

# The SOA Magazine

## Feature Article



### An SOA Case Study: Integrating Adobe LiveCycle Forms using JBossWS

by Rizwan Ahmed

Published: July 30, 2009 (SOA Magazine Issue XXX: July 2009)

[Download PDF](#)

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

*Abstract: JBossWS is a framework which implements the JAX-WS 2.0 (a replacement for the earlier JAX-RPC) specification and defines the programming/runtime model for implementing web services as a remoting mechanism for distributed service-oriented architectures [REF-1]. JBossWS is targeted at the Java SE 6/EE 5 platform and integrates with most current and earlier JBoss Application Server releases. In this article, I demonstrate how easy it is to use the JBossWS framework to harness the power of Java 5 annotations as well as endpoint associated metadata provided by JSR 181 annotations to develop, provide, consume and secure a service. The annotations ensure that the source code will be fairly simple and straightforward POJOs with the onus of processing falling to JBossWS tools at the client and server to create the contract and infrastructure plumbing needed at deployment and runtime. In order to illustrate the concepts, I have used the example of a real life service that allows different systems within our organization (a State Government Agency) to integrate with a centralized Forms system to retrieve, deliver and process intelligent documents to and from end users.*

#### Introduction

As an integral part of daily operations, our agency needs to manage hundreds of different forms in various business functional areas - from customer enrollment and benefit applications to federal tax and compliance reporting. The Adobe LiveCycle® Forms software allows us to create and deploy XML based form templates rendered as PDF and HTML at runtime for use with the Adobe Reader® software or any standardized web browser. The visual fidelity of PDF facilitates usability and having standardized, easily readable document formats for easy archiving and indexing into a Document or Imaging management system.

The adoption of well designed, automated, secure forms processes uniformly integrated with internal legacy systems and public-facing online systems was identified as key to increasing data accuracy, improving service delivery time and reducing administrative costs and manual intervention/data-entry. Leveraging the ubiquity of XML for data exchange and emerging Web services standards allows us to create centralized reusable forms components and processes for consumption by external calling systems.

#### About Adobe LiveCycle Forms

The Adobe LiveCycle Form Server Module (hereafter referred to as FSM) provides a set of services and client APIs which can be used to create interactive applications using data pre-fill capabilities to render pre-populated forms as either PDF or HTML onto a client device such as a browser and capture data from submitted forms for integration with core back-end systems. FSM provides a stateless Axis Web service endpoint and an EJB endpoint for rendering form content and processing form submissions. Both endpoints are invocable only via FSM client APIs which abstract the details of the actual invocation of the SOAP/EJB RMI calls behind factory objects. When the FSM service receives a request for a form, it uses a set of transformations to merge data with a form design and then delivers the form in a format that best matches the presentation and form filling capabilities of the target browser. Processing form submission can happen next which essentially involves using the FSM client APIs to

handle either the interpage navigation or form submit of the data (which may have been updated at this point) from the client browser.

## Typical Use Cases

### A. Client System Rendering a Form

The use case idea here is that the calling application would need to retrieve a form pre-filled with field values obtained from an external datasource. Call the FSM Client API `renderForm()` method (refer to Listing 1). You would need to pass in the form design number, preference option (HTML or PDF), a byte array containing an XML representation of form field name and values, the absolute path to the form designs repository on the file system of the server that hosts the FSM service and the target URL which defines the web-form action for the HTML rendered form content (this URL would typically map to a servlet or action class encapsulating the form submit processing logic - refer to Listing 2). The result set is the pre-populated rendered HTML content as a string or the PDF content as a binary byte array which can then be sent back on the `ResponseStream` to the browser for display.

```
// Create an FSM Endpoint proxy
Properties props = new Properties();
props.setProperty(FormServerFactory.CLASSNAME_PROP,
"com.adobe.formServer.client.SOAPClient");
props.setProperty(FormServerFactory.ENDPOINT_PROP, soapEndpoint); // FSM Axis
WS endpoint

IFormServer ifs = FormServerFactory.create(props);

IOutputContext ioc =
    ifs.renderForm
    (
        sFormQuery, // Desired form design number to render
        sFormPreference, // Preference option eg. HTML/PDF
        cData, // XML representation of data that requires merging into form
        "OutputType=0",
        sUserAgent, // user-agent string from the calling browser
        sApplicationWebRoot, // URL context of the Adobe FSM
        sTargetURL, // URL of the target action the rendered form will submit to
        sContentRootURI, // Absolute path to the form designs repository
        null
    );

// Rendered form content as HTML if requested in preference option
String formContent = ioc.getOutputString();

// Alternately, rendered form content as PDF
byte[] generatedForm = ioc.getOutputContent();
```

