ApacheCon EU 2005: Tutorial DE1201

Open Standards/Open Source Java Web Services with Apache Geronimo

Duration: **3 hours**
Style: **Tutorial**
Level: **Experienced**
Audience: **Developer**
Categories: **Databases, Java, Java and Data, XML**
Speaker: **Tom McQueeney**

Abstract

Writing web services used to be a proprietary process. The lack of standards prompted web service framework vendors to develop their own component model. Programming web service required developers to learn and n lock themselves into a vendor's tools. The release of the J2EE 1.4 specification changed the landscape by offering the first standard component model for Java web services and client applications. Still, not many application servers were available in 2004 that supported the standards. Developers wanting to use the new standards were aided this year and in late 2004 by the release of three open source J2EE 1.4 application servers. Developers now have a choice between JBoss, JOnAS and Geronimo, as well as a handful of commercial servers, that all support the new web service specifications.

This presentation focuses on introducing Java developers to the J2EE 1.4 web service component model and how standards-based web services can be deployed on the Apache Geronimo J2EE server. The presentation will show how to develop servlet- and EJB-based web services that comply with the Java web services specifications (JAX-RPC 1.1 and WSEE 1.1) and show three techniques for writing standards-based web service client applications. The presentation also will demonstrate how a properly written web service can be deployed with no code changes on two additional open source application servers: JOnAS and JBoss. With open source standards and open source servers, developers can now write web services with no up-front cost and with no vendor lock-in.

Contents

- What are web services?
- What is JAX-RPC?
- What is WSEE?
- How to write a J2EE (JAX-RPC) web service
- Example: Creating a JSE web service
- How to deploy a web service to Apache Geronimo
- How to deploy the same web service to JBoss and JOnAS and why not much changes
- How to write a JAX-RPC client application to call a web service
- Interoperable web services.
- Advance topics: holder classes, message handlers, session management

- Security: JAX-RPC requires support for HTTP Basic authentication only. All other mechanisms, like WS-Security, are vendor-specific.
- Future: J2EE 5.0 web services using metadata
- Appendix I: Resources for more information
- Appendix II: Speaker background

What are web services?

There is no standard definition for *web services*. A loose definition: "Sending a message to a remote application over the Internet by using HTTP, and optionally receiving a response message." Some definitions restrict the format of message that can be sent and received to XML, or even more restrictive like a SOAP XML message. Some definitions expand the protocols beyond HTTP to allow SMTP and other Internet protocols.

This tutorial will focus on SOAP web services messages sent over HTTP. This type of web service is the most common being used, and it is the type supported by J2EE 1.4.

Brief history of web services

Microsoft and IBM were the initial supporters for SOAP, an XML messaging format and transmission protocol that has become the de facto standard for web services. SOAP originally was an acronym for "simple object access protocol" but now just stands for *SOAP* to expand its meaning beyond object-oriented remote procedure calls, and maybe to acknowledge that it is no longer simple. SOAP was invented to try to succeed where CORBA failed. That is, a simpler way to perform remote-procedure calls across computer systems using multiple languages and operating systems.

A SOAP message is simple at a high level. It's an XML document containing a message "body" and an optional "header" that contains metadata to describe the message. The header and body are wrapped inside an XML "envelope" element.

If you think of SOAP as defining the message format for remote-procedure calls, another web services standard, called Web Services Description Language, describes the services being offered by a remote web service. WSDL describes the operations (method calls) available using language- and platform-neutral specifications at an interface level, plus defines ways that web service could be called using one or more "binding" methods that define message format transport protocols.

SOAP 1.1 and WSDL 1.1 were submitted to the World Wide Web Consortium in order to become Internet standards. The W3C accepted the SOAP and WSDL specifications as "notes" in 2000 and 2001, respectively, and began working on true recommended standards. But the fact that the W3C published the specifications pushed SOAP and WSDL as de facto standards. In 2003, the W3C published SOAP 1.2 as a recommended standard, and likely will publish WSDL 2.0 as a standard later this year.

