Convert XML to JSON in PHP

Use server-side code to streamline Ajax

Senthil Nathan
Edward J Pring
John Morar

June 05, 2007
(First published January 16, 2007)

With the growing popularity of Web 2.0, a new data interchange format called JavaScript Object Notation (JSON) is emerging as a useful way to represent data in the business logic running on browsers. Learn how PHP-based server programs can convert XML-formatted enterprise application data into JSON format before sending it to browser applications.

As a followup to reader comments, the author made updates to Listing 6 and the download file of source code.

The emergence of Asynchronous JavaScript + XML (Ajax) has created a renewed enthusiasm in Web application development and is causing many architects and developers to rethink the ways in which they create Web applications. JavaScript Object Notation (JSON) is a data interchange format used to represent data in the business logic running on browsers. Many Ajax developers prefer handling data directly using JSON in the browser-side JavaScript code. As the usage of JSON increases, it will become necessary for middleware server programs to provide enterprise application data to the browsers in JSON format rather than in XML format. This means that developers need to convert existing server-side enterprise data encoded in XML to JSON before sending it to the browser. This article shows you how to use PHP-based server programs to convert XML-formatted application data to JSON format before you send it to the browser application.

XML basics

XML is a standard for defining markups. XML-based markup is used to describe data that is represented through tags that need not be predefined. XML is highly extensible because you can invent new tags as needed. Listing 1 shows an example of a data structure represented in XML.

Listing 1. A simple example of XML data

```
<?xml version="1.0" encoding="UTF-8"?>
<contacts>
    <contact id="1">
        <name>John Doe</name>
        <phone>123-456-7890</phone>
        <address>
            <street>123 JFKStreet</street>
            <city>Any Town</city>
            <state>Any State</state>
            <zipCode>12345</zipCode>
        </address>
    </contact>
</contacts>
```

The first line is the XML declaration that specifies the XML version and character encoding used. A root element `<contacts>` follows, which encloses several child elements. The nested structure of the child elements combines to define the data for a contact. Notice that the `<address>` element includes child elements that form a subtree structure under the `<contact>` element. XML also allows the start tags to have attributes that provide additional information about the elements. The `<contact>` element has an attribute that assigns an `id` attribute to that element. The XML document is completed with a close tag, `</contacts>`, for the root element.

JSON basics

JSON is not a markup language like XML. Instead, it is a text-based data interchange format. It is the data syntax of JavaScript objects and arrays. In short, JSON allows you to represent data simply. For example, you enclose objects within curly braces ({}), and you enclose arrays within square brackets ([]). A piece of JavaScript code can readily consume JSON-encoded data without going through any special parsing or additional data transformation. Listing 2 shows a data structure represented in JSON format.

Listing 2. A simple example of JSON data

```
{
   "contacts" : {
      "contact" : {
         "@attributes" : {
            "id" : "1"
         },
         "name" : "John Doe",
         "phone" : "123-456-7890",
         "address" : {
            "street" : "123 JFK Street",
            "city" : "Any Town",
            "state" : "Any State",
            "zipCode" : "12345"
         }
      }
   }
}
```

You can see that every piece of data shown in the XML example in Listing 1 is also present in the JSON example in Listing 2. However, the difference is how JSON encodes the data using JavaScript object and array data types. The data structure starts with an object that includes a property named `"contacts"`, which itself is an object. This object has one property named

`"contact"`, which is also an object. This object includes several properties that constitute the details for a contact. Notice that the `"contact"` object includes another (nested) object named `"address"`, which describes the details of an address. As with XML, JSON-formatted data is self-describing, so both humans and machines can read it easily.

Browser-side data processing

Now let's look at how the data is processed in the browser-side code. When the server sends XML data to the browser, that XML data is processed using Document Object Model (DOM) APIs. However, JavaScript developers must learn all the intricacies of XML processing and write a lot of additional code to parse the XML data before they can do anything with that data. With the advent of JSON, however, the server-side code can package and ship JSON-encoded application data in response to the browser requests. If the server-side code sends the JSON-formatted data to the browser, then JavaScript developers need not write any extra and complex logic to parse the XML. In addition, the data received in JSON format will be readily treated as a JavaScript native data structure in the browser-side code. The code snippet in Listing 3 shows that no extra work is required to process JSON-formatted data.

