

The SOA Magazine

Feature Article



SOA with Spring (Part 1)

by Rizwan Ahmed

Published: February 5, 2010 (SOA Magazine Issue XXXVI: February 2010)

[Download PDF](#)

[Digg This](#) • [De.licio.us](#) • [Slashdot](#) • [Technorati](#) • [StumbleUpon](#) • [Google Bookmark](#)

Abstract: In an earlier article [REF-1], I had described an SOA solution for Adobe LiveCycle Forms functionality that was architected using a JAX-WS framework (JBossWS) and EJB3. In this two-part article series, I'll be demonstrating a similar solution architected and configured using Spring, a lightweight, open-source framework that is specifically created to provide a one-stop shop for all your enterprise application development needs using plain JavaBeans. Specific sections will address core architectural concerns such as the Web service infrastructure (specifically as it relates to endpoint setup, message definition, (un)marshalling, and routing), exceptions handling, security infrastructure and the client setup. Demonstrating Spring's modular and pluggable architecture, we start with Spring-WS, the specific Spring module created for Web services and plug-in other Spring modules such as OXM (Object-XML Mapping) specifically to address the Object/XML impedance mismatch, Security for authentication/authorization services, ORM (Object-Relational Mapping) to deal with persistence at the object level, and finally DAO for data access to a relational database storing user credential information. The objective here is to demonstrate how the Spring framework could be used to provide an end-to-end service-oriented solution as an alternative to JAX-WS and EJB3.

Introduction

Classic JEE application architecture generally employs EJB to implement business logic, which present day EJB3 frameworks have adapted to provide a POJO based programming model employing ideas such as dependency injection (DI) and aspect-oriented programming (AOP) all within the scope of an EJB container integrated within an application server to provide a scalable and highly optimized runtime environment. Spring [REF-2] is a lightweight alternative to EJB3, which provides a POJO based development construct woven into a convenient and highly flexible framework which runs in its own container separately from an application server. "Comparative Analysis of EJB3 and Spring Framework" [REF-3] provides a comprehensive comparison of the EJB3 and Spring framework based on a set of common evaluation criteria. Simplifying the development process while promoting test-driven development is the key goal of Spring which is based on two core features: inversion of control (IoC), which allows for injecting bean dependencies at creation time to have the services assembled via an XML configuration file, and AOP which allows for separation, through wrappers and interceptors, of infrastructure components (such as transaction, security management, logging etc.) from core application concerns (such as fulfillment of the core business functionality).

The Spring framework is composed of several pluggable modules built on top of the core container such as AOP, DAO, ORM, JMX, JCA, JMS, MVC, Security, OXM, Portlet, WS/Remoting which makes it possible to use only the module(s) you need in your particular application. Some of the features of Spring which architects could find advantageous are: its lightweightedness in terms of both size and overhead, its promotion of loose coupling between its various modules and the respective application objects (referenced as "beans") which are then assembled through dependency injection, its rich support for AOP, its concept of a lightweight "virtual" container, which essentially contains and manages the lifecycle and configuration of its

application beans composed declaratively within an XML file (also called application context and will be referred to quite frequently in the paper - you would typically have separate application context XML files for each Spring module that you would need to adapt) and finally, its provision for managing core infrastructure functionality such as transactions, persistence, messaging etc.

