Design and develop JAX-WS 2.0 Web services

Skill Level: Intermediate

Naveen Balani (banaveen@in.ibm.com)
Development Manager
IBM

Rajeev Hathi (rajeevhathi@gmail.com)
Technical Architect
Consultant

20 Sep 2007

Using Java™ API for XML Web Services (JAX-WS) technology to design and develop Web services yields many benefits, including simplifying the construction of Web services and Web service clients in Java, easing the development and deployment of Web services, and speeding up Web services development. This tutorial walks you through how to do all of this and more by developing a sample order-processing application that exposes its functionality as Web services. After going through this tutorial, you'll be able to apply these concepts and your newly acquired knowledge to develop Web services for your application using JAX-WS technology.

Section 1. Before you start

About this tutorial

In this tutorial, you design an develop an order-processing application that exposes its functionality as Web services, whereby various consumers can place order information in a platform-independent manner.

Objectives

After going through this tutorial, you can apply the concepts and knowledge to develop Web services for your application using JAX-WS technology.

Prerequisites

To complete this tutorial successfully, you should have a basic understanding of Web services technology and have some proficiency in Java programming.

System requirements

Related content:

- Web services hints and tips: JAX-RPC versus JAX-WS series
- JAX-WS client APIs in the Web Services Feature Pack for WebSphere Application Server V6.1 series
- Web services on demand demos
- Deliver Web services to mobile apps

To run the examples in this tutorial, you need Java Platform, Standard Edition (Java SE) 6.0 installed.

Section 2. Introduction to JAX-WS

Why JAX-WS?

JAX-WS is a technology designed to simplify the construction of Web services and Web service clients in Java. It provides a complete Web services stack that eases the task of developing and deploying Web services. JAX-WS supports the WS-I Basic Profile 1.1, which ensures that the Web services developed using the JAX-WS stack can be consumed by any clients developed in any programming language that adheres to the WS-I Basic Profile standard. JAX-WS also includes the Java

Architecture for XML Binding (JAXB) and SOAP with Attachments API for Java (SAAJ).

JAXB enables data-binding capabilities by providing a convenient way to map an XML schema to a representation in Java code. The JAXB shields the conversion of the XML schema messages in SOAP messages to Java code without you having to fully understand XML and SOAP parsing. The JAXB specification defines the binding between the Java and XML schemas. SAAJ provides a standard way of dealing with XML attachments contained in a SOAP message.

Furthermore, JAX-WS speeds up Web services development by providing a library of annotations to turn plain old Java object (POJO) classes into Web services. It also specifies a detailed mapping from a service defined in the Web Services Description Language (WSDL) to the Java classes that implement that service. Any complex types defined in the WSDL are mapped into Java classes following the mapping defined by the JAXB specification. JAX-WS was previously bundled with Java Platform, Enterprise Edition (Java EE) 5. The JAX-WS 2.0 specification is developed under JSR 224 of the Java Community Process (JCP).

Section 3. Develop a Web service

Contract-first approach versus code-first approach

A good way to get initiated into JAX-WS is to first develop a Web service. You can develop a Web service using one of two approaches:

- **Contract first:** Start with a WSDL contract, and generate a Java class to implement the service.
- **Code first:** Start with a Java class, and use annotations to generate both a WSDL file and a Java interface.

The contract-first WSDL approach requires a good understanding of WSDL and XSD (XML Schema Definition) for defining message formats. It's a good idea to start with the code-first approach if you're fairly new to Web services, which is what you'll use in this tutorial to develop Web services.

Code-first Web services development

Using the code-first approach, you start with a Java class, or classes, that implements features you want to expose as services. The code-first approach is particularly useful when Java implementations are already available and you need to expose implementations as services.

Develop an order-processing Web service

Let's start by creating an order-processing Web service that accepts order information, shipping information, and ordered items, and ultimately generates a confirmation ID as a response. The code for the order-processing service is provided in Listing 1. This is a dummy implementation that prints the customer ID and number of items at the console, then returns a dummy order ID of *A1234*. (You can download the source code for the complete application in the Download section of this article.) Extract the source code to your C drive, where a folder named JAXWS-Tutorial is created. This folder contains the source code, as shown in Listing 1.