Listing 3. A code snippet for processing JSON-formatted data received from the server

```
var result = httpRequest.responseText;
jsonResponse = eval('(' + result + ')');
```

In the browser-side code snippet in Listing 3, `result` is the JSON-formatted string received from the server. All it takes is a single line of code to turn that string data into a native JavaScript data type by evaluating the JSON-formatted string using the `eval` statement. You can see that it is much easier to deal with JSON data on the browser side. The use of JSON brings simplicity and value to the browser-side code.

The `eval` statement in Listing 3 will execute whatever code the server returns, so there is a security risk here. The risk is limited, since browser security policy restricts HTTP requests in JavaScript to the same server the code was originally loaded from. Still, in some situations, it might be prudent to verify that data received from a server matches expectations, perhaps with a regular expression, before passing it to the `eval` statement.

XML-to-JSON conversion

More and more applications need to convert XML data into JSON. Several Web-based services are already popping up to do such conversions. The IBM T.J. Watson Research Center has developed a particular approach that uses PHP to do the conversion. This approach accepts XML string data as input and converts that into JSON-formatted data output. This PHP-based solution provides several benefits:

- It can be run in a standalone mode, executed from the command line.
- It can be included in an existing server-side code artifact.
- It can be easily hosted as a Web service on the Web.

The XML-to-JSON conversion requires a couple of core PHP features:

- `SimpleXMLElement`
- `Services_JSON` from http://pear.php.net (see Related topics for details)

`SimpleXMLElement` is supported in PHP version 5 and above. It is an object with properties containing the data held in the XML document. `Services_JSON` is a PHP open source proposal approved near the end of 2005. It provides JSON-specific encoder (PHP data to JSON) and decoder (JSON to PHP data) functions. This open source PHP class library is now made available through the PEAR Web site (see Related topics).

By using just these two core features of PHP, it is possible to convert any arbitrary XML data into JSON. First, you need to convert the XML content into a suitable PHP data type using `SimpleXMLElement`. Then, you feed the PHP data to the `Services_JSON` encoder, which in turn produces the final JSON-formatted output.

Understand the PHP code

This `xml2json` implementation has three parts:

- xml2json.php - A PHP class with two static functions
- xml2json_test.php - A test driver to exercise the `xml2json` conversion function
- test1.xml, test2.xml, test3.xml, test4.xml - XML files with varying complexity

For simplicity, this article doesn't show detailed comments in the code. However, the code in the attached source files does include full comments. Refer to the attached source files for complete details about the program logic (see Download).

Listing 4 defines some useful constants. The first line of code imports the `Services_JSON` implementation.

Listing 4. Defining constants in xml2json.php

```
require_once 'json/JSON.php';
// Internal program-specific Debug option.
define ("DEBUG", false);
// Maximum Recursion Depth that we can allow.
define ("MAX_RECURSION_DEPTH_ALLOWED", 25);
// An empty string
define ("EMPTY_STR", "");
// SimpleXMLElement object property name for attributes
define ("SIMPLE_XML_ELEMENT_OBJECT_PROPERTY_FOR_ATTRIBUTES", "@attributes");
// SimpleXMLElement object name.
define ("SIMPLE_XML_ELEMENT_PHP_CLASS", "SimpleXMLElement");
```

The code snippet shown in Listing 5 is the entry function into the `xml2json` converter. It takes XML data as input, and it converts the XML string into a `SimpleXMLElement` object, which is sent as input to another (recursive) function in this class. This function then converts the XML elements into a PHP associative array. That array is then passed as an input to the `Services_JSON` encoder, which gives you the JSON-formatted output.

Listing 5. Using Services_JSON in xml2json.php

```
public static function transformXmlStringToJson($xmlStringContents) {
```

```
    $simpleXmlElementObject = simplexml_load_string($xmlStringContents);

    if ($simpleXmlElementObject == null) {
        return(EMPTY_STR);
    }

    $jsonOutput = EMPTY_STR;

    // Let us convert the XML structure into PHP array structure.
    $array1 = xml2json::convertSimpleXmlElementObjectIntoArray($simpleXmlElementObject);

    if (($array1 != null) && (sizeof($array1) > 0)) {
        // Create a new instance of Services_JSON
        $json = new Services_JSON();
        // Let us now convert it to JSON formatted data.
        $jsonOutput = $json->encode($array1);
    } // End of if (($array1 != null) && (sizeof($array1) > 0))

    return($jsonOutput);
} // End of function transformXmlStringToJson
```

The lengthy code snippet shown in Listing 6 uses a recursion technique inspired by the PHP open source community (see Related topics). It accepts a `SimpleXMLElement` object as input and conducts a recursive tree walk through the nested XML tree. It stores all the visited XML elements in a PHP associative array. You can adjust the recursion depth limit by changing a constant defined in Listing 4.