#### Listing 1

### B. Client System Processing Form Submission

The use case idea here is that the calling application would need to submit data either as a result of page navigation or submit a form to post data back into the datasource. Call the FSM client API method `processFormSubmission()` passing in the byte array of the submitted form field data (refer to Listing 2). Upon determining the processing state of the submitted form, that is, if the user simply requested page navigation then the newly requested HTML content is sent back.

```
// Create an FSM Endpoint proxy
Properties props = new Properties();
props.setProperty(FormServerFactory.CLASSNAME_PROP,
"com.adobe.formServer.client.SOAPClient");
props.setProperty(FormServerFactory.ENDPOINT_PROP, soapEndpoint); // FSM Axis
WS endpoint

IFormServer ifs = FormServerFactory.create(props);
byte[] byteArray = inputObj.getContentArray().getByteArray(); // form field
submitted data
```

```

IOutputContext ioc =
ifs.processFormSubmission(byteArray, "HTTP_REFERER=referrer&HTTP_CONNECTION=keep-
alive&CONTENT_TYPE=
application/x-www-form-
urlencoded", null, "exportDataFormat=XDP&CacheEnabled=False&XMLData=true" );

String formContent = ioc.getOutputString();

```

### Listing 2

## Problem Domain

We have the above use cases (A. and B.) working great when the FSM client APIs and dependant libraries are included with the calling application. But what if, as in our case, we have multiple systems needing to use the Adobe Form services within their individual application contexts. Why would we need to include the client APIs, libraries and method calls within each single application that needs to access the FSM provided services?

Following the principles of lean service-oriented architecture [REF-2], the solution is to create a simple JAX-WS service façade callable from any external application. The objective is that all implementation details of how the form rendering and form processing occurs is hidden away into the service implementation and all clients (including various enterprise applications) that need access can easily lookup and consume the provided services via published contracts (WSDL definitions) in a hub and spoke model that could easily be extended into a message bus based model for a better integrated IT landscape.

Please refer to Figure 1 which shows the high level component diagram for our internal landscape. The EES (Electronic Employer Services), Member Access and PAC (Payments and Claims) systems need access to services provided by Adobe LiveCycle FSM. Since these services are only invocable via FSM client APIs, the form rendering and form submission processing specific code is wrapped into a JAX-WS Stateless Session Bean (SLSB) web service endpoint packaged into a custom Forms application. The external systems also need their forms auto-populated based on case and social security numbers with customer specific information (eg. name, address and other demographic information) available within the ScrsCommon CRM application. Rather than integrate the data lookup within ScrsCommon into each individual application and then pass that data as parameters to the Forms JAX-WS Service, it is more optimal to create JAX-WS service implementation for ScrsCommon data lookup functionality and have it referenced as a domain object in the Forms service implementation.