Listing 1. The order-processing Web service implementation

```
package com.ibm.jaxws.tutorial.service;

import javax.jws.WebMethod;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;
import com.ibm.jaxws.tutorial.service.bean.OrderBean;


//JWS annotation that specifies that the portType name of the
//Web service is "OrderProcessPort," the service name
//is "OrderProcess," and the targetNamespace used in the
generated
//WSDL is "http://jawxs.ibm.tutorial/jaxws/orderprocess."

@WebService(serviceName = "OrderProcess",
                portName = "OrderProcessPort",
                targetNamespace =
"http://jawxs.ibm.tutorial/jaxws/orderprocess")

//JWS annotation that specifies the mapping of the service onto
the
// SOAP message protocol. In particular, it specifies that the
SOAP messages
//are document literal.

@SOAPBinding(style=SOAPBinding.Style.DOCUMENT,use=SOAPBinding.Use.LITERAL,
parameterStyle=SOAPBinding.ParameterStyle.WRAPPED)
```

```
public class OrderProcessService {

        @WebMethod
        public OrderBean processOrder(OrderBean orderBean) {

                // Do processing...
                System.out.println("processOrder called for
customer"
                                   +
orderBean.getCustomer().getCustomerId());

                // Items ordered are
                if (orderBean.getOrderItems() != null) {
                        System.out.println("Number of items is
"
                                           +
orderBean.getOrderItems().length);
                }

                //Process order.

                //Set the order ID.
                orderBean.setOrderId("A1234");

                return orderBean;
        }
}
```

The `OrderBean` holds the order information as shown in the Listing 2. Specifically, it contains references to the customer, order items, and shipping address object.

Listing 2. OrderBean class holding order information

```
package com.ibm.jaxws.tutorial.service.bean;

public class OrderBean {

        private Customer customer;

        private Address shippingAddress;

        private OrderItem[] orderItems;

        private String orderId;

        public Customer getCustomer() {
                return customer;
        }

        public void setCustomer(Customer customer) {
                this.customer = customer;
        }

        public String getOrderId() {
                return orderId;
        }

        public void setOrderId(String orderId) {
                this.orderId = orderId;
        }

        public Address getShippingAddress() {
                return shippingAddress;
        }
```

```
        public void setShippingAddress(Address shippingAddress)
{
                this.shippingAddress = shippingAddress;
        }

        public OrderItem[] getOrderItems() {
                return orderItems;
        }

        public void setOrderItems(OrderItem[] orderItems) {
                this.orderItems = orderItems;
        }

}
```

The starting point for developing a JAX-WS Web service is a Java class annotated with the `javax.jws.WebService` annotation. The JAX-WS annotations used are part of the Web Services Metadata for the Java Platform specification (JSR-181). As you have probably noticed, `OrderProcessService` is annotated with the `WebService` annotation, which defines the class as a Web service endpoint.

The `OrderProcessService` class (this is the class with the `@javax.jws.WebService` annotation) implicitly defines a service endpoint interface (SEI), which declares methods that a client can invoke on the service. All the public methods defined in the class, unless the method is annotated with a `@WebMethod` annotation with the exclude element set to *true*, get mapped to WSDL operations. The `@WebMethod` annotation is optional and used for customizing the Web service operation. Apart from the exclude element, the `javax.jws.WebMethod` annotation provides the operation name and action elements, which are used to customize the name attribute of the operation and the SOAP action element in a WSDL document. These properties are optional; if undefined, default values are derived from the class name.

After the Web service is implemented, you need to generate any artifacts required to deploy the service, then package the Web service as a deployed artifact—typically as a WAR file—and deploy the WAR file to any compliant server that supports the JAX-WS 2.0 specification. Typical artifacts generated are classes that provide conversion of Java objects to XML, and the WSDL file and XSD schema based on the service interface.

For testing purposes, Java 6 bundles a lightweight Web server to which the Web service can be published by invoking a simple API call. Next you take a look at how to test your Web services using this approach.

Section 4. Publish the service

Generate JAX-WS artifacts

You generate the JAX-WS portable artifacts for the order-processing Web service by running the *wsgen* tool. This tool reads a Web SEI class and generates all the required artifacts for Web service deployment and invocation. The wsgen tool generates the WSDL file and XSD schema for the Web service, which needs to be published.

