Sponsored by:

JAVAWORLD
SOLUTIONS FOR JAVA DEVELOPERS

This story appeared on JavaWorld at
http://www.javaworld.com/javaworld/jw-07-2001/jw-0706-soap.html

# Clean up your wire protocol with SOAP, Part 4

## Dynamic proxies make Apache SOAP client development easy

By Tarak Modi, JavaWorld.com, 07/06/01

Version 1.3 of the Java 2 Platform features an extremely savvy addition to the Java Reflection API: *dynamic proxy classes.* A dynamic proxy class is a class that implements a list of interfaces specified at runtime. This proxy can then be used as if it actually implemented these interfaces. In other words, any method on any of these interfaces can be called directly on the proxy -- after properly type-casting it, of course. Thus, you can use a dynamic proxy class to create a type-safe proxy object for an interface list without requiring pregeneration of the proxy class as you would with compile-time tools. (For a much more detailed introduction to the dynamic proxy classes, refer to the Resources section below.)

I will now introduce you to a framework based on these classes that will make the creation of SOAP (Simple Object Access Protocol) clients easy and intuitive. SOAP is a wire protocol that uses XML for data encoding. While building SOAP services in Parts 2 and 3 of this series, you discovered that client developers end up doing a lot of extra work that they wouldn't normally have to do. If you need a refresher, take a look at the SOAP service code in Part 2. See how insignificant the SOAP code is when compared to the client code? The simple services created in earlier articles in this series demonstrate that SOAP-based services only contain code that they would have had irrespective of SOAP. The service developers have little extra code to produce, while the client developers have much more work to complete on their end. The classes introduced in this article will alleviate most of that work.

Read the whole series on SOAP:

- Part 1: An introduction to SOAP basics
- Part 2: Use Apache SOAP to create SOAP-based applications
- Part 3: Create SOAP services in Apache SOAP with JavaScript
- Part 4: Dynamic proxies make Apache SOAP client development easy

## Introducing the SOAP Proxy class

First, let me show you what the client code would look like using the framework that you'll create in this article:

```
package hello;
import soapproxy.*;
public class Client
{
   public static void main(String[] args)
   {
       try
             {
                 Class[] interfaces = new Class[] {hello.Hello.class};
                 Hello hello =
(Hello)(Proxy.newInstance("urn:Hello",interfaces));
                 // Invoke the sayHelloTo method that takes a String
name.
                 System.out.println(hello.sayHelloTo("John"));
                 // Invoke the sayHelloTo method that takes a Name
JavaBean.
```

```
                Name theName = new Name();
                theName.setName("Mala");
                System.out.println(hello.sayHelloTo(theName));
            }
            catch(Exception e)
            {
                e.printStackTrace();
            }
    }
}
```

Now, I might be a bit biased, but I think the client code above looks much better than the client code in Parts 2 and 3. It's okay if you don't follow the code right now. Believe me, you will by the end of this section.

To understand the client code, you must delve deeper into the SOAP `Proxy` class, which is in the `soapproxy` package, available in the `Proxy.java` file (see Resources below). That class has a private constructor, which means that a `Proxy` instance cannot be created from outside `Proxy`. The only way to retrieve a new `Proxy` instance is through the static `newInstance()` method. That method takes two parameters: the SOAP service's object ID and an array of interface names that this proxy will implement. The object ID parameter is straightforward, but what are the interface names, and where did they come from? The SOAP service developer defines these interfaces by simply lumping together all the methods that a client can invoke on the service into a Java interface. Pretty easy, no?

Let's define an interface for our HelloWorld service. The final version of this service in Part 2 had two overloaded versions of the method `sayHelloTo()`: one version took a Java string, and the other took a `Name` JavaBean. This is what one possible interface, called `Hello`, looks like:

```
package hello;
public interface Hello
{
    public String sayHelloTo(String name);
    public String sayHelloTo(Name name);
}
```

The service developer decides how many interfaces to create -- and what to call them. For example, you could create two interfaces for the HelloWorld service, with each interface containing one method. In general, you should avoid creating interfaces with more than seven methods. Also consider lumping together only those methods that make sense together. For example, if the HelloWorld service also had a `sayByeTo()` method that returned a customized good-bye message to the caller, it would make sense to have two separate interfaces: one for the `sayHelloTo()` methods and one for the `sayByeTo()` methods.