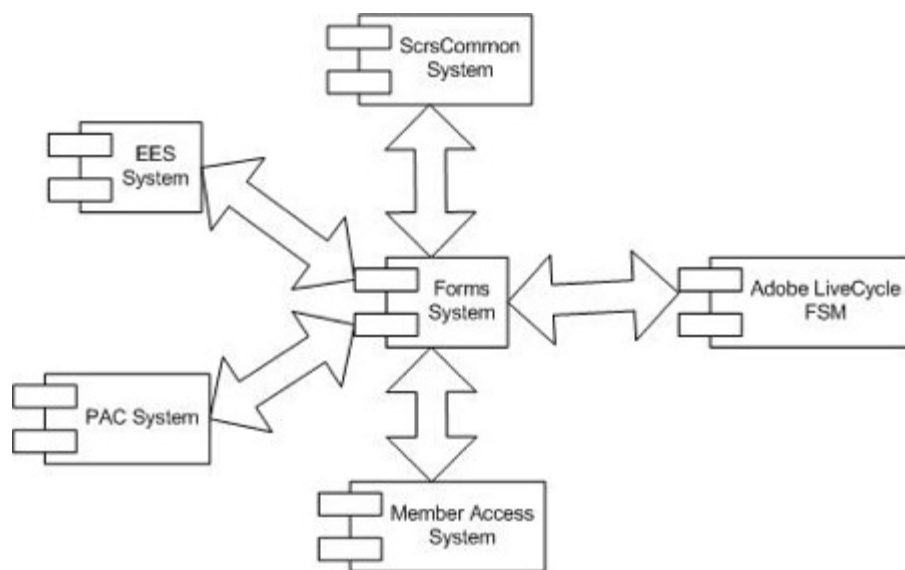


Figure 1: High-Level component diagram of our internal landscape

## Creating Developer Friendly Web Services with JBossWS

The JAX-WS specification, implemented by JBossWS [REF-1], is a high-level API which doesn't require you to know much about Web services metadata (WSDL definitions) which define the XML types (schemas) for each of the message parts used in service operation requests and responses. The ease of development using this API can

be gauged from the fact that the entire XML layer is hidden from developers who instead just work with objects generated by JBossWS tools, which encapsulate all of the work of creating SOAP messages, invoking the service and parsing the response. JAX-WS does not have its own binding or (un)marshalling mechanism and delegates all conversions from Java object to XML and vice versa to the JAXB2.0 API.

There are 2 major approaches to developing Web services with JBossWS:

#### *A. Contract First (Top Down)*

Develop the WSDL definition by hand using an XML editor or IDE. Use JBossWS tools such as wsconsume to generate portable server and client artifacts. To avoid the unnecessary pain that comes with hand crafting the WSDL document from scratch, one would use this approach only for modifying or replacing pre-existing services and contracts only.

#### *B. Contract Last (Bottom Up)*

Create a Service Endpoint interface (SEI) and endpoint implementation classes as Java source files. Use the JBossWS tool wsprovide passing in the SEI to generate the WSDL definition and portable client side artifacts. Generally faster to develop than Top Down and is to be used when you want to provide a new service or contract.

The tools generated client, service and JAXB 2.0 value classes are all heavily annotated. Use of annotations along with runtime tools that provide dependency injections, thankfully hidden away in the processing framework, makes development and maintainability easier for developers (in other words no more bloated code, fancy deployment descriptors and portability issues).

#### *JBossWS Endpoints*

Generally, the Web service endpoint is a server side component that receives SOAP messages, processes or optionally delegates further processing to a business delegate and returns a SOAP message response. Per JAX-WS there can be 2 different types of Web service Endpoints: POJO (Plain Old Java Objects) and EJB3 SLSB (Stateless Session Bean).

A POJO endpoint will be deployed as a servlet and packaged into a WAR file while an SLSB endpoint will be deployed as an EJB3 bean and packaged into a JAR (or EAR). The server (JBoss) automatically generates and publishes the abstract contract (WSDL definition + schema) for client consumption. A successfully deployed service will show up in the JBossWS Service Endpoint Manager (<http://hostname:8080/jbossws/services>). This is also where you will find the links to the generated WSDL document.

#### *Service Endpoint Interface*

All JAX-WS services typically implement a native Java service endpoint interface (SEI) which provide a high level Java-centric abstraction that hides the details of converting between Java objects and their XML representations for use in XML-based messages. Note that in some cases it is desirable for services to be able to operate at the entire SOAP Message or Payload level for which you would use the Provider interface which offers an alternative to SEIs. If your endpoint is an SLSB, the Remote interface for it may serve as the SEI.

#### *Creating a JBossWS SEI and Bean Implementation for Rendering a Form*

The `@WebService` annotation (`javax.jws.WebService`) indicates that a Java class implements a Web service. If used on the Remote interface of an SLSB (decorated with the `@Remote` annotation), it serves as a Web service endpoint interface (refer to Listing 3). The element name maps to a `wsdl:portType` element in the WSDL document. The `targetNamespace` specifies the namespace the service will be defined in. The `@SOAPBinding` defines the style attribute of the implementation detail from the Java domain.