For generating the JAX-WS artifacts, you first need to compile the service and beans sources:

1. Open a command prompt, and navigate to c:\JAXWS-Tutorial.
2. Run the following command to compile the Java files and place the class files into their respective folders:
   ```
   javac com\ibm\jaxws\tutorial\service\*.java
   com\ibm\jaxws\tutorial\service\bean\*.java
   ```
3. Run the following command to generate the JAX-WS artifacts:
   ```
   wsgen -cp .
   com.ibm.jaxws.tutorial.service.OrderProcessService
   -wsdl
   ```

The wsgen tool provides lot of options, like generating the WSDL and schema artifacts for the service by providing the `-wsdl` option. After running this command, you should see OrderProcess.wsdl and OrderProcess_schema1.xsd generated in the JAXWS-Tutorial folder, and the JAX-WS artifacts being created in the com\ibm\jaxws\tutorial\service\jaxws folder.

After the artifacts are generated, you publish the order-processing Web service by running the following Web service publisher client.

4. Compile the `OrderWebServicePublisher` by running the following command from the c:\JAXWS-Tutorial folder:
   ```
   javac
   com\ibm\jaxws\tutorial\service\publish\OrderWebServicePublisher.j
   ```
5. Then run the following command:
   ```
   java
   com.ibm.jaxws.tutorial.service.publish.OrderWebServicePublisher
   ```

After running the Java program, you should see the following message: `The Web service is published at`

```
http://localhost:8080/OrderProcessWeb/orderprocess. To stop
running the Web service, terminate this Java process.
```

This publishes the order Web service at the http://localhost:8080/OrderProcessWeb/orderprocess location. You can verify whether the Web service is running by displaying the WSDL generated by the order-processing Web service:

6. Open the browser, and navigate to http://localhost:8080/OrderProcessWeb/orderprocess?wsdl.

Analyze the OrderWebServicePublisher

Before analyzing the WSDL and schema artifacts, let's analyze the code for the `OrderWebServicePublisher`. Listing 3 provides the source code of the `OrderWebServicePublisher` client.

Listing 3. Code for publishing order-processing Web service

```
package com.ibm.jaxws.tutorial.service.publish;

import javax.xml.ws.Endpoint;

import com.ibm.jaxws.tutorial.service.OrderProcessService;

public class OrderWebServicePublisher {

        public static void main(String[] args) {

Endpoint.publish("http://localhost:8080/OrderProcessWeb/orderprocess",
                                new OrderProcessService());

        }

}
```

The `Endpoint.publish()` method provides a convenient way to publish and test the JAX-WS Web service. `publish()` takes two parameters: the location of the Web service and the JAX-WS Web service implementation class. The `publish()` methods create a lightweight Web server at the URL specified (in this case, it's the local host and port 8080) and deploy the Web service to that location. The lightweight Web server is running in the Java virtual machine (JVM) and can be terminated by calling the `endpoint.stop()` method conditionally or terminating the `OrderWebServicePublisher` client.

Analyze the generated WSDL

To view the generated order-processing Web service WSDL, type the following URL location in the browser: `http://localhost:8080/OrderProcessWeb/orderprocess?wsdl`.

Let's analyze some important WSDL aspects and look at how the WSDL and schema artifacts were generated based on JAX-WS metadata, beginning with analyzing the generated XSD. This is imported in a WSDL file using the `xsd:import` tags (see Listing 4); the `schemaLocation` specifies the location of the XSD.

Listing 4. WSDL file containing order-processing schema definition

```
<types>

    <xsd:schema>
      <xsd:import
namespace="http://jawxs.ibm.tutorial/jaxws/orderprocess"
          schemaLocation="OrderProcess_schema1.xsd"/>
    </xsd:schema>
</types>
```

Put the `schemaLocation` (http://localhost:8080/OrderProcessWeb/orderprocess?xsd=1) in the browser to see the schema definitions render in the browser. Let's analyze what's happening here: The schema definition starts with `targetNamspace` and a `tns` declaration, which maps to the `targetNamespace`, http://jawxs.ibm.tutorial/jaxws/orderprocess, that you've defined in the `@WebService` annotation for `OrderProcessService`. Listing 5 provides the code.

Listing 5. Schema namespace declaration

```
<xs:schema version="1.0"
targetNamespace="http://jawxs.ibm.tutorial/jaxws/orderprocess"
    xmlns:tns="http://jawxs.ibm.tutorial/jaxws/orderprocess"
    xmlns:xs="http://www.w3.org/2001/XMLSchema">
```

The wsgen tool executed earlier generates two wrapper bean classes, `ProcessOrder` and `ProcessOrderResponse`, which hold input and output messages for the order-processing Web service. Based on the wrapper bean classes, the following schema elements are generated:

- `processOrder` is of the type `processOrder`, which represents a complex type containing one element with the name `arg0` and the type `orderBean`. You can see a one-to-one mapping between the `ProcessOrder` class and `processOrder` complex type.
- `processOrderResponse` is similarly of the type `processOrderResponse` whose definitions map to the

`ProcessOrderResponse` class.