Now that we have the interface that defines the contract between the HelloWorld service and the client, let's return our attention to the `newInstance()` method. As I mentioned above, the `newInstance()` method creates a new instance of the `Proxy` class. The method can do that because it is part of `Proxy` and can access the private constructor. `newInstance()` then calls the `initialize()` method on the newly created instance. The `initialize()` method is interesting because the dynamic proxy is created and returned there. `initialize()` is shown below:

```
private Object initialize(Class[] interfaces)
{

return(java.lang.reflect.Proxy.newProxyInstance(getClass().getClassLoader()
,interfaces,this));
}
```

Notice the use of the `newProxyInstance()` method. The only way to create an instance of a *dynamic* proxy class is to call the static `newProxyInstance()` method on it -- that is, the `java.lang.reflect.Proxy` class. The `java.lang.reflect.Proxy` class provides static methods for creating dynamic proxy classes, and it is also the superclass of all dynamic proxy classes created by those methods. In other words, it's not only a factory for creating dynamic proxy classes; it is a dynamic proxy class itself! So, in our example, the SOAP proxy is not the dynamic proxy; rather, the dynamic proxy is actually an instance of the `java.lang.reflect.Proxy` class returned by the `newProxyInstance` static method. As you will see later, the dynamic proxy will actually do all its work through the `invoke()` method that the SOAP proxy implements. So, how does the dynamic proxy know

about the SOAP proxy? Because a reference to the SOAP proxy was passed into the `newProxyInstance()` method. This may sound convoluted now, but it should all become clear when you examine the `invoke()` method.

The class `java.lang.reflect.Proxy` takes a class loader instance as its first parameter; an interface array to dynamically implement (which is the same array that the client passed into `newInstance()`) as its second parameter; and an instance of a class that implements the `java.lang.reflect.InvocationHandler` interface as its third parameter. Since the SOAP `Proxy` class implements the `InvocationHandler` interface, the third parameter is the proxy instance itself (i.e., `this`). The `InvocationHandler` interface has one method: `invoke()`. The Java runtime calls `invoke()` when any method on one of the dynamic proxy's dynamically implemented interfaces is called. So, for example, when the client calls the `sayHelloTo()` method on the dynamic proxy's `Hello` interface, the Java runtime will call `invoke()` on the SOAP proxy.

You may have observed that the `newInstance()` method on the SOAP proxy does not return an instance of the SOAP proxy. Instead, it returns the dynamic proxy that `newInstance()` just created, which dynamically implements the interface array that the client passed in. The client can type-cast this returned dynamic proxy into any of the interfaces passed into `newInstance()` and invoke methods on the dynamic proxy as if the dynamic proxy actually implemented the interface. The relevant portion of the client is shown below for convenience:

```
.
.
.
try
{
    Class[] interfaces = new Class[] {hello.Hello.class};
    Hello hello = (Hello)(Proxy.newInstance("urn:Hello",interfaces));
    // Invoke the sayHelloTo method that takes a String name.
    System.out.println(hello.sayHelloTo("John"));
    // Invoke the sayHelloTo method that takes a Name
JavaBean.
    Name theName = new Name();
    theName.setName("Mala");
    System.out.println(hello.sayHelloTo(theName));
}
.
.
.
```

Now, let's take a look at the `invoke()` method, which, according to the above code snippet, will be invoked twice, once for each call to the `sayHelloTo()` method. In a nutshell, the `invoke()` method does exactly what each client had to manually do in the examples in Part 2. That includes setting up a `Call` object with the proper call parameters and any custom mapping that those parameters might require. Since the `invoke()` method in the SOAP proxy handles all that, the client is released of that burden.

Of the three parameters that the `invoke()` method receives, only the last two are of interest to us. The second parameter, the `Method` object, gives the method name that is being invoked. Remember that the invoked method name corresponds to a well-known method that the service exposes. The `invoke()` method already has the service's object ID, which was passed in as a parameter to the `newInstance()` method. Using that information, the `invoke()` method starts setting up the `Call` object as follows:

```
Call call = new Call();
call.setTargetObjectURI(urn);
call.setMethodName(m.getName());
call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);
```

Now it's time to set the parameters for the remote service invocation. For that, we are interested in the third parameter of the `invoke()` method: an argument array passed into the method invoked on the dynamic proxy. The `zeroth` index is the leftmost argument, the first index is the second from the left, and so on. As an example, if the client called the `sayHelloTo(String name)` method, then that array would be an array of one string. The `invoke()` method processes each element of that array and creates a vector of `Parameter` objects (just like the client did in Part 2):

```
java.util.Vector params = new java.util.Vector();
for( int i=0; i<args.length; i++ )
{
            if( isSimple(args[i]) || isSimpleArray(args[i]) )
            {
                        params.add(new
```

```
Parameter(_paramName+(i+1),args[i].getClass(),args[i],null));

            }
            else if( isVector(args[i]) )
            {

addMapping((java.util.Vector)args[i]);
                        params.add(new
Parameter(_paramName+(i+1),args[i].getClass(),args[i],null));

            }
            // if this is an non-simple array then
            // Assume that this is an array of beans
            else if( isArray(args[i]) )
            {
                        if( smr == null )
                                    smr = new
SOAPMappingRegistry();
                        if( beanSer == null )
                                    beanSer = new
BeanSerializer();
                        //System.out.println("Adding a default
mapping");
                        ArraySerializer arraySer = new
ArraySerializer();
                        smr.mapTypes(Constants.NS_URI_SOAP_ENC,
null, null, beanSer, beanSer);

smr.mapTypes(Constants.NS_URI_SOAP_ENC,null,args[i].getClass(), arraySer,
arraySer);
params.add(new
Parameter(_paramName+(i+1),args[i].getClass(),args[i],null));

            }
            // Assume that this is a bean
            else
            {
                        if( smr == null )
                                    smr = new
SOAPMappingRegistry();
                        if( beanSer == null )
                                    beanSer = new
BeanSerializer();
                        String qnamePart =
args[i].getClass().getName();
                        //System.out.println("qnamePart = " +
qnamePart);
                        smr.mapTypes(Constants.NS_URI_SOAP_ENC,
                                     new QName(urn,
qnamePart),args[i].getClass(), beanSer,
beanSer);
                        params.add(new
Parameter(_paramName+(i+1),args[i].getClass(),args[i],null));

            }
}
```