What is JAX-RPC?

JAX-RPC was Java's first standard to define how web services client and server components should be written in Java. It stands for **Java APIs for XML-based Remote**

Procedure Calls, and as its name implies, it is RPC-centric rather than trying to specify a more general way to exchange XML messages.

Before JAX-RPC was specified, there was no standard way in Java to code or call a web service. With no standards, vendors released proprietary APIs and toolkits to support web services. Apache Axis was one of the proprietary Java toolkits to create and invoke web services. J2EE application server vendor also began offering web services support, using proprietary mechanisms to write and deploy web services.

Even though the Java toolkits were proprietary, they did have the advantage of allowing developers to call web services written in different languages and running on different platforms, because the SOAP and WSDL specifications defined interoperability standards. However, once you coded your Java web service "endpoint" or client application using one of these vendor toolkits, your code was locked into using that vendor's library, defeating one of the design goals of J2EE.

JAX-RPC defines the standard ways to write a web service client or server application that can be implemented by multiple vendors. JAX-RPC's goal is to allow developers not only to write a web service that can be called by software written in non-Java languages, but also to write web services that can run using any vendor's JAX-RPC compliant library.

The JAX-RPC inventors used a Java RMI programming style to model how web services are written and invoked. Java developers create a web service by designing a class that implements a Java remote interface. Writing a JAX-RCP web service client application involves using a remote interface to hide the web service. Because of its RMI nature, web service calls can cause a *java.rmi.RemoteException*, just like other remote Java calls, like EJB. JAX-RPC 1.1 is defined in terms of SOAP 1.1 and WSDL 1.1.

J2EE 1.4 incorporated JAX-RPC 1.1 as the model for writing and calling web services. JAX-RPC 2.0 (JSR-224), renamed in May to "**Java API for XML Web Services**," will be incorporated into J2EE 5.0. It is planned to support SOAP 1.2 and WSDL 2.0.

What is WSEE?

Web Services for J2EE expands upon the JAX-RPC specification by defining how web service client and server applications should be written for a J2EE container. It defines the J2EE web services programming model, expanding upon the servlet and EJB specifications, and describes how web services should be packaged and deployed to a J2EE 1.4 application server.

J2EE 1.4 introduced a new J2EE component type, called a JAX-RPC Service Endpoint, to implement Java web services, and defined how stateless session EJBs can be exposed as web service.

How to write a J2EE (JAX-RPC) web service

There are two common ways to write a J2EE web service. The first approach is to write a Java interface that defines the operations to be made available from the web service, then use a tool to generate a WSDL document to describe the web service. The second

approach is to write a WSDL document describing the web service, and then use a tool to generate the Java interface and a stubbed-out implementation class.

The WSDL document must be deployed with the web service in the WAR/JAR file.

In the examples for this tutorial, we will start with the Java interface and create the WSDL document from it. Starting from Java is the most common approach used by Java developers because it allows them to work with something comfortable: a Java interface. Starting with WSDL has the advantage of helping to create a language-neutral web service, such as avoiding overloaded methods (illegal in WSDL 1.1) and using INOUT operation parameters. However, writing WSDL is a lot harder than writing Java, and there are not as many open source and useful WSDL editing tools available.

EJB and JSE web services

The implementation of your web service, called an "endpoint," can either be a stateless session EJB or a POJO. The POJO endpoint is a new type J2EE object type, introduced in J2EE 1.4, called a **JAX-RPC Service Endpoint**, or JSE. The JSE runs in the web container much like a servlet and can be thought of as a servlet because you must code it using the same rules: not maintaining user state, making it thread safe.

Deciding which type of endpoint to create usually is as easy as answering the question of whether the web service functionality would normally be best implemented as a stateless session bean. That is, does it take advantage of EJB container services. If you want to expose an existing SLSB as a web service, then the decision has already been made.