Listing 6. Conversion logic in xml2json.php

```
public static function convertSimpleXmlElementObjectIntoArray($simpleXmlElementObject,
&$recursionDepth=0) {
  // Keep an eye on how deeply we are involved in recursion.

  if ($recursionDepth > MAX_RECURSION_DEPTH_ALLOWED) {
    // Fatal error. Exit now.
    return(null);
  }

  if ($recursionDepth == 0) {
    if (get_class($simpleXmlElementObject) != SIMPLE_XML_ELEMENT_PHP_CLASS) {
      // If the external caller doesn't call this function initially
      // with a SimpleXMLElement object, return now.
      return(null);
    } else {
      // Store the original SimpleXmlElementObject sent by the caller.
      // We will need it at the very end when we return from here for good.
      $callerProvidedSimpleXmlElementObject = $simpleXmlElementObject;
    }
  } // End of if ($recursionDepth == 0) {

  if (get_class($simpleXmlElementObject) == SIMPLE_XML_ELEMENT_PHP_CLASS) {
    // Get a copy of the simpleXmlElementObject
    $copyOfsimpleXmlElementObject = $simpleXmlElementObject;
    // Get the object variables in the SimpleXmlElement object for us to iterate.
    $simpleXmlElementObject = get_object_vars($simpleXmlElementObject);
  }

  // It needs to be an array of object variables.
  if (is_array($simpleXmlElementObject)) {
    // Initialize the result array.
    $resultArray = array();
    // Is the input array size 0? Then, we reached the rare CDATA text if any.
    if (count($simpleXmlElementObject) <= 0) {
      // Let us return the lonely CDATA. It could even be
```

```
      // an empty element or just filled with whitespaces.
      return (trim(strval($copyOfsimpleXmlElementObject)));
    }

    // Let us walk through the child elements now.
    foreach($simpleXmlElementObject as $key=>$value) {
      // When this block of code is commented, XML attributes will be
      // added to the result array.
      // Uncomment the following block of code if XML attributes are
      // NOT required to be returned as part of the result array.
      /*
if((is_string($key)) && ($key == SIMPLE_XML_ELEMENT_OBJECT_PROPERTY_FOR_ATTRIBUTES)) {
        continue;
      }
      */

      // Let us recursively process the current element we just visited.
      // Increase the recursion depth by one.
      $recursionDepth++;
      $resultArray[$key] =
        xml2json::convertSimpleXmlElementObjectIntoArray($value, $recursionDepth);

      // Decrease the recursion depth by one.
      $recursionDepth--;
    } // End of foreach($simpleXmlElementObject as $key=>$value) {

    if ($recursionDepth == 0) {
      // That is it. We are heading to the exit now.
      // Set the XML root element name as the root [top-level] key of
      // the associative array that we are going to return to the caller of this
      // recursive function.
      $tempArray = $resultArray;
      $resultArray = array();
      $resultArray[$callerProvidedSimpleXmlElementObject->getName()] = $tempArray;
    }

    return ($resultArray);
  } else {
    // We are now looking at either the XML attribute text or
    // the text between the XML tags.
    return (trim(strval($simpleXmlElementObject)));
  } // End of else
} // End of function convertSimpleXmlElementObjectIntoArray.
```

At the end of a successful XML tree walk, this function will convert and store all the XML elements (the root element and all the child elements) in a PHP associative array. For complex XML documents, the resulting PHP array will be equally complex. Once that PHP array is fully constructed, then the `Services_JSON` encoder can easily convert it to JSON-formatted data. To understand this recursive logic, check out the documented source file.

Implement a test driver for xml2json

The code snippet shown in Listing 7 is a test driver that exercises the `xml2json` converter logic.