While there are several styles available, based on interoperability requirements and business model need, I particularly like to use the Document/Literal Bare (`SOAPBinding.ParameterStyle.BARE`) style which uses a Java Bean to represent the entire payload [REF-3] and is particularly useful when working with a single parameter/return value if you don't want to clutter your service methods with additional annotations and multiple parameter elements.

In the Example, I have created a Service Endpoint Interface that returns the requested form design (passed as a

parameter in the FormClientDemographicsInput object) pre-populated with Customer/Client Demographic information from an external web service reference. All implementation details of how the form is rendered is encapsulated within the bean which simply returns the FormClientDemographicsOutput object containing the rendered form's HTML content as a String or PDF content as a binary byte array.

```
// Service Endpoint Interface SEI Class
@Remote
@WebService(name = "FormClientDemographics", targetNamespace =
"http://scrs.forms.service")
@SOAPBinding(parameterStyle = SOAPBinding.ParameterStyle.BARE)
public interface FormClientDemographics {
    @WebMethod
    public FormClientDemographicsOutput
callRenderForm(FormClientDemographicsInput inputObj) throws FormException;
}
```

### Listing 3

In the implementation class (refer to Listing 4), the @Stateless annotation signifies a Stateless Session Bean, the @WebService annotation must have the following properties: endpointInterface - fully qualified class name of the SEI, name - SEI name mappable to the wsdl:portType and serviceName - mappable to the wsdl:service (please refer to the relevant WSDL document elements in Listing 5). The @WebContext defines the application context root that the endpoint bean will be available at. The @Security and @RolesAllowed annotations are addressed in the Security section of this article. The @WebServiceRef is used to declare a runtime injectable reference (resource in the Java EE5 sense) to the CustomerLookupService implementation in the ScrsCommon system, the domain data of which is used for pre-filling the form content.

```
// SLSB Implementation Class
@Stateless
@WebService(endpointInterface = "scrs.forms.service.FormClientDemographics",
name = "FormClientDemographics", serviceName = "FormClientDemographicsService")

@WebContext(contextRoot = "/formsWS", authMethod="BASIC")
@SecurityDomain("JBossWS")
@RolesAllowed("friend")
@BindingType(value = "http://schemas.xmlsoap.org/wsdl/soap/http?mtom=true")
public class FormClientDemographicsBean implements FormClientDemographics
{
    // Runtime injectable WS Reference to the CustomerLookupService on the
ScrsCommon system
    @WebServiceRef(wsdlLocation = "META-INF/wsdl/CustomerLookupService.wsdl")
    private CustomerLookupService customerLookupService;
    private CustomerLookup customerPort;

    public CustomerLookup getCustomerLookupPort() {
        return customerLookupService.getCustomerPort();
    }

    public FormClientDemographicsOutput
callRenderForm(FormClientDemographicsInput inputObj) throws FormException
    {
        try{
            byte[] cData =
getCustomerLookupPort().getCustomerData(inputObj.getInputSSN());
            // Implementation of the Adobe FSM renderForm() specific logic as shown
in Listing 1
            ...
        } catch (Exception ex) {
            throw new FormException(...);
        }
    }
}
```

### Listing 4

```
<portType name='FormClientDemographics'>
    <operation name='callRenderForm' parameterOrder='callRenderForm'>
        <input message .../>
```

```

    <output message ../>
    <fault message ../>
  </operation>
</portType>
.....
<service name='FormClientDemographicsService'>
  <port binding='nsl:FormClientDemographicsBinding'
    name='FormClientDemographicsPort'>
    <soap:address location=
      'http://<machine>:<portnum>/formsWS/FormClientDemographicsBean' />
    </port>
  </service>

```

**Listing 5**

The Service Parameters/Return values are best handled by encapsulating them in individual Value Objects (in our case the FormClientDemographicsInput and FormClientDemographicsOutput respectively). Since the classes will be used by JAXB for serialization they must implement the java.io.Serializable interface. By default, JAXB serializes all public fields and properties so therefore typically one must use the @XmlAccessorType to choose only annotated fields to be serialized. The @XmlType is often used in conjunction to map a class or enum type to an XML Schema type. Properties and Fields on that class would therefore map to a complex type with the propOrder element determining the sequence order of the XML elements. You would also need to specify the target namespace of the XML Schema to that of the Service using the annotation element namespace. You may define as many embedded classes to any level of depth, the only prerequisite is that they all implement the java.io.Serializable interface and be decorated with the @XmlType annotation.

