

¿Qué ofrece Autentia Real Business Solutions S.L?

Somos su empresa de **Soporte a Desarrollo Informático**.
 Ese apoyo que siempre quiso tener...

1. Desarrollo de componentes y proyectos a medida



2. Auditoría de código y recomendaciones de mejora

3. Arranque de proyectos basados en nuevas tecnologías

1. Definición de frameworks corporativos.
2. Transferencia de conocimiento de nuevas arquitecturas.
3. Soporte al arranque de proyectos.
4. Auditoría preventiva periódica de calidad.
5. Revisión previa a la certificación de proyectos.
6. Extensión de capacidad de equipos de calidad.
7. Identificación de problemas en producción.



4. Cursos de formación (impartidos por desarrolladores en activo)

Spring MVC, JSF-PrimeFaces /RichFaces,
 HTML5, CSS3, JavaScript-jQuery

Gestor portales (Liferay)
 Gestor de contenidos (Alfresco)
 Aplicaciones híbridas

Tareas programadas (Quartz)
 Gestor documental (Alfresco)
 Inversión de control (Spring)

Control de autenticación y
 acceso (Spring Security)
 UDDI
 Web Services
 Rest Services
 Social SSO
 SSO (Cas)

JPA-Hibernate, MyBatis
 Motor de búsqueda empresarial (Solr)
 ETL (Talend)

Dirección de Proyectos Informáticos.
 Metodologías ágiles
 Patrones de diseño
 TDD

BPM (jBPM o Bonita)
 Generación de informes (JasperReport)
 ESB (Open ESB)


[Entra er](#)

[Inicio](#)
[Quiénes somos](#)
[Tutoriales](#)
[Formación](#)
[Comparador de salarios](#)
[Nuestro libro](#)
[Charlas](#)

» Estás en: [Inicio](#) [Tutoriales](#) [Spring Integration: Ejemplo completo.](#)



Miguel Arlandy Rodríguez

Consultor tecnológico de desarrollo de proyectos informáticos.

Puedes encontrarme en [Autentia](#): Ofrecemos servicios de soporte a desarrollo, factoría y formación

Somos expertos en Java/JEE



[Ver todos los tutoriales del autor](#)

**Cat
Aut**

Fecha de publicación del tutorial: 2012-02-22
Spring Integration: Ejemplo completo.

Tutorial visitado 4 veces [Descargar en PDF](#)

» Au
las r
Adr

» X\
Myb
Hibe

» Pr
Lag:

» Ct
prep
apa

» ¡¡¡
trafc

Hist

Últi

» Se

» Int
Inte:

» Tr

» Aq

» Aq
Maç

0. Índice de contenidos.

- [1. Introducción.](#)
- [2. Entorno.](#)
- [3. El problema.](#)
- [4. Los Servicios Web.](#)
- [5. Paso 1: Recibiendo las peticiones del cliente.](#)
- [6. Paso 2: Trazando la petición y obteniendo el identificador del usuario por cada entidad bancaria.](#)
- [7. Paso 3: descomponiendo el mensaje.](#)
- [8. Paso 4: Enrutando los mensajes en función de su contenido.](#)
- [9. Paso 5: obteniendo las cuentas del cliente para la entidad 088.](#)
- [10. Paso 6: obteniendo las cuentas del cliente para la entidad 767.](#)
- [11. Paso 7: obteniendo las cuentas del cliente para la entidad 955.](#)
- [12. Paso 8: agregar todos los resultados y devolver la respuesta.](#)
- [13. Probando el ejemplo.](#)
- [14. Referencias.](#)
- [15. Conclusiones.](#)

1. Introducción

Como vimos en el [anterior tutorial](#) Spring Integration es la implementación de una plataforma de integración propuesta por Spring Framework.

En este tutorial vamos a ver un ejemplo de uso más complejo en el que consumiremos diferentes servicios, transformaremos datos y gestionaremos el flujo de los mensajes en función de su contenido.

2. Entorno.

El tutorial está escrito usando el siguiente entorno:

- Hardware: Portátil MacBook Pro 15' (2.2 Ghz Intel Core I7, 8GB DDR3).
- Sistema Operativo: Mac OS Snow Leopard 10.6.7
- Spring Integration 2.0.5.
- Spring 3.0.5.

**Últi
Aut**

» Int

- Apache JMeter 2.4.

3. El problema.

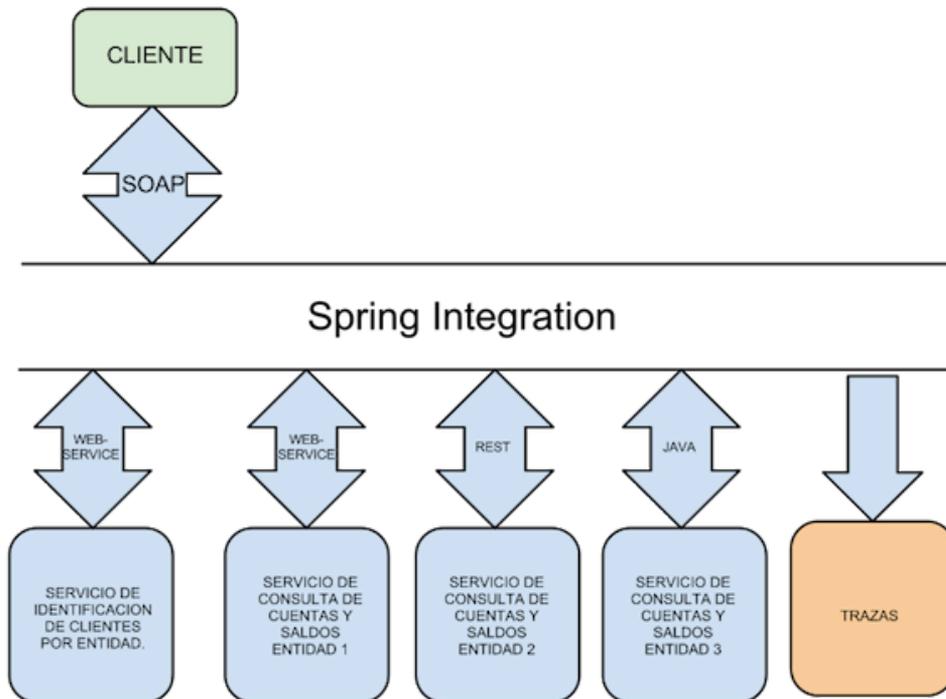
El escenario que proponemos es el siguiente. Trabajamos con tres entidades bancarias, cada una de las cuales tiene un servicio que devuelve las cuentas y saldo de un cliente de dicha entidad. Para que el servicio devuelva las cuentas y saldos debe recibir el identificador del cliente para esa entidad. Además, disponemos de un servicio que, recibiendo un NIF de un usuario devolverá las entidades bancarias y el identificador del usuario de cada una de las entidades con las que tiene cuentas abiertas el usuario.

