ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN
UNIVERSIDAD POLITÉCNICA DE CARTAGENA

Proyecto Fin de Carrera

Implementación de un cliente REST para un servicio de almacenamiento de ficheros basado en metadatos

AUTOR: Alejandro Jerez Fernández
DIRECTOR: Dr. D. Fernando Losilla López

Febrero 2015

Autor	Alejandro Jerez Fernández
Email del autor	alejandro.jerez.fernandez@gmail.com
Director	Dr. D. Fernando Losilla López
Email del director	fernando.losilla@upct.es
Codirector(es)	
Título del PFC	*Implementación de un cliente REST para un servicio de almacenamiento de ficheros basado en metadatos*
Descriptores	Web, REST, Java, Almacenamiento, CouchDB

Resumen

La filosofía o paradigma de programación REST (Representational State Transfer) en la web es cada día más utilizada gracias a sus grandes ventajas entre las que priman la escalabilidad y la simplicidad.

En este proyecto se desarrolla un **cliente REST** para el almacenamiento de ficheros que puede emplearse como herramienta de aprendizaje para futuros alumnos. El cliente y su correspondiente interfaz gráfica han sido desarrollados en lenguaje JAVA y en el lado del servidor hemos empleado una base de datos no-relacional como es **CouchDB.**

Se pretende que este software pueda ser empleado por futuros alumnos de la universidad en algunas asignaturas relacionadas con las TIC.

Titulación	Ingeniería Técnica Telecomunicación Esp. Telemática
Intensificación	
Departamento	Tecnologías de la Información y las Comunicaciones
Fecha presentación	01/02/15

Agradecimientos

Me gustaría dedicar este trabajo a mis padres, puede que sea un tópico pero sin ellos no habría llegado a donde estoy ni a ser lo que soy. Además de un apoyo incondicional a lo largo de todos estos años, han tenido una paciencia que ojalá pueda tener yo en el futuro con mis hijos si es que llegase a tenerlos. Se han sacrificado muchísimo para que yo haya estudiado lo que me gustaba y han tenido incluso más fe que yo en muchos momentos difíciles. Quiero destacar también la importancia de motivar a las personas en las primeras etapas de la vida, algo que a veces está infravalorado pero que sin duda hizo que mis padres me transmitieran curiosidad por la ciencia y amor a la arquitectura e ingeniería.

Además de a mis padres y al resto de mi familia quiero agradecer a mis compañeros de clase haber podido finalizar mis estudios. Creo que tuve la suerte de estar rodeado de los "telecos" más solidarios que pueden haber. Por su puesto también a aquellos profesores que me han dedicado su valioso tiempo, en especial a Fernando Losilla, un ejemplo a seguir por su humildad y profesionalidad.

Por último quiero agradecer todo lo que he aprendido de esas personas que llegan a nuestra vida de paso, pero dejan una gran huella en nosotros y hacen que nos planteemos la vida de una manera diferente.

Índice de contenido

Introducción

Desde el año 2000 la **filosofía REST** ha ido extendiéndose y siendo adoptada por cada vez más desarrolladores de servicios Web que buscan la escalabilidad, simplicidad y amplia tolerancia a fallos en sus aplicaciones. Si bien podemos caracterizar este paradigma por establecer comunicaciones sin estado (Stateless) con los inconvenientes que ello acarrea, esta misma característica es la que nos permite que estos mensajes HTTP sean más simples que los utilizados por ejemplo, por una aplicación basada en SOAP y RPC en cuyo caso deben incluir información concerniente a la sesión como puede ser la autenticación o documentos XML para definir el conjunto de operaciones posibles entre el cliente y el servidor.

Debido la fuerte implementación de aplicaciones con filosofía REST en los servicios Web y su introducción en los programas de asignaturas de titulaciones relacionadas con las Tecnologías de la Información y las Comunicaciones, se ha decidido realizar una **aplicación orientada a la formación** y educación de los futuros alumnos de la universidad mediante una API REST. El objetivo principal es el diseño de una **interfaz gráfica amigable y de uso sencillo** que permita observar a los alumnos el funcionamiento de una comunicación REST. La interfaz gráfica debe ser intuitiva y atractiva para despertar el interés de los alumnos en las primeras tomas de contacto con la asignatura.

Para hacer de la teoría de este paradigma algo práctico y que al mismo tiempo está a la orden del día se ha decidido desarrollar una aplicación que funcione como **cliente de un servicio de almacenamiento** similar al conocido Dropbox. En etapas más avanzadas de la asignatura los alumnos podrán llevar a cabo modificaciones en el código para poder adaptar el servicio a sus necesidades en las prácticas de la asignatura, añadir funcionalidades a la aplicación o utilizar sus funciones para desarrollar futuros proyectos relacionados con con esta filosofía.

En el lado del **servidor** tenemos **Apache CouchDB**, un gestor de bases de datos de código abierto y NoSQL. CouchDB dispone de una API REST para los clientes a través del navegador y además se le pueden añadir funcionalidades mediante unos archivos de configuración, con lo cual la base de datos nos puede devolver datos ya formateados, etc.

Capítulo 1 : Web Services

1.1 Definición de los servicios web

Según el **World Wide Consortium** (W3C) , uno de los organismos responsables de su arquitectura y que regulan la reglamentación de los mismos:

> ***"Un Servicio Web es un sistema de Software diseñado para mantener comunicaciones interoperables máquina a máquina sobre una red"***

Los servicios web son API's a las que se puede acceder dentro de una red y se ejecutan en el sistema que las contiene. Como podemos apreciar, la definición de Web Services engloba muchos tipos diferentes de sistemas pero lo más habitual es encontrar servicios web basados en el intercambio de documentos XML según el estándar SOAP.

Cuando hablamos de **Interoperabilidad** nos referimos a la capacidad de dos o más sistemas con iguales o diferentes características para intercambiar entre sí información y poder usarla. Esta es una característica muy importante ya que permite la comunicación entre máquinas independientemente de las propiedades y las plataformas sobre las que se instalen. La interoperabilidad se consigue mediante el uso de estándares y protocolos abiertos como es el caso de HTTP, XML, WSDL, etc.

1.2 Agentes y servicios

Un servicio Web es una noción abstracta que debe ser implementada por un agente concreto. El **agente** es la unidad de software o hardware que envía y recibe mensajes, mientras que el **servicio** es el recurso que se caracteriza por el conjunto abstracto de funcionalidades que se proporciona. Como decimos, es posible implementar un servicio Web usando un agente un día, y un agente diferente al día siguiente (tal vez escrito en un lenguaje de programación diferente) con la misma funcionalidad. Aunque el agente puede haber cambiado, el servicio Web sigue siendo el mismo.

1.3 Solicitantes y proveedores

El propósito de un servicio Web es proporcionar algunas funciones en nombre de su propietario, ya sea una persona u organización. La entidad **proveedor** es la persona u organización que proporciona un agente apropiado para implementar un servicio particular.

Una entidad **solicitante** es una persona u organización que desee hacer uso del servicio de Internet de una entidad proveedor. Se utilizará un agente solicitante para intercambiar mensajes con el agente proveedor de la entidad proveedor.

En la mayoría de los casos, el agente solicitante inicia este intercambio de mensajes, aunque no siempre.

1.4 Descripción del Servicio Web (WSD)

La **mecánica o protocolo del intercambio de mensajes** se documenta en una Descripción del Servicio Web (WSD). El WSD es una especificación procesable por una máquina de la interfaz del servicio Web, escrita en WSDL. Define el formato de los mensajes así como los tipos de datos, protocolos de transporte y los formatos de serialización de transporte que deben utilizarse entre el agente solicitante y el agente proveedor. También especifica una o más ubicaciones de red a las que un agente proveedor puede invocarse, y puede proporcionar alguna información sobre el patrón de intercambio de mensajes que se espera. En esencia, la descripción del servicio representa un acuerdo que regule los mecanismos de interacción con ese servicio.

Como explicaremos más adelante, este concepto de los servicios web no es necesario cuando utilizamos REST.

1.5 Arquitectura Orientada a Servicios (SOA)

1.5.1 Sistemas Distribuidos

Un sistema distribuido se compone de **diversos agentes** software que deben trabajar juntos para realizar tareas. Además, los agentes en un sistema distribuido **no operan en el mismo entorno de procesamiento**, así que **deben comunicarse** por pilas de protocolos de hardware / software sobre una red. Esto significa que las comunicaciones con un sistema distribuido son obviamente más lentas y menos fiables que si utilizásemos una invocación directa de código en un sistema con memoria compartida. Esto tiene importantes implicaciones arquitectónicas ya que los sistemas distribuidos requieren que los desarrolladores (de infraestructura y aplicaciones) tengan que considerar la latencia impredecible del acceso remoto así como cuestiones de concurrencia y corrección de errores.

Una arquitectura orientada a servicios (**SOA**) es una forma de arquitectura de sistemas distribuidos que se caracteriza típicamente por las siguientes **propiedades**:

- **Contrato de servicios estandarizados**: los servicios establecen un acuerdo de comunicación por el cual negocian su actividad con uno o varios documentos de descripción de servicios (WSDL).
- **Acoplamiento débil de sistemas**: los servicios establecen una relación que reduce las dependencias y unicamente necesita que conozcan la existencia entre ambos.
- **Orientación de red**: Los servicios tienden a orientarse hacia el uso en una red, aunque esto no es un requisito absoluto.
- **Abstracción de servicios**: El servicio se define formalmente en los mensajes intercambiados entre agentes proveedores y solicitantes, y no las propiedades de los propios agentes. La estructura interna de un agente, incluyendo características tales como su lenguaje de implementación, la estructura de, por ejemplo, una base de datos, están abstraídos en la SOA: empleando SOA el agente solicitante no debería necesitar saber cómo se ha implementado el agente proveedor. Un beneficio clave de esta característica son los llamados sistemas heredados. Al evitar cualquier conocimiento de la estructura interna de un agente, se puede incorporar cualquier componente o aplicación software que lo "envuelva", adhiriéndose a la definición del servicio.
- **Servicios sin-estado**: los servicios reducen el consumo de recursos aplazando la gestión de la información de estado cuando sea necesario.
- **Composición de servicios**: los servicios están compuestos en muchos casos a su vez de otros servicios aumentando el tamaño y la complejidad de éstos. Este es un aspecto que se debe regular y optimizar ya que puede ocasionar problemas de rendimiento.
- **Granularidad** : una consideración de diseño para proporcionar un ámbito óptimo y un correcto nivel granular de la funcionalidad del negocio en una operación de servicio ya que tienden a usar un pequeño número de operaciones con mensajes relativamente grandes y complejos.
- **Transparencia de ubicación de servicios**: se refiere a la capacidad de un consumidor de servicios para invocar a un servicio independientemente de su ubicación en la red. Esto también reconoce la propiedad de descubrimiento (uno de los principios fundamentales de SOA) y el derecho de un consumidor para acceder al servicio. A menudo, la idea de la virtualización de servicios también se refiere a la transparencia de ubicación. Aquí es donde el consumidor simplemente llama a un servicio lógico, mientras que un SOA habilita la ejecución del componente de la infraestructura, normalmente un bus de servicios, que mapea este servicio lógico y llama al servicio físico.
- **Plataforma neutral**: Los mensajes se envían en un formato estandarizado de plataforma neutral entregado a través de las interfaces. XML es el formato más obvio que cumple esta limitación.

1.5.2 Estilos estructurales de los Servicios Web

En general los sistemas distribuidos tiene una serie de retos estructurales:

- Problemas introducidos por la latencia y la falta de fiabilidad que ocasiona el transporte.
- La ausencia de memoria compartida entre el invocador y el objeto.
- Los numerosos problemas introducidos por escenarios de fallos parciales.
- El acceso concurrente a recursos remotos.
- La fragilidad de los sistemas distribuidos si una actualización incompatible se introduce alguno de los nodos.

Estos retos se aplican independientemente de si el sistema de objetos distribuidos se implementa utilizando tecnologías COM / CORBA o servicios Web. Los servicios Web no son menos apropiados que sus alternativas si se cumplen los criterios fundamentales para el éxito de las arquitecturas distribuidas. Si se cumplen estos criterios, las tecnologías de servicios Web pueden ser apropiados si los beneficios que ofrecen, en términos de plataforma / neutralidad del vendedor, compensan los problemas de mala implementación que introduzcan.

Sin embargo, el uso de tecnologías de Servicios Web para implementar un sistema distribuido no transforma una arquitectura de objetos distribuidos en una SOA. Tampoco son las tecnologías de Servicios Web necesariamente la mejor opción para implementar SOA. En ocasiones es preferible el uso de COM o CORBA pues en algunos escenarios SOAP / WSDL no añaden suficientes beneficios para justificar sus gastos de funcionamiento, etc.

En general, SOA y los Web Services son más apropiados para aplicaciones que:

- Deban operar a través de Internet, donde no se puede garantizar la fiabilidad ni la velocidad.
- Donde no sea posible administrar la implementación simultánea de una actualización en todos los agentes (proveedores y solicitantes).
- Cuando los componentes del sistema distribuido tengan que ejecutarse en plataformas y productos de distintos proveedores.
- Cuando una aplicación existente necesita ser expuesta para su uso en la red, y puede ser “envuelta” como un servicio Web.

1.5.3 Relación entre las arquitecturas World Wide Web y REST

La World Wide Web funciona como un sistema de información en red que impone varias restricciones:

Los **Agentes identifican** objetos en el sistema, llamados **recursos**, con identificadores uniformes de recursos (**URI**). Estos Agentes pueden **representar**, **describir** y **comunicar** el estado de los recursos a través de las representaciones de los recursos en una gran variedad de formatos de datos (por ejemplo, XML, HTML, CSS, JPEG, PNG). Los agentes intercambian representaciones a través de protocolos que utilizan URIs para identificar directa o indirectamente la dirección de otros agentes y recursos.