My rules of thumb on using an EJB or a JSE to implement a web service endpoint:

Use an EJB if:

- You already have a SLSB you want to expose as a web service.
- Your web service implementation needs EJB container services, such as sharing transaction contexts with other EJBs or using method-level security restrictions.
- You don't want to worry about concurrent threading issues and you don't mind the extra coding and deployment effort inherent with EJBs.
  SLSBs are single threaded. JSE objects can be invoked by multiple threads, making concurrency an issue.
- Your implementation uses other EJBs
  If your web service will be calling other EJBs, you will be using the EJB container and probably its CMT transaction services. If your web container and EJB container are distributed, you already are taking the remote hit, so EJBs referencing other EJBs can be clearer in deployment descriptors

Use a JSE if:

- None of the above are relevant to your needs
  JSEs are easier to code and understand. They're POJOs, for the most part.
- You are an experienced Java web application developer.
  JSEs look and act a lot like servlets. They run in the web container and have access to the web container services, if needed. If you know how to code servlets, you will be comfortable writing JSEs.

- The thought of EJBs make you want to run and hide.
  JSEs really are easier to code. Use them whenever possible.

EJB and JSE web service endpoints both will be invoked by the J2EE 1.4 web container. The container provides a special web service servlet that receives the HTTP web service request, parses the SOAP message, unmarshals the XML into Java objects, finds the appropriate endpoint object to call for the given request, then invokes the appropriate method.

Deployment descriptors

J2EE 1.4 introduced two additional deployment descriptors needed for web services: *webservices.xml* and a jaxrpc-mapping.xml file. (You choose the latter's file name and define it in the *webservices.xml* file.) Since JSE web services share characteristics of a web application, they reuse the *web.xml* deployment descriptor. Web services are defined in a **<servlet>** section, even though they aren't ordinary servlets. EJB web services use the *ejb-jar.xml* deployment descriptor since they are also regular EJBs.

Here is a summary of what is included in the new deployment descriptors.

webservices.xml

The *webservices.xml* deployment descriptor defines the web services to be deployed in the J2EE application. It is packaged in the *WEB-INF* directory for JSE web services and *META-INF* directory for SLSB web services, alongside the *web.xml* and *ejb-jar.xml* deployment descriptors. It contains:

- Port's name: The logical name for each web service in the application.
- Port's bean class: The class that implements the port.
- Port's service-endpoint interface: The web service interface class
- Port's WSDL definition: A pointer to the WSDL file in the module
- Port's QName: The web service's XML qualified name in the WSDL
- JAX-RPC mapping file associated with each WSDL
- Handlers: Optional SOAP handlers (like servlet filters) to pre- and post-process SOAP messages.

jaxrpc-mapping.xml

The mapping file defines the relationships between the objects defined in the WSDL file and the Java objects defined in the Java implementation. This mapping file can be generated by some web services toolkits, and if you can find a tool that generates it for you, use it. The mapping file can be complicated to write by hand.

The file name can be called anything, although including the word "mapping" in it and ending the file with an ".xml" extension are common.

This tutorial will demonstrate how to create a JAX-RPC mapping file from a WSDL.

Example: Creating a JSE web service

Here are the high-level steps to create a JAX-RPC Service Endpoint web service.

1. Write Java interface defining the operations for the service.

2. Write and test the service's implementation class(es).
3. Generate/write WSDL and J2EE deployment descriptors.
4. Generate/write server-specific deployment descriptors, if needed.
5. Package the JSE as a WAR.

The first two steps should be familiar in developing any application. For Step 3, we will create a WSDL document using a tool provided by the Apache Axis project. Axis's **java2wsdl** tool is callable from a command-line interface and from a custom Ant task.

To create the J2EE deployment descriptors, we will use a combination of XDoclet and hand-coding. Unfortunately, open source toolkits haven't matured very far to generate good quality web service deployment descriptors. XDoclet is good at generating the web.xml deployment descriptor, and some of the server-specific deployment descriptors.

