



credera
THE POWER OF PERSPECTIVE



Dallas Spring User Group
*Using Web Service Frameworks in a Spring
Application*
Dallas, TX - June 2009

About Me

John Jacobs

- Architect with Credera
- 10+ years of development experience
- Mostly in the Java/web space
 - EJB2
 - Spring
 - Seam/EJB3
- NOT a professional speaker...
- Software World View: Quality, Simplicity
- Dad, skier, “golfer”, all-around nice guy!
- <http://twitter.com/john99jacobs>

About Credera

- Full-service business and technology consulting firm
- Professional Services
 - Technology Solutions (Java, Microsoft, iPhone/mobile, much more)
 - Management Consulting
 - Business Intelligence
- Our People
 - They Rock!!!

Agenda

- Introduction
- Contract-first v. code-first web services
- JAX-WS overview
- Apache CXF
- Spring Web Services
- Other frameworks
- Spring integration
- Enough already...show me some code!!!!
- Questions and Answers

How this Talk Came to Be

- Preparing for a new client project
 - Spring
 - Needs to make heavy use of web service integration
- [Gee wiz, there's a lot of frameworks out there...](#)
 - Apache Axis, Axis2
 - Apache CXF
 - Sun Metro
 - Spring Web Services
 - JBoss Web Services
 - more
- But which one is best for my project?

What This Talk Is NOT

- A Web Service tutorial
- A SOAP tutorial
- A WS-* tutorial
- Political
- RESTful
- Today we will focus on SOAP-based services

However...

Let me Digress for a Moment on REST...

- So much has been written about REST v. SOAP
- REST is trendy - twitter, Flickr, del.icio.us - they all have RESTful APIs
- In the Enterprise SOAP is still king, why?
- REST may seem simpler than SOAP because it's familiar technology
 - HTTP

Have you ever written one of these?

```
public class FooServlet extends HttpServlet {  
  
    @Override  
    protected void doGet (HttpServletRequest req,  
        HttpServletResponse resp) throws ServletException,  
        IOException {  
        // return whatever data is being requested  
        PrintWriter out = resp.getWriter();  
        out.println("someResponse");  
    }  
  
    @Override  
    protected void doPost (HttpServletRequest req,  
        HttpServletResponse resp) throws ServletException,  
        IOException {  
        // do something with the data that got posted  
        PrintWriter out = resp.getWriter();  
        out.println("someOtherResponse");  
    }  
}
```

Then you've basically written a RESTful endpoint!



More REST

- REST and SOAP both have their places
- Brand new service? Lots of CRUD operations? Trendy?
 - REST sounds like a great choice
- Need a rigid contract? Want built in validation and to take advantage of strong typing?
 - SOAP may be just the thing for you
- Daring? Why not try both?
- Today we will focus on SOAP

Why Use a Web Service Framework?

- Why do you use Spring in the first place?
 - Correct Answer = Reduce Complexity
- Let the framework do the mundane stuff
 - SOAP - it's a complex beast
 - WS-*
 - XML Binding (JAX-B)
 - Marshalling
 - Unmarshalling
- Less code = less bugs
- You have enough to worry about
 - Back end integration
 - Design a solid API
 - Make sure your clients can use it
 - Talk to business people

Contract-First v. Code-First Web Services

Contract First

- Write the XSD schemas and WSDL contract first
- Generate or hand-code the Service Endpoint Interface (SEI) and implementation class
- It's all about the XML!
- Java is just an implementation detail
- The folks at SpringSource are big fans
- JAX-WS can do it too
- Contract-first is the only option if you choose Spring WS

Why Contract First?

- You have an existing API that you MUST conform to
 - Legacy
 - Contractual obligation
- Loose coupling between WS API and implementation
- Changes to the contract are tightly controlled
 - Mismatch will result in an Exception
 - Can be easily validated with a unit test
- Mismatch between XSD schemas and Objects
 - Map
 - Cyclical references

Code First

- Write the Java SEI and implementation class first
- Configure as a Web Service
 - Typically using a set of annotations and/or XML
 - Specify the endpoint
- Generate the WDSL from the implementation class
 - May be done at runtime by the application server
 - You can also do it with a command line tool

Why Code First?

