Build SOA with Web services using WebSphere Studio, Part 2: Create a Web service from a Java class

Skill Level: Intermediate

Warner Onstine (webserv@us.ibm.com)
Senior Mentor
ArcMind

Rick Hightower (webserv@us.ibm.com)
Chief Mentor
ArcMind

11 Nov 2004

Learn how to create a Web service that provides a DVD rental service and a client for that service. This tutorial shows you how to create a Web service from a Java class using WebSphere® Studio Application Developer Integration Edition. The next two tutorials cover two DVD rental services and a rental service search aggregator service. (See all parts of this tutorial series.)

Section 1. About this tutorial

Purpose of this tutorial

Learn how to create a Web service that provides a DVD rental service and a client for that service. The next two tutorials will cover some additional services as outlined below:

- Two DVD rental services
- A rental service search aggregator service

The two DVD rental services have slightly different interfaces to show you how the search aggregator can search two (slightly) disparate sources. Initially we only focus on the client side of the Web service, which will access the aggregator that searches
for and sends rental recommendations. The next 2 tutorials will cover the aggregator and DVD Rental provider's interfaces more in depth.

What you need to know for this tutorial

This tutorial assumes you have a working knowledge of Java programming and XML. Knowledge of J2EE technology is helpful but not required. All of the example applications will be deployed on the IBM WebSphere® Application Server (Application Server) that ships as part of WebSphere Studio Application Developer Integration Edition (Application Developer -- see Resources for link to a trial version).

What this tutorial covers

This tutorial explores the world of Web services and Service-Oriented Architecture. The following topics, tools, and techniques will be covered:

Topics

• Introduction to SOA
• Introduction to creating Web services from Java classes using Application Developer
• Introduction to Web services tools

Tools

• Application Developer Web services wizard
• Application Developer Web services exploration

Techniques

• Create Web services from Java classes
• Deploy that model as a Web service
• Create services that interact with each other

About the labs in this series

All of these tutorials focus on a DVD rental search engine that aggregates search results from one or more DVD rental shops that have made their catalogs available online. The use case we address, Rental Recommendation, involves allowing users to submit the movie category they would like to rent and, based on their rental history, using Web services to suggest other movies. In the first tutorial (“Build SOA with Web services using WebSphere Studio, Part 1: Introduction to SOA and Web services,” see Resources), we simply created a simple Web service and ran it. In this second tutorial, you can expand your exploration of working with Web services with three labs:
• **Lab 1: Build and deploy a Web service** -- In this lab you create and deploy a simple Web service that your aggregator will expose for the client.

• **Lab 2: Exchange complex SOAP objects** -- In this lab you can enhance your Web service by allowing it to return more complex SOAP objects for your rental client to use.

• **Lab 3: Dynamic Web service client** -- In this lab you can dynamically create a Web service client for consuming the complex SOAP objects.

**Tools you need for this tutorial**

At a minimum you need a Java SDK 1.3.1 or higher in order to run Application Developer. The IBM developerWorks site has a trial version [WebSphere Application Developer Integration Edition](http://www.ibm.com/developerWorks) and other information.

---

**Section 2. What is Service-Oriented Architecture and service modeling?**

**Introduction to Service-Oriented Architecture**

_Service-Oriented Architecture_ and _service modeling_ are both relatively new terms and, as such, are still being debated and defined within the Web services community as a whole.

**Service-Oriented Architecture**

Service Oriented Architecture is essentially a collection of services that communicate with each other. The communication can involve two or more services coordinating some activity. Some means of connecting services to each other is needed. A service is a function that is well-defined, self-contained, and does not depend on the context or state of other services. SOA is not restricted to Web services, although IBM considers it the best way to accomplish this.

**Service modeling**

Service modeling is the modeling of different services with each other, this comes into play when one organization has defined several services that need to work together but remain separate entities. One example of this would be where a company wants to expose their ERP system to other business units to allow them to connect to it without needing to install special software on their computers. They also additionally want to expose their accounting software to the ERP system so that they can coordinate data from one system to another. There could also be an
additional service that aggregates all the disparate services under one main service for use by an internal Web application.

Here is where you can see the benefits of a SOA, internal to a company, where the company has control over the service interfaces to its business units. That isn’t to say that it is simple to implement, but it becomes much easier than trying to coordinate with an external vendor who might decide to add or drop pieces from their interface without consulting their customers.