We will use Ant where possible to automate the build.

Here are the detailed steps. The example used in this tutorial is taken from the forthcoming book, *Geronimo Live*.

Step 1: Write a Java interface defining the web service

This interface describes a rudimentary event calendar service. In J2EE 1.4 web services terminology, this interface is referred to as the **Service Endpoint Interface** in deployment descriptors.

```
package com.geronimolive.gcal.ws;

import com.geronimolive.gcal.model.Event;

import java.rmi.Remote;
import java.rmi.RemoteException;


/**
 * A web service endpoint interface for the Event calendar.
 */
public interface EventService extends Remote {
    public Event[] getEvents() throws RemoteException;
    public Event getEvent(long eventId) throws RemoteException;
    public void saveEvent(Event event) throws RemoteException;
    public void removeEvent(long eventId) throws RemoteException;
}
```

The four Java methods will be exposed as web service *operations*, the language neutral web services term for method.

Step 2: Write and test the service's implementation class

Here is the implementation for the above interface.

```
package com.geronimolive.gcal.ws;

import com.geronimolive.gcal.dao.EventDAO;
```

```
import com.geronimolive.gcal.dao.EventDAOMemory;
import com.geronimolive.gcal.model.Event;
import com.geronimolive.gcal.service.DefaultEventManager;
import com.geronimolive.gcal.service.EventManager;

import java.util.List;

/**
 *
* @web.servlet
 *     name="Event web service"
 *     service-endpoint-package="com.geronimolive.gcal.ws"
 * @web.servlet-mapping url-pattern="/event"
 *
 * @wsee.port-component
 *     name="EventService"
 *     namespace-uri="http://ws.gcal.geronimolive.com"
 *     local-part="EventService"
 *     display-name="EventServiceJSE"
 *     description="A web service for accessing calendar events"
 * @author Tom McQueeney
 */
public class EventServiceWS implements EventService {

    private EventManager eventManager;

    public EventServiceWS() {
        EventDAO dao = new EventDAOMemory();
        eventManager = new DefaultEventManager();
        eventManager.setEventDAO(dao);
    }

    public Event[] getEvents() {
        List eventsList = eventManager.getEvents();
        return (Event[]) eventsList.toArray(
            new Event[eventsList.size()]
        );
    }

    public Event getEvent(long eventId) {
        return eventManager.getEvent(String.valueOf(eventId));
    }

    public void saveEvent(Event event) {
        eventManager.saveEvent(event);
    }

    public void removeEvent(long eventId) {
        eventManager.removeEvent(String.valueOf(eventId));
    }

}
```

Step 3: Generate/write WSDL and J2EE deployment descriptors

The JAX-RPC specification requires that every web service endpoint have a corresponding WSDL document. The container will publish each web service's WSDL document to a URL to make it available to external applications. The WSDL document defines the web service (service endpoint interface) using the platform-neutral WSDL language.

We will use Axis's *java2wsdl* tool to create the WSDL document. The Ant task to call the tool from our build file looks like this:

```
<target name="eventservice.java2wsdl" depends="-init,compile-server"
        description="Creates the WSDL file for the EventService"
>
    <mkdir dir="${generated.wsdl}" />
    <axis-java2wsdl
        output="${generated.wsdl}/EventService.wsdl"
        classname="com.geronimolive.gcal.ws.EventService"
        namespace="http://ws.gcal.geronimolive.com"
        style="rpc"
        use="literal"
     location="http://${axis.server}:${axis.port}/gcal/EventService"
    >
        <classpath refid="axis.class.path" />
    </axis-java2wsdl>

    <echo>Created WSDL at ${generated.wsdl}/EventService.wsdl</echo>

</target>
```

We will go over the generated WSDL at a high-level in the tutorial. My philosophy is if you have to look at or debug the WSDL, you have a problem with the tool that generated it or an interoperability issue. Both problems are beyond the scope of this tutorial.

