XML-RPC - QUICK GUIDE

XML-RPC - INTRODUCTION

RPC stands for Remote Procedure Call. As its name indicates, it is a mechanism to call a procedure or a function available on a remote computer. RPC is a much older technology than the Web. Effectively, RPC gives developers a mechanism for defining interfaces that can be called over a network. These interfaces can be as simple as a single function call or as complex as a large API.

What is XML-RPC ?

XML-RPC is among the simplest and most foolproof web service approaches that makes it easy for computers to call procedures on other computers.

- XML-RPC permits programs to make function or procedure calls across a network.
- XML-RPC uses the HTTP protocol to pass information from a client computer to a server computer.
- XML-RPC uses a small XML vocabulary to describe the nature of requests and responses.
- XML-RPC client specifies a procedure name and parameters in the XML request, and the server returns either a fault or a response in the XML response.
- XML-RPC parameters are a simple list of types and content - structs and arrays are the most complex types available.
- XML-RPC has no notion of objects and no mechanism for including information that uses other XML vocabulary.
- With XML-RPC and web services, however, the Web becomes a collection of procedural connections where computers exchange information along tightly bound paths.
- XML-RPC emerged in early 1998; it was published by UserLand Software and initially implemented in their Frontier product.

Why XML-RPC ?

If you need to integrate multiple computing environments, but don't need to share complex data structures directly, you will find that XML-RPC lets you establish communications quickly and easily.

Even if you work within a single environment, you may find that the RPC approach makes it easy to connect programs that have different data models or processing expectations and that it can provide easy access to reusable logic.

- XML-RPC is an excellent tool for establishing a wide variety of connections between computers.
- XML-RPC offers integrators an opportunity to use a standard vocabulary and approach for exchanging information.
- XML-RPC's most obvious field of application is connecting different kinds of environments, allowing Java to talk with Perl, Python, ASP, and so on.

XML-RPC Technical Overview

XML-RPC consists of three relatively small parts:

- **XML-RPC data model** : A set of types for use in passing parameters, return values, and faults *errormessages*.
- **XML-RPC request structures** : An HTTP POST request containing method and parameter information.

- **XML-RPC response structures** : An HTTP response that contains return values or fault information.

We will study all these three components in the next three chapters.

XML-RPC - DATA MODEL

The XML-RPC specification defines six basic data types and two compound data types that represent combinations of types.

Basic Data Types in XML-RPC

Type	Value	Examples
int or i4	32-bit integers between - 2,147,483,648 and 2,147,483,647.	<int>27</int> <i4>27</i4>
double	64-bit floating-point numbers	<double>27.31415</double> <double>-1.1465</double>
Boolean	true 1 or false 0	<boolean>1</boolean> <boolean>0</boolean>
string	ASCII text, though many implementations support Unicode	<string>Hello</string> <string>bonkers! @</string>
dateTime.iso8601	Dates in ISO8601 format: CCYYMMDDTHH:MM:SS	<dateTime.iso8601> 20021125T02:20:04 </dateTime.iso8601> <dateTime.iso8601> 20020104T17:27:30 </dateTime.iso8601>
base64	Binary information encoded as Base 64, as defined in RFC 2045	<base64>SGVsbG8sIFdvcmxkIQ==</base64>

These basic types are always enclosed in *value* elements. Strings *andonlystrings* may be enclosed in a *value* element but omit the *string* element. These basic types may be combined into two more complex types, arrays, and structs. Arrays represent sequential information, while structs represent name-value pairs, much like hashtables, associative arrays, or properties.

Arrays are indicated by the *array* element, which contains a *data* element holding the list of values. Like other data types, the *array* element must be enclosed in a *value* element. For example, the following *array*contains four strings:

```
<value>
   <array>
      <data>
         <value><string>This </string></value>
         <value><string>is </string></value>
         <value><string>an </string></value>
         <value><string>array.</string></value>
      </data>
   </array>
</value>
```

The following array contains four integers:

```
<value>
   <array>
      <data>
         <value><int>7</int></value>
         <value><int>1247</int></value>
         <value><int>-91</int></value>
         <value><int>42</int></value>
      </data>
   </array>
</value>
```

Arrays can also contain mixtures of different types, as shown here:

```
<value>
   <array>
      <data>
         <value><boolean>1</boolean></value>
         <value><string>Chaotic collection, eh?</string></value>
         <value><int>-91</int></value>
         <value><double>42.14159265</double></value>
      </data>
   </array>
</value>
```

Creating multidimensional arrays is simple - just add an array inside of an array:

```
<value>
   <array>
      <data>

         <value>
            <array>
               <data>
                  <value><int>10</int></value>
                  <value><int>20</int></value>
                  <value><int>30</int></value>
               </data>
            </array>
         </value>

         <value>
            <array>
               <data>
                  <value><int>15</int></value>
                  <value><int>25</int></value>
                  <value><int>35</int></value>
               </data>
            </array>
         </value>

      </data>
   </array>
</value>
```

A simple struct might look like:

```
<value>
   <struct>
      <member>
         <name>givenName</name>
         <value><string>Joseph</string></value>
      </member>

      <member>
         <name>familyName</name>
         <value><string>DiNardo</string></value>
      </member>

      <member>
         <name>age</name>
         <value><int>27</int></value>
      </member>
   </struct>
</value>
```

This way you can implement almost all data types supported by any programming language.

