilustración 42: Admin Menú Eliminar Usuarios.. 51
Ilustración 43: Admin Confirma Elimina Usuario..51
Ilustración 44: Admin Grid Usuario Eliminado..52
Ilustración 45: Admin Grid Contenido Eliminar.. 52

1- Introducción

Como Trabajo de Fin de Carrera (TFC) he realizado el análisis, diseño e implementación de una aplicación web utilizando varias tecnologías y una arquitectura separada por capas.

El proyecto consiste en una aplicación accesible a través de cualquier red, en la que los usuarios pueden compartir todo tipo de archivos multimedia en una red de computadoras, ya sea en Internet o como aplicación en una red privada.

El TFC tiene como objetivo realizar un trabajo de síntesis en el que se deben aportar los conocimientos adquiridos durante la carrera y que requiere ponerlos en práctica conjuntamente en un proyecto concreto. Es un desarrollo práctico y bien documentado, vinculado al ejercicio profesional de la informática.

La elección de esta área de trabajo ha estado motivada por varios factores. Por un lado, me ha permitido experimentar el desarrollo de un proyecto desde cero utilizando una metodología rigurosa y académica. Por otra parte, me ha permitido poner mis conocimientos en práctica durante mi experiencia profesional como desarrolladora en el mundo de la consultoría, gestionando el tiempo de mi propio proyecto.

Uno de los inconvenientes que encontré al inicio de mi experiencia laboral, era el de implementar la interfaz gráfica de una aplicación, ya que el aprendizaje académico, pasó por alto esta parte tan importante en el desarrollo de software.

Más adelante explicaré qué framework es aconsejable para programadores Java sin conocimientos de HTML y CSS.

Una vez elegida el área, he buscado un proyecto que pudiese ser desarrollado completamente en los tiempos previstos para la realización del TFC, con una estructura similar a los que nos encontramos en una empresa y que permitiese investigar y ampliar diferentes opciones de implementación dentro de la misma arquitectura. Es en este contexto donde surge la idea de

realizar una aplicación que implemente la gestión de contenidos multimedia por varios usuarios accediendo a la vez desde cualquier red de ordenadores.

La principal aportación del proyecto es brindar una aplicación que permita compartir archivos en cualquier red.

1.1- Descripción del proyecto

Esta aplicación tiene por objetivo compartir, entre la red, cualquier archivo multimedia. Para ello se ha creado una interfaz de usuario sencilla y amigable, un repositorio para el almacenamiento de archivos y una capa con la lógica-de-negocio.

Al acceder a la aplicación, los usuarios (sin importar si se han identificado) podrán consultar los archivos agregados. También podrán realizar búsquedas y ver el detalle del archivo.

Los usuarios podrán darse de alta en la aplicación completando un formulario de registro disponible en la web. Una vez realizado este paso podrán, previa identificación, descargar y subir archivos al repositorio.

La aplicación tendrá al menos un administrador que tendrá los permisos para borrar archivos o usuarios de la aplicación.

2- Análisis

En este documento se elaboran las especificaciones para la construcción de los diferentes elementos software que constituyen el Sistema Final.

A partir de la elaboración de estas especificaciones, se aborda la codificación de los componentes.

El propósito del sistema es crear un sistema de gestión archivos para poder compartirlos en la red y a la vez una distinguida Arquitectura separada por capas para desarrolladores.

2.1- Objetivos del sistema

Los objetivos que se persiguen tanto a nivel de proyecto como a nivel de aplicación son:

a nivel de proyecto:

- Distintas capas de desarrollo para programadores.
- Utilizar software libre
- Frameworks

a nivel de aplicación:

- Facilidad y rapidez de acceso a la documentación generada desde cualquier punto geográfico.
- Diferentes roles de usuario
- Interfaz gráfica sencilla y amigable

2.2- Visión general de la arquitectura

Para ello, el sistema estará basado en 3 componentes:

1. Repositorio de documentación:

   Donde se almacenará toda la información generada. Para el proyecto el repositorio es Alfresco Community Edition v.4.0.

2. Capa REST:

   Aplicación que contendrá toda la lógica de negocio de la aplicación. Será de carácter Web Services con tecnología REST al estilo de una capa de lógica de una aplicación distribuida, tiene por objetivo encapsular las funcionalidades de acceso a Alfresco en forma de servicios web. Gracias a esta capa se simplificará el acceso al repositorio Alfresco, pudiendo otras aplicaciones utilizarla. Esta capa será desarrollada bajo el Framework Spring.

3. Interfaz gráfica:

   Aplicación resultante desde donde se podrán gestionar los diferentes archivos. Esta aplicación se desarrollará con tecnología GWT

En el siguiente apartado avanzaremos en el análisis del problema propuesto, describiendo los actores y los casos de uso más importantes. También se presentará un prototipo del sitio web a desarrollar.

2.3- Actores

Se citan los actores que interactúan con la aplicación:

- Invitado:

  Son aquellos usuarios que consultan el catálogo sin identificarse previamente. Pueden consultar el material disponible pero no pueden descargar o subir archivos al repositorio.

- Usuario Común:

  Este tipo de usuarios, al igual que los invitados ,pueden consultar el repositorio de archivos. Pero a diferencia de estos, están habilitados, previa identificación, para descargar o subir archivos al repositorio.

- Administrador:

  Son usuarios con permisos especiales que permiten gestionar el repositorio con privilegios mayores. Permite eliminar archivos y usuarios.

2.4- Casos de uso textuales

De la descripción del proyecto podemos identificar los siguientes casos de uso y sus respectivos usuarios.

2.4.1- Usuario invitado

Nombre	Registro
Descripción	Tras rellenar un formulario, un usuario se da de alta en la aplicación.

Ilustración 1: Caso de uso - Registro

2.4.2- Usuario común

Nombre	login
Descripción	Al introducir nombre de usuario y password, el usuario se loga en la aplicación.

Ilustración 2: Caso de uso - Login

Nombre	download
Descripción	Si el usuario ya está logado en la aplicación puede descargar archivos multimedia del repositorio

Ilustración 3: Caso de uso - Download

Nombre	upload
Descripción	Si el usuario ya está logado en la aplicación puede subir archivos multimedia al repositorio

Ilustración 4: Caso de uso - Upload

2.4.3- Usuario administrador

Nombre	deleteFichero
Descripción	Elimina un fichero de Alfresco. Comprueba previamente, que el usuario sea el Administrador.

Ilustración 5: Caso de uso - Delete Fichero

Nombre	deleteUser
Descripción	Elimina un usuario del respositorio. Comprueba previamente, que el usuario sea el Administrador.

Ilustración 6: Caso de uso - Delete User

3- Arquitectura

Después de un cuidadoso análisis de los objetivos del proyecto, se determinó que la mejor manera de estructurar el sistema era haciendo uso del muy famoso "patrón de diseño": *Model-View-Controller* que se representa como una arquitectura de 3 niveles.

En este punto se va a detallar aquella parte de la Arquitectura del Sistema relativa a los componentes software desarrollados con objeto de permitir el acceso al sistema y la satisfacción de todos los requisitos.

3.1- Modelo-Vista-Controlador

MVC (por sus siglas en inglés) es un patrón de diseño de arquitectura de software usado principalmente en aplicaciones que manejan gran cantidad de datos y transacciones complejas donde se requiere una mejor separación de conceptos para que el desarrollo esté estructurado de una mejor manera, facilitando la programación en diferentes capas de manera paralela e independiente. *MVC* sugiere la separación del software en 3 capas: Modelo, Vista y Controlador.

Si una misma aplicación debe ejecutarse tanto en un navegador estándar como un navegador de un dispositivo móvil, solamente es necesario crear una vista nueva para cada dispositivo; manteniendo el controlador y el modelo original de esta manera se consigue un mantenimiento más sencillo de las aplicaciones.

3.1.1- Capas

Modelo: Es la representación de la información que maneja la aplicación. El modelo en sí son los datos puros que puestos en contexto del sistema proveen de información al usuario o a la aplicación misma.

Vista: Es la representación del modelo en forma gráfica disponible para la interacción con el usuario. En el caso de una aplicación Web, la "Vista" es una página HTML con contenido dinámico sobre el cuál el usuario puede realizar operaciones.

Controlador: Es la capa encargada de manejar y responder las solicitudes del usuario,

procesando la información necesaria y modificando el Modelo en caso de ser necesario.

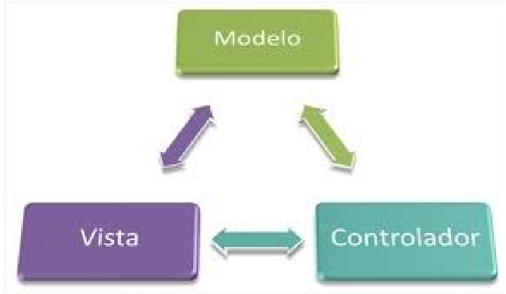

Ilustración 7: Patrón MVC

La capa **Vista** será desarrollada con el framework **GWT** que hace uso de la técnica **AJAX** (se explica en el siguiente punto) para la creación de aplicaciones web asíncronas. Esta capa, envía peticiones HTTP al **Controlador,** el cual captura la petición a través de una llamada **REST** (también será mencionada en este capítulo). El **Modelo** será pues, la capa encargada de la persistencia.

3.2- AJAX Asynchronous JavaScript y XML

AJAX Asynchronous JavaScript y XML. No es un nuevo lenguaje de programación, sino una nueva forma de utilizar las normas existentes. Consiste en una técnica para la creación de páginas web de forma rápida y dinámica. Permite que las páginas web se actualicen de forma asincrónica mediante el intercambio de pequeñas cantidades de datos con el servidor en segundo plano. Esto significa que es posible actualizar partes de una página web, sin volver a cargar la página entera. Las páginas web que no utilizan AJAX deben volver a cargar toda la página si el contenido cambiar.

Aplicaciones como Google Maps, Gmail, Youtube, Facebook, son un ejemplo de tecnología AJAX.

Éstas se ejecutan en el cliente, es decir, en el navegador y mantiene comunicación asíncrona con el servidor en segundo plano. De esta forma es posible realizar cambios sobre la misma página sin necesidad de recargarla. Esto significa aumentar la interactividad, velocidad y usabilidad en la misma.

AJAX es la combinación de:

- Hojas de estilos en cascada (CSS) para el diseño que acompaña a la información.
- Document Object Model (DOM) accedido generalmente mediante Javascript, para mostrar e interactuar dinámicamente con la información presentada.
- El objeto XMLHttpRequest para intercambiar datos asíncronamente con el servidor web.
- XML es el formato usado comúnmente para la transferencia entre el servidor y el cliente, sin embargo, puede usarse cualquier formato, tales como: texto plano, HTML y JSON.

En la siguiente imagen se muestra como trabaja todo en conjunto:

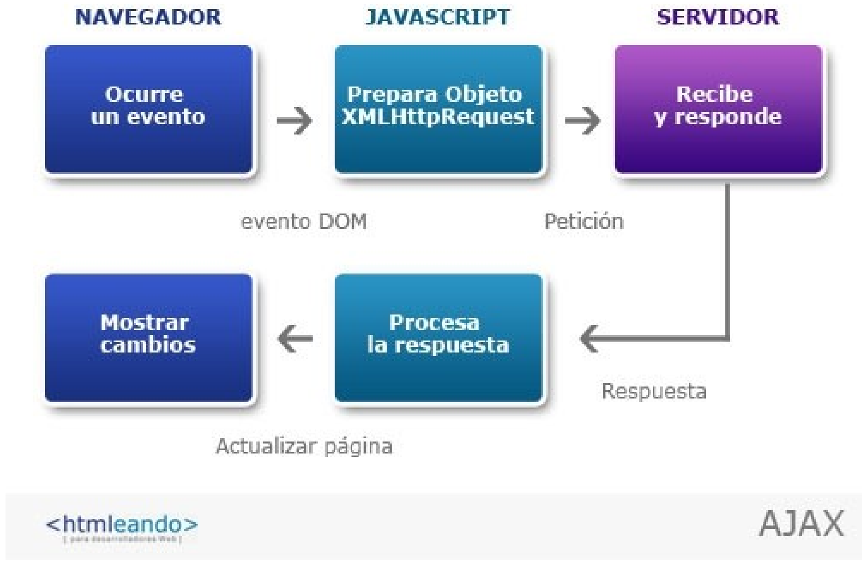

Ilustración 8: AJAX – Motor AJAX

Creando una petición al servidor con XMLHttpRequest

En el gráfico podemos apreciar como el cliente envía peticiones XMLHttpRequest y el flujo de datos.