Nuestra plataforma de integración (Spring Integration) deberá hacer lo siguiente. Recibirá peticiones via Web-Service recibiendo el NIF de un usuario. Con ese NIF consumirá el servicio de entidades por cliente. Con lo que responda ese servicio, irá a cada uno de los servicios de las distintas entidades con las que trabaja el cliente a por las cuentas y saldos del cliente en esa entidad. Devolverá una respuesta con los datos de todas las cuentas y saldos que tiene el cliente en todas las entidades. Además dejaremos un registro en un fichero (log) por cada petición que reciba la plataforma.

Las características de los servicios son las siguientes:

- Servicio de consulta de entidades/identificadores de cliente por entidad: Este es el servicio que devuelve las entidades e indentificadores de cliente por entidad. Recibe un NIF. Es un Servicio WEB.
- Servicio de consulta de cuentas de usuario en la entidad 1 (código de entidad 088): Servicio Web que recibe peticiones con el identificador del cliente para dicha entidad y devuelve sus cuentas con su saldo.
- Servicio de consulta de cuentas de usuario en la entidad 2 (código de entidad 767): Servicio REST que recibe peticiones con el identificador del cliente para dicha entidad y devuelve sus cuentas con su saldo.
- Servicio de consulta de cuentas de usuario en la entidad 3 (código de entidad 955): Clase Java a la que se invoca pasando el identificador del cliente para dicha entidad y devuelve una lista (Collection) con las cuentas y saldo.

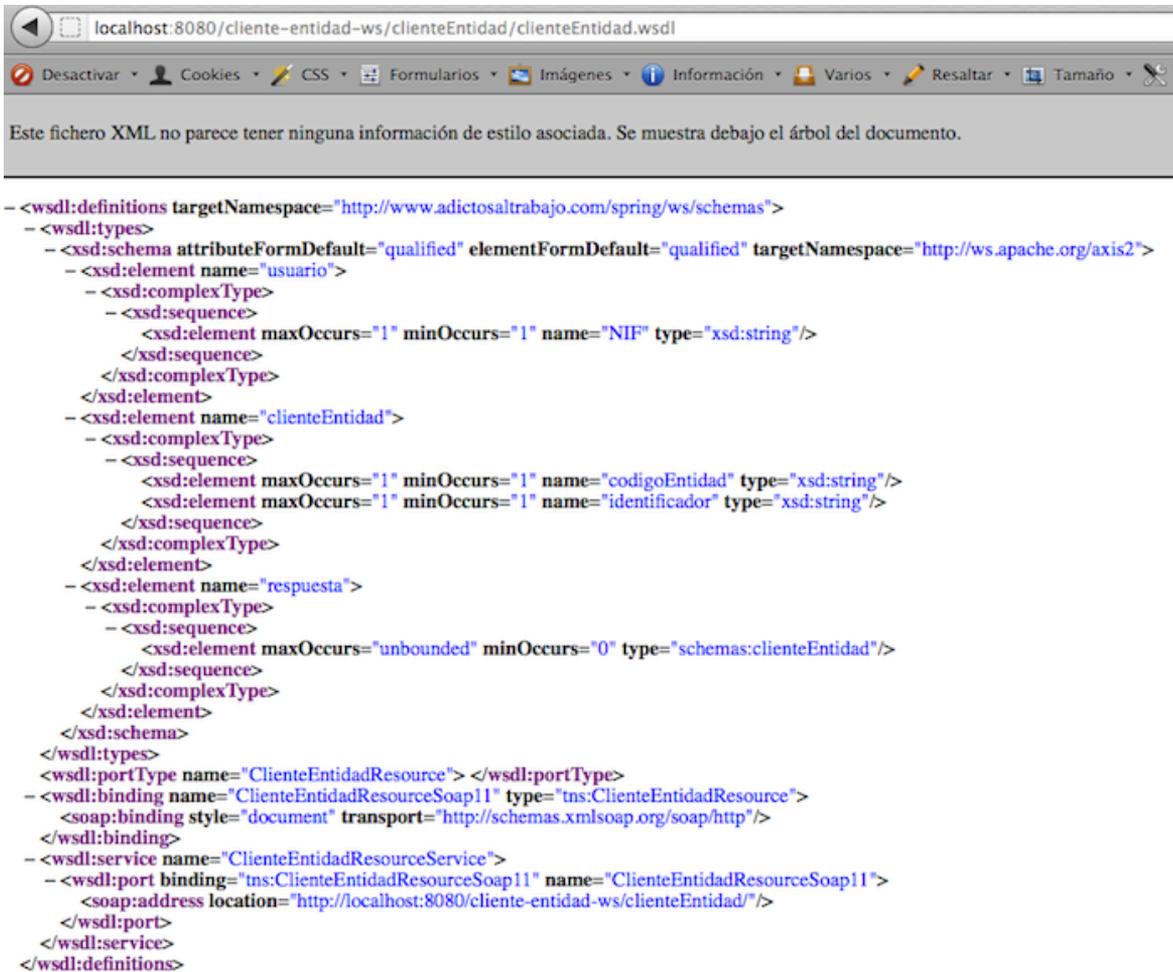
Gráficamente hay que hacer esto:



4. Los Servicios Web.

Como hemos comentado, dos de los servicios son Web-Services.