`invoke()` uses a number of private helper methods, such as `isSimple()`, to determine the type of parameter. If the parameter is a JavaBean or an array, then a custom SOAP mapping registry must be set up and set in the `Call` object via the `setSOAPMappingRegistry()` method (see Part 2). The SOAP proxy assumes that, in the JavaBean case, any JavaBean used by the SOAP service maps as follows: the *NameSpace UR* field is set to the object ID and the *Local Part* is set to the JavaBean's fully qualified class name. Since this is exactly how we've deployed the HelloWorld service, we're fine.

The remainder of `invoke()` is fairly straightforward. The `Call` object parameters (if any) are set; the custom SOAP mapping registry (if one is required) is set; the call is made; and the call results return to the client as shown below:

```
if( params.size() != 0 )
            call.setParams(params);
if( smr != null )
            call.setSOAPMappingRegistry(smr);
// Invoke the call.
Response resp = call.invoke(serverURL, "");
```

```
if( !resp.generatedFault() )
{
            Parameter ret = resp.getReturnValue();
            return(ret.getValue());
}
else
{
            Fault fault = resp.getFault();
            throw new
SOAPException(fault.getFaultCode(),fault.getFaultString());
}
```

## The HelloWorld service

Below you will find the complete implementation of the HelloWorld service. Does this code look familiar?

```
package hello;
public class HelloServer
{
    public String sayHelloTo(String name)
    {
            System.out.println("sayHelloTo(String name)");
            return "Hello " + name + ", How are you doing?";
    }
    public String sayHelloTo(Name theName)
    {
            System.out.println("sayHelloTo(Name theName)");
            return "Hello " + theName.getName() + ", How are you
doing?";
    }
}
```

Recall that `Name` is a simple JavaBean as shown below:

```
package hello;
public class Name
{
    private String name;
    public String getName()
    {
            return name;
    }
    public void setName(String name)
    {
            this.name = name;
    }
}
```

This is exactly the same service code that we created in Part 2. The only additional work that the service creator has to complete is to create the Java interfaces. Deploying the service is also exactly the same as the deployment presented in Part 2, so I will not go through it again. The similarities don't end there. The way you compile and run the client program is no different from Part 2's approach. Why so many similarities? Because the SOAP proxy we created is an example of a nonintrusive framework that does not modify or interfere with the inner workings of any Apache SOAP components -- client or server side.

## Some implementation notes

The SOAP proxy described in this article (and available for download from Resources) supports the following parameter types:

1. The following Java primitives and their objectified forms:

```
boolean, Boolean,
double, Double,
```

```
float, Float,
long, Long,
int, Integer,
short, Short,
byte, Byte
```

**Note:** The server side always receives the primitive.

2. Any JavaBean.

   **Note:**

   - The JavaBean must not contain other JavaBeans.
   - The JavaBean must not contain vectors or arrays that contain any parameter other than strings or those listed in 1 above.

3. The following classes:

   ```
   String, Vector
   ```

   **Note:**

   - A vector might contain all parameters in 1, 2, and strings (with an exception noted below).
   - The server side receives a vector as an array of objects.

4. Arrays of all parameters in 1, 2, and strings (with the exception noted above).

## That's all, folks

In this four-part series, I have not only introduced you to the basics of SOAP, but also to an excellent implementation of the SOAP 1.1 standard: Apache SOAP. In this article, I presented a framework based on the dynamic proxy classes that greatly simplifies the life of client-side developers who use Apache SOAP.

My gut feeling is that SOAP is going to be big. I have at least a couple of reasons to think so. First of all, as you know, SOAP is based on open standards such as XML. This has led to the widespread acceptance of SOAP in both Microsoft and anti-Microsoft camps. That's great news for developers. Second, SOAP is becoming the base for a number of other standards, such as Universal Description, Discovery, and Integration (UDDI). Both SOAP and UDDI are critical pieces of Web services, which many claim will be the next generation of Web application development.

## About the author

A certified Java programmer, Tarak Modi is an architect at North Highland, which specializes in management and technology consulting. He has a master's degree in computer engineering and an MBA concentrating in information systems. He has several years of experience working with C++ and Java, as well as technologies such as DCOM, CORBA, RMI, and EJB. He has written articles for leading software magazines including JavaWorld and is currently working on a book on Java Message Service with Manning Publications. To find out more about Tarak, his articles, and upcoming book, please visit his Website at http://tmodi.home.att.net/.

All contents copyright 1995-2010 Java World, Inc. http://www.javaworld.com