Understanding web services specifications, Part 1: SOAP

Nicholas Chase

May 12, 2006

The current emphasis on Service-Oriented Architectures (SOA) has put the spotlight on web services, but it's easy to get lost in all the information being bandied about. This series gives you the straight story on all of the major web service specifications, starting with Simple Object Access Protocol (SOAP) and working down to WS Business Process Execution Language (WS-BPEL). This tutorial explains the basic concepts of web services and of SOAP, and explains how to build a SOAP server and client.

View more content in this series

Before you start

This tutorial series chronicles the building of a web services-based workflow system at a small fictional newspaper, the Daily Moon. It is for developers who want to learn more about the concepts that underlie web services, so that they can more efficiently create applications for them. Non-technical users will also some find value in these tutorials, as they explain concepts before jumping into programming.

You should have a basic understanding of programming, and, if you want to follow along with the actual programming examples, a grasp of Java. We will be talking about XML, out of necessity, but we won't be getting deep into it, and any necessary concepts will be covered.

About this series

This tutorial series teaches the basic concepts of web services by following the exploits of the fictional newspaper, the Daily Moon, as the staff uses web services to create a workflow system to increase productivity in these turbulent times.

This first part starts simply, explaining the basic concepts behind web services and showing you how to use SOAP, the specification that underlies most of what is to come, connecting the classifieds department with the Content Management System.

Future installments in this series will build upon the basic concepts:
Part two takes things a step further, explaining how to use Web Services Description Language (WSDL) to define the messages produced at expected by web service, enabling the team to more easily create services and the clients that connect to them.

Part three finds the team with a number of services in place and a desire to locate them easily. In response, Universal Description, Discovery and Integration (UDDI) provides a searchable registry of available services at a way to publicize their own services to others.

Parts four and five, WS-Security and WS-Policy, chronicle the securing of the paper's services and the changes the teams need to make in order to access those newly secured services.

Interoperability is the key word in part six, as services from several different implementations must be accessed from a single system. Part six covers the requirements and tests involved in WS-I certification.

Finally, part seven shows how to use Business Process Execution Language (WS-BPEL) to create complex applications from individual services.

Now let's look at what this tutorial covers in a bit more detail.

About this tutorial

This tutorial introduces you to the concept of web services, and to the team at our fictional Daily Moon Classifieds Department. You will follow along as the team integrates with an existing web services system, and witness the creation of a service. The focus will be on the Service Object Access Protocol (SOAP).

During the course of this tutorial, you will learn the following:

- The basic concepts of web services
- The basics of XML
- The structure and purpose of a SOAP message
- How to install an application server on which to run your web service applications.
- How to install a web services implementation into an application server
- How to programmatically create a SOAP message.
- How to create a client for a SOAP-based web service using Java and Apache Axis2
- How to create a SOAP-based web service Java and Apache Axis2

In this tutorial, the Classifieds Department will integrate with the content management system, creating a client. You'll also get a look at the process of creating one of the services with which the department interacts. Programming examples are shown in Java using the Apache Axis2 project, but the concepts apply to virtually any other language and environment.

Prerequisites

In order to follow along with the code for this tutorial, you will need to have the following software available:

- Apache Geronimo or another application server. You will be creating various web services throughout the course of this tutorial, and you will need an application on which to run them.
Because web services are, of course, supposed to be interoperable, it doesn't really matter which one you use. In this tutorial, we will demonstrate the installation and use of Apache Geronimo, which is also the basis for IBM® WebSphere® Community Edition. You can also use other application servers such as WebSphere Application Server. You can download Apache Geronimo from the Apache Geronimo Downloads site.

- Apache Axis2 or another SOAP implementation. You can create SOAP messages by hand, and you can interpret them by hand, but it's much easier to have an implementation handy. You will be using Apache Axis2, which contains implementations of various SOAP-related APIs to make your life significantly easier. You can download Apache Axis2 at: Apache.org. This tutorial uses version 0.94, but later versions should also work.

- Java™ 2 Standard Edition version 1.4 or higher. Both of these tools are Java-based, as are the services and clients you'll build in this tutorial. You can download the J2SE SDK from here.

- You'll also need a Web browser and a text editor, but I'm sure you already have those. If you like, you can also use an IDE such as Eclipse, but because the focus is on the technologies rather than the tools, I'll just be using a text editor and the command line to edit files and compile them.

What are web services?

Let's start by looking at the overall view of what web services actually are, and why they're important to software development.

What's the big deal, anyway?

If you hadn't been hearing lots and lots of information about Services-oriented architecture (SOA) and web services, you wouldn't be here, so the question is, why is this such a big deal? The answer is that it's a big deal because it's a paradigm shift in the way applications communicate with each other. SOAs have been around for a long, long, time. Originally they mostly consisted of middleware applications in which a single type of middleware owns, at the very least, both ends of the wire. web services, on the other hand, consist of a group of standards intended to make it possible for diverse systems to communicate, without requiring a particular type of middleware, programming language or even operating system. Let's look at the progression from where we started out to where we are now.

Traditional applications

In the beginning, there were computers. And it was good. Computers performed seemingly miraculous tasks, automating many of the things that people did by hand, starting with complex calculations, and moving to financials, and many other tasks.

But traditional applications are "silos". The human resources application couldn't really talk to the financials application, which couldn't really talk to the distribution application. All of these applications had their own home, on their own computer, and while they were useful, there wasn't a good way to share data between them. You had the option to write batch processes to move data from one system to another, but that was no substitute for real-time integration.
Distributed computing

The next step in our evolutionary chain is distributed computing. Distributed computing enabled different applications to talk to each other, even if they weren't on the same computer. Technologies such as CORBA, MTS, and Enterprise Java Beans (EJB), provided a system that included a registry of sorts so that applications could find components with which they wished to interact, and then call these components as though they were located on the local machine.

These systems were supported by middleware, or more specifically, message-oriented middleware, which provided both of these requirements. Applications could now be built in such a way that they could access resources on other systems, even if they were in different geographic locations.

But there was still a problem. While applications were free to communicate anywhere within the system, it was still a closed system. At the very least, your client application had to use the same technology as the server application. Also, the systems were not designed, as a rule, for access from outside of the individual organization that had created them.

Web services

The next, almost inevitable link in this evolutionary chain is web services. Based on XML, and, in most cases, HTTP, "web services" still means many things to many people, but in this case, we're going to talk about web services as the exchange of SOAP-based messages between systems.

These messages are composed of XML, which is a text-based open standard, accessible by anyone from any application (any application that's been designed to accept it). This expands the world for your application to include anyone who can reach it over your network. (If that sets off security bells for you, that's OK, you'll learn how to deal with that in part four of this series.)

A SOAP-based web service involves the sending of an XML message such as shown in Listing 1.