El servicio que recibe un NIF y devuelve los identificadores del cliente para cada una de las entidades bancarias con las que tiene cuentas debe recibir un elemento "usuario" que contendrá un nodo hijo "NIF". Devolverá una "respuesta" con una lista de "clienteEntidad" tal y como se aprecia en el esquema del siguiente .wsdl:

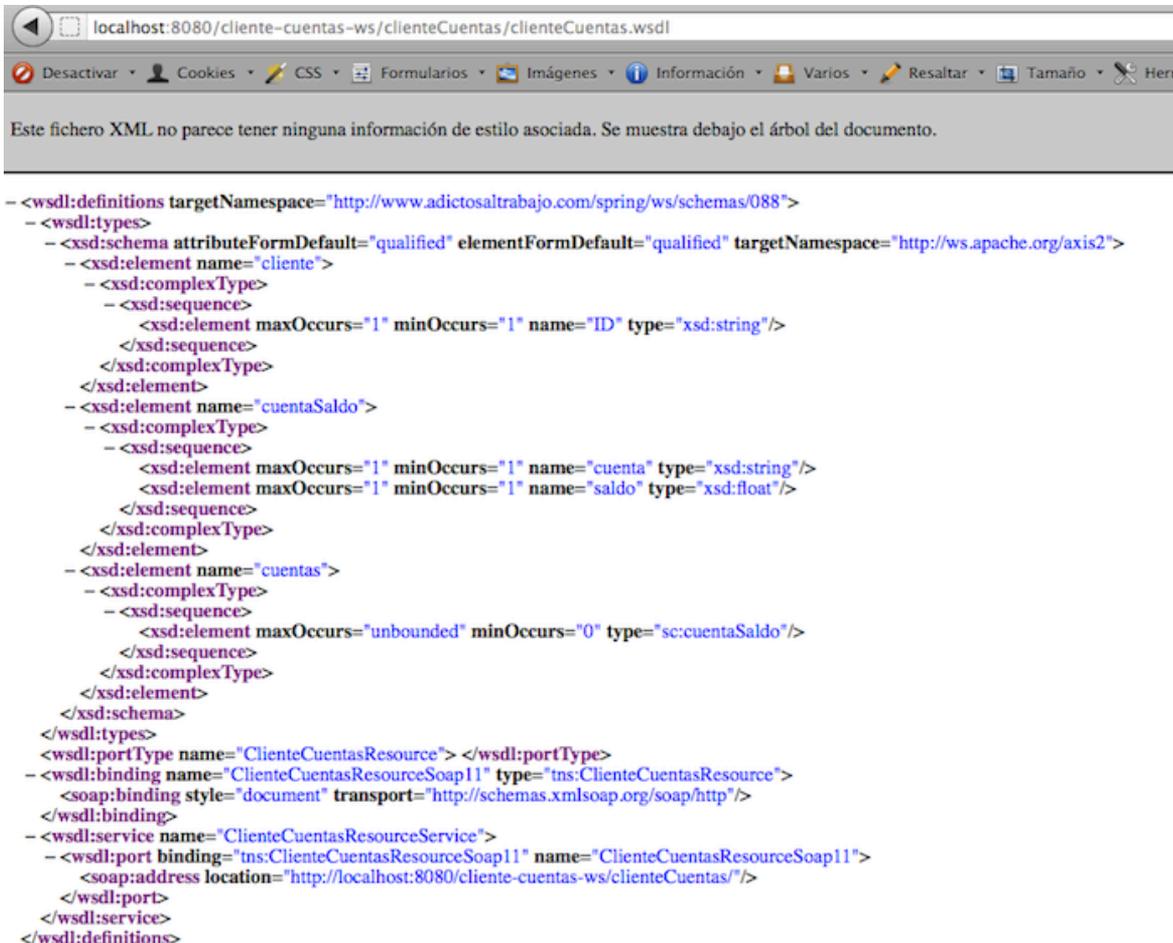


```

- <wsdl:definitions targetNamespace="http://www.adictosaltrabajo.com/spring/ws/schemas">
- <wsdl:types>
- <xsd:schema attributeFormDefault="qualified" elementFormDefault="qualified" targetNamespace="http://ws.apache.org/axis2">
- <xsd:element name="usuario">
- <xsd:complexType>
- <xsd:sequence>
- <xsd:element maxOccurs="1" minOccurs="1" name="NIF" type="xsd:string"/>
- </xsd:sequence>
- </xsd:complexType>
- </xsd:element>
- <xsd:element name="clienteEntidad">
- <xsd:complexType>
- <xsd:sequence>
- <xsd:element maxOccurs="1" minOccurs="1" name="codigoEntidad" type="xsd:string"/>
- <xsd:element maxOccurs="1" minOccurs="1" name="identificador" type="xsd:string"/>
- </xsd:sequence>
- </xsd:complexType>
- </xsd:element>
- <xsd:element name="respuesta">
- <xsd:complexType>
- <xsd:sequence>
- <xsd:element maxOccurs="unbounded" minOccurs="0" type="schemas:clienteEntidad"/>
- </xsd:sequence>
- </xsd:complexType>
- </xsd:element>
- </xsd:schema>
</wsdl:types>
<wsdl:portType name="ClienteEntidadResource"> </wsdl:portType>
- <wsdl:binding name="ClienteEntidadResourceSoap11" type="tns:ClienteEntidadResource">
- <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
</wsdl:binding>
- <wsdl:service name="ClienteEntidadResourceService">
- <wsdl:port binding="tns:ClienteEntidadResourceSoap11" name="ClienteEntidadResourceSoap11">
- <soap:address location="http://localhost:8080/cliente-entidad-ws/clienteEntidad"/>
</wsdl:port>
</wsdl:service>
</wsdl:definitions>

```

El servicio que recibe el identificador del usuario de la entidad 088 y devuelve la lista de cuentas con saldos debe recibir un elemento "cliente" que contendrá un nodo hijo "ID". Devolverá un elemento "cuentas" con una lista de "cuentaSaldo" tal y como se aprecia en el esquema del siguiente .wsdl:



```

- <wsdl:definitions targetNamespace="http://www.adictosaltrabajo.com/spring/ws/schemas/088">
- <wsdl:types>
- <xsd:schema attributeFormDefault="qualified" elementFormDefault="qualified" targetNamespace="http://ws.apache.org/axis2">
- <xsd:element name="cliente">
- <xsd:complexType>
- <xsd:sequence>
- <xsd:element maxOccurs="1" minOccurs="1" name="ID" type="xsd:string"/>
- </xsd:sequence>
- </xsd:complexType>
- </xsd:element>
- <xsd:element name="cuentaSaldo">
- <xsd:complexType>
- <xsd:sequence>
- <xsd:element maxOccurs="1" minOccurs="1" name="cuenta" type="xsd:string"/>
- <xsd:element maxOccurs="1" minOccurs="1" name="saldo" type="xsd:float"/>
- </xsd:sequence>
- </xsd:complexType>
- </xsd:element>
- <xsd:element name="cuentas">
- <xsd:complexType>
- <xsd:sequence>
- <xsd:element maxOccurs="unbounded" minOccurs="0" type="sc:cuentaSaldo"/>
- </xsd:sequence>
- </xsd:complexType>
- </xsd:element>
- </xsd:schema>
</wsdl:types>
<wsdl:portType name="ClienteCuentasResource"> </wsdl:portType>
- <wsdl:binding name="ClienteCuentasResourceSoap11" type="tns:ClienteCuentasResource">
- <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
</wsdl:binding>
- <wsdl:service name="ClienteCuentasResourceService">
- <wsdl:port binding="tns:ClienteCuentasResourceSoap11" name="ClienteCuentasResourceSoap11">
- <soap:address location="http://localhost:8080/cliente-cuentas-ws/clienteCuentas"/>
</wsdl:port>
</wsdl:service>
</wsdl:definitions>

