

The SOA Magazine

Feature Article



RESTful SOA with Open Source

by Rizwan Ahmed

Published: August 30, 2009 (SOA Magazine Issue XXXI: August 2009)

[Download PDF](#)

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

Abstract: With the exponential growth of the Web, REST as an architectural style [REF-1] has found its niche in the modern services landscape with its popularity poised to grow even further. JAX-RS is a new JCP specification [REF-2] that provides a Java API for RESTful Web services over the HTTP protocol. JAX-RS uses annotations on POJOs (Plain Old Java Objects) to map to the RESTful architectural style of presentation and facilitates lookup of distributed enterprise resources via Uniform Resource Identifiers (URIs). In this article I demonstrate a RESTful service-oriented architecture created to wrapper similar functionality as detailed in [REF-3] to enable a client application render an Adobe form and process form submissions abstracted away into Adobe Form Server Module (FSM) factory objects. Corollary to Service Endpoint implementations in JAX-WS, here we have JAX-RS annotated Web Resource classes (the RESTful term for a service) instantiated per each HTTP request that encapsulate the service functionality and Providers that transform the resource parameters and result content into runtime representations in a variety of media types.

Introduction

In an earlier article [REF-3], I had described the solution for a simple JAX-WS service façade for Adobe LiveCycle Forms functionality, callable from any external application. The objective of doing so was to hide away details of how the form rendering and form processing occurs into the service implementation and all clients (including various enterprise applications) that need access to the functionality can easily lookup and consume the provided services via published contracts (WSDL definitions).

While the WS-* SOAP and RESTful implementation approaches represent different architectural styles, they are fundamentally similar in as much as they both facilitate integration at the application level by having two systems talk to each other using agreed upon namespaces, protocols and formats. The roots of the differences between the two technologies (frequently a topic of vigorous debate within the SOA developer community) is due to the fact that while using SOAP you connect the services by defining a layer of abstraction over the method semantics and addressing model. In REST, the Web itself functions as sort of an "integration bus" with a small number of globally understood methods ("verbs"), protocols and a common addressing scheme using URIs. Aside this, it is worthwhile to point out that the message payloads in both SOAP and RESTful service implementations could be the same XML document with the runtime (un)marshalling delegated to the appropriate infrastructure and Provider mechanism (JAXB is the most common).

Despite the preference of most software vendors for SOAP, REST and SOAP are complementary technologies and can work in tandem as this sample architecture demonstrates, wherein we use a RESTful resource to encapsulate the wrapper to a SOAP based endpoint (the FSM Axis Web service). There are, of course, use cases that support the preference of one over the other but a good SOA practitioner has to make that determination weighing in other external factors.

An Open Source Example

RETEasy [REF-4] is a portable, open source reference implementation of the JAX-RS specification, implemented as a ServletContextListener and a Servlet deployed in a Web Application Archive (WAR) and therefore can run under any servlet container (although tighter integration with the JBoss Application server makes the developer experience easier in that environment). The ResteasyBootstrap listener is responsible for initializing some basic components of RETEasy as well as scanning for annotated classes in your custom deployed Java Application Archive (JAR) file. The HttpServletDispatcher servlet serves as the controller for all HTTP requests sent via the RETEasy framework, routing it to the appropriate Web resource.

A JAX-RS custom application consists of one or more Web resources and zero or more providers (resources and providers are detailed in subsequent sections) and is an aggregation of all the application specific components that make up a RESTful solution architecture. While there are alternate mechanisms such as runtime scanning of classes for locating resource classes and providers, implementing the standard JAX-RS `javax.ws.rs.core.Application` class to register with the RETEasy runtime a list of all JAX-RS resources, providers and exception mappers for your custom application, is the only portable means of configuration. An implementation of the Application class is shown in Example 1. By default, a single instance of the Provider class is instantiated per each JAX-RS application and available at runtime. In contrast, a new instance of the resource class is created for each request to that resource.

```
public class SCRSJAXRSApplication extends Application
{
    HashSet<Object> singletons = new HashSet<Object>();

    public SCRSJAXRSApplication()
    {
        singletons.add(new FormClientDemographicsResource()); // JAX-RS
Resource
        singletons.add(new FormGenericSubmitResource()); // JAX-RS Resource
        singletons.add(new FormRestEasyExceptionHandler()); // JAX-RS Exception-
Mapper Provider
        singletons.add(new FormExceptionHandler()); // JAX-RS
MessageBodyWriter<T> Provider
    }

    @Override
    public Set<Class<?>> getClasses()
    {
        HashSet<Class<?>> set = new HashSet<Class<?>>();
        return set;
    }

    @Override
    public Set<Object> getSingletons()
    {
        return singletons;
    }
}
```