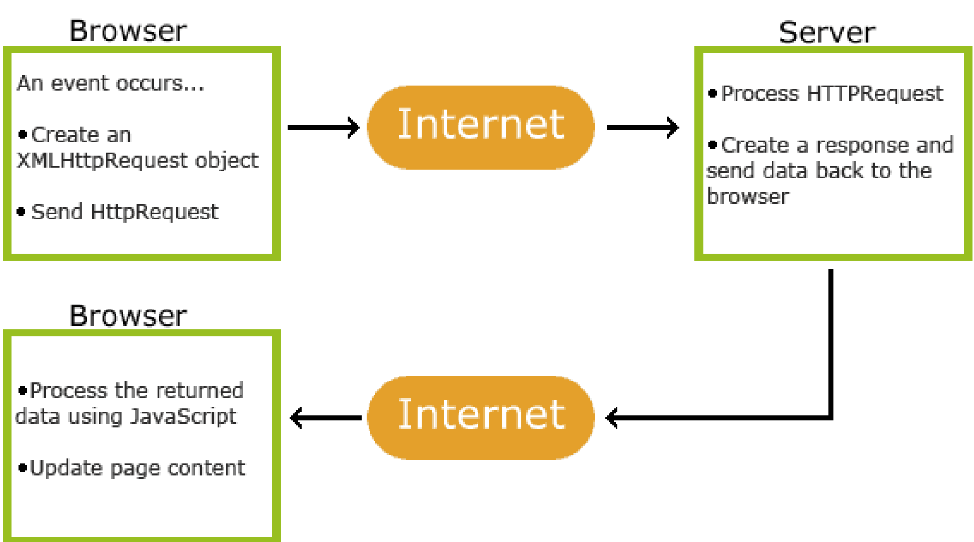

Ilustración 9: AJAX – XMLHttpRequest / HttpResponse

XMLHttpRequest es una interfaz empleada para realizar peticiones HTTP y HTTPS a servidores Web. Para los datos transferidos se usa cualquier codificación basada en texto, incluyendo: texto plano, XML, JSON, HTML y codificaciones particulares específicas. La interfaz se implementa como una clase de la que una aplicación cliente puede generar tantas instancias como necesite para manejar el diálogo con el servidor, éste procesa la petición a través de una llamada REST creando una respuesta **HttpResponse** y retornándola al cliente que refrescará la página usando Javascript. A continuación se menciona la arquitectura *REST* que ha sido utilizada en el servidor de aplicaciones.

3.3- REST - Representational State Transfer

La **Transferencia de Estado Representacional** (Representational State Transfer) o **REST** es un estilo de arquitectura de software para sistemas distribuidos como la World Wide Web. REST se ha convertido en un modelo de diseño Web predominante. Fue introducido y definido en 2000 por Roy Fielding en su tesis doctoral. Fielding es uno de los autores principales del Protocolo de transferencia de hipertexto (HTTP). Los sistemas que siguen los principios REST se llaman con frecuencia RESTful.

Principios de arquitectura y diseños fundamentales clave:

– Un **protocolo cliente/servidor sin estado**: cada mensaje HTTP contiene toda la información necesaria para comprender la petición. Como resultado, ni el cliente ni el servidor necesitan recordar ningún estado de las comunicaciones entre mensajes. Sin embargo, en la práctica, muchas aplicaciones basadas en HTTP utilizan cookies y otros mecanismos para mantener el estado de la sesión.
– Un conjunto de **operaciones bien definidas** que se aplican a todos los recursos de información: HTTP en sí define un conjunto pequeño de operaciones, las más importantes son POST, GET, PUT y DELETE. Con frecuencia estas operaciones se equiparan a las operaciones CRUD que se requieren para la persistencia de datos.
– Una **sintaxis universal** para identificar los recursos. En un sistema REST, cada recurso es direccionable únicamente a través de su URI. El uso de hipermedios, tanto para la información de la aplicación como para las transiciones de estado de la aplicación: la representación de este estado en un sistema REST son típicamente HTML, XML o JSON. Como resultado de esto, es posible navegar de un recurso REST a muchos otros, simplemente siguiendo enlaces sin requerir el uso de registros u otra infraestructura adicional.

Una vez definida la arquitectura del sistema MVC, usando la técnica AJAX para la parte cliente , comunicándose con el servidor a través de llamadas REST; se explica en el siguiente capítulo qué herramientas son necesarias para comenzar a construir el sistema.

4- Implementación

En este capítulo se explica más detalladamente qué software se ha utilizado en esta arquitectura y cómo empezar a construir un sistema fácil de mantener.

El proyecto se ha realizado única y exclusivamente con software libre.

Se han usado distintas tecnologías para cada una de las capas de la aplicación. Este capítulo está compuesto de 4 partes. Se describe cómo empezar a preparar el entorno de desarrollo y se hace una explicación más exhaustiva del software usado en cada capa para la arquitectura MVC en el desarrollo del proyecto describiendo las herramientas o frameworks usados en cada una de ellas.

Se muestra la implementación de los casos de uso más importantes en cada capa, de modo que la aplicación desarrollada es completamente funcional y cumple con todos los requisitos especificados al inicio del proyecto.

Se da por hecho que la instalación es correcta de inicio con el software necesario para construir el sistema final.

4.1- Entorno de desarrollo

En primer lugar debemos escoger un entorno para desarrollar nuestro código. Eclipse es un entorno de desarrollo integrado de código abierto multiplataforma. Esta plataforma ha sido usada para desarrollar entornos de desarrollo integrados (del inglés IDE).

En el proyecto se ha usado Eclipse Indigo con los siguentes plugins:

– **The Google Plugin for Eclipse.**

  Este puglin se explicará como parte de la capa Vista.

– **Web Tools Platform (WTP) Project - Eclipse**

  Instalando este plugin obtenemos el servidor web Tomcat en el que tendremos alojado nuestro proyecto web.

– **M2E - Maven Integration for Eclipse**

Maven será usado como herramienta para construir el proyecto Java. Utiliza un (POM) Project Object Model para configurar el proyecto y gestionar dependencias de otros módulos; esto es, si es necesaria cualquier librería, se debe escribir en el POM y Maven, automáticamente descargará las dependencias en un directorio de nuestro proyecto.

Maven nos ayuda a compilar las clases .java, permite ejecutar test automáticos de Junit, genera ficheros .jar o .war con un simple *Maven Install, etc.*

projectUB/pom.xml

```
1  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2      xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
3      <modelVersion>4.0.0</modelVersion>
4      <groupId>com.ub</groupId>
5      <artifactId>projectUB</artifactId>
6      <packaging>war</packaging>
7      <version>0.1</version>
8      <name>projectUB Maven Webapp</name>
9      <properties>
10         <!-- <env>dev</env> Entorn por defecte --> <!-- packaging per defecte -->
11         <encoding>ISO-8859-1</encoding>
12         <java.version>1.6</java.version>
13         <org.springframework.version>3.0.5.RELEASE</org.springframework.version>
14         <jersey.version>1.7</jersey.version>
15
16         <gwt.version>2.4.0</gwt.version>
17         <smartgwt.version>3.0</smartgwt.version>
18         <alfresco.username>admin</alfresco.username>
19         <alfresco.password>admin</alfresco.password>
20         <alfresco.endpoint>http://127.0.0.1:8080/alfresco/service</alfresco.endpoint>
21         <alfresco.sitios>Sitios</alfresco.sitios>
22
23     </properties>
24     <url>http://maven.apache.org</url>
25     <build>
26         <finalName>projectUB</finalName>
27
28         <plugins>
29             <plugin>
30                 <groupId>org.apache.maven.plugins</groupId>
31                 <artifactId>maven-eclipse-plugin</artifactId>
32                 <configuration>
33                     <wtpapplicationxml>true</wtpapplicationxml>
34                     <wtpversion>2.0</wtpversion>
35                 </configuration>
36             </plugin>
37
38             <plugin>
39                 <groupId>org.apache.maven.plugins</groupId>
40                 <artifactId>maven-compiler-plugin</artifactId>
41                 <version>2.1</version>
42                 <configuration>
```

Overview | Dependencies | Dependency Hierarchy | Effective POM | pom.xml

Ilustración 10: Ejemlo de pom.xml

Desde eclipse debemos construir con el plugin de Maven un New Maven Project, existen varios

foros para la creación y configuración del proyecto.

Los siguientes frameworks y tecnologías se explican a continuación y en que capa del patrón MVC se ha usado cada uno de ellos para construir el sistema final.

- **Modelo**: Alfresco community 4.0
- **Vista**: GWT, smartGWT.
- **Controlador**: Spring

La siguiente imagen muestra cómo han sido separadas la Vista y el Controlador en el proyecto:

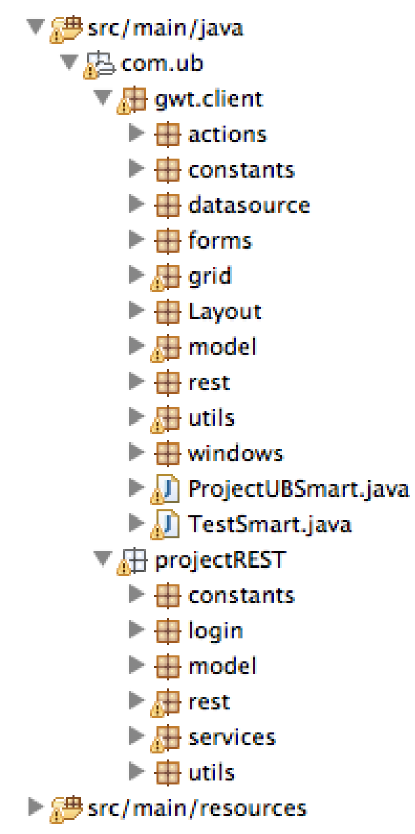

Ilustración 11: Visualización final de las capas Vista y Controlador

El la ilustración 10 vemos el *package* gwt.client el cual contiene otros package con las clases Java que implementan el código usando AJAX (apartado 4.2- Vista).

En el *package* projectREST encontramos las clases Java que construyen el servidor mediante el framework Spring (apartado 4.3- Controlador o lógica/modelo de negocio).

4.2- Vista

La capa del cliente es donde se consumen y presentan los modelos de datos. Para una aplicación Web, la capa cliente normalmente es un navegador web.

El diseño del lado del cliente utiliza una arquitectura Modelo-Vista-Presentador (MVP). La principal ventaja de este patrón para el proyecto es la capacidad de facilitar y minimizar el acomplamiento de componentes UI que son desarrollados independientemente.

Las pantallas son secciones visuales de la aplicación. Sólo una pantalla puede ser visible a la vez y casi se puede considerar únicas en una sola página web-app.

Cada pantalla gestiona la creación, la unión y la visualización de los componentes de interfaz de usuario diferentes que requiere. Cada componente de la interfaz de usuario no es más que un presentador y su objeto vista correspondiente. La comunicación entre los componentes de interfaz de usuario en una pantalla se maneja a través de la EventBus, esto permite el acoplamiento flexible. Cuando un presentador de datos requiere que lo solicite de forma asíncrona desde un DataService compartida. El DataService opcionalmente puede devolver los datos de una memoria caché del lado del cliente, o simplemente llamar directamente al servicio de GWT RPC en el servidor.

4.2.1- Framework GWT

Para el desarrollo de la vista se ha usado GWT o Google Web Toolkit. Es un framework creado por Google que permite ocultar la complejidad de varios aspectos de la tecnología AJAX. Es compatible con varios navegadores. El código es creado en Java y el compilador se encarga de traducirlo a HTML y JavaScript.

4.2.2- SmartGWT

SmartGWT es una API que permite desde GWT utilizar la librería AJAX SmartClient. Está compuesta por un conjunto de componentes (calendarios, árboles de navegación, grids...) que permiten conectar aplicaciones construidas con el servidor, utilizando, por ejemplo, servicios REST.

La aplicación que se construirá utilizará la versión SmartGWT 3.0 con la versión gratuita, la

versión de la librería GWT utilizada es 2.4.

Para desarrollar con este Framework es necesario instalarse un plugin en el navegador.

4.2.3- Prototipo Interfaz Usuario

Se desea desarrollar una aplicación web con una sencilla interfaz de usuario. A continuación se presenta un prototipo de las principales pantallas del sistema, pudiendo variar en las distintas fases de diseño e implementación, incorporando mejoras en la usabilidad y estética.

Este prototipo no pretende ser una presentación exhaustiva de todas las pantallas de la aplicación, sino permitir que en esta fase de la construcción del proyecto, podamos tener una primera aproximación a lo que será el aspecto visual del proyecto una vez finalizado.