Service-Oriented Architecture is typically expressed as a client sending a request to a service provider who might or might not send a response. Here are some typical interactions:

- Receive messages only (no return message)
- Receive and respond (request-response)
- Solicit response (output message and input message)
- Publish/subscribe (outgoing messages only)

So, as you can see there are a wide variety of things that you could conceivably do with this. In this example we want to define how you could use these.

In your search client you need the ability to send a query to the aggregator and have it return a response, so you look at the receive and respond interaction. Additionally, you want the same kind of interface between the aggregator and the rental providers, allowing the aggregator to send its own query to the rental providers and receive a response back that it then combines together to deliver to the client.

You could also have the option in the search client to subscribe to a listing of the latest DVDs. When DVDs are released (for example, every Tuesday) a message could be sent to the client informing them of some of the latest offerings, allowing them to rent these immediately or browse additional new titles.

This same principle works in reverse for the aggregator if it subscribes to a similar service from its rental providers (the receive messages-only interaction).

As you can see, it can get a little confusing, but the interactions are all defined from whatever point of view you are examining it.

### Design and SOA

When you decide that you are going to implement a SOA, there are some factors that you should take into account:

- **Service identification** design decisions -- the fast proliferation of objects has not allowed for organizations to realize economies of scale because it inherently limits the potential for business utility. Many early adopters of SOA and Web services realized quickly that the proliferation of Web
services does not make for a sound SOA model. These design choices are made during Service-Oriented Analysis and Design (SOAD) described later in this series (see Resources for more information on SOAD).

• **Container design and realization** architecture and design decisions are critical for a service to provide the critical qualities for extensibility and maintainability.

• **Granularity design choices** about services must be matched to the level of reusability and flexibility required, given the context. Larger-grained services can help encapsulate changes in finer-grained technical services that can tend to change more frequently than the higher-level business-level service interface. Factors of flexibility are key criteria for encapsulation rather than merely the encapsulation of function.

• **State of service** often requires both architecture and design decisions on the stateless nature of services, yet state must be maintained. This is often addressed by making architectural decisions on choreography engines (such as a BPEL engine) and design choices on the stateless nature of a proposed business service.

• **Loose coupling and dynamic reconfiguration** is a design decision that requires the decoupling of interfaces from implementation and protocol realization through open standards. The connection between service consumers and service providers needs to be stable and well structured, yet flexible and readily reconfigurable. Re-configurable means that existing service consumers and service providers can be re-assembled with their functionality intact to obtain business solutions in different technical environments and with different operational constraints with new business partners across a value chain. Therefore, integration alone is no longer adequate. Dynamic reconfiguration is the name of the game and needs to be pervasive along a spectrum of support in areas of infrastructure (in other words, the enabling software infrastructure), tools, development platforms, reference architectures, patterns, methods and industry-specific reference models.

So, what are the key components of SOA?

• **Services portfolio**: Describes the business services in SOA. This includes a list, classification, and hierarchy of services defined through the technique of Service-Oriented Analysis and Design.

• **Components**: Provide the functional realization of the services.

• **Service providers, service consumers, and optionally, the service broker(s)**: These participate with their service registries where service definitions and descriptions are published.

• **SOA layers**: This is where components and services reside.
Just like any other architecture, SOA has many different layers that it must interact with. These include:

1. **Systems layer**: Databases, Messaging systems, and other packaged software
2. **Services layer**: Other Web services
3. **Business layer**: Business-specific components
4. **Presentation layer**: Your Web or client front-end

Understanding these layers and how they must interact with the services and how the services must interact with them is key in any SOA design.

**Service modeling and UML**

Service modeling is a close cousin of UML (Unified Modeling Language), which was created to allow developers to visually model their code and then generate code from their diagrams and implement specific interfaces that they modeled.

The OMG specification states:

"The Unified Modeling Language (UML) is a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system. The UML offers a standard way to write a system's blueprints, including conceptual things such as business processes and system functions as well as concrete things such as programming language statements, database schemas, and reusable software components."

The important point to note here is that UML is a *language* for specifying and is not a method or procedure. The UML is used to define a software system -- to detail the artifacts in the system, document, and construct -- it is the language in which the blueprint is written. The UML can be used in a variety of ways to support a software development methodology (such as the Rational Unified Process) but, in itself, it does not specify that methodology or process.

UML defines the notation and semantics for the following domains:

- **The User Interaction or Use Case Model**: Describes the boundary and interaction between the system and users. Corresponds in some respects to a requirements model.
- **The Interaction or Communication Model**: Describes how objects in the system will interact with each other to get work done.
- **The State or Dynamic Model**: State charts describe the states or conditions that classes assume over time. Activity graphs describe the workflows the system will implement.
- **The Logical or Class Model**: Describes the classes and objects that will make up the system.
The Physical Component Model: Describes the software (and sometimes hardware components) that make up the system.