Una arquitectura aún más restringida para aplicaciones Web confiables es la denominada **Representational State Transfer** (REST). El REST Web es el subconjunto de la WWW (basado en HTTP) en la que los agentes proporcionan una **semántica de interfaz uniforme** - esencialmente crear, recuperar, actualizar y eliminar - en lugar de las interfaces arbitrarias o específicas de cada aplicación como ocurre con SOAP/WSDL. Utilizando REST, los recursos se manipulan sólo por el intercambio de representaciones. Además, las interacciones REST son **"sin estado"** por lo que el significado de un mensaje no depende del estado de la conversación.

Podemos identificar dos clases principales de Servicios Web:

- Servicios Web compatibles con REST, en cuyo caso el objetivo principal del servicio es manipular representaciones XML de recursos web utilizando un conjunto uniforme de operaciones sin estado.
- Servicios Web arbitrarios, en el que el servicio puede exponer un conjunto arbitrario de operaciones.

Ambas clases de servicios Web utilizan URIs para identificar los recursos y el uso de protocolos web (como HTTP y SOAP 1.2) y formatos de datos XML para la mensajería.

Cabe señalar que SOAP se puede utilizar de una manera compatible o no-compatible con REST.

1.6 Tecnologías principales relacionadas con los Servicios Web

La arquitectura de los Servicios Web implica muchas tecnologías superpuestas e interrelacionadas. Hay muchas formas de visualizar estas tecnologías, así como hay muchas maneras de desarrollar y utilizar servicios Web.

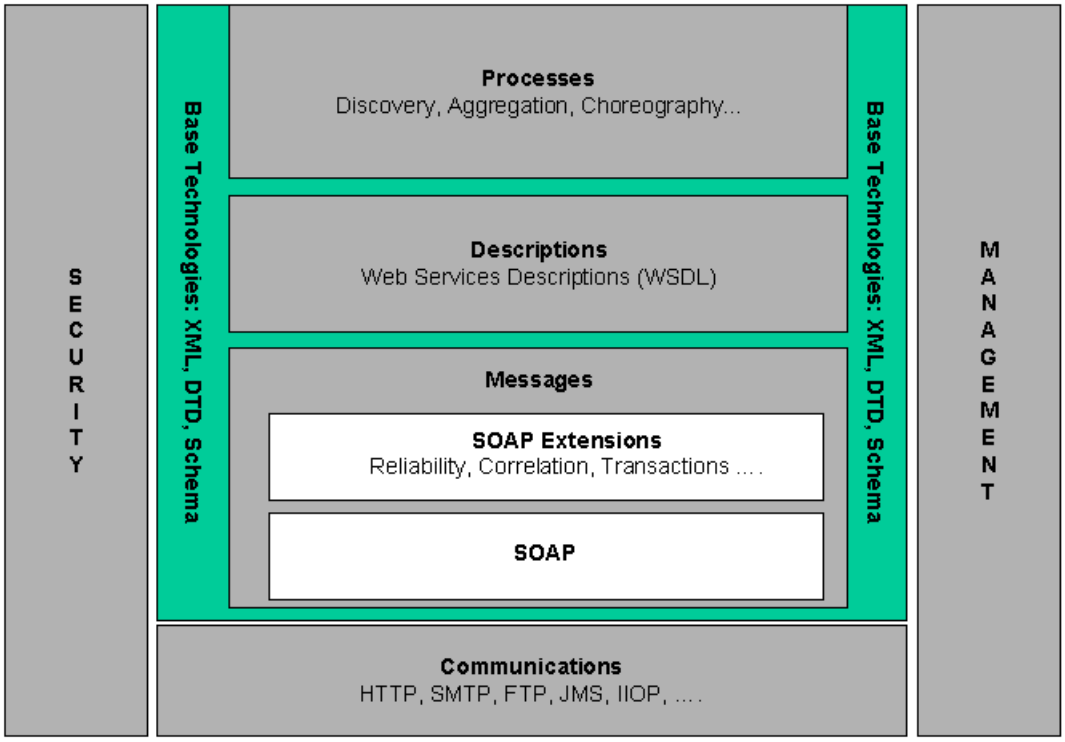

Ilustración 1: Pila estructural de los servicios web.

En esta sección se describen algunas de las tecnologías cuyo papel parece indispensable en relación a esta arquitectura. Como podemos observar en la imagen se puede seguir una estructura de abajo hacia arriba, para poder analizar los servicios Web desde la perspectiva de las herramientas que pueden utilizarse para **diseñar**, **construir** y **desplegar** Servicios Web.

Las tecnologías que consideramos aquí, en relación con la arquitectura, son XML, SOAP y WSDL. Sin embargo, hay muchas otras tecnologías que pueden ser útiles.

1.6.1 XML

XML (e**X**tensible **M**arkup **L**anguage) Resuelve un requisito clave de tecnología que debemos satisfacer en el entorno de los servicios web . Al ofrecer un formato estándar de datos flexible y extensible, reduce significativamente la carga en la implementación de otras muchas tecnologías necesarias para asegurar el éxito de los servicios Web.

El aspecto más importante de XML, a efectos estructurales, es su propia sintaxis, los conceptos del conjunto de información XML (infoset), los schema XML (similares a los DTD) y los espacios de nombres XML.

El “infoset” XML no es un formato de datos por sí mismo, sino un conjunto formal de elementos de información y sus propiedades asociadas, que comprenden una descripción abstracta del documento XML. La especificación Infoset XML proporciona un conjunto coherente y rigurosa de las definiciones para su uso en otras especificaciones que deben remitirse a la información en un documento XML bien formado. A diferencia de otros lenguajes, XML da soporte a bases de datos, siendo imprescindible cuando varias aplicaciones deben comunicarse entre sí y buscamos interoperabilidad entre sistemas.

La serialización de las definiciones Infoset XML de información puede expresarse utilizando XML 1.0 . Sin embargo, esto no es un requisito inherente de la arquitectura. La flexibilidad en la elección del formato de serialización permite una interoperabilidad más amplia entre los agentes del sistema. Una codificación binaria del infoset XML puede ser un sustituto adecuado para la serialización textual. Esta codificación binaria puede ser más eficiente y más adecuada para interacciones máquina a máquina.

1.6.2 SOAP

SOAP proporciona un **Framework estándar y extensible para empaquetar e intercambiar mensajes XML**. En el contexto de esta arquitectura, SOAP 1.2 también proporciona un mecanismo conveniente para hacer referencia a las características mediante el uso de cabeceras.

Los mensajes SOAP pueden ser transportados por una gran variedad de protocolos de red, tales como HTTP, SMTP, FTP, RMI / IIOP, o un protocolo propio de mensajería.

SOAP define tres componentes opcionales: un conjunto de reglas de codificación para expresar instancias de tipos de datos definidos por la aplicación, una convención para representar llamadas y respuestas a procedimiento remoto (RPC), y un conjunto de reglas para el uso de SOAP con HTTP / 1.1.

Mientras que SOAP Versión 1.2 no define más a "SOAP" como un acrónimo, hay dos ampliaciones que reflejan estas diferentes formas en que puede ser interpretado:

- Service Oriented Architecture Protocol : En el caso general, un mensaje SOAP representa la información necesaria para invocar un servicio o refleja los resultados de una invocación de servicio, y contiene la información especificada en la definición de la interfaz de servicio.
- Simple Object Access Protocol: Cuando se utiliza la representación opcional SOAP RPC, un mensaje SOAP representa una invocación de método en un objeto remoto, y la serialización de la lista de argumentos del método debe enviarse desde el entorno local al remoto.

1.6.3 WSDL

WSDL 2.0 es un **lenguaje para describir los Servicios Web**. WSDL describe los servicios web a partir de los mensajes que se intercambian entre los agentes solicitante y el proveedor. Los mensajes en sí mismos se describen de forma abstracta y posteriormente se unen a un protocolo de red y formato de mensaje concretos.

Las descripciones del Servicio Web se pueden asignar a cualquier lenguaje de programación, plataforma o sistema de mensajes. Se pueden implementar servicios Web para la interacción a través de navegadores o directamente dentro de una aplicación. La aplicación podría ser implementada usando multitud de leguajes. Mientras que el emisor y el receptor estén de acuerdo en la descripción del servicio (archivo WSDL), la implementación de dichos agentes será independiente.

Capítulo 2 : Representational State Transfer (REST)

En este capítulo vamos a hablar del paradigma Transferencia de estado representacional con siglas en inglés “REST”. Es un estilo estructural de programación para sistemas hipermedia distribuidos. REST es un estilo híbrido derivado de varios de los estilos estructurales basados en la red descritos en el capítulo anterior y combinado con las limitaciones adicionales que definen una interfaz de conector uniforme.

2.1 Derivados de REST

La lógica de diseño que hay tras esta arquitectura Web puede ser descrita como un estilo estructural que consiste en un conjunto de restricciones aplicadas a los elementos de la arquitectura. Al examinar el impacto de cada restricción que se agrega, podemos identificar las propiedades inducidas en el sistema. Las restricciones opcionales se pueden aplicar para formar arquitecturas que reflejen mejor las propiedades deseadas para una arquitectura Web moderna. Esta sección proporciona una visión general de REST y analiza posibles derivaciones de esta arquitectura. Las secciones posteriores describirán con más detalle las limitaciones específicas que componen la filosofía REST.

2.1.1 Empezando con el Estilo Nulo

Hay dos puntos de vista sobre el proceso de diseño estructural. El primero es que un diseñador comience desde cero y desarrolle sobre la marcha una arquitectura a partir de otros componentes hasta que satisfagan las necesidades del sistema previsto. La segunda es que el diseñador comience con el sistema completo, sin limitaciones para, a continuación, aplicar restricciones a los elementos del sistema. Mientras que el primer procedimiento enfatiza la creatividad y la visión sin límites, el segundo hace hincapié en la moderación y comprensión del sistema. REST se ha desarrollado utilizando el último procedimiento.

El **Estilo Nulo** es simplemente un **conjunto vacío de limitaciones**. Desde un punto de vista estructural, el estilo nulo describe un sistema en el que no hay límites distinguidos entre componentes. Es el punto de partida para nuestra descripción de REST.

2.1.2 Cliente-Servidor

Las primera restricción en nuestro estilo híbrido es una **estructura cliente-servidor**. La separación de los problemas es la idea principal de la restricción cliente-servidor. Al separar los problemas de interfaz de usuario de los de almacenamiento de datos, mejoramos la portabilidad de la interfaz de usuario a través de múltiples plataformas y la escalabilidad mediante la simplificación de los componentes del servidor. Tal vez el beneficio más importante sea que la separación permite que los componentes evolucionen de forma independiente.

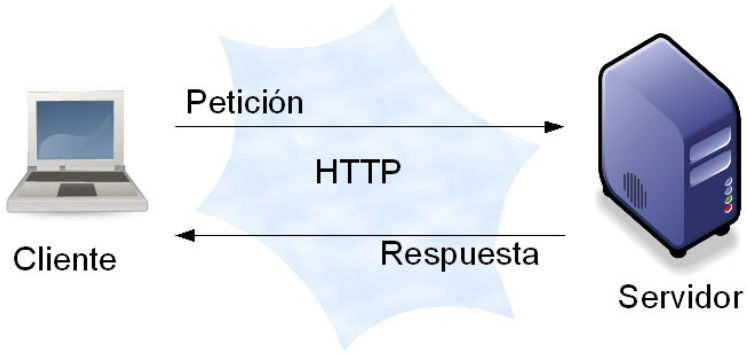

Ilustración 2: Topología Cliente-Servidor

2.1.3 Sin estado

A continuación añadimos una restricción a la interacción cliente-servidor: la comunicación debe ser sin estado, de manera que cada solicitud de cliente a servidor debe contener toda la información necesaria para comprender la solicitud, y no puede depender de cualquier contexto almacenado en el servidor. Por lo tanto, **el estado de sesión se mantiene totalmente en el cliente**. Aunque a primera vista parece que esta restricción sea solamente una desventaja hace que los mensajes intercambiados sea mucho más simples y uniformes y además mejora la tolerancia del sistema ante los fallos relacionados con la conexión.

Esta restricción induce las propiedades de **visibilidad**, **confiabilidad** y **escalabilidad**. La visibilidad se mejora debido a que no se necesita ningún sistema de monitoreo que busque la referencia de la solicitud de a fin de determinar su naturaleza exacta. Ha mejorado la confiabilidad, ya que facilita la tarea de recuperación de fallos parciales. La escalabilidad se mejora debido a que el no tener que almacenar el estado entre solicitudes permite que el servidor pueda liberar los recursos rápidamente y simplifica aún más la implementación porque el servidor no tiene que administrar el uso de los recursos a través de las solicitudes.

Como la mayoría de elecciones estructurales, la restricción sin-estado implica un **sacrificio**. La desventaja es que **puede disminuir el rendimiento** de la red mediante el aumento de los datos repetitivos (sobrecarga por interacción) enviados en una serie de peticiones, ya que los datos no se puede dejar en el servidor en un contexto compartido. Además, colocando el estado de la aplicación en el lado del cliente **reduce el control del servidor** sobre el comportamiento coherente de la aplicación, ya que se vuelve dependiente de la semántica que utilicen las múltiples versiones del cliente.

2.1.4 Caché

Con el fin de mejorar la eficiencia de la red, le añadimos las limitaciones de memoria caché para formar un cliente-servidor con caché y sin estado. La restricción de caché requiere que los datos de una respuesta a una petición sean etiquetados implícita o explícitamente como cacheables o no cacheables. Si la respuesta es cacheable, se le da permiso a la memoria caché del cliente o de los equipos intermedios para volver a utilizar los datos de respuesta para las solicitudes posteriores equivalentes.

La ventaja de añadir de la limitación de memoria caché es que tienen el puede eliminar parcial o completamente algunas interacciones, mejorando la **eficiencia**, **escalabilidad** y **rendimiento** percibida por el usuario mediante la **reducción de la latencia** media de una serie de interacciones. La desventaja, sin embargo, es que la caché puede disminuir la fiabilidad si los datos quedan obsoletos en su memoria o difieren significativamente de los datos que se habrían obtenido si hubiera sido enviada la solicitud directamente al servidor.