Registro de nuevo socio:

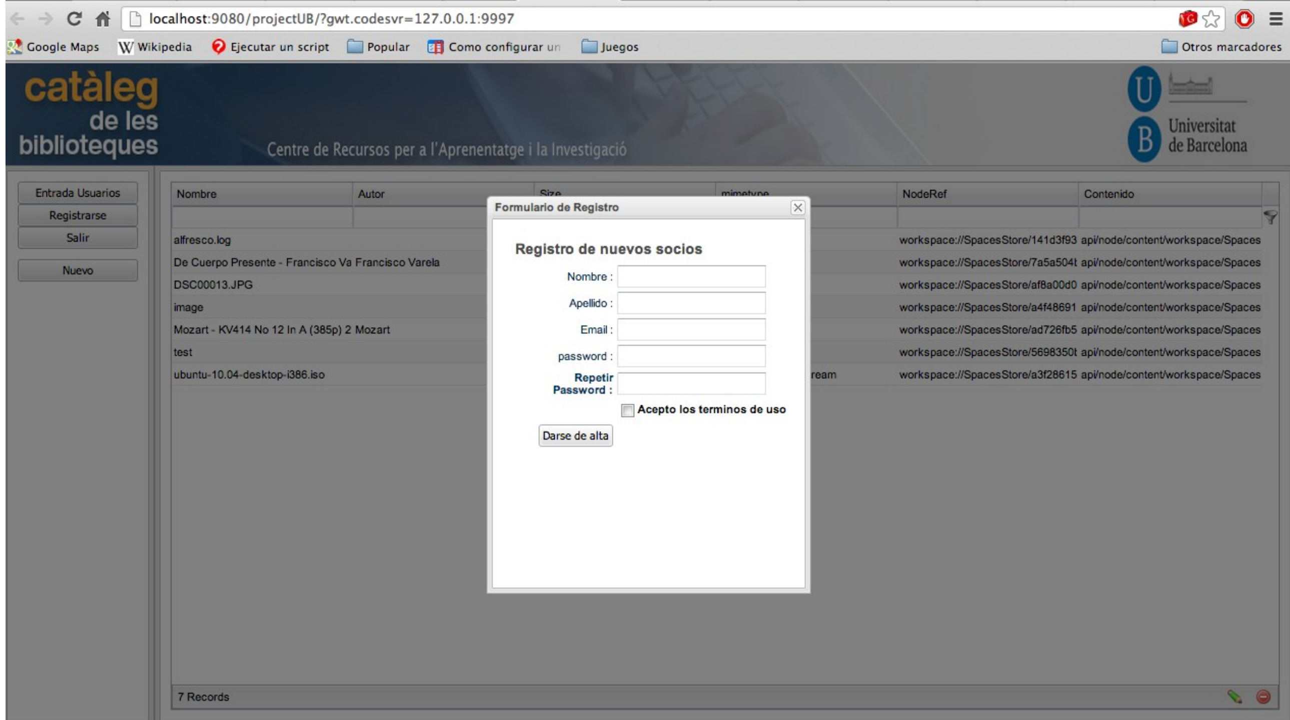

Ilustración 12: Registro de un nuevo socio

Conexión al sistema:

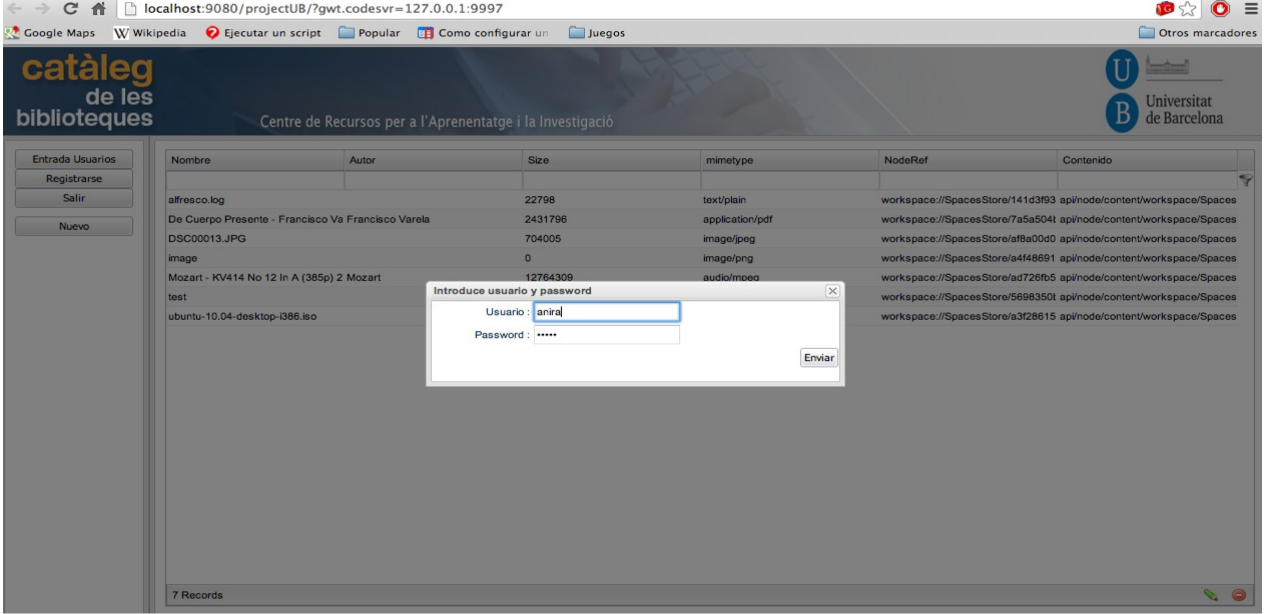

Ilustración 13: Introducir credenciales

4.2.4-Implementación

Ciertos componentes smartGWT son enlazables a fuentes de datos, que proporcionan su estructura y contenido. Los siguientes componentes visuales están diseñados para visualizar, consultar y editar datos estructurados:

DataSource

Los objetos DataSource proporcionan una presentación independiente, independiente de la implementación de un conjunto de campos de datos persistentes. Los DataSources permiten:

- Compartir modelos de datos a través de múltiples aplicaciones y componentes, en el cliente y servidor.
- Mostrar, manipular datos persistentes y modelo de datos relaciones (tales como padres e hijos) a través de componentes visuales (como TreeGrid).
- Ejecutar las operaciones de datos estandarizadas (buscar, ordenar, agregar, actualizar, eliminar) con una función de apoyo tanto en el cliente y como en el servidor para escribir datos, validadores, paginación, claves únicas y más.
- Explotación automatizada que incluye la carga de datos, almacenamiento en caché, filtrado, clasificación, localización y validación.
- Un descriptor de DataSource proporciona los atributos de un conjunto de campos DataSource. Los descriptores DataSource se pueden especificar en formato XML o JSON creado en código Java. El formato se interpreta y es compartido por el cliente y el servidor, mientras que los DataSources creados en Java son utilizados únicamente por el cliente.

Hay cuatro reglas básicas para la creación de datos XML o JSON:

1. Especificar un origen de datos como único atributo ID. El ID se utiliza para enlazar a los componentes visuales, y como un nombre predeterminado para objetos relacionales.
2. Especificar un elemento de campo con un nombre único (único en el origen de datos) para cada campo que se utiliza en un componente de interfaz de enlace de datos.
3. Especificar un tipo de atributo para cada elemento del campo.

4. Marcar exactamente un campo con primaryKey = "true". El campo primaryKey debe tener un valor único para cada objeto de datos (registro) en un DataSource. Un campo primaryKey no es necesario para la lectura de los DataSources, pero es una buena práctica, en general, para actualizar o eliminar datos con sus correspondientes operaciones.

DataSource Documentos

El siguiente DataSource ha sido implementado para mostrar el Grid general.

```
package com.ub.gwt.client.datasource;

import com.smartgwt.client.data.DataSourceField;

public class DocumentsDS extends BaseDS {
    // The DataSource would normally be defined external to any classes that use
    // it.

    public DocumentsDS(String dataUrl, String xpath) {
        super(dataUrl, xpath);
        setDataFormat(DSDataFormat.JSON);

    }

    public DataSourceField[] createDefaultFields() {
        DataSourceField[] result = {
            //  textField("icon","i"),
                textField("displayName", "Nombre"),
                textField("author", "Autor"),
                textField("size", "Size"),
                textField("mimetype", "mimetype"),
                textField("nodeRef", "NodeRef",true),
                textField("contentUrl", "Contenido"),
        };
        return result;
    }

}
```

Ilustración 14: DataSource de documentos

DataSource Usuarios

DataSource usuarios implementado para el caso de uso: “Admin: elimina usuario”.

```
package com.ub.gwt.client.datasource;

import com.smartgwt.client.data.DataSourceField;
import com.smartgwt.client.types.DSDataFormat;

public class UsersDS extends BaseDS {
    // The DataSource would normally be defined external to any classes that use
    // it.

    public UsersDS(String dataUrl, String xpath) {
        super(dataUrl, xpath);
        setDataFormat(DSDataFormat.JSON);

    }

    public DataSourceField[] createDefaultFields() {
        DataSourceField[] result = {
            //  textField("icon","i"),
                textField("userName", "Nombre"),
                textField("email", "Email"),
                textField("url", "Url"),

        };
        return result;
    }

}
```

Ilustración 15: DataSource de Usuarios

4.3- Controlador o lógica/modelo de negocio.

La Lógica de Negocio es un término informal para referirse a la capa intermedia que maneja el intercambio de información entre la Vista y el Modelo; es decir, el Controlador. Sin embargo, en este proyecto se define la Lógica de Negocio como un término que no engloba la transferencia de información entre capas, sino a la lógica aplicativa que permite el correcto funcionamiento del sistema. La Lógica de Negocios, en términos sencillos, es el conjunto de algoritmos que realizan el trabajo que el usuario desea ejecutar. En el proyecto se ha desarrollado con el Framework Spring, a continuación se citan las principales características y sus beneficios clave.

4.3.1- Spring MVC

El Framework Web MVC (modelo-vista-controlador) de Spring está diseñado alrededor de un Servlet (DispatcherServlet) encargado de enviar las peticiones a los diferentes manejadores (handlers) de la aplicación, a partir de la configuración del mapeo, es decir, al configurar este servlet en el fichero web.xml todas las peticiones pasarán por él y se encargará de redirigir a los diferentes manejadores. Los manejadores por defecto se basan en las anotaciones @Controller y @RequestMapping que veremos a continuación.

Spring Framework es un contenedor de objectos basando en el paradigma “inversión del control” o “inyección de dependencias” que será más detallado en la parte del TAD.

Se explican sus características principales, sus beneficios clave, la configuración inicial para comenzar el proyecto con Spring y anotaciones que facilitan la programación JEE.

Servlet Dispatcher

El framework MVC de Spring está diseñado alrededor de un servlet que reenvía las peticiones a los diferentes controladores. Este servlet está plenamente integrado con el contenedor IoC (Inversion of Control) lo que permite usar todas las características del framework Spring.

El **Servlet Dispatcher** es un servlet que hereda de la clase base HttpServlet y se debe declarar en el fichero de configuración web.xml de la aplicación web, en la siguiente imagen se muestra la configuración del web.xml de nuestro proyecto Spring:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5" id="WebApp_ID"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
    xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <display-name>projectUB</display-name>

    <welcome-file-list>
        <welcome-file>index.jsp</welcome-file>
    </welcome-file-list>
    <!-- Jersey Config -->

    <servlet>
        <servlet-name>JerseyWebApplication</servlet-name>
        <servlet-class>com.sun.jersey.spi.spring.container.servlet.SpringServlet</servlet-class>

        <!-- Para que transforme los POJO en JSON -->
        <init-param>
            <param-name>com.sun.jersey.api.json.POJOMappingFeature</param-name>
            <param-value>true</param-value>
        </init-param>
    </servlet>
    <servlet-mapping>
        <servlet-name>JerseyWebApplication</servlet-name>
        <url-pattern>/rest/*</url-pattern>
    </servlet-mapping>

    <!-- Spring config -->
    <listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
    </listener>
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/classes/applicationContext.xml</param-value>
    </context-param>

    <listener>
        <listener-class>org.springframework.web.context.request.RequestContextListener</listener-class>
    </listener>
```

Ilustración 16: web.xml

En la ilustración 13, tal como se puede ver en el mapeo del servlet (*<servlet-mapping>)*, todas las peticiones que empiecen por **/rest/** (*url-pattern)* serán manejadas por el DispatcherServlet.

Controladores

Los controladores proveen acceso al comportamiento de la aplicación, interpretan los inputs del usuario y los transforman en un modelo que es representado al usuario por una vista (modelo-vista-controlador).

Configuración

Spring se configura a partir del *applicationContext.xml.* El ApplicationContext es una interfaz que proporciona información de configuración a la aplicación. Hay varias clases proporcionadas por Spring Framework que implementan esta interfaz y nos ayuda a utilizar la configuración de la aplicación. En la siguiente ilustración se muestra el applicationContext del proyecto.