XML-RPC - REQUEST FORMAT

XML-RPC requests are a combination of XML content and HTTP headers. The XML content uses the data typing structure to pass parameters and contains additional information identifying which procedure is being called, while the HTTP headers provide a wrapper for passing the request over the Web.

Each request contains a single XML document, whose root element is a *methodCall* element. Each *methodCall* element contains a *methodName* element and a *params* element. The *methodName* element identifies the name of the procedure to be called, while the *params* element contains a list of parameters and their values. Each *params* element includes a list of param elements which in turn contain *value* elements.

For example, to pass a request to a method called *circleArea*, which takes a *Double* parameter *fortheradius*, the XML-RPC request would look like:

```
<?xml version="1.0"?>
<methodCall>
   <methodName>circleArea</methodName>
      <params>
         <param>
            <value><double>2.41</double></value>
         </param>
      </params>
</methodCall>
```

The HTTP headers for these requests will reflect the senders and the content. The basic template looks as follows:

```
POST /target HTTP 1.0
User-Agent: Identifier
Host: host.making.request
Content-Type: text/xml
Content-Length: length of request in bytes
```

For example, if the circleArea method was available from an XML-RPC server listening at */xmlrpc*, the request might look like:

```
POST /xmlrpc HTTP 1.0
User-Agent: myXMLRPCClient/1.0
Host: 192.168.124.2
Content-Type: text/xml
```

```
Content-Length: 169
```

Assembled, the entire request would look like:

```
POST /xmlrpc HTTP 1.0
User-Agent: myXMLRPCClient/1.0
Host: 192.168.124.2
Content-Type: text/xml
Content-Length: 169
<?xml version="1.0"?>
<methodCall>
   <methodName>circleArea</methodName>
      <params>
         <param>
            <value><double>2.41</double></value>
         </param>
      </params>
</methodCall>
```

It's an ordinary HTTP request, with a carefully constructed payload.

XML-RPC - RESPONSE FORMAT

Responses are much like requests, with a few extra twists. If the response is successful - the procedure was found, executed correctly, and returned results - then the XML-RPC response will look much like a request, except that the *methodCall* element is replaced by a *methodResponse* element and there is no *methodName* element:

```
<?xml version="1.0"?>
<methodResponse>
   <params>
      <param>
         <value><double>18.24668429131</double></value>
      </param>
   </params>
</methodResponse>
```

- An XML-RPC response can only contain one parameter.
- That parameter may be an array or a struct, so it is possible to return multiple values.
- It is always required to return a value in response. A "success value" - perhaps a Boolean set to true 1.

Like requests, responses are packaged in HTTP and have HTTP headers. All XML-RPC responses use the 200 OK response code, even if a fault is contained in the message. Headers use a common structure similar to that of requests, and a typical set of headers might look like:

```
HTTP/1.1 200 OK
Date: Sat, 06 Oct 2001 23:20:04 GMT
Server: Apache.1.3.12 (Unix)
Connection: close
Content-Type: text/xml
Content-Length: 124
```

- XML-RPC only requires HTTP 1.0 support, but HTTP 1.1 is compatible.
- The Content-Type must be set to text/xml.
- The Content-Length header specifies the length of the response in bytes.

A complete response, with both headers and a response payload, would look like:

```
HTTP/1.1 200 OK
Date: Sat, 06 Oct 2001 23:20:04 GMT
```

```
Server: Apache.1.3.12 (Unix)
Connection: close
Content-Type: text/xml
Content-Length: 124

<?xml version="1.0"?>
<methodResponse>
   <params>
      <param>
         <value><double>18.24668429131</double></value>
      </param>
   </params>
</methodResponse>
```

After the response is delivered from the XML-RPC server to the XML-RPC client, the connection is closed. Follow-up requests need to be sent as separate XML-RPC connections.

XML-RPC - FAULT FORMAT

XML-RPC faults are a type of responses. If there was a problem in processing a XML-RPC request, the *methodResponse* element will contain a fault element instead of a params element. The fault element, like the *params* element, has only a single value that indicates something went wrong. A fault response might look like:

```
<?xml version="1.0"?>
<methodResponse>
   <fault>
      <value><string>No such method!</string></value>
   </fault>
</methodResponse>
```

A fault will also have an error code. XML-RPC doesn't standardize error codes at all. You'll need to check the documentation for particular packages to see how they handle faults.

A fault response could also look like:

```
<?xml version="1.0"?>
<methodResponse>
   <fault>
      <value>
         <struct>
            <member>
               <name>code</name>
               <value><int>26</int></value>
            </member>

            <member>
               <name>message</name>
               <value><string>No such method!</string></value>
            </member>

         </struct>
      </value>
   </fault>
</methodResponse>
```

XML-RPC - EXAMPLES

To demonstrate XML-RPC, we're going to create a server that uses Java to process XML-RPC messages, and we will create a Java client to call procedures on that server.