Los cimientos de la arquitectura Web, son definidos por el conjunto de restricciones cliente-servidor con caché y sin estado. Es decir, el diseño racional presentado para la arquitectura Web antes de 1994 se centró en la interacción cliente-servidor sin estado para el intercambio de documentos estáticos a través de Internet. Los protocolos para las interacciones que se comunican tenían apoyo rudimentario para cachés no compartidas, pero no limitan la interfaz a un conjunto coherente de semántica para todos los recursos. En cambio, la Web se basó en el uso de una librería común cliente-servidor (libwww) para mantener la coherencia entre aplicaciones web.

2.1.5 Interfaz Uniforme

La característica principal que distingue a la arquitectura REST de otros estilos basados en la red es su énfasis en una interfaz uniforme entre los componentes. Aplicando el principio de ingeniería de software de generalidad de los componente de la interfaz, la arquitectura general del sistema se **simplifica** y la **visibilidad** de las interacciones se mejora. Las implementaciones están disociadas de los servicios que prestan, lo que favorece la capacidad de evolución independiente. La desventaja, sin embargo, es que una interfaz uniforme degrada la eficiencia, ya que la información se transfiere de forma normalizada en lugar de ser específica a las necesidades de cada aplicación. La interfaz REST está diseñada para ser eficiente para la transferencia de datos hipermedia a gran escala, optimizada para el caso más común en la Web, pero no resulta tan eficiente si es utilizada para otras formas de interacción estructural.

Con el fin de obtener una interfaz uniforme, se necesitan múltiples restricciones estructurales para guiar el comportamiento de los componentes. REST se define por **cuatro restricciones de interfaz: Identificación de recursos, manipulación través de representaciones, mensajes auto-descriptivos e hipermedia como motor de estado de la aplicación.**

2.1.6 Sistema por Capas

Con el fin de mejorar aún más el comportamiento de los requisitos a escala de Internet, añadimos las limitaciones del sistema por capas. El estilo de sistema por capas permite una arquitectura que se compone de **capas jerárquicas limitando el comportamiento y la complejidad de los componentes** de tal manera que cada uno no pueda "ver" más allá de la capa con la que están interactuando. Al restringir el conocimiento del sistema de una sola capa, ponemos un límite en la complejidad general del sistema y promovemos la independencia de cada capa. Las capas pueden ser utilizadas para encapsular servicios heredados y para proteger a los nuevos servicios de clientes heredados. Se simplifican los componentes traspasando funcionalidades de uso poco frecuente a un intermediario compartido. Los intermediarios también pueden ser utilizados para **mejorar la escalabilidad** del sistema al **controlar el equilibrio de carga** de los servicios a través de múltiples redes y procesadores.

La principal desventaja de los sistemas por capas es que **añade un sobre-coste de recursos y latencia** en el procesado de datos, reduciendo el rendimiento percibido por el usuario. Que un sistema basado en red soporte restricciones de uso de caché puede ser compensado por los beneficios del almacenamiento en caché compartida entre intermediarios. La inclusión de cachés compartidas en los límites de un dominio organizativo puede conseguir importantes beneficios en el rendimiento. Estas capas también permiten aplicar las políticas de seguridad a los datos que cruzan la frontera de la organización, como es requerido por los cortafuegos.

Aunque la interacción REST es bidireccional, los flujos de datos hipermedia a gran escala pueden ser procesados de forma independiente como una red de flujo de datos, con componentes de filtro aplicados selectivamente a la secuencia de datos con el fin de transformar el contenido a medida que pasa. Empleando REST, los nodos intermedios pueden transformar activamente el contenido de los mensajes porque éstos son auto-descriptivos y su semántica es visible para los intermediarios.

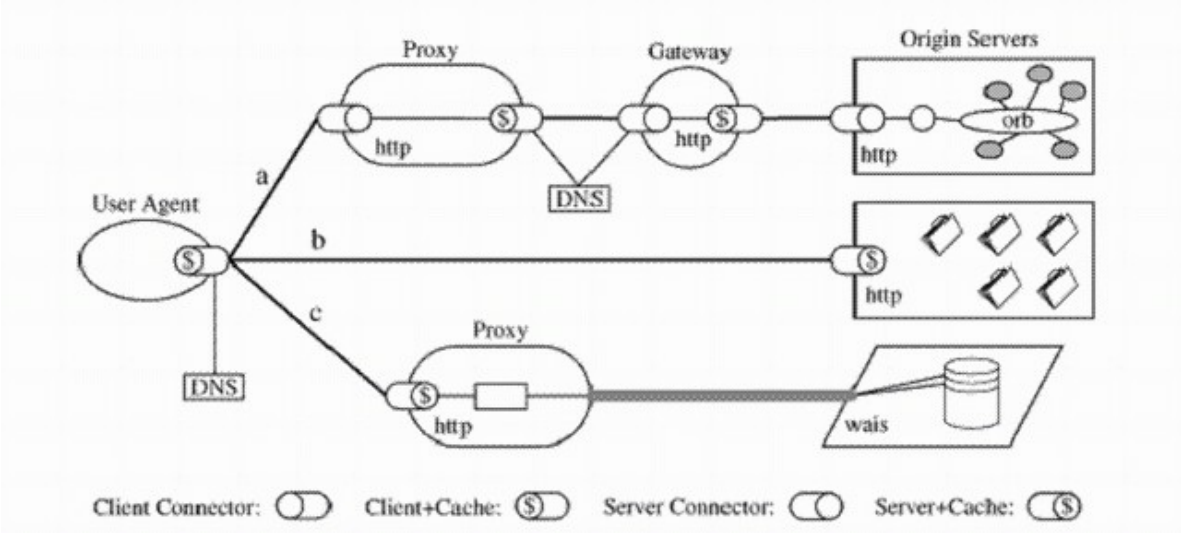

Ilustración 3: Estructura de un sistema REST basado en capas y uso de caché

2.1.7 Código bajo demanda

La adición final a nuestro conjunto de restricciones para REST proviene del estilo code-on-demand. REST permite la funcionalidad de **descarga y ejecución de código** en forma de applets o scripts **en la parte del cliente**. Esto simplifica los clientes mediante la reducción del número de características necesarias que deben ser pre-implementadas. Características que podrán ser descargadas después del despliegue y que mejoran sin duda la extensibilidad del sistema. Sin embargo, también reduce la visibilidad, y por lo tanto es sólo una restricción **opcional** dentro REST.

El sentido de una restricción opcional puede parecer contradictorio. Sin embargo tiene un propósito en el diseño de arquitecturas de sistemas que contengan múltiples limites organizativos. Esto significa que el sistema sólo obtiene el beneficio (y sufre las desventajas) de las restricciones opcionales cuando éstas se apliquen total o parcialmente en el sistema. Por ejemplo, si todo el software para clientes de una organización es conocido por apoyar los applets de Java, entonces los servicios dentro de esa organización se pueden construir de tal manera que ganen el beneficio de una mayor funcionalidad a través de clases de Java descargables. Al mismo tiempo, sin embargo, el firewall de la organización puede impedir la transferencia de los applets de Java de fuentes externas, y para el resto de la Web, aparecerá como si los clientes no admiten código-on-demand. Una restricción opcional nos permite diseñar una arquitectura que soporta el comportamiento deseado en el caso general, pero con la ventaja de poder desactivarla en algunos contextos.

2.2 Elementos estructurales de REST

El estilo Transferencia de estado representacional (REST) es una abstracción de los elementos estructurales dentro de un sistema hipermedia distribuido. REST pasa por alto los detalles de implementación del componente y la sintaxis de protocolo con el fin de centrarse en las funciones de los componentes, las restricciones sobre su interacción con otros componentes y su interpretación de los elementos de datos importantes. Abarca las limitaciones fundamentales sobre los componentes, conectores, y los datos que definen la base de la arquitectura Web, y por lo tanto la esencia de su comportamiento como una aplicación basada en la red.

A diferencia del modelo de sistema de objetos distribuidos, donde todos los datos se encapsulan y ocultan por los componentes del sistema, la naturaleza y el estado de los elementos de datos de una arquitectura es un aspecto clave de REST. La razón de este diseño se puede ver en la naturaleza de hipermedia distribuida. Cuando se selecciona un enlace, la información tiene que ser transferida desde su ubicación donde se almacena hasta la ubicación donde será usada por, en la mayoría de los casos, un lector humano. Esto es diferente a muchos otros paradigmas distribuidos de procesamiento, en los que es posible, y por lo general más eficiente, mover el "agente de procesado" a los datos en lugar de mover los datos al procesador.

Data Element	Modern Web Examples
resource	the intended conceptual target of a hypertext reference
resource identifier	URL, URN
representation	HTML document, JPEG image
representation metadata	media type, last-modified time
resource metadata	source link, alternates, vary
control data	if-modified-since, cache-control

Elementos de Datos en REST

2.2.1 Recursos e identificadores de recurso

La abstracción de información importante en REST es un recurso. **Cualquier información que puede ser nombrada puede ser un recurso**: un documento o imagen, un servicio temporal (por ejemplo, "La temperatura actual en Sierra Nevada"), una colección de otros recursos, un objeto no virtual (como por ejemplo una persona), etc... En otras palabras, cualquier concepto que pueda ser el objetivo de referencia de hipertexto de un autor debe encajar dentro de la definición de un recurso. Un recurso es un mapeo conceptual a un conjunto de entidades, no la entidad correspondiente al mapeo en un punto particular en el tiempo.

2.2.2 Representaciones

Los componentes REST realizan acciones en un recurso mediante el uso de una representación para capturar el **estado actual de ese recurso** y la transfieren entre los componentes. Una representación es una secuencia de bytes, que incluye metadatos para describir esos bytes. Otros nombres de uso común, pero menos precisos para una representación incluyen: documentos, archivos, mensaje HTTP, instancia o variable.

2.2.3 Conectores

REST utiliza varios tipos de conectores, que se resumen en la siguiente tabla, para **encapsular las actividades de acceso a los recursos y transferir representaciones de recursos**. Los conectores presentan una interfaz abstracta para los componentes de comunicación. Mejoran la simplicidad, proporcionando una clara separación de las preocupaciones y ocultan la implementación subyacente de los recursos y mecanismos de comunicación. La generalidad de la interfaz también permite sustitución: Si el acceso de los usuarios al sistema es a través de una interfaz abstracta, la aplicación puede ser reemplazada sin afectarlos. Como los conectores gestionan la comunicación de red para un componente, la información puede ser compartida a través de múltiples interacciones con el fin de mejorar la eficiencia y capacidad de respuesta.

Connector	Modern Web Examples
client	libwww, libwww-perl
server	libwww, Apache API, NSAPI
cache	browser cache, Akamai cache network
resolver	bind (DNS lookup library)
tunnel	SOCKS, SSL after HTTP CONNECT

Conectores en REST y sus ejemplos

2.2.4 Componentes

Component	Modern Web Examples
origin server	Apache httpd, Microsoft IIS
gateway	Squid, CGI, Reverse Proxy
proxy	CERN Proxy, Netscape Proxy, Gauntlet
user agent	Netscape Navigator, Lynx, MOMspider

Componentes REST y sus ejemplos

El **agente de usuario** utiliza un **conector de cliente** para iniciar una solicitud y se convierte en el destinatario final de la respuesta. El ejemplo más común es un navegador Web, que proporciona acceso a los servicios de información y procesa las respuestas de acuerdo a las necesidades de la aplicación.

El **servidor de origen** utiliza un **conector de servidor** para regir el espacio de nombres de un recurso solicitado. Es la fuente definitiva de las representaciones de los recursos y debe ser el destinatario final de cualquier solicitud que tenga la intención de modificar el valor de éstos. Cada servidor de origen proporciona una interfaz genérica a sus servicios estableciendo una jerarquía de recursos. Los detalles de implementación se ocultan detrás de la interfaz.

Los componentes intermediarios actúan como un cliente y un servidor con el fin de remitir, con posibles modificaciones, solicitudes y respuestas.

2.3 Perspectivas futuras del estilo REST

A partir del la introducción del concepto de arquitectura REST en el año 2000 ha ido popularizándose como estilo de diseño llegando a convertirse en el **modelo predominante** de arquitectura de los servicios web.

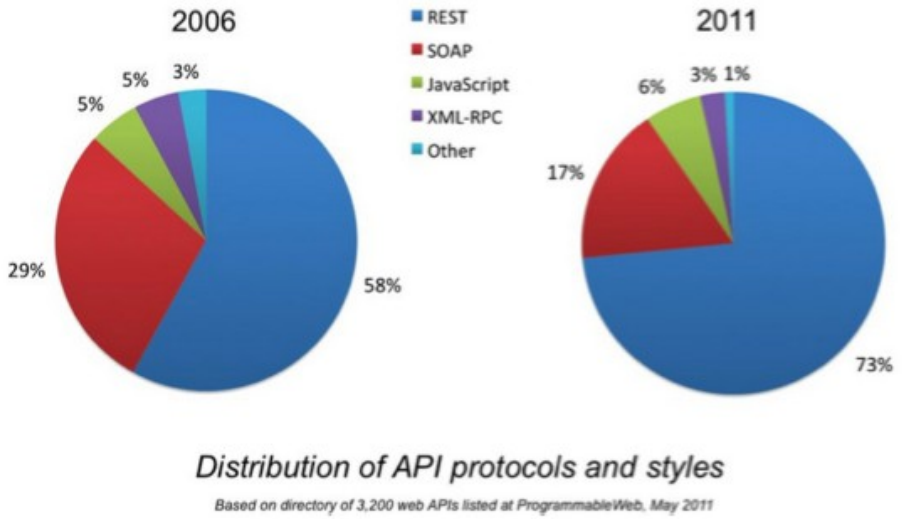

Ilustración 4: Gráfica del creciente uso de la filosofía REST

Como podemos apreciar en la anterior ilustración, la creciente importancia de esta arquitectura de software gracias a su simplicidad y escalabilidad entre otros de sus beneficios, está predominando sobre los demás protocolos.