Listing 1. A SOAP-based web service

```xml
<SOAPenv:Envelope
 xmlns:SOAPenv="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <SOAPenv:Body>
  <req:getNumberOfArticles xmlns:req="http://daily-moon.com/CMS/"
   xsi:type="req:Article">
   <req:category>classifieds</req:category>
  </req:getNumberOfArticles>
 </SOAPenv:Body>
</SOAPenv:Envelope>
```

These messages move from one system to another, usually via HTTP. The receiving system interprets the message, does what it's supposed to do, and sends back a response in the form of another SOAP message.

It is a simple system, and as such, there are many aspects of enterprise-level computing that are not covered by it. Fortunately, many of these aspects have been taken into consideration, and
have their own specifications to determine how this transaction should take place to incorporate many of the security and other aspects of traditional message-oriented middleware.

**Other kinds of web services**

I would be remiss if I didn't mention that SOAP is not the only way to do web services. There are other XML-based means for sending messages between systems, some of which are applicable to an enterprise environment, and some of which are not. For example, Amazon was one of the first Web-based companies to provide web services access to its system to the public. Amazon includes a SOAP-based service, but it also provides a service based on Representational State Transfer (REST).

REST is a type of web service in which the user simply accesses a URL, and the response is a straight XML document such as the one shown in Listing 2.

**Listing 2. A REST response**

```xml
<currentArticles>
    <category>classifieds</category>
    <subcategory>forsale</subcategory>
    <article id="888204">
        <articleHeadline></articleHeadline>
        <articleText>30 ft ladder, only used once. Willing to let go for half it's worth. Has slight dent near the middle. Harder than a human head. $150 OBO.</articleText>
    </article>
    <article id="888242">
        <articleHeadline></articleHeadline>
        <articleText>Vintage 1963 T-Bird. Less than 300 miles. Driven by my daughter until I took it away. Serious inquires only. 555-3264 after 7 PM.</articleText>
    </article>
</currentArticles>
```

There is no particular format to these messages. It is just whatever the data happens to be.

Another type of web service involves the use of a standard such as XML-RPC. In this case, commands are sent to a system via XML such as shown in Listing 3.

**Listing 3. XML-RPC**

```xml
<?xml version="1.0"?>
<methodCall>
    <methodName>CMS.getNumberOfArticles</methodName>
    <params>
        <param>
            <value><string>classifieds</string></value>
        </param>
        <param>
            <value><string>forsale</string></value>
        </param>
    </params>
</methodCall>
```

The response follows a similar format.
As you're learning to use SOAP, in the back of your mind you may be thinking that REST and XML-RPC are much simpler than a SOAP-based system. And you're right. In some ways, they are. However, we are not talking about a simple application to display the weather on your web site. We're talking about enterprise-level applications here, and enterprise-level applications need enterprise-level attributes such as security, interoperability, and so on. These capabilities are covered by additional specifications that have sprung up around SOAP-based web services, which makes SOAP a better choice for enterprise-level applications in the long run.

Let's look at some of these specifications.

**Basic web services specifications**

Web services specifications typically fall into two categories: basic web service specs, and expanded web service specs. The basic specifications are:

- **SOAP**: The foundation of all SOAP-based web services, the SOAP specification details the format of the actual messages. It also details the way applications should treat certain aspects of the message, such as elements in the "header", which enable you to create applications in which a message is passed between multiple intermediaries before reaching its final destination. This tutorial will cover the SOAP specification.

- **WSDL**: Web Services Description Language is a specification that details a standard way to describe a SOAP-based web service, including the form the messages should take, and where they should be sent. It also details the response to such a message. WSDL, when combined with the appropriate tools, enables you to create a call to a web service programmatically without ever actually knowing what the web service is looking for; the application can extract those details from the WSDL file and provide you with programmatic interfaces to use. We'll cover WSDL in part two of this series.

- **UDDI**: Universal Description, Discovery and Integration is a standard that has undergone somewhat of a change since its initial inception. The idea was to provide a way for companies to register their services in a global registry, and search that global registry for services they may be interested in using. However, because many companies are understandably reticent about opening their systems to outsiders, this goal hasn't quite materialized. However, UDDI has taken hold as an internal registry of services and service information; part three of this series details its use.

Those are the basics. There are also literally dozens of extended standards to make SOAP-based services more useful.

**Extended web services specifications**

Of the dozens all of WS-* specifications floating around, several distinguish themselves as being particularly useful to the enterprise. They are:

- **WS-Security**: This specification handles encryption and digital signatures, enabling you to create an application in which messages can't be eavesdropped, and in which non repudiation as possible. Part four of this series covers WS-Security.

- **WS-Policy**: This specification expands on WS-Security, enabling you to more specifically detail how and by whom a service can be used. Part five of this series covers WS-Policy.
• **WS-I**: Although web services are supposed to be designed for interoperability, in actuality there is enough flexibility in the specifications that interpretations between different implementations can cause problems. WS-I provides a set of standards and practices to prevent the sorts of problems, as well as standardized tests to check for problems. WS-I is the subject of part six of this series.

• **WS-BPEL**: a single service is nice, but in most cases it is hardly an application. At the very least, enterprise-level computing requires you to compose multiple services into an overall system, and WS-BPEL provides a way to specify interactions such as branching and concurrent processing that are necessary for creating such systems. Part seven of this series covers WS-BPEL.

Other specifications that play an important role in web services are not covered in this series including **WS-ReliableMessaging**, which enables you to be certain that one and only one copy of a message has been received, and that it has definitely been received; **WSRF**, the Web Services Resource Framework, which enables you to use state in what is essentially a stateless environment; and **Web Services Distributed Management (WSDM)**, which discusses the issue of management of and using web services.

**What we're going to accomplish**

In this tutorial, you will follow the classifieds department of the *Daily Moon* newspaper as they investigate integrating their own systems with the web services-based interface used by the content management system. You will create an application that creates a SOAP-based message, sends it to the service, and receives a response. Having done that, you will see how to create a service that responds to requests and sends a response of its own. You'll do this by creating Java applications, with and without an application server.

**Setting up**

Now that you understand the basic principles involved, let's start looking at actually creating an application. The first step is to get the software in place.

**Setting up Apache Geronimo**

The first piece of software you're going to need is a web application server. Why do you need a Web application server? Well, because you really are going to find it difficult to serve web services without one. A Web application server listens for requests, translates those requests into something the actual service can understand, and then does any necessary processing.

You can actually use virtually any web server for this process, but in most cases you're going to install software that makes the process easier, and that often requires an actual application server of some type. Specifically, this tutorial assumes you're going to use a Java application server. (Actually, it assumes a J2EE application server.)

You have many choices when it comes to J2EE servers, and in this case, you're going to use Apache Geronimo. Geronimo is the open source application server that forms the basis of IBM WebSphere Application Server Community Edition. Geronimo is small, easy to install, and easy to manage. It also works well with other Apache projects, such as Axis2, which you'll install next.
Download the software (see Prerequisites) and extract the files into a target directory. You will find that the extracted files have their own directory, so you can simply unpack them and move them wherever you'd like. Any directory is acceptable, but you want to avoid those with a space in their name, such as "Program Files", "Documents and Settings", or their descendants. For example, the test installation for the tutorial uses e:\geronimo-1.0.