- Reduces complexity
 - Less source code to maintain
 - No XML/XSD programming
 - No WSDL design/coding
- Rapid
- Defining a new service
- You get to specify the contract

So Which one is the Right Choice?

- It depends...
 - Are you writing a new service or converting an old one?
 - Do you have a pre-defined contract?
 - Is versioning a requirement?
 - How many consumers do you have?
 - Internal or External service?
- Many frameworks support both
 - Exception - Spring Web Services
 - JAX-WS lends itself nicely to code-first
- Possible strategy: prototype/design with code-first, get a basic WSDL, then cut over to contract-first

Apache CXF

CXF Overview

- A combination of two open source projects: Celtix (IONA) and XFire (Codehaus)
- Implements JAX-WS, among other things
- Has native APIs that can be used if JAX-WS compliance is not required or to access advanced features
 - Clear separation of programming interface (JAX-WS) from underlying implementation (think Hibernate and JPA)
- First class support for Spring
 - All CXF XML configuration files are Spring bean definitions!
- Supports both contract-first and code-first strategies

A Few Words about JAX-WS

- Java API For XML-Based Web Services (JSR 224)
- A standards-based approach for defining Web Services in Java
- Replaces JAX-RPC
- Uses JAXB for XML data binding
- Annotations-based (requires JDK 1.5+)
- Sun provides a reference implementation under project Glassfish
- There are many other implementations, including CXF

Writing a Code First Service with CXF

- Create the annotated Service Endpoint Interface (optional)

```
@WebService
```

```
public interface UserService {
```

```
    @WebMethod
```

```
        public User readUserByEmployeeId(@WebParam(name =  
        "employeeId")String employeeId);
```

```
}
```

- PerJAX-WS, SEI is not required, but it's a good practice
- `@WebService` - tells CXF that this is an SEI
- `@WebParam` - names the parameter in the generated WSDL

Writing a Code First Service with CXF (continued)

- Create the annotated implementation class

```
@WebService(endpointInterface = "com.credera.example.UserService",  
            serviceName="UserService",  
            portName="UserServicePort")
```

```
public class UserServiceImpl implements UserService {  
    @Override  
    public User readUserByEmployeeId(String employeeId) {  
        .....  
        return user;  
    }  
}
```

- serviceName - name of the wsdl:service element
- portName - name of the wsdl:port element

Writing a Code First Service with CXF (continued)

- Add the CXFServlet to your web.xml

```
<servlet>
    <servlet-name>CXFServlet</servlet-name>
    <display-name>CXF Servlet</display-name>
    <servlet-class>
org.apache.cxf.transport.servlet.CXFServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>

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

Writing a Code First Service with CXF (continued)

- Import CXF-specific resources in your application's root ApplicationContext beans definition file (typically applicationContext.xml)

```
<import resource="classpath:META-INF/cxf/cxf.xml" />
```

```
<import resource="classpath:META-INF/cxf/cxf-extension-soap.xml" />
```

```
<import resource="classpath:META-INF/cxf/cxf-servlet.xml" />
```

- Note Spring's classpath protocol
- These resources come from the CXF JAR file
- This along with the servlet and servlet mapping bootstraps CXF in the Spring container

Writing a Code First Service with CXF (continued)

- Endpoints are configured in the context file cxf-servlet.xml

```
<jaxws:endpoint id="userServiceEndpoint"  
  implementor="#userService" address="/UserService" />
```

- Hash notation on the implementor attribute creates a reference to a bean
- Implementor attribute can also refer to a class