Capítulo 3 : Apache CouchDB

En el proyecto emplearemos el **gestor de bases de datos** Apache CouchDB como servidor. En este capítulo explicaremos los aspectos más relevantes de este gestor así como su puesta en marcha y funcionamiento.

CouchDB es un gestor de bases de datos **no relacional** que trabaja con documentos JSON. Incluye una API para acceder desde el navegador, a través de HTTP. Se pueden distribuir los datos de manera eficiente mediante la replicación de las bases de datos. También soporta configuraciones maestro-maestro con detección automática de conflictos.

A continuación podemos ver la vista general de la API para navegador de couchDB a la cual se accede a través de la dirección **127.0.0.1:5984/_utils/index.html** .

Desde esta vista podemos administrar las distintas bases de datos que hayamos creado así como crear bases de datos nuevas o editar y borrar las existentes.

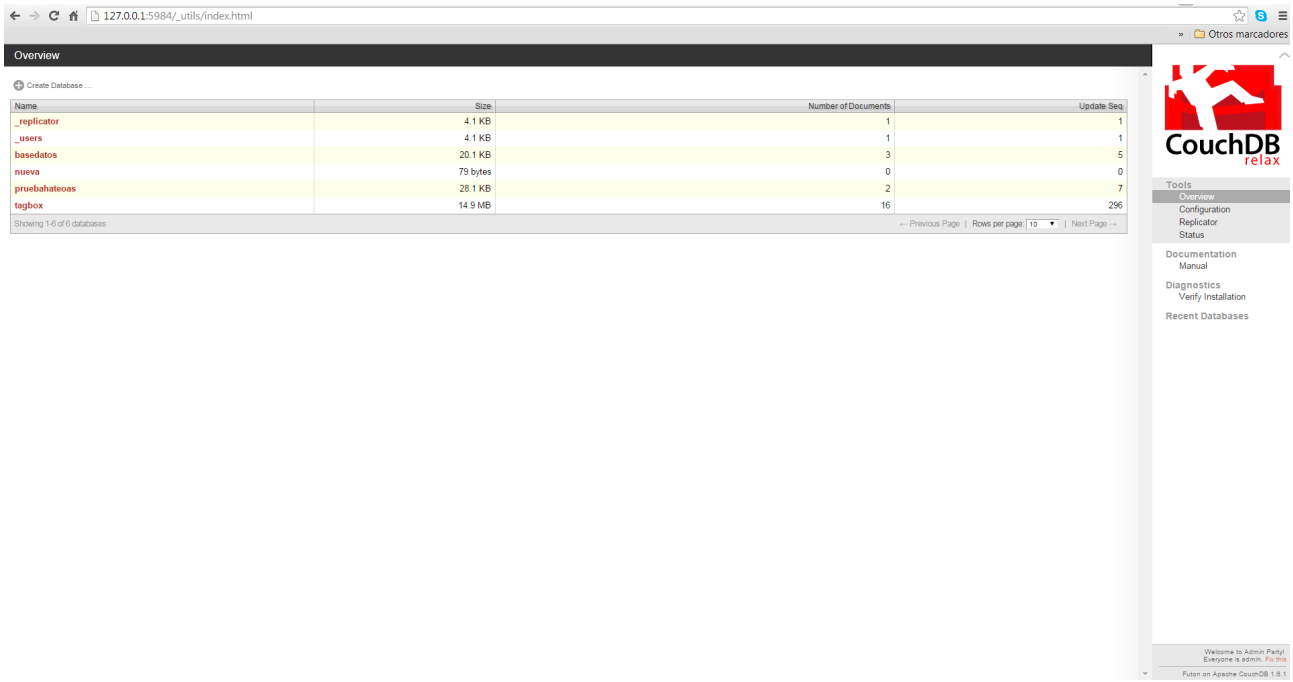

Ilustración 5: Vista principal de la API de CouchDB para el navegador

Al seleccionar alguna de las bases de datos en la vista anterior podemos acceder a la siguiente vista en la que podemos observar los documentos almacenados así como editarlos y anexar archivos (attachments).

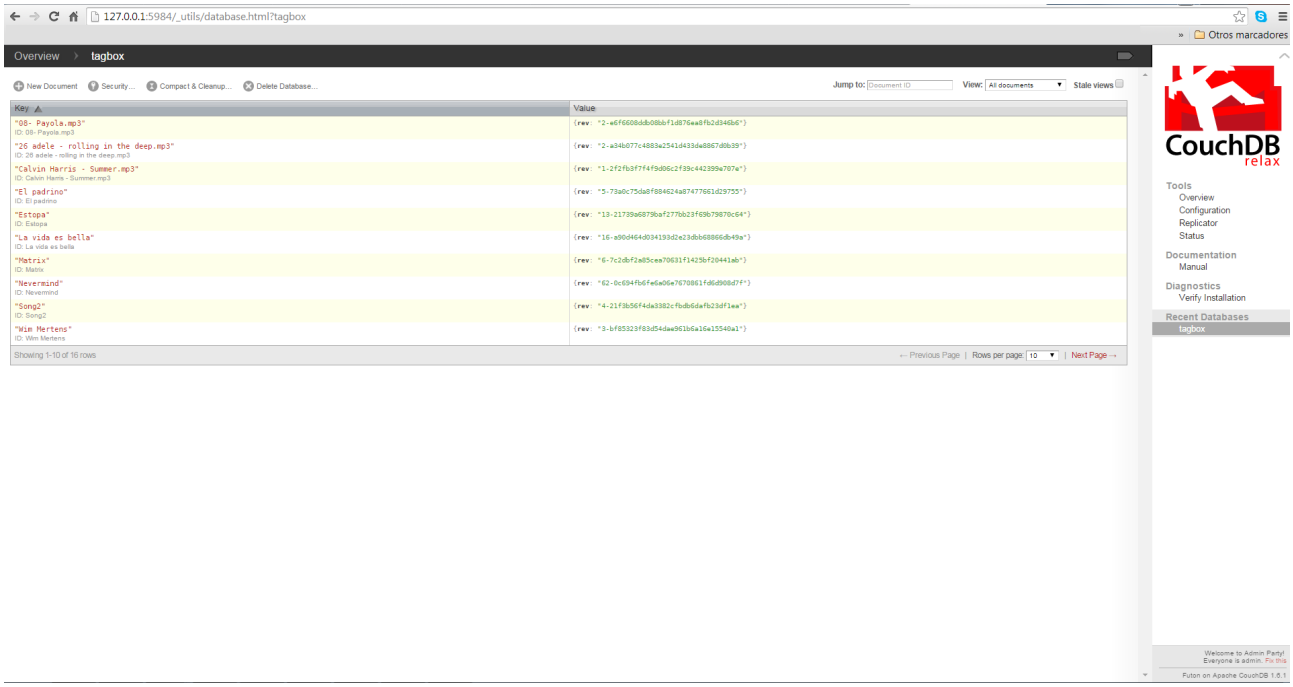

Ilustración 6: Vista de los documentos que componen la base de datos

3.1 Almacenamiento de Documentos

Un servidor CouchDB hospeda bases de datos que almacenan documentos. Cada documento tiene un nombre exclusivo en la base de datos. **CouchDB proporciona una API HTTP RESTful** para la lectura y la actualización (añadir, editar, eliminar) de documentos en la base de datos.

Los documentos son la unidad principal de datos en CouchDB y constan de cualquier número de campos y archivos adjuntos. Los documentos también incluyen metadatos que son administrados por el sistema de base de datos. Los campos del documento se nombran de forma única y contienen valores de tipos diferentes (caracteres, números, booleanos, listas, etc.), y no hay un límite establecido para el tamaño del texto o el recuento de elemento.

El modelo de actualización de documentos en CouchDB es sin bloqueo y optimista. Las aplicaciones cliente pueden editar los documentos. Si otro cliente trata de editar el mismo documento guarda los cambios en primer lugar y obtiene un error de conflicto de edición al intentar guardar. Para resolver el conflicto de actualización, se puede abrir la última versión del documento, editarlo y reintentar la edición.

Las actualizaciones de documentos (añadir, editar, borrar) no se producen de forma parcial, o tiene éxito completo o no se produce y se obtiene un error. De este modo la base de datos no contiene documentos parcialmente editados o corruptos.

3.2 Propiedades ACID

La estructura de los archivos del sistema CouchDB ofrece las propiedades Atomic Isolated Consistent Durable (ACID). En el disco, CouchDB nunca sobrescribe los datos confirmados o estructuras asociadas, asegurando que el archivo de la base de datos sea **siempre coherente**.

Cualquier número de clientes puede leer documentos sin ser bloqueado o interrumpido por actualizaciones concurrentes, incluso en el mismo documento. Las operaciones de lectura en CouchDB utilizan un modelo de Control de Concurrencia Multi-Versión (MVCC), donde cada cliente ve una imagen coherente de la base de datos desde el principio hasta el final de la operación de lectura.

3.4 Compresión

El espacio perdido se recupera mediante una compresión ocasional. Según lo programado, o cuando el archivo de base de datos exceda una cierta cantidad de espacio desperdiciado, el proceso de compresión clona todos los datos activos en un nuevo archivo y luego descarta el archivo antiguo. La base de datos permanece completamente en línea todo el tiempo y todas las actualizaciones y lecturas están permitidas y pueden completarse con éxito. El archivo antiguo de la base de datos sólo se elimina cuando todos los datos se han copiado y todos los usuarios hayan hecho la transición al nuevo archivo.

3.5 Vistas

A diferencia de las bases de datos SQL donde los datos deben ser descompuestos cuidadosamente en tablas, los datos en CouchDB se almacenan en documentos semi-estructurados. Los documentos CouchDB son flexibles y cada uno tiene su propia estructura implícita, lo que alivia los problemas más difíciles de replicación bi-direccionalmente y sus datos contenidos.

CouchDB integra un modelo de vista. Las vistas son el método de agregación y presentación de informes sobre los documentos en una base de datos. Como las vistas se construyen de forma dinámica y no afectan el documento subyacente, puede tener tantos diferentes ver representaciones de los mismos datos que te gusta.

Vistas CouchDB se definen dentro de los documentos de diseño especiales y pueden replicar en todas las instancias de base de datos como documentos regulares, por lo que no sólo los datos se replica en CouchDB, pero los diseños completos de aplicación replicar también.

3.6 Documentos JSON

El nombre JSON proviene del acrónimo JavaScript Object Notation y es un **formato para el intercambio de datos**. Su uso se ha generalizado mucho gracias a su sencilla estructura. JSON se usa normalmente en entornos donde la cantidad de flujo de datos entre cliente-servidor es importante.

La estructura de los documentos JSON está formada por dúos **Clave : Valor** separados por comas y envueltos en llaves. La mejor manera de comprenderlo es observar el ejemplo que viene a continuación, basado en un documento de la base de datos.


```
{
     "_id": "El padrino",
     "_rev": "10-c720292d5b5d91f19818df8636b7ea59",
     "tags": [
          "pelicula",
          "1972",
          "Al Pacino",
          "Marlon Brando",
          "Coppola"
     ],
     "_attachments": {
          "el-padrino.jpg": {
                    "content_type": "image/jpeg",
                    "revpos": 5,
                    "digest": "md5-eRxB2JbMVJR2GupO4cQ1lA==",
                    "length": 32339,
                    "stub": true
          }
     }
}
```


Cada valor va precedido de una clave, la cual no da información acerca del valor.

Capítulo 4 : Desarrollo del proyecto

A continuación se explica el desarrollo del proyecto. Éste consiste en la implementación de un **cliente** que adopta las restricciones necesarias para seguir una filosofía **REST**. La finalidad del cliente es formar un **servicio de almacenamiento** basado en metadatos, que son utilizados para realizar las búsquedas de ficheros. La aplicación se puede utilizar como herramienta para ayuda al aprendizaje de los alumnos cuya tarea es enviar mensajes HTTP de forma similar a como se hace en la aplicación.

Empezaremos explicando lo que son los metadatos. Cuando utilizamos una base de datos para almacenar información, además de guardar lo más importante que es esta información o datos, necesitamos completarla con más información acerca de éstos. Un ejemplo puede ser el almacenar temperaturas de una determinada zona o ciudad en una época determinada del año. Si solo almacenamos los datos más importantes, que en este caso son las temperaturas, puede que lleguen a ser inservibles si en el futuro adquirimos más datos de otra zona o época del año pues no pueden o no se deben mezclar. Es obvio que necesitamos más datos, datos que hablen de los anteriores y aporten más información al sistema y volviéndolos más útiles. Los **metadatos** no **son** otra cosa que **datos que contienen información acerca de otros datos** para así complementarlos.

Como hemos explicado en el capítulo 3 acerca de CouchDB, en nuestro sistema cliente-servidor tratamos unos documentos con extensión JSON. Como hemos visto se componen por estructuras donde prima la dualidad Clave : Valor . Esto es un gran ejemplo para distinguir el empleo de metadatos. Como podemos comprobar el Valor es el dato importante de la arquitectura, pero necesita otro dato que le dé aún más valor, valga la redundancia. La Clave es simplemente un metadato que describe al dato Valor al cual precede. Si tenemos una estructura JSON para almacenar fichas sobre los empleados de una empresa es obvio que necesitaremos metadatos para denominar los datos concernientes al: nombre, primer apellido, segundo apellido, fecha de nacimiento, código postal, etc.

El proyecto puede utilizarse como herramienta de ayuda en el aprendizaje de futuros alumnos de la universidad en el estudio de asignaturas relacionadas con las Tecnologías de la Información y las Comunicaciones. Se ha desarrollado una herramienta con la que poder comprender un poco mejor el funcionamiento de la filosofía REST así como sus restricciones de implementación y sobre todo el uso de protocolos de capas inferiores como es el caso de **HTTP**.

La interfaz gráfica ha sido desarrollada utilizando los componentes de la clase swing de java (javax.swing) para lo que hemos tenido que importar sus librerías. Además hemos utilizado librerías para el manejo de ficheros como es el caso de java.io.File y también hemos necesitado la librería org.json para utilizar unas funciones que formateaban los mensajes JSON y nos permitían extraer e incluir componentes clave-valor como por ejemplo los Tags.

