



Web Service Technology Stack Case Study: Nu Skin Enterprises

Todd Tomkinson
Software Engineer/Consultant
Software Technology Group

The Engagement

- Online file storage and management application
 - Consumer focused
 - Browser-based, rich Internet application
- Interfaces with several third-party partners
 - File manipulation (picture and document editing)
 - Product creation and fulfillment (custom calendars, t-shirts, greeting cards, etc.)
 - Authentication and authorization

Technology Constraints

- Java shop
- Database access managed by client team via stored procedures
- User profile and subscription information stored across many systems
 - SAP
 - Custom enterprise databases
 - New application database
- No established web services architecture

Web Service Needs

- Partner APIs
 - Applications written in several different languages (primarily Python and PHP)
 - Interoperability the highest priority
- Application services accessed through AJAX calls from the browser

Application Architecture

- Google Web Toolkit (GWT) 1.5 (just for RIA)
- Spring 2.5
 - JDBC Support
 - Transaction Management
 - Spring MVC
- Spring Security 2.0 (form-based authentication for users, HTTP Basic authentication for API)
- Jersey (JAX-RS RI) 0.8 with Spring integration

JAX-RS (JSR-311) Overview

- From the original Java Specification Request:
 - “This JSR will aim to provide a high level easy-to use API for developers to write RESTful web services independent of the underlying technology and will allow these services to run on top of the Java EE or the Java SE platforms.”
- Essentially a set of annotations that can be used to create a “service” out of any Java object

Why JAX-RS?

- Are you crazy? It isn't even approved?
 - Final Approval Ballot—September 22nd
- Four implementations keeping up with the spec
 - Jersey (Sun)
 - RESTEasy (JBoss)
 - Apache CXF
 - Restlet
- Tired of the RoRs guys having all of the fun

Why Jersey?

- Good Spring integration (configure resource classes in Spring)
- Excellent XML and JSON support (increases interoperability with some partners)
 - Automatically serializes JAXB objects returned from resource methods
 - Uses BadgerFish to automatically translate JAXB objects to JSON
- Lightweight (only two jars with Spring integration)
- Automatic WADL generation (easy way to describe services to partners)

Our Approach

- Prefer contract first services development
- Define REST resources
- Define an XML schema with an element representation for each resource
- Use JAXB to generate Java classes
- Created resource classes which accept and return generated resource classes
- Annotate resource classes with appropriate JAX-RS and Spring annotations

Example: web.xml

```
<servlet>
  <servlet-name>api</servlet-name>
  <servlet-class>com.sun.jersey.spi.spring.container.servlet.SpringServlet</servlet-class>
  <init-param>
    <param-name>com.sun.jersey.config.property.packages</param-name>
    <param-value>my.app.providers;my.app.resources</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
  <servlet-name>api</servlet-name>
  <url-pattern>/api/*</url-pattern>
</servlet-mapping>
```

Example: Schema Definition

...

```
<element name="user">
  <complexType>
    <attribute name="firstName" type="string" use="required"></attribute>
    <attribute name="lastName" type="string" use="required"></attribute>
    <attribute name="email" type="string" use="required"></attribute>
    <attribute name="accountId" type="string" use="required"></attribute>
    <attribute name="priceType">
      <simpleType>
        <restriction base="string">
          <enumeration value="retail"></enumeration>
          <enumeration value="wholesale"></enumeration>
        </restriction>
      </simpleType>
    </attribute>
    <attribute name="active" type="boolean"></attribute>
  </complexType>
</element>
```

...

Example: UsersResource class

```
@Path("v20081101/users")
@PerRequest //jersey specific annotation :(
public class UsersResource {
    private UserResource userResource;

    public void setUserResource(String userResource) {
        this.userResource = userResource;
    }

    @GET
    @Path("{userId}")
    public User getUser(@PathParam("userId") String userId) {
        userResource.setUserId(userId);
        return userResource;
    }
}
```

Example: UserResource class

```
public class UserResource {
    private String userId;

    //this would be set by the UsersResource class
    public void setUserId(String userId) {
        this.userId = userId;
    }

    @GET
    @ProduceMime({"application/xml", "application/json"})
    public User get() {
        ...
        return user;
    }
}
```

Example: Spring context

```
<bean id="usersResource" class="my.app.resources.UsersResource" scope="prototype">  
  ... dependencies  
  <property name="userResource" ref="userResource"/>  
</bean>  
<bean id="userResource" class="my.app.resources.UserResource" scope="prototype">  
  ... dependencies  
</bean>
```

Our Experience

- Partner integrations have been very smooth
 - Can demonstrate the API by using simple utilities like curl and wget
 - Most partners don't need a “client library”--they use an HTTP client and XML/JSON parser (if they really want to) to get just the information they need
- Required minimal infrastructure changes (mostly to allow Spring Security to handle Basic authentication)
- Learning curve very small

Our Experience (cont.)

- JAX-RS is not that portable
 - JAX-RS itself is just annotations, real power lies in the providers
 - We are using a few Jersey specific features
 - Would probably have to refactor code to deploy on another JAX-RS provider

Moving Forward

- Want to look at REST support in Spring 3.0 (may be a better fit in our Spring MVC application)
- Working with the enterprise architect to see how JAX-RS and REST in general fit into the overall web services strategy