The Physical Deployment Model: Describes the physical architecture and the deployment of components on that hardware architecture.

The UML also defines extension mechanisms for extending the UML to meet specialized needs (for example Business Process Modeling extensions or SOA).

UML history and purpose

During the 1990s many different methodologies, along with their own set of notations, were introduced to the market. Three of the most popular methods were Object Modeling Technique (OMT) (James Rumbaugh), Booch (Grady Booch), and OOSE (Ivar Jacobson). Each method had its own value and emphasis. OMT was strong in analysis and weaker in the design area. Booch was strong in design and weaker in analysis. Jacobson was strong in behavior analysis and weaker in the other areas.

As time moved on, Booch wrote his second book, which adopted a lot of the good analysis techniques advocated by Rumbaugh and Jacobson, among others. Rumbaugh published a series of articles that have become known as OMT-2 that adopted a lot of Booch's good design techniques. The methods were beginning to converge but they still had their own unique notations. The use of different notations brought confusion to the market since one symbol meant different things to different people. For example, a filled circle was a multiplicity indicator in OMT and an aggregation symbol in Booch. You will hear the term method wars being used to describe this period of time -- is a class a cloud or a rectangle? Which one is better?

The end of the method wars as far as notation is concerned comes with the adoption of the Unified Modeling Language (UML). "UML is a language used to specify, visualize, and document the artifacts of an object-oriented system under development. It represents the unification of the Booch, OMT, and Objectory notations, as well as the best ideas from a number of other methodologists. By unifying the notations used by these object-oriented methods, the Unified Modeling Language provides the basis for a de facto standard in the domain of object-oriented analysis and design founded on a wide base of user experience." (From Rational Rose’s description of UML.)

UML is now governed by the Object Management Group (OMG) which controls the specification of new UML notations and has adopted UML as its standard modeling language. You can find out more about OMG at their Web site.

Well, now that you know a little bit about SOA and modeling, let's get into some code!
Section 3. Lab 1: Build and deploy a Web service

Lab purpose and overview

The purpose of this lab is to provide you with hands-on experience building, deploying, and testing a Java Web service using Application Developer. Version 5.1 of Application Developer lets you generate Web services, Web services clients, and WSDL documents from existing artifacts.

The following screenshots show step by step how to expose an existing JavaBean component as a Web service:

1. Create a Web project called SimpleRPC to contain the JavaBean source code (you do not need to check off "Configure Advanced Options" for this part of the tutorial).

2. Create a new package to contain the JavaBean component.
3. Name the package `lpc.dvdonline.simplerpc.server`.

4. Create a new Java class called `DVDOnlineStore`. 
5. Add the following code to the source file:

```java
package lpc.dvdonline.simplerpc.server;
public class DVDOnlineStore {
    public String getDVD() {
        return "Fight Club";
    }
}
```

6. Create the new Web service.

7. Select the "Java bean Web Service" option and click **Next**.
8. Select the Web service project to be used at runtime and click **Next**.
9. Select the JavaBean component to expose and click **Next**.
10. Select the style and use of the Web service and click **Finish**. This step may take awhile.
Test the Web service

Now that you have a service, you need a way to test it. Fortunately, Application Developer makes this very easy.

1. Right-click **DVDOntlineStore.java** and select "Launch the Universal Test Client" from the Web Services menu.

   ![Image of Universal Test Client](image)

   This brings you directly to the Web browser showing your services.

2. Browse to your service by clicking on the object reference for **DVDOntlineStore**.
3. Click on the **getDVD** method and then invoke the service by clicking the **Invoke** button.
4. If you see the response message of "Fight Club," then you have tested the service and it is alive and working. Congratulations!

What you are doing

Let's review what you have done what the Web services wizards did for you.

You created a Web service using a standard Java class. You did not define an interface and create a binding. The Web service wizard took care of most of the heavy lifting. It deployed your Web service as a J2EE Web application bundled in a WAR file bundled in an EAR file.

Although you did not define an interface, the Web service wizard created one for you as follows:

```java
package lpc.dvdonline.simplerpc.server;

public interface DVDOnlineStore_SEI extends java.rmi.Remote {
  public java.lang.String getDVD();
}
```

If you have done RMI development, this interface looks identical to the type of interface you would have created for your RMI servant.
The Web services wizard also created a WSDL file. We leave the WSDL file coverage to the WSDL tutorial, "Describing Web Services in WSDL" (See Resources).