```

5. Paso 1: Recibiendo las peticiones del cliente.

Una vez presentado el escenario, lo primero que haremos será configurar nuestra plataforma de integración para que sea capaz de recibir peticiones del cliente. El cliente nos enviará el NIF del usuario del que queremos obtener sus cuentas y saldos. Añadimos lo siguiente en el fichero de configuración de Spring.

```

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:ws="http://www.springframe
4      xmlns:int="http://www.springframework.org/schema/integration"
5      xmlns:context="http://www.springframework.org/schema/context"
6      xmlns:http="http://www.springframework.org/schema/integration/http"
7      xsi:schemaLocation="http://www.springframework.org/schema/beans
8      http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
9      http://www.springframework.org/schema/context
10     http://www.springframework.org/schema/context/spring-context-3.0.xsd
11     http://www.springframework.org/schema/integration
12     http://www.springframework.org/schema/integration/spring-integration-2.0.xsd
13     http://www.springframework.org/schema/integration/ws
14     http://www.springframework.org/schema/integration/ws/spring-integration-ws-2.0.xsd
15     http://www.springframework.org/schema/integration/http
16     http://www.springframework.org/schema/integration/http/spring-integration-http-2.0.xsd">
17
18     <context:component-scan base-package="com.autentia.spring.integration" />
19
20     <context:property-placeholder location="classpath:springIntegration.properties" />
21
22     <int:channel id="inputChannel" />
23
24     <ws:inbound-gateway id="wsInboundGateway"
25         request-channel="inputChannel" />
26     <int:service-activator input-channel="inputChannel"
27         ref="usuarioEndpoint" output-channel="idsUsuarioChannel" />
28
29 </bean>

```

Lo que estamos haciendo es definir un gateway de entrada vía Web-Service "wsInboundGateway" que hará pasar las peticiones por el canal de entrada "inputChannel" el cual tiene un Service-Activator "usuarioEndpoint" que procesará dichas peticiones y las enviará al siguiente canal "idsUsuarioChannel". El Service-Activator sería el siguiente:

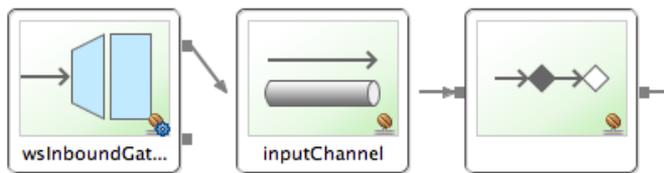
```

1  import javax.xml.transform.dom.DOMSource;
2
3  import org.apache.commons.logging.Log;
4  import org.apache.commons.logging.LogFactory;
5  import org.springframework.integration.annotation.ServiceActivator;
6  import org.springframework.stereotype.Component;
7  import org.w3c.dom.Node;
8  import org.w3c.dom.NodeList;
9
10 @Component
11 public class UsuarioEndpoint {
12
13     private static final Log LOG = LogFactory.getLog(UsuarioEndpoint.class);
14
15     @ServiceActivator
16     public Usuario handleRequest(DOMSource source) throws Exception {
17         final NodeList nodeList = source.getNode().getChildNodes();
18
19         LOG.debug("Receiving request " + nodeList + " length " + nodeList.getLength());
20
21         String nif = "";
22         if (nodeList.getLength() > 1) {
23             for (int i = 0; i < nodeList.getLength(); i++) {
24                 final Node node = nodeList.item(i);
25                 LOG.debug("Nodename " + node.getNodeName());
26                 if (node.getNodeName().equals("NIF")) {
27                     nif = node.getTextContent();
28                 }
29             }
30         }
31
32         return getUsuario(nif);
33     }
34
35     private Usuario getUsuario(String nif) {
36         final Usuario usuario = new Usuario();
37         usuario.setNIF(nif);
38         return usuario;
39     }
40
41 }

```

El Service-Activator recibe en su método handleRequest un DOMSource (XML de la petición) y lo transforma a un objeto Usuario. Dicho objeto será la entrada del siguiente canal "idsUsuarioChannel". Es muy parecido al [ejemplo que vimos en el anterior tutorial](#). Nótese que esta transformación podría haberse hecho usando otro componente distinto a un Service-

Activador, como podría ser un Transformador.



6. Paso 2: Trazando la petición y obteniendo el identificador del usuario por cada entidad bancaria.

Bien, una vez recibidos los datos de entrada, el siguiente paso será trazar la petición en un fichero de log y conectar con el servicio que nos dará el identificador de cliente por cada una de las entidades bancarias con las que tenga cuentas abiertas el usuario. Añadimos lo siguiente en nuestro fichero de configuración de Spring:

```

1 <int:channel id="idsUsuarioChannel" />
2 <!-- Trazamos las peticiones en el fichero (ver log4j.xml) -->
3 <int:logging-channel-adapter id="loggingChannel" level="INFO" expression="'Received requ
4
5 <ws:outbound-gateway id="idsUsuarioGateway"
6   request-channel="idsUsuarioChannel"
7   uri="http://${ws.usuarioentidad.host}:${ws.usuarioentidad.port}/${ws.usuarioentidad.
8   marshaller="usuarioClienteEntidadMarshaller" reply-channel="clienteEntidadSplitterCh
9
10 <bean id="usuarioClienteEntidadMarshaller"
11   class="com.autentia.spring.integration.prueba.spring.integration.UsuarioToClienteEnt

```

Recordemos que en el paso anterior vimos que el Service-Activator enviaba los datos al canal idsUsuarioChannel. Pues bien, a este canal le añadimos un logging-channel-adapter "loggingChannel" que añadirá a un fichero de logs cada mensaje (payload) de entrada que pase por este canal.