About Spring-WS

Spring-WS [REF-4] is a subproject of Spring that is focused on building contract-first Web services. While contract-first is not the popular development methodology of choice for most Web service developers, who instead much prefer contract-last, which essentially means that the service contract is defined after the service endpoint is deployed. In the contract-last development approach there is no need to manipulate complex WSDL documents and XSD files as the Web service framework infrastructure usually does that for you with a little help from the service class and annotations sprinkled therein [REF-1]. However, there is a small drawback: a contract-last Web service ends up being a reflection of its internal application's API which as we know, is subject to change far more than you (or your web clients) would like the external API to be. Changes to the internal API will mean changes to the automatically generated service contract which could ultimately require changes to the clients that are consuming your service. Therefore, a simple refactor at the application end may result in the client code to break. The solution to this problem is to first create the ideal contract to be presented to the world and then decide how it should be implemented. The contract should be written in "business terms" with little regard to what the underlying application will eventually look like and emphasizes what is expected of the service and not how it will be implemented. Next, we'll go over the steps for developing a contract-first Web service with Spring-WS. This article, Part 1 of the series, will cover message (data contract) definition, endpoint setup particularly as it relates to the interceptors and OXM infrastructure along with exceptions handling. In Part 2, I'll go over the security infrastructure, the configuration for automatic WSDL document generation and the client setup for consumption of the Web service.

Steps for Developing a Spring-WS Contract First Web Service

1. Define the Service's Data Contract

The single most important activity in developing a contract-first Web service is defining the contract. The service contract for a SOAP based Web service, generally expressed in a WSDL document, consists of two distinct and mutually separate parts: the data contract and the operational contract. The data contract defines, in an implementation agnostic manner, the messages that are to be sent to and received from the service and generally includes XML Schema (XSD) definitions of the message payloads that the service expects and responds with. XSD allows one to precisely define what should go into the message, which consists of the message elements, their respective types and constraints on the data. Example 1 shows the XSD for a service endpoint, later to be implemented in Spring-WS that takes in a `<GetClientDemographicsRequest>` XML message payload, retrieves an Adobe LiveCycle form pre-filled with field values from an external datasource [REF-1] and responds back with a `<GetClientDemographicsResponse>` XML message wrapping the form content, which will eventually be either sent back to a client browser or involved in a BPEL orchestration workflow [REF-5].

The operational contract, typically expressed using `<wsdl:operation>` will define the abstract operations that the service will perform with the input and output messages defined in the XML schema. Both the data and operational contract parts are defined in a single WSDL file along with binding information which tells the client how to invoke the operations you've just defined wrapped within a `<wsdl:binding>` element; and a service address location which tells it where to invoke it (wrapped within `<wsdl:service>`). WSDL is the standard for defining Web services [REF-6] although most developers would balk at the idea of creating a WSDL document by hand. However in Spring-WS, other than writing the XSD data contract which the developer has to do manually, the WSDL document generation itself is done automatically upon appropriately configuring the endpoint and the Spring-WS infrastructure. We'll get to that in the next part of the article series.

```
<?xml version="1.0" encoding="UTF-8" ?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
```

```

targetNamespace="http://scrs.forms.springws.service/"
xmlns:tns="http://scrs.forms.springws.service/"
xmlns:types="http://scrs.forms.springws.service/types"
elementFormDefault="qualified">

<import namespace=http://scrs.forms.springws.service/types
schemaLocation="types.xsd"/>

<element name="GetClientDemographicsRequest">
  <complexType>
    <sequence>
      <element minOccurs="0" name="contentURL" type="types:ContentURL"/>
      <element minOccurs="0" name="inputSSN" type="string"/>
      <element name="maskSSN" type="boolean"/>
      <element minOccurs="0" name="formNum" type="string"/>
      <element minOccurs="0" name="userAgent" type="string"/>
    </sequence>
  </complexType>
</element>

<element name="GetClientDemographicsResponse">
  <complexType>
    <sequence>
      <element minOccurs="0" name="formContent" type="string"/>
    </sequence>
  </complexType>
</element>
</schema>