CouchDB está formada por bases de datos que contienen documentos. A su vez, estos documentos pueden contener archivos anexos.

La base de datos contendrá documentos de extensión JSON. La estructura de estos documentos está compuesta de cuatro campos principales.

- "_id" : Este campo hace referencia a la identificación única del documento dentro de la base de datos
- "_rev" : En este campo estará el código de revisión del documento, necesario para poder editar o eliminar el mismo.
- "tags" : Este campo será una estructura compuesta por cadenas de caracteres separadas por comas y a su vez envueltas entre corchetes.
- "_attachments" : Este campo también será una estructura compuesta por los diferentes archivos anexionados al documento en forma de subestructuras. Que serán los que se puedan descargar desde la aplicación cliente.

Al crear el documento, el campo tags se encontrará vacío hasta que el usuario lo rellene desde la aplicación. En este campo se añadirán las palabras más importantes que puedan sugerir estos documentos. Estas palabras también pueden ser categorías, de tal manera que si tenemos un conjunto de documentos con formato de audio sería apropiado etiquetarlos a todos con un tag llamado música o audio, **facilitando la búsqueda** de éstos por medio de los tags.

Siguiendo la filosofía REST y teniendo en cuenta una de sus más importantes restricciones, utilizamos una interfaz uniforme. Únicamente utilizaremos cuatro tipos de mensajes HTTP (get, put, post y delete) y las posibles operaciones con la base de datos serán las siguientes:

URI	GET	PUT	POST	DELETE
tagbox.cai:5984/tagbox	Devuelve información sobre la base de datos	Devuelve información sobre la base de datos	Añade un documento a la base de datos	Borra la base de datos y todos sus contenidos.
tagbox.cai:5984/tagbox/{doc}	Devuelve el documento.	Crea o actualiza el documento.	No permitido	Borra el documento
tagbox.cai:5984/tagbox/{doc}/{fich}	Devuelve el archivo anexo.	Crea o actualiza el archivo anexo.	No permitido	Borra el anexo
tagbox.cai:5984/tagbox/tags/{tag}	Devuelve las URIs de los anexos de los documentos etiquetados con ese tag.	No permitido	No permitido	No permitido

Para la implementación del cliente REST hemos empleado el **lenguaje JAVA** y **CouchDB**, como ya hemos explicado anteriormente hará el papel de servidor. El protocolo usado para la transmisión de datos es el conocido protocolo de transferencia de hyper-texto (HTTP). El cliente REST deberá configurar los mensajes HTTP para mandar peticiones básicas a la base de datos CouchDB. Estas peticiones básicas serán por ejemplo, **cargar un documento, eliminarlo, descargarlo, editar los tags que tenga y hacer una búsqueda por tags** para que el servidor nos devuelva todas las URIs de los archivos anexos a los documentos que incluyen dicho tag.

Tras haberme documentado acerca de las librerías disponibles en JAVA para poder llevar a cabo conexiones HTTP exitosas comprendí que el primer paso para afrontar la construcción de la interfaz de cliente era diseñarla a groso modo.

La implementación en JAVA del cliente la he desarrollado bajo el Entorno de Desarrollo Integrado (IDE) **NetBeans** por ser uno de los más populares y recomendados además de tener algo de experiencia previa con el mismo.

El primer paso como decía ha sido diseñar la GUI en base a las necesidades de la aplicación. Sabemos que necesitamos una **interfaz fácil e intuitiva** para potenciar el interés de los alumnos en la filosofía REST y por eso se intenta diseñar un entorno amigable sin un excesivo número de botones en el que la facilidad de las transacciones es primordial.

Procedemos a diseñar en NetBeans el entorno gráfico que deseamos.

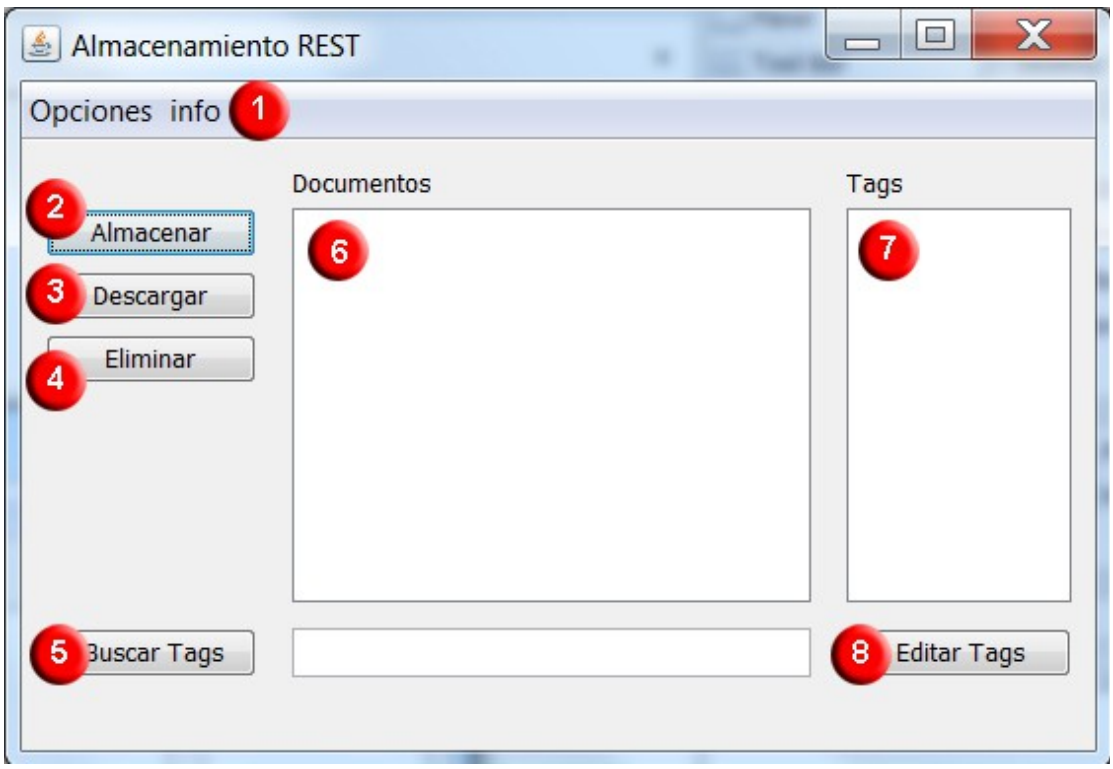

Ilustración 7: Vista principal de la GUI

El diseño de la interfaz no se logró en el primer intento, sino que se fueron haciendo modelos que sufrieron transformaciones a medida que de la experiencia y los errores fueron surgiendo otras necesidades. La interfaz consta de varios componentes los cuales serán comentados uno a uno seguidamente. Como vemos, a cada componente se le ha puesto una etiqueta para reconocerlos más facilmente.

1. El primer componente es la barra de estado. Dentro de las opciones podemos activar la ventana que muestra los mensajes HTTP. También tenemos la información de la aplicación.
2. El segundo componente es el botón Almacenar. Se utiliza para almacenar un documento de nuestro disco duro en la base de datos. Este botón llamará a la función crearDoc() después de elegir el archivo a almacenar.
3. Este botón sirve para descargar un archivo. Desde aquí se invoca a la función descargar(). Solo si previamente se ha seleccionado un archivo en la lista de documentos (6).
4. El botón eliminar sirve para eliminar de la base de datos el documento seleccionado. Desde aquí se invoca a la función borrarDoc().
5. El botón Buscar Tags sirve para buscar todos los anexos de los documentos que hayan sido previamente etiquetados con el tag que se introduce en campo de texto que hay al lado. Desde la función que trata este evento se llama a la función buscarTag().
6. En este componente (jList1) se listan todos los documentos que se incluyen en la base de datos por medio de la función listDir().
7. En esta lista (jList2) se imprimen los tags que han sido etiquetados en el documento seleccionado en el la lista de documentos (6). La función listTags() se encarga de ello cuando salta el evento de que un componente de la lista ha sido clickado.
8. Este último componente es el botón cuya finalidad es editar los tags del documento seleccionado. Dependiendo de si un tag ha sido seleccionado o no, su función será la de editarlo/eliminarlo o crear un tag nuevo.

Una vez diseñado se implementaron las **funciones principales** que se necesitarían al activar estos botones de la interfaz. A continuación enumero las funciones que he necesitado desarrollar para poder establecer comunicación con la base de datos y transmitirle todas las peticiones necesarias.

- `Private void listDir() :`

  Esta es la función más utilizada por el cliente. Envía una petición HTTP al servidor para que devuelva el **conjunto de documentos existentes en la base de datos** y posteriormente los lista en el componente jList1 de la interfaz, que es la lista que hay debajo de la etiqueta Documentos. Esta función se ejecuta desde el mismo instante en que el cliente empieza a correr para mostrar los archivos que hay en la base de datos y también después de modificar los archivos con las operaciones "Almacenar" y "Eliminar" con el fin de actualizar el contenido de la BBDD.

- `Private String crearDoc() :`

  Esta función se utiliza al presionar el botón de **almacenar documento**. Previamente abrimos una ventana JfileChooser de la cual obtenemos el nombre y la ruta del archivo que queremos almacenar en la base de datos, posteriormente se lleva a cabo la carga del archivo así como el campo llamado TAGS en estado vacío. Como vemos devuelve una cadena (String) con el resultado de la carga, que nos devuelve "Documento cargado con éxito" si el código de respuesta del servidor es 201, "El documento ya existe" si el código de respuesta es 409 o "Error al cargar el documento" en cualquier otro caso.

- `Private void listTags(String name) :`

  Esta función nos devuelve el **listado completo de los tags** que han sido etiquetados para el archivo indicado en el argumento `name`. La dificultad de esta función fue trabajar con los documentos JSON que nos devuelve el servidor y poder formatearlos. Para solucionar este problema tuve que incorporar las librerías org.json.* documentarme acerca de las funciones que incluye para el formateo de estos archivos.

- `Private String revDoc(String name) :`

  Esta función sirve para **obtener el código de revisión del documento** indicado por el argumento name. Este código es indispensable a la hora de editar archivos de la base de datos o eliminarlos.

- `Private void borrarDoc(String name, String rev) :`

  Esta función se encarga de mandar una petición HTTP al servidor para que **elimine un documento de la base de datos**. Para esto necesitamos el nombre del documento así como el código de revisión del mismo, que van indicados como argumentos a la hora de invocar la función.

- `private void addTag(String oldTag, String tag,String name) :`

  Esta función tiene la misión de **añadir un nuevo Tag** al documento o editar uno ya existente. Para distinguir entre estas dos funcionalidades comprueba si el primer argumento "oldTag" es una cadena vacía, en cuyo caso añadiremos un tag nuevo con el valor que contenga el argumento "tag". Si el contenido del argumento "oldTag" no estuviese vacío, el cometido de la función sería eliminar este tag previamente antes de incluir el nuevo. La percepción del usuario sería el cambio de un tag por otro.

- `Private void buscarTag(String tag) :`

  El objetivo de esta función es mandar una petición a la base de datos consultando un tag para que ésta nos **devuelva una lista con todas las URI´s** de los archivos adjuntos pertenecientes a documentos etiquetados con este tag. Para ello deberemos pasar el tag como argumento. Este tag se recoge del componente jTextFiedl2 que está situado justo al lado del botón "Buscar Tag" que es justamente el que invoca la función.

- `Private void descargar(String name,String saveDir) :`

  Como indica el nombre de esta función, su funcionalidad es la de **descargar archivos de la base de datos**. Para ello necesitamos pasarle como argumentos el nombre del archivo a descargar y el directorio donde se ubicará.

- `Private String obtenerAtt(String name) :`

  Esta función se encarga de obtener la ruta y el nombre de los archivos adjuntos de un documento para poder descargarlos previamente con la función descargar.

- `Private void mostrarImagen(final URL url) :`

  Mediante esta función se pueden **mostrar las imágenes adjuntas** a los documentos cuando presionamos los enlaces de las URIs que nos proporciona las búsqueda por tags. Simplemente se abre una ventana que muestra la imagen.

Estas funciones que acabamos de ver forman la espina dorsal del cliente REST. Son invocadas desde las funciones que tratan los eventos de pulsación de botones.

La interfaz gráfica se basa en el manejo de eventos, por eso, el método main de la clase lo único que hace es crear y mostrar un objeto de la clase "RESTful".

4.1 Ejemplo práctico de uso del cliente

En primer lugar, para el correcto funcionamiento del cliente debemos estar seguros de que la base de datos está disponible. La base de datos utiliza la URI http://127.0.0.1:5984 a la cual el cliente accede, pero, ¿y si la base de datos es ejecutada en otra máquina? ¿Y si hay que salir a internet para llevar a cabo la comunicación? ¿Dejaría esto obsoleto a nuestro cliente? La respuesta es no, esta dirección ha sido enmascarada por el servicio de nombres. El cliente intentará acceder a la URI **tagbox.cai:5984** y por eso previamente a su utilización deberemos hacer una pequeña configuración en el sistema operativo de tal manera que pueda traducir este nombre por la dirección IP real donde se ubica el servidor.

En nuestra práctica hemos ejecutado el servidor siempre en la misma máquina donde se ejecuta el cliente, por lo tanto, la **configuración** en Windows es la siguiente.