Listing 6 shows the Java value object that gets passed as the IN parameter to the service. Listing 7 shows the relevant schema definition in the generated WSDL document.

```

@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(
  name = "FormClientDemographicsInputType",
  namespace="http://scrs.forms.service",
  propOrder = { "contentURL", " inputSSN ", "formNum", " userAgent " }
)
public class FormClientDemographicsInput implements Serializable {
  private ContentURL contentURL;
  private String inputSSN;
  private String formNum;
  private String userAgent;

  // Public getters and setters
}

```

**Listing 6**

```

<types>
  <xs:schema targetNamespace='http://scrs.forms.service' version='1.0'
    xmlns:tns='http://scrs.forms.service'
    xmlns:xs='http://www.w3.org/2001/XMLSchema'>
  <xs:element name='callRenderForm' nillable='true'
    type='tns:FormClientDemographicsInputType' />
  <xs:complexType name='FormClientDemographicsInputType'>
    <xs:sequence>
      <xs:element minOccurs='0' name='contentURL' type='tns:contentURL' />
      <xs:element minOccurs='0' name='inputSSN' type='xs:string' />
      <xs:element minOccurs='0' name='formNum' type='xs:string' />
      <xs:element minOccurs='0' name='userAgent' type='xs:string' />
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name='contentURL'>
    <xs:sequence>
      <xs:element minOccurs='0' name='scheme' type='xs:string' />
      <xs:element minOccurs='0' name='serverName' type='xs:string' />
      <xs:element minOccurs='0' name='serverPort' type='xs:string' />
      <xs:element minOccurs='0' name='contextPath' type='xs:string' />
    </xs:sequence>
  </xs:complexType>
</xs:schema>

```

&lt;/types&gt;

*Listing 7*

## Transmission Optimization

The MTOM (Message Transmission Optimization Mechanism) is a specification created by W3C [REF-4] to support selective encoding of portions of the message and provides a means of efficiently serializing XML infosets that have binary data content (eg. JPEG images, PDF files etc.). At the server (Endpoint) side binary optimization is enabled via the `@BindingType` annotation (refer to Listing 4). Web service clients generally enable MTOM at the Binding Provider (refer to Listing 11).

The Binary data (PDF) being returned is represented by an instance of `javax.activation.DataHandler`. The `DataHandler` class provides a consistent interface to data available in various different formats and can be constructed passing in a byte array. For example refer to Listing 8. The WSDL document generated for this source will contain the basic XML schema `xs:base64Binary` type which the runtime SOAP mechanism will encode it as.

```
@XmlType(name="pdfContent",
    namespace="http://scrs.forms.service",
    propOrder = { "dataHandler" })
public class PDFContent {
    private DataHandler dataHandler;

    public PDFContent() {
    }

    public PDFContent(byte[] byteArrayPDF, String contentType) {
        ByteArrayDataSource ds = new ByteArrayDataSource(byteArrayPDF,
contentType);
        DataHandler dh = new DataHandler(ds);
        setDataHandler(dh);
    }

    // getter and setter
}
```

*Listing 8*

## Handling Exceptions

The client should handle two types of exceptions both of which are translated into SOAP Fault messages by the binding framework: a runtime `SOAPFaultException` or an application level exception. The runtime exception will already have a fault bean generated for it by the framework. However, all unchecked exceptions thrown within the implementation bean or checked exceptions (eg. those that exist as part of the `renderForm` call), are handled via a generic application exception class which was aptly named `FormException` (refer to Listing 9).