Let's look more closely at that in Listing 6.

Listing 6. Schema declaration for processOrder

```
<xs:element name="processOrder" type="tns:processOrder"/>
  <xs:element name="processOrderResponse"
type="tns:processOrderResponse"/>
  <xs:complexType name="processOrder">
    <xs:sequence>
      <xs:element name="arg0" type="tns:orderBean"
minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
```

The `orderBean` type definition illustrated in Listing 7 maps to the `OrderBean` class. The `orderBean` type definition is comprised of:

- A `customer` element whose type is `customer`.
- An `orderId` whose type is `string`.
- `orderItems` (which is an array, because it specifies the `maxOccurs` attribute as `unbounded`) whose type is `orderItem`.
- `shippingAddress` whose type is `address`.

Listing 7. Schema declaration for processOrder

```
<xs:complexType name="orderBean">
<xs:sequence>
<xs:element name="customer" type="tns:customer" minOccurs="0"
/>
          <xs:element name="orderId" type="xs:string"
minOccurs="0" />
          <xs:element nillable="true" maxOccurs="unbounded"
name="orderItems"
                         type="tns:orderItem" minOccurs="0" />
          <xs:element name="shippingAddress" type="tns:address"
                      minOccurs="0" />
</xs:sequence>
</xs:complexType
```

Similarly, the rest of the schema definitions for `customer`, `orderItems`, and `address` are mapped to the `Customer`, `OrderItem`, and `Address` Java beans, respectively.

With the schema definitions analyzed, let's revisit the message definitions in WSDL, which are shown in Listing 8. The WSDL specifies the messages `processOrder` and `processOrderResponse` whose part elements are `processOrder` and `processOrderResponse` (you've already seen their schema definitions). The `portType` specifies the operation `processOrder` whose input message is

processOrder and whose output message is processOrderResponse.

Listing 8. processOrder message element in WSDL document

```
  <message name="processOrder">
                  <part element="tns:processOrder"
name="parameters" />
  </message>
  <message name="processOrderResponse">
                  <part element="tns:processOrderResponse"
name="parameters" />
  </message>
  <portType name="OrderProcessService">
  <operation name="processOrder">
  <input message="tns:processOrder" />
  <output message="tns:processOrderResponse" />
  </operation>
  </portType>
```

Next, the WSDL bindings are defined. This defines the soap:binding style as document and the soap:body use tag as literal for input and output message formats for the operation processOrder. The generated WSDL definitions map to the @SOAPBinding annotation that you defined on the OrderProcessService class (see Listing 9).

Listing 9. Binding information for WSDL document

```
 <binding name="OrderProcessPortBinding"
type="tns:OrderProcessService">
  <soap:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http" />
 <operation name="processOrder">
  <soap:operation soapAction="" />
 <input>
  <soap:body use="literal" />
  </input>
  <output>
  <soap:body use="literal" />
  </output>
  </operation>
</binding>
```

Next, the WSDL services are defined. These specify the port and corresponding binding type, along with the actual location of the service. This is typically an HTTP location, which in this case is http://localhost:8080/OrderProcessWeb/orderprocess. You can see this in detail in Listing 10.

Listing 10. Service information for WSDL document

```
<service name="OrderProcess">
  <port name="OrderProcessPort"
binding="tns:OrderProcessPortBinding">
  <soap:address
location="http://localhost:8080/OrderProcessWeb/orderprocess"
/>
```

```
</port>
```

With this you've analyzed the generated WSDL and schema artifacts. Listing 11 illustrates a sample SOAP request message sent by the Web service client when it invokes the `processOrder` operation.

Listing 11. Sample SOAP message for processOrder operation

```
<?xml version="1.0"?>
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:ns1=" http://jaxws.ibm.tutorial/jaxws/orderprocess"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<soapenv:Body>
<ns1:processOrder>
<arg0>
<customer><customerId>A123</customerId>
<firstName>John</firstName><lastName>Smith</lastName></customer>
<orderItems><itemId>11</itemId><qty>11</qty></orderItems>
</arg0>
</ns1:processOrder>
</soapenv:Body>
</soapenv:Envelope>
```

Section 5. Create Web service clients

Create Web service clients from WSDL

In this section, you learn how to create Web service clients from WSDL. JAX-WS comes with a tool called *wsimport* that's used to generate JAX-WS portable artifacts from WSDL. The portable artifacts typically generated include the following:

- SEI
- Service (the service implementation class you need to implement)
- JAXB-generated classes from schema types
- Exception class mapped from wsdl:fault (if any)

Clients use the artifacts generated to invoke the Web service. The Web service clients don't need to deal with any SOAP format, like creating or parsing SOAP messages. Rather, this is handled by the JAX-WS run time, which uses the generated artifact code (the JAXB-generated class). The Web service client in turn deals with the Java object (the JAXB-generated class), which eases the

development of Web service clients and invoking operations on the Web service.