1) Modifique el **archivo hosts** (C:\Windows\System32\drivers\etc, según sistema operativo), añada esta línea:

127.0.0.1 tagbox.cai

Según el S.O. pueden necesitarse permisos de administrador para modificarlo (debe ejecutar el programa bloc de notas como administrador y abrir el archivo)

2) Modifique el archivo de configuración **local.ini** (C:\Program Files (x86)\Apache Software Foundation\CouchDB\etc\couchdb)

En la sección **[VHOSTS]** introduzca la siguiente línea:

tagbox.cai:5984 = /tagbox/_design/tagbox/_rewrite

3) En la web http://127.0.0.1:5984/_utils/ entre en configuración (Tools > Configuration, parte derecha de la pantalla) y ponga a "false" el parámetro "secure_rewrites"

"secure_rewrites": false

Una vez hayamos configurado esto y puesto en marcha la base de datos podremos ejecutar el cliente. En caso de que hubiese algún problema con la conexión entre el cliente y el servidor nos aparecería el siguiente mensaje:

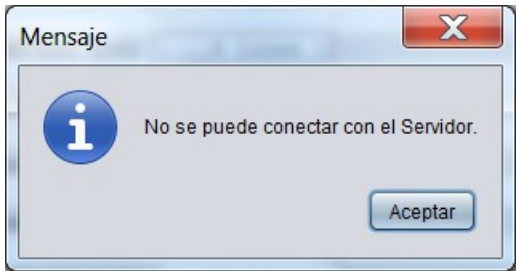

Ilustración 8: Mensaje de error de conexión.

Si por el contrario la conexión es satisfactoria el cliente hará una petición a la base de datos preguntando por los documentos que incluye y los mostrará en la lista correspondiente.

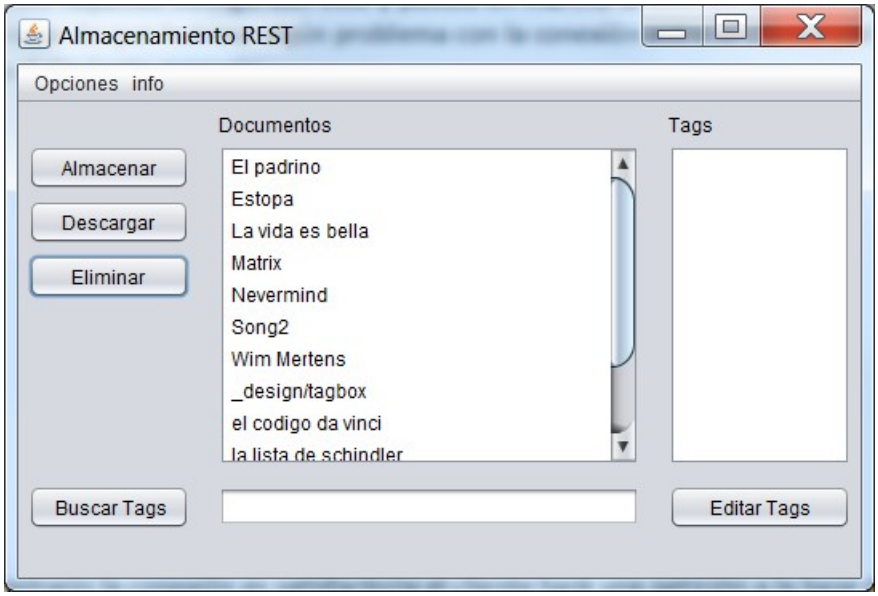

Ilustración 9: Vista principal del cliente en funcionamiento

Como podemos observar, en la ventana central nos aparece una lista con los documentos existentes en nuestra base de datos. La lista de la derecha permanece vacía por el momento porque no hemos seleccionado ningún documento para el cual queramos ver sus tags.

En el momento en el que clickamos sobre uno de los documentos existentes comprobamos que la lista de tags se rellena con los tags que ha sido etiquetado el documento.

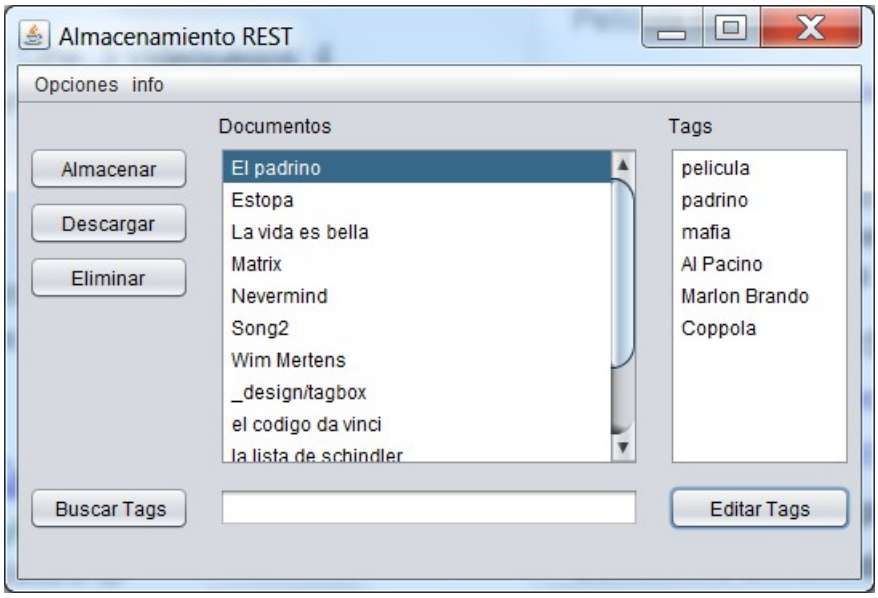

Ilustración 10: Lista de Tags pertenecientes al documento El padrino

A continuación procederemos a añadir un tag y eliminar otro. Para esto, partiendo del estado de ésta última imagen deberemos clickar sobre algún Tag que queramos editar o eliminar, o si lo que queremos es añadir uno nuevo no hace falta que seleccionemos ninguno de la lista. Una vez seleccionado el tag “mafia” pulsamos el botón “Editar Tags” y nos aparece la siguiente ventana.

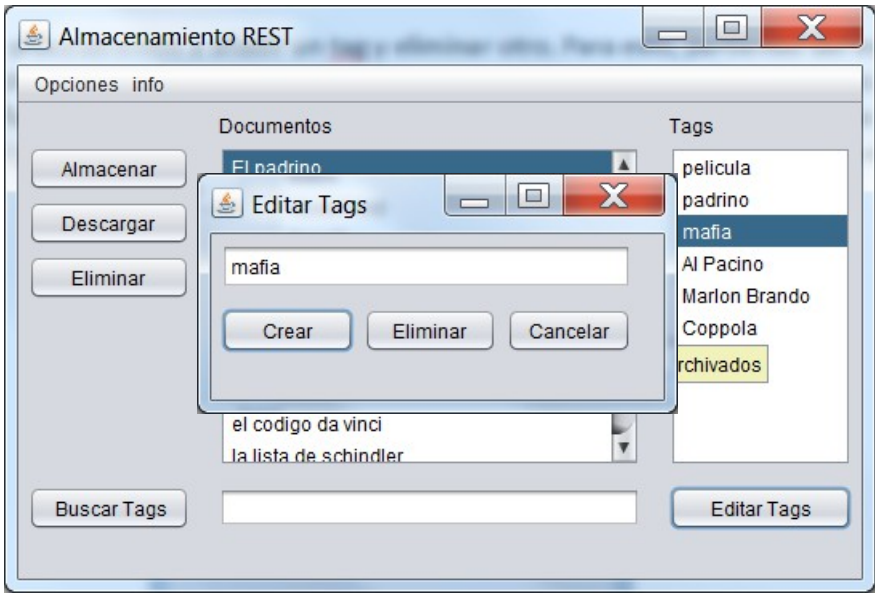

Ilustración 11: Edición de Tags

A continuación cambiamos el nombre “mafia” y lo sustituimos ,por ejemplo, por “1972” que es el año en el que se estrenó la película.

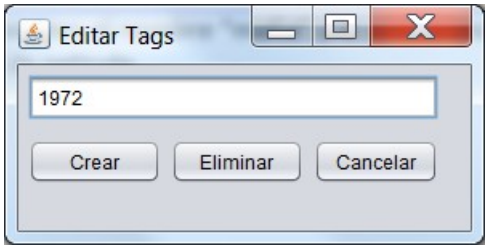

Ilustración 12: Ventana de edición de tags

Ahora le daremos al boton “crear” que al tener seleccionado el Tag “mafia” previamente, tendrá la función de editarlo en lugar de crear uno nuevo. El resultado es el siguiente:

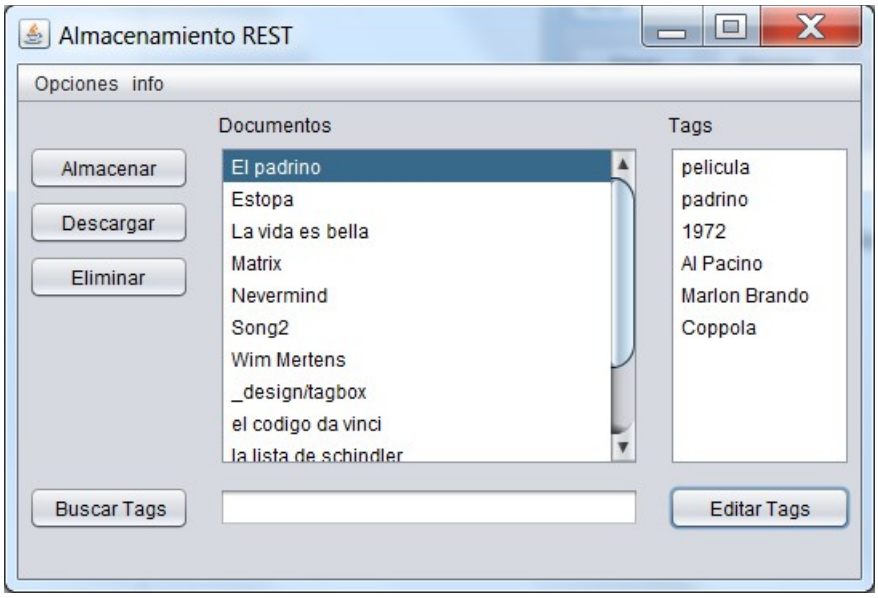

Ilustración 13: El tag "mafia" ha sido editado a "1972"

Como podemos apreciar el contenido del Tag “mafia” ha pasado a ser “1972”. El siguiente paso será eliminar el Tag “padrino” pues nos parece algo redundante. Lo seleccionamos y volvemos a clickar el botón “Editar Tags”:

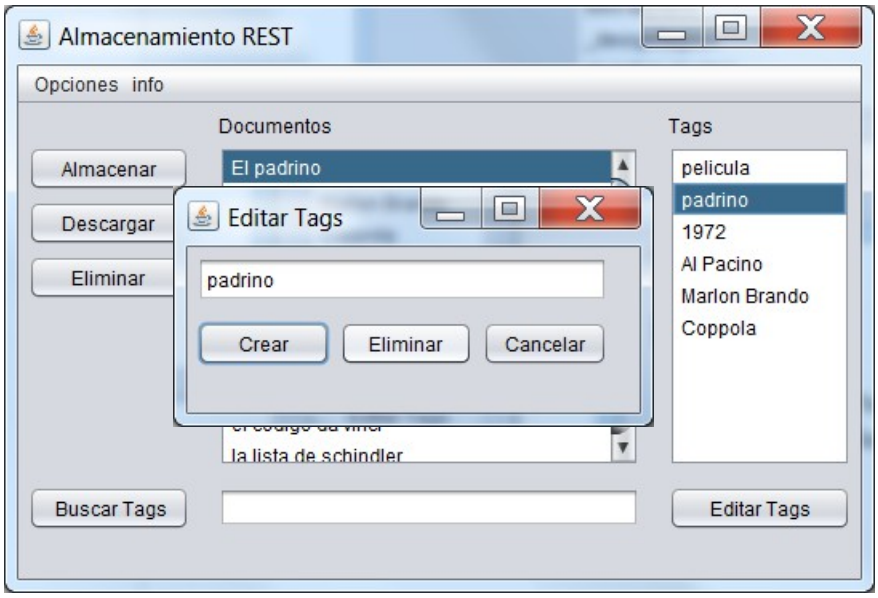

Ilustración 14: Eliminación de Tags

Nos aparece de nuevo la ventana de edición de Tags, en la cual nos aparecería el tag “padrino” previamente seleccionado y lo único que tendríamos que hacer es pulsar el botón “Eliminar”.

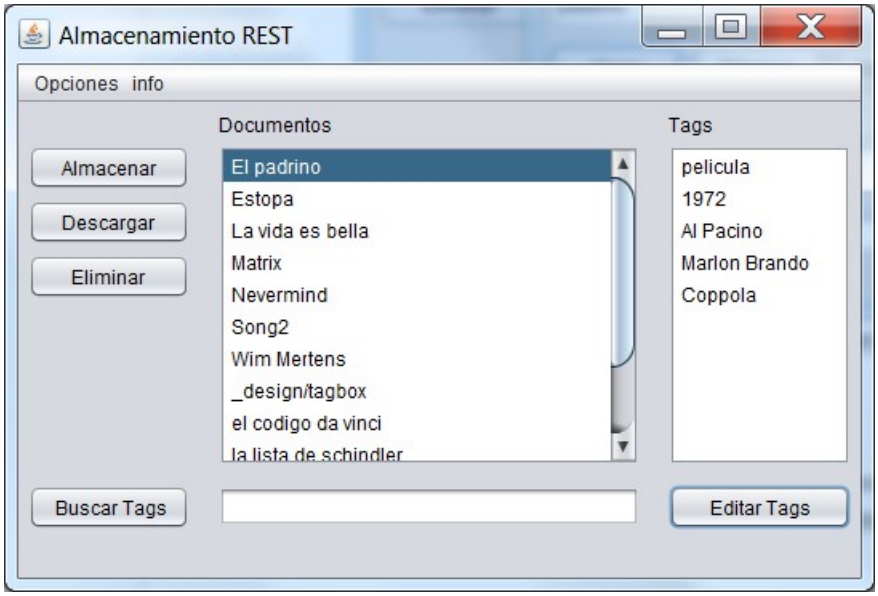

Ilustración 15: El Tag "padrino" ha sido eliminado

Como podemos ver el Tag “padrino” ha sido eliminado exitosamente.