```

Example 1: Service endpoint's data contract defined in messages.xsd

2. Writing and Configuring the Service Endpoint

The data contract we developed earlier (Example 1) only defines the messages sent to and from the service. To handle the message we have to specifically create message or payload endpoints in Spring-WS that will process messages (conforming to the pre-defined data contract) and translate it to API calls into internal objects. A Web service endpoint is typically a class that receives a SOAP message from the client and based on the content of the message payload, is appropriately transformed by the infrastructure, either into an XML object (such as DOM) or Java object (such as JAXB), and makes calls to internal application objects to perform the actual work of implementing the service. Typically, most endpoints are concerned only about the payload of the message, which happens to be the content of the SOAP body. Spring-WS provides a number of payload endpoint options (such as `AbstractDom4jPayloadEndpoint`, `AbstractDomPayloadEndpoint`, `AbstractJDomPayloadEndpoint`, `AbstractSaxPayloadEndpoint` etc.) that you would implement to handle SOAP message payloads as `Dom4j`, `DOM`, `JDOM` and `SAX ContentHandler Elements` respectively depending on which XML parsing technology you prefer (a personal favorite is the `AbstractJDomPayloadEndpoint` due to `JDOM`'s support for `XPath` which offers a simple way to extract information out of `JDOM` elements). Spring-WS's `AbstractMarshallingPayloadEndpoint` is a little different in that it supports automatic marshalling and unmarshalling of XML from Java objects using one of several OXM frameworks supported such as `JAXB`, `Castor`, `XMLBeans`, `XStream` etc. A marshalling endpoint is typically given a Java object to process (as opposed to the other types of endpoint options that are given an XML element to dissect for information) and can help in eliminating XML parsing code. The marshalling endpoint works in tandem with an unmarshaller to convert an incoming XML message into a POJO and once the endpoint is finished to convert the returned POJO into an XML message to be returned to the client.

Our service will be implemented within a Payload endpoint that will, per our data contract defined earlier in Example 1, process `<GetClientDemographicsRequest>` XML messages qualified by the `{http://scrs.forms.springws.service}` namespace (unmarshalled by an appropriate XML binding mechanism such as `JAXB`) and will respond back with `<GetClientDemographicsResponse>` messages (refer to Example 2). However, instead of an `AbstractMarshallingPayloadEndpoint` implementation, I have chosen to go with the Spring-WS provided annotation type endpoint. The `@Endpoint` (`org.springframework.ws.server.endpoint.annotation.Endpoint`) indicates that the annotated class is an endpoint and allows for that class to be auto-detected through classpath scanning. In Spring-WS, all Web service (SOAP message) requests are fronted through the `org.springframework.ws.transport.http.MessageDispatcherServlet` based on the URL mapping (typically

/services) defined in the application's web.xml. In other words, all web requests sent to the URL address "http(s)://<machine>:<port>/services" are handled as SOAP messages. The MessageDispatcherServlet then routes the SOAP message to the appropriate endpoint using an endpoint mapping defined in the Web service's application context XML file (refer to Example 3). The application context XML file is a composition of application objects (referenced as "beans") relevant to the particular action (in this case, the FormClientDemographicsEndpoint) or Spring module that you are trying to invoke. The Spring container will then create the objects, wire them together and manage their complete lifecycle.

There are several ways that message routing occurs: the simplest way is to use the Spring-WS's PayloadRootQNameEndpointMapping class that maps incoming SOAP messages to endpoints by examining the qualified name (QName) of the message's payload (in our case it will be {http://scrs.forms.springws.service} <GetClientDemographicsRequest>) and looking up the endpoint from its list of mappings configured through an "endpointMap" property. I'm demonstrating another approach that uses the PayloadRootAnnotationMethodEndpointMapping class to essentially auto-detect instances of classes annotated with @Endpoint and uses the @PayloadRoot annotation to map request payload elements to endpoint methods. For an example list of the endpoint class, please refer to Example 2. The actual endpoint mapping shown in Example 3 configured in the application context as bean "annotationMapping" that detects the @PayloadRoot annotations on the @Endpoint bean methods. We see that the bean is wrapping several mapped interceptor implementations of the Spring-WS EndpointInterceptor interface (for example PayloadLoggingInterceptor or alternately, SoapEnvelopeLoggingInterceptor which perform logging of the message payload and SOAP envelope respectively), the PayloadValidatingInterceptor that, per its namesake, validates the payload per the data contract XSD defined earlier and finally the bean referenced as "wsSecurityInterceptor" which performs security functions (authentication and authorization of the SOAP message). I'll discuss the security configuration in a separate section. The MessageDispatcherServlet will first call each EndpointInterceptor's handlerRequest method in the given order, finally invoking the endpoint itself after all handlerRequest methods have returned true.