There. You've installed Geronimo. Wasn't that easy?

To start the server, open a command prompt window and execute the following commands:

```bash
cd <GERONIMO_HOME>
java -jar server.jar
```

To make sure the server is running, open a browser and point it to the URL shown in Listing 4:

**Listing 4. Server URL**

http://localhost:8080

You should see a page something like Figure 1.

**Figure 1. Server working**

Now let's install the web services software.

**Installing Apache Axis2**

It is entirely possible to serve web services from a plain HTTP server. It is not, however, advisable. There is a lot to do processing SOAP messages, and there's no reason for you to reinvent the wheel. The Apache Axis project has been making this task simpler for several years now, creating an environment in which it is easy to create and process web services. The software includes
applications that help you create a web service from a plain object, create a Java object from web service, and process both. The Apache group has instituted a new version of Axis, Axis2, which takes all the work done on Axis and ratchets it up a notch by changing the architecture to allow it a greater degree of extensibility. This is important, because web service specifications are springing up all of the time. The construction of Axis2 enables it to more easily integrate with projects such as WSS4J, Apache's WS-Security implementation. Because we'll be using these projects later, go ahead and instal Axis2 now (see Prerequisites for download information).

Make sure to download both the Binary Distribution and the War Distribution. The former will help in building clients, the latter in building services.

To install Axis2 into the web server, copy the axis2.war file to Geronimo’s deploy directory. (You'll need to make sure you've started Geronimo at least once for the directory to be present.) Geronimo detects its presence automatically, so you don't have to manually deploy anything.

**Verifying the axis to installation**

To make sure that everything has been installed properly, point your browser to http://localhost:8080/axis2/index.jsp and click Validate.

You should see a page like that shown in Figure 2.

**Figure 2. Happy axis**

Make sure that Axis is happy before proceeding with the rest of the tutorial.

**Installing a sample service**

The first application you're going to write is a client, so you'll need a service for it to access. Fortunately, the Axis2 distribution comes with several. To start with, you'll install the MyService service as follows:
1. Authenticate yourself to the Axis2 application by pointing your browser at http://localhost:8080/axis2/Login.jsp and logging in. The default username and password are admin and Axis2, respectively.
2. Click Upload service>Browse.
3. Navigate to the MyService.aar file. You can find it in the samples\userguide directory of the standard Axis2 distribution. Click OK.
4. Click Upload.

You should see a notification that the service has been added (see Figure 3). By default, Axis2 includes to "hot deployment", so you shouldn't have to do anything to activate it.

Figure 3. MyService.aar has been uploaded

Now, let's look at what it is you're going to build.

Understanding SOAP

Now that you've installed the software, you can start looking at the actual web service. As I mentioned in Other kinds of web services, you have several formats to choose from. In this series, you will use SOAP.

A quick aside on XML

All of these messages flying back and forth are based on Extensible Markup Language, or XML. If you're not familiar with XML at all, you really should do a little bit of research before getting deep into web services topics. However, here are the basics you need to know before you move forward with this tutorial.

XML is a "markup language", which means that it provides a way to give additional information about the actual content. This information comes in the form of "tags", which denote "elements." For example, consider this simple XML document shown in Listing 5:
**Listing 5. An XML file showing the basics**

```xml
<article articleId="88271" categoryId="classifieds" subcategoryId="forsale">
  <articleHeadline>Fun, fun, fun</articleHeadline>
  <articleText>Vintage 1963 T-Bird. Less than 300 miles. Driven by my daughter until I took it away. Serious inquires only. 555-3264 after 7 PM.</articleText>
</article>
```

Notice a couple of things about this text. First of all, it is text. That makes it readable by just about anybody or just about anything. Second, tags are denoted with the > and <, with an "open" tag having a name, and possibly attributes such as the article ID, and a closing tag starting with a slash (/). Elements must be self-contained and nested properly. In other words, you couldn't have an XML document that looks like the one shown in Listing 6.

**Listing 6. An invalid sample XML document**

```xml
<article articleId="88271" categoryId="classifieds" subcategoryId="forsale">
  <articleHeadline>Fun, fun, fun</articleHeadline>
  <articleText></articleHeadline>Vintage 1963 T-Bird. Less than 300 miles. Driven by my daughter until I took it away. Serious inquires only. 555-3264 after 7 PM.</articleText>
</article>
```

XML also provides a way to separate content into different "namespaces", so it can be treated differently by an application. For example, a SOAP message might look like the following in Listing 7.

**Listing 7. An example SOAP message**

```xml
<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/SOAP-envelope">
  <env:Header>
  </env:Header>
  <env:Body>
    <cms:getNumberOfArticles xmlns:cms="http://www.daily-moon.com/cms">
      <cms:category>classifieds</cms:category>
      <cms:subcategory>forsale</cms:subcategory>
    </cms:getNumberOfArticles>
  </env:Body>
</env:Envelope>
```

Don't worry about the actual structure of the message, but notice that there are two different "prefixes", each of which corresponds to a particular namespace. In this case, we are distinguishing the SOAP "envelope" from the actual payload.

Again, there's a lot to learn about XML, but those are the basics you need to know for this tutorial.

**The SOAP envelope**

The basic unit of a web service message is the actual SOAP envelope. This is an XML document that includes all of the information necessary to process the message (see Listing 8).
Listing 8. An example SOAP envelope

```xml
<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/SOAP-envelope">
  <env:Header>
    </env:Header>
  <env:Body>
    </env:Body>
</env:Envelope>
```

In this case, you have a simple Envelope, with the namespace specified as SOAP version 1.2. It includes two sub elements, a Header and a Body. Let's look at what each of those pieces do.

The SOAP header

The Header in a SOAP message is meant to provide information about the message itself, as opposed to information meant for the application. For example, the Header might include routing information, as it does in this example shown in Listing 9.

Listing 9. Routing information in the Header

```xml
<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/SOAP-envelope">
  <env:Header>
      <wsa:Address>
        http://schemas.xmlsoap.org/ws/2004/08/addressing/role/anonymous
      </wsa:Address>
    </wsa:ReplyTo>
    <wsa:From>
      <wsa:Address>
        http://localhost:8080/axis2/services/MyService
      </wsa:Address>
    </wsa:From>
    <wsa:MessageID>ECE5B3F187F29D28BC11433905662036</wsa:MessageID>
  </env:Header>
  <env:Body>
    </env:Body>
</env:Envelope>
```

In this case you see a WS-Addressing element, which includes information on where the message is going and to where replies should go. The Header and can include all kinds of information about the message itself. In fact, the SOAP specification spends a great deal of time on elements that can go in the Header, and how they should be treated by "SOAP intermediaries". In other words, the SOAP specification makes no assumption that the message is going straight from one point to another, from client to server. It allows for the idea that a SOAP message might actually be processed by several intermediaries, on its way to its final destination, and the specification is very clear on how those intermediaries should treat information they find in the Header. That discussion is beyond the scope of this tutorial, however. So for now, just understand that the Header can provide you much more functionality if you need it.