Por otro lado, tenemos un gateway que conectará con el Servicio Web de consulta de identificadores por entidad y envía los datos a un canal llamado "clienteEntidadSplitterChannel". La transformación de los datos la delega en UsuarioToClienteEntidadMarshaller.

```

1 import java.io.IOException;
2 import java.util.ArrayList;
3 import java.util.List;
4
5 import javax.xml.transform.Result;
6 import javax.xml.transform.Source;
7 import javax.xml.transform.Transformer;
8 import javax.xml.transform.dom.DOMSource;
9
10 import org.apache.commons.logging.Log;
11 import org.apache.commons.logging.LogFactory;
12 import org.springframework.xml.transform.Marshaller;
13 import org.springframework.xml.transform.Unmarshaller;
14 import org.springframework.xml.transform.MappingException;
15 import org.springframework.xml.transform.StringSource;
16 import org.w3c.dom.Node;
17 import org.w3c.dom.NodeList;
18
19 import com.sun.org.apache.xalan.internal.xsltc.trax.TransformerFactoryImpl;
20
21 public class UsuarioToClienteEntidadMarshaller implements Marshaller, Unmarshaller {
22
23     private static final Log LOG = LogFactory.getLog(UsuarioToClienteEntidadMarshaller.class);
24
25     public Object unmarshal(Source source) throws IOException, MappingException {
26
27         final NodeList nodeList = ((DOMSource)source).getNode().getChildNodes();
28
29         LOG.debug("Receiving request " + nodeList + " length " + nodeList.getLength());
30
31         final List<UsuarioEntidad> usuarioEntidades = new ArrayList<UsuarioEntidad>();
32         if (nodeList.getLength() > 1) {
33             for (int i = 0; i < nodeList.getLength(); i++) {
34                 final Node node = nodeList.item(i);
35                 LOG.debug("Nodename " + node.getNodeName());
36                 if (node.getLocalName().equals("infoUsuario")) {
37                     usuarioEntidades.add(createUsuarioEntidad(node));
38                 }
39             }
40         }
41         return usuarioEntidades;
42     }
43
44     private UsuarioEntidad createUsuarioEntidad(final Node node) {
45         final NodeList childs = node.getChildNodes();
46         String id = "";
47         String entidad = "";

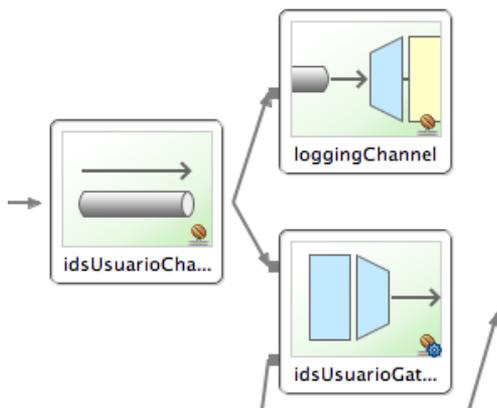
```

```

48     for (int i=0;i<childs.getLength(); i++) {
49         final Node child = childs.item(i);
50         if (child.getLocalName().equals("codigoEntidad")) {
51             entidad = child.getTextContent();
52         } else if (child.getLocalName().equals("identificador")) {
53             id = child.getTextContent();
54         }
55     }
56     return new UsuarioEntidad(id, entidad);
57 }
58
59 public boolean supports(Class<?> clazz) {
60     return false;
61 }
62
63 public void marshal(Object object, Result result) throws IOException, XmlMappingExce
64
65     final Usuario usuario = (Usuario) object;
66
67     final String xmlString = "<schemas:usuario xmlns:schemas=\""http://www.adictosal
68         + "<schemas:nif>" + usuario.getNIF() + "</schemas:nif></schemas:usuario>
69
70     try {
71         final Transformer transformer = new TransformerFactoryImpl().newTransformer(
72             transformer.transform(new StringSource(xmlString), result);
73     } catch (Exception e) {
74         e.printStackTrace();
75     }
76
77 }
78
79

```

El método marshal recibirá los usuarios entran por el canal y los transforma en el XML necesario para enviárselo al WS de identificador de cliente por entidades. El método unmarshal recibe la respuesta del servicio (XML) y la transforma en una lista de UsuarioEntidad (contiene id de usuario e id entidad).



7. Paso 3: descomponiendo el mensaje.

Llegados a este punto ya tenemos una lista con los identificadores de cliente para cada una de las entidades con las que trabaja. Lo que tenemos que hacer ahora con ese mensaje (lista de pares identificador/entidad) es descomponerlo en mensajes identificador/entidad para que sean procesados de manera independiente.

```

1 <int:channel id="clienteEntidadSplitterChannel" />
2 <int:splitter input-channel="clienteEntidadSplitterChannel"
3     output-channel="usuarioEntidadRouterChannel" ref="clienteEntidadSplitter" />

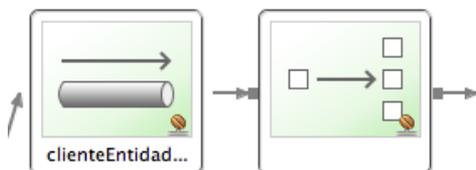
```

La descomposición del mensaje en mensajes más simples será responsabilidad del componente splitter, el cual enviará la salida al canal "usuarioEntidadRouterChannel" que recibirá la lista de mensajes.

```

1 import java.util.Collection;
2 import java.util.List;
3
4 import org.springframework.integration.annotation.Splitter;
5 import org.springframework.stereotype.Component;
6
7 @Component
8 public class ClienteEntidadSplitter {
9
10     @Splitter
11     public Collection<UsuarioEntidad> split (List<UsuarioEntidad> usuarioEntidades) {
12         return usuarioEntidades;
13     }
14
15 }