---

Section 4. Lab 2: Exchange complex SOAP objects

Lab overview

The purpose of this lab is to demonstrate the ability of Application Developer to exchange complex SOAP data types. Although passing Strings and simple arrays is a good way to get you started with Web services, real-world applications typically require that client and service exchange whole objects and similar complex data structures. JAX-RPC and XML Schema support complex data types.

Define a Java type to represent a DVD object

When SOAP was created, the idea was to provide a neutral mechanism for transporting objects within a distributed and heterogeneous environment. That's where SOAP got its name: Simple Object Access Protocol. To accomplish this, each side must have a native object representation and then agree upon a neutral format to map with their native representation. In this case, both client and server are Java code-based, so they both use the same native format: a Java class.

The DVD class is a basic JavaBean component with three fields: title (String), rank (int), and category (String), along with the usual getters and setters.

1. Create a new Web project called ComplexRPC. If desired, modify the EAR that will contain the Web project.
2. Create a package called lpc.dvdonline.complexrpc.service.
3. Create a new Java class called DVD.
4. Add the following code to the source file:

```java
package lpc.dvdonline.complexrpc.service;

public class DVD {

    private String title; // unique id (primary key)
    private int rank;
    private String category;

    public DVD() {
    }
```
public DVD(String _title, int _rank, String _category) {
    title = _title;
    rank = _rank;
    category = _category;
} //end DVD( String, int, String )

public String getCategory() {
    return category;
} //end getCategory()

public int getRank() {
    return rank;
} //end getRank()

public String getTitle() {
    return title;
} //end getTitle()

public void setCategory(String category) {
    this.category = category;
} //end setCategory( String )

public void setRank(int rank) {
    this.rank = rank;
} //end setRank( int )

public void setTitle(String title) {
    this.title = title;
} //end setTitle( String )
} //end class DVD

Create a Web service from a Java class
Page 16 of 34 © Copyright IBM Corporation 1994, 2007. All rights reserved.
Build the service provider for ComplexRPC

In this section you create a Web service that returns complex types. Perform the following steps:

1. Create a new package called lpc.dvdonline.complexrpc.server.
2. Create a Java class called DVDOnlineStore.
3. Add the following code to the source file:

```java
package lpc.dvdonline.complexrpc.server;
import lpc.dvdonline.complexrpc.service.DVD;
public class DVDOnlineStore {

    public DVD getDVD() {
        return new DVD("Fight Club", 10, "Weird but good");
    }
}
```

Notice how the above source is different than the last provider. The getDVD() method returns an instance of the DVD value object class that you just created instead of a plain old String.
4. Right-click on **DVDOntlineStore.java** and select "Deploy as Web Service" from the Web Services menu. Then click **Next**.
5. Select the JavaBean component to expose and click **Next**.

6. Select the style and use of the Web service and click **Finish**. This step might take awhile.
Test the Web service

1. Right-click on `DVDOnlineStore.java` and select "Launch the Universal Test Client" from the Web Services menu.
This brings you directly to the Web browser showing your services

2. Browse to your service by clicking on the object reference for DVDOnlineStore.
3. Click on the `getDVD` method and then invoke the service by clicking the `Invoke` button.
4. You now have an instance of the DVD object.

5. To retrieve information about the DVD object that was returned, you need to click on its instance, which gives you access to all of its methods.
6. To retrieve the title, click on the `getTitle()` method.
7. If you see the response message of "Fight Club," then you have tested the service and it is alive and working. Congratulations!

Section 5. Lab 3: Dynamic Web services client

Lab purpose and overview

In a perfect world, you would always be both the service provider and service requester developer. And, if you were not both the service provider and service requester developer, the service provider developer would be nice and supply you with a WSDL file. Then you could take this WSDL file and generate client stubs. (Note: there is a whole tutorial, "Top-down Web service development: Build a WSDL file to generate a Web service using WebSphere Studio," that shows you how to work with WSDL files to generate provider and requester code (See Resources). The Web services wizard hides a lot of WSDL details.)

However, the world is nowhere near perfect. And service providers can develop SOAP-based services without WSDL files. Sad but true. So how do you invoke these services anyway? You use dynamic SOAP invocation.