Now let's look at the actual payload.

The SOAP body

When you're sending a SOAP message, you're doing it with a reason in mind. You are trying to tell the receiver to do something, or you're trying to impart information to the server. This information is
called the "payload". The payload goes in the Body of the Envelope. It also has its own namespace, in this case corresponding to the content management system. The choice of namespace, in this case, is completely arbitrary. It just needs to be different from the SOAP namespace (See Listing 10).

Listing 10. A payload example

```xml
<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/SOAP-envelope">
  <env:Header>
    ...
  </env:Header>
  <env:Body>
      <cms:category>classifieds</cms:category>
      <cms:subcategory>forsale</cms:subcategory>
      <cms:articleHeadline>Vintage 1963 T-Bird. Less than 300 miles. Driven by my daughter until I took it away. Serious inquiries only. 555-3264 after 7 PM.</cms:articleHeadline>
      <cms:articleText>Vintage 1963 T-Bird. Less than 300 miles. Driven by my daughter until I took it away. Serious inquiries only. 555-3264 after 7 PM.</cms:articleText>
    </cms:addArticle>
  </env:Body>
</env:Envelope>
```

In this case, you have a simple payload that includes instructions for adding an article to the content management system for the Daily Moon.

The choice of how to structure the payload involves the style and encoding.

Style and encoding

You'll get deeper into this subject in part two, which talks about Web Services Description Language (WSDL), but as you create your application, you will need to decide on the structure of the actual payload you're sending back and forth. To that end, let's take a moment to discuss programming style and encoding.

Simply put, two different programming styles for SOAP messages predominate. The first is the RPC style, based on the concept of using SOAP messages to create Remote Procedure Calls. In this style, the idea is that you're sending a command to the server, such as "add an article", and you're including the parameters for that command, such as the article to add and the category to which it should be added, as child elements of the overall method, as in Listing 11.

Listing 11. RPC style

```xml
<env:Envelope xmlns:env="http://www.w3.org/2003/05/SOAP-envelope">
  <env:Header>
    ...
  </env:Header>
  <env:Body>
      <cms:category>classifieds</cms:category>
      <cms:subcategory>forsale</cms:subcategory>
      <cms:articleHeadline>Vintage 1963 T-Bird. Less than 300 miles. Driven by my daughter until I took it away. Serious inquiries only. 555-3264 after 7 PM.</cms:articleHeadline>
      <cms:articleText>Vintage 1963 T-Bird. Less than 300 miles. Driven by my daughter until I took it away. Serious inquiries only. 555-3264 after 7 PM.</cms:articleText>
    </cms:addArticle>
  </env:Body>
</env:Envelope>
```

The alternative to the RPC style involves simply having your data as the content of the SOAP body, and including the information as to what procedure or function it pertains to in the routing of the message by the application server (see Listing 12).
Listing 12. Data as content in SOAP body

```xml
<env:Envelope xmlns:env="http://www.w3.org/2003/05/SOAP-envelope">
  <env:Header/>
  <env:Body>
      <cms:category>classifieds</category>
      <cms:subcategory>forsale</cms:subcategory>
      <cms:articleHeadline>Vintage 1963 T-Bird. Less than 300 miles. Driven by my daughter until I took it away. Serious inquires only. 555-3264 after 7 PM.</cms:articleHeadline>
      <cms:articleText>Vintage 1963 T-Bird. Less than 300 miles. Driven by my daughter until I took it away. Serious inquires only. 555-3264 after 7 PM.</cms:articleText>
    </cms:addArticle>
  </env:Body>
</env:Envelope>
```

One variation on the RPC style is RPC/encoded, as opposed to RPC/literal, as we see above. In that case, the message includes type information, such as that shown in Listing 13.

Listing 13. RPC/encoded example

```xml
<?xml version='1.0'?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/SOAP-envelope">
  <env:Header/>
  <env:Body>
      <cms:category xsi:type="xsd:string">classifieds</category>
      <cms:subcategory xsi:type="xsd:string">forsale</cms:subcategory>
      <cms:articleHeadline xsi:type="xsd:string" />
      <cms:articleText xsi:type="xsd:string">Vintage 1963 T-Bird. Less than 300 miles. Driven by my daughter until I took it away. Serious inquires only. 555-3264 after 7 PM.</cms:articleText>
    </cms:addArticle>
  </env:Body>
</env:Envelope>
```

The second style is known as the document/literal style, which involves simply adding the appropriate data to the message, as that shown in Listing 14.

Listing 14. Document/literal style example

```xml
<?xml version='1.0'?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/SOAP-envelope">
  <env:Header/>
  <env:Body>
    <category>classifieds</category>
    <subcategory>forsale</subcategory>
    <articleHeadline>Vintage 1963 T-Bird. Less than 300 miles. Driven by my daughter until I took it away. Serious inquires only. 555-3264 after 7 PM.</articleHeadline>
  </env:Body>
</env:Envelope>
```

In this case, the message itself doesn't include information on the process to which the data is to be submitted; that is handled by the routing software. For example, all calls to a particular URL
or endpoint might point to a particular operation. Also, you could technically use the document/encoded style, but nobody does, so for now, ignore it.

Different trade-offs are involved with each of these styles, and again, part two of this series will go into them in further detail. It is important to know, however, that there is a third style, "document wrapped," which isn't formally specified anywhere but has gained popularity for various interoperability reasons. In this case, the content of the payload is wrapped in a single element, but that element may not be named after the procedure or function to which the data belongs. To the human eye, these messages are virtually identical to RPC/literal messages.

Message exchange patterns
When it comes to sending messages, you have a wide variety of choices, from sending a request and waiting for a response to sending request and forgetting about it, to sending a request and having it passed through multiple intermediaries before it gets to its final destination. In the end, though, you really have only two choices:

- **Request/Response**: In the Request/Response pattern, you send a request in the form of a SOAP message and you simply wait for a response back. The request may be synchronous or asynchronous.
- **One-way messaging**: This case, also known as the "fire and forget" method, involves sending the request and then simply forgetting about it. You can use this in situations in which you are simply imparting information, or any other situation in which you don't really care what the receiver has to say about it.

Now, note the lack of the terms "client" and "server". The reason for this is that these message exchange patterns can essentially be used to create any number of different alternatives such as the ones mentioned above. For example, you could create a situation in which you send a request, and count on the receiver to work on it, sending a message at some point in the future when it's done what it's supposed to do. To do that, you would use a combination of one-way messages, so it doesn't make sense to talk about the "client" and the "server", because each message has a sender and receiver, and the so-called client and server switch places.

Building a SOAP client
All right, you've seen the theory, now it's time to build the actual applications. Let's start with the client.

**The old way**
When Java APIs for working with SOAP messages first appeared, they were very specific about what they were for. They were, quite frankly, for creating the SOAP message. You were required to create the message, the Envelope, the Header, the Body, and so on. For example, you can build a "old-style" client to access the echo function of the MyService service you installed earlier (see Listing 15).