@Endpoint

```
public class FormClientDemographicsEndpoint
implements FormsWebServiceConstants {

    private static final Log logger =
        LoggerFactory.getLog(FormClientDemographicsEndpoint.class);

    private final FormClientDemographicsService
formClientDemographicsService;

    private ObjectFactory objectFactory = new ObjectFactory();

    public FormClientDemographicsEndpoint(FormClientDemographicsService
formClientDemographicsService) {
        this.formClientDemographicsService = formClientDemographicsService;
    }

    /**
     * This endpoint method uses marshalling to handle message with
     a GetClientDemographicsRequest payload.
     *
     * @param request the JAXB2 representation of a
     GetClientDemographicsRequest
     */
    @PayloadRoot(localPart = GET_CLIENT_DEM_REQUEST, namespace =
MESSAGES_NAMESPACE)
    public GetClientDemographicsResponse
callRenderForm(GetClientDemographicsRequest
request) throws DatatypeConfigurationException, FormServerException {
        if (logger.isDebugEnabled()) {
            logger.debug("Received GetClientDemographicsRequest '"
+ request.getInputSSN() + "' form num '" + request.getFormNum());
        }

        String formContent =
formClientDemographicsService.callRenderForm(request);
```

```

    GetClientDemographicsResponse response = new
    GetClientDemographicsResponse();
    response.setFormContent(formContent);
    return response;
}
}

```

Example 2: A Spring-WS Endpoint that uses annotations to map methods to request payload elements

```

<bean id="messageFactory"
class="org.springframework.ws.soap.saaj.SaajSoapMessageFactory"/>

<bean id="messageReceiver"
class="org.springframework.ws.soap.server.SoapMessageDispatcher"/>

<bean id="schemaCollection"
class="org.springframework.xml.xsd.commons.CommonsXsdSchemaCollection">
  <description>
    This bean wrap the messages.xsd
    (    which imports types.xsd), and inlines them as a one.
  </description>
  <property name="xsds" value="/messages.xsd"/>
  <property name="inline" value="true"/>
</bean>

<!--===== ENDPOINTS ===== -->

<bean id="marshallingEndpoint"
class="scrs.forms.service.springws.ws.FormClientDemographicsEndpoint">
  <description>
    This endpoint handles the Web Service
    messages using JAXB2 marshalling.
  </description>
  <constructor-arg ref="formClientDemographicsService"/>
</bean>

<oxm:jaxb2-marshaller id="marshaller"
contextPath="scrs.forms.service.springws.parameters"/>

<!-- ===== ENDPOINT MAPPINGS ===== -->

<bean id="annotationMapping" class="org.springframework.ws.server.
endpoint.mapping.PayloadRootAnnotationMethodEndpointMapping">
  <description>
    Detects @PayloadRoot annotations on @Endpoint bean methods.
    The FormClientDemographicsEndpoint
    has such annotations. It uses two interceptors: one that
    logs the message payload, and the other validates
    it accoring to the messages.xsd' schema file.
  </description>
  <property name="interceptors">
    <list>
      <bean class="org.springframework.ws.server.endpoint.
interceptor.PayloadLoggingInterceptor"/>
      <bean
        class="org.springframework.ws.soap.server.endpoint.
interceptor.SoapEnvelopeLoggingInterceptor"/>
      <bean class="org.springframework.ws.soap.server.endpoint.
interceptor.PayloadValidatingInterceptor">
        <property name="xsdSchemaCollection" ref="schemaCollection"/>
        <property name="validateRequest" value="true"/>
        <property name="validateResponse" value="true"/>
      </bean>
      <ref bean="wsSecurityInterceptor"/>
    </list>
  </property>
</bean>

<!-- ===== ENDPOINT ADAPTERS ===== ?

