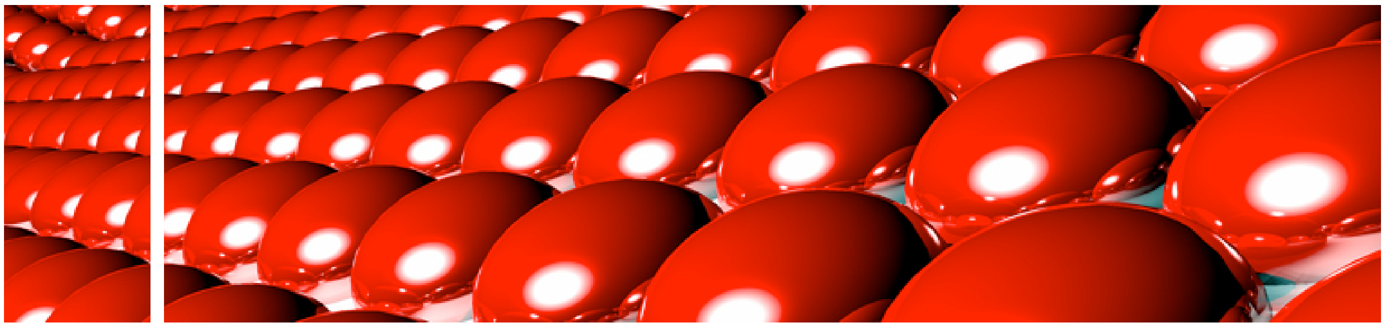
May 2006

SIMPLE JAVA AND .NET SOA INTEROPERABILITY



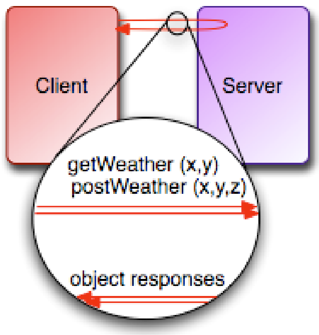
A discussion on REST and a simple, low dependency solution to interop between Java and .Net over the wire.

By Paul Hammant and Ian Cartwright

In this article, we intend to show how simple technology coupled with a document-centric approach can be used to deliver valuable business services without the use of proprietary middle-ware or the complexities of the web services stack. We take our inspiration from the REST architectural style, and the ability to move XML over HTTP.

The Web Service way

The best way to introduce this approach is to contrast it with a simple web service example. Imagine a simple Weather Service exposing a web method called 'WeatherQuery' that returned the temperature and pressure wrapped in an object. In most cases people take existing code and use a tool to expose a method and generate some WSDL.



If you believe the hype all we need to do now is point an equivalent Java tool at the WSDL and generate a stub-method.

Unfortunately things are not quite that simple, WSDL is a broad standard, in fact broad enough to be open to interpretation. In our case we found .Net enforced a document approach and our java tools assumed the opposite, RPC. We also found issues with mixtures of namespaces, inclusion of schemas and tools splitting the WSDL into separate parts. In short the technology starts to distract from the actual problem we are trying to solve.

To compound the issue, we also found inconsistencies within tools for web services. For example versions of Internet Information Server and Web Services Enhancements were only partially compatible with each other or their Java equivalents for automatically published WSDL documents.

	.Net Server(s)	Java Server(s)
.Net Client	√	some
Java Client	some	some

We became very weary of something that may be kludged to work today, but may break tomorrow if a more complicated web method were needed in later version of the service.

A more RESTful style

The above approach made two key assumptions: firstly that just exposing an existing method call would give us a meaningful service, and secondly that tools would make getting at that service via web services trivial.

Instead of thinking about the parameters of the request and the type of the return, we can think of the request as a document that embodies the type of request along with its parameters. Think of that document as something that represents part of the contract for the business process you are trying to model. If we take the same WeatherQuery method, and describe it in an element normal XML we might see something like -

```
<WeatherQuery>
  <locn>chicago</locn>
  <date>2006 02 12 18:52</date>
</WeatherQuery>
```

Also as a document, the return type may look like -

```
<WeatherDetails>
  <locn>chicago</locn>
  <date>2006 02 12 18:52</date>
  <temp>32</temp>
  <pressure>30</pressure>
</WeatherDetails>
```

Now we can define a simple class design that represents the same fields as the documents -

```
public class WeatherQuery {
  private String locn;
  private Date date;
  // .. getters & setters
  // or properties ..
}
public class WeatherDetails {
  private String locn;

  private Date date;
  private int temp;
  private int pressure;
  // .. getters & setters ..
}
```

The interesting thing about these two documents is that they are relatively easy to serialize and deserialize in both .Net and Java. For .Net, the built-in XML serialization (with some annotations) does the job well. Here is the C# for the same classes -

```
[XmlRoot("WeatherQuery")]
public class WeatherQuery {

  [XmlElement("locn")]
  private string locn;
  [XmlElement("date")]
  private DateTime date;
      // property: get{}& set{} ..
}
```