Listing 7. xml2json_test.php

```
<?php
    require_once("xml2json.php");

    // Filename from where XML contents are to be read.
    $testXmlFile = "";

    // Read the filename from the command line.
```

```
    if ($argc <= 1) {
        print("Please provide the XML filename as a command-line argument:\n");
        print("\tphp -f xml2json_test.php test1.xml\n");
        return;
    } else {
        $testXmlFile = $argv[1];
    }

    //Read the XML contents from the input file.
    file_exists($testXmlFile) or die('Could not find file ' . $testXmlFile);
    $xmlStringContents = file_get_contents($testXmlFile);

    $jsonContents = "";
    // Convert it to JSON now.
    // xml2json simply takes a String containing XML contents as input.
    $jsonContents = xml2json::transformXmlStringToJson($xmlStringContents);

    echo("JSON formatted output generated by xml2json:\n\n");
    echo($jsonContents);
?>
```

You can run the program from the command line with an XML filename as a command-line argument:

```
php -f xml2json_test.php test2.xml
```

When executed from a command line, the program reads the XML content from a file into a string variable. Then it calls a static function in the `xml2json` class to get the JSON-formatted result. In addition to running the program from the command line, you can modify the logic in this source file to expose the `xml2json` converter as a remotely callable Web service using Simple Object Access Protocol (SOAP) or Representational State Transfer (REST) access protocols. If needed, you can perform that easily in PHP with minimal work.

Listing 8 shows one of the four test XML files provided with this article to test the `xml2json` implementation. The degree of complexity varies from one test file to the other. You can pass one of these files as a command-line argument to the test driver xml2json_test.php.

Listing 8. Test the xml2json implementation with test2.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<books>
    <book id="1">
        <title>Code Generation in Action</title>
        <author><first>Jack</first><last>Herrington</last></author>
        <publisher>Manning</publisher>
    </book>

    <book id="2">
        <title>PHP Hacks</title>
        <author><first>Jack</first><last>Herrington</last></author>
        <publisher>O'Reilly</publisher>
    </book>

    <book id="3">
        <title>Podcasting Hacks</title>
        <author><first>Jack</first><last>Herrington</last></author>
        <publisher>O'Reilly</publisher>
    </book>
</books>
```

The code snippet shown in Listing 9 is the JSON-formatted result when you use test2.xml as the command-line argument to the test driver xml2json_test.php.

Listing 9. JSON-formatted result for test2.xml

```
{
   "books" : {
      "book" : [ {
         "@attributes" : {
            "id" : "1"
         },
         "title" : "Code Generation in Action",
         "author" : {
            "first" : "Jack", "last" : "Herrington"
         },
         "publisher" : "Manning"
      }, {
         "@attributes" : {
            "id" : "2"
         },
         "title" : "PHP Hacks", "author" : {
            "first" : "Jack", "last" : "Herrington"
         },
         "publisher" : "O'Reilly"
      }, {
         "@attributes" : {
            "id" : "3"
         },
         "title" : "Podcasting Hacks", "author" : {
            "first" : "Jack", "last" : "Herrington"
         },
         "publisher" : "O'Reilly"
      }
   ]}
}
```

Notice that the XML attribute `id` for the `<book>` element is stored in JSON data as the `"@attributes"` object property, and the `<book>` element is stored in JSON data as an array of objects. This JSON output is readily consumable inside JavaScript code using the `eval` statement.

Conclusion

JSON is just beginning to gain momentum among Web developers. Its success, seen mainly with JavaScript developers, is due to its elegance and simplicity. JSON can be a worthy alternative to XML in certain situations. This article sums up the need for XML-to-JSON conversion at the middleware server layer. It further stresses the rationale behind leveraging existing XML-encoded enterprise data as JSON-formatted data so that browser-side programs can consume it easily. It provides PHP code that can do the XML-to-JSON conversion. (See Download.)

You can use the source code provided in this article in multiple ways -- as a standalone tool, as a shared class library to be used in an existing server-side program, or as a SOAP/REST Web service function to participate in an enterprise Service-Oriented Architecture (SOA).

Downloadable resources

Description	Name	Size
PHP code for xml2json	x-xml2json_php.zip	14KB

Related topics

- Ajax and REST, Part 1 (Bill Higgins, developerWorks, October 2006): Learn about the advantages of the Ajax/REST architectural style for immersive Web applications.
- Developing PHP the Ajax way, Part 1: Getting started (Sean Kelly, developerWorks, May 2006): Get started with PHP and Ajax.
- IBM developerWorks Ajax resource center: Read more articles and tutorials on Ajax.
- JSON Web site: Read all about JavaScript Object Notation (JSON).
- PHP open source community: Check out all things PHP.
- PHP Manual: Dig into an extremely useful PHP resource offered in 23 languages.
- PHP Extension and Application Repository (PEAR) Web site: Download the `Services_JSON` library.
- XML technical library: See the developerWorks XML Zone for a wide range of technical articles and tips, tutorials, standards, and IBM Redbooks.
- IBM trial software: Build your next development project with trial software available for download directly from developerWorks.