```
applicationContext.xml
1  <?xml version="1.0" encoding="UTF-8"?>
2  <beans xmlns="http://www.springframework.org/schema/beans"
3      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:p="http://www.springframework.org/schema/p"
4      xmlns:aop="http://www.springframework.org/schema/aop" xmlns:context="http://www.springframework.org/schema/context"
5      xmlns:jee="http://www.springframework.org/schema/jee" xmlns:util="http://www.springframework.org/schema/util"
6      xsi:schemaLocation="http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop.xsd
7          http://www.springframework.org/schema/task http://www.springframework.org/schema/task/spring-task-3.0.xsd
8          http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
9          http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd
10         http://www.springframework.org/schema/util http://www.springframework.org/schema/util/spring-util-3.0.xsd">
11
12
13     <context:annotation-config />
14
15     <!-- Configura beans a partir de clases con la anotacion @Component o derivados
16         (@Controller, @Service, @Repository). -->
17     <context:component-scan base-package="com.ub.projectREST" />
18
19
20
21     <!-- Para que lea las propiedades del pom.xml -->
22     <context:property-placeholder
23         location="classpath:alfrescoProperties.properties, classpath:webscriptsProperties.properties" />
24
25     <!-- Definir los ficheros de configuracion -->
26     <util:properties id="alfrescoProperties"
27         location="classpath:alfrescoProperties.properties" />
28     <util:properties id="webscriptsProperties"
29         location="classpath:webscriptsProperties.properties" />
30 </beans>
```

Ilustración 17: applicationContext.xml

JEE Annotations - @annotations

A partir la versión 3.0, Spring introdujo el modelo de programación basado en anotaciones

En este proyecto se ha programado mediante anotaciones para minimizar la configuración del código. Veremos las anotaciones usadas y las anotaciones más frecuente para facilitar la programación JEE.

Las anotaciones utilizadas durante el proyecto se dividen en dos bloques:

- anotaciones Spring
- anotaciones RESTful Web Service – JAX-RS.

Anotaciones Spring

En la siguiente tabla vemos algunas de las anotaciones usadas en el proyecto con Spring y sus imports:

Anotacion	Package /Import
@Component	import org.springframework.stereotype.Component;
@Controller	import org.springframework.stereotype.Controller;
@Service	import org.springframework.stereotype.Service;
@Repository	import org.springframework.stereotype.Repository;
@Autowired	import org.springframework.beans.factory.annotation.Autowired;
@Scope	import org.springframework.context.annotation.Scope;

Ilustración 18: Spring @annotations

@Component es el estereotipo principal, indica que la clase anotada es un component o Bean de Spring y será cargado en el contexto de la aplicación.

@Repository, @Service y **@Controller** son especializaciones de **@Component** para casos concretos (persistencia, servicios y presentación). Esto significa que puede usarse siempre **@Component** pero lo adecuado es usar sus estereotipos.

En la siguiente imagen se muestra la relación entre ellos:

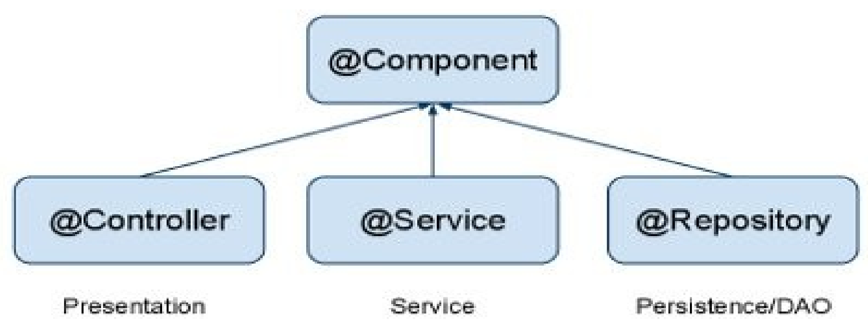

Ilustración 19: Component y derivados

@Autowired

sirve para inyectar automáticamente la dependencia al crear el componente.

@Scope

Tenemos la opción de declarar un scope para cada bean.

Por default, los beans en Spring son Singletons. Un singleton es un patrón de diseño con el que se asegura que solo hay una instancia de un bean en nuestra aplicación. Esto normalmente se logra haciendo que el constructor de la clase sea privado y proporcionando un método estático de fábrica que se encarga de controlar que solo haya una instancia de la clase. Existen 5 tipos de configuración @Scope que serán explicados en el TAD.

RESTful Web Service - JAX-RS Annotations

Jersey es una API Java para servicios web REST basada en anotaciones. Facilita el desarrollo de web services.

Proporciona soporte para Atom's XML format, MIME MultiPart message format , JavaScript Object Notation (JSON) entre otros.

Anotación	Package /Import
@GET	import javax.ws.rs.GET;
@Produces	import javax.ws.rs.Produces;
@Path	import javax.ws.rs.Path;
@QueryParam	import javax.ws.rs.QueryParam;
@POST	import javax.ws.rs.POST;
@Consumes	import javax.ws.rs.Consumes;
@FormParam	import javax.ws.rs.FormParam;
@PUT	import javax.ws.rs.PUT;
@DELETE	import javax.ws.rs.DELETE;

Ilustración 20: JAX-RS @annotations

Cada recurso es llamado **POJO** (Plain Old Java Object) están anotados por su **@Path** el cual designa la ubicación del recurso. Se debe declarar que tipo de operación va a ejecutar, como **@GET, @POST, @DELETE, @PUT**. Si es necesario, se debe declarar que tipo de datos produce **@Produces** o consume **@Consumes** que son especificados con el valor MIME type.

En el ejemplo de código siguiente es un ejemplo muy simple de un recurso utilizando anotaciones spring con anotaciones JAX-RS .

```
package example.jersey.spring;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Component;

// The Java class will be hosted at the URI path "/myresource"
@Path("/myresource")
@Component
@Scope("request")

public class MyResource {

    // The Java method will process HTTP GET requests
    @GET
    // The Java method will produce content identified by the MIME Media
    // type "text/plain"
    @Produces("text/plain")
    public String getIt() {
        return "Hi there!";
    }
}
```

Ilustración 21: Ejemplo Spring y Jersey

4.3.2- Implementación de los casos de uso más importantes

A continuación se muestran los métodos implementados en el modelo de los distintos usuarios.

- Usuario invitado: El usuario invitado, tras rellenar un formulario, se da de alta en la aplicación.

URI	http://localhost:9080/projectUB/rest/users/newUser/
Package	com.ub.projectREST.rest
Clase	LoginWebService.java
Método	- newUser()
Operación	POST

```
@POST
@Path(ConstantesRest.NEW_USER)
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public String newUser(UserBean user) throws HttpException, IOException,
        JSONException {

    log.debug("INICIO - newUser");

    String respuesta = ConstantesRest.RESPUESTA_ERROR;

    if (projectJavaUtils.comprobarParametro(user.getUserName()) != ConstantesRest.PARAMETRO_NO_VALIDO
            && projectJavaUtils.comprobarParametro(user.getFirstName()) != ConstantesRest.PARAMETRO_NO_VALIDO
            && projectJavaUtils.comprobarParametro(user.getLastName()) != ConstantesRest.PARAMETRO_NO_VALIDO
            && projectJavaUtils.comprobarParametro(user.getEmail()) != ConstantesRest.PARAMETRO_NO_VALIDO) {

        respuesta = userService.newUser(user);

    } else
        throw new WebApplicationException(Response
                .status(Status.BAD_REQUEST).entity("Faltan parámetros ")
                .type("text/plain").build());

    log.debug("RESPUESTA  newUser :" + respuesta);

    log.debug(">>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> FIN - newUser");

    return respuesta;

}
```

Ilustración 22: Llamada REST - Registro

– Usuario común:

login; introducir nombre de usuario y password, el usuario se loga en la aplicación.

URI	http://localhost:9080/projectUB/rest/users/getTicket/
Package	com.ub.projectREST.rest
Clase	LoginWebService.java
Método	- getUserTicket()
Operación	GET

```
@GET
@Path(ConstantesRest.GET_TICKET)
@Produces(MediaType.APPLICATION_XML)
public String getUserTicket(
        @QueryParam(ConstantesRest.USER_NAME) String username,
        @QueryParam(ConstantesRest.PASSWORD) String pass)
        throws HttpException, IOException, JSONException {

    log.debug("INICIO - getUserTicket");

    String respuesta = ConstantesRest.RESPUESTA_ERROR;

    if (projectJavaUtils.comprobarParametro(username) != ConstantesRest.PARAMETRO_NO_VALIDO
            && projectJavaUtils.comprobarParametro(pass) != ConstantesRest.PARAMETRO_NO_VALIDO) {

        respuesta = alfrescoSession.iniciarSessionAlfresco(username, pass);

    } else
        throw new WebApplicationException(Response
                .status(Status.BAD_REQUEST)
                .entity("Los parametros username & pass son obligatorios ")
                .type("text/plain").build());

    log.debug("RESPUESTA  getUserTicket :" + respuesta);

    log.debug(">>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> FIN - getUserTicket");

    return respuesta;

}
```

Ilustración 23: Llamada REST - Login

Download: Si el usuario ya está logado en la aplicación puede descargar archivos multimedia del repositorio.

Para mostrar, que se puede llamar directamente al repositorio de Alfresco (ya que es un MVC) el web script se invoca directamente desde la capa de smartGWT.

```
/**
 *
 * download
 *
 */
addRecordDoubleClickHandler(new RecordDoubleClickHandler() {
    public void onRecordDoubleClick(RecordDoubleClickEvent event) {
        Log.debug("file download");
        Record record = event.getRecord();
        final String node = record.getAttribute("displayName");
        Log.debug(node);

        // formatea noderef
        nodeRef = record.getAttribute("nodeRef");
        Log.debug(nodeRef);
        formatNodeRefToId(nodeRef);

        if (ProjectUBSmart.getLoggedUser() != null) {

            SC.ask("Descargar archivo?", new BooleanCallback() {
                String url = "http://localhost:8080/alfresco/s/api/node/content/workspace/SpacesStore/"
                        + formatNodeRefToId(nodeRef)
                        + "/"
                        + node
                        + "?a=true";

                public void execute(Boolean value) {
                    if (value != null && value) {
                        Window.open(url, "_self", "");

                    }
                }
            });
        } else {
            SC.say("Debe hacer login para descargar el contenido. Utilice la opci&oacuten entrar del menu de la izquierda.");
        }
    }
});
```

Ilustración 24: Invocación web script desde smartGWT - Download

– Usuario común: Si el usuario ya está logado en la aplicación puede subir archivos multimedia al repositorio.

URI	http://localhost:9080/projectUB/rest/file/upload
Package	com.ub.projectREST.rest
Clase	FileWebService.java
Método	- fileTransferInviteMultiPart()
Operación	POST

```
@POST
@Path("/upload")
@Consumes(MediaType.MULTIPART_FORM_DATA)
@Produces(MediaType.APPLICATION_XML)
public String fileTransferInviteMultiPart(FormDataMultiPart data,
        @QueryParam(ConstantesRest.TICKET) String ticket)
        throws IOException {
    log.debug("INICIO - fileTransferInviteMultiPart");

    fileService.uploadFile(data, ticket);


    log.debug("FIN - fileTransferInviteMultiPart");
    return "success";
}
```

Ilustración 25: Llamada REST - Upload

– Usuario administrador: deleteUser

Únicamente el administrador tiene permisos para elimina un usuario del repositorio. El siguiente método debe eliminar un usuario comprobando previamente que el usuario que hace la acción es el Administrador.

URI	http://localhost:9080/projectUB/rest/users/deleteUser?username={}
Package	com.ub.projectREST.rest
Clase	UsersWebService.java
Método	- deleteUser()
Operación	DELETE

```
@DELETE
@Path(ConstantesRest.DELETE_USER)
@Produces(MediaType.APPLICATION_JSON)
public String deleteUser(
        @QueryParam(ConstantesRest.USER_NAME) String username,
        @QueryParam(ConstantesRest.TICKET) String ticket) throws HttpException, IOException,
        JSONException {

    log.debug("INICIO - deleteUser");