There are four JAX-RPC classes you need to worry about for dynamic invocation, and they are as follows:

1. javax.xml.namespace.QName
2. javax.xml.rpc.Call
3. javax.xml.rpc.Service
4. javax.xml.rpc.ServiceFactory

ServiceFactory is an abstract class that acts as a factory for instantiating JAX-RPC Services. Service is a factory class for Call objects. You use the Service to create a Call object. Then you pass the Call object its operation name and its endpoint URL so it knows where the service is located. The operation name is identified with a QName. A QName represents an element name that is within a namespace.

Let's walk through this with the code.

1. Import the needed classes:

```java
import javax.xml.namespace.QName;
import javax.xml.rpc.Call;
```
import javax.xml.rpc.Service;
import javax.xml.rpc.ServiceFactory;

2. To create a Service factory object, you need to pass the fully qualified name of the Web service (that is, namespace plus service name) to the createService() method, as follows:

```java
ServiceFactory serviceFactory = ServiceFactory.newInstance();
String targetNamespace = "http://server.simplerpc.dvdonline.lpc";
String serviceName = "DVDOnlineStore";
Service service = serviceFactory.createService(new QName(namespace, serviceName));
```

3. Use the Service to create the call object:

```java
Call call = (Call) service.createCall();
```

4. Set the URL location of the Service:

```java
String endpoint = "http://localhost:6080/SimpleRPC/services/DVDOnlineStore";
call.setTargetEndpointAddress(endpoint);
```

5. Invoke the call object:

```java
String dvdName = (String) call.invoke(null);
```

Create the dynamic service client for SimpleRPC

Now that you get the idea of how to do it, let's write a dynamic client for the SimpleRPC lab.

1. Inside the package lpc\dvdonline\simplerpc\client directory, create a Java source file called DynamicDVDOnlineStoreClient.java. Add the following code to that source file:

```java
package lpc.dvdonline.simplerpc.client;
import javax.xml.namespace.QName;
import javax.xml.rpc.Call;
import javax.xml.rpc.ServiceFactory;
import javax.xml.rpc.Service;
public class DynamicDVDOnlineStoreClient {
    public static void main(String[] args) throws Exception {
        String targetNamespace = "http://server.complexrpc.dvdonline.lpc";
        String serviceName = "DVDOnlineStore";
        String endpoint = "http://localhost:9080/SimpleRPC/services/DVDOnlineStore";
    }
}
```
2. Notice that the client does not need the stubs, interface, and locator classes that the Web service wizard generated.

3. First, to make sure that the test server is running, click on the arrow next to the “Running Man” icon and select “WebSphere Test Environment.”

4. To run the client, click the arrow next to the little “Running Man” icon on the toolbar and choose “Run as > Java Application.”

If you see the response message of “Dynamic Fight Club,” then you have earned
your Web services Kung Fu brown belt!

What about dynamic clients with complex types?

"Fine," you say. Good and well -- you can invoke Web services that return simple types and have simple arguments, but what would happen if the Web service had complex types like this complex example? Simply put: the dynamic example above would break. So far we have managed to stick fairly close to doing dynamic things in a vendor-neutral fashion. Once you start working with complex types, using the dynamic mechanism becomes more vendor-specific. Future versions of the JAX-RPC API will fix this.

With Application Developer you have to register a custom serializer and deserializer for the complex type. It is really quite simple. For completeness, we create a dynamic client for the complex example.

Create the dynamic service client for ComplexRPC

1. Inside the package lpc\dvonline\complexrpc\client directory, create a Java source file called DynamicDVDOnlineStoreClient.java. Add the
following code to that source file:

```java
package lpc.dvdonline.complexrpc.client;

import javax.xml.namespace.QName;
import javax.xml.rpc.Call;
import javax.xml.rpc.Service;
import javax.xml.rpc.ServiceFactory;
import javax.xml.rpc.encoding.TypeMapping;
import javax.xml.rpc.encoding.TypeMappingRegistry;
import lpc.dvdonline.complexrpc.service.DVD;
import com.ibm.ws.webservices.engine.encoding.ser.BeanDeserializerFactory;
import com.ibm.ws.webservices.engine.encoding.ser.BeanSerializerFactory;

public class DynamicDVDOnlineStoreClient {

    public static void main(String[] args) throws Exception {
        String namespace = "http://server.complexrpc.dvdonline.lpc";
        String serviceName = "DVDOnlineStoreService";
        String endpoint = "http://localhost:6080/ComplexRPC/services/DVDOnlineStore";

        /* Service lookup */
        Service service = ServiceFactory.newInstance().createService(
                new QName(namespace, serviceName));

        /* Define and register the type mapping */
        TypeMappingRegistry tmr = service.getTypeMappingRegistry();
        TypeMapping tm = tmr.createTypeMapping();
        QName dvdQName = new QName(namespace, "DVD");
        tm.register(DVD.class, dvdQName, new BeanSerializerFactory(DVD.class, dvdQName),
                new BeanDeserializerFactory(DVD.class, dvdQName));
        tmr.register("", tm);

        /* Service access */
        Call call = (Call) service.createCall();
        call.setProperty(Call.ENCODINGSTYLE_URI_PROPERTY, "");
        call.setProperty(Call.OPERATION_STYLE_PROPERTY, "wrapped");
        call.setTargetEndpointAddress(endpoint);
        call.setReturnType(dvdQName, DVD.class);
        call.setPortTypeName(new QName(namespace, serviceName));
        call.setOperationName(new QName(namespace, "getDVD"));

        /* Service invocation */
        DVD dvd = (DVD) call.invoke(null);
        System.out.println("Dynamic DVD Title " + dvd.getTitle());

        //end main()
    } //end DynamicDVDOnlineStoreClient
}
```