```

```
<sws:marshalling-endpoints/>
```

*Example 3: Extract of the Web service's application context configuration as defined in applicationContext-*ws.xml**

The key to translating objects to and from XML messages is object-XML mapping (OXM). Spring provides an abstraction layer over several popular and widely used OXM frameworks including JAXB and Castor. The central elements of Spring's OXM support [REF-7] are its Marshaller and Unmarshaller interfaces. Implementations of Marshaller would serialize an object to XML and the Unmarshaller would deserialize an XML stream to an object. JAXB stands for Java Architecture for XML binding [REF-8] and consists of two core tools: the XML Schema compiler (xjc) that translates a W3C XML Schema into one or more Java classes and the XML Schema generator (schemagen) that derives a schema from a set of Java classes. Since the data contract is an abstraction of what we want the message request and response payloads to be, the XML Schema compiler (xjc) tool is run off the service endpoint's data contract XSD. The result is a mass of JAXB generated classes representing the schema derived classes representing the binding of schema type definitions (refer to Example 4), element declarations and model groups, endpoint request/response objects, their related object types and the ObjectFactory which allows for programmatically constructing new instances of the Java representation of XML content.

An instance of marshaller (or unmarshaller) can be wired in the application context using the "contextPath" property (please refer to the bean "marshaller" in Example 3), a list of colon separated Java package names that contain the priorly generated schema derived classes. When the FormClientDemographicsEndpoint receives a message after passing through the interceptor chain, it hands it off to to an Unmarshaller to unmarshal the XML message into a Java object using the JAXB infrastructure abstracted into an org.springframework.xml.jaxb.Jaxb2Marshaller instance. After the endpoint has finished its processing, the object returned is given to a Marshaller to marshal the object into XML that is returned to the client. A UML diagram detailing the sequence of processing is shown in Figure 1.

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "", propOrder = {
    "contentURL",
    "inputSSN",
    "maskSSN",
    "formNum",
    "userAgent"
})
@XmlRootElement(name = "GetClientDemographicsRequest")
public class GetClientDemographicsRequest {

    protected ContentURL contentURL;
    protected String inputSSN;
    protected boolean maskSSN;
    protected String formNum;
    protected String userAgent;

    // Getters and setters for the various schema type definitions
    .....
}
```

Example 4: The messages.xsd schema derived GetClientDemographicsRequest class

The actual implementation of the service is shown in Example 5. A properly written Spring-WS endpoint should not perform any business logic of its own, rather it should only mediate between the client and the business API. Referring back to Example 3 we see that our WS endpoint is wired up with the service bean "formClientDemographicsService" through constructor injection.

```
@Service
public class FormClientDemographicsServiceImpl
implements FormClientDemographicsService {

    private static final Log logger =
LogFactory.getLog(FormClientDemographicsServiceImpl.class);

    @Secured({"ROLE_SCRS_FORMS"})
    public String callRenderForm(GetClientDemographicsRequest inputObj)
throws FormServerException {
```

```

if (logger.isDebugEnabled()) {
    logger.debug("Received GetClientDemographicsRequest '"
        + inputObj.getInputSSN() + "' form num '" + inputObj.getFormNum());
}

// Implementation of the Adobe FSM renderForm()
specific logic as detailed in [REF-1]