Example 1

JAX-RS Providers

Provider, an essential component of the JAX-RS runtime, is an implementation of the JAX-RS extension interface and provides the means to marshal and unmarshal many different message bodies and formats. For example, media types such as `application/xml`, `application/json`, `application/fastinfoset`, `application/atom` can be transformed into their respective Java representations via RETEasy implemented JAXB providers. Similarly, a media type of `application/x-www-form-urlencoded`, typically consisting of field data posted from an HTML form, is unmarshalled to a `MultiValuedMap` prior to being sent to the resource. RETEasy will select a different Provider based on the return type or parameter type used to define the resource.

The JAX-RS specification also allows you to define your application specific custom media type and plug in your own Provider implementation to perform application specific content (un)marshalling. You create

implementations of the `javax.ws.rs.ext.MessageBodyWriter` and `MessageBodyReader` respectively annotated with `@Provider`. At runtime, the return media type of the Web resource method is matched up to the `@Produces` annotated media type on the Writer (likewise, the resource parameter to the `@Consumes` type on the Reader) and the type of the requested/returned instance (represented as `Entity`) is used to select the correct entity provider implementation. `RETEasy` can be configured via a context parameter switch to automatically scan its `WEB-INF/lib` or classes directories for classes annotated with `@Provider`, or alternatively, you may register the same within an implementation of `javax.ws.rs.core`. Application as in Example 1.

JAX-RS Resource

A JAX-RS Resource is the server side implementation of the business functionality that you would want to be wrapped as a Web service and can be as simple as a POJO annotated with `@Path` (`javax.ws.rs.Path`), the value attribute of which will indicate the path at which the resource will be available. The POJO would also contain various business methods annotated with one of the HTTP methods (`@GET`, `@PUT`, `@POST`, `@DELETE`). These annotations indicate what business method should be invoked when the resource receives a request using that particular HTTP method or verb. The `@Consumes` annotation on the Resource class or method level would indicate the type that the particular resource or resource method expects as a request entity and the `@Produces` would be used to indicate the media type you want to return on the response.

Creating Contract-First Resources

While one could take a code-first approach to building a RESTful resource, my personal preference is to build services contract-first and present the ideal interface to the calling clients. Since there is no WSDL contract to create in REST you can start off from an XML schema representation of the expected parameters and desired result. The implication here is that your Web service will be working with the application/xml media type. This approach is advantageous as it facilitates JAXB schema compiler (XJC) generation of annotated value classes and ObjectFactory [REF-5] for use with the appropriate `RETEasy` JAXB Provider to unmarshal the request and marshal the response. Example 2 shows a schema representation of the input parameters required for the Resource.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema targetNamespace='http://scrs.forms.resteasy.service/'
  version='1.0'
  xmlns:ns1='http://scrs.forms.resteasy.service/'
  xmlns:tns='http://scrs.forms.resteasy.service/'
  xmlns:xsd='http://www.w3.org/2001/XMLSchema'
  xmlns:jxb='http://java.sun.com/xml/ns/jaxb'
  jxb:version='2.0'>

  <xsd:import namespace='http://scrs.forms.resteasy.service' />
  <xsd:element name='callRenderForm'
type='tns:FormClientDemographicsInputType' />

  <xsd:complexType name='FormClientDemographicsInputType'>
    <xsd:sequence>
      <xsd:element minOccurs='0' name='contentURL' type='tns:contentURL' />
      <xsd:element minOccurs='0' name='inputSSN' type='xsd:string' />
      <xsd:element name='maskSSN' type='xsd:boolean' />
      <xsd:element minOccurs='0' name='formNum' type='xsd:string' />
      <xsd:element minOccurs='0' name='userAgent' type='xsd:string' />
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name='contentURL'>
    <xsd:sequence>
      <xsd:element minOccurs='0' name='scheme' type='xsd:string' />
    </xsd:sequence>
  <xsd:element minOccurs='0' name='serverName' type='xsd:string' />
  <xsd:element minOccurs='0' name='serverPort' type='xsd:string' />
  <xsd:element minOccurs='0' name='contextPath' type='xsd:string' />
</xsd:sequence>
</xsd:complexType>
```