Java is not so lucky. It has no built in tools for XML serialization: however there is an open source tool called XStream (http://xstream.codehaus.org/) that is perfect for the job -

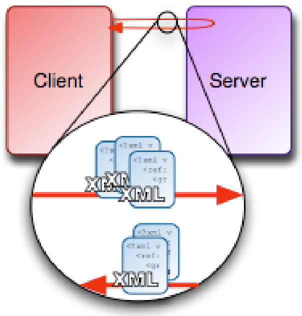
```
XStream xs = new XStream();

String reqXML = xs.toXML(req);

Object req = xs.fromXML(reqXML);
```

Our Weather service currently only facilitates the exchange of documents, and as a consequence, we could use plain HTTP instead of Web Services.

With this shift, we are able to think about multiple document types using the same entry-point into the system giving us a more extensible approach



	.Net Server(s)	Java Server(s)
.Net Client	√	√
Java Client	√	√

Coding the Java server

The technologies we used for a Java implementation of the service are that of the serialization library XStream, and any servlet container, as we chose to host the services inside a Servlet. XStream is open source: the servlet container could be any of WebLogic, WebSphere, JBoss, Geronimo, Orion or just Resin, Tomcat or Jetty if less of the E in J2EE is required.

Our servlet should implement the doPost() rather than doGet() method. We did not use a name/value pair for the XML: we simply used the entire POST body, which is a little outside of the HTTP specification. It is a matter of preference though - as long as its the same on client and server.

When a request comes in, we deserialize the XML into a command object using XStream and appropriately handle it.

Coding the .Net server

The server technologies for .Net are simpler - we just used Internet Information Server (IIS) and .Net's built-in XML serialization. The built-in serialization requires C# attributes to mark fields to as XML elements rather than XML attributes. There is a .Net port of XStream that makes things simpler still, but we have not experimented with that, and we have heard of another port about to be launched on the open source world.

Coding the Java client

For the Java client in additional to XStream we use Apache's HttpClient library (and its dependencies). See http://jakarta.apache.org/commons/httpclient/

The HttpClient library is quite simple to use and allows a POST operation to be programmatically constructed before execution. Remember, the POST body is entirely XML delivered by XStream as either name/value pairs or the entire request without that HTTP construct. The former may be attractive if you want to create a test web form for the service.

Coding the .Net client

For the .Net client you'll need just the framework and this can be either version 1.1 or 2.0.

For the POST operation, we can use the built-in WebRequest class and and the built-in serialization.

Making it extensible and compatible

The beauty of simply POSTing XML that represents commands is that we can add more commands later without changing our messages implementation. The server can decide whether it can deal with the XML at runtime -

```
<PostWeather>
  <locn>chicago</locn>
  <date>2006 02 12 21:00</date>
  <temp>32</temp>
</PostWeather>
```

A good solution to the multitude of potential types is to register a handler for each type, helping avoid a large if/else or switch/case block -

```
map.register("WeatherQuery",
    new WeatherQueryHandler());
```

When using XML there is a temptation to over-rely on the related schema, We have to be careful to understand the difference between schema validation and the information a consumer might actually need.

Schema validation, if performed at runtime, gives a developer a sense of safety that is not appropriate. A failure may still happen - the wrong XML could be sent. But what happens? There would be a schema invalid exception message raised. Alternatively with the XML mapped to a class's design, there would either be a correct object, or it would be missing some fields. In that situation, a real exception could be thrown with a real reason, that could easily be turned into a clear XML reply message for the consumer. The key difference is that an exception is raised only if required information is missing – anything else could change.

Inherently in this design, there is the possibility that elements could be added to the XML (fields to the class) that make it possible to move the API notionally forward a notch. With some careful testing, the service could support consumers sending older versions of the request documents. An API change, as well as being 'extra field' level, could also be more far reaching:

```
<WeatherQuery2>
  <locn>
    <city>chicago</city>
    <zip>60661</zip>
  </locn>

  <altitude>22000</altitude>
  <date>2006 02 12 18:52</date>
</WeatherQuery2>
```

It may be better to encode the version number into the URL though:

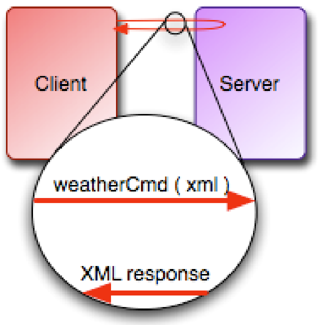
http://x.com/weather/WeatherQuery/2.0

Wrapping it in Web Services

There is nothing to stop you from mounting a second service on the same server to accept formal Webservice requests for the same service. All you would need is a single WSDL specified method like -

```
String weatherCmd(String xmlRe-
            quest)
```

Your tool choices are WSE 2.0 /3.0 for .Net or Glue, AXIS, JAX-WS, or some built-in adapter for one of the J2EE containers for Java. The SOAP encoded method could simply delegate to the same demarshal-execute-marshal code developed for the pure REST implementation. You may be doing this to side-step the corporate standards police.