```



8. Paso 4: Enrutando los mensajes en función de su contenido.

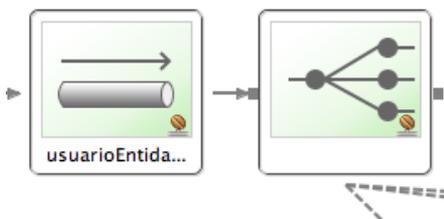
Bueno, pues ya tenemos cada identificador de usuario para cada entidad con la que trabaja en mensajes independientes. Ahora lo que debemos hacer es conectar con el servicio de cada entidad bancaria, pero antes, debemos saber qué mensajes irán por el canal que comunicará con cada entidad. Tendremos 3 canales: uno por el que pasarán los mensajes que vayan a la entidad 1 (088), otro por el que pasarán los que irán a la entidad 2 (767) y otro por el que pasarán los de la entidad 3 (955). Por tanto necesitamos enviar los mensajes, en función de su contenido a su canal correspondiente. Usaremos un enrutador.

```

1 <int:channel id="usuarioEntidadRouterChannel" />
2 <int:recipient-list-router input-channel="usuarioEntidadRouterChannel">
3   <int:recipient channel="cuentasUsuario088"
4     selector-expression="payload.entidad.equals('088')" />
5   <int:recipient channel="cuentasUsuario767"
6     selector-expression="payload.entidad.equals('767')" />
7   <int:recipient channel="cuentasUsuario955"
8     selector-expression="payload.entidad.equals('955')" />
9 </int:recipient-list-router>

```

Creo que se entiende perfectamente, pero lo explico un poco. Este canal contiene un router que, en función del contenido del mensaje (recordemos que en este canal los mensajes entrantes son de tipo UsuarioEntidad), en concreto en función del valor del atributo entidad (método getEntidad) que nos dirá la entidad bancaria del cliente, enviamos el mensaje por un canal u otro.



9. Paso 5: obteniendo las cuentas del cliente para la entidad 088.

Vale, pues vamos a configurar el canal por el que pasarán los mensajes con los clientes de la entidad 088. Recordemos que debemos conectar con un WS para obtener las cuentas y saldos del cliente de dicha entidad. Este caso es muy parecido a cuando conectamos con el WS que devolvía los ids de usuarios en función de su NIF en el paso 2.

```

1 <!-- Conecta con el Servicio WEB que devuelve las cuentas de los clientes de la entidad 088 -->
2 <int:channel id="cuentasUsuario088" />
3 <ws:outbound-gateway id="entidad088Gateway"
4   request-channel="cuentasUsuario088"
5   uri="http://${ws.entidad088.host}:${ws.entidad088.port}/${ws.entidad088.context}"
6   marshaller="entidad088Marshaller" reply-channel="aggregatorResponseChannel" />
7 <bean id="entidad088Marshaller"
8   class="com.autentia.spring.integration.prueba.spring.integration.Entidad088Marshaller" />

```

El marshaller es muy parecido al del caso anterior. En este caso, con la respuesta del servicio devolverá una lista de CuentaSaldo (clase que nos hemos creado para guardar esos datos).



10. Paso 6: obteniendo las cuentas del cliente para la entidad 767.

Recordemos que para obtener las cuentas de un usuario de la entidad bancaria 767 debemos consumir un Servicio REST mediante una petición GET.

```

1 <!-- Conecta con el servicio REST que devuelve las cuentas de los clientes de la entidad 767 -->
2 <int:channel id="cuentasUsuario767" />
3 <http:outbound-gateway request-channel="cuentasUsuario767"
4   reply-channel="aggregatorResponseChannel"
5   url="http://${rest.entidad767.host}:${rest.entidad767.port}/${rest.entidad767.context}/"
6   http-method="GET" expected-response-type="java.lang.String"
7   message-converters="entidad767Converter">

```

```

8 |     <http-uri-variable name="usuario" expression="payload.usuario" />
9 | </http:outbound-gateway>

```

El responsable de convertir los datos de respuesta en una lista de CuentaSaldo será entidad767Converter:

```

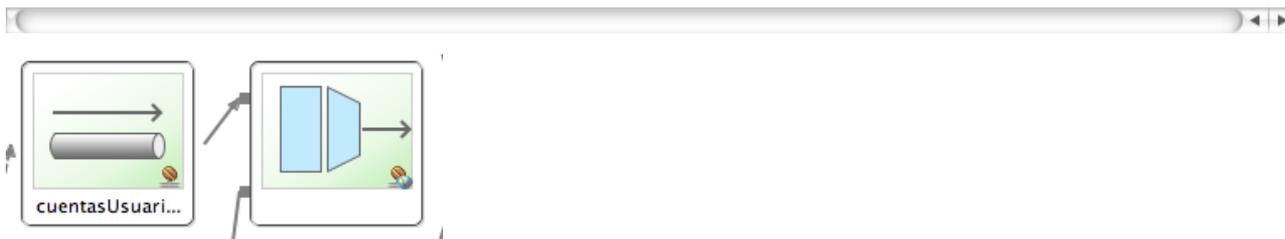
1 | import java.io.BufferedReader;
2 | import java.io.IOException;
3 | import java.io.InputStream;
4 | import java.io.InputStreamReader;
5 | import java.io.Reader;
6 | import java.io.StringWriter;
7 | import java.io.Writer;
8 | import java.util.ArrayList;
9 | import java.util.List;
10 |
11 | import javax.xml.transform.dom.DOMSource;
12 |
13 | import org.apache.commons.logging.Log;
14 | import org.apache.commons.logging.LogFactory;
15 | import org.springframework.http.HttpInputMessage;
16 | import org.springframework.http.HttpOutputMessage;
17 | import org.springframework.http.MediaType;
18 | import org.springframework.http.converter.HttpMessageConverter;
19 | import org.springframework.http.converter.HttpMessageNotReadableException;
20 | import org.springframework.http.converter.HttpMessageNotWritableException;
21 | import org.springframework.integration.xml.source.DomSourceFactory;
22 | import org.springframework.stereotype.Component;
23 | import org.w3c.dom.Node;
24 | import org.w3c.dom.NodeList;
25 |
26 | @Component
27 | public class Entidad767Converter implements HttpMessageConverter<List<CuentaSaldo>> {
28 |
29 |     private static final Log LOG = LogFactory.getLog(Entidad767Converter.class);
30 |
31 |     private static final List<MediaType> SUPPORTED_MEDIA_TYPES = new ArrayList<MediaType>()
32 |     static {
33 |         SUPPORTED_MEDIA_TYPES.add(MediaType.APPLICATION_XML);
34 |         SUPPORTED_MEDIA_TYPES.add(MediaType.TEXT_XML);
35 |     }
36 |
37 |     public boolean canRead(Class<?> clazz, MediaType mediaType) {
38 |         return clazz.equals(String.class);
39 |     }
40 |
41 |     public boolean canWrite(Class<?> clazz, MediaType mediaType) {
42 |         return false;
43 |     }
44 |
45 |     public List<MediaType> getSupportedMediaTypes() {
46 |         return SUPPORTED_MEDIA_TYPES;
47 |     }
48 |
49 |     public List<CuentaSaldo> read(Class<? extends List<CuentaSaldo>> clazz, HttpInputMess
50 |         throws IOException, HttpMessageNotReadableException {
51 |
52 |         final String xmlResponse = getResponse(inputMessage.getBody());
53 |         LOG.debug("Received response from entity 767 " + xmlResponse);
54 |         return getCuentas(xmlResponse);
55 |     }
56 |
57 |     private List<CuentaSaldo> getCuentas(final String response) {
58 |
59 |         final List<CuentaSaldo> cuentas = new ArrayList<CuentaSaldo>();
60 |         final DOMSource source = (DOMSource) new DomSourceFactory().createSource(response
61 |         final NodeList nodelist = source.getNode().getChildNodes();
62 |
63 |         final Node cuentaNode = nodelist.item(0);
64 |         if (cuentaNode.getLocalName().equals("cuenta")) {
65 |             for (int i=0; i<nodelist.getLength(); i++) {
66 |                 final Node cuenta = nodelist.item(i);
67 |                 cuentas.add(getCuentaSaldo(cuenta));
68 |             }
69 |         }
70 |
71 |         return cuentas;
72 |     }
73 |
74 |
75 |     private CuentaSaldo getCuentaSaldo(final Node node) {
76 |         final CuentaSaldo cuenta = new CuentaSaldo();
77 |         String numeroCuenta = "";
78 |         String saldo = "";
79 |         if (node.getLocalName().equals("cuenta")) {
80 |             final NodeList cuentaAndSaldo = node.getChildNodes();
81 |             for (int i=0; i<cuentaAndSaldo.getLength(); i++) {
82 |                 final Node cuentaOrSaldoNode = cuentaAndSaldo.item(i);
83 |                 if (cuentaOrSaldoNode.getLocalName().equals("numeroCuenta")) {