```
</xsd:complexType>
</xsd:schema>
```

Example 2

Value classes for the input schema generated by the XJC will contain an `@XmlType` annotation to map Properties and Fields on that class to the schema defined complex type with the `propOrder` element determining the sequence order of the XML elements. The runtime framework will select the `RETEasyJAXBXmlTypeProvider` when the value class is annotated with an `@XmlType` annotation. This provider attempts to locate the XJC generated `ObjectFactory` class (please refer to Example 3) which contains the factory methods to programmatically construct a new instance of the Java representation for the XML content which is then returned wrapped within a `JAXBElement` instance.

```
@XmlRegistry
public class ObjectFactory {

    private final static QName _CallRenderForm_QNAME = new
QName("http://scrs.forms.resteasy.service/", "callRenderForm");

    /**
     * Create a new ObjectFactory that can be used to create new instances of
     schema derived classes for package:
     scrs.forms.service.resteasy.parameters.formrender.input
     *
     */
    public ObjectFactory() {
    }

    /**
     * Create an instance of FormClientDemographicsInputType
     *
     */
    public FormClientDemographicsInputType
createFormClientDemographicsInputType() {
    return new FormClientDemographicsInputType();
    }

    /**
     * Create an instance of ContentURL
     *
     */
    public ContentURL createContentURL() {
    return new ContentURL();
    }

    /**
     * Create an instance of JAXBElement< FormClientDemographicsInputType>
     *
     */
    @XmlElementDecl(namespace = "http://scrs.forms.resteasy.service/", name =
"callRenderForm")
    public JAXBElement createCallRenderForm(FormClientDemographicsInputType
value) {
    return new JAXBElement(_CallRenderForm_QNAME,
FormClientDemographicsInputType.class, null, value);
    }
}
```

Example 3

Alternatively, if using a code-first approach, you may hand code your JAXB value classes and annotate them with the `@XmlRootElement` in which case the `JAXBXmlRootElementProvider` is selected for handling the basic marshalling and unmarshalling of the JAXB entities.

Please refer to Example 4 for a Web resource that returns the pre-populated form design (passed as a parameter value in the `FormClientDemographicsInputType` object). The implementation details of how the form is rendered, via invoking the Adobe FSM Client API `renderForm()` discussed in-depth in the previous article [REF-3], is encapsulated within the resource which simply returns the `FormClientDemographicsOutputType` object containing the rendered form's HTML content as a `String` or `PDF`

content as a binary byte array. The resource path follows the RESTEasy servlet path mapping defined in its web.xml, so you will be able to reach an instance of this class at `://formsRS/renderForm`.

By default, a new instance of the resource class is created for each request to that resource with all requested dependencies being injected at runtime. By using the `@POST` annotation, you indicate that the `callRenderForm` method is the one you need to respond to HTTP POST requests. The `callRenderForm` method also both expects and produces an XML entity as indicated by the `@Consumes` and `@Produces` annotations. The RESTEasy runtime matches up the MIME type in the request and response to the appropriate Provider class (in our case `JAXBXmlTypeProvider` for the `FormClientDemographicsInputType` and `FormClientDemographicsOutputType` entities) to perform the transformation of XML to object representations and vice versa.