    String respuesta = ConstantesRest.RESPUESTA_ERROR;

    if (projectJavaUtils.comprobarParametro(username) != ConstantesRest.PARAMETRO_NO_VALIDO
            && projectJavaUtils.comprobarParametro(ticket) != ConstantesRest.PARAMETRO_NO_VALIDO

        ) {

        respuesta = userService.deleteUser(username,ticket);

    } else
        throw new WebApplicationException(Response
                .status(Status.BAD_REQUEST).entity("Faltan parámetros ")
                .type("text/plain").build());

    log.debug("RESPUESTA  deleteUser :" + respuesta);

    log.debug(">>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> FIN - deleteUser");

    return respuesta;

}
```

Ilustración 26: Llamada REST - Delete User

4.4- Modelo

Anteriormente se habló del Modelo como parte esencial de la arquitectura *MVC* y se dijo que el Modelo es la capa que contiene los datos y los procesa a petición del usuario para obtener la información necesaria que será desplegada finalmente al usuario. Es decir, el Modelo es la parte que se encarga de los datos, y como tal, necesita mantenerlos accesibles de manera sencilla para su almacenamiento y recuperación. Con tal propósito se buscó una plataforma que facilitara las funciones de persistencia de datos en la Base de Datos y que permitiera transparentemente trabajar siempre con objetos de Java. Se ha usado Alfresco como Gestor Documental.

4.4.1- Alfresco

Alfresco es un sistema de administración de contenidos libre, incluye un repositorio de contenidos. Este sistema de gestión de contenido web está desarrollado con tecnología Java.

Se distribuye en dos variantes diferentes:

- Alfresco Community Edition: Es software libre, con licencia LGPL de código abierto y estándares abiertos.
- Alfresco Enterprise Edition: Se distribuye bajo licencia de código abierto y estándares abiertos con soporte comercial y propietario a escala empresarial.

Para el proyecto ha sido necesario instalar la versión Alfresco Community Edition. Este gestor documental incluye una serie de webscripts (serán explicados a continuación) que facilitan el intercambio de datos entre nuestro Controlador y el Modelo.

4.4.2- Web scripts.

Un web script es un servicio web enlazado a una URI. Este servicio responde a los métodos HTTP como GET, POST, PUT y DELETE. Es posible llamar a los web scripts existentes en Alfresco o crear propios web scripts. Por ejemplo, se puede crear un propio web script para exponer una interfaz RESTful de un repositorio personalizado.

Un web script se compone de tres archivos:

- un archivo .xml. Contiene la descripción del web script, URI, nombre, etc.
- un archivo .js. Contiene el código JavaScript.
- un archivo .ftl. La presentación de la respuesta (Vista).

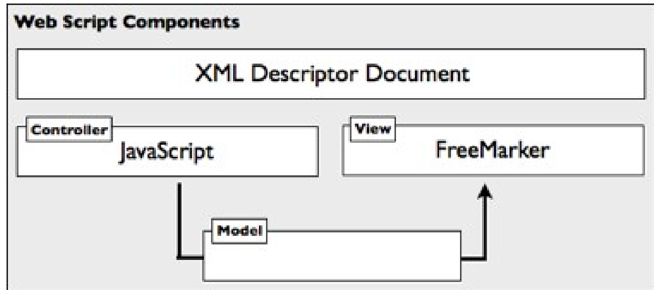

Ilustración 27: Web Script MVC Alfresco

En este proyecto se han usado los web scripts ya existentes en Alfresco, que nos permite usar un código probado y libre de errores para manipular datos en la capa de persistencia o Modelo.

4.4.3- Site Alfresco

Un sitio es un área del proyecto donde se puede compartir contenidos y colaborar con otros miembros del sitio.

Cada sitio tiene un ajuste de visibilidad que marca el sitio como público o privado. Este ajuste controla quién puede ver el sitio y cómo los usuarios se convierten en miembros del sitio.

Para compartir los archivos se ha creado un sitio **UB** que contiene una carpeta llamada **Documentos.**

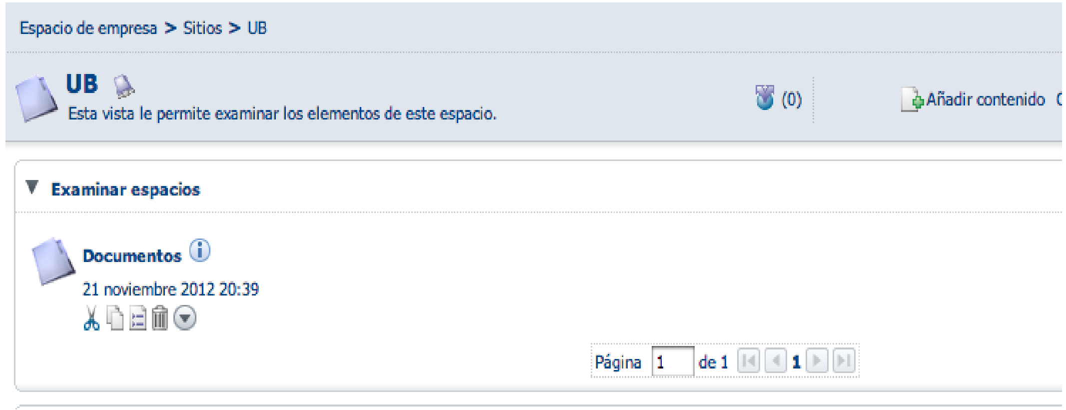

Ilustración 28: Alfresco - Site UB

En la carpeta **Documentos,** es donde se alojaran todos los archivos. Cada vez que se haga un upload o un download se hará sobre esta carpeta.

En la siguiente imagen vemos el contenido de la carpeta **Documentos**

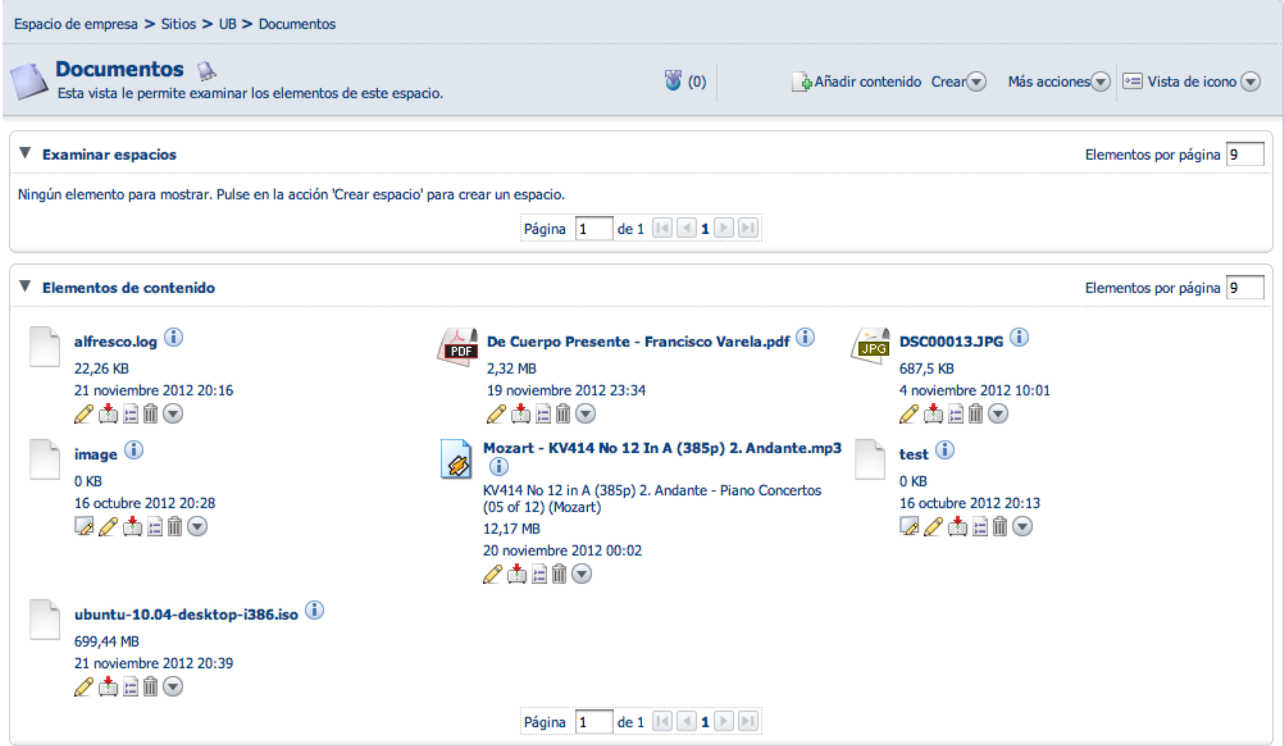

Ilustración 29: Alfresco - Contenido carpeta Documentos

Para mostrar el contenido de la carpeta desde nuestra aplicación, se invoca al web script getDocuments() y lo mostramos en el grid de la aplicación cliente. El resultado se muestra en la siguiente imagen:

Nombre	Autor	Size	mimetype
alfresco.log		22798	text/plain
De Cuerpo Presente - Francisco Va	Francisco Varela	2431796	application/pdf
DSC00013.JPG		704005	image/jpeg
image		0	image/png
Mozart - KV414 No 12 In A (385p) 2	Mozart	12764309	audio/mpeg
test		0	text/plain
ubuntu-10.04-desktop-i386.iso		733419520	application/octet-stream

Ilustración 30: Cliente - Contenido carpeta Documentos desde el GRID de smartGWT

4.4.4- Invocación de los casos de uso más importantes

Las llamadas al modelo se hacen a través de la invocación de los Web Scripts de Alfresco que son invocados a través de una llamada HTTP.

A continuación se presenta la implementación en Spring de las invocaciones a los Web Scripts de Alfresco a partir de los casos de uso definidos para cada tipo de usuario.

Usuario invitado

El usuario invitado, tras rellenar un formulario, se da de alta en la aplicación.

Package	com.ub.projectREST.service.impl
Clase	UserServiceImpl.java
Método	- newUser()

```
public String newUser(UserBean user) {

    String creds = "admin:admin";
    StringBuilder uriCompleta = new StringBuilder("/api/people");

    URI uri = alfrescoSession.getBaseURI(uriCompleta.toString());

    WebResource service = alfrescoSession.getClient().resource(uri);


    MultivaluedMap<String, String> queryParams = new MultivaluedMapImpl();
    service = service.queryParams(queryParams);
    log.debug(service.getURI());

    ClientResponse response = service
            .header("Authorization", "Basic "+ new String(Base64.encode(creds.getBytes())))
            .accept(MediaType.APPLICATION_JSON)
            .type(MediaType.APPLICATION_JSON)
            .post(ClientResponse.class, user);

    return response.getEntity(String.class);

}
```

Ilustración 31: Implementación Web Script Alfresco - Registro

Usuario común

El usuario debe introducir el nombre y password, si es correcto se loga en la aplicación. En caso contrario devuelve mensaje de error.

Package	com.ub.projectREST.login
Clase	AlfrescoSession.java
Método	- iniciarSessionAlfresco()

```
public String iniciarSessionAlfresco(String user, String pass) throws JSONException {

    String ticket = null;
    String jsonResponse = null;
    MultivaluedMap<String, String> queryParams = new MultivaluedMapImpl();
    queryParams.add("u", user);
    queryParams.add("pw", pass);

    logger.debug("Username " + user);
    queryParams.add("format", MediaType.APPLICATION_JSON_TYPE.getSubtype());

    StringBuilder uriWebService = new StringBuilder();
    uriWebService.append(this.endpointAddress);
    uriWebService.append("/api");

    URI uri = UriBuilder.fromUri(uriWebService.toString()).build();

    WebResource serviceLogin = this.client.resource(uri);
    serviceLogin = serviceLogin.path("login").queryParams(queryParams);

    try {
        jsonResponse = serviceLogin.get(String.class);
    } catch (Exception e) {
        throw new WebApplicationException(Response.status(Status.FORBIDDEN)
                .entity(ConstantesRest.ERROR_AUTENTIFICACION)
                .type("text/plain").build());
    }

    JSONObject json = new JSONObject(jsonResponse);
    JSONObject the_json_array = json.getJSONObject("data");
    ticket = the_json_array.getString("ticket");

    StringBuilder xmlResponse = new StringBuilder();
    xmlResponse
            .append("<?xml version=\"1.0\" encoding=\"UTF-8\" standalone=\"yes\"?>");
    xmlResponse.append("<response>");
    xmlResponse.append("<user>" + user + "</user>");
    xmlResponse.append("<ticket>" + ticket + "</ticket>");
    xmlResponse.append("</response>");

    return xmlResponse.toString();
}
```

Ilustración 32: Implementación Web Script Alfresco - Login

– Si el usuario ya está logado en la aplicación puede subir archivos multimedia del repositorio

Package	com.ub.projectREST.services.impl;
Clase	FileServiceImpl.jav
Método	uploadFile()