The Ant build script we will use in the tutorial will generate the *web.xml* deployment descriptor, leaving us with the *webservices.xml* and JAX-RPC mapping files.

The webservices.xml deployment descriptor for our calendar event web service looks like this:

```
<webservices
    version="1.1"
    xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
        http://www.ibm.com/webservices/xsd/j2ee_web_services_1_1.xsd"
>
    <webservice-description>
        <display-name>Event Service</display-name>
        <webservice-description-name>
            EventServiceJSE
        </webservice-description-name>
        <wsdl-file>WEB-INF/wsdl/EventService.wsdl</wsdl-file>
        <jaxrpc-mapping-file>
```

```
            WEB-INF/event-jaxrpc-mapping.xml
        </jaxrpc-mapping-file>

        <port-component>
            <port-component-name>EventServiceJSE</port-component-name>
            <wsdl-port xmlns:gcalNS="http://ws.gcal.geronimolive.com">
                gcalNS:EventService
            </wsdl-port>

            <service-endpoint-interface>
                com.geronimolive.gcal.ws.EventService
            </service-endpoint-interface>

            <service-impl-bean>
                <servlet-link>Event web service</servlet-link>
            </service-impl-bean>
        </port-component>
    </webservice-description>
</webservices>
```

The *webservices.xml* deployment descriptor describes the web service to the container and points to the location in the WAR (or EAR) file where the container can find the WSDL document and the JAX-RPC mapping file. The *webservices.xml* file will be almost identical for JSE and EJB endpoints except for the <service-impl-bean> section. One will point to a servlet defined in the *web.xml* deployment descriptor, the other to an EJB defined in an *ejb-jar.xml* deployment descriptor.

The JAX-RPC mapping file can be very simple or very complicated, depending on the complexity of your web services interface. A simple mapping file will look something like this one:

```
<java-wsdl-mapping
    xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:gcal="http://ws.gcal.geronimolive.com"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
        http://www.ibm.com/webservices/xsd/j2ee_jaxrpc_mapping_1_1.xsd"
    version="1.1"
>
    <package-mapping>
        <package-type>com.geronimolive.gcal.ws</package-type>
        <namespaceURI>http://ws.gcal.geronimolive.com</namespaceURI>
    </package-mapping>
</java-wsdl-mapping>
```

A simple mapping file tells the container how to map between XML namespaces in the WSDL document and Java packages. A complicated mapping file is a lot more verbose, with several additional sections. You can use a simple JAX-RPC mapping file if your web service conforms to rules spelled out in Chapter 7 of the JSR 921 specification, *Implementing Enterprise Web Services 1.1*. Here is a summary of those rules, which will be meaningful to you if you have a good understanding of the WSDL 1.1 specification.

Your JAX-RPC mapping file needs to include only <package-mapping> sections if:

1. The WSDL file contains only one <service> element
2. The WSDL <service> element defines only one <port>
3. The <binding> the WSDL's <port> defines must be a SOAP 1.1 binding with style="rpc". Additionally, all its operations must specify use="encoded", encodingStyle="<the SOAP 1.1 encoding>" for their input, output, and fault messages, and either must omit the parts attribute or include it such that all the parts of both the input and output message are mapped to the SOAP body.
4. No SOAP headers or header faults can be specified in the WSDL binding.
5. Each <operation name="…"> must be unique and the name must follow Java method naming conventions, have exactly one input message, at most one output message
6. Faults must map to an exception that is NOT a runtime exception or remote exception
7. Each <part> must have a type="…" specifying a simple XML schema type, or an <xsd:complexType> using an <xsd:sequence>, <xsd:all>, or <xsd:restriction base="soapenc:Array"/> compositor.

If the WSDL fails to meet all of these restrictions (and a few more minor ones), you must supply a full JAX-RPC mapping file unless your J2EE container is sophisticated and can supply usable default mapping rules. Unfortunately, Geronimo, JBoss, and JOnAS do not supply default mapping files.