```
<jaxws:endpoint id="userServiceEndpoint" implementor="  
  com.credera.example.UserServiceImpl" address="/UserService" />
```

- address is relative to the context root

Writing a Code First Service with CXF (continued)

- And a very nice WSDL is published when you start Tomcat

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions name="UserService" targetNamespace="http://example.credera.com/"
  xmlns:ns1="http://cxf.apache.org/bindings/xformat"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:tns="http://example.credera.com/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <wsdl:types>
    <xs:schema attributeFormDefault="unqualified" elementFormDefault="unqualified"
      targetNamespace="http://example.credera.com/" xmlns="http://example.credera.com/"
      xmlns:xs="http://www.w3.org/2001/XMLSchema">
      <xs:complexType name="user">
        <xs:sequence>
          <xs:element name="active" type="xs:boolean"/>
          <xs:element minOccurs="0" name="employeeId" type="xs:string"/>
          <xs:element minOccurs="0" name="lastLoginDate" type="xs:dateTime"/>
          <xs:element minOccurs="0" name="userName" type="xs:string"/>
        </xs:sequence>
      </xs:complexType>
    </xs:schema>
  </wsdl:types>
</wsdl:definitions>
```

There's more but you get the idea...



Writing a Contract-First Service with CXF

- CXF also supports contract-first using JAX-WS and the wsdl2java utility
- Write the WSDL by hand (or use a tool to help)
- Run CXF wsdl2java command to generate the plumbing and skeleton implementation classes
 - Can also be run via Maven
- Copy the generated classes into your project
- Implement business logic in the endpoint implementation class

Writing a Contract-First Service with CXF (continued)

- Write the WSDL

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:sch="http://www.credera.com/user/schemas"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://www.credera.com/user/definitions"
  targetNamespace="http://www.credera.com/user/definitions">
  <wsdl:types>
    <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
      xmlns="http://www.credera.com/user/schemas"
      elementFormDefault="qualified"
      targetNamespace="http://www.credera.com/user/schemas">
      <xs:complexType name="user">
        <xs:sequence>
          <xs:element name="active" type="xs:boolean"/>

```

. . . . **(truncated)**

Writing a Contract-First Service with CXF (continued)

- Use `wSDL2java` utility to generate the SEI, implementation class and JAX-B mappings

```
C:\dev\apache-cxf-2.2\bin>wSDL2java -impl -ant -d temp  
user.wsdl
```

- Command line options
 - `-impl`: generates shell implementation classes
 - `-ant`: creates an ant `build.xml` for re-generating the code
 - `-d <dir>`: directs output to directory `<dir>`
- Package names are generated based on the target namespace and schema namespace defined in your WSDL

Writing a Contract-First Service with CXF (continued)

- SEI is named `portTypeName.java`
- Endpoint implementation class is named `portTypeNameImpl.java`
- Schema objects are generated with JAX-B annotations
- Write business logic in the endpoint impl class
- Return an instance of the generated response object, representing the XML response payload
- Use `ObjectFactory` to create instances of schema objects

Writing a Contract-First Service with CXF (continued)

```
@javax.jws.WebService(serviceName = "UserServiceService", portName = "UserServiceSoap11",
    targetNamespace = "http://www.credera.com/user/definitions",
    endpointInterface = "com.credera.user.definitions.UserService")

public class UserServiceImpl implements UserService {

    public com.credera.user.schemas.ReadUserByEmployeeIdResponse
    readUserByEmployeeId(com.credera.user.schemas.ReadUserByEmployeeIdRequest
    readUserByEmployeeIdRequest) {

        ObjectFactory factory = new ObjectFactory();

        User user = factory.createUser();

        user.setActive(true);

        user.setLastLoginDate(XMLGregorianCalendarImpl.createDateTime(2009, 03, 14, 10, 46,
30));

        user.setEmployeeId(readUserByEmployeeIdRequest.getEmployeeId());

        user.setUsername("bob");

        ReadUserByEmployeeIdResponse retval = factory.createReadUserByEmployeeIdResponse();

        retval.setUser(user);

        return retval;

    }

}
```

Securing CXF Web Services

- CXF has built in support for WS-Security
 - Authentication - is the client authorized to use the service?
 - Digital Signatures - is the client (and server) who he says he is?
 - Encryption/Decryption - protect sensitive information
- WS-Security is Application layer, use SSL for transport layer security
- Uses Apache WSS4J under the covers
- Configured using WSS4JInInterceptor on the endpoint bean
- WS-Security policies are implemented using a set of callback handlers

Securing CXF Web Services (continued)

- This configuration adds a user name token to the SOAP header and tells WSS4J that the password is in plain text