```
public void uploadFile(FormDataMultiPart form, String ticket){

    log.debug("INICIO: uploadFile");

    try {

        log.debug("ticket:" + ticket);

        StringBuilder urlUploadFile = new StringBuilder(alfrescoSession.getSimpleURL(this.uriUploadFile));
        urlUploadFile.append(".html");
        log.debug("URL " + urlUploadFile);

        WebResource service = alfrescoSession.getClient().resource(urlUploadFile.toString());

        MultivaluedMap<String, String> queryParams = new MultivaluedMapImpl();
        queryParams.add("alf_ticket", ticket);
        service = service.queryParams(queryParams);

        String response = service.type(MediaType.MULTIPART_FORM_DATA_TYPE)
                .accept(MediaType.TEXT_PLAIN).post(String.class, form);
        log.debug("response: " + response);

    } catch (Exception e) {
        e.printStackTrace();
        log.debug("ERROR SUBIENDO ARCHIVO \n");
        return;
    } finally {
        log.debug("FINAL PROCESO!");
    }
}
```

Ilustración 33: Implementación Web Script Alfresco - Upload

Usuario Administrador

Elimina un usuario del repositorio. Comprueba previamente, que el usuario sea el Administrador.

Package	com.ub.projectREST.services.impl
Clase	UserServiceImpl.java
Método	- deleteUser()

```
public String deleteUser(String username , String ticket) {

    StringBuilder uriCompleta = new StringBuilder("/api/people/"+username);

    URI uri = alfrescoSession.getBaseURI(uriCompleta.toString());

    WebResource service = alfrescoSession.getClient().resource(uri);

    System.out.println("Alfresco ticket: " + ticket);

    MultivaluedMap<String, String> queryParams = new MultivaluedMapImpl();
    queryParams.add("alf_ticket", ticket);
    service = service.queryParams(queryParams);
    log.debug(service.getURI());

    ClientResponse response = service.accept(MediaType.APPLICATION_JSON)
            .type(MediaType.APPLICATION_JSON)
            .delete(ClientResponse.class);

    return response.getEntity(String.class);