```
@Path("formsRS")
public class FormClientDemographicsResource
{
    @POST
    @Path("/renderForm")
    @RolesAllowed("friend")
    @Produces("application/xml")
    @Consumes("application/xml")
    public FormClientDemographicsOutputType
callRenderForm(FormClientDemographicsInputType inputObj) throws
FormRestEasyException
    {
        try{
            // Implementation of the Adobe FSM renderForm() specific logic as
            detailed in [REF-3]
            ..
        } catch (Exception ex) {
            throw new FormRestEasyException(..);
        }
    }
}
```

Example 4

Exception Handling

As with any remote Web service, you would want your RESTful resource to throw exceptions under certain conditions and deal with them appropriately at the client. The simplest way is to create a response with the HTTP error code you want to set, encapsulate it into an instance of a runtime `javax.ws.rs.WebApplicationException` and then have it thrown from the RESTful business method [REF-6]. This class suffers from the primary downside in that it does not allow you to set your own custom error mapping elements. Ideally, you would like to create your own application specific class encapsulating the error message, error code, category and stack trace array of the primary exception cause at the resource end (described as *FormRestEasyException* in Example 5).

JAX-RS allows for the `ExceptionHandler` interface which can be implemented in custom, application provided components that can catch thrown application exceptions and write specific HTTP responses. You annotate the mapper implementation with `@Provider` and register it (refer to Example 1) so that the RESTEasy runtime can pick it up. When a resource throws an exception that the runtime has a mapping for, it uses the mapped exception provider class to generate an instance of `javax.ws.rs.core.Response`, consisting of a HTTP error status code (typically 500) and an entity body upon which you can piggy back details of the specific exception, `FormRestEasyException` as in our case (an example is shown in Example 6). Conversion between the Java object into the entity body then becomes the responsibility of the entity Provider. The exception class could be marshalled by the RESTEasy built-in default JAXB Provider framework if the class was annotated with binding specific `@XmlType` or `@XmlRootElement` annotations.

As an academic exercise, for the sole purpose of demonstration, I have chosen not to take this route instead relying on a custom `MessageBodyWriter` Provider implementation which the runtime will invoke to marshal an instance of the `FormRestEasyException` entity (please refer to Example 7). The response is processed at the client end as if the method in your resource that threw the checked or runtime exception had returned it.

```
public class FormRestEasyException extends Exception
```

```

{
    private int errorCode;
    private String errorCategory;
    private String[] stackTraceArr;

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

    // getters and setters
}

```

Example 5

```

@Provider
public class FormRestEasyExceptionHandler implements ExceptionMapper
{
    public Response toResponse(FormRestEasyException formRestEasyException)
    {
        return Response.status(500)
            .entity(formRestEasyException)
            .build();
    }
}

```

Example 6

```

@Produces("application/xml")
@Provider
public class FormExceptionHandler implements
MessageBodyWriter<FormRestEasyException> {

    public boolean isWritable(Class<?> type, Type genericType,
        Annotation[] annotations, MediaType mediaType) {
        return FormRestEasyException.class.isAssignableFrom(type);
    }

    public void writeTo(FormRestEasyException formRestEasyException, Class<?>
type, Type genericType,
        Annotation[] annotations, MediaType mediaType,
        MultivaluedMap<String, Object> headers,
        OutputStream out) throws IOException {
        out.write("<ns1:exception
xmlns:ns1='http://scrs.forms.resteasy.service/'>".getBytes());
        out.write("<error-code>".getBytes());
        out.write(formRestEasyException.getErrorCode().getBytes());
        out.write("</error-code>".getBytes());
        out.write("<error-message>".getBytes());
        out.write(formRestEasyException.getMessage().getBytes());
        out.write("</error-message>".getBytes());
        ...
        out.write("</ns1:exception>".getBytes());
    }

    public long getSize(FormRestEasyException formRestEasyException,
        java.lang.Class<?> type,
        java.lang.reflect.Type genericType,
        java.lang.annotation.Annotation[] annotations,
        MediaType mediaType) {
        return -1;
    }
}

```

Example 7

An alternate, albeit much simpler, approach to exceptions handling would involve creating a complex type for the application specific exception elements and embedding that as an element within the schema representing the output FormClientDemographicsOutputType object that is sent back from the RESTful resource. No custom Exception Mappers need to be defined here as the exception is simply sent back as part of the return result object.

Security Considerations

The option we have for authentication and authorization of HTTP requests routed to a RESTful resource is via configuring the appropriate security related elements in the deployment descriptor web.xml of the RESTEasy Web application. You need to first add a standard security constraint element to the web.xml defining the Web resource collection, that is, the HTML files, servlets, URL patterns and HTTP methods that must be protected from public access. A user must have the proper authorization to access resources identified and secured under the Web resource collection. Therefore, you define an authorization constraint element which identifies the role (assignable to the user) that is allowed access to the Web resource collection. Please refer to the relevant snippet of the RESTEasy web.xml in Example 8.

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>Resteasy</web-resource-name>
    <description>An example security config that only allows users with
the
    role 'friend' to access the RESTEasy web application
    </description>
    <url-pattern>/formsRS</url-pattern>
    <http-method>GET</http-method>
    <http-method>POST</http-method>
  </web-resource-collection>
  <auth-constraint>
    <role-name>friend</role-name>
  </auth-constraint>
</security-constraint>

<login-config>
  <auth-method>BASIC</auth-method>
  <realm-name>resteasy</realm-name>
</login-config>

<security-role>
  <role-name>friend</role-name>
</security-role>
```