Note: To compile and run this client, you will need a SAAJ implementation such as the original Axis software. You can download Axis from http://ws.apache.org/axis/. Axis2 0.95 reportedly has this implementation as well, but it hasn't been tested for this tutorial.
Listing 15. An old-style SOAP client

```java
import javax.xml.SOAP.*;
import javax.xml.transform.*;
import java.io.FileInputStream;
import javax.xml.transform.stream.*;
import org.w3c.dom.*;

public class SendSOAP {
    public static void main(String args[]) {
        try {
            MessageFactory messageFactory = MessageFactory.newInstance();
            SOAPMessage message = messageFactory.createMessage();
            //Create objects for the message parts
            SOAPPart SOAPPart = message.getSOAPPart();
            SOAPEnvelope envelope = SOAPPart.getEnvelope();
            SOAPBody body = envelope.getBody();
            SOAPElement bodyElement = 
                body.addChildElement(envelope.createName("echo",
                "req", "http://localhost:8080/axis2/services/MyService/");
            bodyElement.addChildElement("category")
                .addTextNode("classifieds");
            message.saveChanges();
            SOAPPart SOAPpartbefore = message.getSOAPPart();
            SOAPEnvelope reqenv = SOAPpartbefore.getEnvelope();
            System.out.println("REQUEST:");
            System.out.println(reqenv.toString());
            //Now create the connection
            SOAPConnectionFactory SOAPConnFactory = 
                SOAPConnectionFactory.newInstance();
            SOAPConnection connection = 
                SOAPConnFactory.createConnection();
            SOAPMessage reply = connection.call(message,
                "http://localhost:8080/axis2/services/MyService");
            SOAPPart SOAPpart = reply.getSOAPPart();
            SOAPEnvelope replyenv = SOAPpart.getEnvelope();
            System.out.println("\nRESPONSE:");
            System.out.println(replyenv.toString());
            connection.close();
        } catch (Exception e){
            System.out.println(e.getMessage());
        }
    }
}
```

Notice that you are directly creating the SOAPEnvelope, SOAPBody, and so on. You can add elements such as echo and the category element to that body. From there, you create the connection, make the call, and you can once again work your way through the structure of the SOAP message to get to the actual content. If you were to run this client, you would see a response something like that shown in Listing 16.
**Listing 16. The client so far**

REQUEST:

```xml
<SOAPenv:Envelope xmlns:SOAPenv=
    "http://schemas.xmlSOAP.org/SOAP/envelope/
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <SOAPenv:Body>
    <req:echo xmlns:req="http://localhost:8080/axis2/services/MyService/">
      <req:category>classifieds</req:category>
    </req:echo>
  </SOAPenv:Body>
</SOAPenv:Envelope>
```

RESPONSE:

```xml
<SOAPenv:Envelope xmlns:SOAPenv=
    "http://schemas.xmlSOAP.org/SOAP/envelope/
    xmlns:wsa=
    "http://schemas.xmlSOAP.org/ws/2004/08/addressing">
  <SOAPenv:Header>
    <wsa:ReplyTo>
      <wsa:Address>
        http://schemas.xmlSOAP.org/ws/2004/08/addressing/role/anonymous
      </wsa:Address>
    </wsa:ReplyTo>
    <wsa:From>
      <wsa:Address>
        http://localhost:8080/axis2/services/MyService</wsa:Address>
    </wsa:From>
    <wsa:MessageID>ECE5B3F187F29D28BC11433905662036</wsa:MessageID>
  </SOAPenv:Header>
  <SOAPenv:Body>
    <req:echo xmlns:req="http://localhost:8080/axis2/services/MyService/">
      <req:category>classifieds</req:category>
    </req:echo>
  </SOAPenv:Body>
</SOAPenv:Envelope>
```

All the echo service actually does is apply back with the request it has received, which is a good chance to see the difference between old-style and new-style processing. Let's talk about that difference.

**The new way**

These days, there's a growing movement to hide the complexity of working with XML-based web services messages from the programmer. All sorts of initiatives have sprung up, most of them having the purpose of making programming to web services as much like programming for any other architecture as possible.

In Axis2, it actually means more than that. Axis2 introduces an entirely new way of working with the XML that represents a SOAP message, although on the surface it does look much like using the Document Object Model. The AXIs Object Model, or AXIOM, makes a number of changes, but for the moment I'll just mention that it focuses on the Infoset of the message, which is the actual information contained in elements and attributes, rather than the serialized version of tags we normally see. More importantly, however, Axis2 takes care of the SOAP envelope for you, enabling you to concentrate on just building the payload, or in the case of the actual services, analyzing the payload and creating a response. Let's see how that works.
Create the request

To start creating the client, make sure that all of the *.jar files in the Axis2 lib directory -- this would be the Standard distribution, not the War distribution -- are on your CLASSPATH and create a new class called `ClassifiedClient`. Create the payload as shown in Listing 17.

Listing 17. The payload

```java
import org.apache.axis2.addressing.EndpointReference;
import org.apache.axis2.client.Options;
import org.apache.axis2.client.ServiceClient;
import org.apache.axis2.om.OMElement;
import javax.xml.stream.XMLOutputFactory;
import javax.xml.stream.XMLStreamException;
import java.io.StringWriter;
import org.apache.axis2.om.OMAbstractFactory;
import org.apache.axis2.SOAP.SOAPFactory;
import org.apache.axis2.om.OMFactory;
import org.apache.axis2.om.OMNamespace;

public class ClassifiedClient {
    public static OMElement getEchoOMElement() {
        SOAPFactory fac = OMAbstractFactory.getSOAP12Factory();
        OMMNamespace omNs = fac.createOMNamespace("http://daily-moon.com/cms", "cms");
        OMElement method = fac.createOMElement("echo", omNs);
        OMElement value = fac.createOMElement("category", omNs);
        value.addChild(fac.createText(value, "classifieds"));
        method.addChild(value);
        return method;
    }

    public static void main(String[] args) {
        try {
            OMElement payload = ClassifiedClient.getEchoOMElement();
        } catch (Exception e) { //XMLStreamException e) {
            System.out.println(e.toString());
        }
    }
}
```

First, you create a factory and a namespace, and you use them to create elements. In this case, you're creating the same elements you created in the previous example, as you'll once again use this client to access the `echo` function. (Later, you'll change it to access the real service.)

Next, you'll create the request.

Create the request

The next step is to create the actual request. Here again you see the march of time. Rather than simply sending the request to a URL, you are setting an "endpoint reference", as shown in Listing 18.