2. To run the client, click the arrow next to the little "Running Man" icon on
the toolbar and choose "Run as > Java Application."

If you see the response message of "Dynamic DVD Title Fight Club,"
congratulations! You are a Web services Kung Fu master!
Section 6. Tutorial wrap-up and resources

Tutorial review

In this tutorial, you've done the following:

- Learned about JSR-101/JAX-RPC.
- Learned how to use the WebSphere Studio Application Developer Integration Edition tools to create RPC-style Web services.
- Created, deployed, and consumed a Web service using simple types.
- Created, deployed, and consumed a Web service using complex types.
- Created dynamic clients that used both simple and complex types.

You now have a good start toward developing with Web services. We've also shown that it is quite possible to develop with Web services using the specifications and tools available today. Some businesses are already using these kinds of Web
services for integrating their systems enterprise-wide, and doing so with partners, suppliers, and customers. Web services are starting to become a key technology in Enterprise Application Integration (EAI).

In later tutorials, we will explore more complex topics, such as our aggregator interfacing with two DVD rental providers as well as discovering those provider interfaces through UDDI from the aggregator.
<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
<th>Download method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source code for this tutorial</td>
<td>ws-soa-2code.zip</td>
<td>2945KB</td>
<td>HTTP</td>
</tr>
</tbody>
</table>

Information about download methods
Resources

Learn

- **Elements of Service-Oriented Analysis and Design** -- facilitate successful SOA deployments using Service-Oriented Analysis and Design (SOAD).
- Top-down Web service development: Build a WSDL file to generate a Web service using WebSphere Studio
- JAX-RPC (JSR-101)
- JSR-109 - Implementing Enterprise Web services
- SOAP 1.1 Specification
- UDDI Specification
- WS-I Basic Profile
- WSDL 1.1 Specification
- Standards roadmap -- understand the impact and importance of standards and specifications for the development of SOA and Web services.
- WSDL4J Project
- Apache Ant Project
- Take all parts of this tutorial series.
- UDDI4J Web site
- UDDI Home Page
- Web Services Interoperability Organization (WS-I)
- Oasis Web Services Security TC
- OASIS XML.org
- SOA and Web services -- hosts hundreds of informative articles and introductory, intermediate, and advanced tutorials on how to develop Web services applications.
- Architecture: Build for the future -- visit the Architecture area on developerWorks and get the resources you need to advance your skills in the architecture arena.

Get products and technologies

- WebSphere Studio Application Developer -- get a trial download.
- Eclipse.org

Discuss

- developerWorks blogs -- get involved in the developerWorks community.
About the authors

Warner Onstine
Warner Onstine, Senior Mentor at ArcMind, Inc., is a developer with more than 8 years of experience in the industry, the majority of that spent developing Web applications. Warner is co-author of the book *Professional Java Tools for Extreme Programming*, with chapters focusing on Maven, Swing unit testing, and code coverage using jcoverage.

Rick Hightower
Rick Hightower, Chief Mentor at ArcMind, Inc., is a developer with multiple achievements, industry awards, and certifications. Rick is co-author of the book *Professional Jakarta Struts, Java Tools to Extreme Programming*, and wrote 1/5 of the book, *Mastering Tomcat*. Rick wrote many well-received tutorials on EJB 2.0 CMP CMR, XDoclet, Apache Axis, ETTK, WSDK, Struts Tiles, and more for IBM developerWorks.