```

```

84         numeroCuenta = cuentaOrSaldoNode.getTextContent();
85     } else if (cuentaOrSaldoNode.getLocalName().equals("importe")) {
86         saldo = cuentaOrSaldoNode.getTextContent();
87     }
88     }
89 }
90 cuenta.setCuenta(numeroCuenta);
91 cuenta.setSaldo(Float.valueOf(saldo));
92 return cuenta;
93 }
94
95
96 private String getResponse(InputStream inputStream) throws IOException {
97     if (inputStream != null) {
98         final Writer writer = new StringWriter();
99
100        char[] buffer = new char[1024];
101        try {
102            final Reader reader = new BufferedReader(new InputStreamReader(inputStream));
103            int n;
104            while ((n = reader.read(buffer)) != -1) {
105                writer.write(buffer, 0, n);
106            }
107        } finally {
108            inputStream.close();
109        }
110        return writer.toString();
111    } else {
112        return "";
113    }
114 }
115
116 public void write(List<CuentaSaldo> t, MediaType contentType, HttpOutputMessage output
117     HttpMessageNotWritableException {
118 }
119 }
120
121 }

```



11. Paso 7: obteniendo las cuentas del cliente para la entidad 955.

Y nos queda hacer lo mismo con la entidad 955. Recordemos que en este caso es una clase quien nos devolverá la lista de cuentas y saldos, por tanto, con un simple Service-Activator nos valdrá. En este ejemplo la clase corre dentro de la misma aplicación que Spring Integration (CuentasCliente955.java) pero si estuviese en otro entorno bastaría una invocación RMI.

```

1 <!-- Conecta con la clase java que devuelve las cuentas de los clientes de la entidad 955 -->
2 <int:channel id="cuentasUsuario955" />
3 <int:service-activator input-channel="cuentasUsuario955"
4   ref="cuentasCliente955ServiceActivator" output-channel="aggregatorResponseChannel" />
5 <bean id="cuentasCliente955ServiceActivator"
6   class="com.autentia.spring.integration.prueba_spring_integration.CuentasCliente955ServiceActivator"
7   <constructor-arg name="cuentasService" ref="cuentasCliente955" />
8 </bean>

```

Nuestro Service-Activator:

```

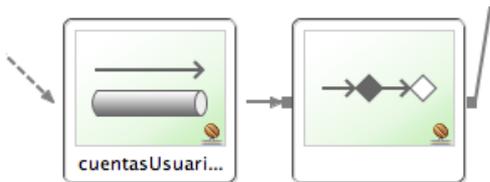
1 import java.util.ArrayList;
2 import java.util.List;
3
4 import org.apache.commons.logging.Log;
5 import org.apache.commons.logging.LogFactory;
6 import org.springframework.beans.factory.annotation.Autowired;
7 import org.springframework.integration.annotation.ServiceActivator;
8 import org.springframework.stereotype.Component;
9
10 import com.autentia.spring.integration.prueba_spring_integration.clientes_955.CuentaSaldo;
11 import com.autentia.spring.integration.prueba_spring_integration.clientes_955.CuentasCliente955;
12
13 @Component
14 public class CuentasCliente955ServiceActivator {
15
16     private static final Log LOG = LogFactory.getLog(CuentasCliente955ServiceActivator.class);
17
18     private final CuentasCliente955 cuentasCliente;

```

```

19
20 @Autowired
21 public CuentasCliente955ServiceActivator(CuentasCliente955 cuentasCliente) {
22     this.cuentasCliente = cuentasCliente;
23 }
24
25 @ServiceActivator
26 public List<CuentaSaldo> handleCuentas(UsuarioEntidad usuario) {
27     final List<CuentaSaldoCliente955> cuentas = cuentasCliente.getCuentasCliente(usu
28     LOG.debug("Received cuentas:" + cuentas + " from user: " + usuario.getUsuario())
29     return convertCuentaSaldo(cuentas);
30 }
31
32 private List<CuentaSaldo> convertCuentaSaldo(List<CuentaSaldoCliente955> cuentas) {
33     final List<CuentaSaldo> convertedCuentas = new ArrayList<CuentaSaldo>();
34     for (CuentaSaldoCliente955 cuenta : cuentas) {
35         convertedCuentas.add(new CuentaSaldo(cuenta.getCuenta(), cuenta.getSaldo()))
36     }
37     return convertedCuentas;
38 }
39