You generate JAX-WS artifacts from the `OrderProcess` WSDL using the wsimport tool. Then you create a Web service client, which uses generated artifact code to invoke the order-processing Web service. To generate the JAX-WS artifacts, navigate to the JAXWS-Tutorial directory, and run the wsimport command shown in Listing 12. Before doing this, though, make sure you've published the Web service by running `OrderWebServicePublisher` as pointed out in step 5 in the Generate JAX-WS artifacts section.

Listing 12. wsimport command for generating JAX-WS artifacts used by Web service client

```
wsimport -keep -p com.ibm.jaxws.tutorial.service.client
        http://localhost:8080/OrderProcessWeb/orderprocess?wsdl
```

The `-keep` option indicates that you keep the generated files, and the `-p` option specifies the package name where the artifact needs to be generated. `http://localhost:8080/OrderProcessWeb/orderprocess?wsdl` specifies the location of the WSDL file. The following artifacts are generated from the `OrderProcessService` WSDL:

- **JAXB classes (`Address`, `Customer`, `OrderBean`, and `OrderItem`):** Generated by reading the schema definitions defined in the `OrderProcessService` WSDL
- **`RequestWrapper` and `ResponseWrapper` classes (`ProcessOrder` and `ProcessOrderResponse`):** Wrap the input and output for document literal-wrapped style
- **Service class (`OrderProcess`):** The class that your clients use to make requests to the Web service
- **Service interface (`OrderProcessService`):** Class contains the interface, which your service implements

Now take a look at how to create a Web service client using the artifacts you generated above. A sample reference code is provided in the com\ibm\jaxws\tutorial\service\client folder. The code for the Web service client is provided in Listing 13.

Listing 13. Code listing for order-processing Web service client

```
package com.ibm.jaxws.tutorial.service.client;

import java.net.MalformedURLException;
import java.net.URL;
```

```
import javax.xml.namespace.QName;

public class OrderClient {

        final QName qName = new QName(
"http://jawxs.ibm.tutorial/jaxws/orderprocess", "OrderProcess");

        public static void main(String[] args) {
                if (args.length != 1) {
                        System.out
                                        .println("Specify the URL of
the OrderProcess Web Service");
                        System.exit(-1);
                }
                URL url = getWSDLURL(args[0]);
                OrderClient client = new OrderClient();
                client.processOrder(url);
        }

        private static URL getWSDLURL(String urlStr) {
                URL url = null;
                try {
                        url = new URL(urlStr);
                } catch (MalformedURLException e) {
                        e.printStackTrace();
                        throw new RuntimeException(e);
                }
                return url;
        }

        public void processOrder(URL url) {

                OrderProcess orderProcessingService = new
OrderProcess(url, qName);

                System.out.println("Service is" +
orderProcessingService);

                OrderBean order = populateOrder();

                OrderProcessService port =
orderProcessingService.getOrderProcessPort();
                OrderBean orderResponse = port.processOrder(order);

                System.out.println("Order id is " +
orderResponse.getOrderId());

        }

        private OrderBean populateOrder() {

                OrderBean order = new OrderBean();
                Customer customer = new Customer();
                customer.setCustomerId("A123");
                customer.setFirstName("John");
                customer.setLastName("Smith");
                order.setCustomer(customer);

                // Populate Order Item.
                OrderItem item = new OrderItem();
                item.setItemId("11");
                item.setQty(11);

                order.getOrderItems().add(item);
                return order;
        }
}
```

The Web service client code listed above:

- Creates an instance of the `OrderProcess` class by passing in the WSDL URL of the `OrderProcess` Web service along with the `QName` of the service.
- Creates an instance of `OrderBean` and populates the order information in the `populateOrder()` method.
- Retrieves a proxy to the service, also known as a port, by invoking `getOrderProcessPort()` on the service. The port implements the service interface defined by the service.
- Invokes the port's `processOrder` method, passing the `OrderBean` instance created in the second list item above.
- Gets the `OrderBean` response from the service and prints the order ID.