}
```

Ilustración 34: Implementación Web Script Alfresco - Delete User

5- Anexos

Imágenes del sitio web

A continuación se muestran algunas capturas de pantalla del sitio web desarrollado siguiendo los casos de uso por tipo de usuario.

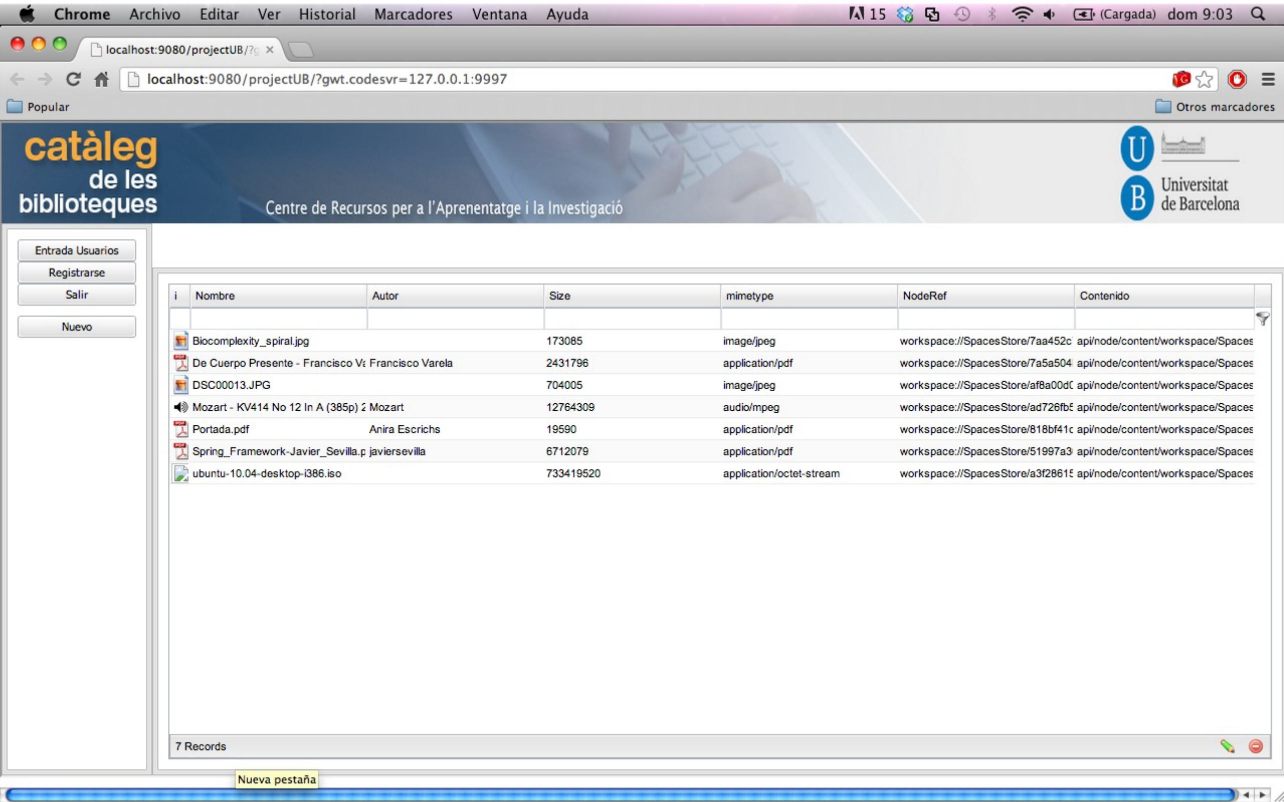

Ilustración 35: Pantalla Inicial

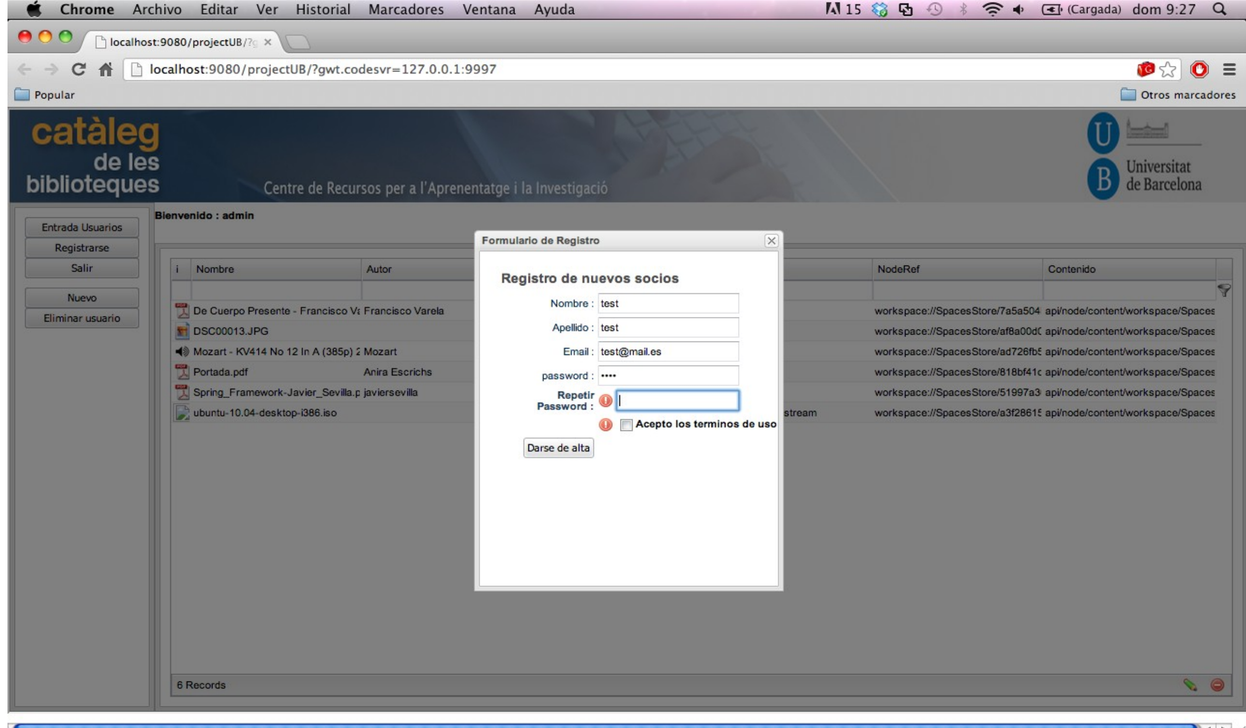

Ilustración 36: Formulario de Registro y validaciones

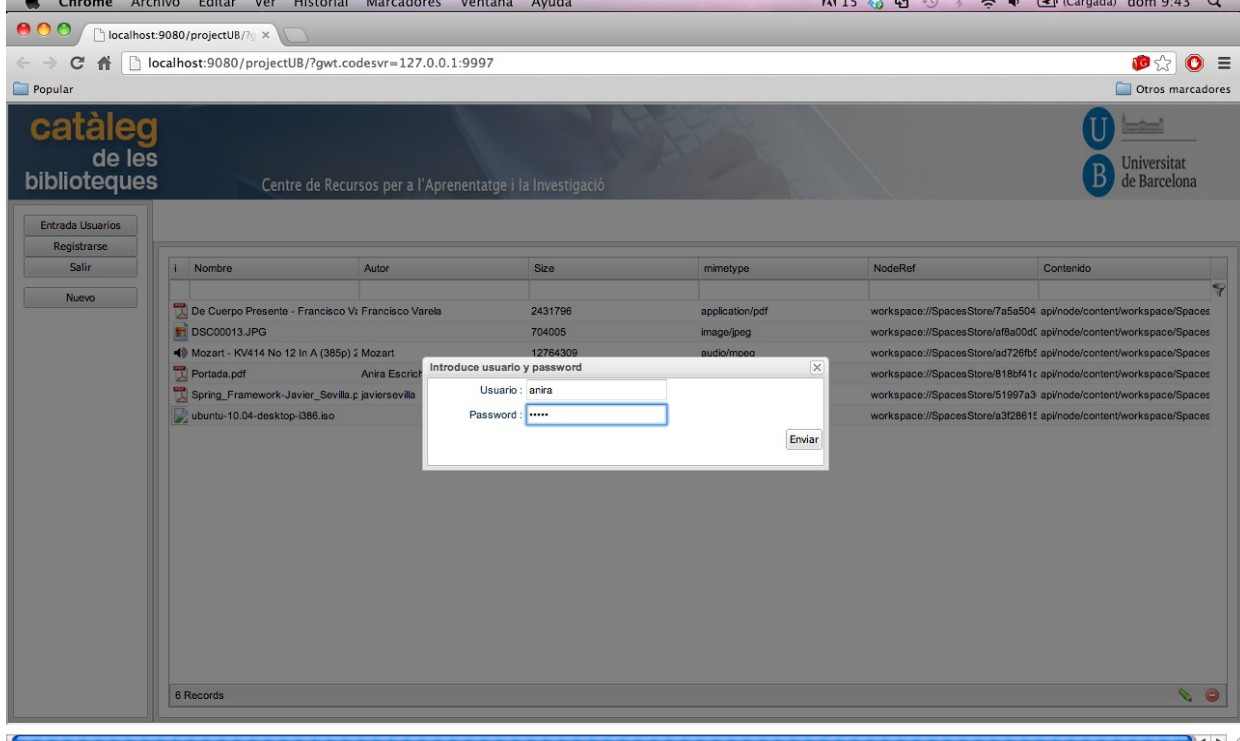

Ilustración 37: Entrada usuario

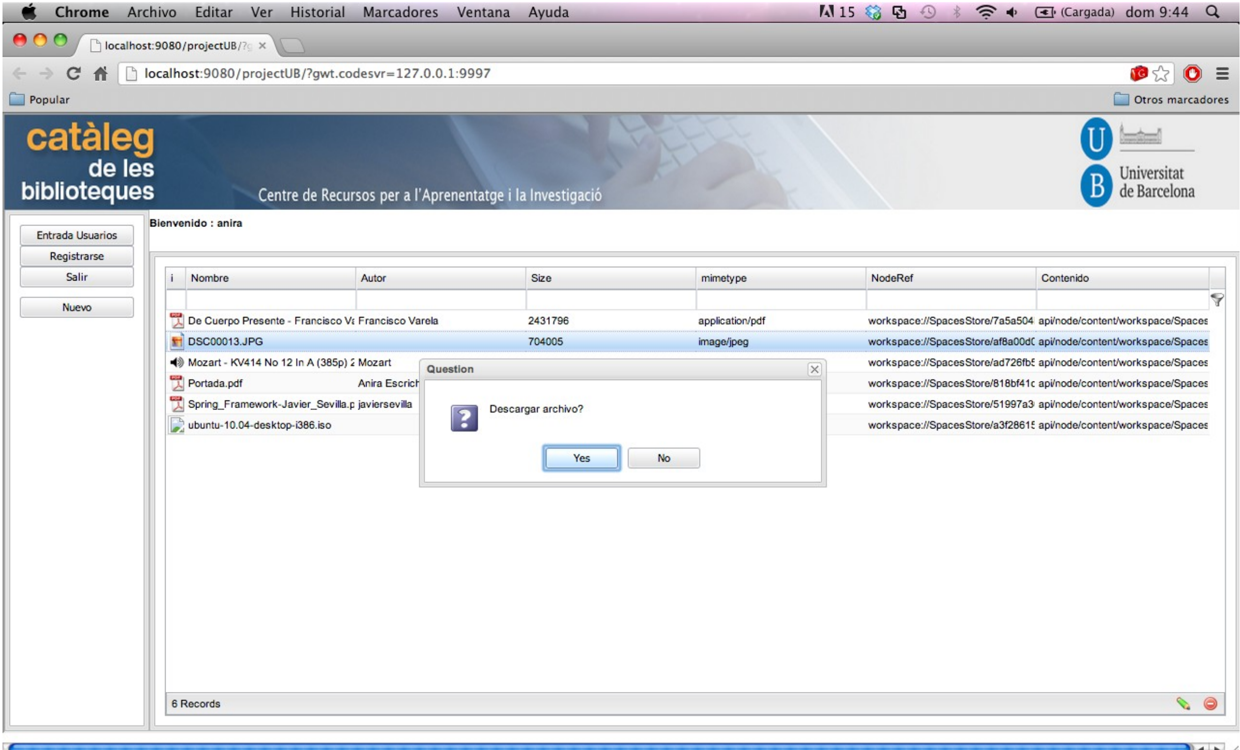

Ilustración 38: Usuario descarga contenido

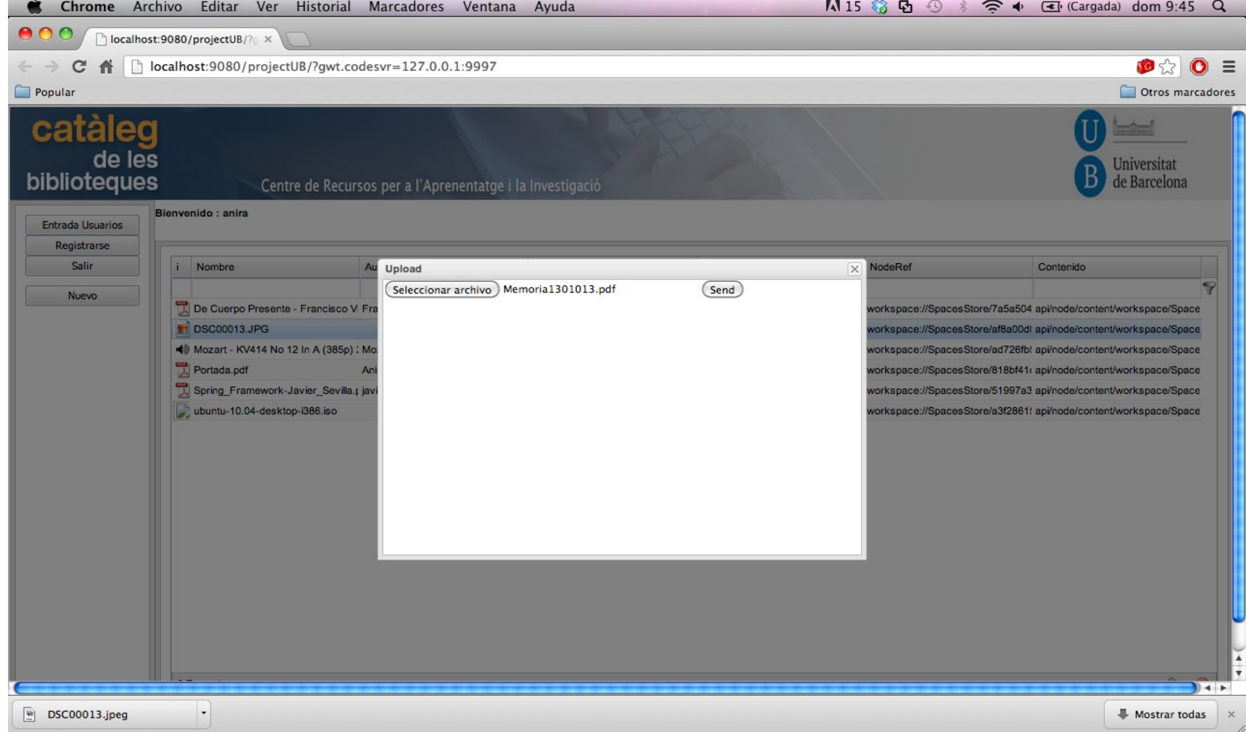

Ilustración 39: Usuario sube contenido

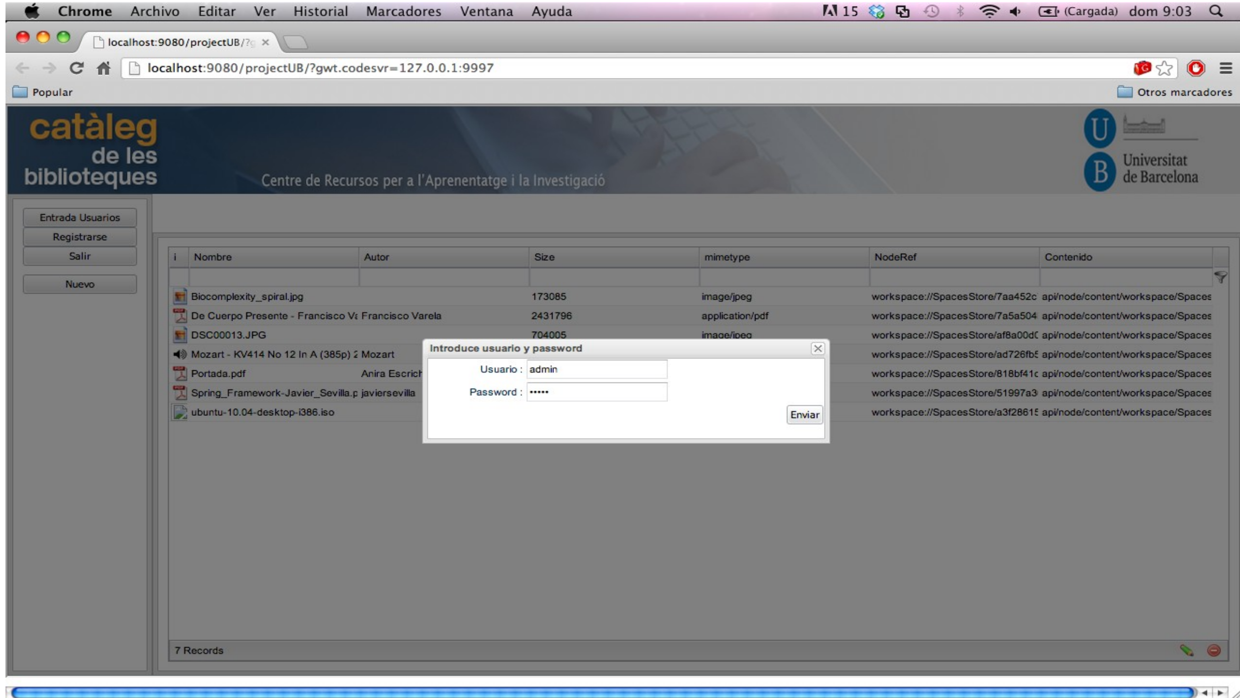

Ilustración 40: Login Admin

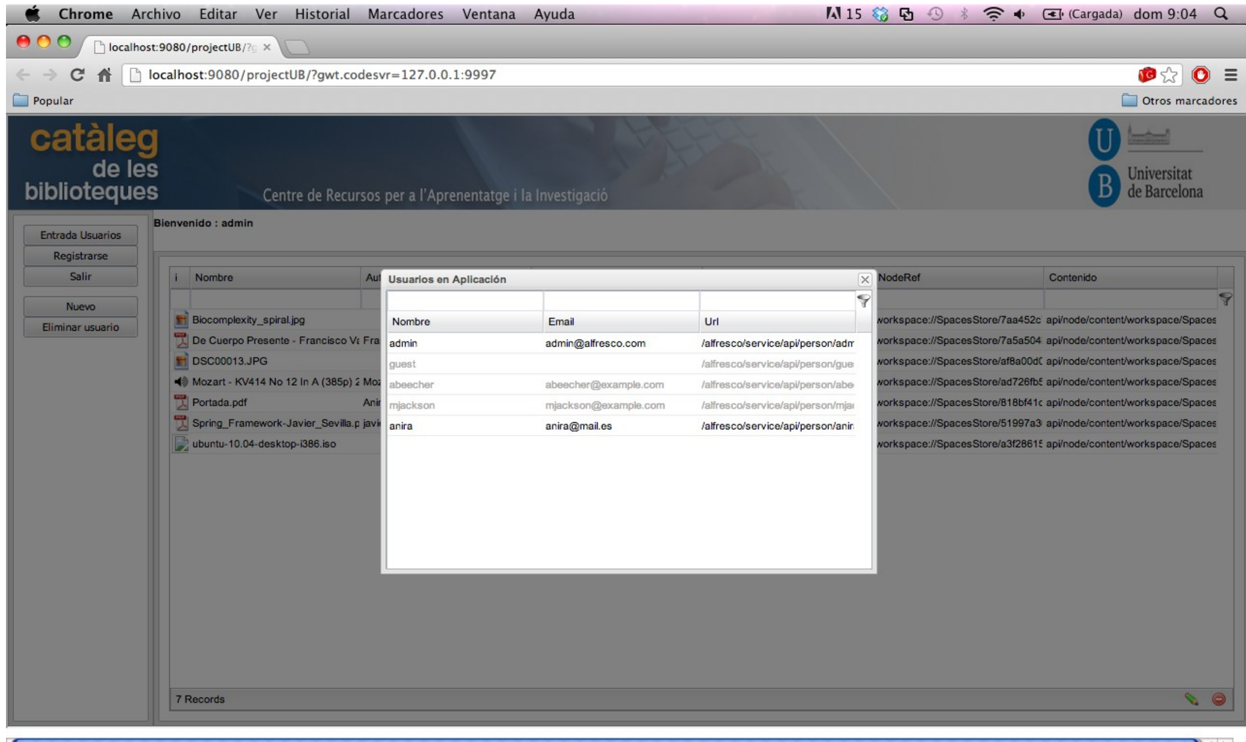

Ilustración 41: Admin Eliminar Usuarios

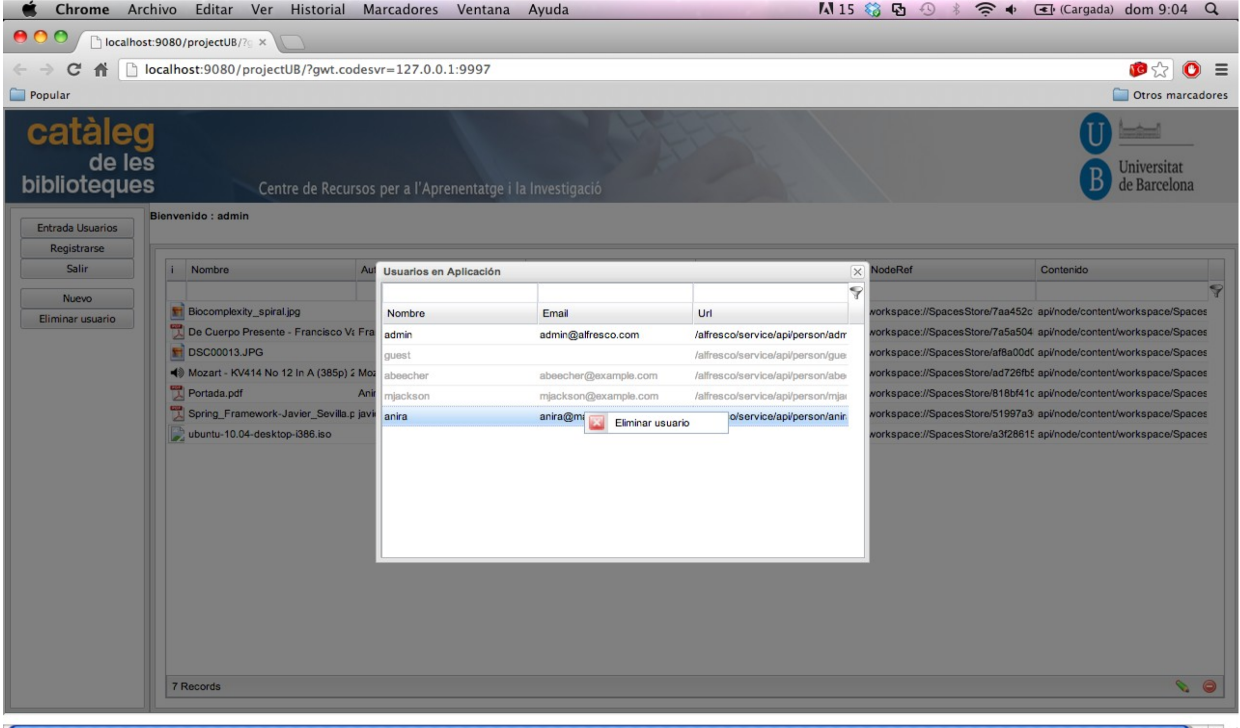

Ilustración 42: Admin Menú Eliminar Usuarios

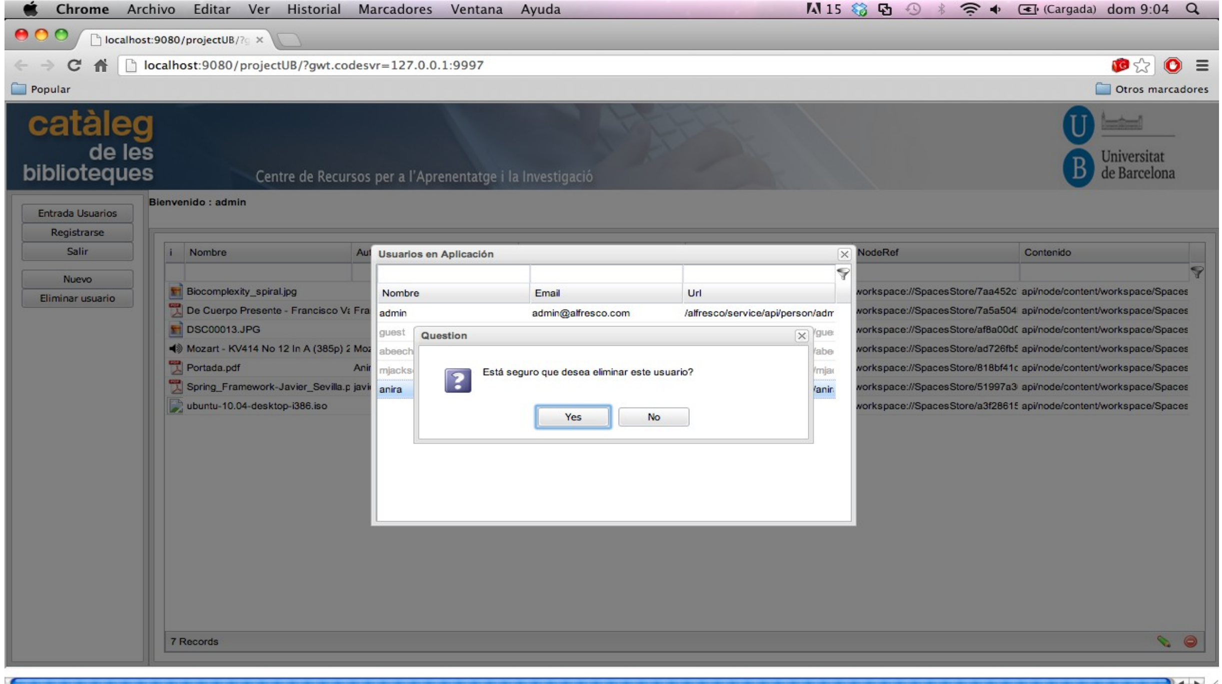

Ilustración 43: Admin Confirma Elimina Usuario

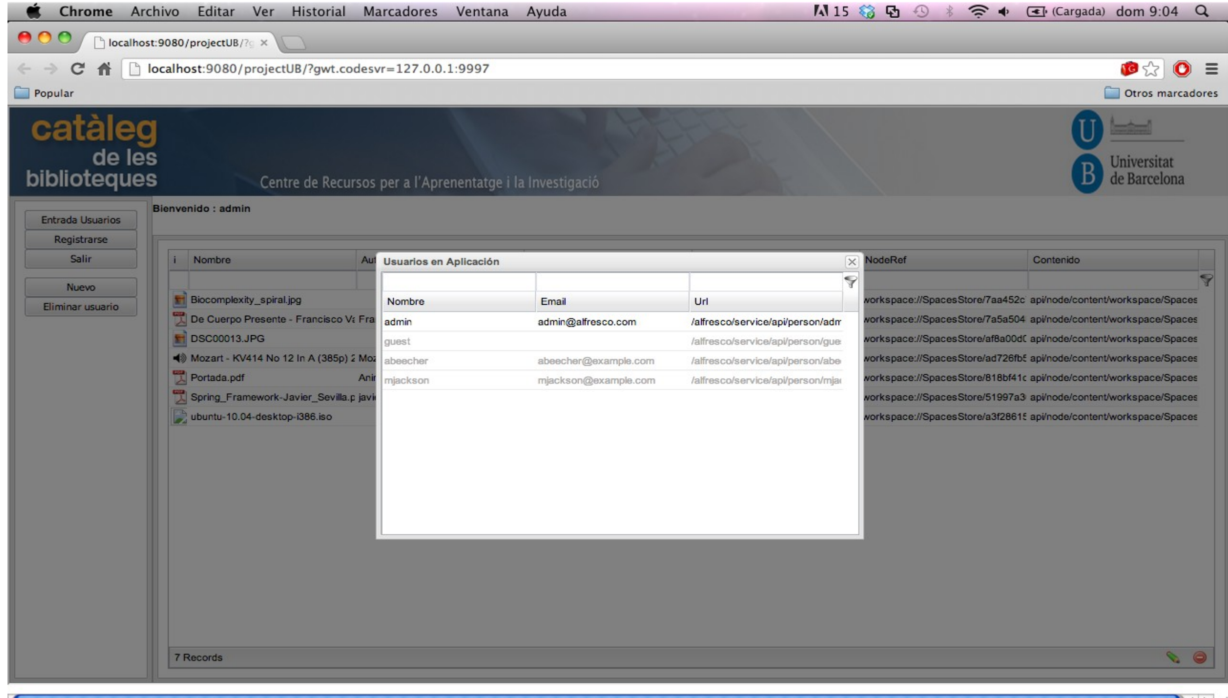

Ilustración 44: Admin Grid Usuario Eliminado

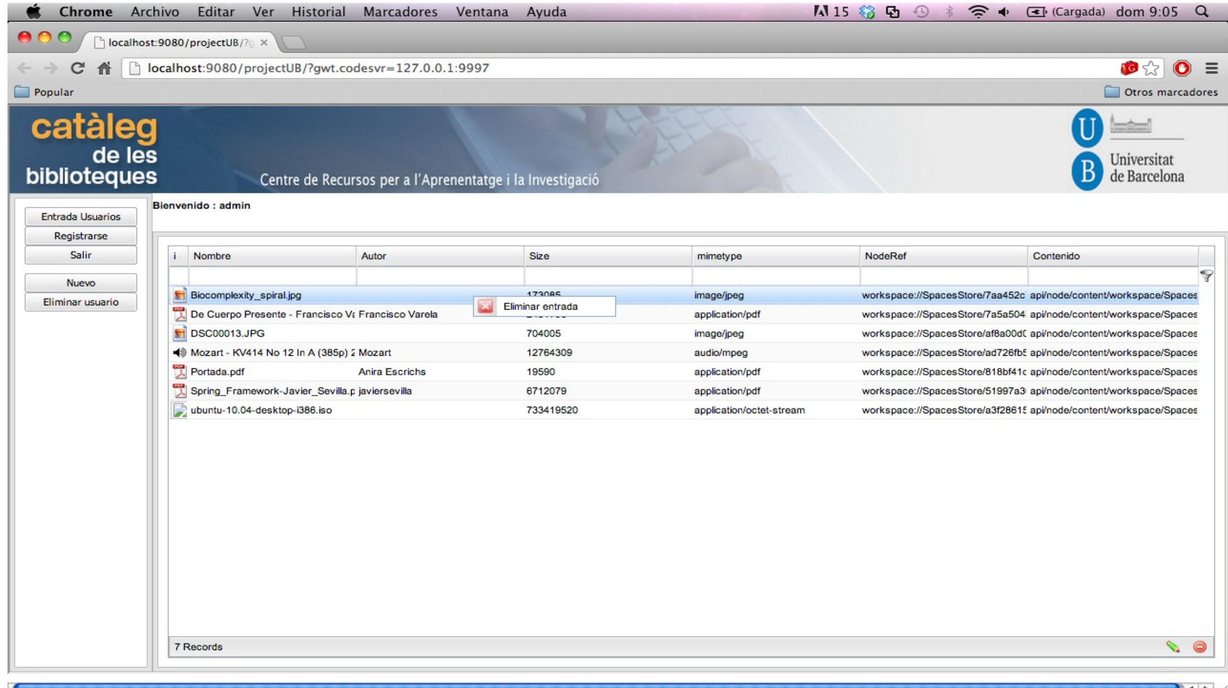

Ilustración 45: Admin Grid Contenido Eliminar

6- Conclusiones

Se pretendía realizar una introducción a la programación de aplicaciones ***REST*** con ***Spring*** 3.0, ***smartGWT*** y ***Alfresco***. Con una Arquitectura robusta y fácil de extender y mantener como es la Arquitectura ***MVC***

Con el excelente framework que nos brida **Google .** Eliminando las pruebas del navegador y depuración, trabajo que hace el **framework GWT** nos ofrece un claro enfoque orientado a objetos para el desarrollo de la interfaz de usuario protegiendo de los errores del navegador y sus peculiaridades. Con el mismo lenguaje de programación que se ha usado en el servidor **JAVA.**

Para crear un aspecto totalmente único y simplificado sólo se requiere conocimientos básicos o nulos de diseño **HTML/CSS Javascript**.