```
<jaxws:inInterceptors>
  <bean class="org.apache.cxf.ws.security.wss4j.WSS4JInInterceptor">
    <constructor-arg>
      <map>
        <entry key="action" value="UsernameToken"/>
        <entry key="passwordType" value="PasswordText"/>
        <entry key="passwordCallbackRef">
          <ref bean="myPasswordCallback"/>
        </entry>
      </map>
    </constructor-arg>
  </bean>
</jaxws:inInterceptors>
</jaxws:endpoint>
```

Turn on different security policies and options by setting properties in this map – per WSS4J documentation

Securing CXF Web Services (continued)

- Implement a callback handler containing application logic that authenticates the user

```
public class PasswordCallback implements CallbackHandler {  
    @Override  
    public void handle(Callback[] callbacks) throws IOException,  
        UnsupportedCallbackException {  
        WSPasswordCallback pc = (WSPasswordCallback) callbacks[0];  
        String identifier = pc.getIdentifier();  
        // call a business service to retrieve the password for this  
        identifier  
        // compare to the password from the request - this only works for  
        // plain text passwords!!  
        if (!"password".equals(pc.getPassword())) {  
            throw new IOException("wrong password");  
        }  
    }  
}
```

Securing CXF Web Services (continued)

- Obviously this example is contrived
- You don't want to transmit clear text passwords, for hashed passwords use

```
<entry key="passwordType" value="PasswordDigest"/>
```

- And of course store your passwords securely within your application

CXF Summary

Strengths

- Contract First and Code First
- First class support for Spring
- JAX-WS implementation
- Support for WS-Security
- Minimal configuration to run in the Spring container
- Java-centric, no XML programming required except to access advanced features
- Can incorporate other Spring features like AOP

Weaknesses

- Contract First method requires code generation
- WSDL generated using Code First may not be the most efficient or readable
- WSS4J is only WS-Security implementation available

Spring Web Services

Spring Web Services Overview

- Brought to you by SpringSource
- Must use a contract-first approach
- Data-driven, focus is on the data being sent
 - Spring WS will generate WSDL based on an XSD schema
 - This is the preferred method
- Process the incoming XML using your choice of handlers (DOM, SAX, JDOM, DOM4J)
- Also supports several marshalling technologies (JAXB, Castor, XMLBeans, others)

Spring Web Services Overview (continued)

- Write the XSD schema
- Write the endpoint implementation class
 - Spring provides many abstract endpoint classes
 - Transform XML request into Java objects
 - Call an internal business service to satisfy the request
 - Map the response back into XML
- Wire it together with Spring configurations
 - Define the endpoint bean
 - Associate the incoming message to the endpoint class
 - Configure and publish the WSDL

Create XSD Schema

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns="http://www.credera.com/user/schemas"
  elementFormDefault="qualified"
  targetNamespace="http://www.credera.com/user/schemas">
  <xs:complexType name="user">
  <xs:sequence>
    <xs:element name="active" type="xs:boolean" />
    <xs:element name="employeeId" type="xs:string" />
    <xs:element name="lastLoginDate" type="xs:dateTime" />
    <xs:element name="userName" type="xs:string" />
  </xs:sequence>
</xs:complexType>
```

Create XSD Schema (continued)

```
<xs:element name="readUserByEmployeeIdRequest"
  type="readUserByEmployeeIdRequest" />

<xs:complexType name="readUserByEmployeeIdRequest">
  <xs:all>
    <xs:element name="employeeId" type="xs:string" />
  </xs:all>
</xs:complexType>

<xs:element name="readUserByEmployeeIdResponse"
  type="readUserByEmployeeIdResponse" />

<xs:complexType name="readUserByEmployeeIdResponse">
  <xs:all>
    <xs:element name="user" type="user" />
  </xs:all>
</xs:complexType>
</xs:schema>
```