Also unfortunately, rule 3, above, says the web service must use the RPC/encoded style. This style violates the WSI's Basic Profile 1.0 on how to create an interoperable web service. More on interoperability, below. But for our needs, if we want to write a web service that is likely to work with non-Java languages, we must write the complicated version of the JAX-RPC mapping file.

We'll go over how to create the complicated mapping file in this tutorial.

Step 4: Generate/write server-specific deployment descriptors

For many web services applications deployed to Geronimo, JBoss, and JOnAS, the only server-specific deployment descriptor is the usual one used for web applications when you want to define your own application context URL. All three open-source servers supply default application contexts based on the name of the deployed WAR file. You need to supply (or generate) an application-server specific web application deployment descriptor only if you don't like the default context URL.

Step 5: Package the JSE as a WAR

To create a J2EE 1.4 deployable web service, all application class files, library JARs and deployment descriptors need to be JAR'd into a WAR file. For a JSE endpoint, the *webservices.xml* descriptor must go in the WEB-INF directory. The container will look for it there.

The WSDL file should be stored in the *WEB-INF/wsdl* directory. This location is recommended but not required by the WSEE 1.1 specification. The *webservices.xml* descriptor defines the actual location of all WSDL files in the <wsdl-file> element.

How to deploy a web service to Apache Geronimo

The tutorial will show how to deploy web services to Apache Geronimo. We will use either a reusable Ant target via an <antcall> Ant task, or use a better deployment technique if available from the Geronimo project. Deployment tools have been one of the lagging areas of the Geronimo project.

Here is an Ant task I wrote that calls the Geronimo supplied command-line deployer.jar tool:

```
<target name="deploy-jar"
    description="Deploy to Geronimo. Define property 'jar'"
>
    <fail unless="jar" message="Use -Djar=&lt;path-to-jar&gt;" />

    <java jar="${geronimo.home}/bin/deployer.jar" fork="true">
        <arg value="--user" />
        <arg value="${geronimo.user}" />
        <arg value="--password" />
        <arg value="${geronomo.password}" />
        <arg value="deploy" />
        <arg value="${jar}" />
    </java>
</target>
```

How to deploy the same web service to JBoss and JOnAS

The tutorial will demonstrate deploying the same web service to JBoss and JOnAS with almost no changes. JBoss's web services support is good. JOnAS's support is a little flaky as of version 4.4.3. The tutorial will cover some of the JOnAS quirks.

How to write a JAX-RPC client application to call a web service

The Java client model to invoke a web service is defined by the JAX-RPC specification. JAX-RPC 1.1 defines three styles of writing web services clients. The three styles can be used by web service clients running inside and outside a J2EE container.

The three JAX-RPC client styles are:

1. Generated stub
2. Dynamic proxy
3. Dynamic invocation interface

Here is a summary of these techniques. Each will be demonstrated in the tutorial.

Generated stub

The generated stub technique is conceptually the easiest because it hides all JAX-RPC specific classes from your client code. "Generated stub" means you use a vendor tool to generate several Java helper classes at *development time* that know how to call the web service. The tool uses the web service's WSDL document to generate the helper Java classes.

One of the helper classes is the web services service endpoint interface, which will be remote interface. Your client source code will be coupled only to the remote interface

and, if your client runs outside a J2EE container, a helper service-locator class that will return an instance of the remote interface.

During development, you generate the helper Java source files, then write your web service client class to use these generated classes. You compile the generated helper classes as well as your own client classes, and deploy them all together.

Your deployed client will be coupled to the vendor's library/classes that provide the helper classes. In the case of Apache Axis, this is the *jaxrpc.jar* library. You replace the vendor's JAX-RPC library with another one, but you will need to regenerate the helper classes and recompile your client code. Your client source code does not need to be modified, since it uses only interfaces and a generic POJO service locator class.