Section 6. Run the Web service client

To run the Web service client, first compile the Web service client by running the following command from the JAXWS-Tutorial folder:

```
javac com\ibm\jaxws\tutorial\service\client\OrderClient.java
```

Execute the web service client by providing the WSDL URL for the order process Web service using this command:

```
java com.ibm.jaxws.tutorial.service.client.OrderClient
http://localhost:8080/OrderProcessWeb/orderprocess?wsdl
```

When the Web service client is executed, you'll see the following output at the console, where `OrderWebServicePublisher` is running:

```
processOrder called for customer A123
Number of items is 1
```

At the console where the Web service client is executed, you get the following output:

```
Order id is A1234
```

As you see in the client code, you don't deal with any SOAP or XML-based format for invoking Web service operations; instead you deal with generated JAXB classes for input and output messages and use the service interface and service class objects, which act as stubs for Web service invocation. The stubs are responsible for creating SOAP requests from JAXB annotations and converting the SOAP response

back to the Java object.

You have now successfully created and published your Web service and executed it via a Web service client!

Section 7. Summary

In this tutorial, you learned how to design and develop Web services using the code-first development approach and JAX-WS technology. JAX-WS is a great choice because it provides a complete Web services stack to simplify the development and deployment of Web services.

The order-processing Web service you developed in this tutorial uses the document-style Web service, which ensures that the service consumer and service provider communicate using XML documents. The XML documents adhere to well-defined contracts, typically created using XML Schema definitions. The XML Schema format specifies the contract of the business messages that service consumers can call, and adheres it. Document-style Web services should be the preferred approach of developing enterprise Web services.

Downloads

Description	Name	Size	Download method
JAX-WS Web services code	jaxws.zip	32KB	HTTP

Information about download methods

Resources

Learn

- Read the *Hands on Web Services* book for comprehensive hands-on information about how to design and develop real-world Web services applications.
- Check out the article "Web services architecture using MVC style" (developerWorks, February 2002) to learn how the MVC architecture can be applied to invoke static or dynamic Web services.
- "Deliver Web services to mobile apps" (developerWorks, January 2003) explains how to access Web services using J2ME-enabled mobile devices.
- The SOA and Web services zone on IBM® developerWorks hosts hundreds of informative articles and introductory, intermediate, and advanced tutorials on how to develop Web services applications.
- The IBM SOA Web site offers an overview of SOA and how IBM can help you get there.
- Stay current with developerWorks technical events and webcasts. Check out the following SOA and Web services tech briefings in particular:
  - Get started on SOA with WebSphere's proven, flexible entry points
  - Building SOA solutions and managing the service lifecycle
  - SCA/SDO: To drive the next generation of SOA
  - SOA reuse and connectivity
- Browse for books on these and other technical topics at the Safari bookstore.
- Check out a quick Web services on demand demo.

Get products and technologies

- Innovate your next development project with IBM trial software, available for download or on DVD.

Discuss

- Participate in the discussion forum for this content.
- Get involved in the developerWorks community by participating in developerWorks blogs, including the following SOA and Web services-related blogs:
  - Service Oriented Architecture -- Off the Record with Sandy Carter

- Best Practices in Service-Oriented Architecture with Ali Arsanjani
- WebSphere® SOA and J2EE in Practice with Bobby Woolf
- Building SOA applications with patterns with Dr. Eoin Lane
- Client Insights, Concerns and Perspectives on SOA with Kerrie Holley
- Service-Oriented Architecture and Business-Level Tooling with Simon Johnston
- SOA, ESB and Beyond with Sanjay Bose
- SOA, Innovations, Technologies, Trends...and a little fun with Mark Colan

About the authors

Naveen Balani
Naveen Balani works as a development manager for WebSphere Business Services Fabric in IBM India. He's a regular contributor to developerWorks and has written about such topics as Web services, ESB, JMS, SOA, architectures, open source frameworks, semantic Web, J2ME, Persuasive Computing, Spring, Ajax, and various IBM products. He is currently coauthoring a book about Spring 2 and Web services.

Rajeev Hathi
Rajeev Hathi has been working as a software consultant for the J2EE platform. His interests are architecting and designing J2EE-based applications. His favorite subject is Web services, through which he likes to apply and impart SOA concepts. His hobbies are watching sports and listening to rock music.

Trademarks