*Xs:elements will
be converted
into messages
and WSDL
operations -
convention over
configuration*

Create Endpoint Implementation - JDOM

```
public class UserEndpoint extends AbstractJDomPayloadEndpoint {  
    private XPath employeeIdExpression;  
  
    // Use constructor to read the request XML  
    public UserEndpoint() throws JDOMException {  
        Namespace namespace = Namespace.getNamespace("user",  
            "http://www.credera.com/user/schemas");  
        employeeIdExpression =  
XPath.newInstance("//user:employeeId");  
        employeeIdExpression.addNamespace(namespace);  
    }  
}
```

Create Endpoint Implementation - JDOM (continued)

```
// Callback method - business logic, create response XML

    protected Element invokeInternal(Element userRequest) throws Exception {

        String employeeId = employeeIdExpression.valueOf(userRequest);

        Namespace namespace =
Namespace.getNamespace("http://www.credera.com/user/schemas");

        Element rootElement = new
Element("readUserByEmployeeIdResponse").setNamespace(namespace);

        Element userElement = new Element("user").setNamespace(namespace);

        userElement.addContent(new
Element("active").addContent("true").setNamespace(namespace));

        userElement.addContent(new
Element("employeeId").addContent(employeeId).setNamespace(namespace));

        userElement.addContent(new Element("lastLoginDate").addContent("2009-
02-14T12:00:00-05:00").setNamespace(namespace));

        userElement.addContent(new
Element("userName").addContent("user01").setNamespace(namespace));

        return rootElement.addContent(userElement);

    }
}
```

Add spring-ws Servlet to web.xml

```
<servlet>  
    <servlet-name>spring-ws</servlet-name>  
    <servlet-  
class>org.springframework.ws.transport.http.MessageDisp  
atcherServlet</servlet-class>  
</servlet>
```

```
<servlet-mapping>  
    <servlet-name>spring-ws</servlet-name>  
    <url-pattern>/*</url-pattern>  
</servlet-mapping>
```

- Default configuration file is spring-ws-servlet.xml

Configure Spring Beans in spring-ws-servlet.xml

- Endpoint bean configuration

```
<bean id="userEndpoint"  
      class="com.credera.example.UserEndpoint" />  
  
<bean  
      class="org.springframework.ws.server.endpoint.mapping.PayloadRootQNameEndpointMapping">  
  
  <property name="mappings">  
  
    <props>  
  
      <prop  
        key="{http://www.credera.com/user/schemas}readUserByEmployeeIdRequest">userEndpoint</prop>  
  
    </props>  
  
  </property>  
  
</bean>
```

Configure Spring Beans in spring-ws-servlet.xml (continued)

- Tell Spring how to publish the WSDL

```
<bean id="user"  
  class="org.springframework.ws.wsdl.wsdl11.DefaultWsdl11Definition">  
  <property name="schema" ref="userSchema"/>  
  <property name="portTypeName" value="UserService"/>  
  <property name="locationUri"  
value="http://localhost:8080/techfestspring/UserService"/>  
  <property name="targetNamespace"  
value="http://www.credera.com/user/definitions"/>  
</bean>  
  
<bean id="userSchema" class="org.springframework.xml.xsd.SimpleXsdSchema">  
  <property name="xsd" value="/WEB-INF/user.xsd"/>  
</bean>
```

- WSDL location

<http://localhost:8080/techfestspring/UserService/user.wsdl>



Securing Spring Web Services

- Spring-WS has built in support for WS-Security
 - Authentication - is the caller authorized to use the service?
 - Digital Signatures - is the caller (and you) who he says he is?
 - Encryption/Decryption - protect sensitive information
- XwsSecurityInterceptor
 - Based on Sun XML and Web Services Security package (XWSS)
 - Requires JDK 1.4+
- Wss4jSecurityInterceptor
 - Based on Apache WSS4J

Securing Spring Web Services (continued)