La siguiente tarea será almacenar un documento en la base de datos. Partiendo de la vista principal deberemos pulsar el botón “Almacenar” y se nos abrirá la siguiente ventana:

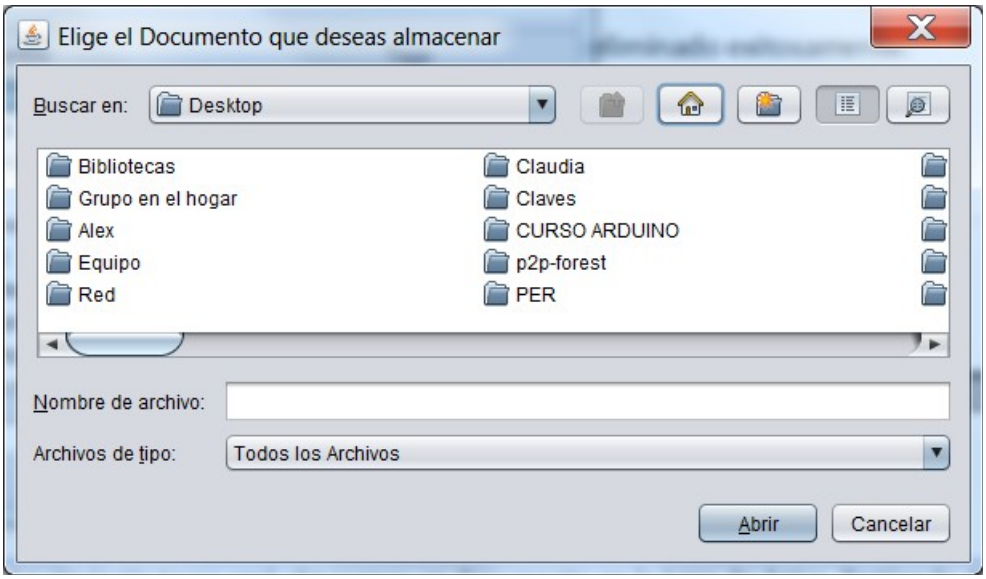

Ilustración 16: Ventana jFileChooser

Esta ventana nos permitirá elegir un documento de entre las distintas ubicaciones dentro de nuestra máquina. Una vez seleccionado seleccionado el documento únicamente tendremos que pulsar el botón “Abrir”

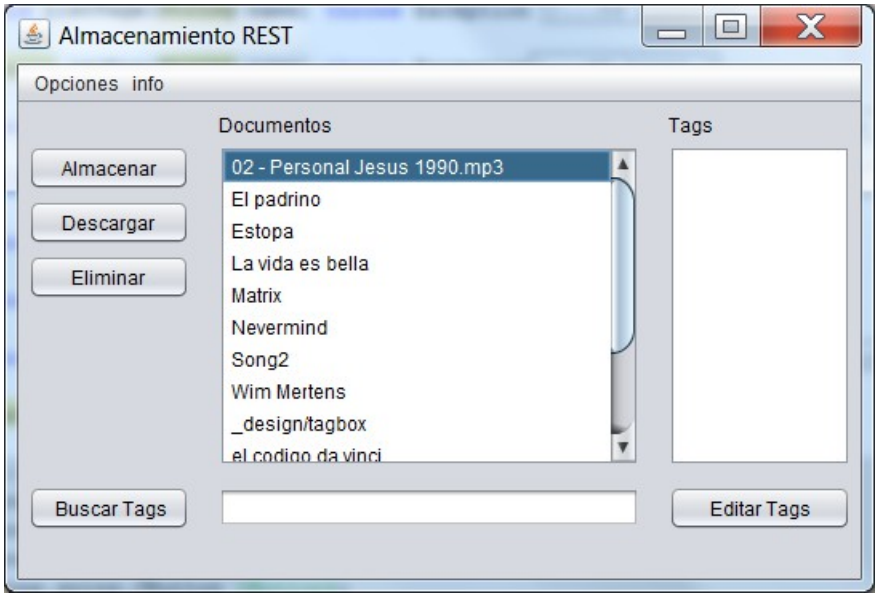

Ilustración 17: Almacenamiento exitoso de un fichero con extensión mp3

Como podemos ver el almacenamiento ha sido exitoso y la lista se ha actualizado apareciendo el documento de extensión mp3 que acabamos de almacenar. Como vemos, el documento está seleccionado pero el campo Tags aparece vacío. Procederé a añadir algunos tags.

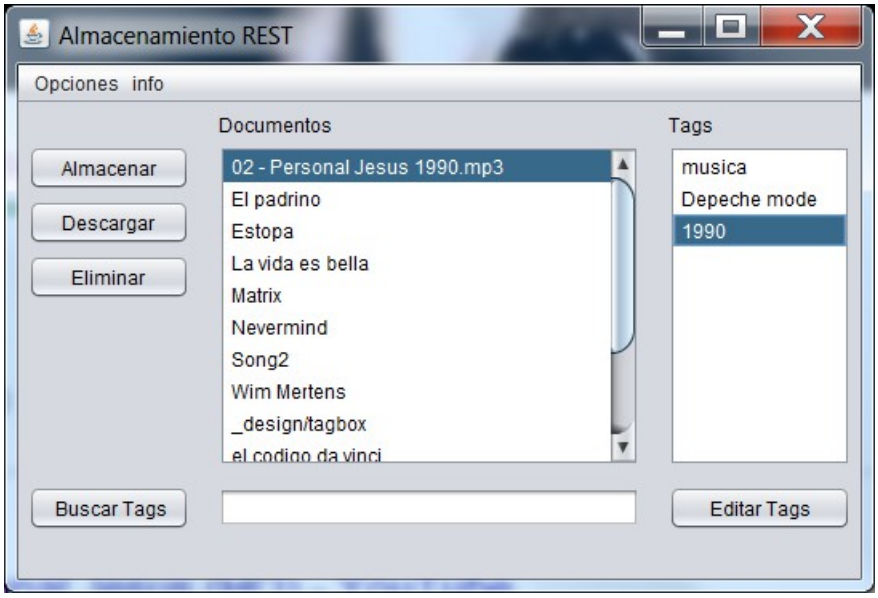

Ilustración 18: Añadimos algunos Tags al nuevo documento

Una vez hecho esto vamos a descargar otro documento. Por ejemplo, la portada de la película de matrix. Seleccionamos el documento y pulsamos el botón descargar y obtenemos el siguiente resultado:

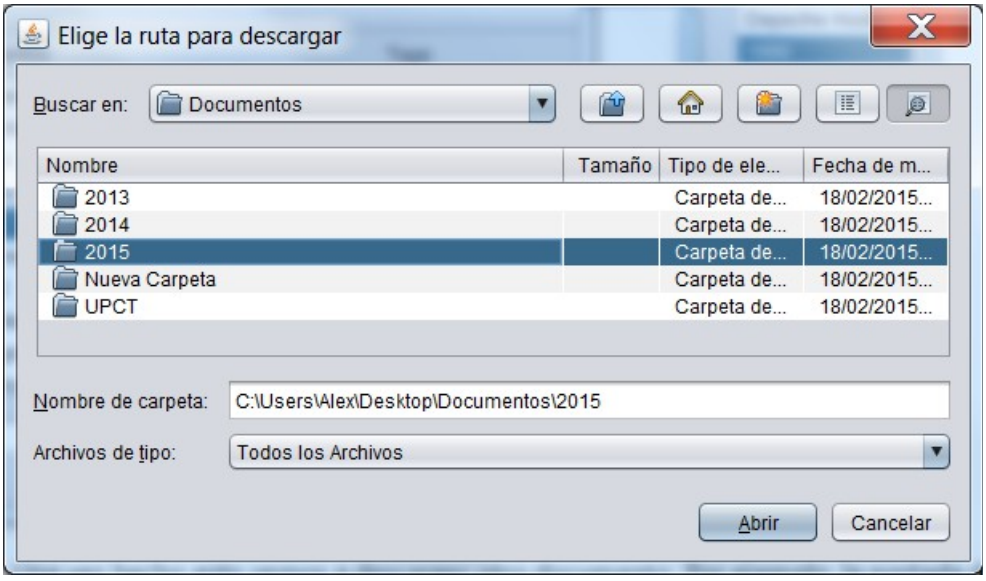

Ilustración 19: Selección de carpeta para descargar el documento

Y procedemos a seleccionar la ubicación donde será descargado el documento.

Nos aparece el siguiente mensaje indicando que todo ha salido perfecto:

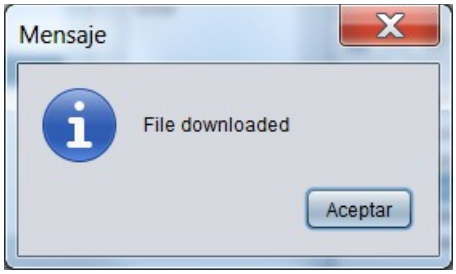

Ilustración 20: Archivo descargado

Comprobamos si en la carpeta de destino se encuentra el documento en buen estado.

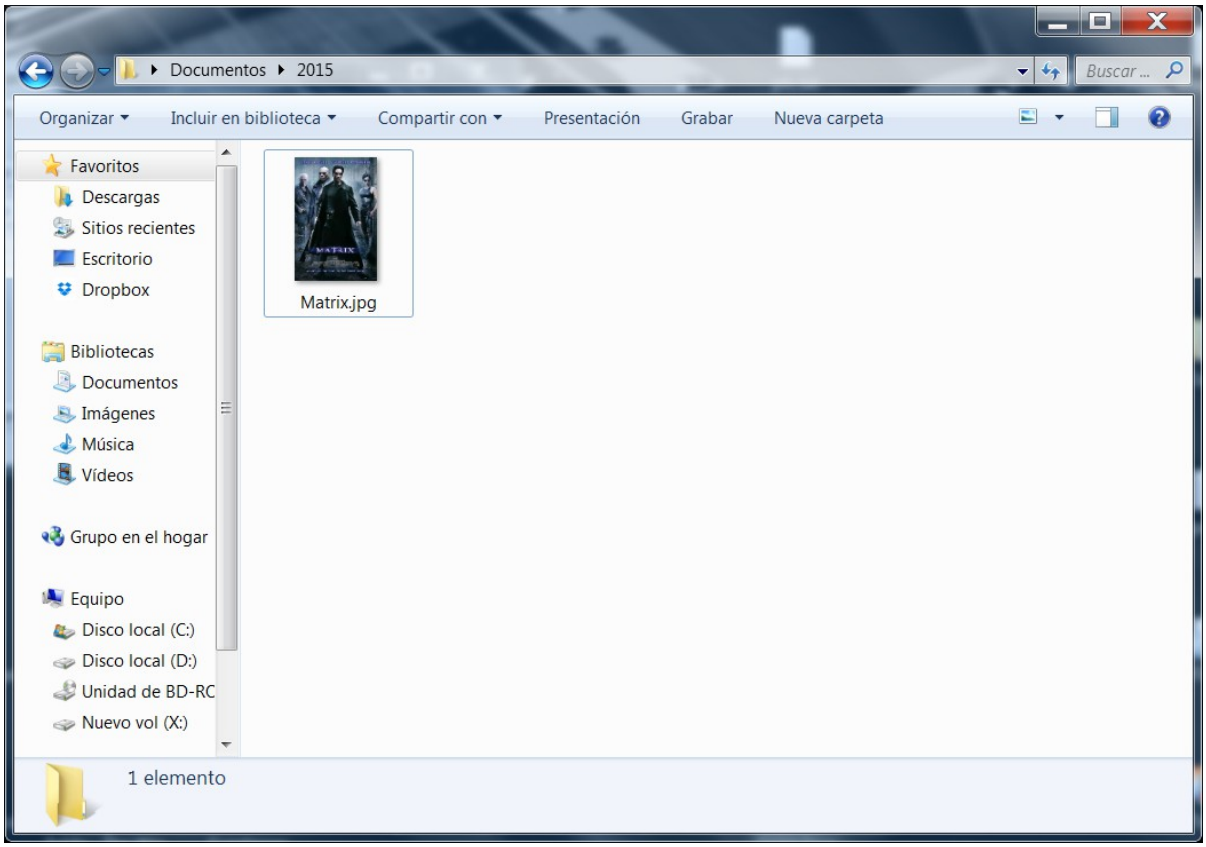

Ilustración 21: Descarga exitosa

La carátula de la película se ha descargado correctamente en la ubicación que se eligió.

Para eliminar un documento simplemente debemos seleccionarlo en la vista principal del cliente y pulsar el botón “Eliminar”.