if (logger.isDebugEnabled()) {
    logger.debug("Returning formContent " + formContent);
}
return formContent;
}
}

```

Example 5: The actual service implementation

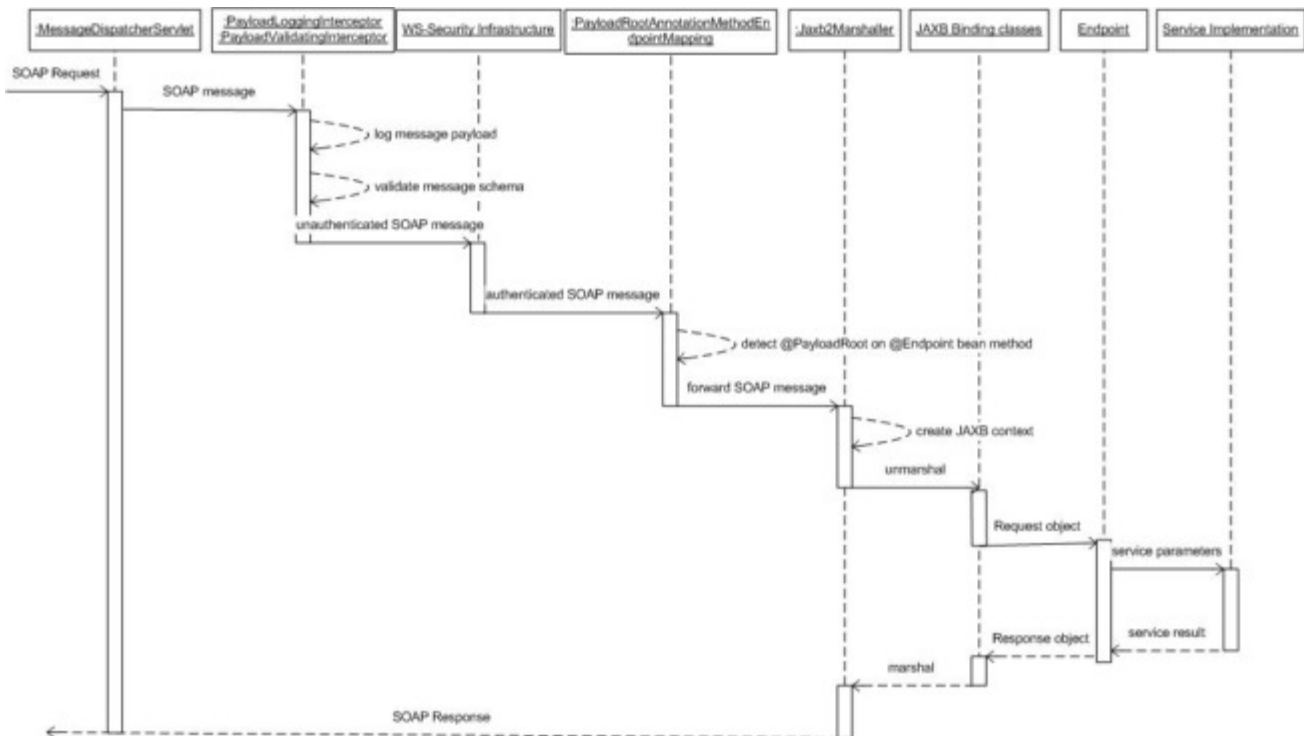


Figure 1: Sequence diagram of the Web service infrastructure with Spring. (Click [here](#) to view a larger version of this image.)

Handling Endpoint Exceptions

If an exception is thrown in the course of processing a message, a SOAP fault will need to be sent back to the client. SOAP based Web services generally communicate failure using SOAP faults, not Java exceptions, so therefore, we need a way to convert any Java exceptions thrown by our Web service or even Spring-WS into SOAP faults. For that purpose, Spring-WS provides a `SoapFaultMappingExceptionResolver` exception mapper which will handle any uncaught exceptions that occur in the course of handling a message and produce an appropriate SOAP fault that will be sent back to the client. The "exceptionsMapping" property is configured with property elements in key-value pairs containing Java exceptions mapped to SOAP fault codes (refer to Example 6 - snippet of the Web service's application config file `applicationContext-ws.xml`). The key of each element is a Java exception that needs to be translated to a SOAP fault. The value of the `<prop>` is a two part value where the first part is the type of fault that is to be created and the second part is a string that describes the fault.

SOAP faults come in two types: sender and receiver faults. Sender faults typically indicate that the problem is on the client (i.e. sender) side. Typical sender issues would be sending an incorrectly formatted XML message that either cannot be unmarshalled or validated by the underlying OXM framework (JAXB). The JAXB thrown exception is converted by Spring into its own exception hierarchy with the