- Define the security interceptor and related beans

```
<bean id="wsSecurityInterceptor"  
  class="org.springframework.ws.soap.security.xwss.XwsSecurityIn  
  terceptor">  
  
  <property name="policyConfiguration"  
  value="classpath:securityPolicy.xml"/>  
  
  <property name="callbackHandlers">  
    <list>  
      <ref bean="certificateHandler"/>  
      <ref bean="authenticationHandler"/>  
    </list>  
  </property>  
</bean>
```

- Add the interceptor to your endpoint bean

Securing Spring Web Services (continued)

- Setup security policy (if using XWSS) and keystore(s)

```
<xwss:SecurityConfiguration
  xmlns:xwss="http://java.sun.com/xml/ns/xwss/config">
  <xwss:RequireUsernameToken
    passwordDigestRequired="false" nonceRequired="false"/>
  <xwss:RequireSignature requireTimestamp="false"/>
</xwss:SecurityConfiguration>
```

- Use KeyTool to generate a keystore or use the one provided by your certificate authority
- Write callback handlers
- We will look at an example of authentication using Spring Security during the demo section

Other Goodies

- PayloadValidatingInterceptor - validates that request and/or response XMLs conform to the schema definition

```
<bean id="validatingInterceptor"  
  class="org.springframework.ws.soap.server.endpoint.interceptor.PayloadValidatingInterceptor">  
  <property name="schema" value="/WEB-INF/user.xsd"/>  
  <property name="validateRequest" value="false"/>  
  <property name="validateResponse" value="true"/>  
</bean>
```

- PayloadLoggingInterceptor - logs request/response messages

```
<bean  
  class="org.springframework.ws.server.endpoint.interceptor.PayloadLoggingInterceptor"/>
```

Other Goodies (continued)

- transformWsdLocations - overrides the hard coded host and port, uses the URL from the incoming request
- Turned off by default, turn it on in your web.xml

```
<servlet>
    <servlet-name>spring-ws</servlet-name>
    <servlet-
class>org.springframework.ws.transport.http.MessageDispatcherS
ervlet</servlet-class>
<init-param>
<param-name>transformWsdLocations</param-name>
<param-value>true</param-value>
</init-param>
</servlet>
```

Other Goodies (continued)

- You can also configure endpoint implementation beans using Annotations (Java 1.5+)
- @Endpoint
- @PayloadRoot

Spring Web Services Summary

Strengths

- Only does Contract First
- No code generation (i.e. wsdl2java)
- Made exclusively for and by Spring
- Supports two WS-Security implementations (WSS4J and XWSS)
- Many different XML marshalling and handling technologies supported
- Generates WSDL contract from XSD schema

Weaknesses

- Only does Contract First
- Requires programmatic processing of XML message payloads
- No support for JAX-WS

Other Frameworks

Apache Axis2

Strengths

- Next generation of the very popular Axis product
- Very comprehensive
- Well documented
- Has a Spring integration package
- Services can be deployed at runtime
- Has REST support

Weaknesses

- A little heavyweight, especially in the deployment model
- Must deploy the Axis2 WAR to your Servlet container
- Web services are packaged into AAR files (Axis Archive)
- Positioned as a broader SOA solution

Sun Metro Web Services

Strengths

- JAX-WS reference implementation
- Unlike some other RIs Metro is widely viewed as production-quality software

Weaknesses

- Documentation
- Spring integration sub-project is lacking features found in CXF and Spring WS
- Spring integration may not be production ready

JBoss Web Services

Strengths

- Implements JAX-WS
- Adds some other goodies on top of the spec
 - Management tools
 - Security extensions
 - Stateful endpoints
- Tight EJB3 integration
- Part of the JBoss Application Server
- Has a native implementation, can also use Metro and CXF

Weaknesses

- No direct Spring integration

Code

Real-World Examples Brought to you by...



<http://sourceforge.net/projects/broadleaf>

<http://twitter.com/broadleaf>

A Real E-Commerce API

- Common retail e-commerce API - lookup product availability by location
 - Existing Broadleaf Commerce Java API
 - Exposed as a web service using both Spring WS and CXF
 - Demonstrate both contract first and code first approach
 - Focus on WS configurations - annotations, XML config, already wrote most of the Java code
 - Security

Questions