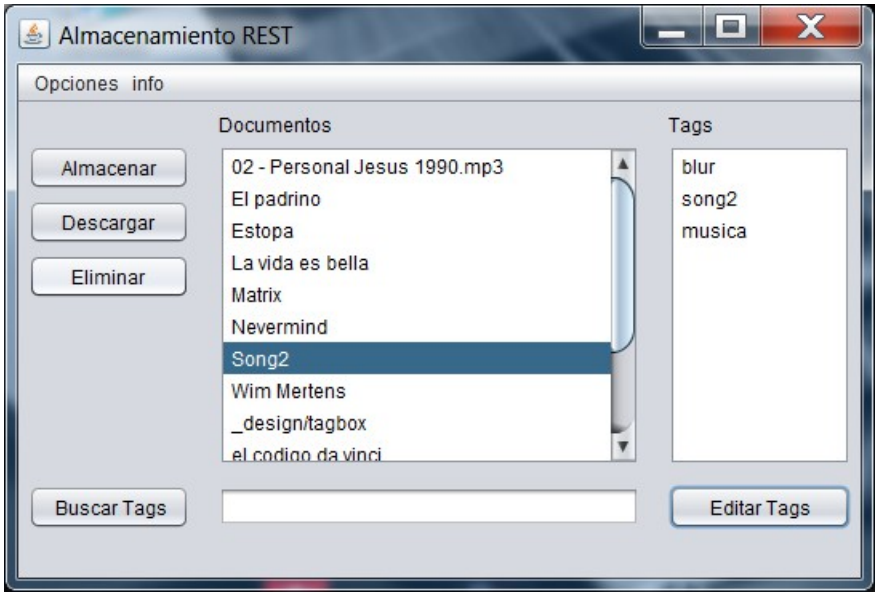

Ilustración 22: Eliminamos el documento Song2

Después de pulsar el botón eliminar se actualiza la lista mostrando la siguiente imagen:

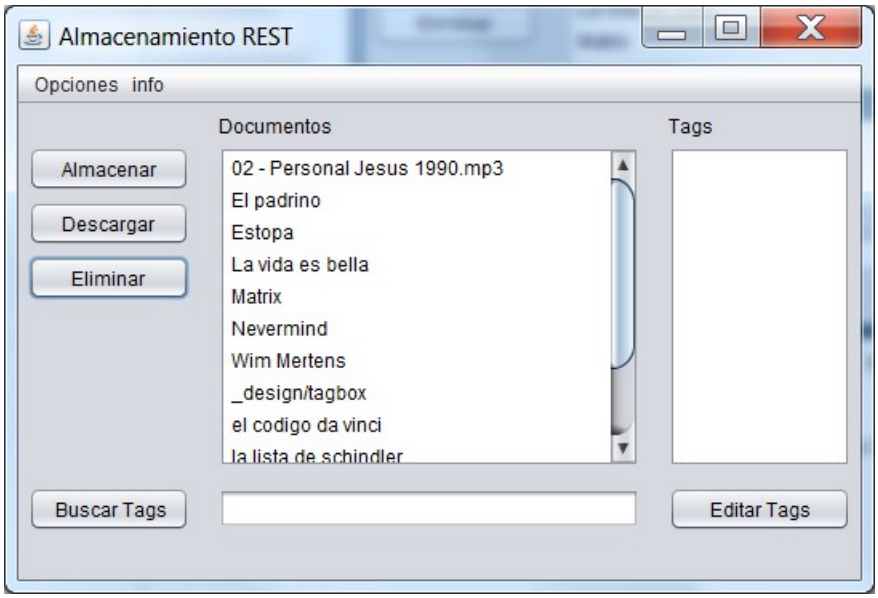

Ilustración 23: Eliminado exitoso

Como podemos ver el documento Song2 ha sido eliminado correctamente de la base de datos.

A continuación vamos a utilizar una de las herramientas más importantes del cliente, la búsqueda de documentos por Tags. En primer lugar debemos escribir un tag en el campo de texto situado a la derecha del botón “Buscar Tags”.

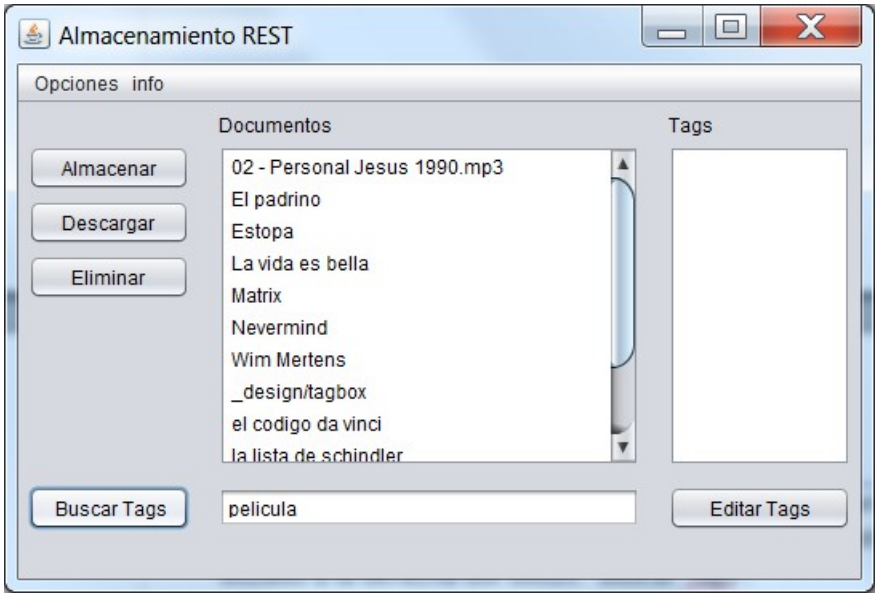

Ilustración 24: Búsqueda del Tag "película"

Como vemos, vamos a buscar todos los documentos que han sido etiquetados por el Tag “pelicula”. Seguidamente nos aparece una ventana con los resultados de la búsqueda:

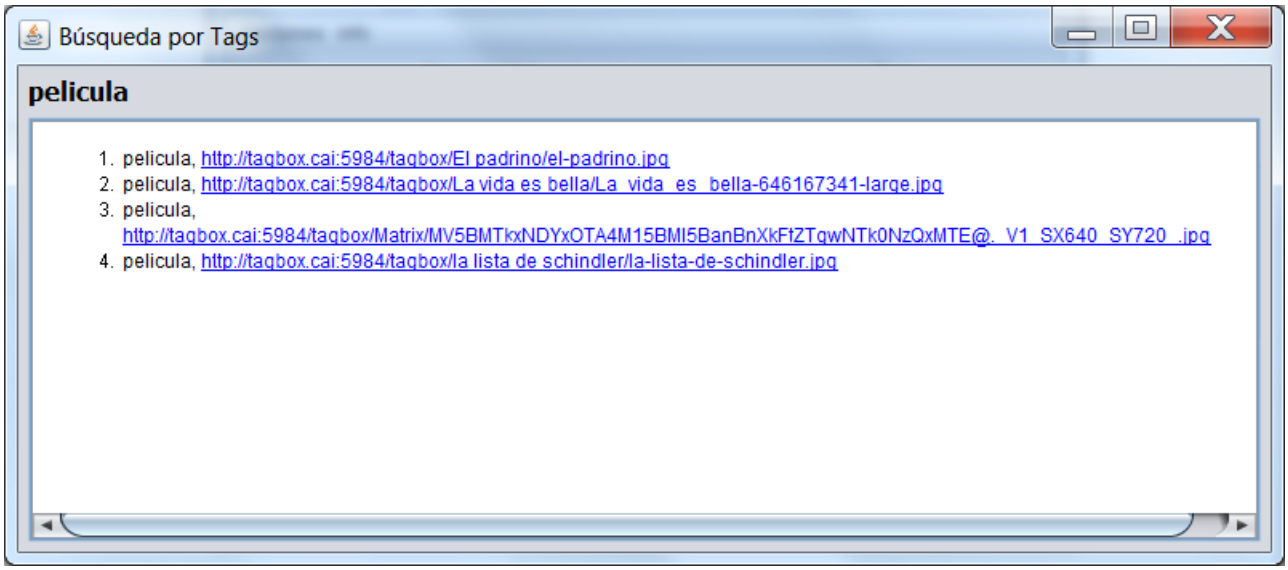

Ilustración 25: Resultados de la búsqueda

Como vemos, el resultado son hipervínculos a las URIs de los archivos adjuntos a los documentos. Si pulsamos sobre alguna de estas URIs se nos abre una ventana para previsualizar el archivo hipermedia.

Ilustración 26: Representación del documento

Como podemos ver ser muestra la imagen que había almacenada en la base de datos.

Por último vamos a mostrar el fin educativo de esta aplicación, si nos fijamos en la pestaña superior de opciones se encuentra un checkbox mediante el cual podemos mostrar una ventana que ha ido recogiendo el log de peticiones HTTP para que los alumnos puedan observar los mensajes que el cliente tiene que mandar para realizar las operaciones pertinentes con la base de datos.

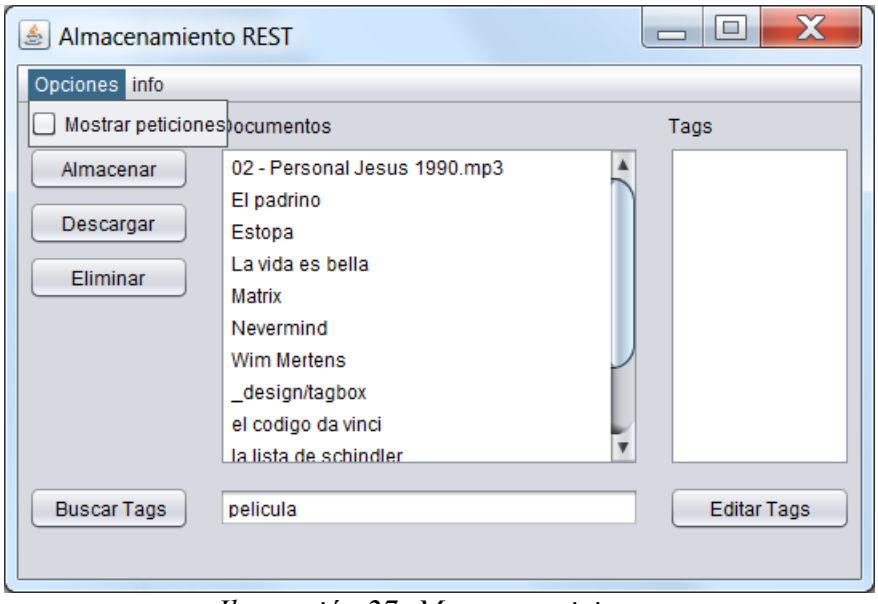

Ilustración 27: Mostrar peticiones

Si marcamos el anterior Checkbox se mostrará la siguiente ventana donde podemos observar todos los mensajes enviados al servidor y el código de respuesta de éste último.

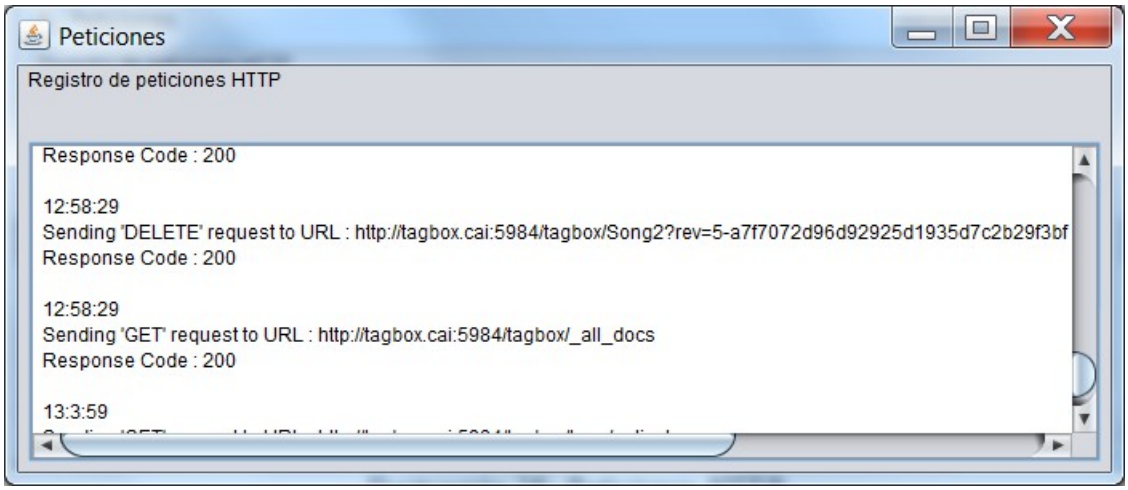

Ilustración 28: Lista de peticiones

En esta ventana podemos ir viendo las distintas peticiones, los códigos de respuesta y la hora cuando se han efectuado. Se puede apreciar que, por ejemplo, cuando hemos eliminado el documento Song2 hemos mandado una petición HTTP con cabecera DELETE y seguidamente se muestra la URL a la cual se ha mandado, sin olvidar el código de revisión después del nombre del recurso, el cual es necesario para operaciones de edición y eliminación de documentos en la base de datos.

Capítulo 5 : Conclusión y líneas futuras

En este Proyecto Final de Carrera hemos desarrollado una aplicación cliente según la filosofía REST. Este cliente forma parte de un servicio web para almacenamiento de documentos basado en metadatos. Estos metadatos forman parte de la estructura de los documentos y nos sirven para poder hacer búsquedas en el sistema formando "categorías" de documentos. Hemos utilizado el gestor de bases de datos CouchDB como servidor del servicio web de almacenamiento.

Para concluir debemos comentar los aspectos más complicados del proyecto como pueden ser la utilización de librerías para establecer la comunicación con el servidor, el formateo de los mensajes en formato JSON para poder editar la estructura donde van incluidos los tags o el código perteneciente a la función "Almacenar". Esto último generaba bastantes problemas ya que el servidor nos devolvía un error al intentar cargar el documento.

Creo que este cliente implementa unas funcionalidades bastante básicas en el manejo de archivos con la base de datos. Básicas no quiere decir sencillas, pues implementarlas ha sido lo más lo más complicado del proyecto. La parte de diseño ha sido mucho más amena y sencilla, es algo más atractivo de realizar aunque no por eso ha sido una tarea fácil. La interfaz gráfica ha sido editada numerosas veces pero sobre todo tuvo tres etapas en las que cambió muy significativamente.

Veo esta implementación como un trampolín para creación de nuevos Servicios Web basados en la filosofía REST. A partir de las funciones que he desarrollado como lisDir(), crearDoc(), etc, se pueden llevar a cabo otros proyectos desde unas bases más desarrolladas. Puede ampliarse el número de formatos de archivo anexo para pre-visualizar en la aplicación, ya que en este proyecto solo se pueden pre-visualizar imágenes. Como hemos visto, en la actualidad la filosofía REST está en auge y cada día se desarrollan más y más aplicaciones siguiendo estas restricciones. Además de desarrollar en nuevas aplicaciones a partir de ésta, los alumnos pueden trabajar añadiendo funcionalidades a la misma para convertir este "primitivo" cliente de almacenamiento en un potente cliente sin nada que envidiar a otros tan populares como DropBox.

Capítulo 6 : Bibliografía

[1] Dissertation: Roy Fielding, 2000.

[2] http://docs.couchdb.org/.

[3] World Wide Web Consortium http://www.w3.org/TR/ws-arch/.

[4] RESTful Java with JAX-RS: Bill Burke, O'Reilly Media , 2009.

[5] http://stackoverflow.com