XmlMappingException as the abstract root. This runtime exception wraps the original exception thrown within the OXM framework so that no information is lost (please refer to Figure 2). In addition, the MarshallingFailureException and UnmarshallingFailureException provide a distinction between marshalling and unmarshalling operations even though the underlying OXM framework does not do so. In our case, if the endpoint receives an XML message that cannot be unmarshaled, the marshaller will throw an org.springframework.oxm.UnmarshallingFailureException. Because the sender created the useless XML, this is the sender's fault and the appropriate SOAP fault message will be sent back as "Invalid request received". An org.springframework.oxm.ValidationFailureException is handled the same way.

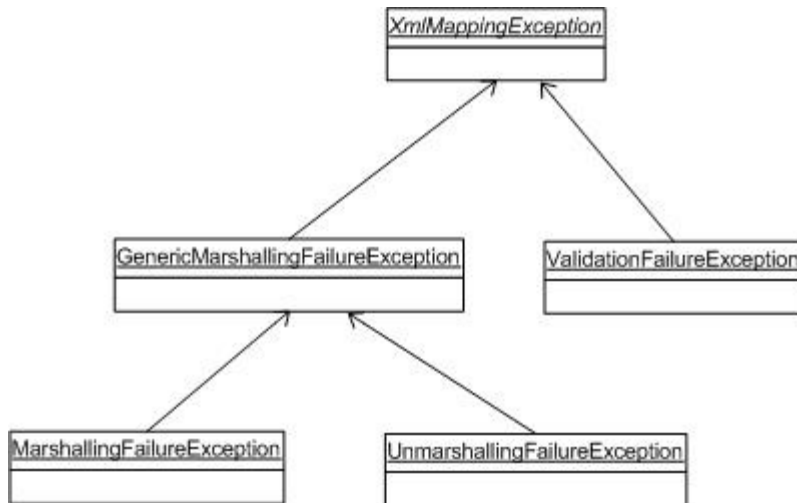


Figure 2: Spring's OXM exception hierarchy [REF-7]

Receiver fault indicates that the problem is with the Web service (i.e. receiver) that received the message but is having some problem processing the message. In our case, we're assuming that if the exception thrown does not match any of the mapped exceptions then it must be a problem on the receiving side. Therefore, any exceptions not explicitly mapped in the "exceptionMappings" property will be handled by the fault definition in the "defaultFault" property (again, refer to Example 6). Thus, it's the receiver's fault and the message simply says "Server error".

In addition to SoapFaultMappingExceptionResolver, which in effect, stores mappings of Java exceptions to SOAP faults within the application configuration properties, Spring-WS also provides the SoapFaultAnnotationExceptionResolver which basically maps exception classes with @SoapFault annotations to SOAP faults (see Example 6). Therefore, we broadly classify two sorts of exceptions that we want to handle: business logic exceptions which are typically thrown in the service implementation class (refer to the callRenderForm method in the service implementation class in Example 5 that throws a FormServerException) and for which we have an exception bean defined annotated with @SoapFault (refer to Example 7); along with other runtime exceptions which do not have the annotation necessitating the need for the two different exception resolvers defined in Example 6.