```



12. Paso 8: agregar todos los resultados y devolver la respuesta.

Bueno, pues ya queda poco. Lo único que nos falta es agrupar todos los resultados que nos han devuelto los servicios de las entidades bancarias y devolver los datos. Para ello haremos uso de un agregador (juntará todas las respuestas en un único mensaje) y un Service-Activator (devolverá la respuesta).

```

1 <int:channel id="aggregatorResponseChannel" />
2 <int:aggregator input-channel="aggregatorResponseChannel"
3     ref="cuentasAggregator" method="addCuentas" output-channel="cuentasResponseChannel" />
4
5 <int:channel id="cuentasResponseChannel" />
6 <int:service-activator input-channel="cuentasResponseChannel"
7     ref="cuentasResponseServiceActivator" />

```

El agregador recibe la lista de respuestas de todos los servicios anteriores y las une en un único mensaje (Cuentas):

```

1 import java.util.Collection;
2 import java.util.List;
3
4 import org.apache.commons.logging.Log;
5 import org.apache.commons.logging.LogFactory;
6 import org.springframework.integration.annotation.Aggregator;
7 import org.springframework.stereotype.Component;
8
9 @Component
10 public class CuentasAggregator {
11
12     private static final Log LOG = LogFactory.getLog(CuentasAggregator.class);
13
14     @Aggregator
15     public Cuentas addCuentas(List<Collection<CuentaSaldo>> cuentasEntidades) {
16         final Cuentas cuentas = new Cuentas();
17         for (Collection<CuentaSaldo> cuentasEntidad : cuentasEntidades) {
18             LOG.debug("Adding " + cuentasEntidad + " to final data");
19             for (CuentaSaldo cuentaSaldo : cuentasEntidad) {
20                 cuentas.addCuenta(cuentaSaldo);
21             }
22         }
23         return cuentas;
24     }
25 }

```

El Service-Activator recibe los resultados unificados en el bean Cuentas y prepara la respuesta:

```

1 import javax.xml.transform.Source;
2
3 import org.springframework.integration.annotation.ServiceActivator;
4 import org.springframework.stereotype.Component;
5 import org.springframework.xml.transform.StringSource;
6
7 @Component
8 public class CuentasResponseServiceActivator {
9
10     @ServiceActivator
11     public Source handleCuentas(final Cuentas cuentas) {

```

```

12     final StringBuilder stringBuilder = new StringBuilder();
13     stringBuilder.append("< cuentas >");
14     for (CuentaSaldo cuenta : cuentas.getCuentas()) {
15         appendCuenta(cuenta, stringBuilder);
16     }
17     stringBuilder.append("< / cuentas >");
18     return new StringSource(stringBuilder.toString());
19 }
20
21 private void appendCuenta(final CuentaSaldo cuenta, final StringBuilder stringBuilder)
22     stringBuilder.append("< cuenta >").append("< numero >").append(cuenta.getCuenta()).app
23         .append("< saldo >").append(cuenta.getSaldo()).append("< / saldo >").append("< /
24     }
25 }
26 }

```



13. Probando el ejemplo.

Hemos colgado todo el código fuente del ejemplo: tanto servicios como la plataforma con Spring Integration en un repositorio público <https://github.com/autentia/esb-tutorial>, para que se lo descargue quien quiera.

Para probarlo arrancamos los Servicios Web (localhost:8081) y el Servicio REST (localhost:8081), así como la aplicación de Spring integration (localhost:8080). Todas las aplicaciones que hemos colgado en el repositorio tienen su correspondiente .jmx para que puedan ser probados con JMeter.

Una vez tenemos todo arrancado enviamos la siguiente petición a la plataforma (aplicación con Spring Integration):

Y la respuesta que nos devuelve es la esperada, la lista de todas las cuentas y saldos de todas las entidades bancarias del cliente.

Ver Árbol de Resultados

Nombre: Ver Árbol de Resultados

Comentarios

Escribir todos los datos a Archivo

Nombre de archivo Log/Display Only: Escribir en Log Sólo Errores Successes

Petición Soap/XML-RPC

Resultado del Muestreador | Petición | Datos de Respuesta

- SOAP-ENV:Envelope
 - xmlns:SOAP-ENV = "http://schemas.xmlsoap.org/soap/envelope/"
 - SOAP-ENV:Header
 - SOAP-ENV:Body
 - cuentas
 - cuenta
 - numero
 - 088-2264-000325-0009
 - saldo
 - 1098.7
 - cuenta
 - numero
 - 767-0057-2291-000373-001
 - saldo
 - 505.5
 - cuenta
 - numero
 - 767-0059-0016-001082-006
 - saldo
 - 12.0
 - cuenta
 - numero
 - 955-88990-0098-0006
 - saldo
 - 13206.5

14. Referencias.

- [Spring Integration: Reference Manual.](#)
- [Introducción a Spring Integration](#)
- [Código fuente del tutorial.](#)

15. Conclusiones.

Después de la [introducción a Spring Integration](#) hemos querido hacer un ejemplo mucho más completo para demostrar todo lo que se puede hacer, o al menos una parte, con este sistema de integración. Recordemos que Spring Integration nos provee de mucha más funcionalidad de la que hemos visto en este tutorial, pero creo que con este ejemplo, cualquiera se puede hacer una idea del alcance de Spring Integration y, en general, de las plataformas de integración.

Espero que este tutorial os haya sido de ayuda. Un saludo.

Miguel Arlandy

marlandy@autentia.com

Twitter: [@m_arlandy](#)

A continuación puedes evaluarlo:

[Regístrate para evaluarlo](#)

Por favor, vota +1 o compártelo si te pareció interesante

Share |

¿Te gusta adictosaltrabajo.com? Síguenos a través de:



Anímate y coméntanos lo que pienses sobre este TUTORIAL:

Puedes opinar o comentar cualquier sugerencia que quieras comunicarnos sobre este tutorial; con tu ayuda, podemos ofrecerte un mejor servicio.

Enviar comentario

(Sólo para usuarios registrados)

» **Regístrate** y accede a esta y otras ventajas «



Esta obra está licenciada bajo [licencia Creative Commons de Reconocimiento-No comercial-Sin obras derivadas 2.5](#)

IMPULSA

Impulsores

Comunidad

¿Ayuda?

0 personas han traído clicks a esta página

sin clicks

+ + + + + + + +

powered by [karmacracy](#)

Copyright 2003-2012 © All Rights Reserved | [Texto legal y condiciones de uso](#) | [Banners](#) | [Powered by Autentia](#) |