Example 8

A security realm for RESTEasy is then defined by creating a new descriptor jboss-web.xml containing a element referencing the security context defined in login-config.xml and using the UserRolesLoginModule with user/password and user/role combination property files. The client can use HTTP Basic authentication and pass in user credentials in the "Authorization" header of the HTTP request. At the resource end the @RolesAllowed specifies the role(s) allowable access to the Resource methods by performing a runtime injection of HttpServletRequest.isUserInRole(rolename). If one of the @RolesAllowed passes, then the request is allowed, otherwise a response is sent back with a 401(Unauthorized) response code.

Additionally, you may use SSL technology to provide all clients needing access to the JAX-RS forms resource the ability to communicate securely with an encrypted session. The URL address needs to use a secure protocol HTTPS which utilizes port 443 by default and the connector to that port needs to be enabled to point to the keystore containing the SSL digital certificate.

Developing RESTful Clients for SOA

RESTEasy has a client proxy framework that allows you to use JAX-RS annotations to invoke on a remote HTTP resource. You need to create a Java class proxy that is a mirror interface of the resource class. You

will next need to programmatically create an instance of this client class from a `org.resteasy.plugins.client.httpclient.ProxyFactory` instance which builds an HTTP request that it uses to invoke the remote RESTful Web service. While this is certainly an elegant way to create RESTful clients, I did not take this approach as it would have necessitated that we bundle the RESTEasy library JARs with the calling applications. Since all the resource class expects is POX (Plain Old XML) over HTTP there are other simpler ways to construct it without the RESTEasy client framework.

Having taken a contract-first approach at constructing the server resource, it was decided to take a similar approach to develop the client side components. Since the ideal "contract" (this word is a misnomer as RESTful implementations do not work off WSDL documents) was already available in the form of XML schemas which defined the resource input and output entity structures, it was deemed straightforward to simply generate value classes using XJC (the JAXB schema compiler) containing client side bindings and JAXB representations of the types defined in the schema. Usually hidden in the midst of the generated classes is the `ObjectFactory` class which provides an easy way to construct Java representations of XML content [REF-5] (also refer back to Example 3). A simple approach for marshalling the XML payload that needs to be sent to the Web resource would use the schema-derived classes, the `ObjectFactory`, a `JAXBContext` object and the call to marshal the content on the `OutputStream` (refer to Example 9).

Similarly, unmarshalling an XML document, typically the response from the JAX-RS Web resource, consists of creating a `JAXBContext` object and the call to unmarshal the document (refer to Example 10). The `JAXBContext` object provides the entry point to the JAXB API and maintains the binding information between XML and Java. One way of creating a context instance is by calling the static method `newInstance` with the package name containing the JAXB schema-derived classes. From this context, an `Unmarshaller` or `Marshaller` object is obtained, which functions as the driver for processing XML to create the equivalent set of Java objects and vice versa.

To simplify portability across various systems that need to use the same client side functionality you would typically package the generated client artifacts (JAXB schema-derived classes, marshaller and unmarshaller utilities) into a separate java application archive (in our case called `FormsRSClient.jar`) and simply include the file into the classpath of the calling system module.