```

<!-- ===== ENDPOINT EXCEPTION RESOLVER ===== -->

<!--
Endpoint exception resolvers can handle exceptions
as they occur in the Web service. We have two sorts of
exceptions we want to handle: the business logic
exceptions which have a @SoapFault annotation, and other
exceptions, which don't have the annotation.
Therefore, we have two exception resolvers here.
-->

<bean class="org.springframework.ws.soap.server.endpoint.
SoapFaultAnnotationExceptionResolver">
  <description>
    This exception resolver maps exceptions
    with the @SoapFault annotation to SOAP Faults. The business
    logic exception FormServerException has this.
  </description>
</bean>
  
```

```

<bean class="org.springframework.ws.soap.server.endpoint.
SoapFaultMappingExceptionResolver">
  <description>
    This exception resolver maps other exceptions to SOAP Faults.
    Both UnmarshallingException and
    ValidationFailureException are mapped to a SOAP
    Fault with a "Client" fault code.
    All other exceptions are mapped to a "Server" error code, the
default.
  </description>
  <property name="defaultFault" value="SERVER, Server error"/>
  <property name="exceptionMappings">
    <props>
      <prop key="org.springframework.xml.UnmarshallingFailureException">
        CLIENT, Invalid request received
      </prop>
      <prop key="org.springframework.xml.ValidationFailureException">
        CLIENT, Invalid request received
      </prop>
    </props>
  </property>
</bean>

```

Example 6: The two different types of exception resolvers defined in the Web service's application context applicationContext-ws.xml

```

@SoapFault(faultCode = FaultCode.SERVER)
public class FormServerException extends Exception
{
  private int errorCode;
  private String errorCategory;
  private String[] stackTraceArr;

  public FormException(String errorCategory,
int errorCode, String message, String[] stackTraceArr)
  {
    super(message);
    this.errorCategory = errorCategory;
    this.errorCode = errorCode;
    this.stackTraceArr = stackTraceArr;
  }

  // getters and setters
}

```

Example 7: The business level exception bean resolved by SoapFaultAnnotationExceptionResolver

Conclusion

Spring is a leading open source, POJO framework providing a one-stop shop for all your enterprise application development needs. A lightweight container, it provides automated configuration and wiring of application objects in XML context files, which the framework is then capable of assembling a complex runtime system from. In this article we've seen how to define the data contract, a key component of Spring-WS driven Web services, configure the endpoint particularly with regards to message routing, OXM infrastructure and handling exceptions. In Part 2, we'll go over the security infrastructure, the configuration for automatic WSDL document generation and the client setup for consumption of the Web service.

References

- [REF-1] "An SOA Case Study: Integrating Adobe LiveCycle Forms using JBossWS" by Rizwan Ahmed, SOA Magazine July 2009, <http://www.soamag.com/I30/0709-3.php>
- [REF-2] "About Spring", Spring Source, <http://www.springsource.org/about>
- [REF-3] "Comparative Analysis of EJB3 and Spring Framework" by Janis Graudins and Larissa Zaitseva, International Conference on Computer Systems and Technologies - CompSysTech 2006,

<http://ecet.ecs.ru.acad.bg/cst06/Docs/cp/SIII/IIIA.18.pdf>

[REF-4] "Spring Web Services", Spring Source, <http://static.springsource.org/spring-ws/sites/1.5/>

[REF-5] "Showcasing the Key Design Principles of SOA - A Case Study: Orchestrating Services using open source BPEL" by Rizwan Ahmed, SOA World Magazine November 2009, <http://soa.sys-con.com/node/1199986>

[REF-6] "Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language", W3C Recommendation, <http://www.w3.org/TR/wsdl20/>

[REF-7] "Spring Framework, Chapter8. Marshalling XML using O/X Mappers", Spring Source, <http://static.springsource.org/spring-ws/sites/1.5/reference/html/oxm.html>

[REF-8] "Unofficial JAXB User Guide", Java.net, <https://jaxb.dev.java.net/guide/>

[REF-9] "Spring in Action" by Craig Walls, Manning Publications Co., <http://www.manning.com/walls3/>

THE PRENTICE HALL SERVICE-ORIENTED COMPUTING SERIES FROM THOMAS ERL



[Home](#) [Past Issues](#) [Contributors](#) [What is SOA?](#) [SOA Glossary](#) [SOA School](#) [SOA Books](#) [About/RSS](#)
[Legal](#)

Copyright © 2006-2010
SOA Systems Inc.