The Java side of the conversation uses the Apache XML Project's Apache XML-RPC, available at http://xml.apache.org/xmlrpc/

Put all the .jar files in appropriate path and let us create one client and one small XML-RPC server using JAVA.

XML-RPC Client

Let us write an XML-RPC client to call a function called *sum* function. This function takes two parameters and returns their sum.

```
import java.util.*;
import org.apache.xmlrpc.*;

public class JavaClient
{
   public static void main (String [] args)
   {
      try {

         XmlRpcClient server = new XmlRpcClient("http://localhost/RPC2");
         Vector params = new Vector();
         params.addElement(new Integer(17));
         params.addElement(new Integer(13));

         Object result = server.execute("sample.sum", params);

         int sum = ((Integer) result).intValue();
         System.out.println("The sum is: "+ sum);

      } catch (Exception exception) {
         System.err.println("JavaClient: " + exception);
      }
   }
}
```

Let us see what has happened in the above example client.

- The Java package org.apache.xmlrpc contains classes for XML-RPC Java clients and XML-RPC server, e.g., XmlRpcClient.
- The package java.util is necessary for the Vector class.
- The function *server.execute*. . . sends the request to the server. The procedure sum$17, 13$ is called on the server as if it were a local procedure. The return value of a procedure call is always an Object.
- Here "sample" denotes a handler that is defined in the server.
- Note that all the parameters of the procedure call are always collected in a Vector.
- The XmlRpcClient class is constructed by specifying the "web address" of the server machine followed by /RPC2.
  - localhost - means the local machine
  - You can specify an IP number instead of localhost, e.g. 194.80.215.219
  - You can specify a domain name like xyz.dyndns.org
  - You can specify a port number along with domain name as xyz.dyndns.org:8080. The default port is 80
- Note that the result of the remote procedure call is always an Object and it has to be casted to the appropriate type.
- When problems occur $noconnection, etc.$, an Exception is thrown and it has to be caught using *catch* statement.

Due to the above call, a client sends the following message to the server. Note that this is handled by *server.execute*. . . internally and you have nothing to do with it.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<methodCall>
   <methodName>sample.sum</methodName>
   <params>
      <param>
         <value><int>17</int></value>
      </param>

      <param>
         <value><int>13</int></value>
      </param>
   </params>
</methodCall>
```

XML-RPC Server

Following is the source code of XML-RPC Server written in Java. It makes use of built-in classes available in *org.apache.xmlrpc.**

```
import org.apache.xmlrpc.*;

public class JavaServer
{

   public Integer sum(int x, int y)
   {
      return new Integer(x+y);
   }

   public static void main (String [] args)
   {
      try {

         System.out.println("Attempting to start XML-RPC Server...");
         WebServer server = new WebServer(80);
         server.addHandler("sample", new JavaServer());
         server.start();
         System.out.println("Started successfully.");
         System.out.println("Accepting requests. (Halt program to stop.)");
      } catch (Exception exception)
      {
         System.err.println("JavaServer: " + exception);
      }
   }
}
```

Let us see what we have done in the above example server.

- The package org.apache.xmlrpc contains the class WebServer for a XML-RPC Server implementation.
- The procedure *sum* that is called remotely is implemented as a public method in a class.
- An instance of the same server class is then associated with a handler that is accessible by the client.
- The server is initialized by the port number $here: 80$.
- When problems occur, an Exception is thrown and has to be caught using the *catch* statement.

For the call mentioned in the given example client, the server sends the following response back to the client:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<methodResponse>
   <params>
```

```
      <param>
         <value><int>30</int></value>
      </param>
   </params>
</methodResponse>
```

Now your server is ready, so compile and run it at your prompt as follows:

```
C:\ora\xmlrpc\java>java JavaServer
Attempting to start XML-RPC Server...
Started successfully.
Accepting requests. (Halt program to stop.)
```

Now to test the functionality, give a call to this server as follows:

```
C:\ora\xmlrpc\java>java JavaClient
30
```

XML-RPC - SUMMARY

In this tutorial, you have learnt what is XML-RPC and why do we need XML-RPC. We have discussed about its data model, as well as the request and response message format to be exchanged between the client and the server. We have given one example to demonstrate how XML-RPC client and server work to exchange information.

XML-RPC is a very simple concept with a limited set of capabilities. Those limitations are in many ways the most attractive feature of XML-RPC, as they substantially reduce the difficulty of implementing the protocol and testing its interoperability.

While XML-RPC is simple, the creative application of simple tools can create sophisticated and powerful architectures. In cases where a wide variety of different systems need to communicate, XML-RPC may be the most appropriate lowest common denominator.

What's Next?

The next step is to learn WSDL and SOAP.

WSDL

WSDL is an XML-based language for describing Web services and how to access them.

WSDL describes a web service, along with the message format and protocol details for the Web service.

If you want to learn more about WSDL, please go through our WSDL tutorial.

SOAP

SOAP is a simple XML-based protocol that allows applications to exchange information over HTTP.

If you want to learn more about SOAP, please go through our SOAP tutorial.