Cómo cualquier aplicación **Spring** MVC, hemos configurado el DispatcherServlet, hemos aprendido a declarar controladores mediante la anotación @Controller. También hemos mapeado peticiones con la anotaciones y por último hemos asociado valores de variables de la URL a parámetros de los métodos controlador mediante su URI. Con todo esto podemos comenzar a construir una aplicación REST en la que cada recurso sea accesible mediante una URL amigable.

En resumen, conociendo un lenguaje de programación tan potente como **JAVA** es posible construir aplicaciones en poco tiempo con los frameworks más potentes usados en la programación; como son el **framework Spring** y la tecnología que nos regala **Google** con su **framework GWT**.

7- Fuentes

AJAX

http://www.w3schools.com/ajax/

http://www.htmleando.com/archivo/tutorial-ajax

GWT

https://developers.google.com/web-toolkit/?hl=es

smartGWT

https://developers.google.com/ https://developers.google.com/web-toolkit/tools/gwtdesigner/features/gwt/smartgwt?hl=en http://code.google.com/p/smartgwt/ http://www.smartclient.com/smartgwt/showcase/

Spring

http://jersey.java.net/nonav/documentation/snapshot/jaxrs-resources.html

http://static.springsource.org/spring/docs/current/spring-framework-reference

Inheritance Mapping Reference: http://docs.jboss.org/hibernate/core/3.5/reference/en/html/inheritance.html

Jersey JAX-RS Annotations: https://wikis.oracle.com/display/Jersey/Overview+of+JAX-RS+1.0+Features

JAXB Annotations: http://docs.oracle.com/javaee/6/api/javax/xml/bind/annotation/package-summary.htm

Alfresco

http://www.alfresco.com/resources/documentation

http://wiki.alfresco.com/wiki/3.0_Web_Scripts_Framewor

Ingeniería inversa:

http://www.visual-paradigm.com/

http://www.omondo.com/

http://www.soyatec.com/euml2/

http://argouml.tigris.org/