Dynamic proxy

The dynamic proxy technique is similar to the generated stub technique , except its purpose is to decouple your client code from any one vendor's JAX-RPC library. The price you pay is that your client code must use JAX-RPC specific classes instead of being code to more simple remote interfaces.

Like the generated stub technique, you still code your client to call the web service using a remote interface, the service endpoint interface. However, your client also uses a JAX-RPC class instead of service locator. The JAX-RPC class will generate the proxy helper classes (that know how to call the web service) at *run time*.

The advantage to the dynamic proxy technique is you can replace a vendor's JAX-RPC library with another one and not have to recompile your client. It provides flexibility to change web service client libraries at the cost of coupling your application to JAX-RPC and some performance hit as the proxy classes are generated.

Dynamic invocation interface

The dynamic invocation interface technique couples your client strongly to the JAX-RPC package. Your client directly uses several JAX-RPC classes to set the web service's URL, the operations you want to call, the parameters to pass to the web service, the expected return type, and any XML serializers or deserializers that will be needed for the web service call.

The advantage of the DII technique is your client needs to know nothing about the remote web service during development. All information about the remote web service can be set at run time. No WSDL document is needed to invoke the web service. The disadvantage to DII is your client application is tightly coupled to JAX-RPC. You can no longer code your client application to call a web service as if it were any other remote object.

The DII technique is great for tools that need to discover details about a web service at run time, and dynamically invoke it. The technique is also useful for quickly writing a test application to invoke a web service. You don't need to pre-generate any helper classes or use a remote Java interface.

Interoperable web services

One of the best techniques you can use to help ensure your web service endpoint can be called from clients written in different languages and running on different platforms is to use only XML schema types as parameters and return types. The JAX-RPC specification defines what Java types should map to XML schema types.

Here are the Java types considered safe to use for maximum interoperability, and what schema type they will be mapped to in the SOAP message. In addition to the values in the table, below, the Java primitive types will map to their equivalent object wrapper class (e.g. java.lang.Integer) if the XML element type is declared to be nillable="true". Using the object wrapper is the only way the null value can be represented.

Simple XML Type	**Java Type**
xsd:string	java.lang.String
xsd:integer	java.math.BigInteger
xsd:int	int
xsd:long	long
xsd:short	short
xsd:decimal	java.math.BigDecimal
xsd:float	float
xsd:double	double
xsd:boolean	boolean
xsd:byte	byte
xsd:unsignedInt	long
xsd:unsignedShort	int
xsd:unsignedByte	short
xsd:QName	javax.xml.namespace.QName
xsd:dateTime	java.util.Calendar
xsd:date java	util.Calendar
xsd:time java	util.Calendar
xsd:anyURI	java.net.URI or java.lang.String
xsd:base64Binary	byte[]
xsd:hexBinary	byte[]
xsd:anySimpleType	java.lang.String

For maximum interoperability, dozens of major web services tool vendors and application service providers created the Web Services Interoperability Organization to come up with standards they agree to follow so their web services will work with each other's. WSI also provides tools you can use to help verify your web service will be interoperable with other languages, tools, and platforms that conform to its Basic Profile 1.0.

The WSI tools can tell you if your web service is likely to be *non*-compliant, but it isn't able to definitively say your web service is interoperable. That's because no one has

discovered a good way to prove a web service will be compliant under all use cases of incoming and outgoing SOAP messages.

Advanced topics:

- Holder classes
  Used for "out" and "inout" parameters. We probably will not cover this.
- Message handlers
  Like servlet filters, but can be used on client and server side to manipulate the message before and after it gets sent to the implementation class. Can use handlers for specialized encryption/decryption of messages, logging, auditing, billing.
- Session management
  Because of the stateless nature of HTTP, the stateless nature of session beans, and the stateless nature of SOAP, the J2EE 1.4 web service specifications define no standard way to maintain state between web service calls. And the fact that the web service call is implemented by a shared stateless object (the JSE or SLSB), techniques for maintaining client state are left up to the developer, using the usual web techniques. Maintaining state usually is implemented much like servlets, using SSL sessions or cookies or another "handback" string passed back and forth between the client and the endpoint.