Listing 18. The endpoint reference in the request

```java
... public class ClassifiedClient {
    private static EndpointReference targetEPR = new
```
EndpointReference(
  "http://localhost:8080/axis2/services/MyService");

  public static OMElement getEchoOMElement() {
  ...
  }

  public static void main(String[] args) {
      try {
      OMElement payload = ClassifiedClient.getEchoOMElement();
      Options options = new Options();
      options.setTo(targetEPR);
      options.setTransportInProtocol(Constants.TRANSPORT_HTTP);

      } catch (Exception e) { // (XMLStreamException e) {
          System.out.println(e.toString());
      }
  }

A complete discussion of WS-Addressing is outside the scope of this tutorial, but suffice it to say that an endpoint reference includes the URL to which a message is directed, but can optionally include other information such as a reply-to address, and other resource properties.

First, you create the Options for the request, which enables you to set the EPR for the request, as well as other information such as the transport you intend to use. Once you have all of that down, you can actually send the request.

**Send the request**

Once you've got everything set, it's time to send the request. As of Axis, to version 0.94, the preferred way to send a message is through the `ServiceClient` class, shown in Listing 19.

**Listing 19. Sending the request**

```java
public static void main(String[] args) {
    try {
        OMElement payload = ClassifiedClient.getEchoOMElement();
        Options options = new Options();
        options.setTo(targetEPR);
        options.setTransportInProtocol(Constants.TRANSPORT_HTTP);

        ServiceClient sender = new ServiceClient();
        sender.setOptions(options);
        OMElement result = sender.sendReceive(payload);

    } catch (Exception e) { // (XMLStreamException e) {
        System.out.println(e.toString());
    }
}
```

You create the `ServiceClient` object, and set the `options` you created earlier for it. From there, you can simply send the message. Because you want a response, you'll use the `sendReceive()` method, which is an in/out message. You'll look at one-way messaging in the **one-way service** section of this tutorial. The `sendReceive()` method returns the actual response, as it was received by the client.
Output the result

Actually, sendReceive() doesn't return the entire response, it returns only the payload. If you output the result to the command line, you can see this clearly in Listing 20.

Listing 20. The sendReceive output

```java
...  
  sender.setOptions(options);
  OMElement result = sender.sendReceive(payload);
  System.out.println(result.toString());
  }
    } catch (Exception e) { //XMLStreamException e) {
      System.out.println(e.toString());
    }
  }

Running this client gives you the response shown in Listing 21.

Listing 21. The sendReceive response


You can, of course, do many things with this data once you receive it. For now, let's build the actual getNumberOfArticles() service.

Building a SOAP service

If the process of building a web service client seems fairly straightforward to you, you're right. And building a service is, in many ways, just as simple.

The overall process

The overall process of creating an Axis2 web service involves the following steps:

1. Create the service manifest
2. Create the class
3. Package them up into an Axis archive file
4. Upload the Axis archive file to the Axis2 web application
5. Restart the server, if necessary

That's all there is to it. Let's start with the service manifest.

Create the manifest

The service manifest tells the Axis2 application (and by extension, the application server) what requests correspond to what classes. For example, you can specify two service functions, as shown in Listing 22.
Listing 22. Specifying two service functions in the manifest

```
<service name="CMSService">
<description>
    This is a sample web service for the newspaper's Content Management System.
</description>

<parameter name="ServiceClass" locked="false">
    CMSService</parameter>

<operation name="getNumberOfArticles">
    <messageReceiver class="org.apache.axis2.receivers.RawXMLINOutMessageReceiver"/>
</operation>

<operation name="addArticle">
    <messageReceiver class="org.apache.axis2.receivers.RawXMLINOnlyMessageReceiver"/>
</operation>
</service>
```

First, you define the overall service, giving it a name and a description and specifying the class to actually serve the request. Next, you define the actual operations. Notice that this example specifies two different types of `messageReceiver`. The first, `RawXMLINOutMessageReceiver`, is for the traditional request/response service. The second, `RawXMLINOnlyMessageReceiver` is for one-way messages. The name of the operation corresponds to the root element of the payload, as well as the method to execute.

Save this file as services.xml.

Now let's create the actual application.

Create the application

Let's start by creating a class that mimics the `echo` function we saw earlier, simply returning a copy of the original payload, shown in Listing 23.

Listing 23. The CMSService class

```
import org.apache.axis2.om.OMElement;
import javax.xml.stream.XMLStreamException;

public class CMSService {
    public OMElement getNumberOfArticles(OMElement element)
        throws XMLStreamException {
        element.build();
        element.detach();
        return element;
    }
}
```

To compile this application, make sure all of the *.jar files in `<axis2_home>/lib` are on your `CLASSPATH`. 
The application is pretty simple, with just one function corresponding to the `getNumberOfArticles` operation. This function, and any function intended as an operation, takes a single `OMElement`, which represents the payload. Here you first use the `build()` method to make sure all of the data has been received -- AXIOM uses a pull method of data access -- and then detach the element from its current tree so you can return it.

If you feel adventurous, feel free to Deploy the service and Access the service to access the service and see the results. You should see something along the lines of that shown in Listing 24.

**Listing 24. The CMSService class response**

```xml
<cms:getNumberOfArticles><cms:category>classifieds</cms:category></cms:getNumberOfArticles>
```

Now let's look at actually dealing with the data.

**Extract the payload**

Extracting information from the payload is a matter of manipulating the received payload element using techniques that are very similar to DOM (see Listing 25).

**Listing 25. Extracting payload information**

```java
import javax.xml.stream.XMLStreamException;
public class CMSService {
    public OMElement getNumberOfArticles(OMElement element)
        throws XMLStreamException {
        element.build();
        element.detach();

        String rootName = element.getLocalName();
        OMElememt categoryElement = element.getFirstElement();
        String categoryName = categoryElement.getLocalName();
        String categoryValue = childElement.getText();

        return element;
    }
}
```

Remember, the root of a payload is the element received by the `getNumberOfArticles` function. In this case, you are extracting the name of that element, and then moving on to its first element child (as opposed to its first child, which could be a whitespace text node) and extracting its name and value. Notice that you are using the `getText()` method to pull what is essentially the value of a text node child of the category element. It's definitely a handy shortcut!

**Create and return the response**

Finally, you'll need to use the data you pulled from the payload of the request to create a response. In this case, you'll feed it to a second function that would, in an actual application, do other work (see Listing 26).

**Listing 26. Creating a response**