Lastly, the client calling system or application module's servlet or action class needs to create a URL that points to the JAX-RS server resource and open a `java.net.HttpURLConnection` to that location (please refer to the code segment in Example 9). The HTTP protocol has built in content negotiation headers that allow the client and server to specify what content they are transferring and what content they would prefer to get. We saw earlier how the JAX-RS server resource declares content preferences via the `@Produces` and `@Consumes` headers. Based upon the "Content-Type" header sent by the HTTP client, the JAX-RS runtime at the server end would then match up to the correct resource method (annotated with `@Consumes`) and select the appropriate `Provider` to unmarshal the request. Likewise, the "Accept" header would be matched up with the method annotated with `@Produces` and the appropriate `Provider` would marshal the response. Likewise, authentication credentials from the client application would need to be sent in the "Authorization" header which would include the Base64 encoded value of the "user-name:password" string.

```
public class FormRenderClientInput {
    private ObjectFactory of;
    private FormClientDemographicsInputType formClientDemographicsInputType;
    private ContentURL contentURL;

    public FormRenderInput() {
        of = new ObjectFactory();
        formClientDemographicsInputType =
of.createFormClientDemographicsInputType();
        contentURL = of.createContentURL();
    }

    public void makeContentURL( String scheme, String serverName, String
serverPort, String contextPath ) {
        contentURL.setScheme(scheme);
        contentURL.setServerName(serverName);
        contentURL.setServerPort(serverPort);
    }
}
```

```

        contentURL.setContextPath(contextPath);
    }

    public void make( String inputSSN, boolean maskSSN, String formNum, String
    userAgent ) {
        formClientDemographicsInputType.setContentURL(contentURL);
        formClientDemographicsInputType.setInputSSN(inputSSN);
        formClientDemographicsInputType.setMaskSSN(maskSSN);
        formClientDemographicsInputType.setFormNum(formNum);
        formClientDemographicsInputType.setUserAgent(userAgent);
    }

    public void marshal(OutputStream out) {
        try {
            JAXBElement jel =
of.createCallRenderForm(formClientDemographicsInputType);
            JAXBContext jc = JAXBContext.newInstance(
"scrs.forms.client.resteasy.parameters.formrender.input" );
            Marshaller m = jc.createMarshaller();
            m.marshal( jel, out );
        } catch( JAXBException jbe )
        {
            jbe.printStackTrace();
        }
    }
}

```

Example 9

```

public class FormRenderClientOutput {

    public FormClientDemographicsOutputType unmarshal( InputStream inputStream
    ) throws JAXBException {
        JAXBContext jc = JAXBContext.newInstance(
"scrs.forms.client.resteasy.parameters.formrender.output" );
        Unmarshaller u = jc.createUnmarshaller();
        JAXBElement<FormClientDemographicsOutputType> doc =
(JAXBElement<FormClientDemographicsOutputType>)u.unmarshal( inputStream );
        return doc.getValue();
    }
}

```

Example 10

```

public class FormRestEasyGetAction extends Action {
    ...
    URL postURL = new URL("https://<machine>:<port>/<RESTEasy servlet
mapping>/formsRS/renderForm");

    HttpURLConnection connection = (HttpURLConnection)
postURL.openConnection();
    connection.setDoOutput(true);
    connection.setInstanceFollowRedirects(false);
    connection.setRequestMethod("POST");
    connection.setRequestProperty("Content-Type", "application/xml");
    connection.setRequestProperty("Accept", "application/xml");
    String authorization = "<user-name>:<password>";
    String encodedAuthorization= new
String(Base64.encodeBase64(authorization.getBytes()));
    connection.setRequestProperty("Authorization", "Basic " +
encodedAuthorization);

    OutputStream os = connection.getOutputStream();

    FormRenderInput formInput = new FormRenderInput();
    formInput.makeContentURL(request.getScheme(), request.getServerName(),
request.getServerPort()+"", request.getContextPath());
    formInput.make(inputSSN, maskSSN, sFormQuery, request.getHeader("User-
Agent"));
    formInput.marshal(os);
}

```

```

FormRenderOutput formOutput = new FormRenderOutput();
try {
    formClientDemographicsOutputType =
formOutput.unmarshal(connection.getInputStream());
} catch (Exception ex) {
    throw new IOException(ex.getMessage());
}

formContent = formClientDemographicsOutputType.getFormContent();
...
}
    
```

Example 11

Sequence Diagrams

Finally, Figures 1 and 2 illustrate the sequence diagrams of the RESTful solution architecture created for integrating our external systems with Adobe LiveCycle Forms (for further details about the existing applications landscape please refer to [REF-3]).