Extending it to Messaging

Leveraging Tibco Rendezvous, we were able to send the same XML representations of requests around for execution against an implicitly asynchronous service. We did not try MQ Series but it should work too.

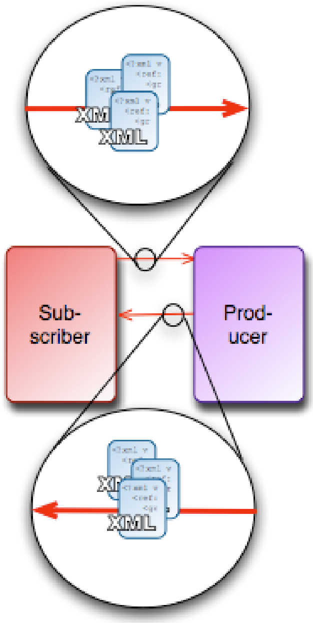
For our example this means the request for weather details is not going to be immediately satisfied. Instead, some time later, a response may come back. It is a huge shift that might mean changing or abandoning some of the simple designs you might have for an API. For example, the following facade method may have to go :

```
WeatherDetails weatherQuery
        (String locn, Date date)
```

In its place two interfaces, each with a single method, may be appropriate -

```
void weatherQuery
        (String locn, Date date)
void acceptWeather
        (WeatherDetails details)
```

There is lots to learn about messaging patterns, which we'll not be going into here. However, it may be worth noting that there are two general designs to consider. The first is a queue design, where requests from you receive responses directed only to you. The second is a multicast concept, where there are events on the wire that are being sent to many subscribers (pub/sub). Also, implicitly, there is the possibility that you could engineer a queue to continue streaming revised details over time. If you are familiar with JMS, Rendezvous works slightly differently in this respect.



Whither WSDL

The producer/consumer design we have coded trades XML in a simple design. But the specification checking that is inherent in the discovery phase of a formal Web Services design is missing from ours.

We suggest it is not really needed. Instead, and wholly in line with Agile thinking, have a comprehensive Continuous Integration suite of integration tests. Your service incompatibilities will be discovered before you deploy to live. So instead of a complex WSDL specification you have a series of unit tests that make assertions about the service from both a provider and consumer point of view. Anyway, WSDL only flags incompatibilities at runtime, when recovery is very hard. Thus is it a false god?

Schematron is one way of creating such tests in a platform independent way.

To aid debugging and documentation, you may want to host some sample documents :

```
<WeatherDetails>
  <locn>AAAAAA</locn>
  <date>CCYY MM DD HH:MM:SS/date>
</WeatherDetails>
```

You could also have an XML Schema (XSD) controlling the document format :

```
...
<xs:element name="WeatherQuery">
  <cs:complexType>
    <xs:all>
      <xs:element name="locn"
                  type="xs:string"/>
      <xs:element name="date"
                  type="xs:string"/>
    </xs:all>
  </cs:complexType>
</xs:element>
...
```

Serving both the XSD and the sample XML statically could be a good idea (the authors differ on the XSD as it happens). Served statically means that the API can be queried by a human with a web browser -

http://x.com/weather/xsd/WeatherQuery

http://x.com/weather/sample/WeatherQuery

Remember, both the XSD and sample document are optional and could easily be generated too.

Recap and Key Message

The magic here was to use Codehaus' XStream to participate in a element-normal document exchange with .Net via HTTP-POST operation rather than GET. Just about everything else has been blogged and white-papered before. Choosing XStream meant that the 'specification' for the message was in Java and/or C# and not an XML based design as is usually encountered with WS-* specs.

Also, conventional REST wisdom suggests that HTTP-GET is better for encoding the command...

http://x.com/weather/WeatherQuery?locn=Chicago

... especially when the result is cacheable by a web server. Perhaps our style is best for communication where attempts at caching are pointless. There are other advantages to the GET approach that we have lost with ours. It is more elegant and eminently testable by a humans with a browser by virtue of a complete URL. It lacks, however, the versatility we get with POST which allows more than just name/value pairs for parameters; the XML payload can be arbitrarily complex. There is room for GET and POST, of course, in larger solutions.

While writing this paper, we made a suggestion to the XStream team (by way of a patch) that would make it more able to support attributes where appropriate. They implemented the required functionality and future versions of XStream will be able to support XML attributes for more seamless introp with .Net.

We also suggested that the approach works with async transports like Tibco RV. Current WS-* spec tools have little to no capabilty in this regard.

In the end, we are not sure whether this is POST-REST (PREST) or just good old REST without much caching and URL simplicity and no new coined terms.