```java
...
```
import javax.xml.stream.XMLStreamException;

public class CMSService {
    public OMElem getNumberOfArticles(OMElement element)
        throws XMLStreamException {
        element.build();
        element.detach();

        String rootName = element.getLocalName();
        OMElem childElement = element.getFirstElement();
        String childName = childElement.getLocalName();
        String categoryValue = childElement.getText();

        SOAPFactory factory = OMAbstractFactory.getSOAP12Factory();
        OMNamespace namespace = factory.createOMNamespace(
                "http://daily-moon.com/cms/", "resp");
        OMElem resultElem = factory.createOMElement(
                "numberOfArticles", namespace);

        String actualValue =
                (articleCount(categoryValue)).toString();
        resultElem.setText(actualValue);

        return resultElem;
    }

    private Integer articleCount(String catId){
        //Perform some function such as searching the CMS
        //database, and return the actual value. For our
        //purposes, you'll hardcode it.
        return new Integer(42);
    }
}

First, create the factory you'll use to create all of the other objects, and then create the namespace
you'll add to the payload of the response. Next, create the actual result elements, in this case and
element called numberOfArticles.

The contents of the numberOfArticles element will be a number returned by the articleCount() function, which is completely arbitrary in this case. In a real application, you would do whatever
you needed to do to get this data. Once you have it, you will set it as the contents of the
numberOfArticles element and simply return that element.

All that's left now is to deploy the service.

Deploy the service
In order to deploy the service, you need to create an Axis archive file. This file is like a *.jar or a *
*.war file, in that it is simply a zip file with the special file extension, in this case .aar. Follow these
steps to create this file:

1. Add all of the files in the <AXIS2_HOME>/lib directory to your CLASSPATH and compile the
   CMSService.java file.
2. Create a new directory called META-INF in the same directory as the CMSService.class file.
3. From the directory containing the CMSService.class file, issue this command: <code
   type="section" width="100">jar cvf CMSService.aar ./</code> You should see
results something like: `<code type="section" width="100"> added manifest adding: CMSService.class(in = 513) (out= 330)(deflated 35%) adding: CMSService.java(in = 328) (out= 182)(deflated 44%) ignoring entry META-INF/ adding: META-INF/services.xml(in = 391) (out= 229)(deflated 41%) </code>`

4. Add the service to the server using the steps outlined in **Installing a sample service**. (If you see servlet errors on the web interface, make sure that you log in to the Axis2 application. If your session has expired, the application won't necessarily tell you, but may just display an error.)

5. If necessary, restart Geronimo. (You likely will not have to do this after adding the service, but you may have to do it after making changes.)

If you click the View services link, you should see something similar to that shown in Figure 4.

**Figure 4. Available services**

![Available services](image)

Access the service

Now that you've got the service built, it's time to access it via a client. Make the following changes to the ClassifiedClient.java file you created earlier (see Listing 27).

**Listing 27. Modifying ClassifiedClient**

```java
public class ClassifiedClient {
    private static EndpointReference targetEPR =
        new EndpointReference("http://localhost:8080/axis2/services/CMSService");

    public static OMEElement getEchoOMElement() {
        SOAPFactory fac = OMAbstractFactory.getSOAP12Factory();
        OMNamespace omNs = fac.createOMNamespace("http://daily-moon.com/cms", "cms");
        OMEElement method = fac.createOMElement("getNumberOfArticles", omNs);
        OMEElement value = fac.createOMElement("category", omNs);
        value.addChild(fac.createText(value, "classifieds"));
        method.addChild(value);
        return method;
    }
}
```
When you compile and run this application, you should see the response shown in Listing 28.

**Listing 28. The ClassifiedClient response**

There are 42 classifieds at the moment.

### The one-way service

Before moving on, let's look at the differences involved when you're dealing with a one-way service rather than a request/response service.

### The service

Creating a one-way service is pretty straightforward. The process is exactly the same as creating a request/response service, except that you don't actually return anything. For example, you can create the `addArticle` operation to the `CMSService` class, as shown in Figure 29.

**Listing 29. addArticle operation in the CMSService class**

```java
private Integer articleCount(String catId){
    ...
}

public void addArticle(OMElement element)
    throws XMLStreamException{
    element.build();
    System.out.println(element);
}
```

In the services.xml file, you specified the `addArticle` operation to be an “in only” operation, so it won't be expected to return anything anyway, but just so that you can see that something is actually happening, output the received payload to the command line. You'll see it in the Geronimo window.
In a real application, this method would extract information from the payload and actually added to some sort of database or other repository.

**The client**

The client for this service is also similar to what you would use for a request/response service (see Listing 30).

**Listing 30. Creating the client**

```java
import org.apache.axis2.addressing.EndpointReference;
import org.apache.axis2.client.Options;
import org.apache.axis2.client.ServiceClient;
import org.apache.axis2.om.OMElement;
import org.apache.axis2.SOAP.SOAPFactory;
import org.apache.axis2.om.OMAbstractFactory;
import org.apache.axis2.om.OMNamespace;

public class AddArticleClient {
    private static EndpointReference targetEPR =
        new EndpointReference("http://localhost:8080/axis2/services/CMSService");

    private static OMElement getOMElement() {
        SOAPFactory fac = OMAbstractFactory.getSOAP12Factory();
        OMMNamespace omNs = fac.createOMNamespace("http://daily-moon.com", "cms");
        OMElement method = fac.createOMElement("addArticle", omNs);
        OMElement category = fac.createOMElement("category", omNs);
        category.setText("classifieds");
        OMElement subcategory = fac.createOMElement("subcategory", omNs);
        subcategory.setText("wantads");
        OMElement adtext = fac.createOMElement("article", omNs);
        adtext.setText("Do you have good head for numbers+
                       and a great deal of patience? Do you like+
                       to sit for hours sorting objects by their+
                       size? If so, then you could be the+
                       next goober counter in the world famous+
                       Murphy Brothers peanut factory.
                       Willingness to dress up as our mascot+
                       helpful, but not required.");

        method.addChild(category);
        method.addChild(subcategory);
        method.addChild(adtext);

        return method;
    }

    public static void main(String[] args) {
        try {
            OMElement payload = AddArticleClient.getOMElement();
            ServiceClient serviceClient = new ServiceClient();

            Options options = new Options();
            serviceClient.setOptions(options);
            options.setTo(targetEPR);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```
serviceClient.fireAndForget(payload);

} catch (AxisFault axisFault) {
    axisFault.printStackTrace();
}

Although the payload is different, as you can see in the `getOMElement()` method, the only real change as far as programming is concerned is the use of the `fireAndForget()` method rather than the `sendReceive()` method. This method does not return a response.

If you run this client you should see output in the Geronimo window similar to that shown in Figure 5.

**Figure 5. Command line output**

![Command line output](image)

**Accessing the service via GET**

Prior to SOAP 1.2, the only way to access a SOAP-based web service using HTTP was through the use of a `POST` request. You would need to create a client that creates a `POST` request with the SOAP message as the content of that request. SOAP 1.2, however, defines the means for accessing a SOAP-based web service using a `GET` request.

**GET versus POST**

Before moving on, it's important to understand the distinction between `GET` and `POST` requests over HTTP. Although many web programmers act as though these two requests are interchangeable, there is actually a purpose behind each of them. `GET`, in which all of the information about the resource you're requesting is contained in the URL, usually as parameters, is meant to be used only for idempotent requests. Those are requests for which there are no "side effects". In other words, you should be able to call that request a dozen times, a hundred times, a thousand times, and it shouldn't change anything. For example, a web request for the current temperature in Albuquerque is idempotent. A web request that inserts a comment into a blog database is not.

The reason for this is that `GET` requests can be added to a user's bookmarks, and accessed without warning. They can also be refreshed without warning. On the other hand, a `POST` request includes its information in the body of the request, and so it is difficult to accidentally repeat.
As far as SOAP is concerned, this means that you should be able to use GET for SOAP requests that simply retrieve information without making changes. You should still use POST for any operations that do make changes.