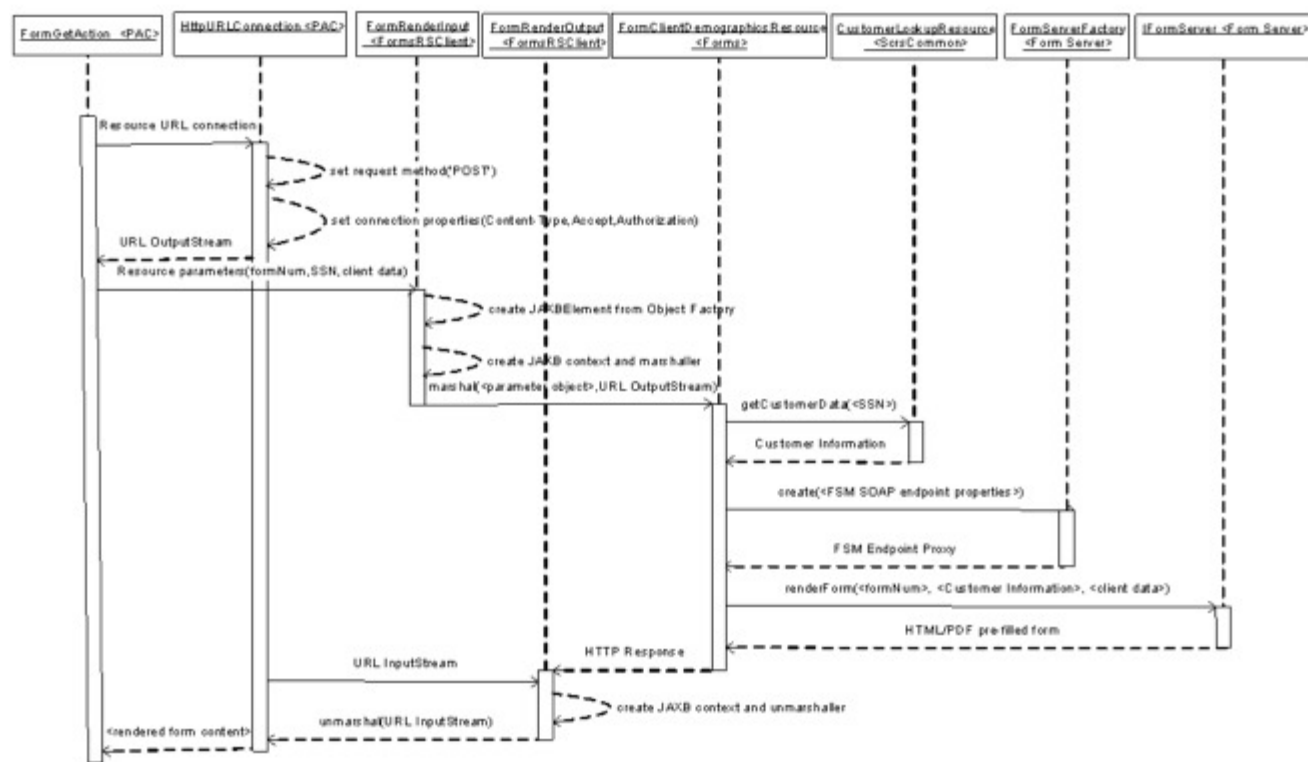


Figure 1: RESTful Implementation for Rendering Adobe LiveCycle Forms

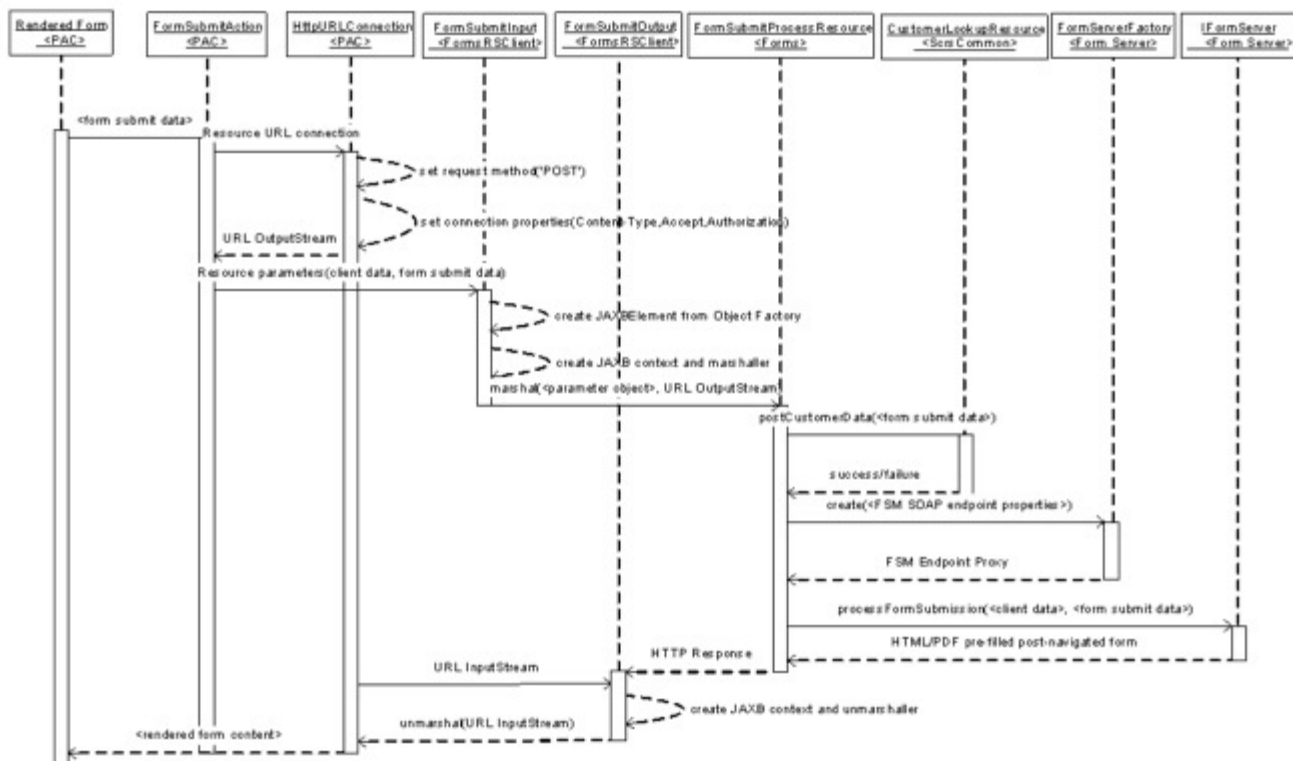


Figure 2: RESTful Implementation for Processing Form Submissions

Conclusion

The sheer variety of Web based devices and tools, pretty much anything that can access and be accessed from the Web (via its common addressing scheme for identification of resources using URIs) has resulted in HTTP increasingly becoming a popular application protocol of choice. The availability of the new JAX-RS specification and reference frameworks (JSR 311 implemented by RESTEasy) has made implementing RESTful architectures convenient and developer-friendly. RESTEasy standardizes the use of annotations to define Web resources which can have multiple runtime representation in a variety of different media types with additional convenience features such as support for mapping Exceptions, enforcing security and an easy to use client API. All transformations and infrastructure plumbing to read/write resource content are facilitated by RESTEasy supplied default and a JAX-RS supported extensible Provider mechanism.

References

- [REF-1] "Architectural Styles and the Design of Network-based Software Architectures" by Roy Thomas Fielding, Doctoral dissertation, University of California, Irvine, <http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>
- [REF-2] "JAX-RS: Java API for RESTful Web Services", Java.net, <https://jsr311.dev.java.net/drafts/spec20090313.pdf>
- [REF-3] "An SOA Case Study: Integrating Adobe LiveCycle Forms using JBossWS" by Rizwan Ahmed, SOA Magazine July 2009, <http://www.soamag.com/I30/0709-3.asp>
- [REF-4] "RESTEasy User Documentation", JBoss.org, <http://jboss.org/reteasy/docs.html>
- [REF-5] "Unofficial JAXB User Guide", Java.net, <https://jaxb.dev.java.net/guide/>
- [REF-6] "Java SOA Cookbook" by Eben Hewitt, O'Reilly Media Inc., <http://oreilly.com/catalog/9780596520724/>