This class consists of a constructor which takes in the error category, error code, short error message and a string represented array of the last 5 stack trace elements of the primary throwable cause which would likely make more sense for debugging purposes to the calling system or invoker of the WS. The `FormException` class was decorated with `@WebFault` used to annotate the service specific exception class (see Listing 10) to customize to the local and namespace name of the fault element and the name of the fault bean which eventually gets translated into the Fault message and accompanying schema mapping on the WSDL document.

```
@WebFault(faultBean="forms.server.exception.FormExceptionBean")
public class FormException 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
}

```

Listing 9

```

@XmlRootElement(namespace = "http://scrs.forms.service/", name =
"FormException")
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(namespace = "http://scrs.forms.service/", name = "FormException",
propOrder = {
    "errorCategory",
    "errorCode",
    "message",
    "stackTraceArr"
})
public class FormExceptionBean {
    @XmlElement(namespace = "", name = "errorCategory")
    private String errorCategory;
    @XmlElement(namespace = "", name = "errorCode")
    private int errorCode;
    @XmlElement(namespace = "", name = "message")
    private String message;
    @XmlElement(namespace = "", name = "stackTraceArr")
    private String[] stackTraceArr;

    //getters and setters
}

```

Listing 10

## Security

Security is very important for any service consumable within an enterprise environment. There are essentially 3 options for securing the service:

- Transport level security using Secure Socket Layer technologies that secure the payload at the transport layer
- Optionally, message level security using WS-Security that secures the payload at the SOAP message (application) level
- Authentication and authorization of the client application credentials once the message arrives at its endpoint destination

Due to the sensitive nature of data being transmitted (customer demographics information including Social Security Numbers etc.), SSL technology is used to provide all clients needing access to the Forms services the ability to communicate securely with an encrypted session. Specifying a `transportGuarantee=CONFIDENTIAL` attribute in the `@WebContext` annotation for the endpoint enforces the secure transport requirement for all inbound/outbound messages. The endpoint address in the WSDL document was also changed to use a secure protocol HTTPS which utilizes port 443 by default and the connector to that port was also enabled to point to the keystore containing the CA signed certificate.

The security model implemented consists of basic authentication/authorization wherein access to the SLSB endpoint is secured via the `@SecurityDomain` annotation which specifies the security realm and the `@RolesAllowed` annotation (see Listing 2) which specifies the list of roles permitted to access the endpoint methods. The realm can store identity information in a variety of ways: property files, LDAP directories or databases accessible via JDBC. Authentication at runtime involves the Security Interceptor loading the preconfigured LoginModule for the endpoint annotated realm which it then delegates to verify the credentials of the calling client application. In JBoss, the simplest method is to configure the security context (realm) in `login-config.xml` and use the `UserRolesLoginModule` with property files containing combinations of user/password to authenticate and user/roles to authorize the role allowable access to the endpoint methods.

## Web Service Clients

In order to generate client side artifacts with JBossWS, typically you will use the wsconsume tool to generate the stubs from the WSDL document. This usually gives a bunch of files, one of which is called the Service Implementation class which can be recognised as it will have two public constructors, one with no arguments and one with two arguments, representing the wsdl location (a java.net.URL) and the fully qualified wsdl service name (a javax.xml.namespace.QName) respectively. The servlet, action class or whatever client side component that calls the service would need to get an instance of this class (instantiated with the WSDL document URL and the QName of the service defined in it) and then call the getPort method to return a dynamic proxy which is then used to invoke the target service endpoint similar to Listing 11. Use the javax.xml.ws.BindingProvider to set the username/password combination to authenticate against JBoss security. To simplify portability across various systems that need to use the same client side functionality (as in PAC, EES and Member Access in our case) you would typically package the generated client artifacts (SEI, wrapper classes for request and response messages, service classes) into a separate java archive (in our case called FormsClient.jar) and simply include the jar file into the classpath of the calling system module.

```
public class FormGetAction extends Action {
private FormClientDemographics formClientDemographicsWS ;

public ActionForward execute(..) throws ServletException {
    wsPortSetup();
    try {
        <resultObject> = seiWS.callRenderForm ();
    } catch(FormException) {... }
}

private void wsPortSetup()
{
    QName serviceName = new QName("http://scrs.forms.service/",
"FormClientDemographicsService");
    URL wsdlURL = new URL();
    Service service = Service.create(wsdlURL, serviceName);
    formClientDemographicsWS =
(FormClientDemographics)service.getPort(FormClientDemographics.class);

    BindingProvider bp = (BindingProvider) seiWS;
    SoapBinding binding = (SoapBinding) seiWS.getBinding();
    binding.setMTOMEnabled(true);
    bp.getRequestContext().put(BindingProvider.USERNAME_PROPERTY, "username");
    bp.getRequestContext().put(BindingProvider.PASSWORD_PROPERTY, "password");
}
}
```

*Listing 11*

## Process View

Finally refer to Figure 2 and 3 for the view of the end-to-end architecture which diagrams the major objects (on the various intermediate systems), processes and related activity timelines involved in the system execution and their interactions.