**Accessing the service**

In Axis2, you can create a GET request and the server will translate it into a SOAP message and return the payload as the result. For example, point your browser to the location shown in Listing 31.

**Listing 31. Accessing the services**

```
http://localhost:8080/axis2/services/CMSService/getNumberOfArticles?category=classifieds
```

As of version 0.94 you will see a response shown in Listing 32.

**Listing 32. The SOAP payload response**

```
<resp:numberOfArcticles>42</resp:numberOfArcticles>
```

This is not quite accurate, however. According to the SOAP 1.2 Recommendation, you should be seeing the entire SOAP response. This will likely change in future versions of Axis2.

**Dealing with attachments**

Another variation on the simple SOAP message is the attachment. Attachments have been getting under people's skin for years, but because they are now required by certain extended specifications, you've got to deal with them.

**Binary data and XML**

Although XML is a text-based format, you just can't ignore the fact that it's a binary world out there. As such, there will be a time when you're called upon to pass binary information to or from a web service.

You can handle this situation in one of two ways. The first option is to actually include the binary data inside your document. One example of this occurs when you save a Microsoft Word document as an XML file. If you have any images embedded in that document, Word embeds them in the XML document as binary data, encoded in Base64. The second option is to simply refer to the data so that the application processing the document can find it. One extremely prevalent example of this is a Web browser and how it deals with images referenced from an XHTML file. The XHTML file includes an img element, (or, if you're suitably advanced, an object element) and that element includes a src attribute that contains the URL pointing to the actual data. The application can then load the data from that location and use it appropriately.

The same situation applies with SOAP documents. If you were, say, submitting an image to a SOAP-based service, you have two options. You can embed that binary data in the payload, or you can find a way to reference it. Historically, it has been referenced, because of issues involving bandwidth.
XML-binary Optimized Packages

XML, you see, is already more verbose than binary opponents would like. And as such, it does use lot more bandwidth. So when you consider the fact that the preferred method for adding binary data to an XML text document -- encoding it as Base64 -- has a tendency to increase its size by a factor of two or more, you have a real issue.

In fact, in the last couple of years, there was such an uproar about the lack of support for binary data in any real way, that there was almost a revolt, and the W3C took up the issue. The result was XML-binary Optimized Packages (XOP). This protocol provides a way for you to reliably reference external data from within an XML document. For example, the SOAP with Attachments specification said that binary data could be sent as part of a multipart MIME document, with the XML data making up the first part, and the binary data added as additional parts. The problem with this is that although your program may know that the data exists, the document does not. It also does not allow for selective optimization of the document, or the retroactive processing of an existing document that includes binary data.

XOP improves on this situation by providing a mechanism by which you can selectively extract information in need of optimization, add it to a multipart MIME message that also includes your SOAP message, and explicitly reference it. Let's look at an example.

Suppose, for example, that rather than adding a new article as a text element, staff wanted to add it as a binary document, say, from a word processing program. It would be awfully messy to include that content in the body of the message, as shown in Listing 33:

Listing 33. Adding a binary document

```xml
<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/SOAP-envelope">
  <env:Header>
  </env:Header>
  <env:Body>
      <cms:category>classifieds</category>
      <cms:subcategory>forsale</cms:subcategory>
      <cms:articleHeadline><cms:articleHeadline>
        wvetwenptiubnweoirntuwopeirt4m[456n99ew7nv sa0tv043u6y384o5mu60ew9rebtm45bm4-69nw-0er9utnv3994nb-2628495u6-49kv6-m34956h-wb09emjb-0n67u-34ov,=qw-enr7w8b64b03278-ANDLOTSMOREBASE64ENCODEDDATAHERE</cms:articleText>
      </cms:addArticle>
    </env:Body>
</env:Envelope>
```

XOP specifies that instead, you extract the data, and replace it with an Include element that references it's new location, as in this example shown in Listing 34.

Listing 34. Using XOP

```xml
MIME-Version: 1.0
Content-Type: Multipart/Related;boundary=MIME_boundary;
type="application/xop+xml";
start="<soapmsg.xml@daily-moon.com>"
```
Notice that the location specified in the `Include` element matches the `Content-ID`, minus the protocol of `cid:`. It is this message that now gets sent, rather than a plain text SOAP message.

**SOAP, binary data, and Axis2**

The process of using XOP in SOAP documents is called MTOM (for SOAP Message Transmission Optimization Mechanism). Axis2 provides support for this means of working with SOAP data, but you must be sure to configure the application appropriately.

Specifically, you must enable it in the axis2.xml file within the axis2.war file (see Listing 35).

### Listing 35. Using XOP with Axis2

```xml
<axisconfig name="AxisJava2.0">
  <!-- Parameters -->
  <!-- Uncomment this to enable REST support -->
  <parameter name="hotdeployment" locked="false">true</parameter>
  <parameter name="hotupdate" locked="false">true</parameter>
  <parameter name="enableMTOM" locked="false">true</parameter>
  <parameter name="userName" locked="false">admin</parameter>
  <parameter name="password" locked="false">axis2</parameter>
  ...
```
If necessary, you can extract the axis2.war file, make this change, and re-compress it into a .war.

To replace the Axis2 application, visit the Geronimo console, using the URL shown in Listing 36.

**Listing 36. Geronimo console**

```
http://localhost:8080/console
```

Log in as system/manager and click **Application>Web App WARs**, then uninstall and reinstall the Axis2 application. (Remember, you will have to reload your web services after performing this step.)

Programmatically working with MTOM is beyond the scope of this tutorial. Just be aware that things may not work as expected in Axis2 versions prior to 0.95, which includes a functioning SOAP with Attachments API for Java (SAAJ) implementation.

**Summary**

In today’s world, where interoperability between systems is so important, web services can be the foundation of your Service-Oriented Architecture. And SOAP is the foundation of enterprise-level web services. This tutorial showed you the basic concepts of web services, and explained both the concepts and programming necessary to understand and work with SOAP in your own applications. Along the way, you learned the following:

- The important concepts surrounding web services
- How to install and use the Geronimo application server
- How to install and use the Axis2 web services application
- How to create a client to access a SOAP service
- How to create a SOAP service
- Additional issues surrounding SOAP services, such as **GET** requests and attachments

In Part 2 of this series, Web Services Description Language, you will learn how to use WSDL to automate many of the steps you executed here, as well as to provide the means for others to more easily use the services you build.
## Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source code</td>
<td>ws-understand-web-services1.zip</td>
<td>12KB</td>
</tr>
</tbody>
</table>
Related topics

- SOAP 1.2 Primer
- SOAP Message Transmission Optimization Mechanism
- XML-binary Optimized Packaging
- Apache Axis2/Java
- Apache Geronimo
- Apache Geronimo download

© Copyright IBM Corporation 2006

Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)

Understanding web services specifications, Part 1: SOAP

Page 33 of 33