Security

JAX-RPC requires support for HTTP Basic authentication only. All other mechanisms, like WS-Security, are vendor-specific.

Future: J2EE 5.0 web services using metadata

- JSR 175: Java metadata facility (J2SE 5.0)
- JSR 181: Metadata for web services (J2EE 5.0)

Definitions

Here are definitions of J2EE 1.4 terms used in this document.

- **Endpoint**: the server-side implementation of a web service.
  In J2EE, endpoints are coded as a POJO that runs in the web container or as a stateless session bean that runs in the EJB container.
- **JAX-RPC** 1.1: Java API for XML-based Remote Procedure Calls.
  JAX-RPC defines a client and server remote procedure call programming model for web services. It also defines how to map web service definitions (WSDL) and XML data into Java classes.
- **Handler:** a web services SOAP filter.
  Handlers can be set up like servlet filters but receive and process a web service SOAP message rather than a HTTP request message.
- **Holder**: A Java value object that represents special web service parameter types.

Web services may accepts parameters meant only to return values, and parameters that can pass in a value but also be changed by the web service operation. Since Java method parameters can do neither, these web service parameter types must be represented by special classes that can return values.

- **Servant**: a JAX-RPC service endpoint class or a SLSB exposed as a web service.
- **SEI**: a Service Endpoint Interface
  Every J2EE 1.4 web service endpoint must be defined via a Java interface.
- **WSDL port type**: The interface defining the services (ports) available from the web service.
- **WSEE** 1.1: Web Services for J2EE
  WSEE builds upon JAX-RPC to define a web services client and server programming model for J2EE containers.

Appendix I: Resources for more information

Here are reference to books and articles on Java web services.

Specifications:

- JAX-RPC 1.1: jcp.org/en/jsr/detail?id=101
- WSEE 1.1: jcp.org/en/jsr/detail?id=921
- SOAP 1.1: http://www.w3.org/TR/2000/NOTE-SOAP-20000508/
- WSDL 1.1: http://www.w3.org/TR/wsdl
- Basic Profile 1.0, www.ws-i.org/Profiles/BasicProfile-1.0-2004-04-16.html

Articles:

- Deploy Web services in Apache Geronimo
  http://www-128.ibm.com/developerworks/opensource/library/os-ag-wsvs/
- "Build interoperable Web services with JSR-109"
  www-128.ibm.com/developerworks/webservices/library/ws-jsrart/
- Developer's introduction to JAX-RPC, Part 1: Learn the ins and outs of the JAX-RPC type-mapping system
  www-106.ibm.com/developerworks/webservices/library/ws-jaxrpc1/
- "… Part 2: Mine the JAX-RPC specification to improve Web service interoperability"
  www-106.ibm.com/developerworks/webservices/library/ws-jaxrpc2/

Books:

- "J2EE Web Services" by Richard Monson-Haefel (Addison-Wesley).
  This is a great book for understanding the Java web services specifications. This book focuses on theory. As such, it lacks practical examples and step-by-step tutorials.
- "Designing Web Services with the J2EE Platform" by Inderjeet Singh et al. (Addison-Wesley)
  java.sun.com/blueprints/guidelines/designing_webservices/html/index.html

Appendix II: Speaker background

Tom McQueeney has been designing and coding Java applications since about 1997 and has been writing Java web services applications since 2000, before they commonly were called *web services*. He works as a Java developer, architect, mentor and trainer for Idea Integration, a nationwide consulting company in the United States.

He is writing a book on the Apache Geronimo server, *Geronimo Live*, expected to be published later this summer by Sourcebeat, a publisher focusing on open-source software projects. In 2004, Tom served as president of the Denver Java Users Group in Colorado, named by Sun Microsystems as one of the top 25 Java users groups in the world.

When he leaves the ApacheCon conference, Tom will be packing to move from Denver to Washington, DC.