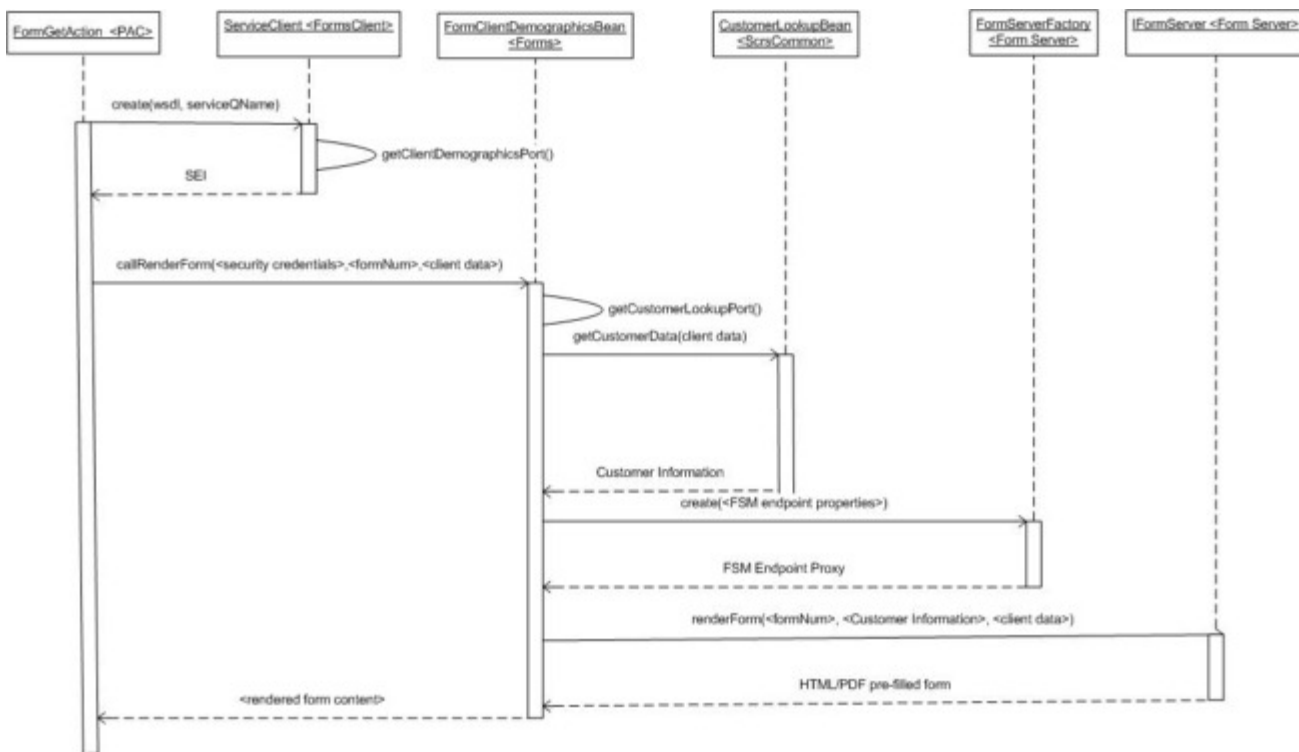


Figure 2: Sequence Diagram - Rendering Adobe LiveCycle Forms

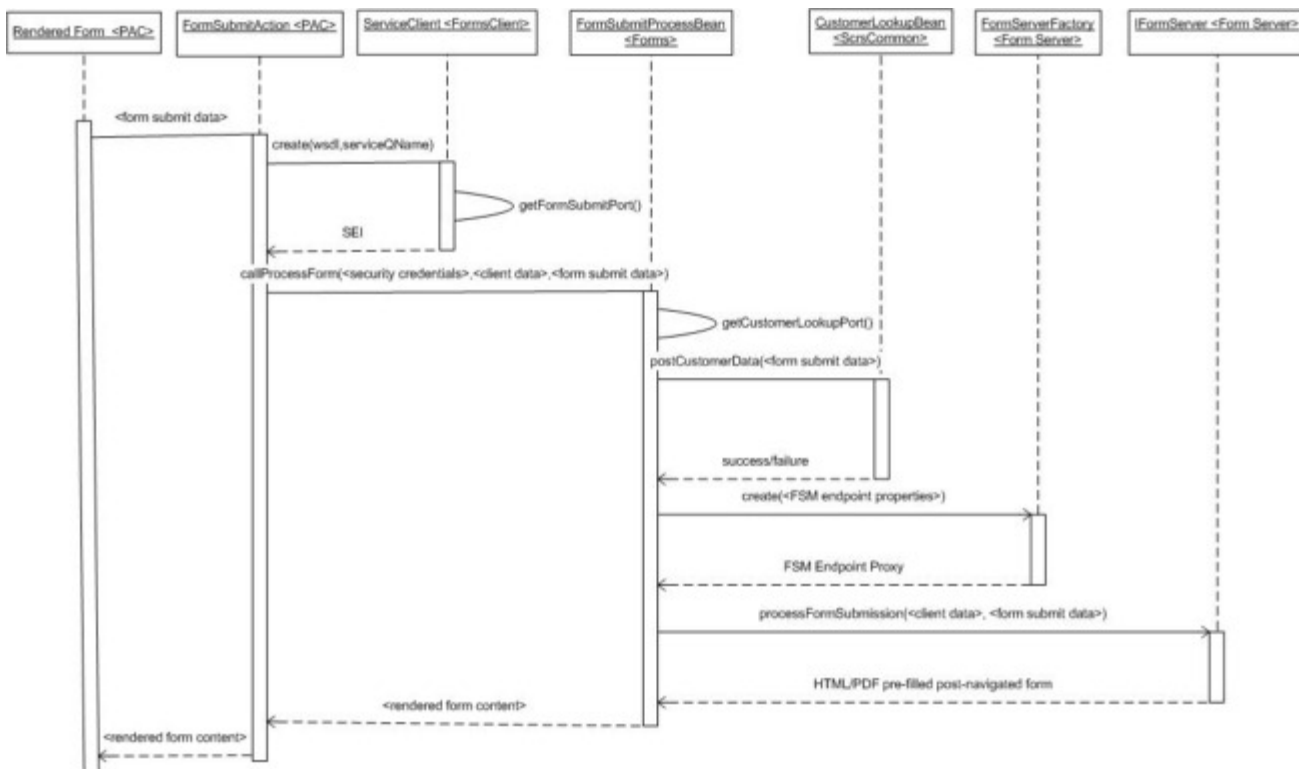


Figure 3: Sequence Diagram - Processing Form Submissions

### Conclusion

Creating a JAX-WS service façade for automated Adobe LiveCycle forms components allows for the linking of forms to multiple processes across different external systems. The result is a leaner, efficient and better integrated process for accessing centralized forms functionality leading to increased data accuracy, reduced manual entry and substantive improvements in service delivery time for the agency. Using a high level framework like JBossWS

obfuscates the complexity and details of converting between Java method invocations and the corresponding SOAP messages, replacing the need for a custom Java/XML data mapping and instead leveraging JAXB which allows mapping for any XML schema. Development and maintenance is greatly simplified with the use of annotations to define Web service endpoints, business methods, web context, security domain and service clients.

## References

[REF-1] JBossWS User Documentation", JBoss.org, <http://jbossws.jboss.org/mediawiki/index.php?title=JBossWS>

[REF-2] "Lean Service Architectures with Java EE 6" by Adam Bien, JavaWorld.com,

<http://www.javaworld.com/javaworld/jw-04-2009/jw-04-lean-soa-with-javasee6.html>

[REF-3] Java SOA Cookbook" by Eben Hewitt, O'Reilly Media Inc., <http://oreilly.com/catalog/9780596520724/>

[REF-4] "SOAP Message Transmission Optimization Mechanism", W3C Recommendation,

<http://www.w3.org/TR/soap12-mtom/>

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 © 2006-2009  
SOA Systems Inc.